Politecnico di Torino

POLITECNICO
DI TORINO

Corso di Laurea Magistrale in Ingegneria Informatica

Tesi di Laurea Magistrale

Data-flow analysis and optimization in Convolutional Neural Networks on-chip

Relatori:
Prof. Andrea Calimera
Dott. Valerio Tenace

Candidato:
Luca Mocerino
matricola: 229006

Ottobre 2017

Abstract

In recent years machine learning has become an increasingly important topic in the computer science field, computer vision in particular. Their domain of application is classification, object detection/recognition and identification of handwritten digits. These technologies can be used in many fields such as automotive (autonomous guide), aerospace, but also domotic applications.
In the last few years, a new class of machine learning algorithms, the so called *"deep learning"*, has substantially improved in terms of accuracy and efficiency. However, these algorithms require high-performance platforms with huge computational power and large memory resources. Cutting edge research is looking for viable solutions to implement such deep-learning algorithms into portable/wearable devices. Unfortunately such lightweight embedded platforms are resource constrained. Here the challenge: map these algorithms on low-power/low-energy architectures while preserving a reasonable level of accuracy.

Within this context, this work introduces a dataflow analysis and optimization technique for a specific class of deep learning algorithm, i.e., Convolutional Neural Network (CNN). A CNN consists of a set of different layers: *pooling layer* for reducing feature maps dimensions; *dropout* for avoiding overfitting; *softmax* for the final classification and *fully-connected* layer for forwarding and backwarding propagation. For the intrinsic nature of the computation, the most energy consuming is the CONV layer that performs matrix convolution. The number of CONV layers in a CNN determines the overall energy consumption of the network. It is worth noticing that CONV layers are not just computational bounded, but also memory bounded; indeed, the ceaseless memory access for reading/writing operands represents a substantial contribution to the total power consumption. Even if the convolution operation is computationally intensive, also in parallel compute paradigms the data movements can be more energy consuming than the computation [34].
A heterogeneous architecture is the target for our simulation, since, among the possible implementation strategies, tightly coupled processing arrays with embedded memory hierarchy represent the most suited solution. As main advantage, those architectures improve (if compared to conventional CPUs and GPUs). More precisely, this heterogeneous architecture is made of a CPU, an off-chip memory, an on-chip memory hierarchy and a processing element (PE) plan for computation. Using this topology as reference template, this thesis introduces a simulation engine able to simulate the dataflows and gives as feedback the performance evaluation in terms of energy and latency. Moreover, it uses textual representation for the memories and PE plan configuration. This tool provides the right support to design and properly size the architecture. The simulation engine is integrated into a popular framework for deep learning, called Caffe, in order to extract the network topology and give a fast energy estimation of CNN. The proposed solution is faster than logic simulators (i.e. ModelSim) and gives information about performance compared to other software frameworks (Caffe,Tensoflow, etc ...). Using the described architecture as a reference, we implement and simulate convolutional layers in order to extract the performances (energy/latency). Using the proposed tool, this work investigates two

different dataflows: a modified *"No Local Reuse dataflow"* and the *"Window Stationary Dataflow"*.
The former, as the name suggests, does not use the PE registers to store any kind of data. It only uses a big buffer (> 2Mbyte) in order to store the complete input feature maps and kernels. The intermediate results are accumulated in an inter-PE buffer. This dataflow is used as a baseline for the performance.
The latter belongs to the "weights stationary dataflow" family. The weights stay stationary in the PEs' register file in order to maximize the weights and reuse during convolution. The partial sums are accumulated into an inter-PE buffer and the PE plan is configured to implement a tree of adder for speeding up partial sums accumulation. The data reuse mechanism implemented through the local buffer guarantees a substantial reduction of the energy consumption and latency: 35% and 49% (on average), respectively.
The same simulation engine is used for the assessment of a new pooling strategy aimed at improving energy efficiency. It consists of the classical average pooling, but preserving the feature maps dimensions. This technique, used with a mechanism that can detect equal value in the feature map in order to skip MAC operations, allows to reduce the energy consumption (~40%) working on two factors: number of multiply and add operations and size reduction on the input feature map, with a little loss in terms of network accuracy (~5-6%).

Acknowledgements

I would like to express my special appreciation and thanks to Professor Andrea Calimera, who has been a true mentor to me; without his guidance and constructive advice this thesis would not have been possible. Also, I am particularly grateful for the assistance given by Valerio Tenace. Last but not least, I would like to thank all my affections, whose support over these years was invaluable.

