POLITECNICO DI TORINO

Master Degree in Electronic Engineering

Master Degree Thesis

Design of a Flexible Hardware Accelerator for Ultra-Low Power Quantized Neural Networks based on Serial Multipliers

Supervisors
prof. Maurizio Martina
prof. Guido Masera
dr. Francesco Conti (ETH Zürich)

Candidate
Mattia Carlo Petruzzellis

April, 2019

Acknowledgements

First of all, I would like to thank prof. Maurizio Martina and doc. Francesco Conti. Your patience in listening to every little concern I had and the experience and guidance you provided have been invaluable components in making this thesis work come to life.

Thanks to Maurizio Capra and Riccardo Peloso for both the technical and moral support you showed during this thesis journey. Thank you for all the insights you gave me and for being there for even the silliest doubt (and the silliest jokes).

Thanks to all the people in the VLSI lab: Luca, Umberto, Yuri, Gianni, Simone, Giulia, Fabio, Rossana and again Maurizio and Riccardo. You guys are truly amazing and your willingness to help the others are second to none. I believe there is a bit of you all in this thesis and for that I am grateful.

Thanks to Valeria, Luciana, Giancarlo and my brother Francesco for being my irreplaceable life coaches. You truly showed me how every cloud has its silver lining, that it is never to late to follow our passions and it takes bravery to make some tough life-changing decisions.

Thanks to $C_{257}H_{383}N_{65}O_{77}S_6$. I know we have our up and downs, but I believe you always act in my best interest. Thanks to the medical research, doc. Cristina Matteoda and the team of nurses of the ASL of Turin. Hiccups happen in life, but being surrounded by such experienced and helpful professionals made it feel as if nothing really happened.

A huge thank you to my family. The patience, support and sacrifices you have made during all these years can not be described with just words. You have always been there and you will always be, like a lighthouse to the ships at sea, a pillar always pointing to my own safe place.

Last but not least, just thank you to all my friends for always being there and for the time we have spent together. There is no need to name any names, you know if you are in this list!

Summary

Today, our society is experiencing a new revolution, almost comparable to the discovery of electricity and its application to industry, healthcare, communication and everyday life. This revolution goes under the name of Artificial Intelligence (AI) and it may come as a surprise the amount of times we use it without even knowing it. In particular, the part of AI that is increasingly gaining more and more attention is Deep Learning (DL), whose main idea is to use a Deep Neural Network (DNN) in order to let a machine learn through training and perform, through inference, several tasks, as if these were performed by a human being. Among these tasks we find autonomous driving, speech recognition, computer vision and many others.

The reason why DL is taking off compared to other well known and documented Machine Learning (ML) algorithms is due to its capability to take advantage of huge amount of data. Indeed, whereas the latter have no considerable performance boost when increasing the available data over a certain threshold, the first can get huge benefits out of it the larger is the designed Neural Network (NN). Moreover, the digitization of our society and its transferring many human activities to the digital realm, created a mechanism on which DNN could thrive and get better and better, without the hustle of looking for data somewhere else.

While training is usually a latency-insensitive process performed on big server machines, inference is extremely latency-sensitive. To perform inference, a rather naive and energy hungry approach requires gathering data on the end nodes and send it all to a big server so that it can be consumed. However, this is not really the best scenario for the development of future smart sensor nodes for edge computing. The idea behind edge computing is to move the analytic part from data centers closer to sensors in order to tackle the power limitations that come with the amount of data that needs to be sent and the available bandwidth. By doing so, through inference one is able to extract the useful information on the spot and send only that to the data center, which is a way less expensive solution than sending it all.

Depending on the specific application for which a DNN is developed, one may be pushed towards the adoption of different hardware platforms and architectures. The main operation that a DNN is required to perform is a multiply-and-accumulate (MAC) and being DL such a computational hungry solution, graphical processing units (GPUs) are often used, especially for training purposes, due to their highly

parallel computational capabilities as well as their superior accuracy. However, GPUs and even Filed Programmable Gate Arrays (FPGAs) are not particularly good for ultra-low power applications, as they deliver too little performance per Watt of power, which can only be achieved by hardware acceleration in application specific integrated circuits (ASICs) or special-purpose accelerators in highly integrated Systems-on-Chip (SoCs).

This thesis focuses on the development of an Ultra-Low Power strongly quantized Convolutional Neural Network (CNN) hardware accelerator, potentially applicable to Internet of Things (IoT) end-nodes for near-sensor analytics and designed to be integrated in the Parallel Ultra Low Power (PULP) platform developed by ETH Zurich and the University of Bologna as a Hardware Processing Engine (HWPE).

Using quantized data allows both for an increase in computational speed, due to simpler operations to be performed which enable a better exploitation of algorithmic parallelism, and a reduction in memory usage, which leads to a non-negligible power saving. Indeed, for these kind of applications, fetching data from memory has a higher cost than the actual computation. The developed Serial-MAC-Engine (SMAC-Engine) works on a 8 bits parallelism for the activations and on a 4 bits parallelism for the weights, as this is considered the state-of-the-art for ultra-low-power inference, and performs multiplications serially to be able to achieve a higher maximum operating frequency (416 MHz) compared to the parallel counterpart (250 MHz) while keeping the throughput, i.e. the number of operations per unit cycle, unchanged at approximately 12.69 GMAC/s when standalone and 7.81 GMAC/s once integrated in the PULP platform. This, however, comes at the cost of an increase in area (347673.2 μm^2) of about $\times 2.91$ compared to the parallel solution. All this provided a rather flexible solution, consuming 1.34 pJ/MAC @ 0.9 V and 25 °C as will be discussed further in the following pages.

Contents

Chapter 1

Introduction

1.1 General principles

In recent years, there was a realization that the only way to let machines do more interesting things, such as autonomous driving, object detection, speech recognition and many other once "human-only" related tasks, was to let them learn by themselves. This is why Machine Learning (ML), one of the fields of Artificial Intelligence (AI), has been gaining an increasing attention. Specifically, nowadays an even narrower area of ML, namely Deep Learning (DL), is attracting a very large number of researchers and practitioners due to its extremely wide range of applications, from industry to advertising, healthcare and many others.

DL is based on the development and adoption of Deep Neural Networks (DNNs). DNNs are brain inspired structures [1], meaning that rather than trying to create a brain, they emulate some of its aspects based on how scientists think it works. In particular, similarly to how a neuron is considered to be the elementary computational element of our brain, which is composed of billions of them, a DNN consists of several layers (the higher the number the deeper the network), each containing some neurons contributing to the computation of the output result. During the training process, a 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 possible DNN solutions, some that were specifically developed for computer vision applications but that are actually used in other areas as well, thus becoming by far the most popular model, are Convolutional Neural Networks (CNNs). The number of complex tasks these structures are able to perform is outstanding. Since the day AlexNet won the ImageNet Large Scale Visual Recognition Competition (ILSVRC) [2], DNNs scored some once unbelievable results in several

areas, increasing their accuracy by adopting different solutions. To name a few, inception modules in the Inception Network [3] have been introduced to approximate a sparse CNN with a normal dense construction having a smaller size to reduce the computation requirements of about one order of magnitude while capturing details at various scales depending on the chosen kernel size. Moreover, using skip connections, in Residual Networks (ResNets) [4] proved to be an effective way to tackle the vanishing and exploding gradients problem when going deeper with the number of layers. In 2015, the latter structure provided a solution able to even overcome human accuracy in an image classification task.

Another consistent boost in the diffusion of DNN came from the adoption of transfer learning: for instance, if one is building a computer vision application, rather than training the weights from scratch with random initialization, a much faster progress can be obtained by downloading weights that someone else has already trained on the network architecture, use these as pre-training and transfer that to the new task of interest. The computer vision research community has been posting lots of data sets on the Internet, like ImageNet, MSCOCO or Pascal on which computer researchers have trained their algorithms on. Sometimes these training take several weeks and many Graphics Processing Units (GPUs) and the fact that someone else has done this and gone through the painful hyperparameter search process, means that one can often download open source weights that took someone else many weeks or months to figure out and use them as a very good initialization for his own neural network, ultimately speeding up the training process.

Finally, the development of several open source frameworks tailored for DNN applications, such as Tensoflow, Caffe, PyTorch, Keras and many others, further enlarged the accessibility to newcomers thus broadening their evolution and diffusion even more.

The previously mentioned reasons together with the deepening of neural networks allowed to score better and better results at the cost, however, of an increase in computational complexity. Whilst training usually can't help but require the adoption of hugely parallel, higher precision and power consuming systems, mainly GPUs, to speed up the training process, such constraint is not necessarily true for inference or at least it is not the primary concern. This is why there is a growing interest in moving the latter towards specialized hardware solutions, such as ASIC, FPGA or complex SoCs. In particular, even though FPGAs allow for a greater flexibility thanks to their intrinsic post-fabrication programmability, their power consumption is still too high compared to what higher integration solutions are able to achieve. This is why the other two solutions are the only way to go when energy is the most important resource to preserve. For instance, with their open-source Parallel Ultra Low Power (PULP) platform, ETH Zurich and the University of Bologna proposed a paradigm where an ultra-low power multicore SoC can be augmented by specialized accelerator units, called Hardware Processing Engines

(HWPEs) to greatly accelerate specific functions such as the DNN inner kernels.

The reason why power consumption is so important is that it limits the diffusion of this technology to new areas such as Internet of Things (IoT) end-node sensor analytics [5, 6, 7]. In particular, many applications are nowadays based on data center computing, mainly due to the computational requirements of dealing with billions of operations. However, this solution is detrimental not only due to the huge power consumption but also because of the high latency introduced by the transmission requirements as well as the limited bandwidth intrinsic to every radio communication system. Being able to overcome such issues may potentially unleash several new solutions: the idea is to move the computation "at the edge", at the sensor level, so that rather than having to send an entire stream of data to a remote data center, one is able to perform some preliminary computation in loco, ultimately having to send simply the extracted features and thus saving a lot of power.

Still, meeting the power envelope required by such solutions is non-trivial. Even though the basic operation of every DNN is a relatively simple Multiply And Accumulate (MAC), the number of times it has to be performed and especially the number of required memory accesses to fetch the needed weights and activations is massive, so any technique capable of reducing the energy that is spent is highly welcome. This is why intelligent data flows have been thought to maximize the data reuse before a new fetching is required. As a matter of fact, a big portion of the energy consumption comes from the system interaction with memory where data needs to be read from or written to. Moreover, reducing the numerical precision of both activations and weights through quantization, at the cost of some acceptable loss in terms of accuracy, has proven to be optimal not only thanks to the obtainable increase in throughput but also due to the reduced memory occupation which allows for a greater amount of data to be read from or written to memory in a single cycle.

These are the outlines on which the inference hardware accelerator for CNN discussed in this thesis, Serial-MAC-Engine (SMAC-Engine) has been developed. In particular, this was computationally inspired by the serial approach used by Loom (LM) [8], where parallel multiplications have been substituted by serial ones whilst keeping the throughput unchanged, or potentially increased, at the price of an increase in area occupation. For the data flow instead, this was inspired by an output stationary solution similar to the one used in XNOR Engine (XNE) [7]. Finally, the developed Data Path (DP) has been integrated in PULP as HWPE and its final area, frequency and power consumption derived to test what its performance would be on a real system.

1.2 In the following chapters

The chapters in this thesis work will be organized as follows:

- Neural Networks and Deep Learning. In this chapter, a historical and mathematical background on NNs will be given to explain what they're inspired on and how they work for DNNs and especially CNNs applications, trying to point out in what the latter differ from standard DNN structures.
- State of the art hardware solutions for accelerating CNNs, with a focus on ASIC hardware accelerator solutions that inspired SMAC-Engine. Here, some context regarding the current state of the art will be given explaining what are the strength and weaknesses of the available inference hardware solutions derived through the exploration period.
- SMAC-Engine starting hypothesis, Data Path and low-level control explanation. In this chapter, the methodology followed to derive the Data Path and low-level control will be discussed in details, from the starting hypothesis to the final architecture.
- Integration on Zurich ETH's HWPE. Here, the procedure followed to integrate the derived architecture on a HWPE of a PULP platform will be described.
- Analysis and discussion on the obtained results. In this chapter, after a description regarding how the employed software tools have been set up, the results obtained by performing the simulation, synthesis and the place and route will be discussed, focusing on the adopted technology, area, power and throughput compared to other known implementations.
- Conclusion and future development. In this final chapter, a summary on the work will be provided together with some possible future developments that may stem from this work.

Chapter 2

Neural Networks and Deep Learning

In the following sections, some mathematical concepts concerning neural networks will be introduced trying to explain how they turned out to be the state of the art technique for many machine learning (ML) and deep learning (DL) problems. Then, an overview on convolutional neural networks (CNNs) will be given trying to depict the differences and similarities compared to the usual deep neural network structures [1].

2.1 Why Neural Networks

There are two possible formal definitions of ML. The first, by Arthur Samuel (1959), defines ML as: "the field of study that gives computers the ability to learn without being explicitly programmed". The second instead, by Tom Mitchell (1998), defines ML as a well-posed learning problem: "a computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E".

Learning algorithms may fall within two big families: supervised and unsupervised learning. In supervised learning, the idea is to teach a machine to do something provided with a labeled set of data, whereas in unsupervised learning, one lets the machine learn by itself.

Focusing on supervised learning problems, being given a labeled set of data means that each data comes with one or a set of features based on which a learning algorithm will try to predict the correct output. In this scenario, NNs were able to

[1] Most of the material exposed in this chapter comes from notes taken from Coursera video lessons on Machine Learning and Deep learning [9, 10].

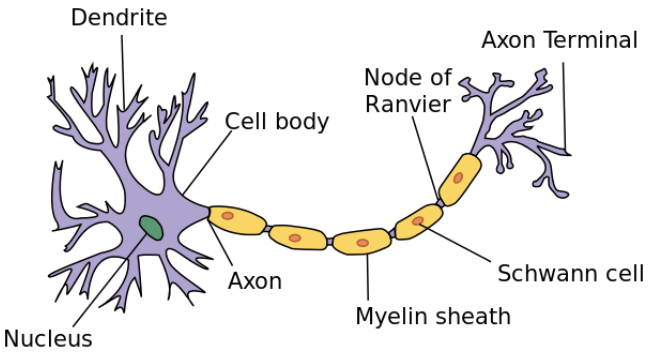


Figure 2.1. Image of a single neuron [11].

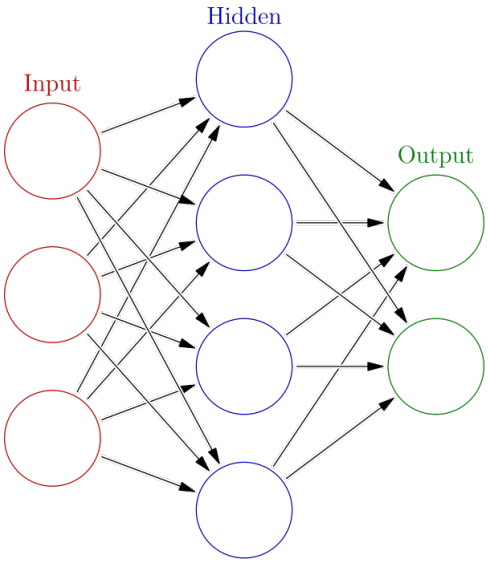


Figure 2.2. Scheme of a neural network [12].

take over other ML solutions such as linear or logistic regression because of their outstanding capability to deal with multiple features rather than just one or two. NNs are quite well suited for learning complex non-linear hypothesis compared to the logistic regression counterpart.

The origins of Neural Networks was to develop algorithms that try to mimic our brain, which is perhaps the most amazing learning machine we know about. Their history dates back to the '60s (perceptron), however, due to their intensive computational requirements, they have been only recently rediscovered.

Neural networks were developed as simulating neurons or networks of neurons in the brain, like the one depicted in figure 2.1. In particular, a neuron can be seen

as a computational unit that gets a number of inputs through its input wires, the dendrites, does some computation and then it sends output via its axon to other nodes or to other neurons in the brain. A possible scheme of a NN is shown in figure 2.2.

Here, we can distinguish the presence of some input features (in red), e.g. x_1, x_2, x_3 stacked up vertically to form the input layer of the neural network. Then, there is another layer of circles (in green), called hidden layer of the neural network and finally, the last layer forms the output layer, as it is responsible for generating the predictions. The term hidden layer refers to the fact that in the training set, the true values for the nodes in the middle are not observed, that is, one can't see what they should be in the training set.

To introduce some notation, one could either use the vector x to denote the input features, or, alternatively, $a^{[0]}$, where the term a stands for activations. Activations are the values that different layers of the neural network are passing on to the subsequent layers. Therefore, $a^{[1]}$ will be the vector of activations for the next layer and, in this case, $\hat{y} = a^{[2]}$ will be the output activations containing the predicted values for this specific network. Finally, both hidden and output layers will have some parameters associated with them, namely weights W and biases b that are necessary for the computation every single neuron is required to perform.

2.2 Mathematical basics on Neural Networks: inference

If one were to focus on the computation required by a single neuron, say the j-th, this is usually of the form:

$$a_j = f\left(\sum_i W_{ij} x_i + b\right) \tag{2.1}$$

where x are the input activations, W are the weights, b is the bias, a_j is the output activation and $f(\cdot)$ is a non-linear function also known as activation function. This computation is repeated for every neuron in every layer and this is why it is also possible to vectorize the above equation, for the l-th layer, as in 2.2.

$$a^{[l]} = f\left(z^{[l]}\right) = f(W^{T[l]} a^{[l-1]} + b^{[l]}) \tag{2.2}$$

The process of computing the final output of a NN is also known as forward propagation because it starts with some activations $a^{[0]}$ at the input neurons and then it forward propagates them through the hidden layers, compute again the activations of the hidden layers and then finally forward propagate them to compute the activations of the output layer to obtain the prediction $\hat{y} = a^{[L]}$, begin L the number of layers of the designed NN.

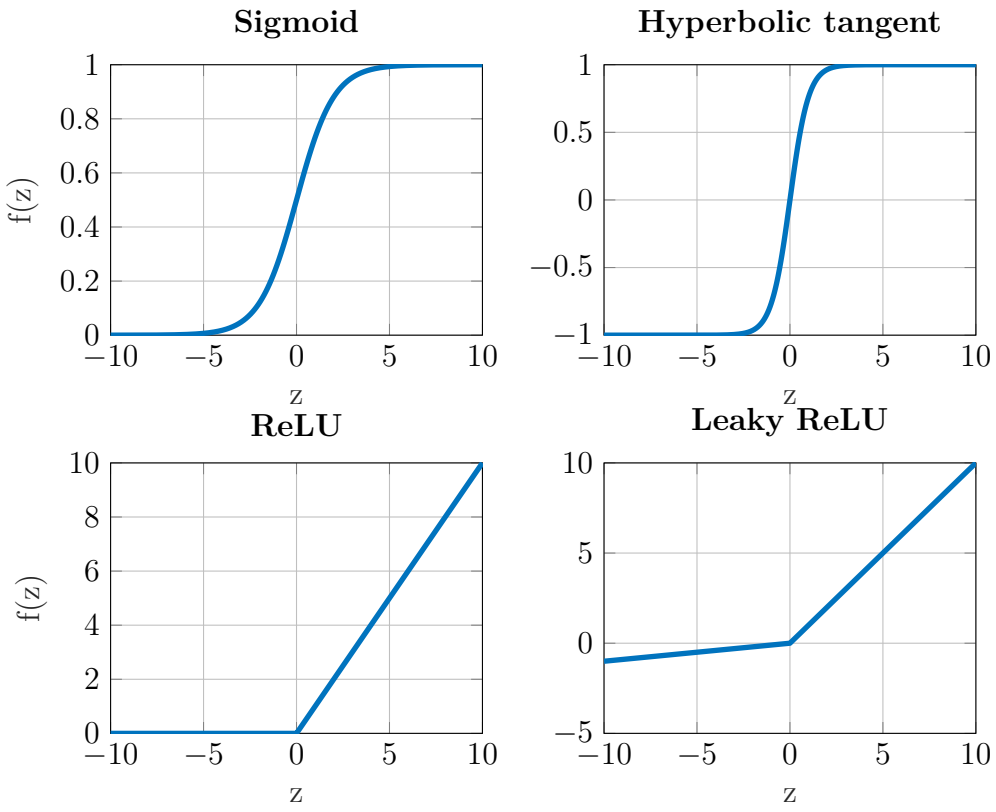


Figure 2.3. Possible activation functions.

When building a neural network, one of the choices we get to make is what activation function $f(z)$ to use in the hidden layers. Among the possible choices, we find the following:

- Sigmoid function:
$$f(z) = \sigma(z) = \frac{1}{1+e^{-z}} \tag{2.3}$$
- Hyperbolic tangent function:
$$f(z) = \tanh(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}} \tag{2.4}$$
- Rectified Linear Unit (ReLU):
$$f(z) = \max(0, z) \tag{2.5}$$
- Leaky ReLU:
$$f(z) = \max(0.1z, z) \tag{2.6}$$

There is actually not one that always works better than the others and the activation functions can actually be different for different layers. However, one of the downsides of both the sigmoid function and the hyperbolic tangent function is that if z is either very large or very small, then the gradient of this function becomes very small. This can ultimately slow down gradient descent and back propagation, that are the most widely used algorithms during the training process to derive the parameters that work best for the designed network. This is why in recent years ReLU has been the choice for many neural network designs due to its rather simple implementation as well as its faster learning rate during training compared to the sigmoid or hyperbolic tangent.

The importance of using non-linear activation functions comes from the fact that if one were to use linear activation functions, or identity activation functions, then the neural network would simply output a linear function of the input, being the composition of linear function still a linear function itself. If this was the case, then no matter how many layers a neural network has, what it would always be doing is just computing a linear activation function making hidden layers useless. Indeed, unless one adopts a non-linear function, no interesting functions would be computed, even going deeper in the network, thus invalidating the training process.

2.2.1 Softmax regression

A generalization of the binary classification with logistic regression is the Softmax regression, which makes the user perform a prediction over C multiple classes rather than just two. This means that the number of units (neurons) in the output layer will be exactly equal to the number of classes over which the prediction is performed and therefore the prediction $\hat{y}$ will no longer be a number but rather a $C \times 1$ vector whose elements are probabilities that will sum up to one. To do so, a Softmax layer using a Softmax activation function has to be used. First, the linear part for the output layer L is computed:

$$z^{[L]} = W^{[L]} a^{[L-1]} + b^{[L]}$$

where $z^{[L]}$ will be a $C \times 1$ vector. Then, the activation function requires to fist compute a temporary variable:

$$t = e^{z^{[L]}} \tag{2.7}$$

with an element-wise operation that will return another $C \times 1$ vector, and then the actual activation:

$$a^{[L]} = \frac{e^{z^{[L]}}}{\sum_{i=1}^{C} t_i} \tag{2.8}$$

which is basically the vector t but normalized to sum up to one. So, the i-th element of the vector $a^{[L]}$ is going to be:

$$a_i^{[L]} = \frac{e^{z_i^{[L]}}}{\sum_{i=1}^{C} t_i} \tag{2.9}$$

The peculiar thing regarding this activation function is that, differently to the one used for binary classification problems, it takes as input a $C \times 1$ vector and outputs again a $C \times 1$ vector thus generalizing what the logistic activation function would do. Here, the prediction that is most likely to be true will be the output with the maximum value out of the C possible ones. This is extremely useful, for instance when dealing with computer vision classification tasks.

2.3 Training a neural network

The technique used by a NN to learn how to optimally perform a task is adopting an algorithm known as gradient descent. First of all, the loss function, or error function, must be defined. This measures how well the algorithm is doing on a single training example. In logistic regression problems, when dealing with binary classifications tasks, this function has the following form:

$$\mathcal{L}(\hat{y}, y) = -(y \log \hat{y} + (1 - y) \log(1 - \hat{y})) \tag{2.10}$$

where $\hat{y}$ is the prediction provided by the NN through the forward propagation process while y is the "ground truth" that comes from the labeled data. Now, to know how well the network is doing compared to an entire training set, the cost function must be defined. Hence, if the training set is composed of m training examples and the neural network has L layers, then, for a binary classification problem:

$$\begin{aligned} J(W^{[1]}, b^{[1]}, \ldots, W^{[L]}, b^{[L]}) &= \frac{1}{m} \sum_{i}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) = \\ &= -\frac{1}{m} \sum_{i}^{m} \left[y^{(i)} \log \hat{y}^{(i)} + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)}) \right] \end{aligned} \tag{2.11}$$

where $\hat{y} = a^{[L]}$. Therefore, through gradient descent one tries to find $W^{[l]}$ and $b^{[l]}$, with $l = 1, \ldots, L$ that minimize the cost function J in order to get the predictions as close as possible to the ground truth. In particular, gradient descent is an iterative algorithm that starts from an initial point, that can usually be randomly picked, and for every step it takes, it tries to move towards the steepest downhill direction until eventually converging to a global optimum minimizing J. For every iteration, weights and biases will be updated following the expressions in 2.12:

$$W^{[l]} := W^{[l]} - \alpha \frac{\partial J}{\partial W^{[l]}} \tag{2.12}$$

$$b^{[l]} := b^{[l]} - \alpha \frac{\partial J}{\partial b^{[l]}} \tag{2.13}$$

with $l = 1, \ldots, L$, where α is the learning rate controlling the size of the step taken by each gradient descent iteration and one of the hyperparameters that needs to be properly tuned to obtain optimal results. The derivative terms, instead, represent the update or the change to be applied to the parameters.

As previously introduced, computations of a neural network are organized in terms of forward propagation steps, in which we compute the output predictions of the neural network. After forward propagation is performed, a way to compute the gradients in 2.12 needs to be introduced and this is what the backpropagation algorithm does: it uses the chain rule derived from calculus and operates by passing the values from the output backwards in a fashion similar to the one used during forward propagation [2].

When using gradient descent to train a neural network, it is actually important to randomly initialize the weights rather than initializing everything to 0 to solve the symmetry breaking problem. Indeed, through a proof by induction, it is possible to show that initializing weights with zeros leads to hidden units being symmetrical, meaning they compute exactly the same function for every iteration, which is definitely not helpful.

