POLITECNICO DI TORINO

Master Degree in Electronic Engineering

Master Degree Thesis

Serial Bit Accelerator with Sparsity Managment

Supervisors

Prof. MAURIZIO MARTINA

Prof. GUIDO MASERA

Candidate

Francesco DILEVRANO

OCTOBER 2023

ACKNOWLEDGMENTS

Questa è l'uinica parte di questa tesi ad essere scritta in italiano, dire grazie a tutti in un altra lingua non avrebbe la stessa valenza a mio avviso. Parto con il ringraziare il professor Maurizio Martina e il suo assistente Maurizio Capra per avermi dato questa possibilità ed avermi supportato durante il lavoro di tesi. Ora il mio più grande ringraziamento va ai miei genitori, Ferdinando e Claudia, per avermi dato la possibilità di raggiungere questo traguardo. Un ringraziamento speciale va anche a mio fratello Alessandro. Vorrei ringraziare i miei nonni tutti purtroppo in questo momento posso abbracciare solo mio nonno Gregorio ma mando un abbraccio ai restanti che spero siano fieri di me ovunque siano. Ringrazio Imma, Salvatore, e gli zii tutti che ci sono sempre stati durante questo percorso. A questo punto il mio pensiero va a mio zio Giacinto che purtroppo come i nonni non c'è più e mi manca tanto, ti ricorderò sempre con il sorriso e avrei voluto vederti sorridere anche ora. Ringrazio i miei cugini tutti, anche loro come i miei zii ci sono sempre stati e sempre ci saranno. Quelli più vicini a me in questi anni sono stati però senza dubbio i miei coinquilini, vecchi e attuali, con i quali ho condiviso momenti belli e brutti di questa avventura, grazie di avermi sopportato. Una fortuna che ho avuto in anni è sicuramente quella di aver avuto tanti amici a supportarmi e spronarmi, da quelli storici del GIUKD a quelli conosciuti al poli che hanno incrociato il loro percorso al mio, e infine i miei amati zanzi, a tutti voi un grazie con tutto il cuore. Credo di essere stato abbastanza fortunato fino a questo punto della mia vita visto che ho incontrato davvero tante persone meravigliose con le quali ho stretto un bel rapporto e di consequenza a tutti voi chiedo scusa per non citarvi nel dettaglio ma vi ringrazio tanto. Per essere un attimo meno seri a questo punto vorrei ringraziare anche PPP che negli ultimi tempi mi hanno tenuto compagnia nei momenti tristi rendendoli allegri.

Table of Contents

Chapter 1

General introduction on Neural Networks

1.1 Summary

This thesis work is organized into five chapters that are briefly described in the following lines.

General introduction on Neural Networks

This chapter introduces the background of the NNs, starting from the neuron structure, and shows how this structure can be replicated. This allows also us to understand the cause of their great exploitation in recent years. Will also explain some critical points such as the train problem and also some famous models will be described.

Based work and methodologies

This chapter will introduce the state of the art of NN that will include the different kinds of structure, the convolution idea, and the data flow. Then will be explained which is the starting structure that is the base to develop an engine able to manage sparsity.

SMAC Engine with sparsity managment

Here will be described all the changes applied to the starting structure starting from the choice of a compression method to exploit sparsity management and achieve better performances.

Verification

Here are shown all the verification done to verify that the new engine works correctly, Also will be shown the synthesis and the performance's results.

Conclusion and future work

A summary will be provided united to some possible development that can be done in the future.

1.2 Principles

Technology evolution has brought to the realization of machines that can do tasks independently and to adapt themselves learning. Some examples are autonomous driving, object detection, speech and image recognition, and many other once "human-only" related tasks. All these tasks are central in human life nowadays. In parallel with the need for machines able to manage these tasks Machine Learning (ML), one of the fields of Artificial Intelligence (AI), has grown up. An even narrower area of ML, namely Deep Learning (DL), is attracting many researchers. This is mainly due to the large application of DL, following there are some examples:

- computer visions
- business and finance
- healthcare
- robotics
- smart energy management

DL is based on the development and adoption of Deep Neural Networks (DNNs). DNNs are structures that want to be similar as much as possible to the human brain. They emulate some of his aspects, in a human brain neuron is considered the base computational element, a DNN consists of several layers (the higher the number the deeper the network), each containing some neurons contributing to the computation of the output result. In training the DNN tries to properly tune the weights and biases parameters based on the so-called hyperparameters, like the learning rate, the number of hidden layers, the number of neurons, and so on. The weights and biases parameters are the ones that will ultimately be used during inference to perform the specific task the DNN has been developed for. Among the NNs are the Convolutional Neural Networks (CNNs) that had a great exploit and now are the most popular in terms of applications. Central themes in NNs

are the training time, which can be in the order of hours or days depending on the accuracy, and the hardware stress. The first problem is the need for a hardware platform capable to provide ready-to-use models within a reasonable time. The second brings to the necessity to have ad-hoc applications with a trade-off between accuracy and complexity.

1.3 Neuron and Math

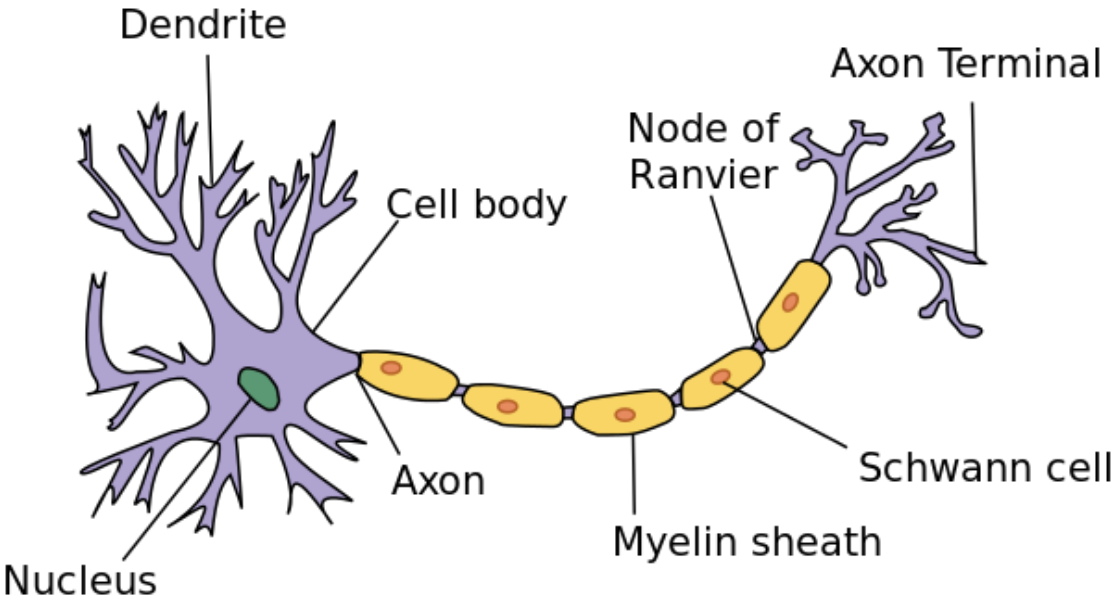

Figure 1.1: Neuron structure [1]

As mentioned before NNs are human brain-based [2] structures, so algorithms try to emulate the human brain's ability to learn. Among all the ML solutions, such as linear regression, NNs can deal with multiple features so are more suitable for learning complex non-linear hypotheses. The starting point is that the brain is composed of a neuron network. The neuron shown in figure 1.1 can be seen as something that receives data through the dendrites, operate some internal computations, and provide an output through the axon. The NNs are so based, there are the input features that are elaborated from the hidden layers and the output layers that are responsible for generating the prediction. Neurons can be seen as nodes of an oriented graph, if the graph is acyclic we have a feedforward NN (figure 1.2), on the contrary, if it is cyclic we have a recurrent NN (figure 1.3). Having deeper NN is better than having simply big networks due to that the network itself can extract more and more complex features during inference. This

allows the DNNs with multiple hidden layers to be able to have at earlier layers learn lower-level simple features, the advantage is that later deeper layers put together the simpler things. While the earlier layers are computing relatively simple functions of the input, by the time one gets deep into the network it can do surprisingly complex things. Dealing with DNNs means dealing with hyperparameters. These parameters are given to the learning algorithm and will affect ultimate parameters $W^{[l]}$ and $b^{[l]}$. The following are some of the principal hyperaparameters:

- The learning rate α, because it will determine how our parameters evolve.
- The number of iterations of gradient descent.
- The number of hidden layers L.
- The number of hidden units or neurons for each layer.
- The chosen activation function (ReLU, tanh, sigmoid, etc.).
- The momentum term.
- The mini-batch size.
- Various forms of regularization parameters.

Activations in NN are generated from layer to layer, the first layer generates activations for the second, the second generates activations for the third, and so on. This is known as forward propagation. About the way that a machine learns is possible to distingue between:

- supervised learning
- unsupervised learning

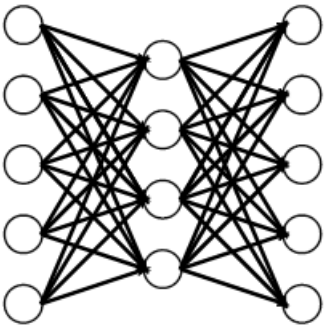

Figure 1.2: Feedforward [3]

There are different kind of layers:

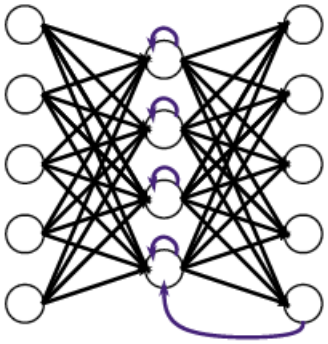

Figure 1.3: Recurrent [3]

- Fully Connected
- Convolutional
- Polling

The computation required by a single neuron, in the case the neuron n^th, is:

$$a_n = f\left(\sum_i W_{in}x_i + b\right)$$

a_n is the output for the next layer generated with the non-linear function that uses x, w, and b which are the activations, the weights, and the bias. This is done by every neuron but the inputs are the outputs of the previous neuron. Some examples of these non-linear functions are:

- Sigmoid
- hyperbolic tangent
- ReLU

1.4 How NN Learns

The way a NN learns how to perform a task in the best way is to use the gradient descent algorithm. How well the algorithm is doing on a single training can be measured by the loss or error function. Expression of this function is:

$$L(\widehat{y}, y) = -(y \log \widehat{y} + (1 - y) \log(1 - \widehat{y}))$$

Here $\widehat{y}$ is the prediction provided by the NN in the forward propagation and y is the "ground truth" that comes from the labeled data. The cost function instead

is useful o know if and how the network is working well compared to an entire training set. Expression of the cost function is:

$$J(W^{[1]}, b^{[1]}, ..., W^{[m]}, b^{[m]}) = \frac{1}{m} \sum_{i}^{m} L(\widehat{y}^{(i)}, y^{(i)})$$

In this cost function we have a training set composed of m examples and the NN has L layers, while $\widehat{y} = a^{[L]}$. Through gradient descent tries to find W[l] and b[l], with l = 1, . . . , L that minimizes the cost function J to have the predictions closer possible to the ground truth. Gradient descent is an iterative algorithm that starts from a starting point, that can be randomly picked, and for every step, it tries to move towards the steepest downhill direction until eventually converging to a global optimum minimizing J.

$$W^{[l]} := W^{[l]} - \alpha \frac{\partial J}{\partial W^{[l]}}$$

$$b^{[l]} := b^{[l]} - \alpha \frac{\partial J}{\partial b^{[l]}}$$

$$l = 1, ..., L$$

Here α is the learning rate controlling the size of the step taken by each gradient descent iteration and one of the hyperparameters that need to be properly tuned to obtain optimal results. The derivative terms represent the update applied to the parameters. After forward propagation is performed the backpropagation algorithm is introduced to compute the gradients. This algorithm uses the chain rule derived from calculus and passes the values from the output backward in a way similar to the forward propagation. Use gradient descent to train a neural network to solve the symmetry-breaking problem, it randomly initializes the weights rather than initializes everything to 0. It is also possible to show that initializing weights with zeros leads to hidden units being symmetrical, meaning they compute the same function for every iteration, which is not helpful. There are different ways that a machine can be trained

Supervised learning

In this case, a proper set of data is provided to the machine, and parameters are tuned based on the difference between the expected output and the actual, so there is feedback. The exploit of this type of learning is due to the coming of the big-data era that makes available an impressive amount of datasets. The main tasks of this learning are classification and regression. For Classification main applications are:

- image classification

- diagnostics

for regression instead, main applications are:

- Market forecasting
- Weather forecasting

Unsupervised learning

In this case, non-labeled data are available and are searching for common patterns in them. Opposite of supervised learning there isn't feedback. The main tasks of this learning are Clustering and Dimensionality reduction. For clustering main applications are:

- Recommender system
- Customer segmentation

For dimensionality reduction main applications are:

- Meaningful compression
- Structure discovery

Reinforcement learning

Similar to unsupervised learning doesn't need labeled data but aims to make a decision based on the actual environment. The decision is evaluated by an interpreter in terms of reward and then communicate back. Scope at this point is trying to maximize reward so future decisions are based on this. This peculiarity makes it also similar to supervised learning that has feedback. For this type of learning main applications are:

- Real-time decisions
- Robot navigation
- Skill acquisition
- Game AI

1.5 Layer

There will be explained the different kinds of layers, before doing this some keywords useful to understand how some layers work have to be explained.

- activations: are the values that layers of the NN are passing on to the subsequent layers
- weight: parameters associated with both hidden and output layers necessary for the computation every neuron has to perform
- feature map: 2D structure in which neurons are organized
- receptive fields: sub-portions of the previous layer used from the actual layer
- kernel size: The size of the weights matrix, this size is equal to the receptive field size
- stride: the distance between two adjacent receptive fields

Full Connected

In this layer, every neuron is connected to all the neurons of the previous layer. Can be seen as a matrix-vector product. Sometimes in FC layers, not all the inputs must be processed so there is a time saving. The downside of this layer is that the complexity and number of parameters make them unsuitable for some applications like detection and recognition.

Convolutional

The base idea of this layer is the local receptive fields and shared weights. Shared weights allow all neurons of the layer to have the same weight matrix, extracting a particular feature from the previous layer. To detect multiple features, a Conv layer is composed of several feature maps called channels and several kernels.

Polling

For a receptive field only a single value is based on a specific characteristic that could be for example the average or the maximum value. Stride is equal to the kernel size to have non-overlapping windows. The results of this constriction are the reduction of the number of activations and the reduction of the feature map dimensions so neurons have fewer operations to do. In figure 1.4 we have on the left an example of polling with maximum value and on the right, an example of polling with average A downside of this layer is that the downsampling makes it less sensitive to small local translaction.

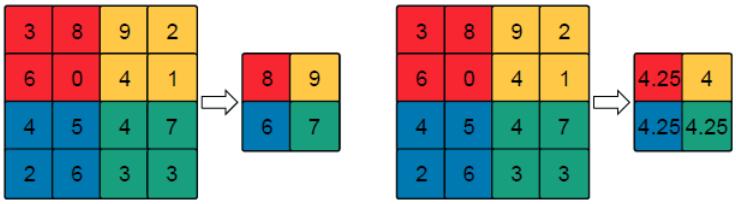

Figure 1.4: Maximum value and average polling [3]

1.6 Non-linear function

A non-linear function is always necessary, without it no interesting function would be computed even going deeply into the NN and thus invalidating the training process. Instead with a linear function despite a non-linear, the NN will simply generate an output that is a linear function of the input and this will make hidden layers useless. Anticipating that depending on the application one function can suit better than another and it can also be different from layer to layer, among these solutions, ReLU has been a great exploit in NN in recent years. This is due mainly to the simple implementation and also to the faster learning rate of this function.

Rectified Linear Unit

This function forces the output to be greater than 0, the main consequence of this is a very low computation cost. The mathematical expression is shown in figure 1.5. A problem occurs when x is negative so the outcome is 0, in this case, the neurons are not trained. To overcome this problem the two variants, Leaky-ReLU and Exponential Linear Unit (ELU), of this function have been introduced. These two variations are also balanced towards zero and so speed up the train.

$$y = \begin{cases} 0 & if x < 0 \\ x & otherwise \end{cases}$$

Sigmoid

This function has a computation cost much higher than the ReLU function. Sigmoid normalizes the output in the range (0, 1), a mathematical expression is shown in figure 1.6

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

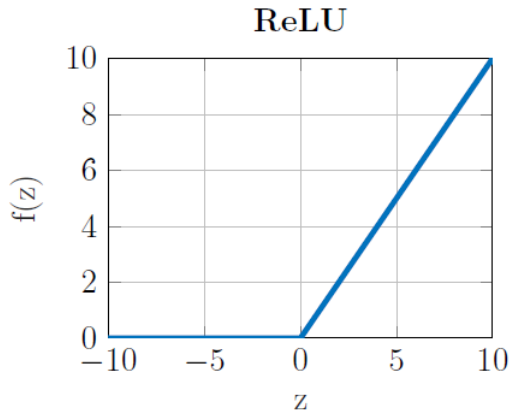

Figure 1.5: ReLU function

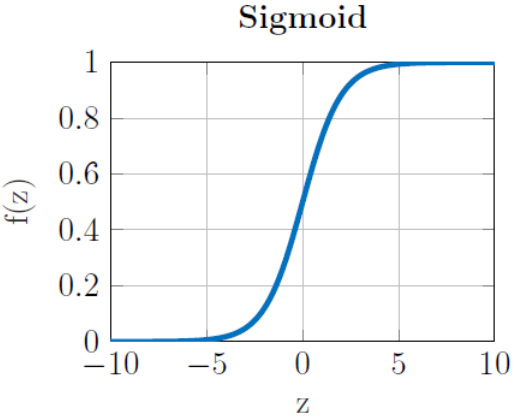

Figure 1.6: Sigmoid function

Hyperbolic Tangent

The main difference is that this function can assume negative values since it normalized the output in the range (1, -1). The mathematical expression is shown in figure 1.7. The negative value overcomes the main problems of the ReLU but as the Sigmoid has a higher computational cost and is towards zero or far from zero the gradient of this function becomes very small.

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

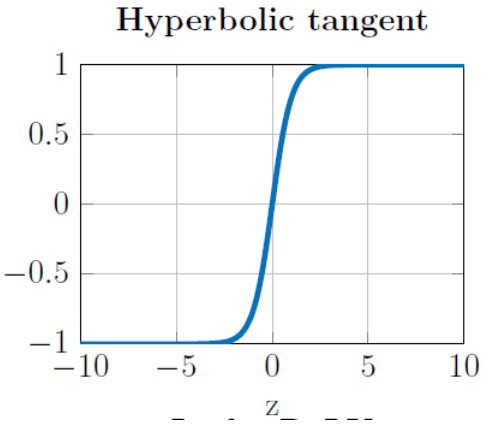

Figure 1.7: Hyperbolic tangent function

Softmax

This function performs a prediction over N multiple entries. Input is a vector of N entries: each entry is normalized in the range (0,1), and the total sum of all vector elements is equal to 1. Math expression is shown below.

$$y_i = \frac{e^{x_i}}{\sum_{j=0}^{N-1} e^{x_j}} for i = 0,1,....,N-1$$

1.7 DNN Models

Now some of the fundamental CNNs developed in recent years will be described. Have been proven that these CNNs effectively work but in many cases are also the starting point to develop new ones.

LeNet

This CNN [4] is one of the first that was designed for convolutional structures and also one of the first trained with backpropagation whose aim is to recognize handwritten digits. An implementation of this network is LeNet-5 which is composed of five layers that act in this way

- first and second layers are convolutional with a 5x5 kernels
- third, fourth and fifth layers are FC layers, in this case, 2x2 average polling layers

Is characterized by the hyperbolic tangent as the main non-linear function and also softmax function for the outputs. This network handled around 60 000 parameters

whereas today it is quite common to see networks using 10 to 100 million parameters. This CNN has been the first to be successfully applied for commercial use in ATMs to recognize the handwritten digits of check deposits. This has been trained using 28×28 images and overall performs around 340 000 MACs.

AlexNet

This network is famous for being the first in winning the ImageNet [5] challenge and adopting a ReLU non-linearity reducing so the training phase that uses 256x256 pixels high-resolution images. This network follows the idea of LeNet and is composed of eight layers that act in this way

- from first to fifth are Convolutional Layers
- from sixth to eighth are FC Layers

This net use 61 million parameters and operates 724 million MACs, these numbers are much larger than LeNet. But both structures are very similar and also the basic building blocks are the same.

VGG16

The starting point of this network is the structure introduced in LeNet and adopted in AlexNet, so there are 16 layers [6] that act in this way:

- from one to thirteen are Convolutional Layers with 3x3 kernels and a unitary stride
- from fourteen to sixteen are 2x2 polling layers with stride 2

AlexNet won the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) in 2012 while VGG19 (another version of the VGG16 but with 19 layers) won it in 2014

GoogLeNet

This network adopts an inception module [7] that extracts features at various scales and concatenates them at the output, forwarding them to the next layer. An example with four scales computed in parallel and then merged in one output is shown in figure 1.8. The deeper we go into the models, the higher the accuracy but beyond the deeper level the vanishing gradient becomes relevant. Values of the gradients in backpropagation are in the range of 0,1 or minus 1,1, the magnitude of the gradients becomes smaller with the depth of the network. Small gradients in the first layer can make bad training problems, this can be mitigated by increasing

the magnitude of the gradients and is done by adding two classifiers that take the activations at the network's earlier stages. The first version of this network employed 7 million parameters and performed 1.43 billion MACs on 224×224 input RGB images.

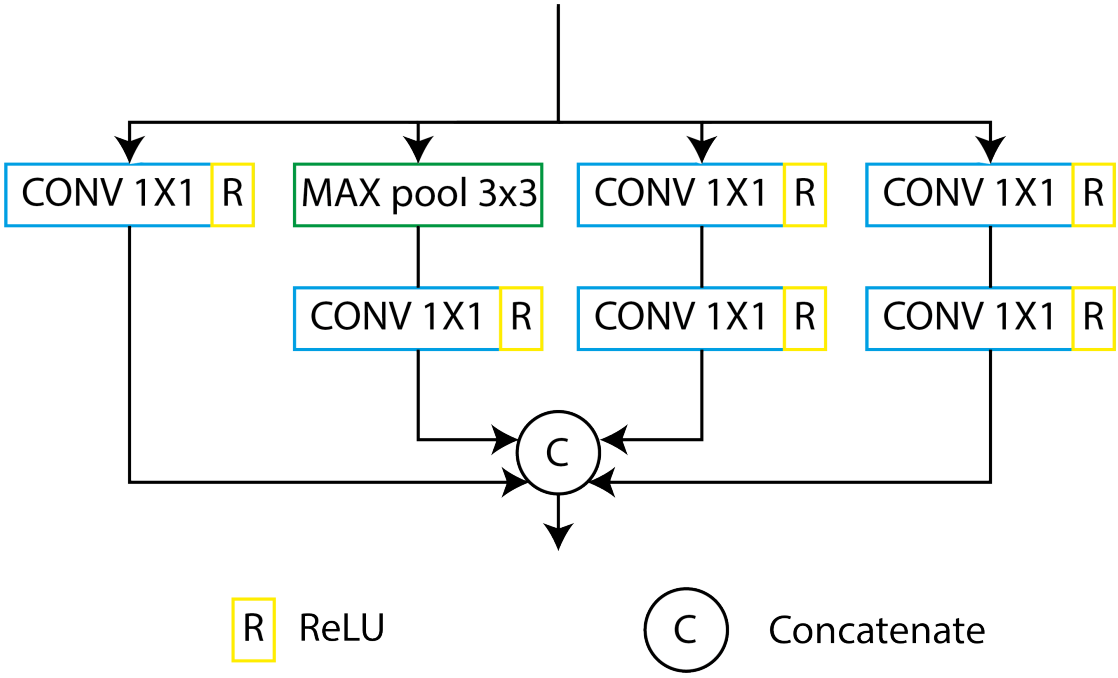

Figure 1.8: Inception Module in GoogLeNet [3] [7]

ResNet

[8]To make marginal the problem of vanishing gradient this net adopts a connection that runs in parallel to the Conv layer called skip connection, this allows it exceed human-level accuracy in the ImageNet competition. Is the first to adopt the batch normalization layer. Skip connection modules are shown in figure 1.9, for both solutions, there are three convolutions performed in series, and the results are summed to a convolution 1x1 done in parallel, on the right is simply summed to the identified function. This net has several layers that go from 34 to 152. Taking a ResNet50 as a reference for 224 × 224 input RGB images, the number of used parameters is around 25.5 million and the number of performed MACs is 3.9 billion.

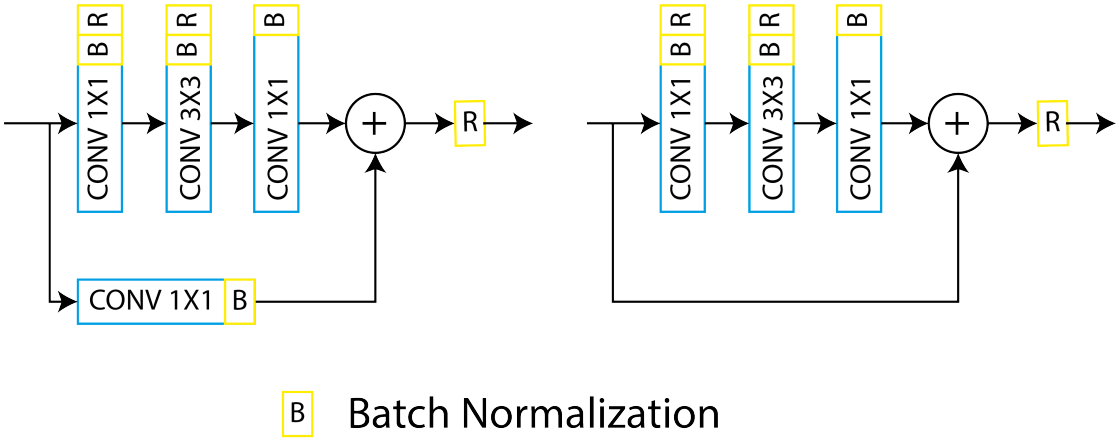

Figure 1.9: Skip connection module in ResNet [3] [8]

DenseNet

[9]Following the example of ResNet with skip connections this net adopts a more regular connection pattern based on Dense Block. In a Dense Block, each layer input is a concatenation of the activations of all the preceding layers. A DenseNet comprises Dense Blocks of different depths. For dimensionality reduction, two concatenations are interleaved by the sequence of convolution and a polling layer. The dense block is shown in figure 1.10, here the series of two convolutions are concatenated.

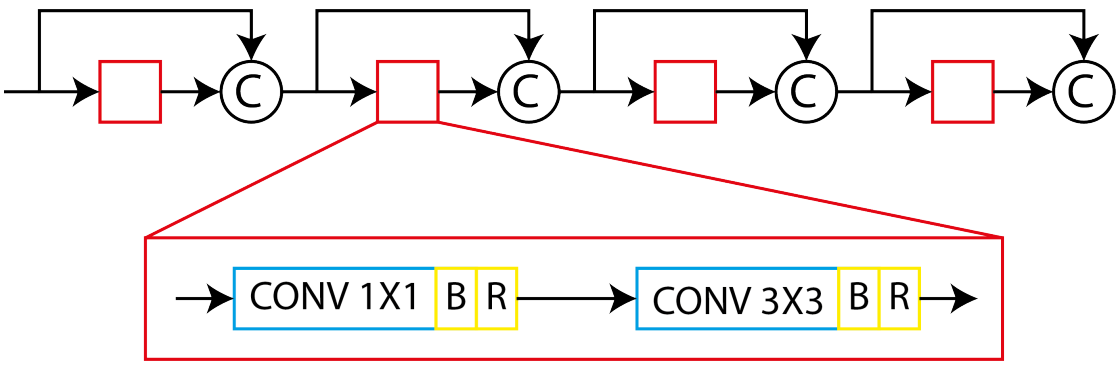

Figure 1.10: DenseNet Block [3][9]

SENet

[10]Squeeze-and-Excitation Networks model the relationship between the different channels present in the feature maps reworking classic layers. Figure 1.11 is shown

how the residual model is modified in SENet. Two skip connections are present, one in parallel to the whole block and one in parallel to the two FC and the polling layers. SENet-154 is the NN winner of ILSVRC-2017.

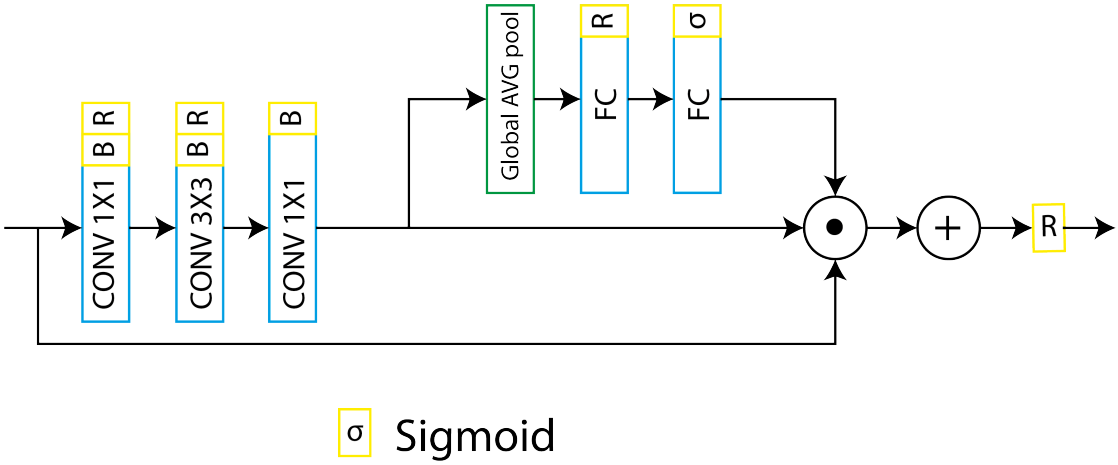

Figure 1.11: SENet modified residual model [3][10]

Capsule Network

[11]The main idea of this net is to use capsules instead neurons, this is done to reduce some problems of the CNNs such as the data loss due to the downsampling in the polling layers or the input rotation and shift. Input is a vector whose length represents a probability while each entry of the vector encodes a parameter such as rotation. Different from the nets seen before this adopts a squash function, shown below, and instead of polling layers adopts a dynamic routing algorithm. Structure of the Capsule model is shown in figure 1.12

$$\vec{y} = \frac{|\vec{x}|^2}{1+|\vec{x}|^2}\frac{\vec{x}}{|\vec{x}|}$$

NasNet

NASNet is the first popular NN model designed with neural architecture search. NasNet was created by searching a simple cell for a dataset in a small search space. The complexity of the model can be determined by the number of cells stacked together.