2.3.1 Training a softmax

In softmax classification, the loss function will be the following:

$$\mathcal{L}(\hat{y}, y) = -\sum_{j=1}^{C} y_j \log \hat{y}_j \tag{2.14}$$

Since the quantities in $a^{[L]}$ are probabilities, they can never be bigger than one and so, what this loss function does is it looks at whatever is the ground truth class in the training set and it tries to make the corresponding probability of that class as high as possible. Then, the cost function will be:

$$J(W^{[1]}, b^{[1]}, \ldots, W^{[L]}, b^{[L]}) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) \tag{2.15}$$

and gradient descent can again be used to minimize it. In particular, for the back propagation step it will be important to perform the following initialization:

$$dz^{[L]} = \hat{y} - y \tag{2.16}$$

[2] This explains how sometimes techniques that are efficient for inference also perform well on training [1]. However, one has to keep in mind that, differently from forward propagation, backpropagation needs to keep track of the intermediate values thus increasing the storage requirements. Moreover, the precision in the computed gradients generally needs to be higher compared to the one used for inference. This is why, hardware accelerators working on reduced precision like the one proposed by this work would not be suitable for training.

where $dz^{[L]}$ is an easier notation to express the partial derivative of the cost function with respect to $z^{[L]}$ and, as for $z^{[L]}$, it will be a $C \times 1$ vector. After computing $dz^{[L]}$ it is possible to start the backpropagation process and compute all the derivatives through the neural network. However, an important thing to keep in mind is that, when dealing with deep learning, there actually are several deep learning programming frameworks out there where the user simply has to focus on properly set the forward propagation process and then it will be the framework itself to implement the backpropagation process for the user, which hugely simplifies the process. Indeed, frameworks like Tensorflow, Keras, Caffe, PyThorch and others are nowadays largely used thanks to their ease of programming, efficient running speeds and thanks to them being open source and frequently updated and maintained.

2.4 Why switching to Deep Neural Networks

Over the last several years, the machine learning community has realized that there are functions that very deep neural networks can learn that shallower models are often unable to. In particular, since for any given problem it might be hard to predict in advance exactly how deep a neural network should be, the number of hidden layers can be seen as another hyperparameter to be tuned to get optimal results.

The reason why having deeper rather than simply big networks is important is because moving deeper through the network, the network itself is able to extract more and more complex features during inference. In other words, DNNs with multiple hidden layers might be able to have the earlier layers learn lower level simple features and then have the later deeper layers put together the simpler things they detected in order to detect more complex things. Moreover, whereas the earlier layers are computing what seems to be relatively simple functions of the input, such as where are the edges of an input image or simple phonemes out of a speech, by the time one gets deep in the network it can actually do surprisingly complex things, such as detect faces or detect words or phrases or sentences.

Finally, it is important to underline how dealing with DNNs means dealing with many possible hyperparameters compared to the earlier shallow solutions. Some of these hyperparameters are:

- 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.

All of these are parameters to give to the learning algorithm and that will control the ultimate parameters $W^{[l]}$ and $b^{[l]}$ and this is why they are called hyperparameters.

Hence, applying deep learning is a very empirical and iterative process, where one may often have an idea, implement it and try it out to see how that works. Then, based on the outcome, decide what to do next. The range of applications for DL today is so wide, ranging from computer vision to natural language processing, online advertising and many more, that it is not said that what works best for a solution can automatically be transferred to a new specific application. Moreover, it is also possible that the best hyperparameters values that are working today may change after a while due to a change in the hardware infrastructure like the used CPUs or GPUs or in the available data. Hence, it is also crucial not to stick with using always the same hyperparameters but systemically explore the space of hyperparameters every now and then to double check whether the used ones are still best or they require some further tuning.

2.5 Boosting training

Traditionally, the correct way to implement a training algorithm is to first divide the available data into three sets, namely the training set, development set (or hold-out cross validation set) and the test set. However, while before the amount of available data was not high enough and required to split it like 60% to the training set, 20% to the development set and 20% to the test set, in the modern big data era, where we might have millions of examples in total, the trend is that the development and test sets have become a much smaller percentage of the total, even down to 1% each.

Now, the workflow is to keep training algorithms on the training sets, use the development set to see which of many different models performs best on the development set and finally, after having done this long enough, take the best found model and evaluate it on the test set. All this in order to get an unbiased and confident estimate of how well the algorithm is doing.

Another reason why dividing the examples into three sets is important is because of the bias-variance trade-off. In fact, having high bias may lead to data underfitting while having high variance may lead to overfitting the data. In the first case, one would have an algorithm that is not performing well enough on the given data set while in the second case, one would have an algorithm that is performing too well on the given data set but that could have some problems once new data is added to the set. This means it is crucial to find a good balance in order to avoid both

of these extreme cases. Actually, there are some techniques that can be adopted to tackle these phenomena. For instance, using regularization, dropout and data augmentation to avoid overfitting.

To speed up the training process, normalization is a very useful technique since it allows to change the shape of the cost function J and make it much more symmetric in order to make it easier to run gradient descent on it to let it find a minimum to converge at. Normalizing data requires to compute the mean μ and standard deviation σ and then update it as:

$$x := \frac{x - \mu}{\sigma} \tag{2.17}$$

An issue that is typical of very deep neural networks is the vanishing and exploding gradient problem. What happens is that sometimes the computed gradients can either get too small or too big thus affecting the speed at which gradient descent operates and making it more difficult for it to converge. A partial solution to this is a more careful choice of the random initialization for the neural network. Furthermore, another complication with DNNs is that they tend to work best when using huge data sets but training on these can be extremely slow and this is why optimization algorithms like mini-batch gradient descent have been introduced.

The idea behind mini-batch gradient descent is to split up the training set into smaller sets called mini-batches and perform gradient descent one mini-batch at a time rather than on the entire training set. Now, while with gradient descent a single pass through the training set allows to take only a single step of gradient descent, with mini-batch gradient descent a single pass through the training set is equivalent to perform what is called an epoch. Again, one hyperparameter to properly tune becomes the size of the mini-batch. When this is the same as the training set, one ends up again with the usual gradient descent whereas when the size is one, meaning every mini-batch contains only one training example, then one ends up with an algorithm known as stochastic gradient descent, which is usually not the best in converging to a global minimum. In practice, batch normalization influences the batch size in one sense (>16 works best) and practical reasons keep it relatively low.

Other solutions to get even better results lie on the usage of exponentially weighted averages, bias correction, gradient descent with momentum, RMSprop, Adam optimization and other techniques all contributing to speeding up the learning process.

2.5.1 Batch normalization in Neural Networks

One of the most important ideas for deep learning has been the adoption of an algorithm known as batch normalization, created by Sergey Ioffe and Christian Szegezy. This solution makes the hyperparameter search much easier thus making

training very deep nets easier. The idea behind it is that similarly to how normalization is applied at the input features to make the life of gradient descent simpler, one could also normalize the activations coming out of an hidden layer to train the parameters for the next hidden layer faster. To do so, one has to take the mean for the l-th layer:

$$\mu = \sum_i a_i \tag{2.18}$$

then compute the variance on the m training examples:

$$\sigma^2 = \frac{1}{m}\sum_i (a_i - \mu)^2 \tag{2.19}$$

and finally normalize by subtracting off the mean and dividing by the standard deviation to which, for numerical stability, a quantity ε is added:

$$a_{i,norm} = \frac{a_i - \mu}{\sqrt{\sigma^2 + \varepsilon}} \tag{2.20}$$

Now, the obtained normalized activation values $a_{i,norm}$ have zero mean and unitary variance. However, it actually makes more sense for hidden units to have a different distribution to better take advantage of the used activation function and this is why usually one further scales and shifts the obtained value computing:

$$\tilde{a}_i = \gamma a_{i,norm} + \beta \tag{2.21}$$

where γ and β are learnable parameters of the model and, when using gradient descent, they are updated in the same way as one would update the weights. The effect of these two parameters is that through them one can set the mean of $\tilde{a}_i$ to be any desired value. For instance, by setting $\gamma = \sqrt{\sigma^2 + \varepsilon}$ and $\beta = \mu$ one obtains the identity function so that $\tilde{a}_i = a_{i,norm}$.

Therefore, batch normalization reduces the problem of the input values changing, causing them to be more stable so that the later layers of the neural network have more firm ground to stand on. Hence, even though the input distribution changes, it changes less, so that even as the earlier layers keep learning, the amounts that these force the later layers to adapt to are reduced, allowing each of the layers of the network to learn by themselves, somewhat independently from the others and ultimately speeding up learning in the whole network. Furthermore, batch normalization also provides a slight normalization effect because it adds some noise to each hidden layer activations thus forcing the downstream hidden units not to rely too much on any of the hidden units upstream.

2.5.2 Transfer Learning

A very powerful idea in deep learning is that sometimes one can take what an existing neural network has learned from one task and apply that knowledge to a

separate task. This technique is called transfer learning and for instance one could use it on an image recognition task to take what a neural network has learned trying to recognize a subject like cats and transfer part of that knowledge to a new task, such as X-ray scan.

Specifically, given a trained neural network, one could take the last output layer of that network, delete it together with the weights feeding into that and replace it with a new output layer with a new set of randomly initialized weights just for that layer. Then, retraining the network on the new data set (e.g. X-ray scans instead of cats' images) one could just decide to only change the weights and biases in the last layer or to retrain all the parameters in the network. Hence, the idea is to take a pre-trained network and then apply some fine-tuning to tailor the given network for a new specific task by retraining the network on the new data set.

In general, transfer learning makes sense when one has a lot of data for the starting network and relatively less data for the problem onto which transferring is applied. The other way around would simply be not helpful.

2.6 Convolutional Neural Networks basics

Computer vision is one of the areas that experienced and is still experiencing a remarkable advancement enabling several different applications that were never possible before, such as helping self-driving cars figure out where and what are the objects around them, like other cars or pedestrians. Furthermore, these advancements inspired to create a lot of cross-fertilization in other areas as well, like speech recognition, robotics, game play etc.

One of the challenges of computer vision problems is the number of features they have to deal with which can be huge, considering the amount of information that can come out of a full HD RGB image ($\approx$ 6 million input features). Hence, even using a hundred hidden units for the first hidden layer would get the weight matrix $W^{[1]}$ to be extremely large, thus making it difficult to get enough data to prevent a neural network from overfitting. Moreover, both the computational and the memory requirements to train such neural network would be too stringent, thus making their application infeasible. This is why one of the fundamental building blocks of Convolutional Neural Networks (CNNs), that are DNNs applied to computer vision, is the convolution operation, which allows to strongly reduce the number of needed parameters thanks to:

- parameter sharing: storage requirements but also computations become less complex and more efficient if the same set of weights, grouped to form a filter (or kernel) are repeatedly used over different parts of the image to calculate the outputs. Indeed, filters act as feature detectors, such as vertical edge detectors in the earlier layers, so it is likely that a feature detector useful in one part of the image will also be useful in another part of that same image.

- sparsity of connections: every output value depends only on a subset of the input pixels whose size is the same as the used filter. Hence, the rest of the input pixels will not affect the output and so their connections can be removed without affecting accuracy.

Therefore, it is common for CNNs to rely mainly on the so-called Convolutional (CONV) layers rather than on the Fully Connected (FC) layers that operate as described in the previous sections. Furthermore, what the convolution operation does on the input image is to generate, as it moves through each CONV layer of the network, an higher level of abstraction of the input data, called feature map. Every filter used while moving from one layer to another will provide an output feature map able to preserve some essential and unique information about the input data.

2.6.1 Dimensions in a CNN

In a CNN, one usually starts with an input RGB image having width $n_W^{[1]}$, height $n_H^{[1]}$ and three channels (red, green and blue) $n_C^{[1]}$ and, moving towards deeper layers, the spatial dimensions $n_W^{[l]}$ and $n_H^{[l]}$ decrease while the depth, or the number of channels $n_C^{[l]}$, increases as shown in figure 2.4. To understand how dimensions change with convolution, moving from the one layer to the next one, if $n_W^{[l-1]} \times n_H^{[l-1]} \times n_C^{[l-1]}$ are the input dimensions and $n_F^{[l-1]} = n_C^{[l]}$ filters of dimension $f^{[l]} \times f^{[l]}$ are used, then the output volume will have dimension $n_W^{[l]} \times n_H^{[l]} \times n_C^{[l]}$ where:

$$n_W^{[l]} = n_W^{[l-1]} - f^{[l]} + 1 \tag{2.22}$$

$$n_H^{[l]} = n_H^{[l-1]} - f^{[l]} + 1 \tag{2.23}$$

$$n_C^{[l]} = n_F^{[l-1]} \tag{2.24}$$

2.6.2 Padding and strided convolutions

There are some variations on the basic convolution operation that use padding as well as strided convolutions.

As for padding, this is introduced because, as can be seen in 2.22, there are two downsides in how the output dimension change when the convolution operation is performed:

1. every time convolution is applied, the image shrinks and therefore it can be used only a few times before the image starts getting too small to detect other features on it;

2. a lot of information near the edge of the image, where pixels are touched only once, would be thrown away.

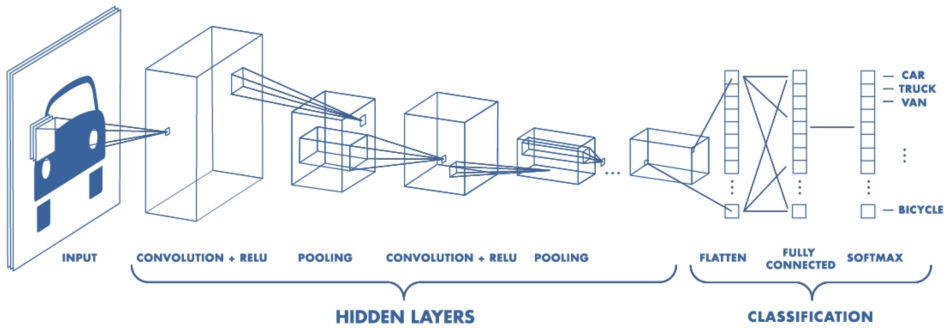


Figure 2.4. Architecture of a CNN [13].

Both these problems can be fixed before applying the convolution operation using padding on the image. The idea is to pad the image with an additional border of zero valued pixels around the edges (this is why it is also called zero padding). Therefore, if p is the padding amount, the output dimension becomes:

$$n_W^{[l]} = n_W^{[l-1]} + 2p - f^{[l]} + 1 \tag{2.25}$$

$$n_H^{[l]} = n_H^{[l-1]} + 2p - f^{[l]} + 1 \tag{2.26}$$

$$n_C^{[l]} = n_F^{[l-1]} \tag{2.27}$$

Moreover, in terms of how much to pad, there are two common choices: valid convolutions, where basically no padding is applied ($p = 0$) and same convolutions, where the padding amount is adjusted in order to have the output size same as the input size, so that the padding amount is as in 2.28.

$$p = \frac{f^{[l]} - 1}{2} \tag{2.28}$$

As for strided convolutions, it is another basic building block of convolutions. To understand how they work, some basics on how a common convolution operation is performed must be given. An usual convolution operation is performed by:

- taking the input volume, superimposing a filter on it, e.g. starting from the upper leftmost position;
- performing the element-wise multiplication and adding up the result, which is also known as a Multiply and Accumulate (MAC) operation, to obtain the output value;
- moving the filter by one position and repeat the operation until all the output elements are calculated.

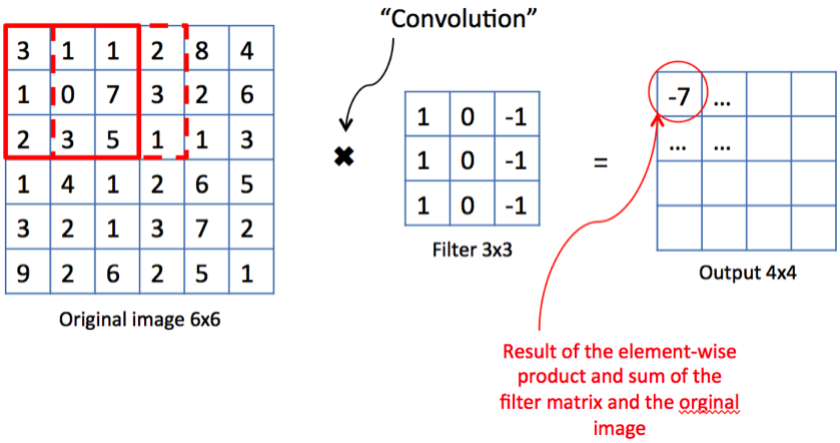


Figure 2.5. Numerical example of a convolution operation [14].

A Numerical example of this is shown in figure 2.5.

The idea behind strided convolutions is that instead of stepping over by one step, e.g. moving from the red square to the dashed red square in figure 2.5, it is possible to take larger steps. Hence, introducing the stride step s, the output dimension can be changed as:

$$n_W^{[l]} = \lfloor \frac{n_W^{[l-1]} + 2p - f^{[l]}}{s} + 1 \rfloor \tag{2.29}$$

$$n_H^{[l]} = \lfloor \frac{n_H^{[l-1]} + 2p - f^{[l]}}{s} + 1 \rfloor \tag{2.30}$$

$$n_C^{[l]} = n_F^{[l-1]} \tag{2.31}$$

To sum up, stride s tells how big the step taken by the convolutional kernel is when it jumps to the next set of data and it allows to decide how much overlap one wants between the output values in a layer.

2.6.3 Performing a convolution

After discussing how the dimension change thanks to the convolution operation, it is useful to underline what the convolution operation will actually do to the data in a CONV layer. Hence, given an input volume with dimension $n_W^{[l-1]} \times n_H^{[l-1]} \times n_C^{[l-1]}$ and $n_F^{[l-1]} = n_C^{[l]}$ filters of dimension $f^{[l]} \times f^{[l]}$, the output volume elements, if stride is unitary and no padding used, will be given by:

$$y[k_{out}][w_{out}][h_{out}] = \sum_{k_{in}=0}^{n_C^{[l-1]}} \sum_{i=0}^{f^{[l]}} \sum_{j=0}^{f^{[l]}} x[k_{in}][w_{out}+i][h_{out}+j] \times W[k_{out}][k_{in}][i][j] \tag{2.32}$$

where:

$$0 \leq k_{out} \leq n_F^{[l]},\ 0 \leq w_{out} \leq n_W^{[l]},\ 0 \leq h_{out} \leq n_H^{[l]}$$

To perform this operation, there are actually several possible solutions, as will be explored in the next chapter.

2.6.4 Pooling layers

Convolutional neural networks are not only made of CONV layers and FC layers. In fact, CNNs are typically made by properly alternating three types of layers:

1. Convolutional (CONV) layers;
2. Pooling layers;
3. Fully connected (FC) layers.

Usually, a CNN architecture (figure 2.4) alternates the use of some CONV layers and pooling layer and, only at the end, after flattening the output, few FC layers are used ending up with a softmax classifier (for classification problems). Here, pooling layers are used both to reduce the size of the representation to speed up the computation and to make some of the detected features slightly more robust. There are two possible kind of pooling:

1. max pooling: here, the output of a CONV layer is divided, for instance, into 2×2 non-overlapping regions and only the maximum value inside these regions is kept at the output;
2. average pooling: this is similar to max pooling but rather than keeping the maximum value, the average value is provided at the output.

Another interesting thing regarding pooling layers is that they have no parameters to learn during training and this is why when people report the number of layers in a neural network only layers having parameters are counted. In fact, pooling basically down-samples the input volume to reduce the spatial dimension while keeping just the important information and thus helping the later layers in performing their operations quicker and more effectively.

2.7 Some DNN Models

In the past few years, computer vision research has been focusing on how to put together the building blocks described in the previous sections to create efficient CNNs. It is actually useful to introduce some of these both because they have been proven to work effectively and because they are often taken as starting basis from people that want to develop their own CNN. Some examples are:

- LeNet [15]: this CNN was designed to recognize handwritten digits and the most famous implementation of this network is LeNet-5, which uses two CONV layers with 5×5 filters , two average pooling layers and two FC layers. This network handled around 60 000 parameters whereas today it is quite common to see networks using 10 to 100 millions of parameters. This CNN has been the first to be successfully applied for a commercial use in ATM machines to recognize the handwritten digits of check deposits. This has been trained using 28×28 images and overall performs around 340 000 MACs.

- AlexNet [2]: this CNN is named after the author who wrote the paper describing his work. This network is famous for being the first in winning the ImageNet challenge and adopting a ReLU non-linearity (which made it a much better network than the sigmoid or hyperbolic tangent used in LeNet). Furthermore, it uses 227×227 RGB images as input, filter sizes ranging from 3×3 to 11×11, five CONV layers, three max pooling layers and three FC layers. Here, the number of parameters employed is much larger compared to LeNet, around 61 millions and also the number of MACs, which is 724 million. What is interesting, however, is how the structure is still pretty similar to LeNet and the basic building blocks are still the same.

- VGG16 [16]: this CNN contains the number of used layers in its name. As a matter of fact, it goes deeper to using 13 CONV layers and 3 FC layers. What is remarkable about this network is that instead of having many hyperparameters, it uses a much simpler and uniform structure with 3×3 filters, unitary stride and same padding in the CONV layers. Then, it also employs 2×2 max pooling layers with a stride $s = 2$. This network uses 224×224 RGB images as input, around 138 million parameters and performs an overall number of MACs equal to 15.5 billion.

- GoogLeNet [3]: this CNN goes as deep as 22 layers and through its inception modules it approximates a sparse CNN with a normal dense construction. The latter is characterized by a smaller size thus reducing the computation requirements of about one order of magnitude while capturing details at various scales depending on the chosen kernel size (1×1, 3×3 and 5×5). Since its introduction in 2014, several versions of the network have been developed achieving increasingly better results. Overall the first version of this network employed 7 million parameters and performed 1.43 billion MACs on 224×224 input RGB images.

- ResNet [4]: this CNN went even deeper with configurations ranging from 34 layers up to 152. Thanks to the adoption of residual blocks employing skip connections, the authors were able to exceed human level accuracy in the ImageNet competition, effectively tackling the vanishing and exploding gradient problem typical of very deep networks. In ResNets, the idea is that

instead of computing the activations as in 2.2, a skip connection is realized which allows to compute the activation for the l-th layer as:

$$a^{[l]} = f(z^{[l]} + a^{[l-2]}) \tag{2.33}$$

of course, in order for it to work both $z^{[l]}$ and $a^{[l-2]}$ must have the same dimension and this is why ResNet use a lot of same convolutions so that the dimension of the input to the first layer is equal to the dimension of the output of the layer that is two positions ahead. Only in such condition the operation can be performed. Taking a ResNet50 as 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.

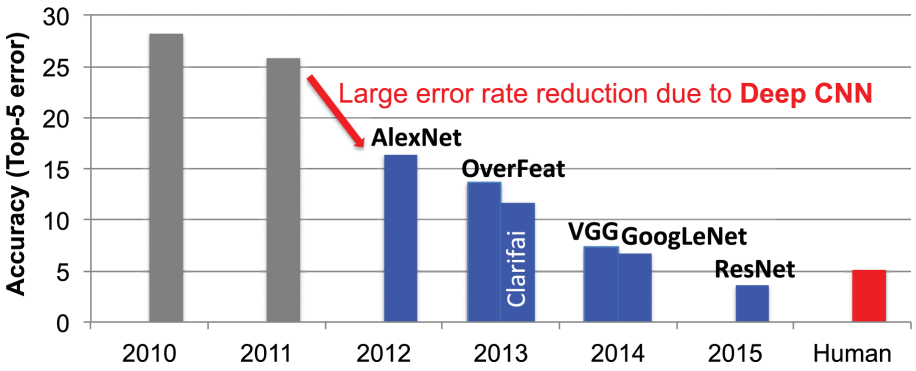


Figure 2.6. Comparison between popular DNN models [1].

Table 2.1. Summary of popular DNN.

Model	LeNet-5	AlexNet	VGG16	GoogLeNet v1	ResNet50
Input Size	28×28	227×227	224×224	224×224	224×224
Total Parameters	60k	61M	138M	7M	25.5M
Total MACs	341k	724M	15.5G	1.43G	3.9G

It turns out that a lot of these CNN architectures are difficult or finicky to replicate, because there are a lot of details about tuning the hyperparameters that are not trivial. Fortunately, many researchers constantly open source their work on the web making it easier for a newcomer to start developing his own network taking inspiration from one that has already been implemented. This is why, when developing a computer vision application, a very common workflow is to pick an existing architecture and start building the desired architecture starting from there.

In the following chapter, some state of the art hardware solutions will be introduced, with a focus on the ASIC hardware accelerator design and what inspired the Serial-MAC-Engine (SMAC-Engine).

Chapter 3

Exploring the state of the art hardware solutions for CNNs

Nowadays, the number of hardware platforms on which is possible to execute the operations required by DNNs is constantly increasing. The advancement in technology is both pushing towards the realization of more and more complex DNNs and to applying DNNs to areas where they could not be used before, e.g. in Internet of Things (IoT) end-nodes. Inference is today not limited to CPUs or GPUs only but also accessible on embedded System-on-Chips (SoCs), e.g. the Nvidia Tegra, as well as on field-programmable gate arrays (FPGAs). However, it is important to underline how processing differs depending on the chosen hardware platform as well as on the applications for which a DNN is meant to be applied.

3.1 Temporal vs spatial architectures

When dealing with CNNs, even though the fundamental operation for both the CONV and FC layers are MACs, the way these can be performed and especially parallelized changes depending on the adopted hardware platform and on what the user wishes to achieve. In particular, it is possible to distinguish among two different "extreme" architectures:

- Fully temporal architectures: these are mainly CPUs and GPUs and they aim at reaching the highest possible performance by adopting highly-parallel computing through vectorizing techniques such as Single Instruction Multiple Data (SIMD) or adopting Single Instruction Multiple Threads (SIMT) solutions. Here, there may be several Arithmetic Logic Units (ALUs) fetching data from the memory but not communicating with each other (figure 3.1). The

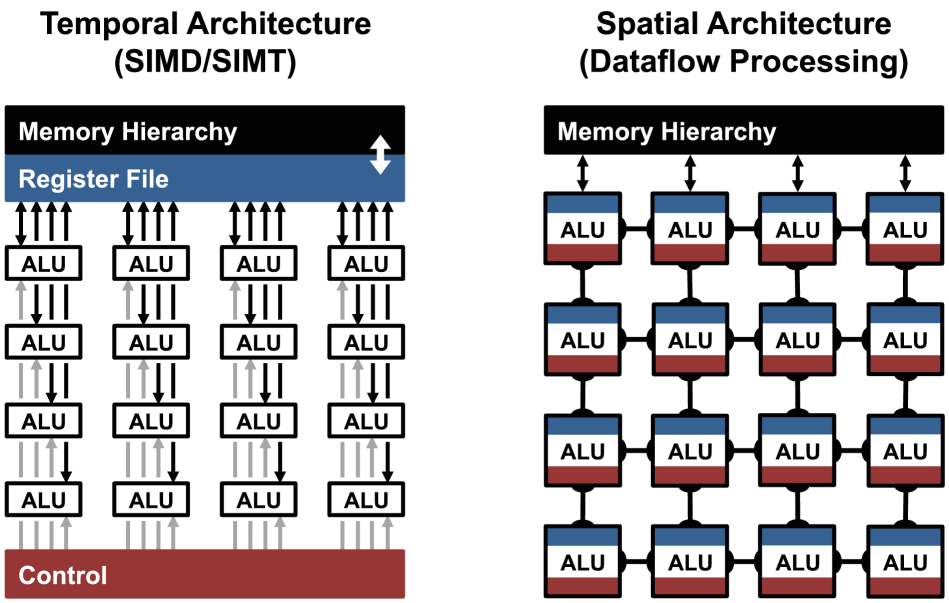


Figure 3.1. Fully temporal vs fully spatial architectures [1].

focus here is trying to increase throughput as much as possible and, for this purpose, techniques like mapping convolutions to Toeplitz matrices, employing Fast Fourier Transforms (FFTs) [17], Winograd's algorithm [18] or Strassen's algorithm [19] can be used. However, all these techniques have the downside of not being particularly memory efficient, as they require larger storage capacities, which makes them not really attractive for embedded applications.