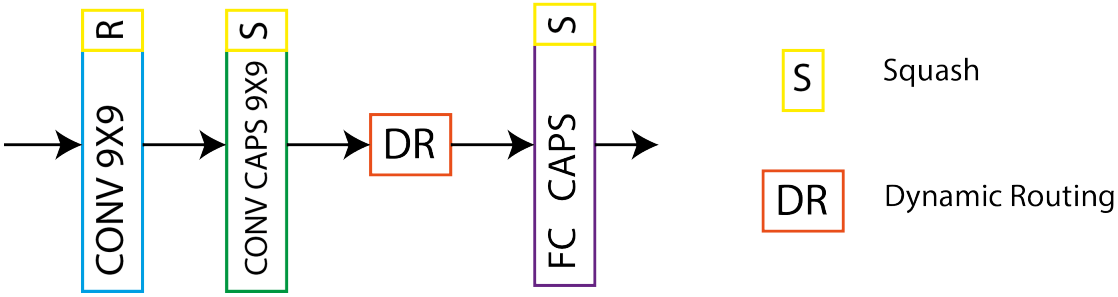

Figure 1.12: NasNet module[3][11]

Chapter 2

Based work and methodologies

2.1 Based work and methodologies

2.1.1 Introduction to based work

In this chapter previous work is explained to understand well what is our starting point to improve performances. In order will be explained:

- Neural network parallelism types: Spatial and Temporal
- Dataflow processing
- Stand-alone vs system on chip
- Comvolution remind
- Base SMAC engine: how the original engine, that will be our starting point, was derived

2.1.2 Spatial and Temporal

Is important to clarify that the fundamental operation done by the architecture is the MAC (Multiply And Accumulation) operation for both CONV and FC layers. This operation can be done in different ways based on which performance wants to be achieved so different hardware can be used and different architectures can be identified. Mainly two types of architectures, that behave in opposite ways, can be identified:

- Spatial: GPU and CPU are typically fully temporal

- Temporal: ASIC and FPGA based designs are usually fully spatial

Both of these architectures have several numbers of PE (Processing Elements), an example of the two types of architecture, which will be explained in the next subsection, is shown in figure 2.1. Despite this classification real architecture adopts solutions from both architectures to have a final result that is more efficient than going only in one direction.

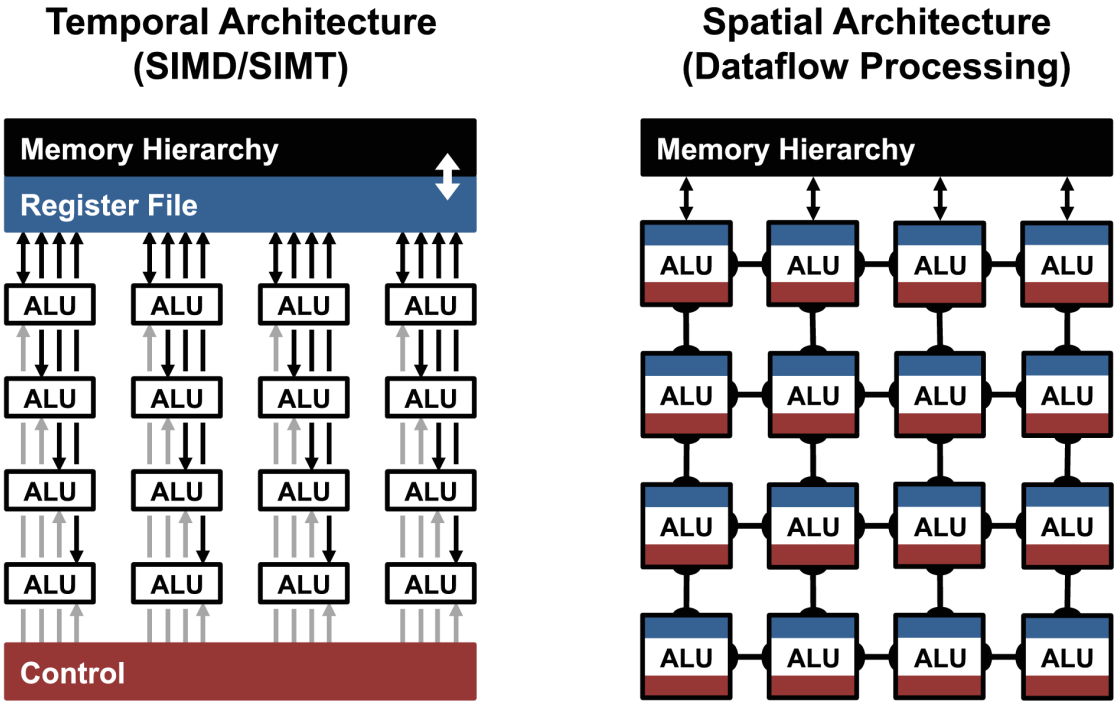

Figure 2.1: Temporal and spatial architecture [2]

Temporal Architecture

This kind of architecture can adopt vector processing as SIMD (Single Instruction Multiple Data) or SIMT (Single Istruction Multiple Threads) There are several instances of ALU (Arithmetic Logic Unit) that work simultaneously. The scope is to extremize the performances. The ALUs fetch data from the memory but don't communicate between them so this architecture is memory inefficient. To improve the throughput based on dimensions also an algorithm can be applied to the input feature maps and weights. In general for large dimensions is convenient applying FFT (Fast Fourier Transform), instead for small dimensions is convenient to apply the Winograd algorithm [12].

Spatial Architecture

Opposite of temporal architecture here we want to optimize as much as possible the dataflow. To obtain this ALUs can communicate and share data, so the data fetched from memory are kept locally and used for more computation. This allows having lower energy consumption due to the lower number of memory access since the computational cost can be several orders of magnitude lower than the memory access.

2.1.3 Dataflow

As seen before a bottleneck in terms of power consumption is the memory access. One solution that can be adopted to mitigate this problem is a hierarchical memory organization. Hierarchical memory has as an aftermath the different approach to dataflow. Between this approach is possible to distinguish:

- weight stationary
- output stationary
- no local reuse
- row stationary

Weight Stationary

This solution, shown in figure 2.2, maximizes the weights reuse keeping them local, on the other side partial sums are continuously fetched and written back to memory. This solution is employed in the HWCE (Hardware Convolutional Engine) [13].

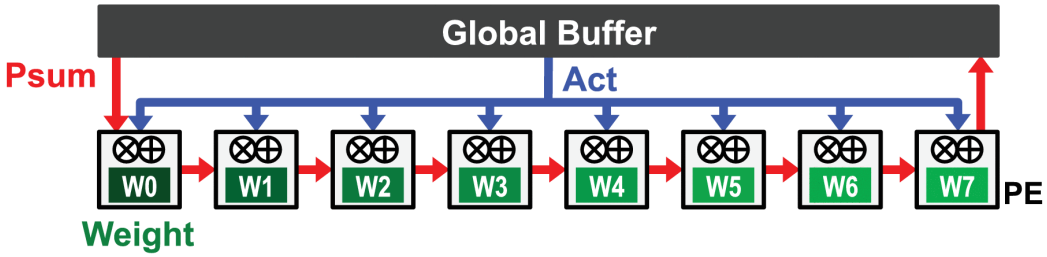

Figure 2.2: Weight Stationary Example [2]

Output Stationary

This solution, shown in figure 2.3, keeps local the partial sums, this minimizes the memory access to read them and write back. Only when the operation is completed are written back. This solution can be combined with the Wight Stationary solution. A solution that employs weight and output stationary is ShiDianNao[14]. A second solution can have input and output stationary, an example is the XNOR Neural Engine [15].

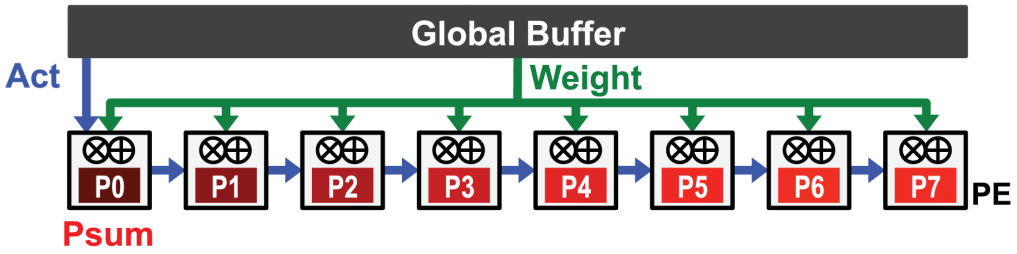

Figure 2.3: Output Stationary Example [2]

No Local Reuse

Despite the other this solution, shown in figure 2.4, has the highest power consumption due to the elimination of the storage elements and trying to increase the global buffer. This solution is employed in the DianNao but also here to mitigate a minimum the power consumption some registers are employed inside the processing engines

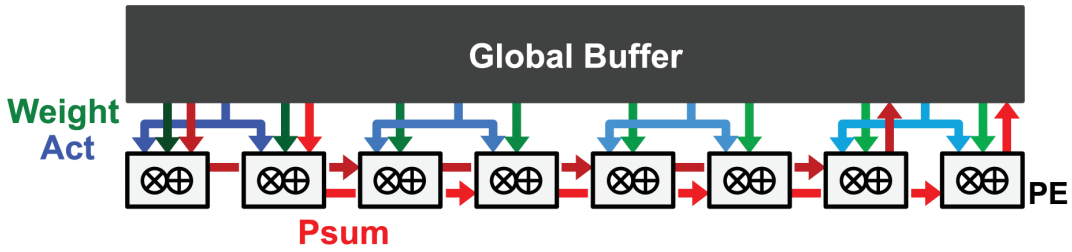

Figure 2.4: No Local Reuse Example [2]

Row Stationary

This solution try to extremize the data reuse, so not only weights or partial sums but also activations. Respect the previous solution this optimizes furthermore

the power consumption. This solution is employed in the Eyeriss [16] where an intelligent memory hierarchy is adopted.

Reduce precision

To optimize further in terms of power consumption, memory optimization, and execution time, is possible to make a trade-off between precision and energy. This need comes from moving our application from the data center to the sensor. Many applications (ex Internet of Things) have been limited due to the amount of data that has to be transferred from the sensor to the data center. The higher the data amount higher is the power consumption. If the operands have a reduced precision is possible to:

- reduces the overall memory: this allows to have a size and power reduction.
- reduces the cost of MAC operation in terms of cycles (this will be clear later with the introduction of the serial approach) and also power.

At the beginning the focus was on reducing the precision of the weights due mainly to the great advantages of reducing memory without affecting significantly the final result. The quantization should be done to minimize the error between the quantized data and real data. 8 bits quantized operands have a memory footprint 4 times smaller than one of 32-bit. In terms of power consumption, a MAC operation with 8 bits consumes 20 times less energy compared to a 32 bits MAC operation. Some recent researches show up that also reducing the precision of the activations, despite the initial trial that focused only on weights, can introduce advantages without affecting strongly the final result. A reduction of the operands between 4 and 9 bits shows up a smooth accuracy on the final results (less than 1%). About quantization many methods can be applied, the most hardware-friendly are:

- Linear quantization: all the quantization intervals have the same dimension/length
- Non-linear quantization: quantization intervals have different dimension length, this is more suitable when weights and activations assume a non-uniform distribution

Despite the reduced precision of the operands MAC operation is done by the engine taking into account the worst case model, so the internal parallelism is greater than the operands but before the write back the result is reduced again.
An important mention should be done at this point due to the fact that there is also the possibility to reduce the number of operations to improve performance but this is the starting point of our work so the possible solution and our choice and application will be explained later.

2.1.4 Stand-alone vs System on Chip

Some of the hardware mentioned before have been realized to work as stand-alone components so they perform the same task continuously. In opposition to this hardware are the HWPEs (Hardware Processing Engines), which are special-purpose memory-coupled accelerators. These accelerators aim to efficiently perform a specific task and not the whole job to get better performance and energy efficiency. A peculiarity of HWPEs, shown in figure 2.5, is the presence of DMA (Direct Memory Access) so they can operate directly on the memory that is shared among the elements. This solution allows data to be seamlessly exchanged between accelerators and cores as it happens in Fulmine [17] and XNOR Neural Engine [15].

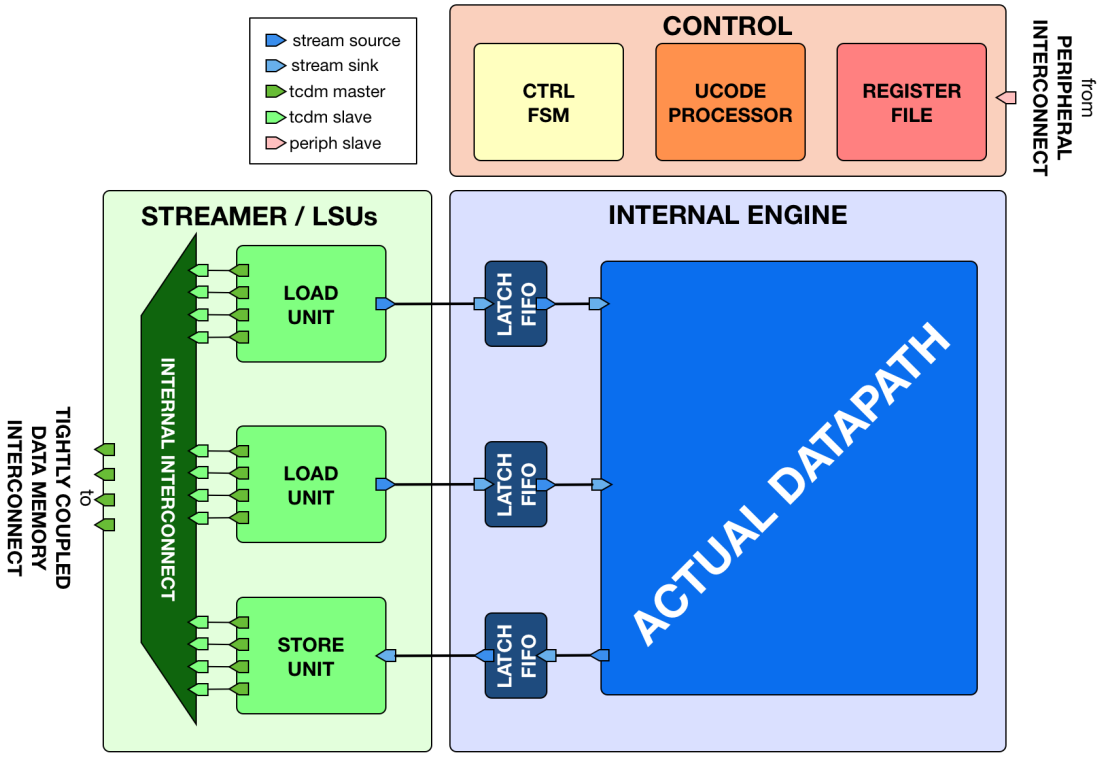

Figure 2.5: Hardware Processing Engine

2.1.5 Convolution Remind

Convolution Approach

Before introducing the architecture is important to clarify which kind of operation is done. Consider a generic convolutional volume, a 3D structure whose dimensions are n_H n_W n_C for height, width, and channel; and n_F filters with dimension f f

n_C. Cells of the volume represent the activations. As has already been explained activations are the input for a layer of a NN and are generated from the previous layer. Applying the rules of convolution, which will be explained better in the next section, between them, without padding, will produce a volume with dimensions n_H n_W n_F. The convolution starts from the upper left angle of the original volume and finishes at the bottom right. The engine should be able to work with the original volume and filters to produce the output volume. The three kinds of volume are shown in Figure 2.6.

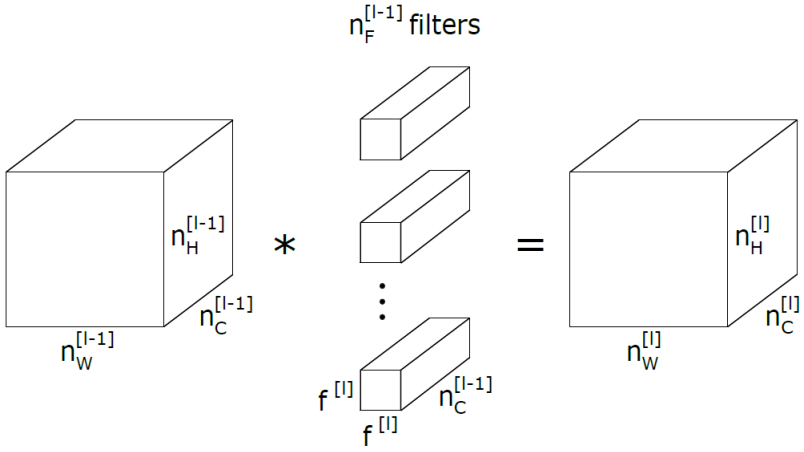

Figure 2.6: Convolutional volume: original, filters and final [18]

Convolution rules

The basic idea for convolution and the model applied to our structure will be explained here. Consider a generic convolutional volume, in this case, a 6x6 volume, and a generic filter, in this case, a 3x3. Looking at figure 2.7, the volume on the left indicates the starting volume in the middle there is the filter and on the right the output volume, the red non-continuous line indicated the portion of the starting volume that we are considering. Starting from the upper left position a basic multiply and accumulation operation is done. after that, all the accumulation between multiply has been completed one output is produced, and the dot in the volume on the right indicates the position of the output produced by the convolution. Then looking at step 2 the starting position is slid to the right of the position and the MAC operation is repeated. this will be repeated until the right end of the volume, step 3. At the right end, it slides down from one position and slides also to the extreme left position, step 4. This goes on until the bottom right position, step 5.

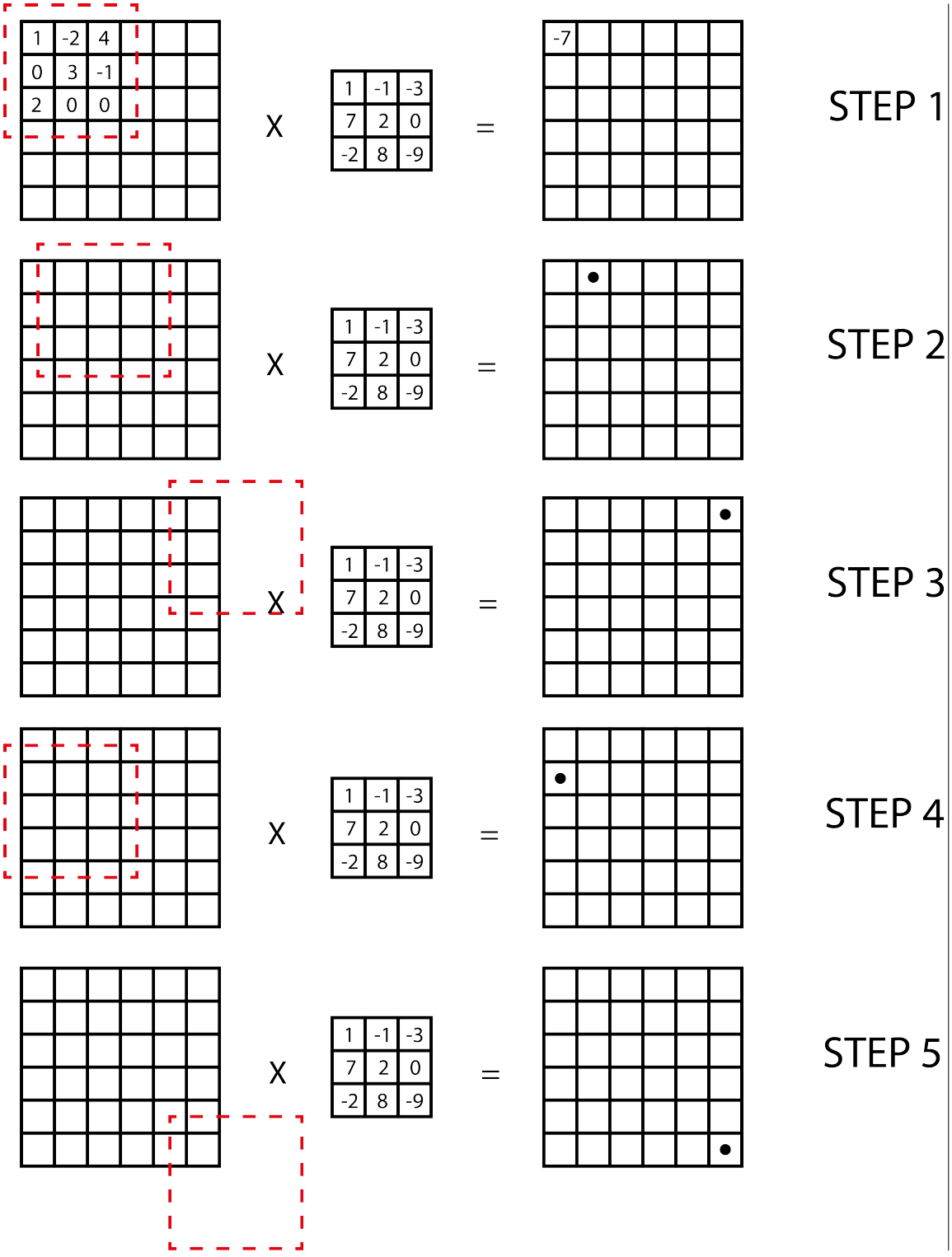

Figure 2.7: Convolution Step

Before going on a quick reminder to the 3D structure is necessary. As for the 2D structure, the MAC operation is done and then from the starting point (upper left), we reach the bottom right. A single filter produces a 2D structure, and applying more filters produces a 3D structure. For example, consider a 6x6x6 volume and a 3x3x6 filter, doing the convolution between them produces a 6x6x1 volume, and repeating it with the other three different filters produces at the end a 6x6x4 volume.

2.1.6 Base SMAC engine

What follow is the description of the derived architecture and explains also the reasons that brought the necessity of a new one. Starting from the original structure that employs a parallel approach and step-by-step deriving a structure that employs a serial one.

Parallel structure

The starting point is an engine able to manage to multiply and accumulate. The first structure possible is a parallel structure like the DaDinNao [19] where the operations are performed in parallel. A structure similar to the DaDinNao is shown in figure 2.8. In this structure, there is one multiplier for each activation and weight, so if M activations and weights are taken into account for time there are M multipliers and an adder tree for accumulation. Is clear that this structure is the fastest possible, in terms of the cycle's number, and can do M MAC operations for the cycle, since the multiply between weights and activations is done in one cycle, but there is also an explosion of the area due to the presence of M multipliers. Depending on the order of M also clock frequency could be much reduced.

Changes

The first change is in the approach to the bits, is possible to switch from a parallel approach to a serial approach close to the Loom structure[20]. As in the parallel approach M activations and weights are considered. The serial structure allows us to simplify the multipliers in AND ports, this is possible because we consider one bit for time, so instead of M multipliers there are M AND ports. Starting from this change the AND port does the operation between LSB of Weight and LSB of activation, then the weight shift from cycle to cycle from LSB to MSB while the activation does not shift. When the operation is done with the MSB of the weight the activation shift of one position and the weight comes back to the LSB. the bits shift in this way until the MSB of activation. the out of all AND ports are summed and then shifted and accumulated two times. the first accumulation is in the AC1 when the weight bits shift and the second accumulation is in the AC2 when the

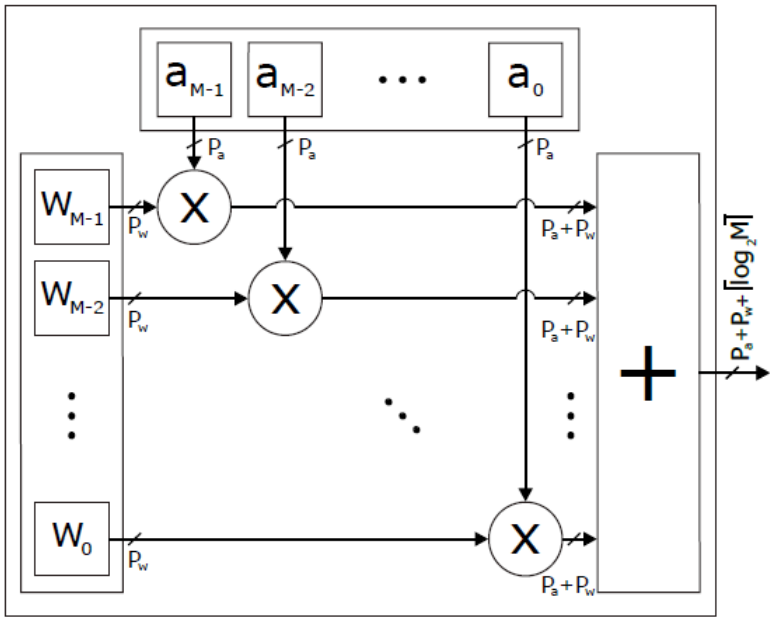

Figure 2.8: SMAC parallel structure [18]

activation bits shift. Between the AC1 and AC2, there is the negative block that samples and complements the output of AC1; refers to the figure 2.9 for the serial structure with AC1 and AC2. After the AC2 there is the AC3 that accumulates

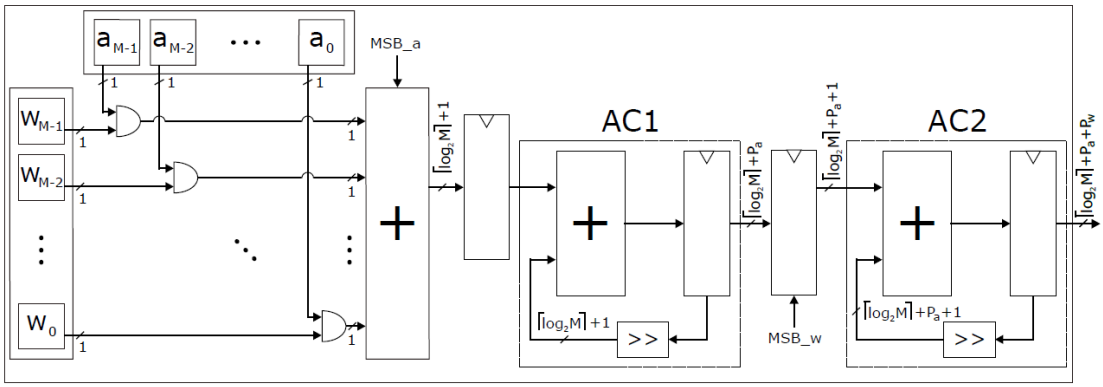

Figure 2.9: serial structure with AC1 and AC2[18]

the results of successive accumulation between M activations and weights. This structure requires less hardware but for M activations, with parallelism Pa, and weights, with parallelism Pw, are necessary Pa X Pw cycles. So to reach the previous throughput this structure must be replicated Pa X Pw times. In the final

structure, the activations are shared between the Pa X Pw structure but every structure works with a different filter. To improve efficiency in activations use AC2 and AC3 have four accumulation registers. The complete single SMAC with AC3 is shown in figure 2.10 while the complete engine is shown in figure 2.11.

Operands number, frequency and area

An important parameter is M, the number of operands that act in parallel. This parameter influences the frequency and area of both parallel and serial structures. Is possible to see that for both the lower number of M provides the higher operating frequency and minimizes the area, it also seems to have a more linear area incrementation when the operating frequency is raised. The parallel solution has a higher area with fixed frequency but can operate at the end at a higher one, almost double. A single SMAC has an area that is one order of magnitude smaller than the parallel solution, and a serial SMAC that reaches the same throughput as the parallel solution instead has an area that is around 2.9 times greater. Chose an M = 16 seems to be a reasonable solution that makes a deal between area and frequency.

2.1.7 RELU

As explained before the internal parallelism is much larger than the operand parallelism, based on the worst-case model to avoid the computational error, but the output is still reduced respect to the internal parallelism. At the full range output, the RELU function is applied.

- if the result is negative the output is taken to 0
- if the result is positive but cannot be expressed on 8 bits it is saturated to the maximum value that we can express with 8-bit
- the result is positive and lower then it is simply truncated to 8 bit

As explained also in chapter 1 ReLU function is fundamental to train the NN. ReLU cut to zero the negative values and this makes the null activation percentage in a range that goes from 50 to 70.

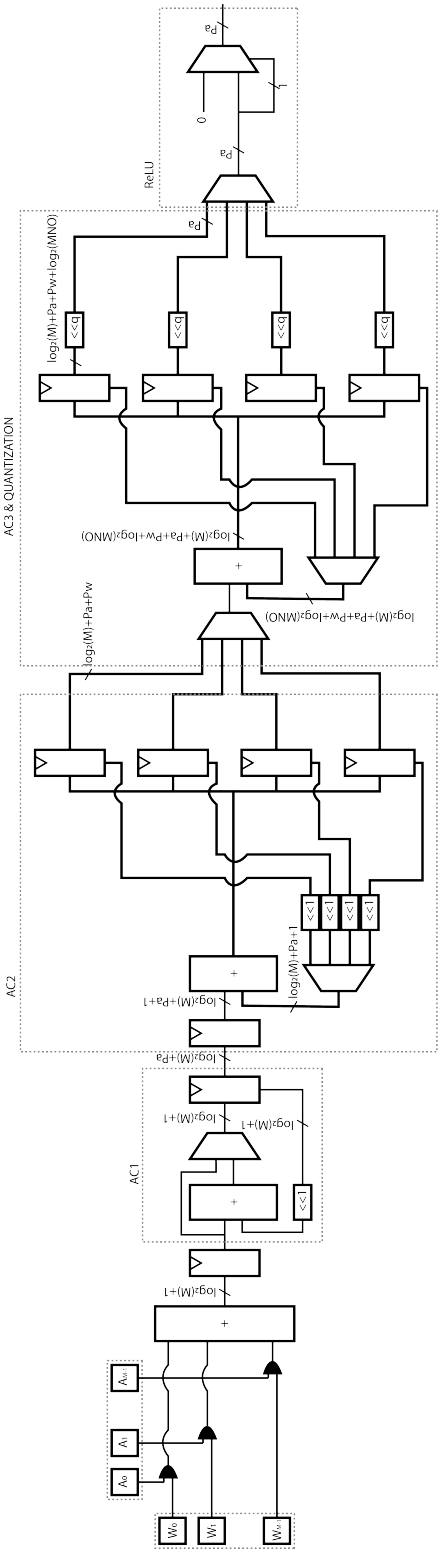

Figure 2.10: serial structure complete [18]

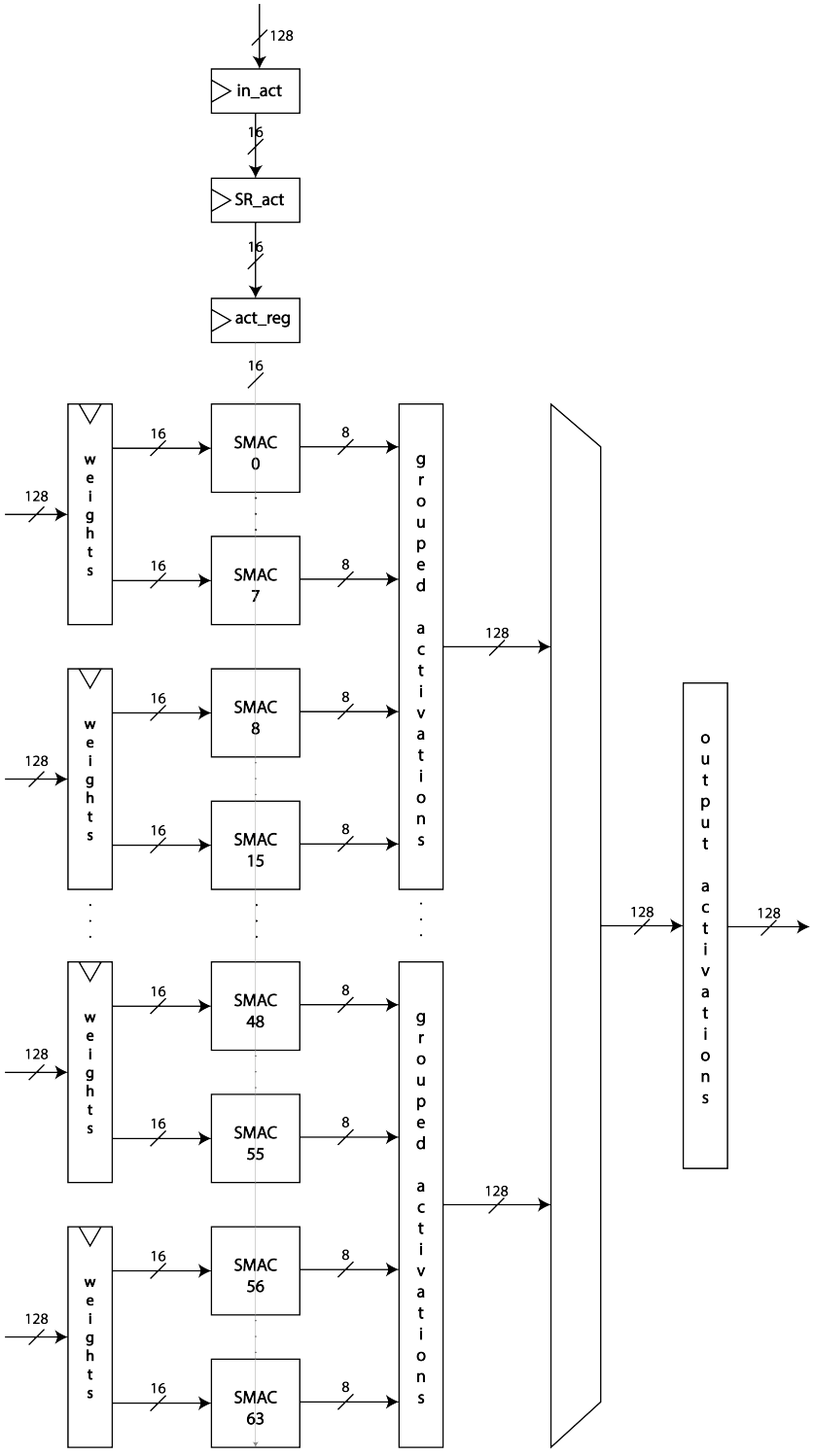

Figure 2.11: full engine [18]

Chapter 3

SMAC Engine with sparsity managment

3.1 SMAC Engine with sparsity managment

3.1.1 From sparsity map to SMAC structure

The starting point to improve performance is the choice of a correct compression method and then adapting the previous structure to the new data format. The compression method is introduced due to the presence of sparsity. Sparsity is the presence of zero-value activations in the original volume. Considering that the engine does MAC operations so the final scope is skipping the operation when there is a zero-value.