- Fully spatial architectures: these are usually ASIC or FPGA-based designs and they focus on adopting intelligent dataflow processing through which the computational units can share data communicating with each other and increase as much as possible the data reuse before fetching new data from memory thus saving a lot of energy. Every MAC requires reading the filter weight, the activation and the partial sum from memory and then writing back the updated partial sum to memory. However, these memory accesses not only constitute a bottleneck in terms of energy consumption, as every memory access requires several orders of magnitude more energy compared to the computation, but they will also impact the throughput. For instance, memory accesses to a LPDDR3 memory are measured in tens of pJ/bit [20] compared to the computational cost, which can be measured from one [5] to several orders of magnitude less for some applications, even in fJ/op [7]. Hence, being able to locally reuse as much data as possible leads to a greater power saving, ultimately expanding the areas where DNNs can be employed.

Actually, it is unlikely for an architecture to go either fully temporal or fully spatial. In fact, all architectures tend to employ tricks coming from both solutions and so it would be better to refer to these two different approaches in relative terms as every real accelerator is in the middle.

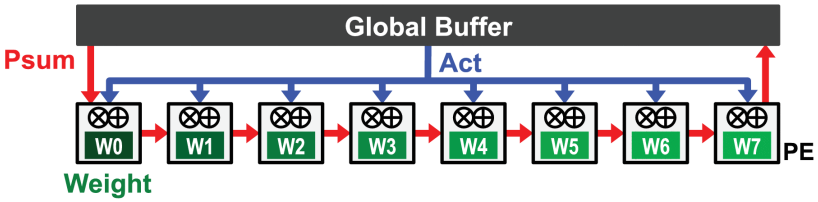
Another big difference concerning the architectural model to follow comes from the adopted number representation, which could either go for a floating point representation or for a fixed point representation. Even though floating point representation allows for a greater accuracy in the representation of data, performing operations with floating point numbers is more complex and more power consuming compared to the fixed point representation, which can reduce all to integer operations and is thus cheaper both on the employed hardware complexity and on the energy consumption. Furthermore, while floating point operations are usually performed on 32 bits data, fixed point operations can be performed on a reduced bitwidth if quantization is adopted and this will further enhance both computational speed and power saving at the cost, however, of some accuracy loss.

3.2 The architectures data flows

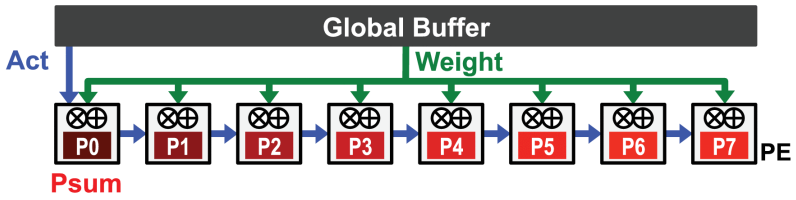
As discussed, a great amount of energy is spent because of memory accesses and trying to increase data reuse is crucial if the aim is realizing low power architectures. A beneficial approach, in this regard, comes from the adoption of a hierarchical memory organization so that the closer the data is to the processing units, the lower will be the cost to access it. Then, depending on the data that is kept closer to the processing engines, it is possible to distinguish among the following dataflows (depicted in figure 3.2):

- Weight Stationary: the minimization comes from trying to increase weight reuse. In this case the fetched weights are kept locally as much as possible before new ones are needed whereas activations and partial sums are continuously fetched from memory and written back to memory (only the partial sums) until the entire convolution operation is completed and the output activations are available. An example of this approach is the one employed by the Hardware Convolutional Engine (HWCE) [21].

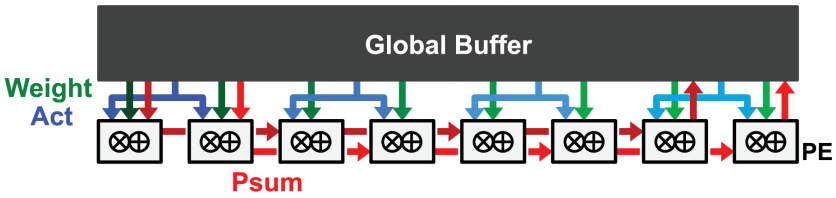
- Output Stationary: the minimization comes from keeping the partial sums of the output activations locally, without reading and writing them back to memory. Here, the options could be either a weight and output stationary solution or an input and output stationary solution. An example of the first solution is ShiDianNao [22], where weights are shared among several neurons and the DRAM accesses to them are eliminated to obtain a great power saving. Furthermore, partial sums are locally accumulated until the operation is completed. Following the second approach instead, one would stream the



(a) Weight Stationary



(b) Output Stationary



(c) No Local Reuse

Figure 3.2. Possible data flows in spatial architectures [1].

same activations to all the processing engines and broadcast the different filters weights to different processing engines while locally retaining the partial sums until the computation is done. This is the approach followed by the XNOR Neural Engine [7].

- No Local Reuse: this approach tries to increase the dimension of the global buffer at the cost, however, of removing storage elements from inside the processing engines. The downside of this, however, will be an increase in the power consumption compared to the other solutions. This approach is followed by DianNao [23], where however some registers are kept inside the processing engines to partially tackle the energy consumption.

- Row Stationary: the minimization comes from trying to maximize the reuse of all type of data, either activations, weights or partial sums. This dataflow has been introduced in the Eyeriss architecture [24], where the data reuse is maximized adopting an intelligent memory hierarchy where information can flow at different levels thus minimizing the DRAM accesses and hence strongly

reducing the energy consumption.

Of these solutions, weight stationary and output stationary can minimize the cost of accessing either weights or partial sums but the row stationary approach is the one providing the lowest energy consumption, as it optimizes accesses for all type of data. However, it must be underlined that the data reuse is something usually limited to CONV layers since there isn't usually much reuse in the FC layers.

The topic of data reuse has been further discussed also in [25], where a new analytical memory performance model to evaluate dataflow schedules in terms of local memory requirements and overall external memory traffic for DNNs has been proposed.

3.3 Edge computing applications and techniques

In the introduction, some clues regarding edge computing have been given. Indeed, moving the computation from big data centers closer to sensors may further enlarge the areas where DNNs are applied. Many IoT applications, in fact, have been limited due to the large amount of energy required by transferring information from the sensors to the data centers as well as the non-negligible latency that comes with such transmission systems. Here, any solution that is able to reduce the energy consumption without overly affecting the final accuracy is highly welcome. This is why there have been several attempts in reducing the precision of the operands and the overall number of operations to be performed.

By reducing the precision of the operands, one should accept a reasonable reduction in the final accuracy compared to the full precision model whilst greatly reducing the energy consumption thanks to:

- the overall lower memory occupation, which allows to reduce the memory size and hence the energy spent in accessing it;
- the cost for every MAC operation, which will allow for both smaller operands and faster operations;
- the reduced size of the data that has to be transferred, both thanks to the preprocessing performed at the sensor level and thanks to the reduced precision data representation which allows to transfer a higher amount of information with the same available bandwidth.

The energy consumption could be further pushed downwards when reducing the number of operations. In fact, there are many techniques that dramatically reduce the number of operations and, having different operands equal to zero and not contributing to the final computation, it is possible to:

- skip these useless operations thus saving up energy;
- further reduce the memory occupation, with the same advantages discussed above, by employing compressed representations.

However, this compressed representation will require some additional complexity in the control in order to be able to unpack this data and extract the correct positions. Still, as long as this additional complexity allows for a greater power saving, these techniques are highly welcome.

Another crucial aspect regarding applications at the end-nodes is the one concerning safety. Indeed, one may not want sensitive data like the one collected by a biomedical device to be shared or accessed by anyone and this is why it is crucial to employ encryption, such as the Advanced Encryption Standard (AES), to protect sensitive data from thieves or malicious users. Also encryption could be deployed to some hardware specific platform, as in Fulmine [5], to relieve the task from the processor. Although important, this aspect will not be further analyzed in this thesis work.

3.3.1 Reducing precision

The idea behind quantization is to take some real valued data r, such as a number represented in 32-bit floating point and map it to a smaller set of available values $[-2^{(B-1)}, 2^{(B-1)} - 1]$ to obtain a quantized representation q on B bits of the original data that minimizes the error between the real and quantized data:

$$r = S \times (q - z) = \frac{r_{max} - r_{min}}{2^B - 1} \times (q - z) \tag{3.1}$$

where r_{max} and r_{min} are the maximum and minimum values that r can have and z is the zero point for r. Hence:

$$q = \frac{r}{S} + z \tag{3.2}$$

Even though initially the focus was on reducing only the precision of the weights to increase the available memory, recent research has shown how also reducing the precision of the activations can be beneficial without greatly affecting the final precision. The simpler way to adopt quantization is to use uniform quantization, which exploits the above written equations 3.1 and 3.2. However, weights are usually not uniformly distributed but rather distributed in a fashion shown by Han et al. [26]: here (figure 3.3), it is clear how nonlinear quantization by k-means clustering and fine tuning allows for a way better representation of the data but such non-uniform quantization would require some special operations that may greatly complicate the hardware.

It must be underlined again that quantization is limited to inference as the training process requires a higher precision, generally 32-bit floating point, that

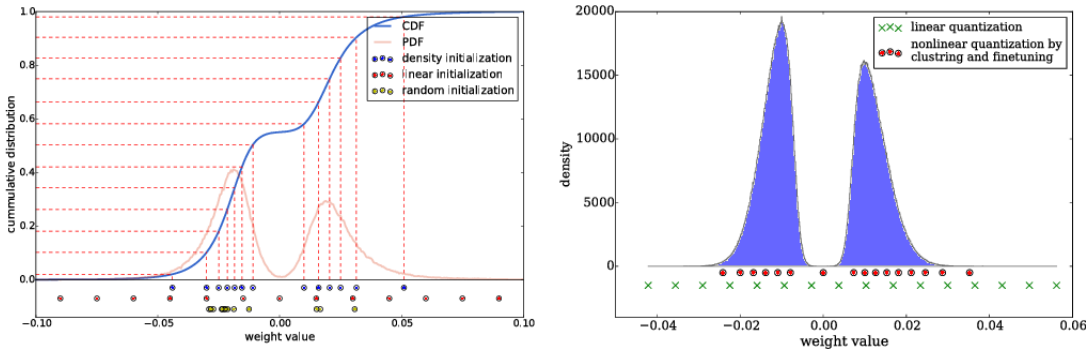


Figure 3.3. Weights distribution: linear vs nonlinear quantization [26].

only GPUs or CPUs are capable of handling (with some exceptions). Moreover, all the advantages of adopting the fixed point format over the floating point one, especially in terms of storage, comes only from the reduction in the bitwidth of the data.

Another important aspect to highlight is that it is quite common to keep, inside the data path performing the MAC operations, an internal precision that is higher than the bitwidth used to represent weights and activations. For instance, if both these operands are on N-bits, internally the precision is usually greater than $2N$-bits but, after MACs have been completed, the precision of the final output activations is again reduced to N-bits.

Reducing the weight or activation precision between 4 to 9 bits has been proven to have a rather slight impact ($< 1\%$) on the final accuracy for an AlexNet architecture. In this regard, an interesting solution where both activations and weights have been quantized down to 4-bits while keeping the accuracy comparable to the full precision model is a technique known as PArametrized Clipping acTivation (PACT) [27]. Still, there has also been some extreme solutions that further reduce precision to 1 bit [6, 7] at the price, however, of a greater accuracy loss. As far as quantization is concerned, another interesting approach is the one proposed by Incremental Network Quantization (INQ) [28], where a pre-trained full precision CNN is turned into a low precision version whose weights are either zero or powers of two. Then, a great improvement in accuracy is obtained through an iterative process that prunes the weight below a certain threshold and retrains the remaining ones. The big advantage of the representation in powers of two is that performing a multiplication becomes way easier as it will just require a proper shifting of the input activation to be performed.

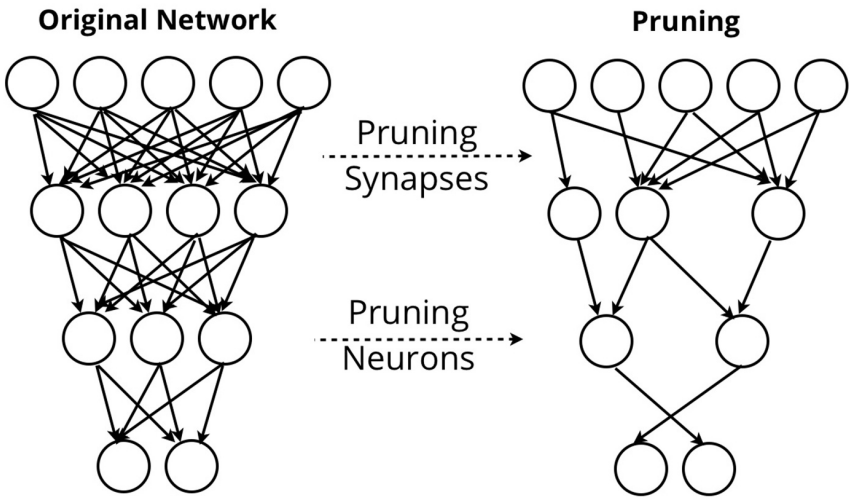


Figure 3.4. Network pruning [29].

3.3.2 Reducing the number of operations

A considerable amount of energy saving can also come from reducing the number of operations to be performed. Indeed, energy can be saved both from avoiding to perform multiplications or additions by zero-valued data and storing zeros in memory. Actually, there are several reasons that may lead to having to deal with zero-valued data, as it will now be discussed.

First of all, as introduced in the previous chapter, the non-linearity that is most widely used today is the ReLU activation function 2.5. Looking at it, it becomes quite clear how all negative values will turn out to be zeros and if activations are distributed with a Gaussian-like distribution centered around zero, considering how weights are usually distributed [26], it should not come as a surprise having output feature maps that are around 50% sparse. Hence, rather than allocating a memory where half of the saved values are zeros, solutions employing compression seem a reasonable approach to save up both area and energy consumption. An even higher sparsity can be obtained for activations if low-valued activations below a certain threshold are zeroed-out.

Another common approach, widely used to speed up the training process, consists in the adoption of network pruning. The idea behind pruning is to remove weighs inside a network that are redundant and, as such, do not contribute in the training process but only slow it down. Indeed, it is possible to remove around 50% of the weights while losing little to no accuracy. Furthermore, this technique may even maintain the original accuracy if fine tuning is adopted, that is retraining the network after pruning has been performed to recover for the accuracy loss, reaching up to 80% sparsity. However, developing hardware solutions that are able to tackle

both sparse activations and weight is not trivial.

Some compression techniques that are commonly adopted employ the use of a Compressed Sparse Row (CSR) or Compressed Sparse Column (CSC) representation. Both of these keep track just of the indexes of the non-zero values and of the non-zero values themselves, either in the row order or in the column order, as the name suggests. Here, solutions like the Huffman coding can be used to compress the indexes [30]. For instance, in the CSC model adopted by EIE [31], for every column of the weight matrix W, a vector v is used to store the non-zero weights, and another vector z is used to encode the number of zeros before the corresponding entry in v. This is done to better exploit activation sparsity.

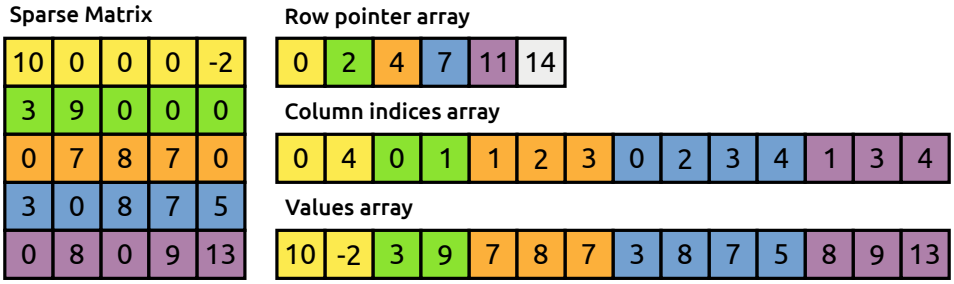


Figure 3.5. Compressed Sparse Row (CSR) format [32].

Another compression technique is the one employed by NullHop [20]. Here, a Sparsity Map (SM) is employed as a mask to keep track of where zero and non-zero values are located inside a feature map, which is easier to decode compared to the Huffman coding solution. Then, a Non-Zero Value List NZVL is used as a vector where non-zero values are stored.

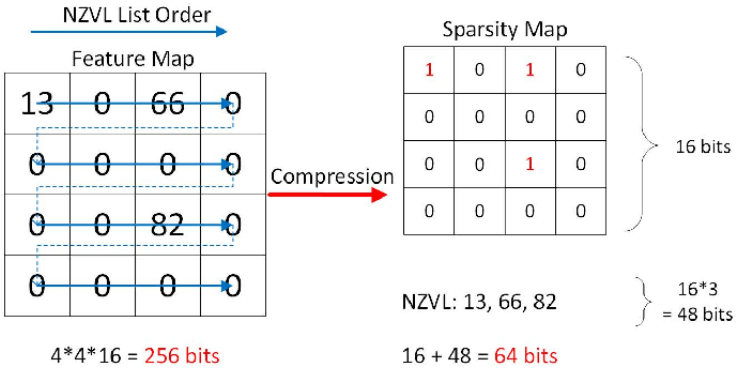


Figure 3.6. Compression technique employed by NullHop [20].

Finally, lately the trend has been to replace large filters with a set of smaller ones, either before training the network and deriving its architecture or after training,

adopting solutions such as filter decomposition. Through filter decomposition, it is possible to replace a 5×5 convolution with two 3×3 ones, even though this solution is somehow less flexible than training a network architecture from scratch and let it use just one filter size. Another approach is the one proposed by MobileNets [33], where depthwise separable convolutions are introduced. This particular kind of convolution is made up of two layers: depthwise convolutions, where a single filter is applied to each input channel and pointwise convolution, that is basically a 1×1 convolution used to create a linear combination of the output of the depthwise layer. This leads to a slight reduction in the accuracy compared to a standard convolution operation but the number of MACs to be performed and the number of needed parameters is strongly reduced, thus allowing to save a large amount of energy.

3.4 Stand-alone vs System on Chip or cluster integrated solutions

Among the cited hardware solutions, many have been realized to work as stand-alone components whose aim, supposedly, is to continuously perform the specific task they have been developed for. However, a different paradigm is the one followed by the Hardware Processing Engines (HWPEs). These are special-purpose memory-coupled accelerators that can be connected to the SoC or cluster of a PULP system [34] developed by Zurich ETH and the University of Bologna. Indeed, the latter is not meant to work as a stand-alone component, but rather as an object that is specialized to efficiently perform a specific kind of computation. In other words, the HWPEs accelerators are not meant to do all the job but rather to perform extremely well only parts of job thus amplifying the system performance and its energy efficiency.

What is peculiar regarding the HWPEs is that, differently from many other accelerators which rely on Direct Memory Access (DMA) controllers to either fetch data from a memory or write data to a memory, they can operate directly on a memory that is shared among the other elements in the PULP system, namely the Tightly-Coupled Data Memory (TCDM). This solution allows data to be seamlessly exchanged between the accelerators and the cores of the system, as it happens in Fulmine [5] or XNOR Neural Engine [7].

Looking at the structure of a HWPE (figure 3.7), it is possible to notice, besides the presence of the actual Data Path inside the internal engine, a streamer which acts as an interface between the internal engine and the TCDM connected to the accelerator as well as a peripheral interconnect through which the control con be programmed. This can be thought of as an extremely specialized DMA controller for the accelerator operation.

Since the Data Path developed in this thesis work has been later on integrated

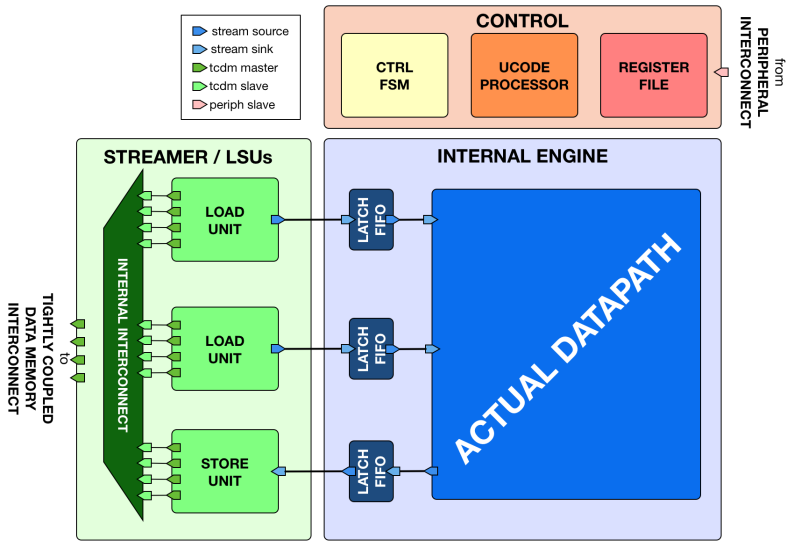


Figure 3.7. Example of Hardware Processing Engine (HWPE).

in the HWPE structure, the details on the procedure followed to do so as well as some further details regarding the structure of this system will be discussed in the following chapters.

Chapter 4

Serial-MAC Engine: from the starting hypothesis to the realization

4.1 The starting hypothesis

The starting hypothesis in deriving the SMAC-Engine was trying to find a reasonable approach to perform a convolution for a setup like the one depicted in figure 4.1.

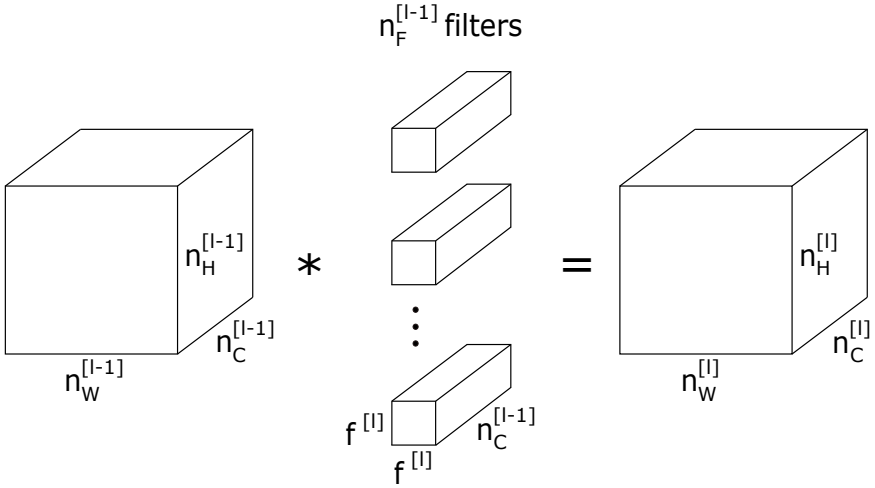


Figure 4.1. Starting hypothesis convolutional layer.

Here, there is a $n_W^{[l-1]} \times n_H^{[l-1]} \times n_C^{[l-1]}$ input volume, with $n_W^{[l-1]} = n_H^{[l-1]} = 32$ and $n_C^{[l-1]} = 128$, $n_F^{[l-1]} = 128$ filters with size $f^{[l]} \times f^{[l]} \times n_C^{[l-1]}$, with $f^{[l]} = 3$ and an output

volume $n_W^{[l]} \times n_H^{[l]} \times n_C^{[l]}$ preserving the same spatial dimension, which suggests zero padding has been used to perform a same convolution and that $n_F^{[l-1]} = n_C^{[l]}$. The architecture could easily support padding, being it widely used in several state of the art architectures and a technique whose implementation does not really require additional hardware to be employed for the Data Path (DP). Moreover, the chosen volume is typical for many CNNs and thus is a good candidate to generalize the common operations performed in several networks. A further starting assumption was to have quantized sparse data for activations and kernel weights with a sparisty of 50%.

As discussed in the previous chapters, exploiting sparsity one can strongly reduce the memory occupation adopting a compressed representation of the data as well as avoid performing multiplications or additions by zero thus saving a great amount of memory. Furthermore, such sparsity percentage seemed reasonable since adopting ReLU activation functions usually clamps to zero around half of the activations whereas for weights it is a condition that can easily stem from the adopted pruning strategy during training while keeping a reasonable accuracy loss.

As far as the CNN architecture is concerned, VGG16 seemed a reasonable starting point, mainly due to its high regularity (it employs 3×3 kernels in powers of two), even though, as reported in the previous chapters, this requires a larger number of MACs to perform and parameters to store compared to other state of the art architectures.

Taking sparsity as a starting point, two were the possible ways to tackle it: either allocating a larger number of computational units but enabling the operations only when non-zero operands are provided, thus adopting a zero-skipping approach, or reducing the number of multiplications to perform by increasing the complexity at the control level to exploit a compressed representation format. The latter solution would require to decode the compressed data after fetching it, perform the MAC operations and then encode it again when the result has to be written back in memory, unless one is able to perform operations directly on the compressed data, which is not simple to achieve. Out of these two possibilities, the latter seemed more intriguing and hence triggered the search for a some intermediate representation that could be an acceptable trade-off between minimizing the memory occupation and maximizing the ease of deployment to the DP.

Among the available formats provided by the state of the art, the CSR and CSC with Huffman coding, although proven to be working for several architectures, did not seem particularly appealing for architectures working on aggressively quantized networks. In particular, while effective for architectures working on full precision, when the bitwidth of the operands is reduced, there is not really a great difference between saving the indexes to derive the positions of non-zero values and directly saving data regardless of its value, unless the degree of sparsity of such data is extremely high. Furthermore, one should also consider the additional complexity required by the control to decode the encoded information. This is why, the

compression format proposed by NullHop [20] could probably adapt better to architectures working on very low precision data, unless one wants to realize binary networks. In the latter case, the Sparsity Map introduced in NullHop would be enough and no Non-Zero Value List would be needed to keep track of the non-zero operands.

For what concerns the parallelism of data, the reasons discussed in the previous chapter lead to choose a bitwidth of $P_a = 8$ for the activations and $P_w = 4$ for the weights. In fact, even though there was interest over binary and ternary networks, a network that could be more easily adapted to a change in parallelism looked like a better starting point and something that could eventually be applied to a wider range of applications. Furthermore, such precision should not degrade the final accuracy too much and this is why going below four bits was not so appealing. The reason leading towards the above mentioned values for P_a and P_w, was to try not to deteriorate too much the final accuracy while at the same time both speeding up the computation and strongly reducing the memory occupation, especially for the weights, whose overall number tends to be greater than the number of activations and would thus require a higher storage capacity.