3.1.2 Compression

There is a different kind of compression method but our choice was basically between the following four that can result more adaptable to our engine.

- CSR: Compressed Sparse RoW
- CSC: Compressed Sparse Column
- RLC: Run Length Coding
- Sparsity Map

Spartisty MAP

After some initial consideration of the various compression methods, the choice follows on Sparsity Map compression method. This particular compression method

follows these rules:

- In the convolutional volume, non-zero values become a 1 and the zero value becomes a 0. This allows us to convert the volume into a 1b volume.
- Non-zero values are stored in an array that follows the row direction, there is an array for every row.

Refers to figure 3.1 Here we have a clear example of how the volume is after the rules of the Sparsity map are applied. The volume on the left is the original, after the compression (the arrows indicate the compression) in the center we have a volume composed only of 0 and 1 values. On the right, are the arrays with the effective value in the adjacent position.

The previous example was on a 2-D volume now focusing on our real volume

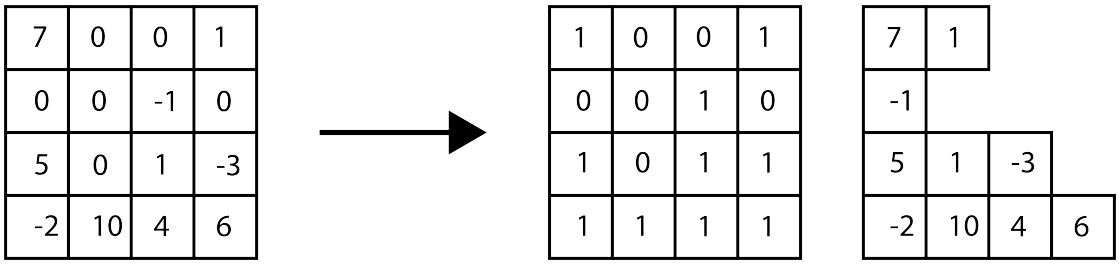

Figure 3.1: Sparisty Map example

which is 3-D. Consider that we analyze the volume in the channel way, so before going on in the row way we have to analyze the whole channel. The rules applied in the 2-D structure are the same. The structure becomes a 1-bit structure as before. Instead, the array stores the non-zero values of the first channel (row 1 column 1), then stores the values of the second channel (row 1 column 2). This array structure is replicated for every row.

3.1.3 Diffrent approach to the convolutional volume

Considering the new data format of the convolutional volume we have to change the approach to the data. First of all, consider a different approach to the input feature map. the original structure considers a 3x3 sliding window that slides to the right of one position for time. In the end, it goes to the beginning and slides down one position; and then restarts to slide to the right of one position for time. The new sliding window is a 4x1 window that slides to the right of 1 position and at the end slides down of 2 positions as shown in the image below. These new sliding windows are due to the new idea of elaborating the data that will be explained in the next section. Figure 3.2 shows the new sliding windows. For example, here there is a 6x6 volume, from left to right we have:

- the starting position, so the first portion considered
- the first sliding to the right so the portion considered after the slide
- portion considered after the last sliding right
- portion considered after the first sliding down, now it starts again sliding to the right till the last column
- portion considered after the last sliding down, as before it will slides again to right till the last column

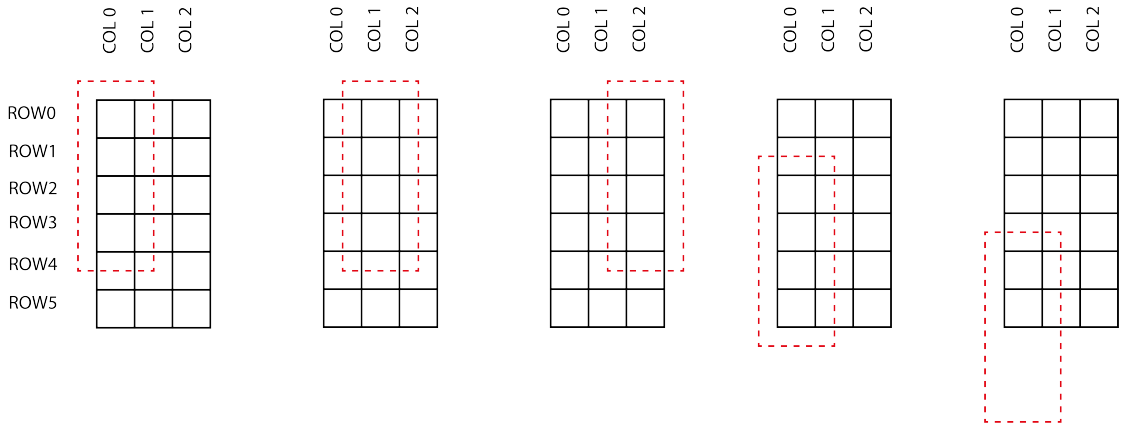

Figure 3.2: New volume approach with sliding window

3.1.4 New SMAC Structure

Reducing SMAC number

The original SMAC structure was composed of a 64 SMAC that shares the activations. Refers to the figure 3.3 that shows the new SMAC structure composed of 6 SMAC and 2 AC3. As explained before the new windows consider 4 rows at times, so as in the original the activation are shared but not between all SMAC, the activation of row 0 is not shared, row 1 is shared among SMAC 2 and 4, and row 2 is shared among SMAC 3 and 5, row 3 is not shared. Now consider what happens when the third column has been computed, at this time our structure can produce two outputs, and since now every time before the windows slide to the right till the end minus one it produces 2 outputs, and at the end, it produces 6 outputs.

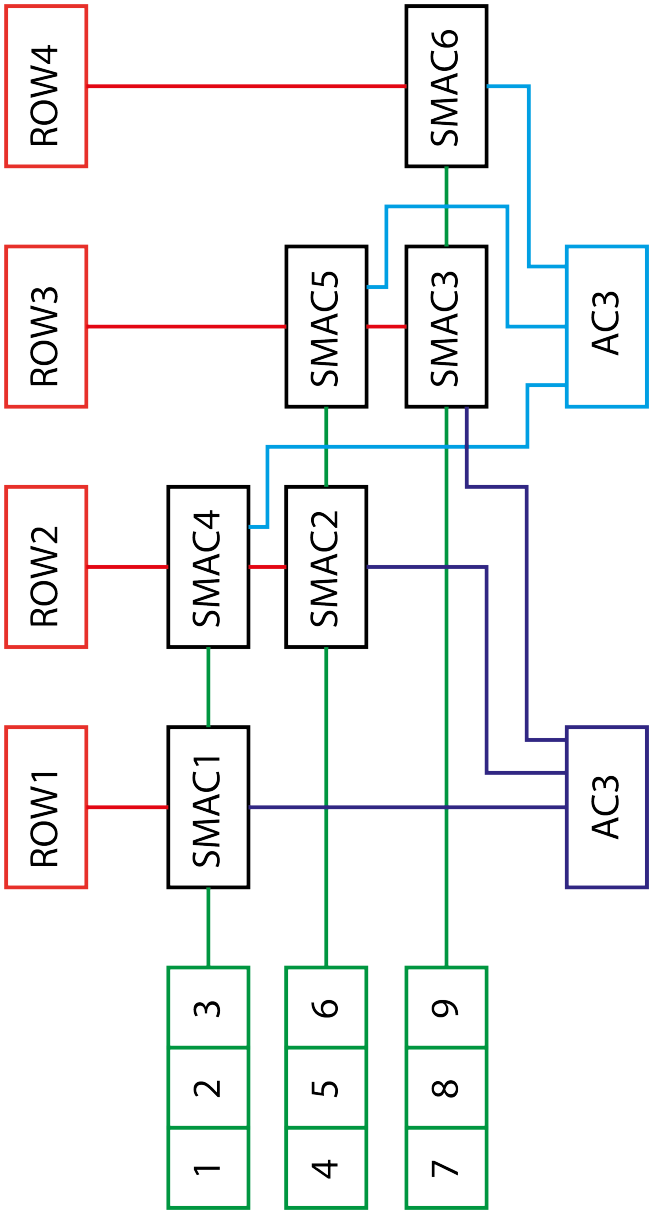

Figure 3.3: Full structure with 6 smac and 2 AC3

3.1.5 New SMC structure

A direct consequence of the changes applied in the sliding window and the number of SMAC is a change in the internal structure of the SMAC.

AC1 and AC2

first of all, let's analyze the first and the second accumulator. before the ac1 the following block is present

- the AND GATE that operates between the activations and weights bits
- bit adder that doing a bit-a-bit addition with the output of the AND GATE
- bit register, a simple register out of the bit adder

AC1 is the same as the original SMAC. It is composed by

- the ac1 adder used to accumulate the new output of the bit register with the previous one
- ac1 register that samples and shift the output of the bit adder
- a mux out of the ac1 adder is present to bypass it for a new value for the ac1 register

between ac1 and ac2 is present the negative block that samples and complements the out of the ac1. The AC2 instead is changed. The number of output registers is reduced from 4 to 1 due to a different approach to the weights. with different parallelism, the internal block is the same as the AC1 exception is the mux that is not present. So as in the original, the AC1 accumulates the weights shifting and the AC2 accumulates the activations shifting. the structure of the AC1 and AC2 with full parallelism and intermediate block (neg block) is shown in figure 3.4.

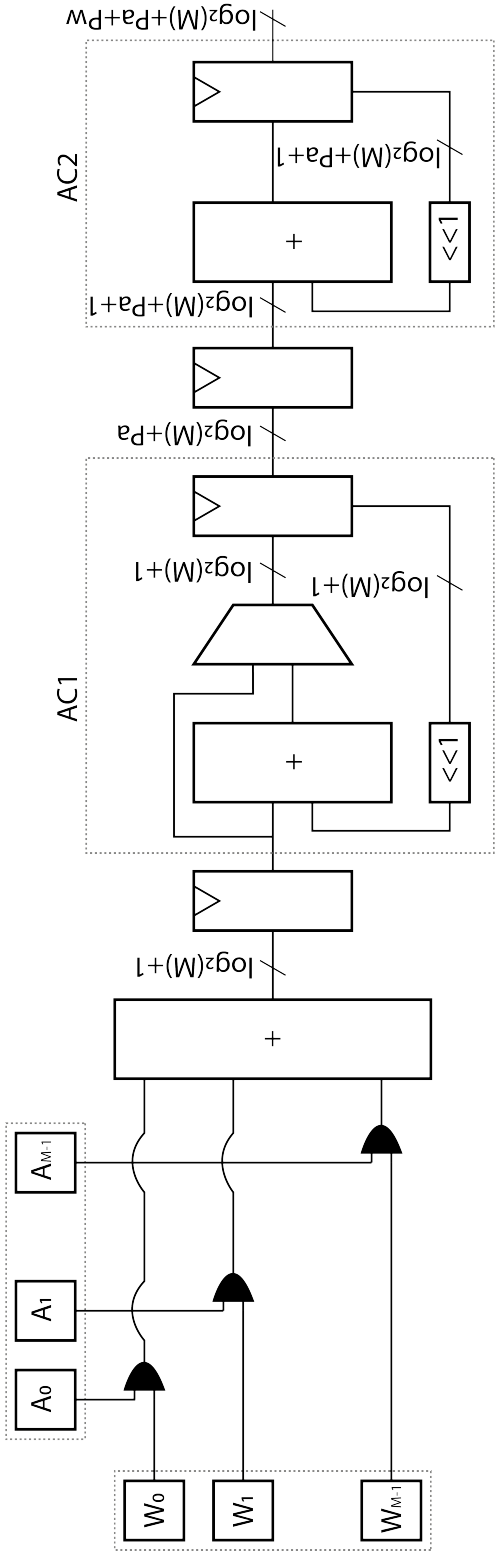

Figure 3.4: SMAC with AC1 and AC2

3.1.6 SMAC low-level FSM

A low-level FSM that can manage the accumulation in AC1 and AC2 is necessary. the state evolution is shown in figure 3.5.

Is possible to notice that there are two types of states:

- wait states
- elaboration states

Low-level FSM wait states

The low-level wait states are all the states in which the engine is not elaborating. These states are:

- IDLE: this is the reset state
- WAIT: this is the first state after the idle and after the end elaboration. here is waiting for the two start signals, the first from ac3 and the second from the high level that notify that the activation and weights are ready in input ad the engine can start the elaboration.
- WAITAC3: in this state, the engine is waiting for the starting signal from the AC3
- WAITACT: in this state, the engine is waiting for the starting signal from the high-level FSM
- ENDACC: is the state that notifies the end of the accumulation.

Low-level FSM elaboration states

these are all the states in which the SMAC is working doing the accumulation. they are:

- INBITREG: in this state the bit register is sampling the input weights bits.
- ACC1: here we have the accumulation of the ACC1
- NEGBLOCK: here the neg block is sampling the out of AC1
- AC2: in this state, the AC2 accumulates the out of the negative block.