Finally, with the idea to eventually integrate the derived DP on a Hardware Processing Engine (HWPE) in the PULP system developed by Zurich ETH and the University of Bologna, an additional constraint has been the one concerning the available bandwidth, which for such structure is limited to 128 bits per cycle. The methodology followed to manage this bandwidth will be further discussed in the following sections.

4.2 From the basic to the final Data Path structure

After the preliminary hypothesis were made, the search drifted towards deriving a possible DP that could perform the required computations efficiently as well as tackle the presence of sparsity. Here, the choice was either to deal with sparsity at the DP level by avoiding computations by zero or to leave to the control the complexity and schedule to the DP only the useful operands. With the idea of following the second solution, an interesting structure for the DP seemed to be the one proposed by Sharify et al.'s Loom [8]. In Loom parallel multiplications are substituted by serial ones while at the same time keeping the throughput unchanged by allocating a greater number of computational engines. Loom structure was inspired by DaDianNao [35], where MAC operations are performed in a parallel fashion close to the one depicted in figure 4.2. Such structure is able to perform M MAC operations per cycle. However, allocating M multipliers and an adder tree capable of handling all the operands coming out of the multipliers would severely

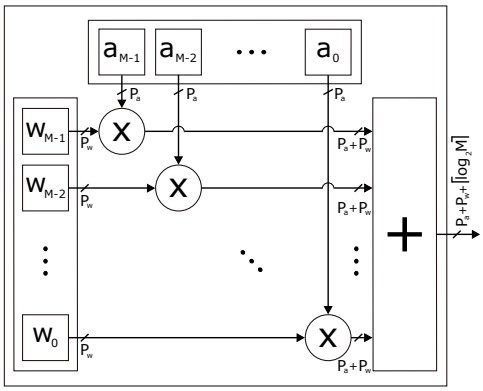


Figure 4.2. Data path parallel structure example.

impact the final clock frequency, especially when M is large. This is why Loom introduces a serial structure, where multipliers are substituted by simple AND gates, as depicted in figure 4.3. Here, to perform M MAC operations that are actually equivalent to the ones performed by the parallel structure, two accumulators after the first block have to be inserted in order to complete the entire operation. In fact, while the weight bits are stationary, all the activation bits are shifted serially and all the partial sums relative to the fixed weight bit have to be summed together by a first accumulator AC1, that keeps the coherence with every addend by performing a shift to the right. This is correct provided that activations and weights are shifted from the less significant bit (LSB) to the most significant bit (MSB), otherwise the shifting would be performed in the opposite direction[1].

Similarly to what the accumulator AC1 does, the accumulator AC2 takes care of the results obtained for each of the weight bits. Furthermore, the register between AC1 and AC2 is controlled by a signal MSB_w inverting the stored content whenever the output of AC1 has been obtained while the weight bits were fixed to the MSB[2].

Of course, differently from the parallel solution, the serial one will require an additional number of cycles to perform M MACs. In fact, instead of M MACs per cycle, it will be able to perform M MACs after $P_a \times P_w$ cycles. This means that for the serial solution to match the same throughput as the parallel one, the number

[1]Actually Loom proposes a solution where shifting is performed to the left direction and both the weights and activation bits are shifted from the MSB to the LSB. The final result is exactly equivalent to the one presented here that is what the final DP has been based on.

[2]Note that the parallelism coming out of the first adder is $\lceil \log_2 M \rceil + 1$ because it takes into account the case when M is exactly a power of two and to represent the output correctly an additional bit is required. Furthermore, a register between the bit adder and the first accumulator has been added to reduce the critical path delay.

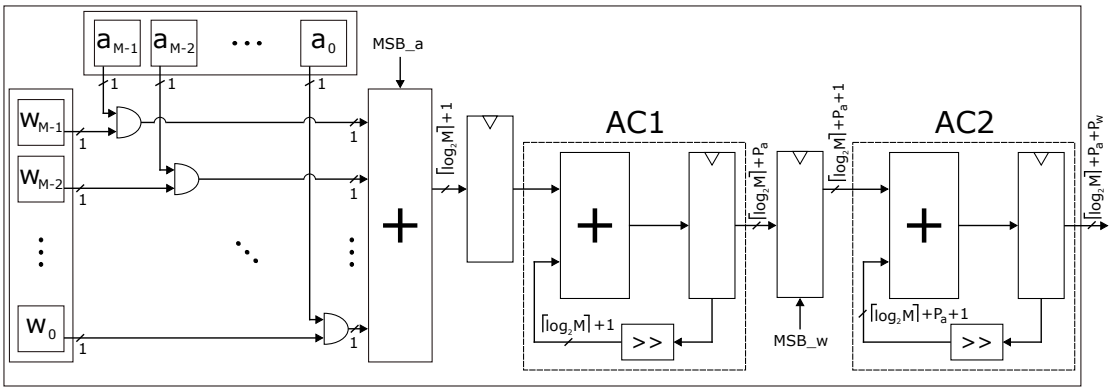


Figure 4.3. Data path serial structure example.

of basic blocks needs to be increased from one to $P_a \times P_w$[3]. This is still reasonable, considering that the serial solution employs way less hardware than the parallel counterpart. The comparison in terms of area and maximum frequency of the two solutions will be later on discussed in details, but of course an area overhead for the serial solution was expected due to the number of computational engines that needed to be allocated to match the performance of the parallel solution.

An interesting addition that Loom adopts in its architecture is the presence of a leading one detector in its control to further push the throughput. In fact, Loom can potentially overcome the DaDianNao counterpart whenever the operands are actually on a reduced parallelism compared to their possible full extension, that is 16 bits, provided that the basic serial block has been replicated $P_a \times P_w$ times. This is because with lower bitwidths a lower number of shifting will be required and hence an even higher throughput will be achieved. However, due to the chosen quantization, adopting such solution did not seem to be extremely beneficial, especially considering the complications required in the control and the number of possible cycles that could be saved. This is why a solution that could match the parallel one when all computational units are working seemed a reasonable achievement already. Furthermore, the serial computation should still provide a higher maximum frequency compared to the parallel one due to the shorter critical path and could potentially outperform the parallel approach while keeping the same throughput.

Another important aspect to mention is that, differently from Loom, the architecture provided here and that will be later described in detail does not support pooling layers. Pooling layers are usually employed after some CONV layers to perform a downsampling that ultimately reduces the spatial dimension of the input

[3]Here throughput is intended as the number of operations per cycle and not the maximum operating frequency.

volume while keeping just the relevant information. However, to ease the implementation, max pooling has been thought to be designated by some other hardware block or performed in software thus relieving the SMAC-Engine from such task.

4.2.1 Area comparison

For analysis reasons, a comparison in terms of area vs operating frequency between the parallel and serial approach has been made. To do so, Synopsys Design Compiler® has been employed together with a umc-65 nm library in worst case conditions (0.9 V supply voltage and 125 °C). Here, it was interesting to see how area and frequency varied depending on the number of operands M for both configurations as reported in figure 4.4. Here, as expected, it is possible to see how

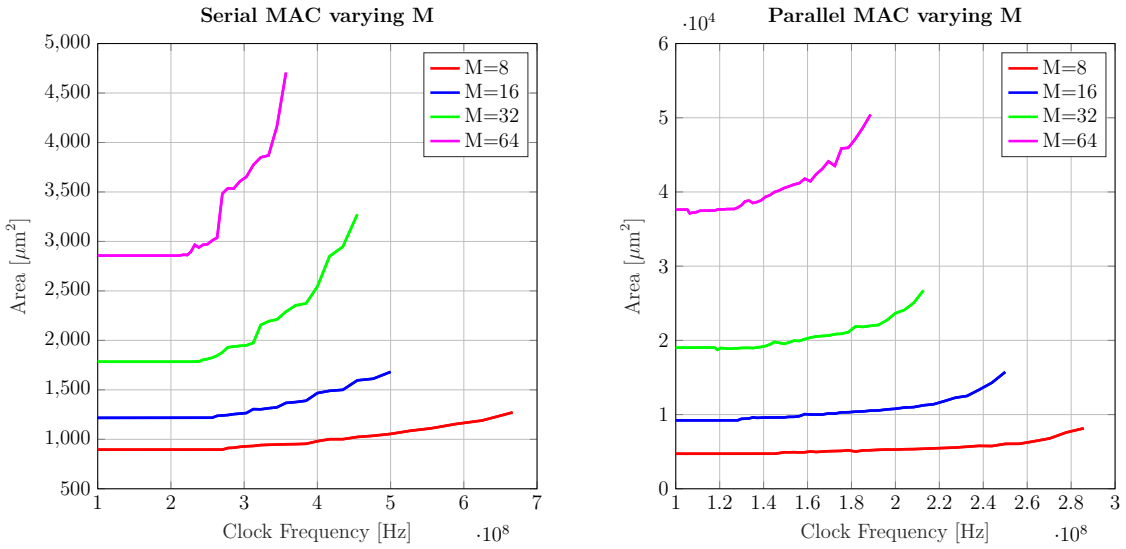


Figure 4.4. Single SMAC vs Parallel MAC varying M.

the solution providing the highest operating frequency is the one with a reduced number of operands whereas the solution requiring the largest area is the one with the maximum number of operands for both the parallel and serial case. Moreover, a single SMAC block requires way less area than the parallel counterpart (around an order of magnitude less) and is able to reach a maximum frequency that is nearly twice the one reached by the parallel solution.

However, as previously mentioned, in order for the SMAC to match the same throughput of the parallel solution it is necessary to increase the number of SMAC blocks from one to $P_a \times P_w$. This is why an area vs frequency analysis has been repeated after replicating the SMAC blocks $P_a \times P_w$ times to match the throughput of the parallel solution. The result when $M = 16$ is reported in figure 4.5 and shows how, despite a single SMAC block is smaller than the parallel counterpart,

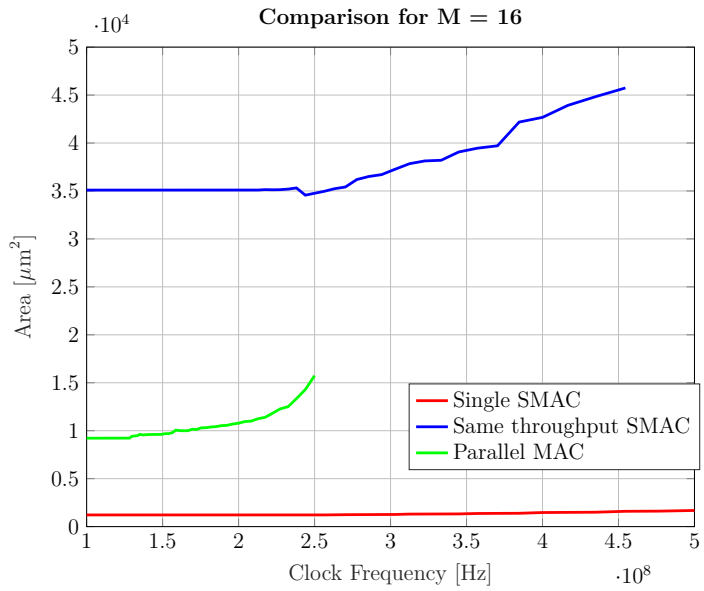


Figure 4.5. Single SMAC, same throughput and Parallel solutions comparison.

employing $P_a \times P_w$ SMAC blocks leads to an overall area that is around $\times 2.91$ larger than the parallel solution but with the advantage of working at around twice the frequency.

Hence, since there were no significant area limitations in the starting hypothesis that would severely affect the adoption of the serial approach, the obtained result seemed a promising starting point. Furthermore, among the possible number of operands M to choose from, a value of $M = 16$ seemed a reasonable compromise between area, frequency, available bandwidth and the number of filters that could be computed in parallel and that could maximize the use of the SMAC blocks. The latter two elements will be further discussed in the next sections.

4.2.2 Deriving the data flow

When dealing with spatial architectures, there are several data flows that could be adopted. Indeed, even though a convolution operation requires to perform a loop that is usually as the pseudocode presented in 1, and which could be depicted as in figure 4.6[4], this is not the only way to do it. The linearity of such operation allows to change the loop order without affecting the final result.

[4]Here the same notation used in chapter two to introduce the convolution operation has been adopted 2.32.

Algorithm 1 Convolution: weight stationary

1: **for** $k_{out} = 0 : n_F^{[l]}$ **do**
2: **for** $k_{in} = 0 : n_C^{[l-1]}$ **do**
3: **for** $h_{out} = 0 : n_H^{[l]}$ **do**
4: **for** $w_{out} = 0 : n_W^{[l]}$ **do**
5: **for** $i = 0 : f^{[l]}$ **do**
6: **for** $j = 0 : f^{[l]}$ **do**
7: $y[k_{out}][w_{out}][h_{out}] += x[k_{in}][w_{out} + i][h_{out} + j] \times W[k_{out}][k_{in}][i][j]$

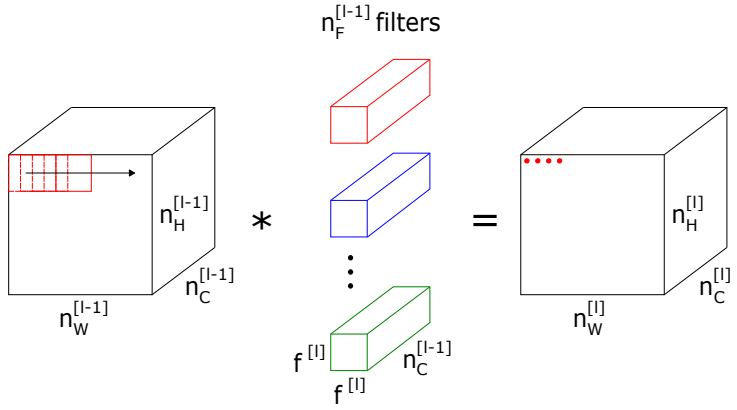


Figure 4.6. Weight stationary data flow visualization: sliding windows algorithm.

The loop in algorithm 1 is the basic convolution loop that uses the sliding window approach, as presented in figure 2.5, which could be adopted with a weight stationary data flow where the same filter channel is moved along an input feature map before moving to the next one. Here, however, every time a MAC is performed, the partial sum has to be written back to memory and fetched back until the convolution has been completed. In this case, one could fetch weights coming from the first channel of multiple filters and share the same activations among them, thus computing more partial sums at once but also increasing the amount of data that has to be written back to memory once the MAC operation is completed. Moreover, considering the structure of VGG16, it is possible to notice how the spatial dimension shrinks pretty fast whereas the number of channels, and hence of used filters, tends to increase as one moves deeper in the network. This is why an alternative output and input stationary data flow like the one in algorithm 2 has been explored:

Algorithm 2 Convolution: output and input stationary

1: **for** $h_{out} = 0 : n_H^{[l]}$ **do**
2: **for** $w_{out} = 0 : n_W^{[l]}$ **do**
3: **for** $k_{out} = 0 : n_F^{[l]}$ **do**
4: **for** $i = 0 : f^{[l]}$ **do**
5: **for** $j = 0 : f^{[l]}$ **do**
6: **for** $k_{in} = 0 : n_C^{[l-1]}$ **do**
7: $y[k_{out}][w_{out}][h_{out}] += x[k_{in}][w_{out} + i][h_{out} + j] \times W[k_{out}][k_{in}][i][j]$

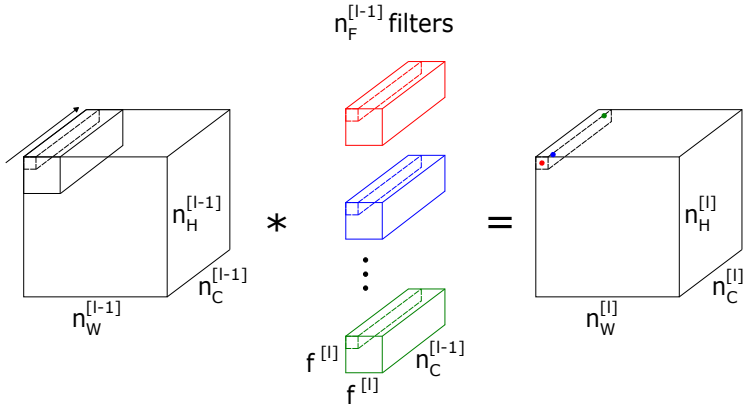


Figure 4.7. Output stationary data flow visualization.

The derived output and input stationary data flow, reported in figure 4.7, is such that convolution is performed by first fetching the needed activations along the feature maps direction rather than the spatial ones, then use these activations for as many filters as possible inside the considered convolutional volume and finally by locally accumulating the partial sums until the entire convolution is done. Of course, choosing a data flow over the other inevitably affects the DP structure.

In particular, the latter approach requires the adoption of another accumulator to locally preserve the partial sums whereas the first structure would require allocating some registers at the input to locally save the fetched weights and use them as long as they are needed before new ones have to be employed.

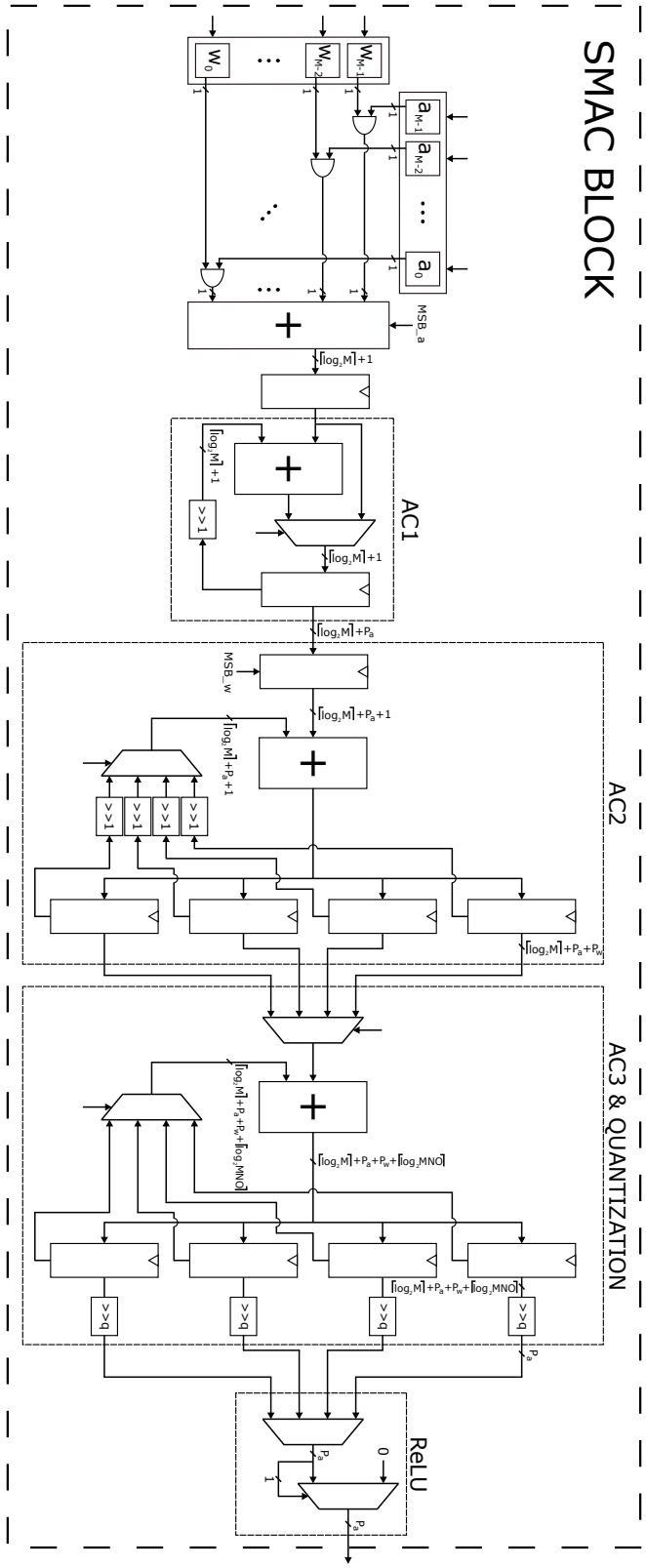


Figure 4.8. Final structure for a single SMAC block.

After choosing the output stationary data flow, a solution to try to also exploit the reuse of input activations would be to increase the number of registers after the above mentioned second and third accumulators in order to exploit the same activations for as many filters as possible. Here, replicating these registers four times, as in figure 4.8, seemed a good compromise between the inevitable area overhead and the local reuse of the input activations. Furthermore, considering the number of filters used by VGG16 and the area vs frequency analysis performed in the previous section, $M = 16$ seemed a thoughtful choice for the number of operands each SMAC block works with, since it allows a single SMAC block to locally retain the input activations for up to four different filters and thus helping to deal with the increasing number of filters in deeper layers.

4.2.3 The available bandwidth

To derive the number of SMAC blocks to allocate, the available bandwidth was a necessary constraint. In particular, thinking at a possible integration of the structure in a HWPE[5] of a PULP system, being compliant with such structure allows for a typical bandwidth of 128 bits [7, 5]. In addition, choosing not to overlap operands fetching and writing back to memory, it was possible to assume such bitwidth to be available both at the input and at the output of the DP structure. Taking this into account as well as the parallelism of the activations and weights, this meant being able to fetch, in a single cycle, either $128/P_a = 16$ activations or 128 weight bits. This is another result that somewhat justified the usage of $M = 16$ operands for each computational block.

Considering $M = 16$, the parallel implementation with such available bandwidth would allow to allocate two of the blocks in figure 4.2 and hence perform 32 MACs per cycle. Equivalently, to obtain the same results with the SMAC blocks one would need to replicate the SMAC structure $2 \times P_a \times P_w = 64$ times. Here, considering the flexibility with which weights can be saved in memory, it was possible to replicate the structure in a direction that could maximize the number of usable filters with the same activations. Specifically, allocating 64 SMAC blocks sharing the same activations was possible thanks to the serial approach in performing multiplications: both the available bandwidth to fetch 128 weight bits per cycle (instead of $128/P_w = 32$ weights on the entire parallelism) and the $P_a = 8$ cycles required to shift the activation bits helped in reaching a total of $128 \times 8 = 1024$ bits, that is exactly the number of bits required to feed 64 SMAC blocks, each working on $M = 16$ operands.

In other words, the idea was to use a cycle to fetch 16 activations and then exploit

[5] https://github.com/pulp-platform/hwpe-mac-engine
https://github.com/pulp-platform/hwpe-stream
https://github.com/pulp-platform/hwpe-ctrl

the latency (in full regime) introduced by the shifting of these activations to fetch a total of 1024 weight bits to later feed to all the 64 SMAC blocks at once. Hence, this allowed to avoid any latency as far as weights need to be fetched. However, whenever a new group of 16 activations needs to be fetched, an additional cycle will inevitably be lost due to the further cycle needed to get all the 1024 weight bits required by the structure.

Therefore, the replication of the SMAC blocks, combined with the replication of four registers after the second and third accumulators, provided a structure able to work with up to 64 filters at once and able to locally retain up to 256 partial sums before writing back to memory. Of course, during the writing back operation, due to the non overlapping assumption between inputs and outputs neither new input activations nor new input weights can be fetched and therefore this will also introduce some latency that is proportional to the amount of output activations that has to be written back to memory.

4.2.4 The final Data Path structure

After deriving the structure for a single SMAC block and discussing the constraints imposed by the available bandwidth, the final Data Path structure can be depicted as shown in figure 4.9. As far as a single SMAC block is concerned, the structure in figure 4.8 shows some further blocks after the third accumulator. Ideally, after the full convolution operation has been completed, one should properly quantize the output so that it can be again represented with the same parallelism of the input activations, thus with P_a bits. Here, the amount of shifting depends on the specific layer where convolution is performed. However, the idea of employing barrel shifters to perform such operation did not seem particularly convenient, mostly due to the further area overhead requirements, which would eventually lead to an unacceptable area occupation. A solution to this was to perform quantization serially for each of the AC3 registers. Hence, once the convolution operation is done, a pre-loaded programmable counter can be exploited to serially shift the values in the registers of the third accumulator. Then, when the shifting is completed, a writing back to memory is performed while the DP components are stalled, thus both guaranteeing the non overlapping of the input and output as well as avoiding unintended switching in the internal structure of the DP.

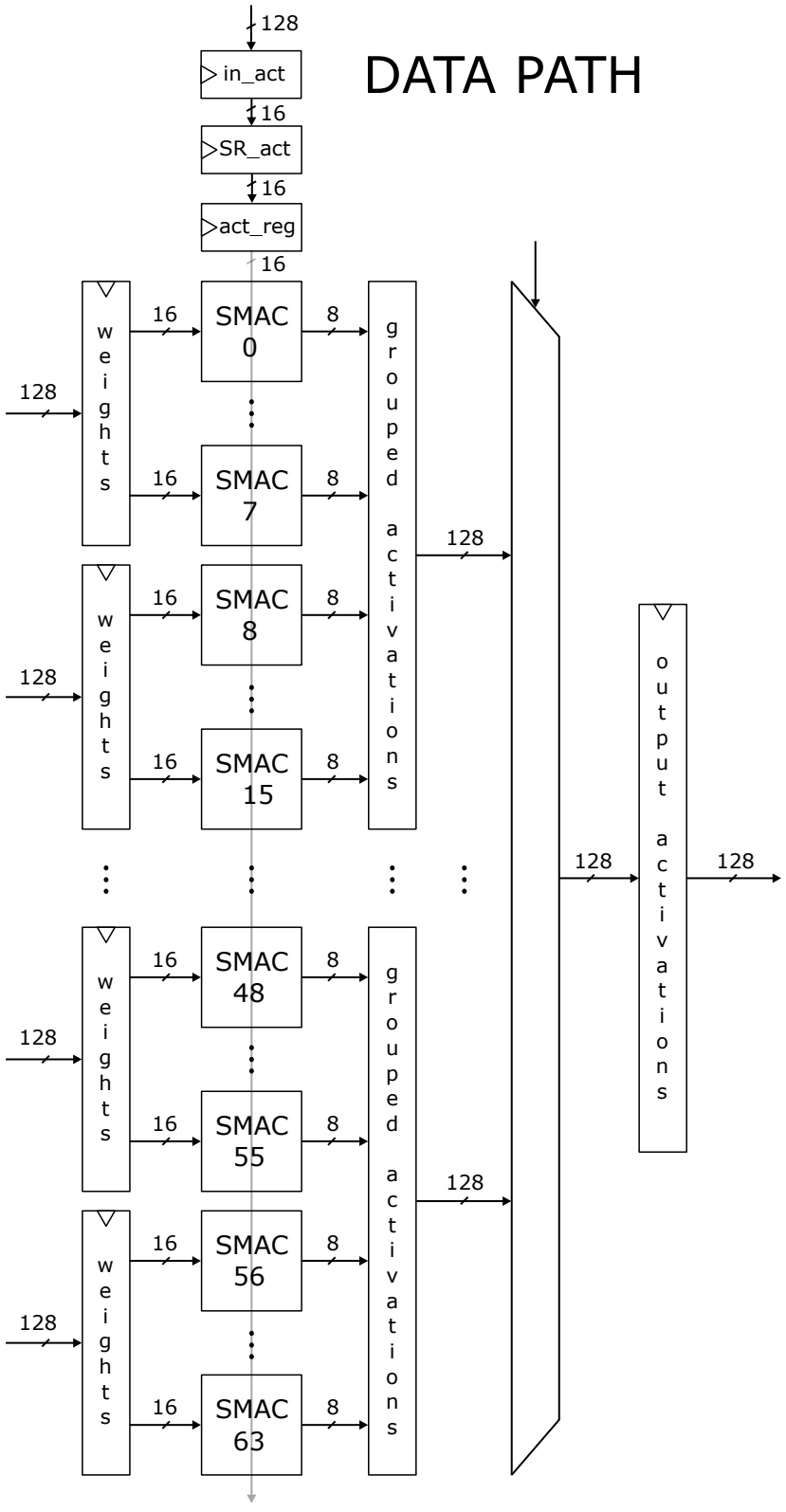


Figure 4.9. Full Data Path Structure.

Another aspect to discuss concerns batch norm and the activation function. In particular, employing batch norm requires to implement a thresholding mechanism where some thresholds are fetched from memory, usually at the beginning of a new layer, and locally kept to later normalize the quantized output with them. However, for the sake of simplicity and as a first implementation, batch norm has not been introduced in the architecture of the SMAC block, though it would be useful to investigate its implementation in the future. As for the activation function instead, since the choice was to implement a basic Rectified Linear Unit (ReLU), it was sufficient to employ a two way multiplexer whose selection signal is coincident with the most significant bit of the quantized output, namely the sign bit. Indeed, the output of the multiplexer will be zero if the MSB of the quantized output is one, hence negative, and it will coincide with the quantized output if the MSB is zero and hence it is a positive number. Here, employing more complex activation functions or even moving the ReLU threshold would require some additional or specific hardware whose implementation has not been addressed.

In this Data Path structure, the 128 bits at the input, as shown in figure 4.9, can either feed the activations register in_act, from which they are later sent to the SR_act block containing 16 shift registers in charge of streaming 16 bits at once first to the act_reg register and later to all the SMAC blocks, or feed one of the eight weights registers at the boundary of the structure. For these registers, the control will be in charge of generating the proper write enable signals that are necessary to sample the correct data at the right time and in the correct order. The 128 bits at the output, instead, are generated by first grouping the outputs of 16 SMAC blocks together and later employing a multiplexer to select which one, out of the four groups, will be the one to be written back to memory. Therefore, during the writing back phase, the DP will be able to write back 16 activations per cycle, meaning that the cycles required by the writing back phase will inevitably be dependent on the number of filters employed by a specific layer.

Similarly to the analysis performed for the basic Data Path implementation, Synopsys Design Compiler® has been employed to perform an area vs frequency estimation of the structure in figure 4.9, again in worst case conditions (umc-65 nm library with 0.9 V supply voltage and 125 °C temperature). The results are reported in figure 4.10. The analysis shows how the structure is capable of reaching almost the same maximum operating frequency (416 MHz) as the one provided by a single SMAC block. The area overhead, compared to the results reported in figure 4.5 is about $\times 3.22$ larger. Indeed, note that the results in figure 4.5 refer to a structure replicated 32 times to match the same throughput of the parallel one, hence the values reported there have to be doubled for a proper comparison. Furthermore, on should also take into account the registers at the boundaries of the structure as well as the third accumulator and the multiplexer for the ReLU activation in each SMAC block of the full DP.

A possible improvement to the provided DP structure, which may require some

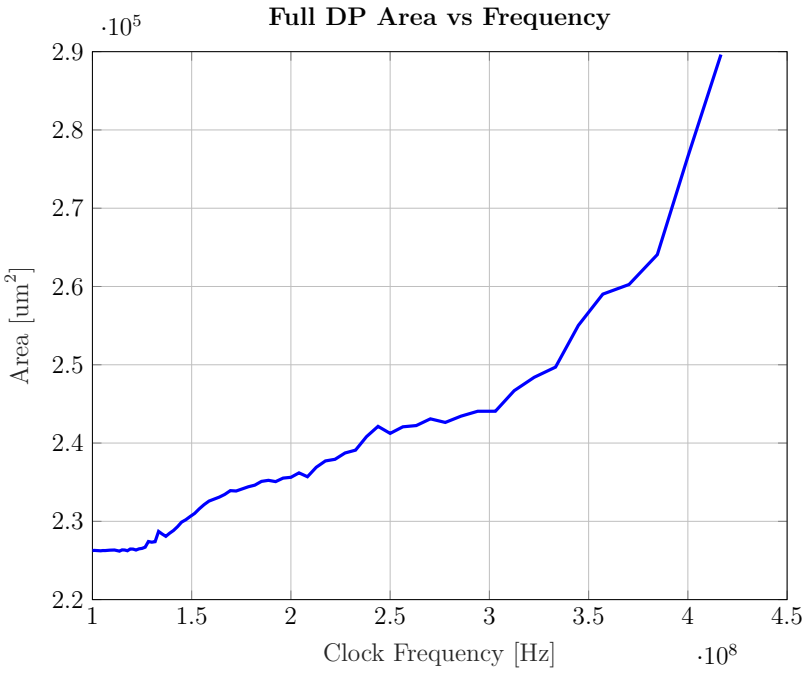


Figure 4.10. Area vs frequency analysis for the final DP structure.

further investigation, would be allocating SMAC blocks not only in the vertical direction but also in the horizontal direction to maximize the number of operations that can be performed per cycle. With this approach, at the cost of some further cycles lost when fetching new activations and during the write back, it could be possible to strongly increase the achievable throughput as this would allow to work with multiple convolutional volumes at once. Furthermore, with this solution weights could be shared horizontally and thus increase their usage, ultimately reducing the number of cycles required to fetch them from memory and saving a considerable amount of power. However, there are three non trivial downsides to this solution. First of all, the area overhead may get to a point where the structure “explodes" making it impossible to employ it. Second, weights can be horizontally shared assuming the network is dense (or not sparse) and hence their position is not altered in any way. However, if one were to tackle sparsity and only fetch weights corresponding to the non-zero positions of the activations, there would be no guarantee that two different spatial coordinates in the input volume share the same non-zero values over the channel direction. Hence, even though this could potentially boost the architecture throughput, it will be limited to the dense case. Third, working with multiple convolutional volumes at once requires a further complication in the control, which will need to take care of generating the correct addresses, with the correct offsets, where the input activations are stored in memory as well

as properly handle the addresses where the output activations will be stored once the convolution is completed.

A final aspect to discuss is related to how the structure handles working with more than 256 filters. For instance, if 512 filters are used, being the structure able to retain only up to 256 output activations, to complete the convolution the same input activations will inevitably have to be fetched twice and therefore the computation will be split in two parts, each working with 256 filters.

4.2.5 Analysis on VGG16 and MobileNet

An additional interesting analysis that has been performed concerns a comparison in terms of achievable throughput for two state of the art architectures, namely VGG16 and MobileNet [33] employing SMAC blocks for their computations. Such analysis has been carried out on a dense (not sparse) hypothesis neglecting the overhead introduced by possible FIFOs or registers at the boundary of the architecture as well as the presence of lost cycles due to non valid handshakes at the interfaces. Furthermore, this analysis focused on the behavior of both networks when dealing with hidden CONV layers, since this is where the largest computational effort is required.

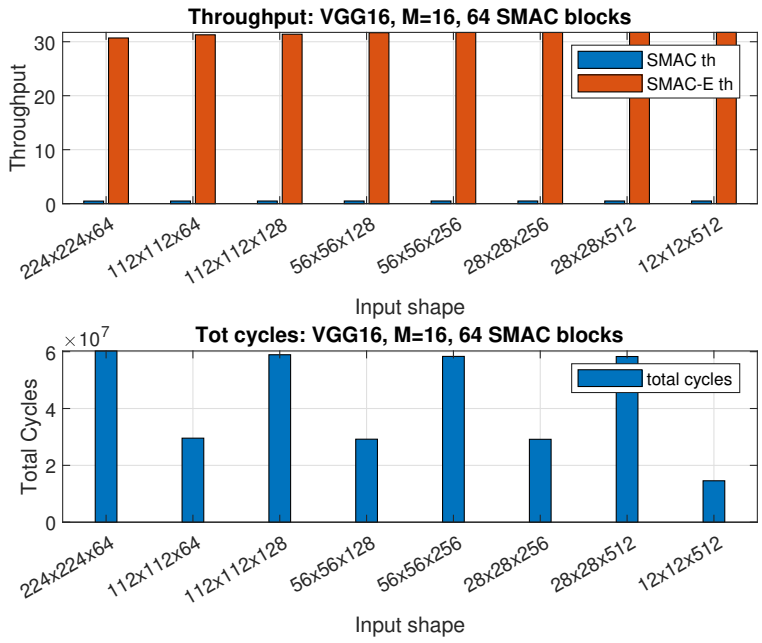


Figure 4.11. Throughput for a single SMAC and for SMAC-Engine for each layer.

In figure 4.11 there are two histograms showing the behavior of SMAC-Engine in terms of achievable throughput expressed in MAC/cycle and total number of cycles

required by each layer of VGG16 when $M = 16$ and 64 SMAC blocks are allocated. As expected, the throughput that a single SMAC block is able to achieve is nearly 0.5 MAC/cycle. This is because each SMAC block is able to perform $M = 16$ MACs in $P_a \times P_w = 32$ cycles. However, replicating the structure 64 times, allows to reach a throughput that is comparable to the parallel solution, that is 32 MAC/cycle.

Table 4.1. Analysis on VGG16, 64 SMAC blocks with $M = 16$.

Net	**Input Shape**	**Filter Shape**	**# cycles**		**Output Shape**	**Throughput**	
			sing_vol [k]	**total [M]**		**SMAC [MAC/cycle]**	**SMAC-E [MAC/cycle]**
VGG16	$224 \times 224 \times 64$	$3 \times 3 \times 64 \times 64$	1.20	60.26	$224 \times 224 \times 64$	0.480	30.69
	$112 \times 112 \times 64$	$3 \times 3 \times 64 \times 128$	2.36	29.57	$112 \times 112 \times 128$	0.489	31.28
	$112 \times 112 \times 128$	$3 \times 3 \times 128 \times 128$	4.70	58.92	$112 \times 112 \times 128$	0.490	31.39
	$56 \times 56 \times 128$	$3 \times 3 \times 128 \times 256$	9.30	29.21	$56 \times 56 \times 256$	0.491	31.67
	$56 \times 56 \times 256$	$3 \times 3 \times 256 \times 256$	18.60	58.33	$56 \times 56 \times 256$	0.496	31.71
	$28 \times 28 \times 256$	$3 \times 3 \times 256 \times 512$	37.20	29.17	$28 \times 28 \times 512$	0.496	31.71
	$28 \times 28 \times 512$	$3 \times 3 \times 512 \times 512$	74.35	58.29	$28 \times 28 \times 512$	0.496	31.73
	$12 \times 12 \times 512$	$3 \times 3 \times 512 \times 512$	74.35	14.57	$14 \times 14 \times 512$	0.496	31.73

The values reported in table 4.1, have been derived as follows. By introducing the following notation:

- A: number of cycles to fetch activations;
- B: number of cycles to fetch weights;
- C: number of cycles to fetch each $1 \times 1 \times M$ volume;
- D: number of cycles per MAC;
- E: number of cycles for write back;
- F: number of convolutional volumes to compute;
- G: filter shape not including the number of filters;
- H: number of filters to consider;
- I: number of filters SMAC-E is able to compute in parallel;

one can compute:

$$sing\_vol = A + B + C \times \left(D \times \frac{H}{I} + 1 \right) + E \tag{4.1}$$

$$total = sing\_vol \times F \tag{4.2}$$

$$th_{SMAC-E} = \frac{\#\,MAC\,op}{total} = \frac{G \times H \times F}{total} \tag{4.3}$$

$$th_{SMAC} = \frac{th_{SMAC-E}}{I} \tag{4.4}$$

Here, in computing A and B, only the latency introduced at the beginning of the computation has been considered, since in full regime it is possible to reduce the amount of latency to just C (this is taken care of by the +1 addend in 4.1) thanks to the serial approach.

For what concerns the second type of network, even though the architecture of the SMAC-Engine has not been tailored around the technique used by MobileNet to perform convolutions, namely depth-wise separable convolutions [36], it was interesting to analyze what this network is capable to achieve compared to VGG16, being this structure oriented towards a heavy reduction of the total amount of cycles to perform convolutions. During depth-wise (DW) convolutions a number of kernels corresponding to the number of input feature maps (channels) are exploited to reduce the spatial dimensions without modifying the depth dimension. Hence, $n_C^{[l-1]}$ kernels of size $f^{[l]} \times f^{[l]} \times 1$ will be used so that the dimension will shrink as follows:

$$n_W^{[l-1]} \times n_H^{[l-1]} \times n_C^{[l-1]} \xrightarrow{\text{depth-wise conv}} n_W^{[l]} \times n_H^{[l]} \times n_C^{[l-1]} \tag{4.5}$$

With $n_W^{[l]}$ and $n_H^{[l]}$ begin defined as in 2.29, with stride $s = 2$. After the DW convolution has been performed, a point-wise (PW) convolution follows to modify the depth dimensions while keeping the spatial dimensions unchanged. PW convolutions will exploit $1 \times 1 \times n_F^{[l]}$ kernels and this is where the name comes from. With PW convolutions the dimensions will change as follows:

$$n_W^{[l]} \times n_H^{[l]} \times n_C^{[l-1]} \xrightarrow{\text{point-wise conv}} n_W^{[l]} \times n_H^{[l]} \times n_F^{[l]} \tag{4.6}$$

As for the usage of the SMAC blocks, while VGG16 can maximize the usage of the architecture while computing an hidden layer, MobileNet can't maximize the architecture usage while performing a DW convolution. This is because each kernel works with different activations so there can't be any sharing across multiple SMAC blocks. In addition, one should actually consider the different fetching order for the activations while performing these two different kind of convolutions: whereas DW convolutions require to fetch activations belonging to the same feature map, PW convolutions require fetching activations having the same spatial coordinates and in the depth direction.

In the histograms in figures 4.12 and 4.13 and in table 4.2, the throughput and the number of cycles required to perform convolutions with such architecture are reported, even though the latter consideration regarding data fetching has not been taken into account to ease the analysis.

The throughput for DW convolutions does not actually change among different layers and is always obtained as the ratio:

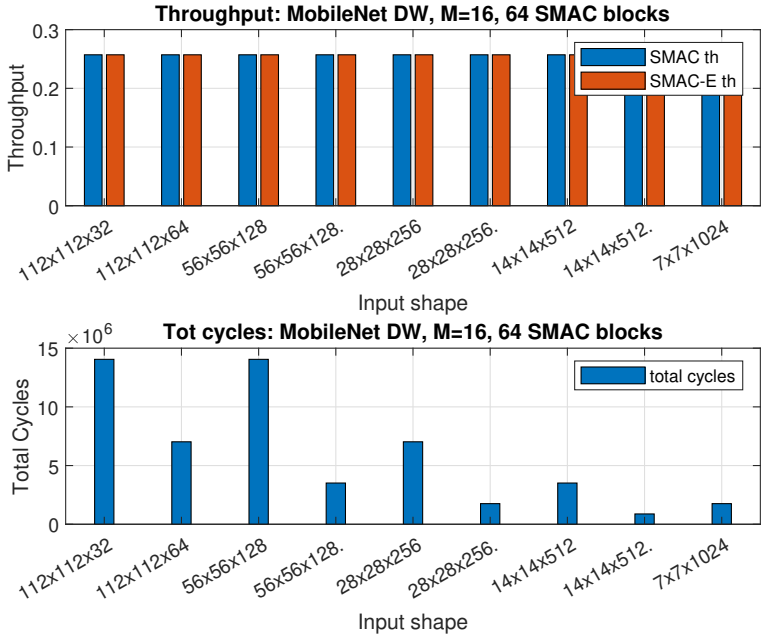


Figure 4.12. Throughput: single SMAC vs SMAC-Engine for DW convolution layers.

Table 4.2. Analysis on MobileNet, 64 SMAC blocks with $M = 16$.

Net	Input Shape	Filter Shape	Stride	# cycles			Output Shape	Throughput			
				DW [M]	PW [M]	tot [M]		SMAC [MAC/cycle]		SMAC [MAC/cycle]	
								DW	PW	DW	PW
MobileNet	$112 \times 112 \times 32$	$3 \times 3 \times 32$ dw	1	14.05	0	14.93	$112 \times 112 \times 32$	0.26	0	0.26	0
		$1 \times 1 \times 32 \times 64$	1	0	0.88		$112 \times 112 \times 64$	0	0.46	0	29.44
	$112 \times 112 \times 64$	$3 \times 3 \times 64$ dw	2	7.02	0	7.87	$56 \times 56 \times 64$	0.26	0	0.26	0
	$56 \times 56 \times 64$	$1 \times 1 \times 64 \times 128$	1	0	0.84		$56 \times 56 \times 128$	0	0.48	0	30.57
	$56 \times 56 \times 128$	$3 \times 3 \times 128$ dw	1	14.05	0	15.71		0.26	0	0.26	0
		$1 \times 1 \times 128 \times 128$	1	0	1.66			0	0.48	0	31.03
		$3 \times 3 \times 128$ dw	2	3.51	0	4.34	$28 \times 28 \times 128$	0.26	0	0.26	0
	$28 \times 28 \times 128$	$1 \times 1 \times 128 \times 256$	1	0	0.83		$28 \times 28 \times 256$	0	0.48	0	31.03
	$28 \times 28 \times 256$	$3 \times 3 \times 256$ dw	1	7.02	0	8.66		0.26	0	0.26	0
		$1 \times 1 \times 256 \times 256$	1	0	1.63			0	0.49	0	31.51
		$3 \times 3 \times 256$ dw	2	1.76	0	2.57	$14 \times 14 \times 256$	0.26	0	0.26	0
	$14 \times 14 \times 256$	$1 \times 1 \times 256 \times 512$	1	0	0.82		$14 \times 14 \times 512$	0	0.49	0	31.51
	$14 \times 14 \times 512$	$3 \times 3 \times 512$ dw	1	3.51	0	5.14		0.26	0	0.26	0
		$1 \times 1 \times 512 \times 512$	1	0	1.62			0	0.49	0	31.63
		$3 \times 3 \times 512$ dw	2	0.88	0	1.69	$7 \times 7 \times 512$	0.26	0	0.26	0
	$7 \times 7 \times 512$	$1 \times 1 \times 512 \times 1024$	1	0	0.81		$7 \times 7 \times 1024$	0	0.03	0	31.63
	$7 \times 7 \times 1024$	$3 \times 3 \times 1024$ dw	1	1.76	0	3.38		0.26	0	0.26	0
		$1 \times 1 \times 1024 \times 1024$	1	0	1.62			0	0.49	0	31.69

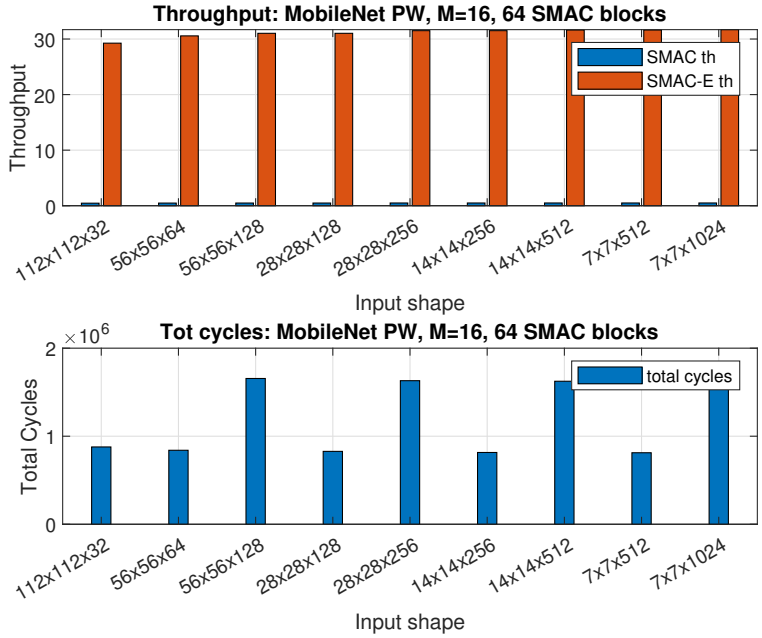


Figure 4.13. Throughput: single SMAC vs SMAC-Engine for PW convolution layers.

$$th_{DW} = \frac{9}{(A + B + D + E)} \tag{4.7}$$

where 9 is the number of MACs to perform for each 3×3 feature map. Here, only one SMAC block at a time can be used and operations on multiple SMAC blocks can not be performed. Furthermore, the employed SMAC block will be underutilized, since out of the $M = 16$ MACs it could perform, only 9 at a time will be computed. For the PW convolutions instead, the throughput can be easily derived by employing the same relations used above to derive the total cycles and throughput for the VGG16 analysis.

Interestingly, all the latency introduced during the DW convolution can be recovered during the PW convolution, where MobileNet can both maximize the usage of the SMAC blocks as well as finish its computation faster due to the unitary spatial dimensions required by PW convolutions. By doing so, considering only the CONV layers for both architectures, MobileNet will require a total number of cycles that is around one fifth of what VGG16 needs, which is what makes it particularly convenient for mobile platforms.

Another aspect to underline is the achievable throughput when the SMAC structures are replicated in the horizontal direction. Taking figure 4.9 as reference, replicating the structure horizontally allows to exploit some weight sharing that, at the cost of a slight increase in latency introduced to fetch the needed activations, would

further enhance throughput as well as energy efficiency, since weight fetching could be performed a lower number of times thus saving up on memory accesses and leading the structure towards a data flow that is closer to the Row Stationary one. However, this is possible only if the design is kept dense and no sparsity is taken into account. Differently, there would be no guarantee to keep coherence in the spatial position of the weights that are shared horizontally and thus one should fetch all the possible weights and later on dispatch, to each "column" of SMAC blocks, the needed weights based on the information coming, for instance, from a Sparsity Map.

Finally, considering the frequency estimation derived for the DP structure ($\approx$ 400 MHz) and the throughput information derived in this section ($\approx$ 30 MACs per cycle for VGG16), a rough performance comparison with Fulmine [5], in terms of GMACs per second (GMACs/s), could be derived. Indeed, SMAC-Engine allows to reach approximately a maximum of 12.69 GMACs/s. Fulmine, instead, is able to reach up to 6.35 GMAC/s. Even though this comparison was not particularly accurate, since performed on what the derived DP is able to achieve as an ideal standalone component, obtaining a result that was not too distant from the ones of a real working accelerator provided a positive starting idea about the SMAC-Engine capabilities.

4.3 The low-level Control Unit

After outlining the structure of the DP, a first low-level Control Unit (CU) has been realized, with the idea of allowing the DP to work independently of the address generation or the sparsity assumption and which could potentially allow to employ filters that are larger than 3×3, provided that the parallelism of the registers in the AC3 accumulator as well as of the programmable counters handling quantization and generating the done signal are properly adjusted.

The low-level CU is based on two main modules:

1. a 16 states Mealy Finite State Machine (FSM) able to generate control signals for the DP;

2. a module containing all the counters required to help the FSM in evolving through its states as well as status signals to send to a higher level control in charge of handling the address generation and other more complex tasks.

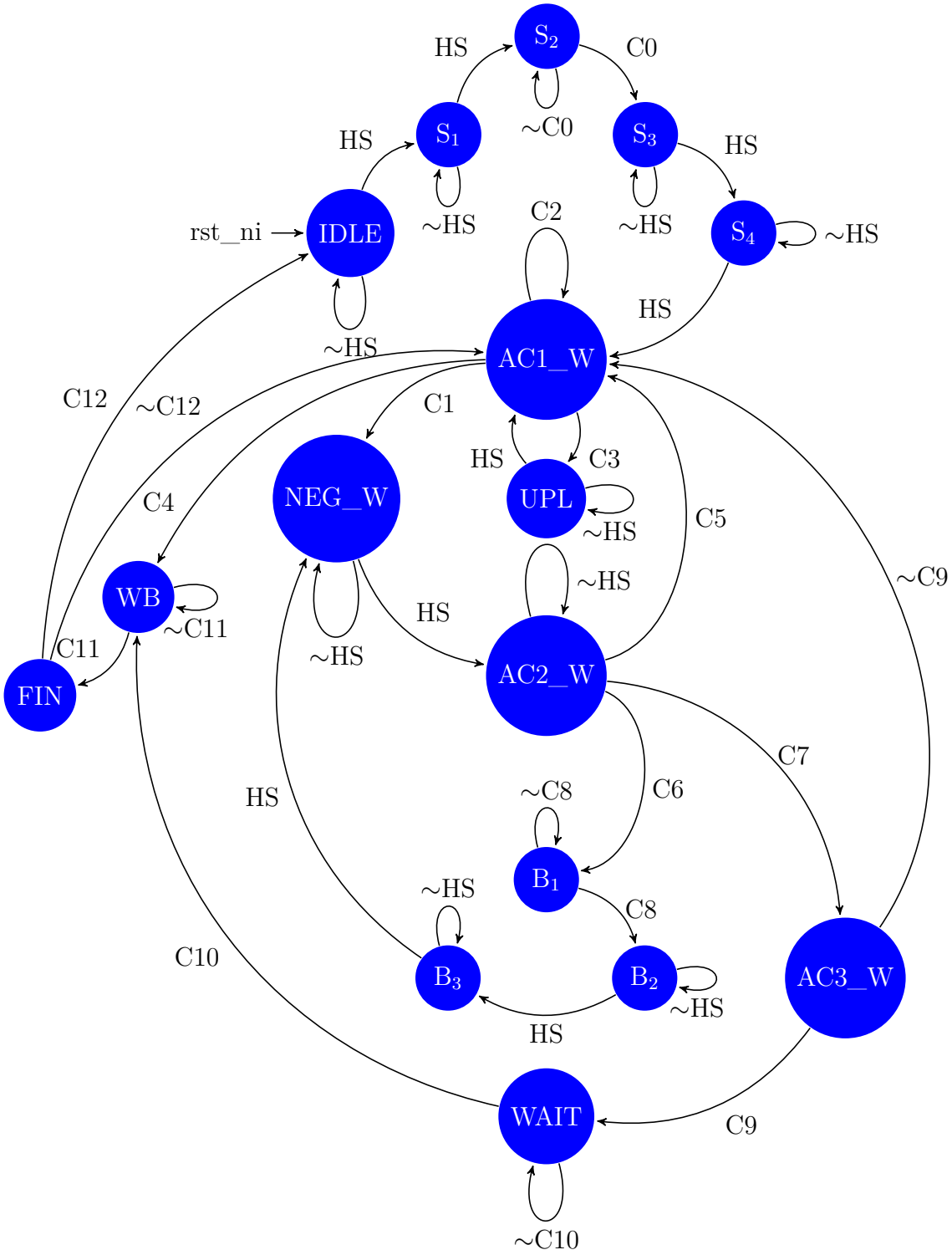


Figure 4.14. Low-level FSM state transition diagram.