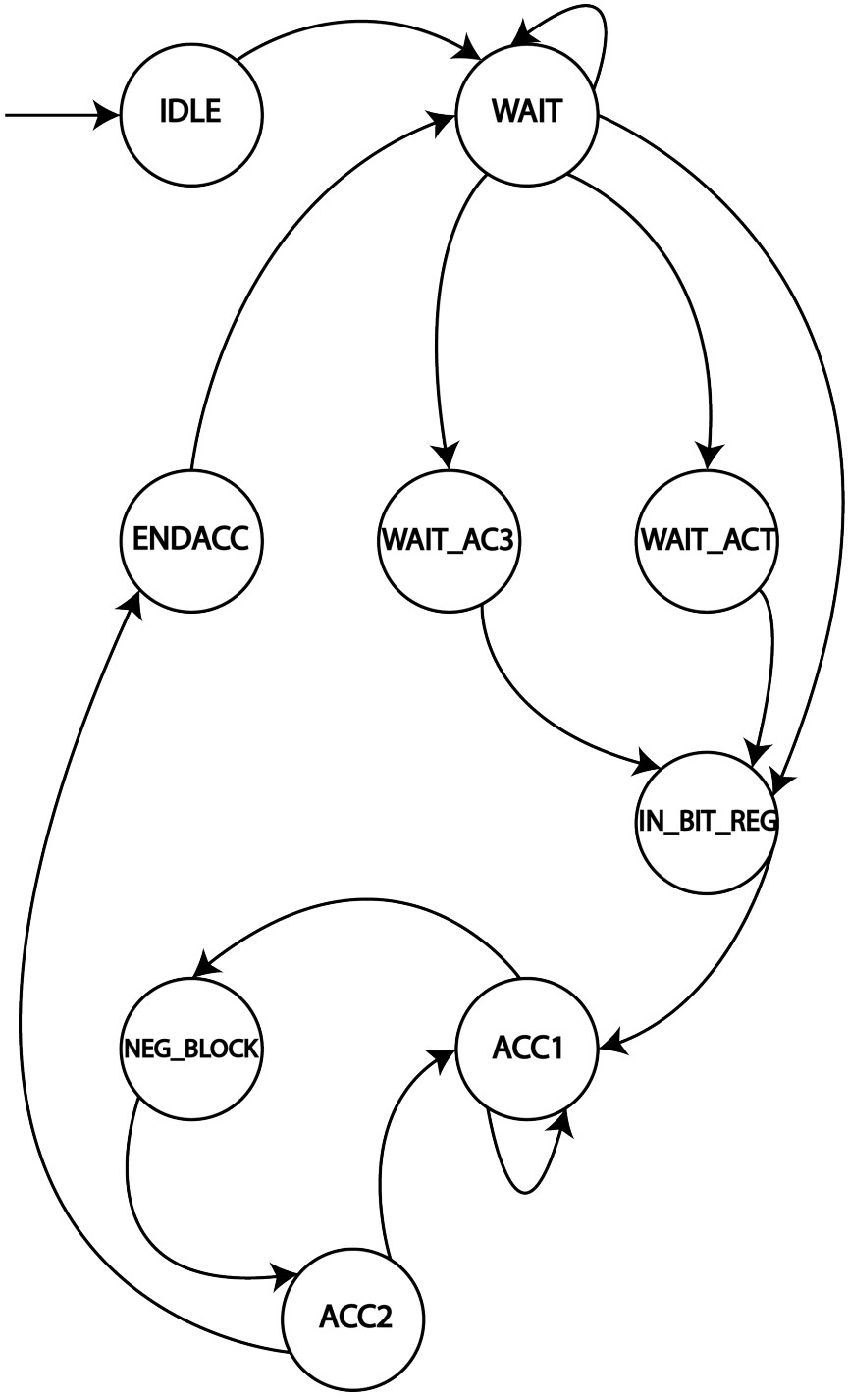

Figure 3.5: Low Level FSM

AC3

The main change is in the AC3. Here we have as in the AC2 a reduction of the accumulation registers from 4 to 3, the parallelism has also been resized to be adapted to our structure and accumulation way. The main reason for the reduced register number is due also to a different approach to weights, in the original structure every register accumulates results of different weights filters, in this structure, the weights are from the same filter but every register accumulates different weights column based also on the position of the sliding windows. In this state, the engine is waiting for the starting signal from the AC3. The image below is shown the ac3 with full parallelism. In the second image instead is shown the full structure is composed of three smac and one ac3.

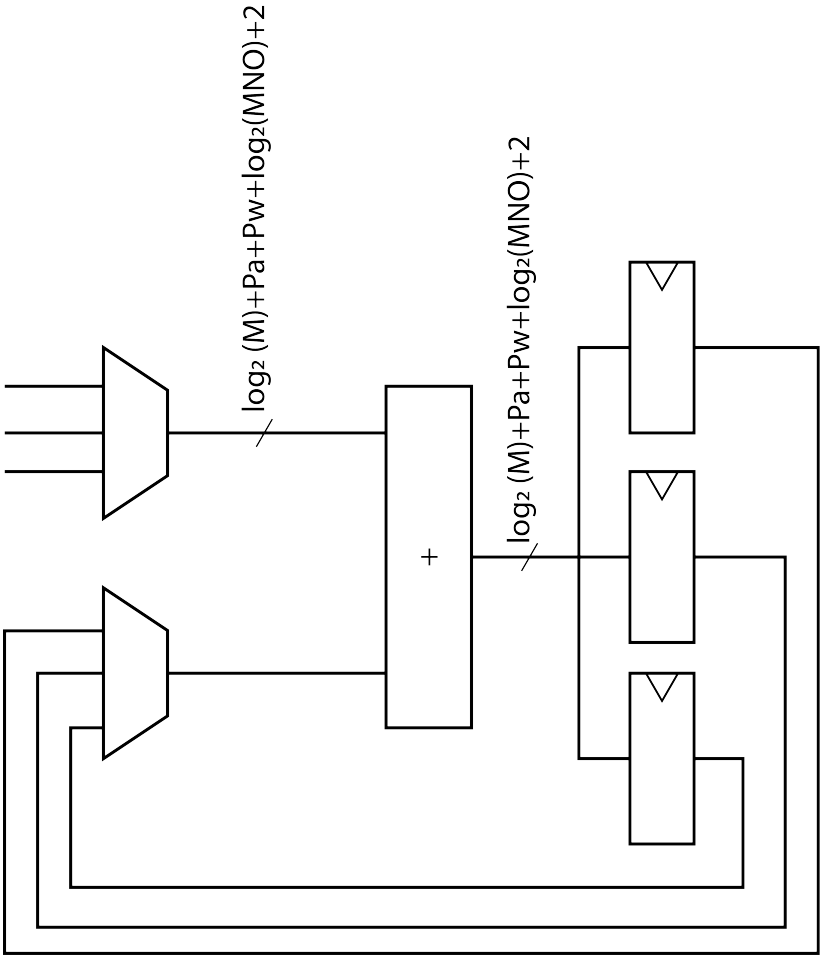

Figure 3.6: AC3 structure with parallelism

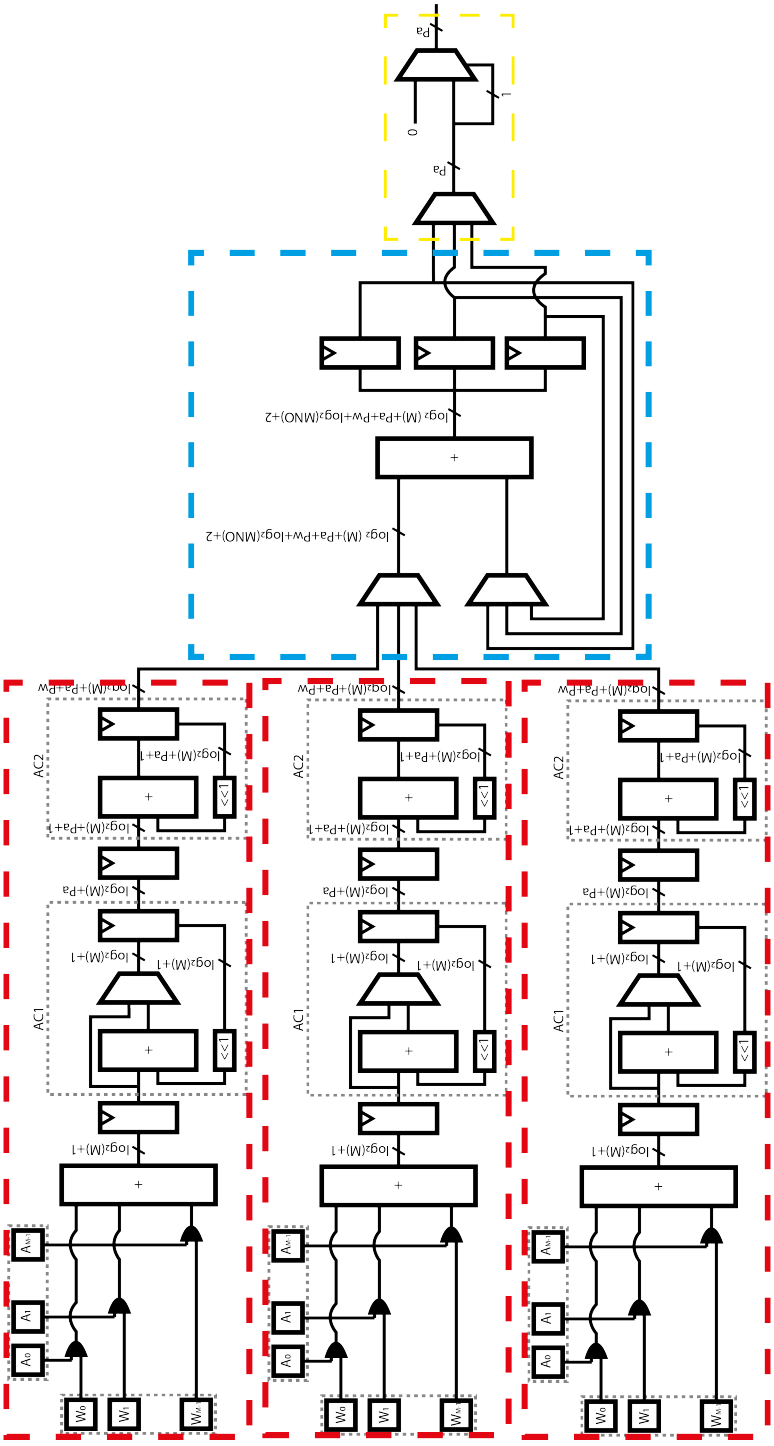

Figure 3.7: Complete SMAC engine

SMAC and AC3 FSM

the low-level fsm for ac1 and ac2 communicates with the AC3 FSM with a handshake protocol. When a SMAC has finished an accumulation an end accumulation signal is asserted, now the engine waits that the AC3 has accumulated it. when it is done the AC3 communicates it to the SMAC with the proper signal asserted so the engine can begin a new accumulation.

Accumulation scheme

Consider the image below that shows the position of the sliding window and based on this position, the accumulation scheme. Starting from the first position of the window the first block of SMAC (1 to 3) work with the channels of the first weights column (channel 1, 4, 7), results of this first accumulation are accumulated in AC3 register 1. Since there is no padding at the beginning while the first block works the second does not then the role are inverted so we are still in the first position of the sliding window but only the second block of SMAC (4 to 6) works, still with the channels of the first weights column (channel 1, 4, 7) and accumulates the results in AC3 register 1 of block 2. At this point, the sliding window goes on of one position, and now we are in the second column of convolutional volume, in this case, the first block work as before with the weights but the accumulation is done in register 2 of AC3 but also the second block work at the same time but with the second weights column, results of the second block are accumulated in the register 1 of AC3. After this, the block starts work with the second weights column (channel 2, 5, 8) and accumulates results in register 1 of AC3 while block 2 does not work, at last, the second block works with the first weights column and accumulates in the register 2 of AC3 while the first block does not work. Now the window goes on further of one position, the first block as previously works at the beginning with the first weights column weights and accumulates in register 3 of AC3 while the second block works with the second weights column and accumulates in register 2 of AC3, then the first block work as previous with the second column and accumulate in the register 2 of AC3 but at the same time the second block works with the third weights column (channel 3, 6, 9) and accumulate in the register 1 of AC3. Now the final step before that the first output is available, at this point the first block works with the third weights column and accumulates in register 1 of AC3 and the second block works with the first weights column accumulating the results in register 3 of AC3. Now register 1 of AC3 of both smac blocks produces the first output and then is clear. When the sliding window goes on the working scheme with the weighs is always the same as in the third position the only changes are in which register is accumulated and which register goes into output, starting from the fourth position the registers that go into output register 2 then in fifth position registers 3 and restart from the sixth that are registers 1. At the last position, all

three registers of both SMAC block goes into output.

3.2 Weights

Memory introduction

Due to the new scheme and weights approach a local memory introduction is necessary. based on the worst case of the weights parallelism and channel length the memory adopted is 1x1x8x512. this memory is replicated 9 times in a 3x3 structure that is the dimension of the original sliding windows. for simultaneous reading of different memory blocks memory is considered in 3 rows and every row is composed of a 3 channel. every row has also 3 2to1 mux that are used for the correct address and 2 3to1 mux that are for the output data.

3.2.1 FSM

One of the main consequences of the new structure is that radical changes in the high-level FSM are necessary. There are also some secondary FSM that are subordinated to the main and that manages all the other aspect such as memory filling, filtering, and data elaboration. These are:

- MAIN FSM: is the highest level FSM that manages the various fase like memory fill, map scrolling, data elaboration, column and row updates, data output, etc...
- Memory fill: this FSM manages the memory fill with the weights.
- Kernel FSM: this FSM generates the correct kernel position.
- Filtering and starting elaboration: is a merging of other FSM that manage when start filtering, starting elaboration, resume filtering.

going deeper in the smac structure there are also other FSM:

- ac1 and ac2: the FSM that manages the accumulation in this two accumulator, this has been described before
- ac3 FSM
- filter: this FSM manage the sparsity map scrolling and filtering

3.2.2 Main FSM

We can categorize the states of the high-level FSM in three different kinds of states:

- Starting states
- Update states
- Elaboration states

before going deeply into states function the state evolution is shown in figure 3.8.

Starting States

The starting states are all the states in which the engine is ready to start the elaboration

- IDLE: the reset state
- WAIT: in this state, the engine waits for the starting signal to start filling the memory
- END:
- FILL: In this state, a continuous stream of weights data is provided and all the memories are filled with them.

Elaboration States

Here we have the real elaboration that is SCMAP. In this state, the four-channel row of the sparsity map is provided to the filter from 16 to 16 bits. Bits are filtered to remove 0 and select the correct weights that are sent to the SMAC that do the accumulation, this is repeated until the end of the four channels. the second elaboration state is OUTMAP. In this state, the output is ready so a signal is provided to the out when the output has been read the FSM goes on. Except for the last column 2 outputs are provided, and in the last column, 6 outputs are provided.