4.3.1 The low-level FSM

The developed low-level FSM is a Mealy FSM based on 16 states. The reason why this has been developed as a Mealy FSM is to make it compliant with the handshake (HS) mechanism employed in the HWPE. Here, data at the input of the DP will be available only when a valid handshake with the streamer occurs and only then the FSM should be allowed to evolve through its states. Furthermore, the FSM states, depicted in figure 4.14 could be gathered into five groups:

1. IDLE: the FSM waits for a valid handshake to be asserted to start its job;
2. STARTUP: these states are labeled with an S and are needed at the startup to generate the controls to load the activations and weights registers before starting with the real computation;
3. COMPUTATION: these are the states that handle the writing in the accumulators (AC1_W, AC2_W, AC3_W), inverting the output of AC1 when dealing with the weight MSBs (NEG_W) as well as uploading either new weights or new activations when the serial shifting of the activation is completed (UPL);
4. BUBBLE HANDLING: these states are labeled with a B and are needed to handle the cycle that is lost when a new group of activations is fetched. This will introduce a bubble inside the structure that needs to be properly propagated without violating the internal state of the DP and this is what the bubble states have been introduced for.
5. WRITE BACK: these are the final three states, WAIT, WRITE BACK (WB) and FINISH (FIN) and are the states involved when the convolution is completed and the result needs to be written back to memory.

The state transition of this FSM is either controlled by the only HS signal, and in this case it is explicitly stated in the state transition graph in figure 4.14, or it is dependent on the HS and some further conditions. In the latter case the conditions are labeled with a C on the corresponding arc. The only states that will not be dependent on the valid HS condition will be the ones belonging to the WRITE BACK group. Indeed, during writing back no data can be fetched from memory, due to the non overlapping input and output structure and therefore there will be no need for the HS at the input to be valid.

Another aspect concerning this FSM is that it has been designed to operate in two possible modes:

1. Single mode: when working in single mode, whenever the AC3 accumulator has completed its task and computed the final partial sum, the counter in charge of performing quantization through serial shifting in the AC3 registers is triggered. During this phase, the FSM moves from the AC3_W state to

the WAIT state (condition C9 is asserted) to wait for the quantization to be done before moving to the WB state and finally write the output activations to memory. Here, some latency will inevitably be introduced while performing serial quantization, even though the overall amount is somewhat negligible, compared to the overall number of cycles required to complete an entire convolution volume.

2. Continuous mode: when working in continuous mode, whenever the AC3 accumulator has completed its task and computed the final partial sum, the counter in charge of performing the quantization through serial shifting in the AC3 registers is triggered. This time, however, instead of moving to the WAIT state and wait for the quantization to be completed, the latency introduced by quantization can be masked by the beginning of a new convolution and hence the FSM will move to the AC1_W state. Then, when quantization is completed, the same signal that is asserted to move from the WAIT to the WB state will be used to move from AC1_W to WB. Then, once WB is done, the FSM will be able to move back from FIN to AC1_W and start back from where it left. The only time when latency will be introduced due to quantization will be when the very last convolutional volume is computed and therefore there will be no new activations to fetch. Again, this will be reasonably negligible, compared to the cycles required to perform the entire convolution over the input volume.

Even though the support for continuous mode has been implemented at the low-level, to keep the integration in the HWPE simpler, this has not been taken care of in the top-level control, as it will be discussed in the next chapter.

For further details on how the FSM has been implemented it is suggested to take a look at the RTL in the *CTRL_FSM.sv* file.

4.3.2 The counters

The counters helping the FSM evolve through its states have been all gathered inside a top-level module named *CTRL_CNT_TOP.sv*. Out of these, the ones that need to be programmed by the top-level control are the following[6]:

- *CTRL_CNT_FIL_GROUP.sv*: this counter keeps track of how many groups of 64 filters SMAC-Engine will work with and helps the FSM decide whether the input data is a new set of activations or a new set of weight bits. For instance, for a layer where 64 filters are employed, this counter is programmed

[6]Here, the name of the RTL file will be provided to make it easier to keep track of which block the description is referring to.

to 1 whereas for a layer where 256 filters are employed, this counter is programmed to a value of 4. When dealing with a number of filters that is higher than 256, that is the maximum number of output activations SMAC-Engine is able to retain before writing back, the convolution operation will be split into multiple parts and for each of these this counter will need to be programmed accordingly;

- *CTRL_CNT_DONE.sv*: this counter keeps track of how many $1 \times 1 \times M$ volumes need to be computed in a $3 \times 3 \times n_C^{[l-1]}$ convolutional volume and will be incremented every time a partial sum is updated in the AC3 accumulator. Whenever the terminal count is reached, a signal triggering the execution of the quantization will be asserted;

- *CTRL_CNT_DONE_QUANT.sv*: this counter has to be programmed with the amount of shifting necessary to perform quantization. The shifting amount will vary depending on the number of $1 \times 1 \times M$ volumes that are computed for a specific layer, accordingly with the relation in 3.1;

- *CTRL_CNT_ReLU_MUX.sv*: this counter controls, for each SMAC block, to which of the four outputs coming out of the AC3 block the ReLU activation function should be applied. The value this counter is programmed with can be shared with *CTRL_CNT_FIL_GROUP*, since the number of filter groups to deal with will be coincident with the number of selection signal values the multiplexer, before the ReLU block, is expected to switch among.

- *CTRL_CNT_OUT_MUX.sv*: this counter is in charge of generating the selection signal for the output multiplexer in figure 4.9 and it has been made programmable to allow the structure to handle the case when less than 64 output activations are computed. For instance, when one deals with 32 filters, this counter would be set to 2 rather than 4, to stop the writing back operation accordingly. For all the other cases when the number of filters is 64 or higher, this will be fixed to 4.

- *CTRL_CNT_IN_VOL.sv*: this counter may be used to choose whether to let the FSM work in single or continuous mode. When this is programmed with a value of one, the FSM will work in single mode whereas when it is set to a value different than one, namely the number of $3 \times 3 \times n_c^{[l-1]}$ volumes that need to be computed for a full convolution of the input volume, it will work in continuous mode.

The remaining components in the top-level module are either shift registers or counters that do not require to be programmed, since they are strictly related to either the structure of the DP or the parallelism of data. Here, there are:

- *CTRL_SR_WE.sv*: this module is implemented as a serial to parallel 8 bits shift register where a "token bit" with value one is shifted across the flip-flops inside the register while the other entries are fixed to zero. Here, whenever the write enable signal is asserted, the token bit moves thus enabling only one, out of the eight weights registers at the input of the DP structure to sample data;

- *CTRL_CNT_Wbit.sv*: this module is implemented as a P_w bits shift register where a "token bit" with value one is shifted across the flip-flops inside the register while the other entries are fixed to zero. This is needed to keep track of what weight bits were the last to be sampled. The outputs of this module will be two signals stating whether the LSB, the MSB-1 or neither of the two were the last weight bits to be sampled thus helping the low-level FSM evolve through its states;

- *CTRL_CNT_AC1.sv*: this module is a counter that is used to keep track of how many times data has been written inside the AC1 accumulator. Here, every P_a times the write enable in the AC1 accumulator is valid, a terminal count signal is asserted, informing the low-level FSM that the data at the output of AC1 is valid and can be sampled inside the register at the input of the accumulator AC2

- *CTRL_CNT_AC2.sv*: this module is a counter that is used to keep track of how many times data has been written inside the AC2 accumulator. Here, every P_w times the write enable in the AC2 accumulator is valid, a terminal count signal is asserted, informing the low-level FSM that the data at the output of AC2 is valid and can be sampled by one of the registers in the AC3 accumulator.

Finally, some of the control signals generated by the FSM and the counters will be used to generate the update_sink and update_source signals, that are sent to the top-level FSM to let it evolve through the states, as it will be discussed in the following chapter.

4.4 Sparsity analsys

Going back to the sparsity assumption, a possible choice to deal with it could be to use a Sparsity Map as in NullHop [20] to both reduce the memory occupation and have an easy way to keep track of the positions without employing too complex encodings like CSR or CSC. However, the distribution of sparsity is nor uniform nor deterministic in an input volume, hence there is no certainty that, while fetching a predetermined number of activations, one is able to either completely fill with non

zero values the SMAC blocks or ends up with some spare values that need to be deployed at the next turn.

In particular, for what concerns the activations, the available bandwidth would allow to fetch, in a single cycle, four $1 \times 1 \times 16$ sparsity maps containing the positions of non-zero values. Considering the 50% sparsity assumption for the activations, this means that, if $M = 16$, it is statistically reasonable to take into account two $1 \times 1 \times 16$ sparsity maps at a time, as it is likely they will contain sixteen non zero-values positions. Then, in the next cycle the actual non zero values from the non zero value list (NZVL) could be fetched and deployed to the SMAC blocks. As for the weights, instead, fetching single bits out of a memory would hardly be possible. Therefore, a possible solution could be to fill a weight FIFO and then employ a scattered approach where, out of 128 bits separable per byte, every byte will statistically have 4 valid bits, taking into account the 50% weights sparsity assumption.

The non deterministic sparsity inevitably leads to adopting solutions that will hardly work at their best 100% of the times, both in terms of hardware resources usage and data scheduling. Here, a possible approach could be a best effort one, for instance assuming that things will work correctly 75% of the times. This means that it would be likely for the structure to be able to deploy $M = 16$ non-zero activations to the SMAC blocks but, whenever this is not possible due to an unfortunate data distribution inside an input volume, there should also be a recovery mechanism able to either deal with an excess or a lack of fetched data.

Due to the above discussed reasons, it was possible to conclude that the derived structure would not really be sparsity friendly for both activations and weights so such assumption should be dropped for one of the two. Here, the most likely choice would be to keep sparsity for the activations, due to the ReLU activation function and data distribution being an almost guarantee of ending up with 50% of the output activations trimmed off to zero.

After this final analysis on sparsity, the choice was either to realize a homemade scheduling to tackle activations sparsity, or realize something more complete that could be integrated in the PULP platform [34], but with a lower power performance due to the dense approach and non sparsity handling. Out of the two, though equivalently interesting, the second seemed better suited to derive a first benchmark out of the realized structure when implemented in a real system. Also, such structure could still be improved in the future if proven to be effective in “worst case" dense conditions. The process employed to integrate the SMAC-Engine inside a HWPE will be described in detail in the next chaper.

Chapter 5

Integration on PULP HWPE

After deriving the structure for the DP and managing the low-level control, the developed modules have been integrated inside the HPWE of a PULP system exploiting the provided open source IPs[1]. In this chapter, an introduction to how a HWPE is organized will be given as well as a detailed description regarding the integration of the SMAC-Engine in it. Whereas for the first most of the information has been obtained by referring to the provided documentation and the relative papers where the HWPEs have been employed [5, 7], the latter required gaining some experience with the platform, which will be shared and explained in the following pages.

5.1 The Hardware Processing Engine

As introduced in the previous chapters, the HWPEs have been developed as accelerators that live within the PULP system realized by Zurich ETH and the University of Bologna. As such, these accelerators are not meant to do all the job but rather to perform extremely well only parts of a job thus amplifying the system performance and its energy efficiency.

In figure 5.1, the structure of a HWPE is again reported. Here, it is possible to identify three different sub-modules, each in charge of a specific task:

1. a streamer module, needed to interface the HWPE internal engine with the Tightly Coupled Data Memory (TCDM), a 64 kB memory organized in eight

[1] https://github.com/pulp-platform/hwpe-mac-engine
https://github.com/pulp-platform/hwpe-stream
https://github.com/pulp-platform/hwpe-ctrl

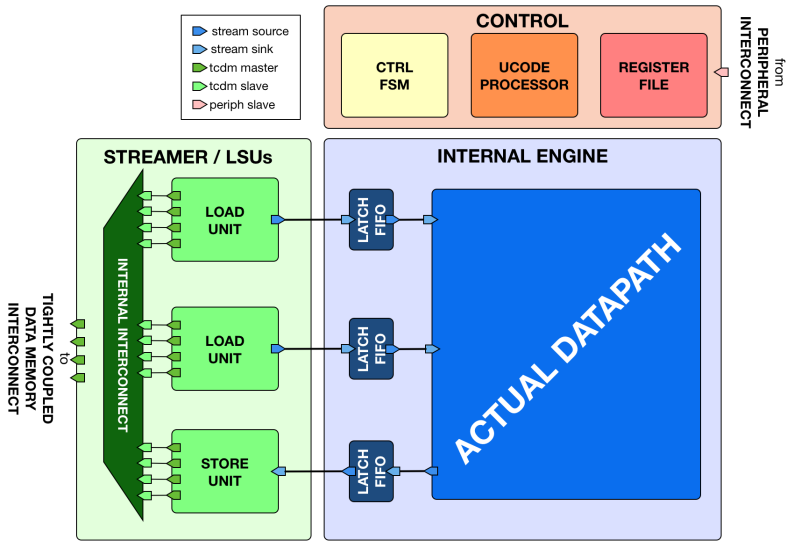


Figure 5.1. Example of Hardware Processing Engine (HWPE).

word-interleaved SRAM banks that allows a seamless and efficient communication among the resources belonging to the cluster;

2. a control module, consisting of a register file, a microcode processor and a control FSM that can be exploited to store parameters needed to let the system work properly, handle the loops required to perform a convolution (like in 2) or the address generation. This is connected to a peripheral interface that is used to program it;

3. an internal engine, where the actual DP will be placed.

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

5.1.1 The streamer

The streamer is a module acting as a transactor between internal engine and the memory system so that they can share data with each other. Furthermore, this module does not only transfer streams but is also capable of transforming data streams whose width is a multiple of 32 bits into byte-aligned accesses to either the TCDM memory or the internal engine.

The protocol on which this is based on is such that, when interfaced with the internal engine, the generated streams flow from a source to a sink direction using a valid/ready handshake similar to the one used by the AXI4-Stream protocol. As such, it is subject to the following rules:

1. a valid handshake can occur only in a cycle where both valid and ready signals are asserted;

2. the data (multiple of 32 bits) and strb (strobes indicating which bytes in the data payload have to be considered valid) can change their value when either the valid signal is deasserted or in the cycle after a valid handshake occurs, even if the valid signal is still asserted;

3. there can not be a combinational dependence between the assertion of the valid signal and the state of the ready signal to avoid deadlock conditions, even though the opposite could happen;

4. the valid signal can be deasserted only a cycle after a valid handshake has occurred, hence whatever data is produced by a source has to be consumed by the sink before the valid signal is deasserted.

Similarly to the interface with the internal engine, the interface with the TCDM is based on a TCDM protocol that connects a master to a slave using a request/grant handshake following these rules:

1. a valid handshake can occur only in a cycle where both request and grant signals are asserted;

2. the r_valid signal, from the slave to the master, must be asserted the cycle after a valid read handshake occurs and, in this cycle, the loaded data word r_data must be valid;

3. there can not be a combinational dependence between the assertion of the request and the state of the grant signal to avoid deadlock conditions, even though the opposite typically happens.

In other words, the streamer will be in charge of converting data between these two protocols so that it can be consumed or produced to the external shared memory[2].

Another aspect related to the streamer is that the streamer itself will have no notion about the address where data needs to be either fetched or written to memory. It will be a specific address generation module inside the provided IPs the one in charge of generating the required addresses based on some 3D geometrical space information it receives such as the width, height and depth (number of input features). Furthermore, some source_realign and stream_realign modules can be exploited to transform vectors starting from a non-word-aligned base into streams with a proper strobe to indicate what is the valid data to keep. The IPs also

[2]For further information concerning the signals employed by these protocols, it is suggested to consult the provided open source documentation in the "doc" directory of the hwpe-stream GitHub repository.

provide some merge and split modules that either allow streams to be longer than 32 bits, merging them together when loaded from the TCDM, or to be unpacked into groups of 32 bits when they need to be stored back to the TCDM. Finally, some multiplexer and demultiplexers are provided in the streamer IPs to properly funnel data where it is needed and FIFOs can be used to decouple the load and store operations to help mitigate the occurrence of non valid handshakes.

5.1.2 The control

The controller module embeds three different sub-modules:

1. a control FSM, which will need to be designed from scratch as it will be specific for the developed accelerator;

2. a memory-mapped register file implemented with latches to save area ad power, which includes two different set of registers:

   - generic registers (or job-independent): these are registers holding parameters that are supposed not to change during the execution of multiple jobs executed by the control. In these registers, the code required to program the loops implemented by the microcode processor can be stored;
   - job-dependent registers: these are registers holding parameters that can actually change at every new job. In other words, their content can be modified even when the HWPE is executing its tasks. These kind or registers may hold information such as the base addresses for the input activations, input weights and output activations, the loop ranges of the loops performed by the microcode processor, parameters needed by the engine and so on;

3. a microcode processor: this is a very simple processor capable of handling up to six nested loops without needing to hard-code them, but rather using a custom tiny ISA based on two "imperative" instructions, ADD (add/accumulate) and MV (move) and one "declarative" LOOP instruction. Moreover, this processor supports two different type of registers:

   - four R/W registers, used to store the offsets needed to compute new address generation bases;
   - twelve R/O, used to store parameters. also called mnemonics, coming from the register file such as loop ranges and iteration values and that are needed to update the values in the R/W registers.

   An imperative instruction is such that the result is always written back to R/W register, being it a R-R operation. To implement the behavior described, the microcode processor is organized with a set of four finite state machines in

charge of computing the address of the next micro-instruction to execute, its index within the current loop, the next iteration index of the current loop, and the next loop to be taken into account[7].

The final microcode that will be employed by the microprocessor can be derived by describing, in a high-level fashion, the iterative behavior of the developed accelerator in the YAML markup language. A sample of such code, named *code.yml* is provided in the hwpe-mac-engine GitHub repository, inside the ucode folder. Besides, here there is also a python script *ucode_compile.py* (currently running on version 2.7 and requiring the bitstring package) that, once executed, will return the microcode to fill the generic registers of the register file.

For what concerns the connections to the control, this module is the direct target of a slave port following a protocol, named PERIPH protocol, which is basically an extension of the TCDM protocol introduced in the streamer section and to which the id and r_id signals are added and used during load operations through the PERIPH interface.

5.1.3 The engine

The internal engine will contain all the blocks responsible to perform the operations the accelerator has been developed for. This means that this module will need to be designed from scratch, but keeping in mind that it will need to be compliant with the handshaking protocol with the streamer. For instance, taking the realized SMAC-Engine as an example, all its RTL description will be included in this module to let the HWPE be able to efficiently perform several MAC operations at once, making it particularly well suited to deal with CONV and FC layers that are typical of CNNs.

5.2 Integrating SMAC-Engine in a HWPE

Out of the open source IPs provided on the GitHub HWPE repositories, the ones that have been modified to fully integrate the developed SMAC-Engine in the HWPE are the ones contained in the hwpe-mac-engine/rtl repository. The other IPs, in the hwpe-stream and hwpe-ctrl repositories respectively, have been used as is to exploit both the interfaces and some RTL modules defined there. The following subsections will go through a detailed description about the changes made to each of the RTL files inside this repository compared to the practical example that was already provided.

5.2.1 mac_engine.sv

In this module, the DP provided in the example presented three 32 bits wide input sink ports (a_i, b_i and c_i) and one 32 bits wide output source port (d_o). However, since the designed SMAC-Engine is supposed to work with 128 bits wide data, two of the input ports (b_i and c_i) have been removed. Note that the "sink" and "source" interfaces are compliant to the streamer protocol defined in the previous section. Furthermore, the provided example propagates the handshake condition over the defined processes. However, for the SMAC-Engine integration, this has been limited at the boundaries of the architecture.

In particular, an input register for the incoming data has been defined before sending it to the actual SMAC-Engine DP module. Then, for the handshake condition at the input, the a_i.valid signal has been directly propagated both as a write enable signal to this input register as well as the core_stall_n signal that wakes up the low-level Mealy FSM described in the previous chapter and starts the computation. Besides, a design choice was to integrate all the designed modules, namely the DP *Data_Path_1x64.sv* and low-level control *CTRL_UNIT.sv*, inside the *mac_engine.sv*, in an attempt to ease interfacing with all the other modules in the repository. However, whereas for the provided example the control is minimal and leaving it in the engine module was a reasonable choice, probably for the designed SMAC-Engine control the most advantageous choice would be to either move it to another sub-module or transfer some of its complexity to the higher level control, in the *mac_fsm.sv*. The latter consideration may be something reserved for a future work.

Finally, two processes to generate the output valid signal as well as sample the output data in an output register have been defined before connecting them to the actual output d_o. Furthermore, the conditions on the generation of the ready signals have also been changed and adapted to the design, together with the rules defined at the bottom (this mainly required a change in the signal names, since the rules to follow do remain the same, as stated in the previous section).

5.2.2 mac_streamer.sv

In this module, the first change compared to the provided example was to extend the FIFO depth from 2 to 8, to better decouple the producer and the consumer and reduce the probability stall occurrence. Again, out of the four interface ports defined in the example, two have been removed (b_o and c_o) as they were not needed for the SMAC-Engine. Furthermore, the DATA_WIDTH of the remaining stream interfaces, namely the a_prefifo and the d_postfifo, have been extended from 32 bits to 128 bits, again to match the bandwidth of SMAC-Engine.

Another change, compared to the provided example, was to "virtually" extend the number of ports to the TCDM. These are considered "virtual" as it is like istantiating eight ports (eight in the SMAC-Engine case) but only four of these are

physically there: all these eight ports will think they are attached to the memory but they actually are not. This is necessary to handle 128 bits stream at the input and at the output. Indeed, a TCDM multiplexer can then be used to funnel more input "virtual" TCDM channels (eight) into a smaller set of master ports (four). Hence, together with the definition of a virtual_tcdm interface, a hwpe_stream_tcdm_mux has been allocated to handle this.

After defining the virtual_tcdm interface, the corresponding streams TCDM ports, both for the source i_a_source and for the sink i_d_sink, have been connected to the input of the multiplexer instantiated above and their DATA_WIDTH again extended from 32 bits to 128 bits while the unnecessary source streams, for b_o and c_o respectively, have been removed. Finally, for both streams' FIFOs i_a_fifo and i_d_fifo, the DATA_WIDTH has again been extended to 128, the FIFO_DEPTH to 8 and the parameter LATCH_FIFO has been set to 0, as the latter was not needed.

5.2.3 mac_package.sv

This module is a package where all the parameters and structures required by several modules in the HWPE IPs are defined. Both these parameters and structures are helpful to avoid redefining them internally to each module, as well as to make the definition of each of these modules look cleaner, especially when they are interfaced with other components.

First of all, the MAC_CNT_LEN definition has been left unchanged, as this will be a parameter employed by the *mac_ctrl.sv* module to define the dimension of the transaction size for both the weights and output activations. The transaction size is a quantity that is sent to the *mac_fsm.sv* and that helps this high-level FSM generating the correct information to send to the address generators in the streamer.

The following parameters define the job-dependent register file addresses (or indexes to their content) and have been changed to match the quantities needed by the SMAC-Engine. Some of the job-dependent registers in the register file have been thought to store multiple parameters, since their parallelism is usually lower than the 32 bits words these registers can retain. In particular, these addresses have been defined as follows:

- MAC_REG_X_ADDR: this is the address to a register in the register file storing the base address for the input activations in the TCDM;
- MAC_REG_W_ADDR: this is the address to a register in the register file storing the base address for the input weights in the TCDM;
- MAC_REG_Y_ADDR: this is the address to a register in the register file storing the base address for the output activations in the TCDM;

- MAC_REG_NIF: this is the address to a register in the register file storing the offset required to move, in the TCDM memory, of a quantity equal to the number of input features $n_C^{[l-1]}$;
- MAC_REG_NOF: this is the address to a register in the register file storing the offset required to move, in the TCDM memory, of a quantity equal to the number of output features $n_C^{[l]}$;
- MAC_REG_IW_X_NIF: this is the address to a register in the register file storing the offset required to move, in the TCDM memory, of a quantity equal to the product between the input volume width $n_W^{[l-1]}$ and the number of input features $n_C^{[l-1]}$;
- MAC_REG_NFA: this is the address to a register in the register file storing the offset required to move, in the TCDM memory, of a quantity equal to the number of fetched input activations;
- MAC_REG_NWA: this is the address to a register in the register file storing the offset required to move, in the TCDM memory, of a quantity equal to the number of written output activations;
- MAC_REG_ZERO: this is the address to a register in the register file storing the zero value, which is needed in the loop execution to reset the weights' address offset to the base value to restart convolution with a new convolutional volume;
- MAC_REG_NFW: this is the address to a register in the register file storing the offset required to move, in the TCDM memory, of a quantity equal to the number of fetched weights;
- MAC_REG_LOOP1_LOOP0: this is the address to a register in the register file storing the loop ranges for first two innermost loops;
- MAC_REG_LOOP3_LOOP2: this is the address to a register in the register file storing the loop ranges for the third and fourth loops;
- MAC_REG_LOOP5_LOOP4: this is the address to a register in the register file storing the loop ranges for first two outermost loops;
- MAC_REG_CNT_PROG1: this is the address to a register in the register file storing the values to program the low-level control counters;
- MAC_REG_CNT_PROG2: this is the address to a register in the register file storing the values to program the low-level control counters;

- MAC_REG_ITER_LEN_WEI_OUT: this is the address to a register in the register file storing the transaction size, which is an info concerning how many packets of 128 bits one expect to send/fetch to/from memory. This is not used for input activations as they are fetched one packet at a time and so their transaction size can be fixed to a value of one.

Following the addresses to the register file, the addresses to the register file (or indexes to their content) in the microcode processor are defined. Here, three out of the four R/W register addresses and seven out of the twelve R/O register addresses have been specified. These offset indexes should be aligned to the microcode compiler as well as match the addresses provided while writing the high-level YAML code "code.yml". In particular, here are defined:

- The R/W registers addresses/indexes:
  - MAC_UCODE_W_OFFS;
  - MAC_UCODE_X_OFFS;
  - MAC_UCODE_Y_OFFS;
- The R/O registers addresses/indexes:
  - MAC_UCODE_MNEM_NIF;
  - MAC_UCODE_MNEM_NOF;
  - MAC_UCODE_MNEM_IW_X_NIF;
  - MAC_UCODE_MNEM_NFA;
  - MAC_UCODE_MNEM_NWA;
  - MAC_UCODE_MNEM_ZERO;
  - MAC_UCODE_MNEM_NFW;

  Note that to all these, a value of three is subtracted. This is to match the index definition with the "ucode_registers_read" logic type vector in *mac_ctrl.sv*;

In this module, the definition for the parameters employed in the SMAC-Engine structure has been added, namely the activations' parallelism P_a, the weights' parallelism P_w, the number of operands M, the bandwidth BW, the maximum number of possible $1 \times 1 \times M$ volumes MNO during a CONV or FC layer for a VGG16 network (which is $3 \times 3 \times 512/M$ and finally a maximum number of convolutional volumes MNV to compute for the reference network (this has been defined to support the continuous mode, even though the counter associated to this parameter can be programmed to one to let the low-level control work in single mode).

Here, there are also two structures related to the control and status signal required by the *mac_engine.sv* module. In particular, the ctrl_engine_t structure

has been modified to contain the signals that are necessary to program the programmable counters in the low-level control whereas the flags_engine_t has been modified to contain two status signals, namely update_in and update_out, that are required by the top-level FSM to evolve though its states.

As for the streamer related types, ctrl_streamer_t and flags_streamer_t, the only change has been to remove the definition of the control and status signals that are not needed by the SMAC-Engine, hence the signals related to the b_source and c_source streams.

The content of the ctrl_fsm_t structure, instead, has been substituted to include the transaction sizes len_wei and len_out that are taken from the register file and sent to the top-level FSM to correctly generate the addresses where data needs to be either fetched or stored.

Finally, this package also includes a structure state_fsm_t defining, as the name suggests, the states for the top-level FSM in *mac_fsm.sv*.

5.2.4 mac_fsm.sv

The example provided for the FSM in the GitHub repository has been changed and some states have been added to decouple the data fetching process from the data storing process. This has been achieved by defining two different UPDATEIDX states, one for the input and one for the output data (figure 5.2). This is due to the chosen output stationary data flow. Differently from the provided example, where every time some data is fetched, some computation is performed and the result written back to memory, the SMAC-Engine will be internally accumulating the partial sums until the convolution operation is completed and so data fetching will happen at a different time with respect to data storing, as they will not be overlapped. Moreover, the ctrl_engine_o signal has been removed and the control signals going to the SMAC-Engine have been moved to the *mac_ctrl.sv* module[3].

In the combinational process of this FSM, which is also in charge of handling the state evolution, the b and c streams definitions have been removed since they are not used for SMAC-Engine. Then, the trans_size, line_lenght and base_addr of the remaining streams, a and d, have been modified to match the parameters definitions in the *mac_package.sv*. Another aspect that may require some clarification is the one concerning the ready_start flags and the req_start control signals: the ready_start is a streamer signal that will go low as soon as a req_start is received

[3]This is not necessarily the best choice to make, as all depends on how both the low-level and the top-level controls have been conceived. In this case, this could be done because the low-level control is actually more complex than the top-level one, which is again not really the common hierarchical approach one would expect and could definitely be improved in some future work. For instance, some of the complexity may be moved to the top-level FSM thus simplifying the low-level control and therefore requiring again the usage of the ctrl_engine_o signal.

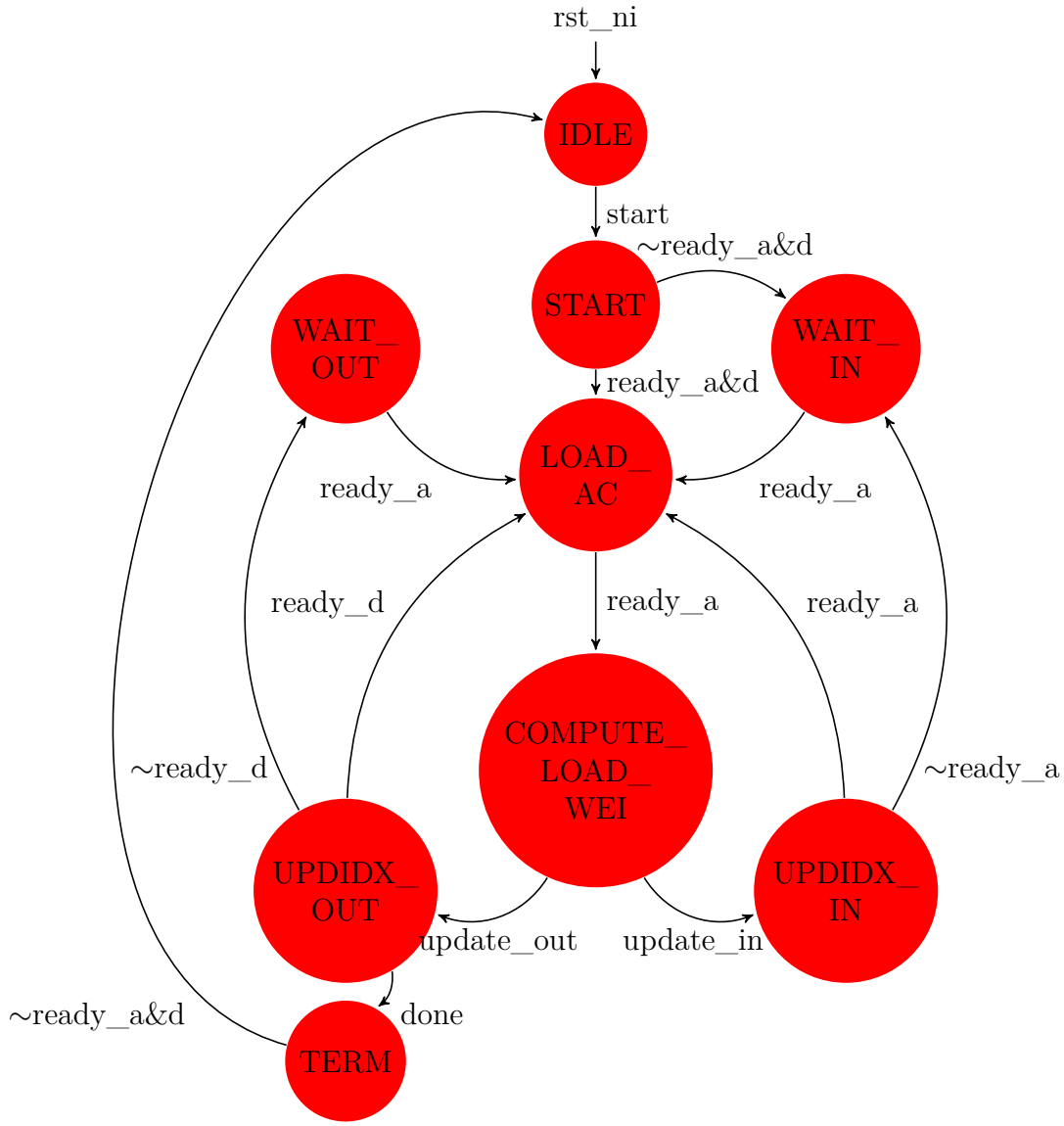


Figure 5.2. Top-level FSM state diagram.

and it will be asserted again as soon as the called streamer ends its job, that is it reaches the trans_size. So, this should be taken into account for the proper evolution of the FSM through its states. Furthermore, even though the input data shares the same streamer a, fetching activations requires different addresses with respect to the ones required by the weights. This is why, in the COMPUTE_LOAD_WEI state, all the parameters defining the address for the streamer a are adjusted to match the values required to fetch weights.

Finally, this FSM is actually not yet optimized for the execution of the SMAC-Engine in continuous mode and therefore some further changes, such as the introduction of properly delayed flags, may be introduced to support this working condition[4].

5.2.5 mac_ctrl.sv

In this module, the static_reg definitions have been changed to match the values that are supposed to be stored in the register file (the job-dependent side). After doing so, to each of the defined static_reg the corresponding value inside the register file (the hwpe_params) has been assigned.

Then, the static_reg containing the loop ranges values have been concatenated to the ucode_flat definition[5], the ones containing the values to program the low-level control counters have been directly sent as ctrl_engine_o signals to the *mac_engine.sv* module and the remaining ones, being the mnemonics necessary to the microprocessor to correctly perform its job, have been assigned to the corresponding R/O registers in its register file. Lastly, the transaction sizes to send to the top-level FSM have been set in the combinational process at the bottom.

Even though the values to program the generic registers of the register file are passed to the accelerator control at the test bench level (since they are supposed to come directly from a PULP core), it is worth mentioning the procedure that has been followed to write down both the microcode behavior in a high-level fashion, through the YAML markup language, and to use the provided python code to generate the actual code to place inside the generic registers.

In Listing 5.1, the written YAML code (adapted from the *code.yml* file sample in the ucode folder of the hwpe-mac-engine repository) describing the microprocessor behavior is reported. Here, after writing down the mnemonics with their corresponding indexes, which will need to be the same as the ones defined inside the *mac_package.sv*, the actual microcode behavior is shown, which basically resembles the algorithm in 2. In particular, six loops have been written with their corresponding label, starting from the innermost loop to the outermost one. In every loop,

[4]The synthesis results have shown the presence of a combinational path from the register file to the base address generation. This is due to the FSM begin a Mealy FSM as well as the presence of eight muxing levels, probably due to the sum operations with the indexes, which may be too complicated for the synthesis tool to handle. A better approach could be to use two signals, source_base_addr_x and source_base_addr_W, whose value is determined separately and stored inside a flip-flop. In this way, the FSM would just select one or the other. Alternatively, the FSM could be turned into a Moore FSM achieving the same results, since in both cases a further bit is required.

[5]Here, the UCODE_HARDWIRED parameter has been left to 0, since the microcode that is necessary to program the microprocessor has been assumed to be stored inside the generic registers of the register file, rather than hardwired to the accelerator itself.

ADD or MV operations are performed. For each operation, a always indicates the destination where the result will be written (which necessarily needs to be a R/W register of the microprocessor) whereas b may either be an offset quantity coming from the R/W registers or from the R/O registers. Moreover, some comments have been written to help the reader understand what happens in each loop.

In the third and fourth loop, it is possible to notice the presence of NOP instructions. These instructions, as the name suggests, are not actually needed in performing the loop, but they have still been included due to the presence of a bug in the current version of the hwpe-ctrl IPs which does not allow to have single instructions inside a loop. Indeed, if a single instruction is written, the operations inside the loop will not be executed thus leading to an unexpected behavior of the accelerator.

```
# LOOP0 loop_stream_inner: for k_in in range(0,nif/nfa)
# LOOP1 loop_filter_x    : for j in range(0,f)
# LOOP2 loop_filter_y    : for i in range(0,f)
# LOOP3 loop_stream_outer: for k_out in range(0,nof/nfW)
# LOOP4 loop_spatial_x   : for w_out in range(0,n_W)
# LOOP5 loop_spatial_y   : for h_out in range(0,n_H)

# mnemonics to simplify microcode writing
mnemonics:
#needed to update the weight index
W:          0
#needed to update input activations index in a conv volume
x:          1
#needed to update the output activations index
y:          2
#needed to update the input index in the input volume
x_maj:      3
#number of input features, this is basically n_C^[l-1]
nif:        4
#number of output features, this is basically n_F^[l]
nof:        5
#stride to move one pixel down in a conv or input volume
iw_X_nif:   6
#number of fetched activations per cycle
nfa:        7
#number of activations written back to memory
nwa:        8
#zero value to erease W value before reuse
zero:       9
#number of fetched weights for each group of M activations
nfW:        10

# actual microcode: loop order is from the inner to the outermost
code:
loop_stream_inner: #for k_in in range(0,nif/nfa)
- { op : add, a : x, b : nfa, } # move to next subset of input features
- { op : add, a : W, b : nfW, } # move to next subset of input weights
loop_filter_x: #for j in range(0,f)
- { op : add, a : x, b : nfa, } # move one pixel to the right
- { op : add, a : W, b : nfW, } # move filter index one position right
loop_filter_y: #for i in range(0,f)
- { op : add, a : x, b : iw_X_nif, } # move one pixel down
- { op : add, a : W, b : nfW,       } # NOP
loop_stream_outer: #for k_out in range(0,nof/nwa)
- { op : add, a : y, b : nwa, } # move to next subset of output features
```

```
- { op : mv,   a : y, b : y,    } # NOP
loop_spatial_x: #for w_out in range(0,n_W)
- { op : add, a : y,      b : nof,    } # move one pixel to the right
- { op : add, a : x_maj, b : nif,    } # move one pixel to the right
- { op : mv,   a : W,      b : zero,   } # return to first weight
- { op : mv,   a : x,      b : x_maj, } # reload x to align with new y
loop_spatial_y: #for h_out in range(0,n_H)
- { op : add, a : y,      b : nof, } # move one pixel down
- { op : add, a : x_maj, b : nif, } # move one pixel down
```

Listing 5.1. YAML code describing the microprocessor behavior.

Once the YAML code has been written, it is possible to run the *ucode_compile.py* script to derive the actual code to place inside the generic registers in the register file. Currently, this script requires Python 2.7 version together with pyyaml and bitstring packages. The latter can be installed by issuing the following commands in the Python shell:

```
pip install pyyaml
pip install bitstring
```

Then, after running the script, this will return something like the following:

```
ucode bytecode: 176'h0000000046788c08c0546488b12205c2780909205c28
   ucode loops: 48'h5a3c2a211202
```

that are the bytecode and loops to place inside the generic registers.

5.2.6 mac_top.sv

The *mac_top.sv* module has been the last to be modified. Here, the unused streams have again been removed from the i_engine and i_streamer modules declaration and the DATA_WIDTH of the input and output streams have been adjusted to the bandwidth of the SMAC-Engine. Finally, to the enable signal a value of one has been assigned. This module did not require any further adjustment. However, for simulation purposes, since the simulator is not particularly fond of the interfaces defined and employed by the above mentioned modules, a further module, named *mac_top_wrap.sv* has been provided in the wrap directory of the GitHub repository. The latter module will be the one to be instantiated inside the test bench to perform the needed simulations.

Chapter 6

Results analysis

In this chapter, the procedure followed to set up and use the employed software tools will be explained and the obtained results reported. In particular, the used software were:

- QuestaSim 10.6c® for the logic simulation;
- MATLAB® to verify the correctness of the simulation result;
- Synopsys Design Compiler® for the logic synthesis;
- Innovus 17.11® for the place and route;

6.1 Setting up the test bench

To properly set up the test bench, the tool chain for the accelerator standalone usage provided in the hwpe-mac-engine GitHub repository has been first downloaded by issuing the command:

```
git clone https://github.com/pulp-platform/hwpe-mac-engine.git -b standalone
```

After downloading the hwpe-mac-engine IPs repository, the files in the "rtl" folder have been substituted with the ones described in the previous chapter and adapted to work with the SMAC-Engine. Furthermore, all the RTL files necessary to the SMAC-Engine have also been added to this repository. Then, in the same directory, the *src_files.yml* file has been modified with a list of all the RTL files added to the "rtl" folder, including the test bench files *tb_acc_top.sv* and *tb_dummy_memory.sv*.

As far as the test bench files are concerned, the *tb_acc_top.sv* includes the necessary port maps for the test as well as defines the values to store inside the generic and job-dependent registers of the register file in the control sub-module of the HWPE. In the generic registers, the code generated by running the *ucode_compile.py* script with the written *code.yml* has been assigned as follows:

```
acc_set_generic_register(7, 0);
acc_set_generic_register(6, 32'h62443222); // loops [47:16]
acc_set_generic_register(5, 32'h12020000); // loops [15:0], bytecode [175:160]
acc_set_generic_register(4, 32'h02324450); // bytecode [159:128]
acc_set_generic_register(3, 32'h46026324); // bytecode [127:96]
acc_set_generic_register(2, 32'h45085122); // bytecode [95:64]
acc_set_generic_register(1, 32'h05426815); // bytecode [63:32]
acc_set_generic_register(0, 32'h09e05427); // bytecode [31:0]
```

Moreover, here a *tb_acc_common.sv* file[1] is included where there are several tasks necessary to the test bench to correctly work, one of which is the task generating the clock signal, whose timing parameters are set at the beginning.

The *tb_dummy_memory.sv* instead, whose RTL file can be found in the hwpe-stream IPs, inside the "tb" folder, has also been modified by:

- adding the following line to the second sequential process:

  ```
  process::self().srandom(32'hacab6143)
  ```

  which introduces a seed in the random generation of the data that will be stored inside the dummy memory, thus allowing to make the simulation repeatable because the generated data will always be the same.

- employing the system I/O tasks $fopen, $fwrite and $fclose to write into a text file *dummy_memory_content.txt* the values generated in the dummy memory so that they could later be loaded and used by a MATLAB® script to check the correctness of the result.

Going back to the main directory "hwpe-mac-engine", after opening a Python 3® environment and assuring that the pyyaml package was installed, the *update-ips* script has been executed so that all the needed IPs for the standalone usage could be downloaded and placed in the "ips" folder.

Before going on with the simulation, it is worth mentioning how the dummy memory was supposed to be organized. In particular, the *tb_dummy_memory.sv* module randomly generates the content of a memory whose words are on 32 bits. Hence, if the memory is organized into 32 bits words, this means that every row is separated from the other by an offset of four, since every word contains four bytes. Hence, fetching 128 bits data will require an offset that is four times as large, that is

[1] This file does not need to be added to the *src_files.yml*.

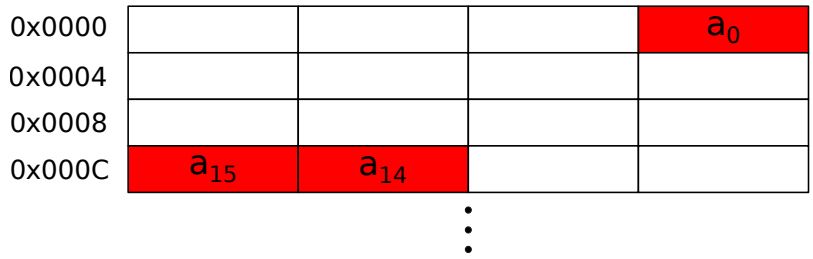


Figure 6.1. Activations memory organization.

16, in decimal, or 10 in hexadecimal notation. This was an important note to keep in mind while setting the correct mnemonics values in the job-dependent registers of the register file. The activations have been thought to be fetched sixteen at a time. This means that, since their parallelism is on $P_a = 8$ bits, every time sixteen activations are fetched, the offset pointing to the TCDM memory needs to be moved of a quantity equal to 10 in hexadecimal, which is actually the value that has been assigned to the variable "nfa". In figure 6.1, there is a visual representation of how activations should be organized in memory.

As for the weights, instead, these have been thought to be organized in such a way that each row of the memory can be divided into two parts of 16 bits, each feeding a specific SMAC block. Hence, this means that in order to feed 64 SMAC blocks, an offset of 0x0200, in hexadecimal notation, will be required. This quantity is also equal to the offset parameter "nfW" as long as the accelerator is working with 64 filters (as shown in figure 6.2). Differently, this will need to be varied accordingly. For instance, when 128 filters are used, the offset will be equal to 0x0400: the first group of rows, up to the base_addr + 0x01FC will store the weights' bits for the first group of 64 filters whereas the second group of rows, from base_addr + 0x0200 up to base_addr + 0x031FC will store the remaining 64 filters weights' bits. Finally, to actually complete a MAC operation, weights' bits will need to be fetched a number of times equal the weights parallelism $P_w = 4$.

As for the output activations, sixteen of these will be stored per cycle, similarly to what has been explained for the input activations, with the only difference that the offset will vary depending on the number of write backs that are performed from the SMAC-Engine towards the TCDM memory.

Having chosen how the memory should be organized, the next step was to set up the simulation with Questasim 10.6c®. Hence, after going to the "sim" directory and initializing the software tool issuing:

source /software/scripts/init_questa10.6c

the *Makefile* provided has been executed as follows:

make clean lib build

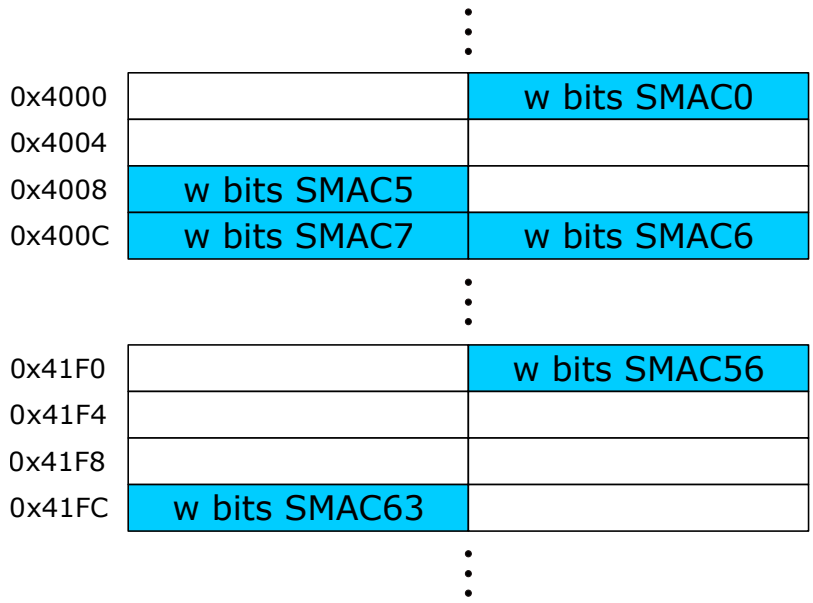


Figure 6.2. Weights memory organization.

to respectively:

- erase the content of the current work directory and IPs libraries;
- recreate the working libraries;
- building the content of the "ips" folder as well as the "rtl" folder;

Once these steps have been followed, the simulator software could finally be launched and the simulation executed with the following command:

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

Here, the optimizations have been disabled due to some discrepancies in the expected result when they were enabled during simulation.

6.2 Simulation results

The simulation performed with Questasim 10.6c® has been carried out on a $3 \times 3 \times 128$ convolutional input volume, employing 128 filters with the same size. The choice was to keep the sizes not too large to ease the debugging processes. First of all, the waveforms related to the input stream "a" have been checked to verify their correctness.

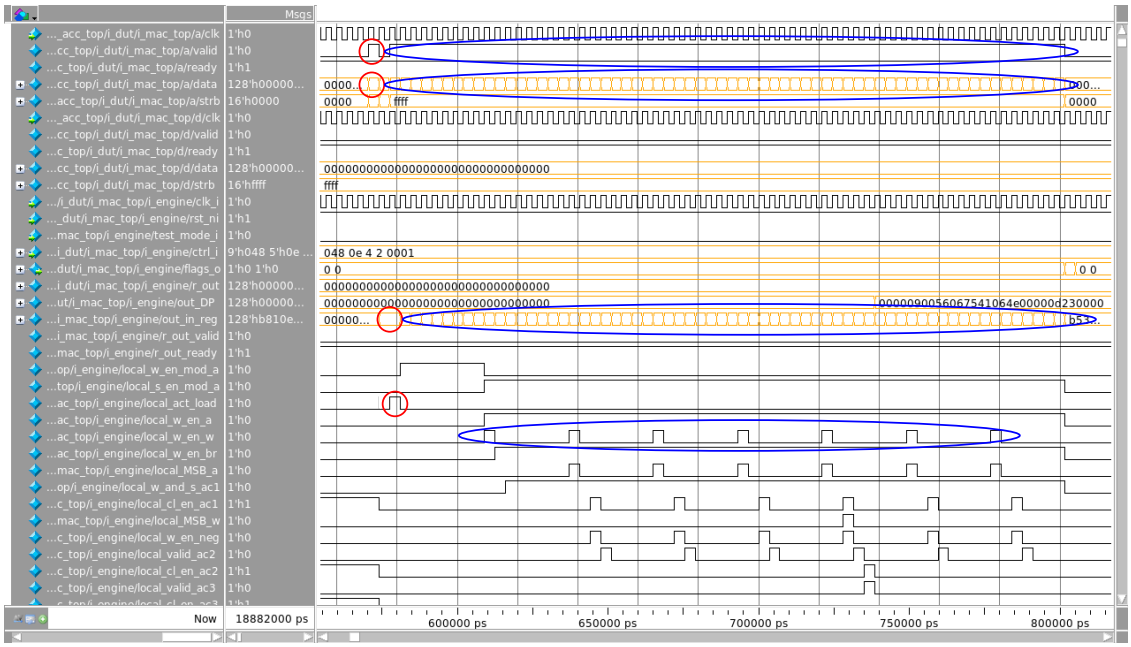


Figure 6.3. Input stream “a” waveforms.

From figure 6.3, it is possible to notice how the valid handshake signal behaves. As expected, it is valid for one cycle when a group of sixteen input activations are fetched (circled in red), then deasserted and asserted again when the weights need to be gathered from the TCDM memory (circled in blue). Furthermore, the values related to the actual data have been compared to the ones randomly generated and stored inside the dummy memory. This was relatively easy to do, thanks to the text file generated by the system I/O tasks added to the *tb_dummy_memory.sv* file. Moreover, it is possible to see how the fetched values are sampled by the register at the input of the DP and later sent to either the register storing the activations or to the ones storing weights' values.

In addition, from the same figure it is possible to see how the input activations are sampled by the internal SMAC-Engine structure when the act_load signal is asserted whereas the weights are sampled inside the corresponding flip-flops of every SMAC block every eight cycles the input data weights were valid and hence valid data has been correctly stored inside the weights' registers at the boundary of the DP structure. The latter is reflected by the assertion of the w_en_w signal. Furthermore, it is possible to see how some weights' bits are fetched but never sampled by the SMAC blocks flip-flops due to the DP being stalled before this happens. Of course, these values are not actually lost, since they are saved inside the registers at the boundary of the DP structure and will be correctly loaded as soon as the DP and the low level FSM return active.

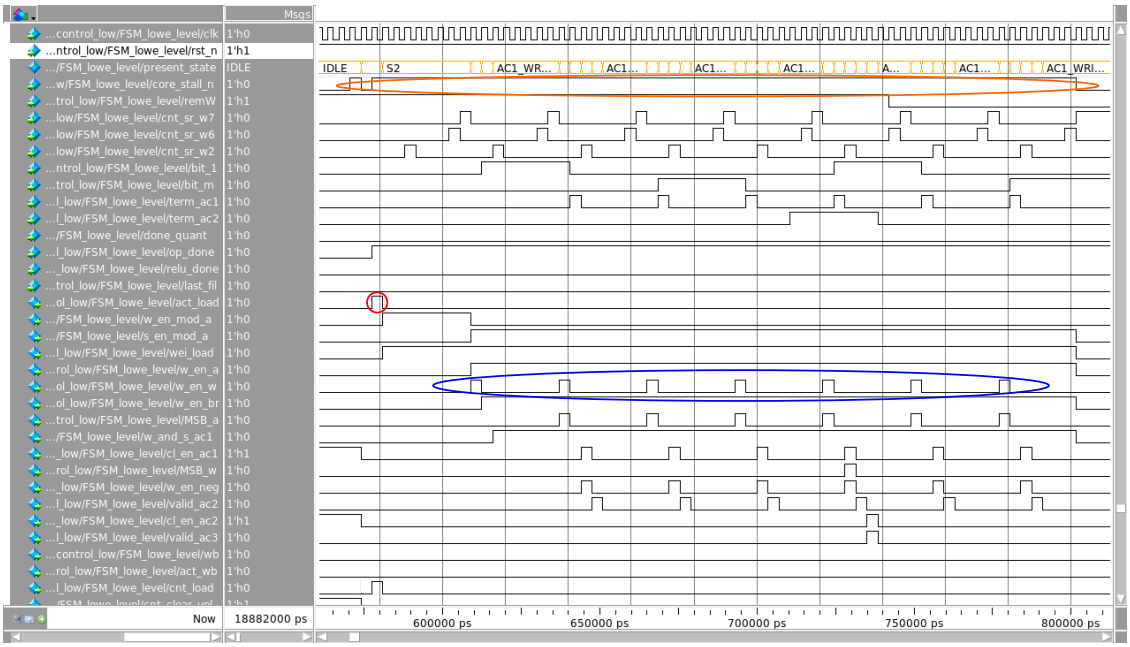


Figure 6.4. Low level FSM state evolution and control signals.