UPDATE States

These are all the states where the position of the volume is updated. these states are:

- KERplus1: Here simply the kernel counter is updated and some counters and registers are reset

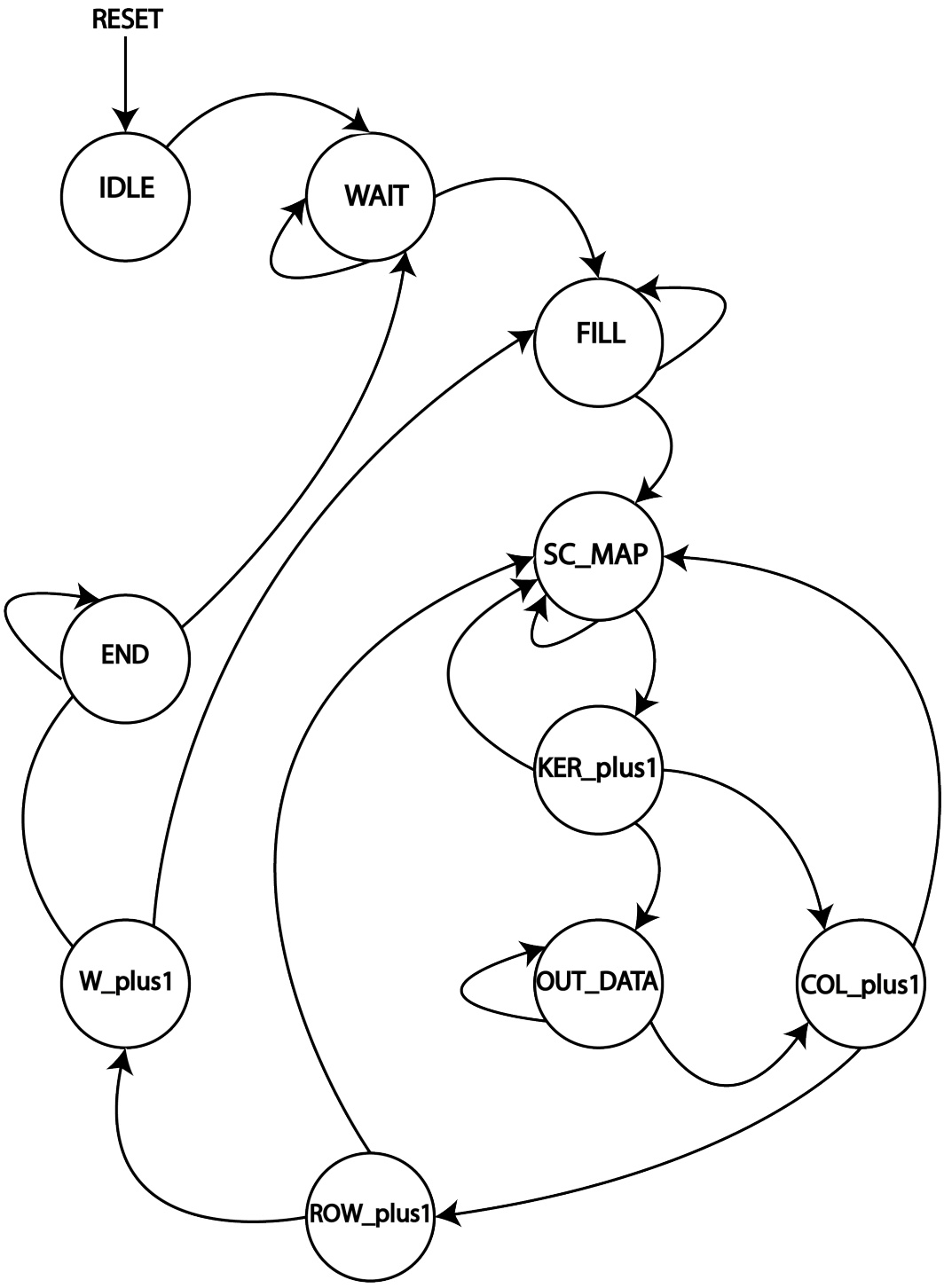

Figure 3.8: High Level FSM

- COLplus1: As in KERplus1 here when all the 3 kernels have been computed for the whole channel of the four rows the column counter is updated. based on in which column is also a register of the AC3 is clear.
- ROWplus1:Here all the registers of the AC3 and the col counter are clear while the row counter is updated.
- Wplus1: The row counter is clear and the weights counter is updated, the next state, if we are not in the last weights group, is FILL so the memory has to be refilled with the new weights.

3.2.3 Elaboration counters

To help the main FSM manage the filter and elaboration there are three counters:

- M counter: this counter counts up to CH/M-1 and is updated when a new portion of the sparsity map is available.
- Elbaoration counter: count when an elaboration has terminated
- M filtered Counter: this count is updated when M non-zero value has been filtered and so ready to be elaborated.

When the elaboration counter and M filtered counter are equal and we have filtered the last M bits of the channel of the sparsity map it means that we have filtered and elaborated the whole channel so kernel position and so on can be updated.

3.2.4 Memory filling FSM

This FSM fills the memory with the weights. it only works when the main FSM is the state FILL.

3.2.5 Filtering FSM and elaboration

The filtering and elaboration are managed by a collage of various FSM.

Startfiltering

The starting point is when a start filtering signals could be generated. first of all, memory must be filled so we have to be in the state SCMAP of the main FSM. There could be two types of signals that could be generated, the first is when a new portion of the sparsity map is available, and the second is when previous filtering was interrupted, so it has to be resumed and brought to an end.

Filtering HLFSM

This FSM in a way similar to the filter FSM manages the memory reading when filtering, so it takes into account the previous situation and based on this manages the memory signals and the address update.

Start After Wait2

Here we manage when a new filtering operation can start. this FSM takes into account if the smac elaboration and accumulation are on the run or if it has been brought to an end.

3.2.6 KERNEL FSM

This FSM takes trace at which point of the accumulation the engine is. For every column, except for the first and the second, the engine has to compute the accumulation for the three columns of the memory. This taking trace is important for the SMAC to know from which column read the weights and for the AC3 to know in which register it has to accumulate results of the SMAC. The exceptions of the first and second columns are due to the absence of padding.

3.2.7 FILTER

The introduction of the sparsity map compression method has the main consequence that a filtering system is necessary. This filter has to:

- verifies that the portion of the sparsity map that is analyzed has non-zero values
- filter the non-zero values
- corresponding non-zero values to the correct weights

All the blocks, that compose the filter block, will be described in the following subsections are subordinates to an FSM that manage the filtering.

1 COUNTER

The first block is a 1 counter. When a portion of the sparsity map is available this block verifies that there is at least one non-zero value that has to be filtered. if there isn't there is no need to filter so we can skip this portion and pass to the next. it is a basic combinational block. Out of this block, there is a simple register that stores the value

SHIFT REGISTER

In this shift register we load the portion of the sparsity map. if there is at least one non-zero value the filter starts to scroll the map one bit for time. the output of this shift register is the enable for:

- A counter that counts the number of non-zero values until they are equal to the output of the 1 counter. if they are equal the filtering terminates and we skip to the next portion of the sparsity map.
- A second counter that counts if we have filtered an M number of non-zero values through the next portion of the map.
- An external shift register that stores the correct weight corresponding to the position of the non-zero value.

COUNTER

There are there counters in the filter with three different scopes. the first and the second have been described in the previous section, the third instead is a counter that counts if we have scrolled the whole sparsity map portion loaded in the shift register.

Filter FSM

This FSM manages the filtering of the sparsity map. It has three main counters. the first count when we have filtered all the non-zero values so we can stop the filtering of this section of the sparsity map and pass it to the next. The second count is when we have filtered 16 non-zero values of the sparsity map and so the register is full and the smac can elaborate. The third count is simply when we have filtered the whole section of the sparsity map (every section is 16 bits).

Chapter 4

Verification

4.1 Verification

4.1.1 Memory

The first verification done is for the memory. In this case, we verify that the data are written correctly in the correct position of every memory block. Consider figure 4.1 that shows the filling of row 1 of the memory from the first block. In figure 4.2 is shown the end of the filling of the first row (third block) and the beginning of the second row. The figure 4.3 show at the end of the third block of the third row. Here is also verified the FSM that manages this memory filling. A second verification is done during the filter verification, here is verified that the wanted memory cell is read correctly.

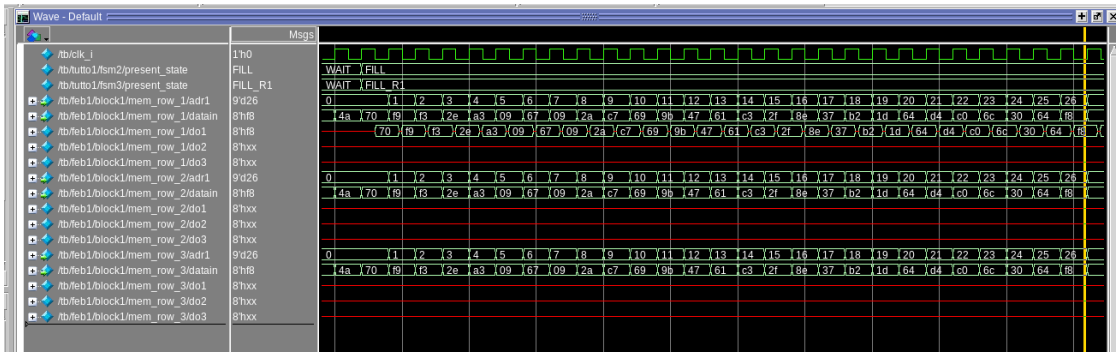

Figure 4.1: Memory fill verification

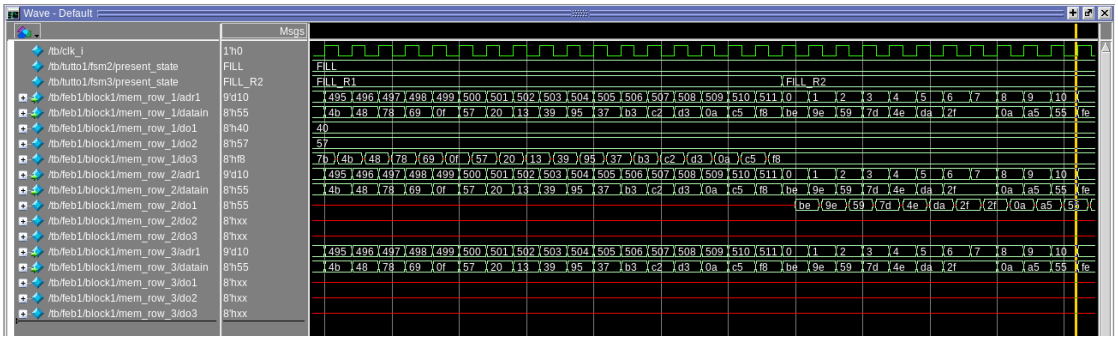

Figure 4.2: Memory fill verification

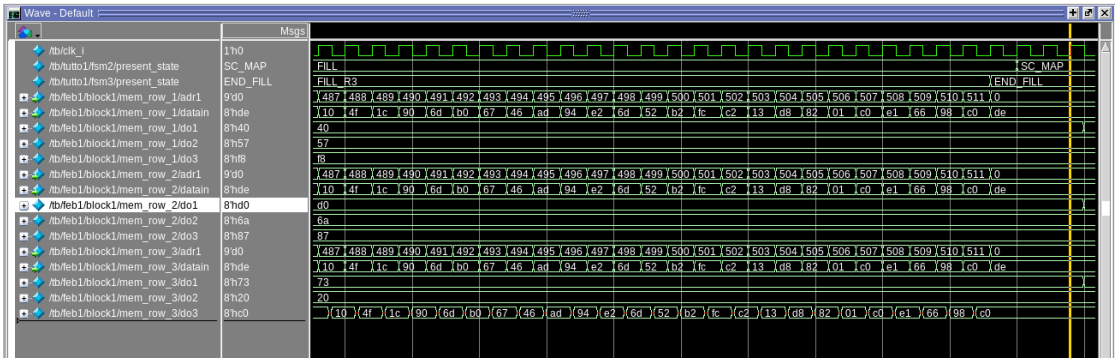

Figure 4.3: Memory fill verification

4.1.2 Filtering

1counter

First of all, for the filter system, we verify that the 1 counter analyzes correctly the portion of the sparsity map so that it counts the right number of 1. Mention the case of no non-zero values, this case has been used to verify also that the FSM that manages the filtering evolves correctly by skipping the filtering and waiting directly for the next map portion.

Counters

Once verified the one counter the next step is to verify the correct update of the counters and so the correct evolution of the FSM.

- the first counter is counter3, this counter counts how many bits of the portion of the sparsity map we have scrolled. We verified that it was 0 at the beginning of every map portion analyzed, then that it was updated in parallel to the shift

register. When it reaches the end count we also verify the correct evolution of the FSM.

- The second counter is the full counter. Here is verified that this count is updated every time the filter found a non-zero value. when it reaches the end count it means that we have filtered 16 non-zero values, so the filtering of this map portion is stopped to be later resumed.

- the last non-zero value of the map portion could not be in the last position, this register is updated as the previous when we have a non-zero value but is compared to the out of the 1 counter. when these two numbers are equal is possible to stop filtering this map portion and pass to the next.

Shift register for weights

A shift register that stores the weights is present. Here we verify that in the presence of a non-zero value of the map, it samples the correct weights that come out from the memory. Is possible to see in figure 4.4 that the shift register samples consecutively the correct weight in the presence of the map "1000000000011101". Figure 4.5 shows that it is cleared correctly when it is full and the smac weights register has sampled it. Figure 4.6 also shows what happens when the shift register is full but the SMAC hasn't finished the elaboration, the filtering waits until the end of elaboration to resume/start again.

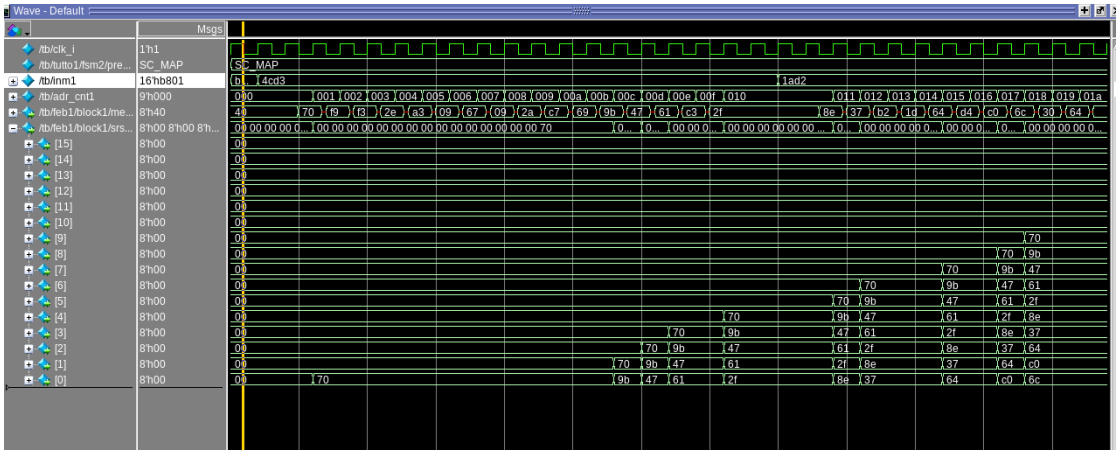

Figure 4.4

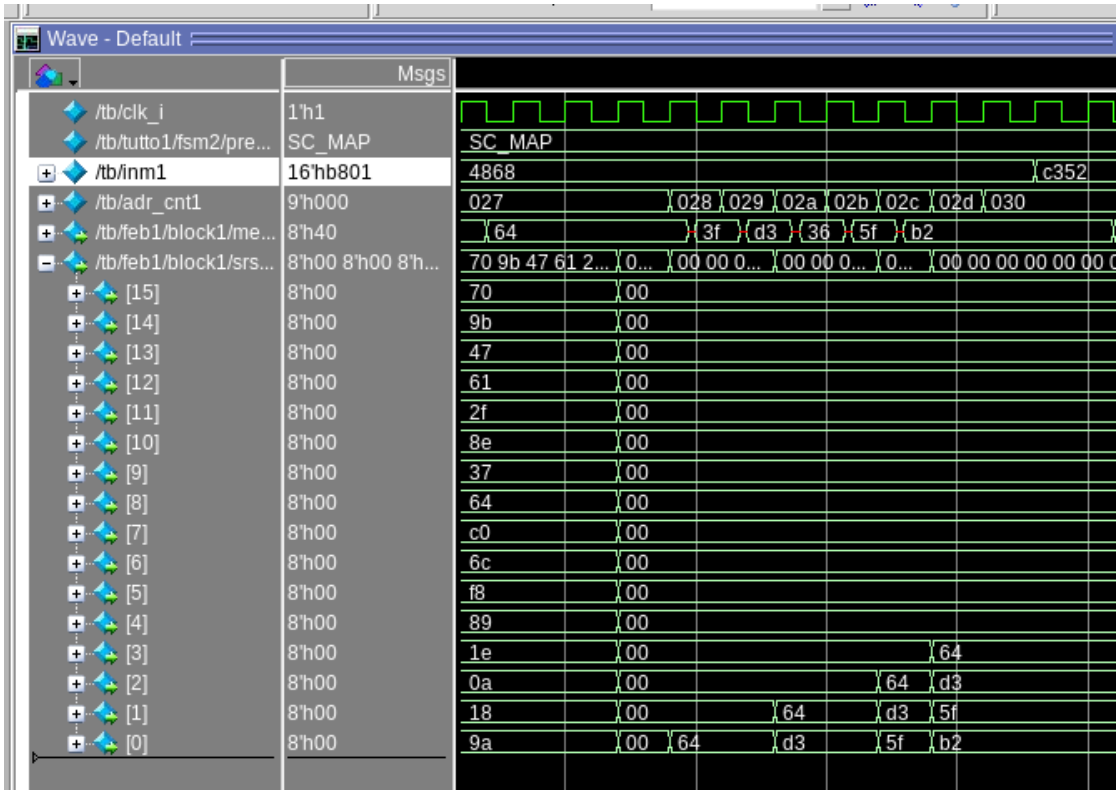

Figure 4.5: Register reset

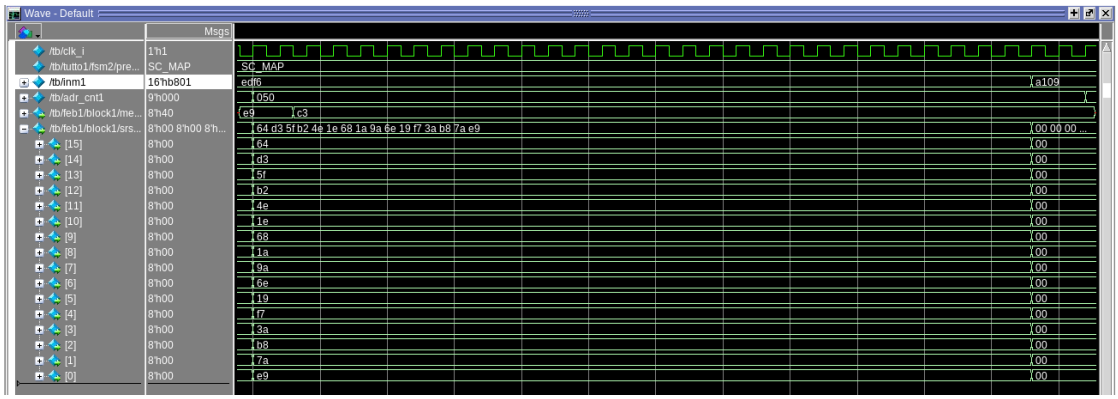

Figure 4.6: Wait to reset the register

4.1.3 Memory Block Address

Due to a combination of the position of the volume, column, and row in which we are, and of the kernel position (memory column) we have to verify first of all

that we read from the correct memory block. Then we verify the correctness of the address, in particular:

- The correct update of consecutive address
- stopping the address update when we are at the end of the full counter and the correct resuming
- the correct updated to the next correct starting address when the out of the found counter is equal to the 1 counter.