The above mentioned behavior can also be observed in figure 6.4. Here, the evolution of the FSM is shown as well as its dependence on the valid handshake condition (circled in orange), which is reflected by the assertion of the core_stall_n signal. Again, it is possible to see the presence of the act_load and w_en_w signals being correctly asserted when data needs to be sampled inside the DP registers.

The example depicted in figure 6.4 actually shows what happens during the start-up of the FSM. In fact, when the FSM returns active, it will start cycling through the same states until the entire convolution operation is completed and hence data needs first to be quantized, going to the WAIT_END state, and finally written back to memory going to the WRITE_BACK state.

For what concerns the "d" output stream, it needs to be active only when data needs to be written back to memory after a convolution operation has been completed. This is shown in figure 6.5, where it is possible to notice the last group of input activations and weights being correctly fetched as well as the output results, after some cycles needed to apply quantization, setting the "d" stream data to a value different than zero (circled in green) and the corresponding valid signal being asserted while this happens.

Finally, to check whether the outputs of the SMAC blocks were actually correct, the generated text file containing the values generated and stored inside the dummy memory has been imported in MATLAB® and a script simulating the accelerator behavior, named *simulate_accelerator_behavior.m*, has been written. Moreover, as

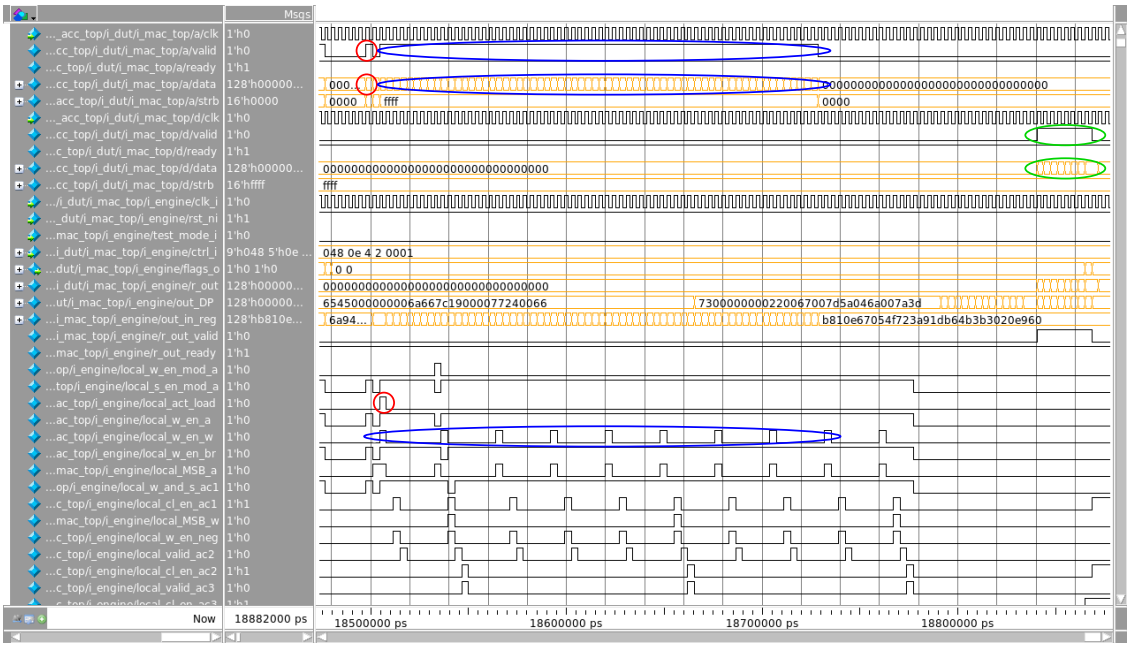


Figure 6.5. Input stream “a” and output stream “d” waveforms..

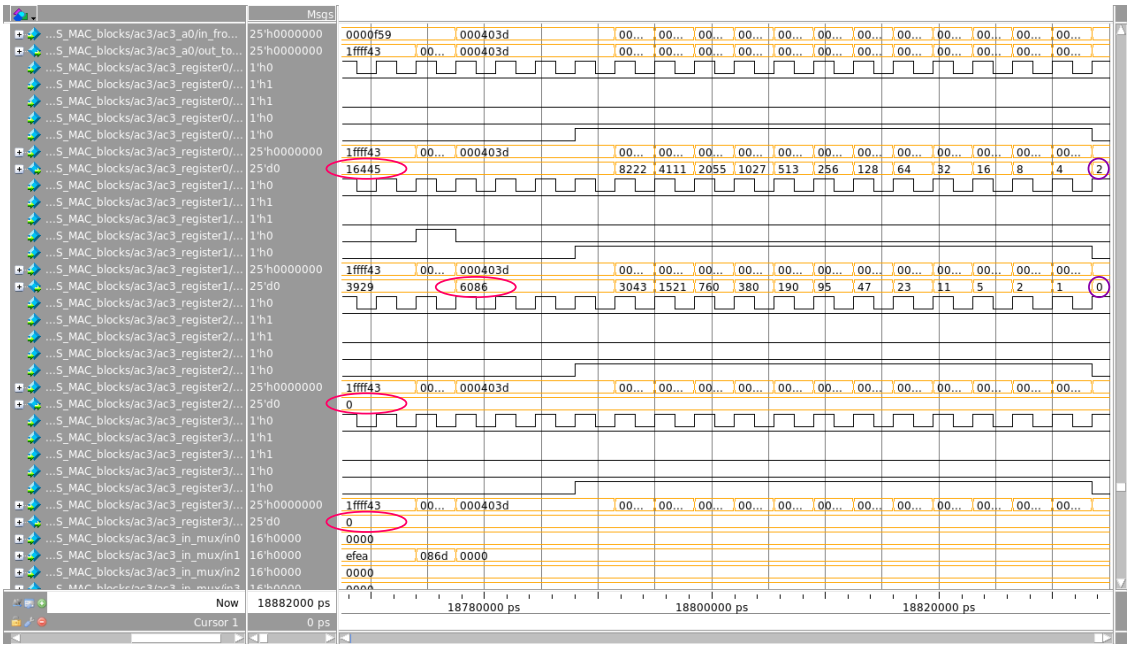


Figure 6.6. Data waveforms at the output of the SMAC0 block.

it is possible to see from the waveforms reported in figure 6.6, only two out of the four registers inside the AC3 accumulator are actually written (circled in magenta), since the adopted example works with 128 filters, while the other registers are stuck to zero. In addition, it is also possible to see the content of such registers being correctly quantized before applying the ReLU activation function (values circled in purple).

6.3 Setting up the synthesis tool

Before proceeding with the logic synthesis performed with the software Synopsys Design Compiler®, there were some adjustments to adopt to correctly perform it. First of all, as mentioned in the previous chapters, the technology on which the synthesis has been conducted was the umc-65 nm in worst case condition, that is with 0.9 V supply voltage at 125 °C. Then, in order for clock gating cells to be correctly instantiated, the latch process in the *cluster_clock_gating.sv* file has been substituted with the respective cell provided by the library, named LAGCEPM12R and to which the corresponding signals have been connected.

The next step was to change the adopted *script_syn.tcl* script onto which the commands given to the software tool are gathered. First of all, due to the IPs consisting in several files whose hierarchical dependence is not always straightforward, to the analyze command the autoread attribute with the corresponding path containing the ips has been added. In addition, the recursive attribute has been added when also the files in the sub-directories of the provided path needed to be analyzed. An example of this kind of command is the following:

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

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

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

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

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

Finally, to the compile_ultra command, the following attributes have been added:

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

to specify the software to use the clock gating cells and to not perform auto ungrouping. For further insights on the employed script it is suggested to check the provided *script_syn.tcl* script file.

As a side note, even though it has not been specifically employed for this last synthesis, the area and frequency values reported in the area comparison in chapter 4 have been gathered after writing a simple bash script, named *auto_syn_script*, able to iteratively perform the synthesis for a particular configuration, starting from a very relaxed time constraint, 10 ns, and then proceeding downwards in steps of 0.1 ns until the timing closure is violated. The violation triggers the stop of the script execution and provides the final text files onto which the respective area and timing values are reported. Finally, these two files could later be loaded and fed to a simple MATLAB® script able to plot them as shown in figure 4.4.

6.4 Synthesis Results

Due to the synthesis taking some time to be performed, this has been executed on a separate terminal through the screen command. However, in order not to lose some important information or warning that the software could report, all the messages given by Synopsys Design Compiler® have been redirected to an output log that could later on be consulted.

Hence, initializing the software through the command:

source /software/scripts/init_synopsys

and opening a screen terminal, the synthesis and output log have been respectively performed and generated by issuing the following command:

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

The first thing after performing the synthesis was to check the elaborate report and assure the absence of inferred latches, unless explicitly instantiated like in the register file of the HWPE control modules. Then, some synthesis have been conducted by providing an increasingly relaxed timing constraint, until the slack reported by the timing report resulted to be MET at 3.5 ns ($\approx$ 285 MHz), which has been taken as the new minimum clock period (or maximum working frequency) of the SMAC-Engine as integrated in a system based on the HWPE paradigm. Here, the synthesis results have shown the presence of a combinational path from the register file to the base address generation. This is due to the FSM begin a Mealy FSM as well as due to the presence of eight muxing levels related to the sum operations with the indexes, which may be too complicated for the synthesis tool to handle[2]. Even though the critical path moved from the DP to the address

[2]A better approach could be to use two signals, source_base_addr_x and

generation mechanism and there is definitely room for improvement, the obtained results shows what are the intrinsic limits of integrating the developed architecture in a real working system are.

As for the estimated area, Synopsys Design Compiler® reported 322431.84 μm^2, which is around $\times 1.11$ larger compared to the one estimated by the synthesis of the sole DP of the SMAC-Engine (289626.48 μm^2).

6.5 Place and Route and post-layout simulation

Having completed the logic synthesis by meeting the timing constraint as well has having generated the corresponding netlist, the software Innovus 17.11® has been employed to perform the place and route, targeting around 60% utilization on a square area. Here, after the placement, post clock tree synthesis (CTS) and routing with the corresponding optimizations, the layout for the architecture appeared as shown in figure 6.7. Although the accelerator has been synthesized as a standalone component, without any memory coupled with it, thus not providing an extremely accurate overview of how it would behave on a real platform, it is still a more accurate result than what the post-synthesis would give.

Then, the post-layout netlist has been generated. In order to perform a dynamic power estimation, a value change dump (.vcd) file needed to be extracted exploiting both the derived netlist and QuestaSim 10.6c® simulator. Since there was an interest in extracting just the activity of the signals inside the accelerator, the .sdf file has been ignored and both the technology library netlist and the netlist generated by Innovus 17.11® have been compiled in functional mode by issuing the following commands in the simulator tool:

- compile the technology library netlist:

  vlog +define+FUNCTIONAL -work ./work /software/dk/umc65/Core-lib_LL_Multi-Voltage_Reg.Vt/verilog/uk65lscllmvbbr_sdf21.v

- compile the netlist generated by Innovus 17.11®:

  vlog +define+FUNCTIONAL -work ./work ../innovus/mac_top_wrap.v

source_base_addr_W, whose value is determined separately and stored inside a flip-flop. In this way, the FSM would just select one or the other. Alternatively, the FSM could be turned into a Moore FSM achieving the same results, since in both cases a further bit is required.

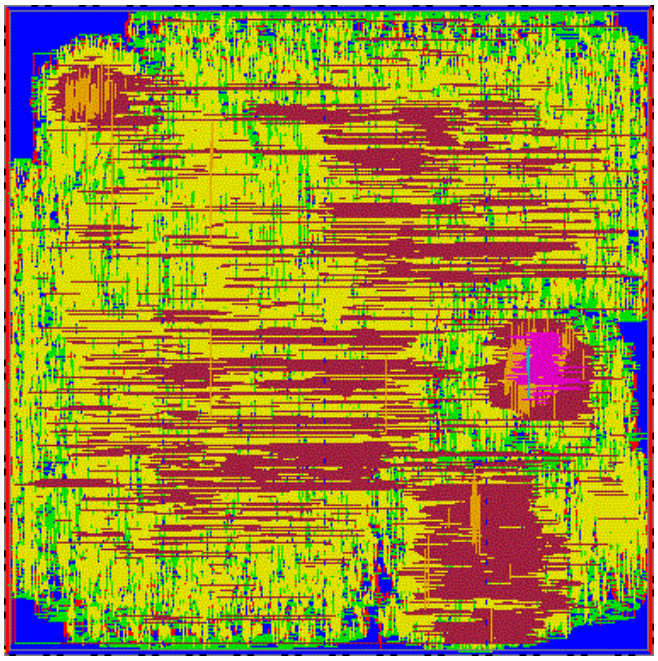


Figure 6.7. Accelerator layout post place and route.

- compile the test bench and packages files:

```
vlog +define+FUNCTIONAL -work ./work
                ../ips/hwpe-stream/rtl/hwpe_stream_package.sv
                ../ips/hwpe-stream/rtl/hwpe_stream_interfaces.sv
                ../ips/hwpe-ctrl/rtl/hwpe_ctrl_package.sv
                ..rtl/mac_package.sv
                ../rtl/tb_dummy_memory.sv
                ../rtl/tb_acc_top.sv
```

In particular, before compiling, the clock period defined in the *tb_acc_common.sv* file has been changed to match the one provided by the logic synthesis tool, hence 3.5 ns. Then, a .vcd file has been created and filled with the activity related information by running the simulation for a time interval going from the start up of the accelerator to the result being written back to the TCDM memory (≈ 18.89 μs), again for the computation of a single $3 \times 3 \times 128$ convolutional volume with 128 filters.

Going back to Innovus 17.11®, the design has been restored and the power analysis has been set up by providing both the generated .vcd file and the time range of the performed simulation. In particular, the dynamic power estimation

has been conducted in typical condition, hence at 25 °C, both at 0.9 V and 1.2 V supply voltages. Then, having retrieved the power results the energy consumption has been derived as follows. For the 1.2 V supply voltage case, the reported dynamic power consumption is 19.62 mW.

Considering the execution time of $ex_{time} = 18882$ ns imposed for the power analysis as well as the clock period being set at $T_{cyc} = 3.5$ ns, the overall number of cycles has been derived as the ratio:

$$\#\text{ cycles} = \frac{ex_{time}}{T_{cyc}} = \frac{18882}{3.5} \approx 5394 \text{ cycles}$$

then, knowing the number of MAC operations that needs to be performed in order to complete a full convolution, the number of MACs per cycle has been estimated as:

$$\frac{\#\text{ MACs}}{\#\text{ cycles}} = \frac{3 \times 3 \times 128 \times 128}{5394} \approx 27.34 \,\frac{\text{MAC}}{\text{cycle}}$$

and the number required by each MAC operation as:

$$\frac{\text{time}}{\text{MAC}} = \frac{\#\text{ cycles}}{\#\text{ MACs}} \cdot T_{cyc} = 0.0366 \,\frac{\text{cycle}}{\text{MAC}} \cdot 3.5 \,\frac{\text{ns}}{\text{cycle}} = 128.03 \,\frac{\text{ps}}{\text{MAC}}$$

Finally, knowing the latter quantity and the estimated power, the energy employed by each MAC operation has been obtained as:

$$\frac{\text{time}}{\text{MAC}} \cdot \text{Dynamic Power} = 128.03 \,\frac{\text{ps}}{\text{MAC}} \cdot 19.62\,\text{mW} = 2.512 \,\frac{\text{pJ}}{\text{MAC}}$$

Similarly, by performing the same operation with the power obtained when the voltage is set to 0.9 V (10.48 mW), the energy per MAC operation diminishes down to:

$$\frac{\text{time}}{\text{MAC}} \cdot \text{Dynamic Power} = 128.03 \,\frac{\text{ps}}{\text{MAC}} \cdot 10.48\,\text{mW} = 1.341 \,\frac{\text{pJ}}{\text{MAC}}$$

Another information provided by Innovus 17.11® is related to the actual gate count, that is the number of employed gates inside the architecture, from which a better area estimation can be obtained. The number of employed gates resulted to be equal to 321919[3], the number of cells 97402 and the overall area equal to 347673.2 μm^2.

As a side note, it should be mentioned that the MAC/cycle value derived above is quite pessimistic, due to the cycles required to program the microcode playing a non-negligible role in the overall estimation. For cases when the control module is programmed and several convolutions are performed, for instance on an entire input volume rather than just a $3 \times 3 \times n_C^{[l-1]}$, the result may be closer to the estimated 32 MAC/cycle.

[3]With the area per gate being 1.08 μm^2.

6.6 Final Results and comparisons

The analysis performed with the employed software tools resulted in an architecture whose overall characteristics are reported in the following summary table 6.1 and compared to other known state of the art convolutional accelerators.

Table 6.1. Summary table.

HW solution	Technology	Frequency	Area	Throughput	Power (@ 25°C)	Energy Efficiency
	[nm]	[MHz]	[mm^2]	[GMAC/s]	[mW]	[pJ/MAC]
SMAC-E standalone	65	416 (wc)	0.29	12.69	-	-
SMAC-E + HWPE @ 0.9 V	65	285 (wc)	0.35	7.79	10.48	1.34
SMAC-E + HWPE @ 1.2 V	65	285 (wc)[4]	0.35	7.79	19.62	2.51
Fulimine [5] @ 0.8 V	65	108 (tc)	0.35	6.35	13	2.05
ShiDianNao [22]	65	1000 (tc)	4.86	64	320	5
Eyeriss [24] @ 1 V	65	200 (tc)	12.25	23	278	12.09
XNE [7] @ 1.2 V	65	400 (tc)	0.092	35	5.92	0.15

The reported values for power refer to an average power consumption where the accelerators are under a full activity and excluding I/Os. Here, the frequency values provided for SMAC-Engine are quite pessimistic when compared to the other solutions, as they have been derived in worst case conditions (wc) rather than typical (tc). For the comparison to be slightly more meaningful, accelerators based on the same technology have been taken into account.

The values provided for the other accelerators have been taken from the respective papers and converted considering 1 MAC = 2 ops. Furthermore, the numbers provided for Fulmine refer to a configuration where the weights bitwidth is equal to 4 whereas the value provided for XNE refer to a configuration where the throughput parameter has been set to $TP = 128$. Looking at the table, it is clear how the performance estimated in chapter four for the standalone DP structure have now been resized to the limitations of integrating such DP on a real working system. Hence, from the too optimistic 12.69 GMAC/s, the actual throughput went down to 7.79 GMAC/s. Although a comparison is not always straightforward, the obtained results are definitely comparable and in line with the others state of the art accelerators. Moreover, the values reported for the SMAC-Engine have been obtained as a first attempt to realize something that could work on an real system such as the PULP platform and there is definitely still room for improvements that could

[4]The values reported for the configuration with 1.2 V supply voltage at 25 °C has been obtained with the same worst case library at 0.9 V due to a lack of time to better learn using Innovus 17.11®. As such, the maximum operating frequency, power and energy efficiency have to be intended as a pessimistic estimation and the real values may actually be more promising than the ones here reported.

further enhance both the energy efficiency and the maximum operating frequency.

Chapter 7

Conclusions and Future Work

This thesis work consisted in the development of the architecture of a SMAC-Engine, a flexible engine for CNN based on serial multiplications and its integration in a HWPE of a PULP platform. When integrated in a HWPE, the obtained accelerator proved to be able to work at 285 MHz when supplied with 0.9 V and to consume as little as 1.34 pJ/MAC, without a too aggressive voltage scaling. Such result is a promising starting point for its future development, as there is definitely room for improvements to make it even more flexible and adaptable to other NN architectures. Moreover, the energy consumption and maximum operating frequency could be further boosted by adopting some proper refinements to the developed RTL.

For instance, a further note concerning the derived DP is that it could eventually be improved to not only support batch normalization but also ResNet architectures. For the first, a threshold fetching would be needed when starting computations for a new layer as well as some additional hardware to perform the actual normalization. To support ResNet architectures instead, a possible solution could be to initialize the AC3 accumulator registers in each SMAC block with the value of the activations from the preceding two layers $a^{[l-2]}$, in order to compute the output activations as in 2.33. To do so, one should not only give access to the above mentioned registers but also consider that the activations are actually quantized on P_a bits and before adding any partial sum to them they should be dequantized to the full internal precision of the accelerator. The dequantization can be performed by changing the structure of each register in AC3 to also support left shifting and again by exploiting the serial structure of each SMAC to dequantize the pre-loaded activation before the first partial sum is computed. Of course, some accuracy loss due to the dequantization process will inevitably be introduced.

The structure derived for the DP is also potentially capable of handling filter

sizes that are larger than 3×3. However, the internal parallelism of the third accumulator varies depending on the size of the considered filter. Hence, to support larger filter sizes but also in order not to downgrade too much the performance of the architecture, a possible solution could be to adapt the internal parallelism of the accelerator to the filter size and disable the unnecessary bits, thus speeding up the computation. As for the control, the low-level control, besides requiring its counters to be properly programmed, would not require any change. In particular, the changes should be introduced at the microprocessor level, where both the microcode and the mnemonics should be adapted for the larger filter sizes.

The low-level control FSM together with the top-level FSM currently supports only the single mode but, with some proper changes in the design, may be improved to support continuous mode, which would both allow to further save cycles as well as make the structure more efficient. Probably, a more accurate design may lead to a solution able to balance out the complexity among these two FSM as well as recover some slack time from the address generation process as suggested in the previous chapters. Finally, the encoding adopted for these FSM may be changed from a binary encoding to a one-hot encoding, to both make the structure resilient to glitches and to reduce the number of commutations when switching among states. All this provided that the number of states is kept rather low.

Another possible improvement for power saving purposes may be to not limit clock gating only in the HWPE structure but also employ it at the boundary of the DP as well as internally to the AC2 and AC3 accumulators. Indeed, a proper use of such technique may allow to reduce the dynamic power consumption by avoiding spurious commutations at the inputs of the unused registers .

Some further enhancements to the structure may either go towards the development on an intelligent scheduler able to handle the activations sparsity so to reduce the overall number of MAC operations to perform by deploying to each SMAC block only the weight bits corresponding to non zero activations or by replicating the SMAC-Engine structure in the horizontal direction thus allowing some weights sharing across the same rows in order to again reduce the number of fetches that needs to be performed from memory both speeding up the execution of the convolution over an entire volume as well as the overall number of fetching from memory that needs to be performed. Of course, both of the latter solutions would require to introduce some further complexity at the control level as well as area occupation, especially the last one, which may eventually impair its adoption in a real working system.

Bibliography

[1] 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.

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Neural Information Processing Systems*, vol. 25, 01 2012.

[3] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1–9, June 2015.

[4] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015.

[5] F. Conti, R. Schilling, P. D. Schiavone, A. Pullini, D. Rossi, F. K. Gürkaynak, 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.

[6] R. Andri, L. Cavigelli, D. Rossi, and L. Benini, "Yodann: An ultra-low power convolutional neural network accelerator based on binary weights," *CoRR*, vol. abs/1606.05487, 2016.

[7] F. Conti, P. D. Schiavone, and L. Benini, "XNOR neural engine: a hardware accelerator IP for 21.6 fj/op binary neural network inference," *CoRR*, vol. abs/1807.03010, 2018.

[8] S. Sharify, A. D. Lascorz, P. Judd, and A. Moshovos, "Loom: Exploiting weight and activation precisions to accelerate convolutional neural networks," *CoRR*, vol. abs/1706.07853, 2017.

[9] A. Ng, "Machine learning on coursera."

[10] A. Ng, "Deep learning on coursera."

[11] Wikipedia, "Neurone."

[12] Wikipedia, "Artificial neural network."

[13] Matlab, "Introduction to deep learning: What are convolutional neural networks?."

[14] S. Barter, "Convolutional neural net in tensorflow."

[15] Y. L. Cun, L. D. Jackel, B. Boser, J. S. Denker, H. P. Graf, I. Guyon, D. Henderson, R. E. Howard, and W. Hubbard, "Handwritten digit recognition: applications of neural network chips and automatic learning," *IEEE Communications Magazine*, vol. 27, pp. 41–46, Nov 1989.

[16] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2014.

[17] M. Mathieu, M. Henaff, and Y. Lecun, "Fast training of convolutional networks through ffts," 12 2013.

[18] A. Lavin, "Fast algorithms for convolutional neural networks," *CoRR*, vol. abs/1509.09308, 2015.

[19] J. Cong and B. Xiao, "Minimizing computation in convolutional neural networks," vol. 8681, pp. 281–290, 09 2014.

[20] A. Aimar, H. Mostafa, E. Calabrese, A. Rios-Navarro, R. Tapiador-Morales, I. Lungu, M. B. Milde, F. Corradi, A. Linares-Barranco, S. Liu, and T. Delbrück, "Nullhop: A flexible convolutional neural network accelerator based on sparse representations of feature maps," *CoRR*, vol. abs/1706.01406, 2017.

[21] F. Conti and L. Benini, "A ultra-low-energy convolution engine for fast brain-inspired vision in multicore clusters," vol. 2015, 03 2015.

[22] 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.

[23] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," vol. 49, pp. 269–284, 02 2014.

[24] 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.

[25] A. Stoutchinin, F. Conti, and L. Benini, "Optimally scheduling CNN convolutions for efficient memory access," *CoRR*, vol. abs/1902.01492, 2019.

[26] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding," *CoRR*, vol. abs/1510.00149, 2015.

[27] J. Choi, Z. Wang, S. Venkataramani, P. I. Chuang, V. Srinivasan, and K. Gopalakrishnan, "PACT: parameterized clipping activation for quantized neural networks," *CoRR*, vol. abs/1805.06085, 2018.

[28] A. Zhou, A. Yao, Y. Guo, L. Xu, and Y. Chen, "Incremental network quantization: Towards lossless cnns with low-precision weights," *CoRR*, vol. abs/1702.03044, 2017.

[29] B. Lorica, "Compressed representations in the age of big data."

[30] G. Aashish Barnwal, "Huffman coding."

[31] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally,

"EIE: efficient inference engine on compressed deep neural network," *CoRR*, vol. abs/1602.01528, 2016.

[32] Kiarasht, "Compressed-sparse-row."

[33] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *CoRR*, vol. abs/1704.04861, 2017.

[34] Z. ETH, "Pulp platform."

[35] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "Dadiannao: A machine-learning supercomputer," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 609–622, Dec 2014.

[36] C.-F. Wang, "A basic introduction to separable convolutions."