4.1.4 AC3

For the AC3 two verifications are necessary. the first verification is the correct accumulation in the correct register based on the volume position (row and column) and kernel signal. this is shown in the figure, is possible to see that in the presence of a determinate kernel and column signal when a smac end signal is asserted the register value is updated with the new accumulation between the old value and smac out. the second verification is the clearing of the correct register after the updates of the volume position. This is shown in the figure below, after that the data has been read correctly in output at column updates it is reset.

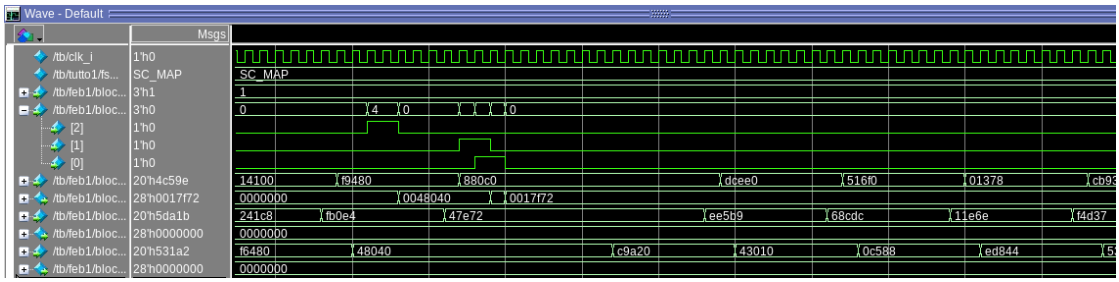

Figure 4.7: AC3 register update verification

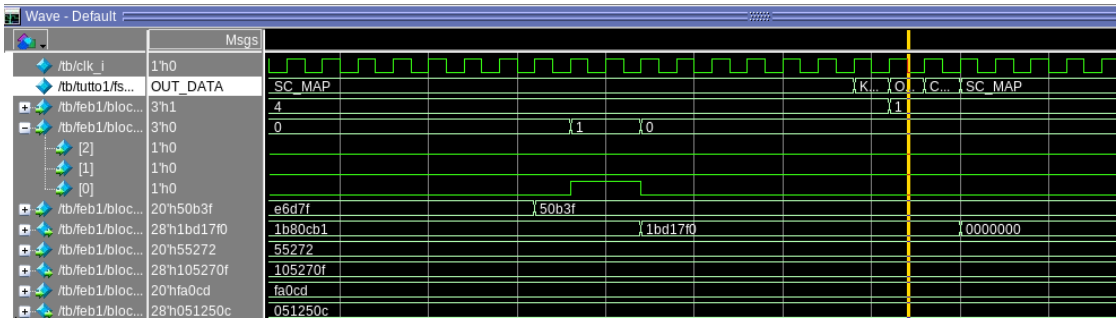

Figure 4.8: AC3 register reset verification

4.1.5 AC1 and AC2

Before talking about the verification of this block is important to clarify that two starting signals are necessary for the accumulation.

- the first signal comes from the filtering section, it tells that the input data (weights and activations) are ready to be elaborated
- the second signal comes from the AC3, it tells that the previous result of accumulation has been accumulated in the ac3 register and is possible to start with a new one.

In the verification of this block, we verify that it waits correctly for the two signals before starting an elaboration as shown in the figure below. We also verify the correct evolution of the FSM that manages the accumulation (also possible to see in the image).

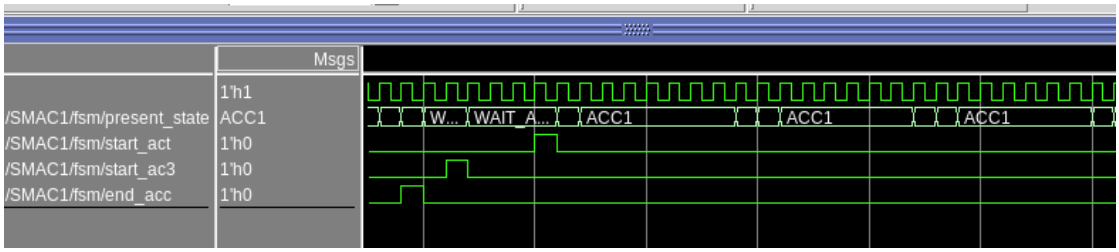

Figure 4.9: AC1 and AC2 verification

4.1.6 Synthesis

After all the verifications have been completed the next step is to extrapolate data from the synthesis analysis. To reach desired performances in terms of throughput the whole structure composed of the 6 SMAC, 2 AC3, and memory is replicated 16 times so there is a total of 96 SMAC, 32 AC3, and 144 memory blocks.

Performances

A crucial point is to analyze how the new engine structure impact the throughput in the presence of different sparsity percentage. As explained before the scope of sparsity management is to avoid useless operations this can virtually increase the throughput due to the consequence that skipping useless operations reduces the number of total cycles per layer. As benchmark has been taken the VGG16 layer 8. After some analysis of various sparsity percentages and various operand parallelism is possible to notice that there is a trend that is quite similar for different parallelism and shows an increase of the throughput for an increasing sparsity percentage.

Compared to the 0 sparsity the trend goes from a double value of the throughput for a 50 % sparsity to a triple value of the throughput for a 80 % sparsity. In figure 4.10 is possible to see the trend mentioned before for operands parallelism combined, in particular, weight can be of 4, 6, and 8 bits while activations can be of 4 and 8 bits. The number of operands for times is maintained at 16 while the sparsity increase by 10 % for times from a base of 50 % to a cap of 80 %.

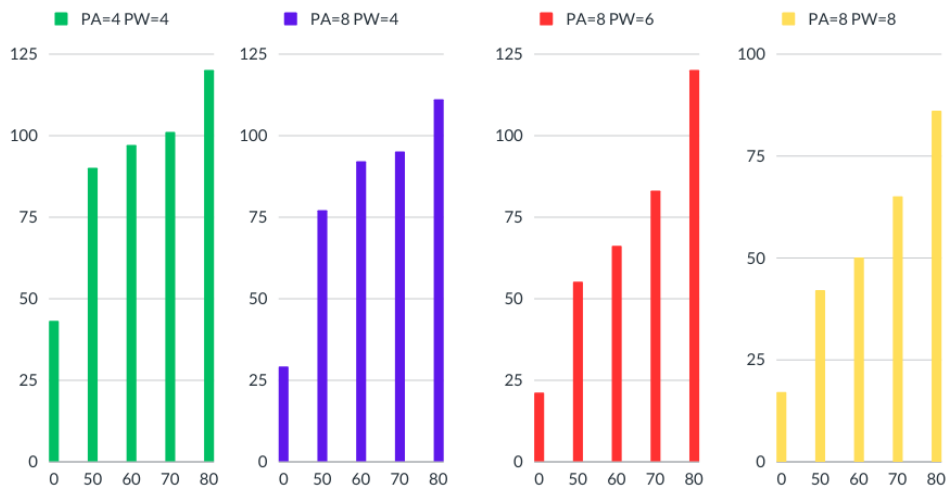

Figure 4.10: Throughput MAC/cycle for various sparsity percentage and various operand parallelism

Area

Firs data is the total area cells for 1.2 V and 1V synthesis. For 1.2V we have a total of 113700.93 μm^2 and for 1 V a total area of 114050.13 μm^2. These area values are for only 6 SMAC with related memory blocks and FSMs, for the whole structure with 96 SMAC the area is about 1,8 mm^2. Also considering more but smaller SMAC and shared AC3 there is an increase in the area. The great area overhead is mainly due to the presence of the 144 memory blocks, every block is composed of 512 cells of 8 bits each, the area of these blocks alone is about 1,2

mm^2 which is the 66 % of the whole system.

Chapter 5

Conclusions and Future Work

This thesis work consisted of the development of a system able to manage sparsity in a CNN. Among the various possible compression methods that help to manage sparsity, the choice is the sparsity map. The starting point is the already developed SMAC engine, this has been modified in the internal structure and a system able to filter the zero-activation based on the sparsity map has been added. With respect to the original engine, there is an increase in the total area due to the introduction of the local memory, which represents 66 % of the total area, but also an increase of the throughput of three times for the higher sparsity percentage. Main future upgrades concern the other existing compression methods, for example, the structure is flexible enough to be modified to work with CSR (Compressed Sparse Row), in this case, major changes regard the filtering system that should be thought to work with a different compression. About the CSR, since the CSC (Compressed Sparse Column) has the same compression idea but in the column way it can be developed starting from the solution for CSR and modify only the approach, can be interesting to see if a different approach to the convolutional volume can produce better results. Another improvement concern the clock frequency limited by the memory block.

Bibliography

[1] Wikipedia. *Neurone — Wikipedia, L'enciclopedia libera*. [Online; in data 4-luglio-2023]. 2023. URL: http://it.wikipedia.org/w/index.php?title=Neurone&oldid=134001976 (cit. on p. 3).

[2] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer. «Efficient Processing of Deep Neural Networks: A Tutorial and Survey». In: *CoRR* abs/1703.09039 (2017). arXiv: 1703.09039. URL: http://arxiv.org/abs/1703.09039 (cit. on pp. 3, 18–20).

[3] Maurizio Capra, Beatrice Bussolino, Alberto Marchisio, Guido Masera, Maurizio Martina, and Muhammad Shafique. «Hardware and Software Optimizations for Accelerating Deep Neural Networks: Survey of Current Trends, Challenges, and the Road Ahead». In: *CoRR* abs/2012.11233 (2020). arXiv: 2012.11233. URL: https://arxiv.org/abs/2012.11233 (cit. on pp. 4, 5, 9, 13–16).

[4] Y. Le Cun, I. Guyon, L. D. Jackel, D. Henderson, B. Boser, R. E. Howard, J. S. Denker, W. Hubbard, and H. P. Graf. «Handwritten Digit Recognition: Applications of Neural Network Chips and Automatic Learning». English (US). In: *IEEE Communications Society Magazine* 27.11 (Nov. 1989), pp. 41–46. ISSN: 0163-6804. DOI: 10.1109/35.41400 (cit. on p. 11).

[5] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. «ImageNet Classification with Deep Convolutional Neural Networks». In: *Advances in Neural Information Processing Systems*. Ed. by F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger. Vol. 25. Curran Associates, Inc., 2012. URL: https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf (cit. on p. 12).

[6] Karen Simonyan and Andrew Zisserman. «Very Deep Convolutional Networks for Large-Scale Image Recognition». In: *arXiv 1409.1556* (Sept. 2014) (cit. on p. 12).

[7] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. «Going Deeper with Convolutions». In: *CoRR* abs/1409.4842 (2014). arXiv: 1409.4842. URL: http://arxiv.org/abs/1409.4842 (cit. on pp. 12, 13).

[8] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. «Deep Residual Learning for Image Recognition». In: *CoRR* abs/1512.03385 (2015). arXiv: 1512.03385. URL: http://arxiv.org/abs/1512.03385 (cit. on pp. 13, 14).

[9] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q. Weinberger. «Densely Connected Convolutional Networks». In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2017, pp. 2261–2269. DOI: 10.1109/CVPR.2017.243 (cit. on p. 14).

[10] Jie Hu, Li Shen, and Gang Sun. «Squeeze-and-Excitation Networks». In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2018 (cit. on pp. 14, 15).

[11] Sara Sabour, Nicholas Frosst, and Geoffrey E Hinton. «Dynamic Routing Between Capsules». In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett. Vol. 30. Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper_files/paper/2017/file/2cad8fa47bbef282badbb8de5374b894-Paper.pdf (cit. on pp. 15, 16).

[12] Andrew Lavin. «Fast Algorithms for Convolutional Neural Networks». In: *CoRR* abs/1509.09308 (2015). arXiv: 1509.09308. URL: http://arxiv.org/abs/1509.09308 (cit. on p. 18).

[13] Francesco Conti and Luca Benini. «A Ultra-Low-Energy Convolution Engine for Fast Brain-Inspired Vision in Multicore Clusters». In: vol. 2015. Mar. 2015. DOI: 10.7873/DATE.2015.0404 (cit. on p. 19).

[14] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. «ShiDianNao: shifting vision processing closer to the sensor». In: June 2015. DOI: 10.1145/2749469.2750389 (cit. on p. 20).

[15] Francesco Conti, Pasquale Davide Schiavone, and Luca Benini. «XNOR Neural Engine: a Hardware Accelerator IP for 21.6 fJ/op Binary Neural Network Inference». In: *CoRR* abs/1807.03010 (2018). arXiv: 1807.03010. URL: http://arxiv.org/abs/1807.03010 (cit. on pp. 20, 22).

[16] Yu-Hsin Chen, Tushar Krishna, Joel S. Emer, and Vivienne Sze. «Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks». In: *IEEE Journal of Solid-State Circuits* 52.1 (2017), pp. 127–138. DOI: 10.1109/JSSC.2016.2616357 (cit. on p. 21).

[17] Francesco Conti et al. «An IoT Endpoint System-on-Chip for Secure and Energy-Efficient Near-Sensor Analytics». In: *IEEE Transactions on Circuits and Systems I: Regular Papers* PP (Dec. 2016). DOI: 10.1109/TCSI.2017.2698019 (cit. on p. 22).

[18] Petruzzellis M. «Design of a Flexible Hardware Accelerator for Ultra-Low Power Quantized Neural Networks based on Serial Multipliers». Tesi di laurea. Politecnico di torino, 2019 (cit. on pp. 23, 26, 28, 29).

[19] Yunji Chen et al. «DaDianNao: A Machine-Learning Supercomputer». In: *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-47. Cambridge, United Kingdom: IEEE Computer Society, 2014, pp. 609–622. ISBN: 9781479969982. DOI: 10.1109/MICRO.2014.58. URL: https://doi.org/10.1109/MICRO.2014.58 (cit. on p. 25).

[20] Sayeh Sharify, Alberto Lascorz, Patrick Judd, and Andreas Moshovos. «Loom: Exploiting Weight and Activation Precisions to Accelerate Convolutional Neural Networks». In: (June 2017) (cit. on p. 25).