Luca Mocerino, Torino, October 2017

Contents

List of Figures

List of Tables

Acronyms

AI	Artificial Intelligence
ASIC	Application specific integrated circuit
ALU	arithmetic logic unit
CGRA	Coarse Grain Reconfigurable Architecture
CNN	Convolutional Neural Networks
CPU	Control Processing Unit
DSP	Digital signal processor
EDA	Electronic Design Automation
FPGA	Field-Programmable Gate Array
FPU	Floating-point unit
IC	Integrated circuit
LUT	Look-up table
MAC	Multiplier–accumulator
PE	Processing element
SoC	System-on-a-Chip
RTL	Register transfer level
VLSI	Very-large-scale integration

1
Introduction

In order to clarify its purpose and consistency, it is of prominent interest to contextualize this work and illustrate the goals.

1.1 Context and Motivation

The first idea of a neural network was theorized in the forties [2]. Since then, modelling and the approaches to Artificial Intelligence (AI) have drastically changed. In recent years, deep convolutional neural networks have been widely used since they achieved record-breaking accuracy, especially in Computer Vision tasks. The main drawback of CNN is the computational complexity. The popularity of CNN in the last few years [3] and the market interest for the possible CNN applications have pushed a very dynamic field of research. The computational complexity makes CNN incompatible with an embedded environment. This was true for years, when machine learning algorithms were only used in power demanding systems, datacenters, etc...
The increasing efficiency and accuracy of CNNs allow the utilization of these networks in a wide range of applications. The automatic inspection could be used in manufacturing applications, object recognition for visual surveillance; navigation may be used by an autonomous vehicle or mobile robot. The majority of these environments requires an embedded system, a portable system. The main trade-off, in this case, is related to energy efficiency and throughput, or area overhead and performances. In order to sustain a certain level of performances while maintaining power consumption low, new design methodologies and new platforms have to be developed.

1.2 Goals and Challenges

This work is contextualized in the situation described in the previous section. In that context, first we have to identify a platform that can satisfy the availability of resources that can be used to parallelize the computation and a memory hierarchy with resources *near* the computation. Then, we have to spot an energy efficient dataflow, for scheduling and mapping the highest energy-demanding CNN layer, convolution. Finally, we have to find a technique to exploit the peculiar format of the data involved in the computation in order to reduce energy consumption and at the same time sustain a certain level of performances (throughput and network

accuracy). In this work we address these problems by developing a new dataflow and a technique for exploiting data statistics.

1.3 Thesis Organization

The report is organized as follows:

Chapter 2 Background and Concepts: in chapter two we introduce the reader to deep learning algorithms, with a special focus on CNN; also, we described the state-of-the-art for ConvNet on embedded world.

Chapter 3 Energy-Efficient Dataflows: modeling and simulation: in chapter three we describe the developed dataflows (No Local Reuse and Window Stationary dataflow) and the developed optimization.

Chapter 4 Evaluation and Results: in chapter four we comment the performance results of the simulated dataflows and the optimization applied on them.

Chapter 5 Conclusion: in chapter seven we summarize the achievements of our work and we give recommendations for future works.

2

Background and Concepts

In this chapter the reader can find a description of the "deep learning" field with a special focus on the Convolutional Neural Networks. An overview on the existing and potential hardware platforms for CNN follows.

2.1 Convolutional Neural Networks

This section gives a brief overview of neural networks describing the main concepts. Then, the focus is moved to ConvNet in detail.

2.1.1 Introduction to Neural Networks

Artificial neural networks (ANNs) are computing systems that can learn (a progressively performance improvement) how to do tasks by considering examples, generally without task-specific programming [1]. The first mathematical model was provided by W. McCulloch and W. Pitts [2] in the late 1943. Their work gave a new perspective on the human brain logic and it started a new field of research which was unexplored at that time. In the 1954, Rosenblatt introduced the concept of *perceptron*, a solution for pattern recognition problem [4]. Over those years, the research continued (e.g. in 60', the first multilayer network was theorized [5]), but achievements in developing new theoretical models did not follow progresses in computing capability. This is the reason why the research slowed down for years. In the last few decades, since computing platforms addressed the computational request, the research has increasingly speeded up.

Neural networks are originally inspired by the biological neural network of the nervous system. Many analogies link artificial neural networks to the biological ones. The biological networks are composed of the neurons and the synapses. The synapses are not only the means of transmission of the information, but they can also control the information working on the amplitude of the signal or blocking the transfer.

In the artificial neural network the basic unit is the artificial neuron. As formally described in the equation(2.1), neuron has an input signal x_i that is multiplied by one or more weights (w_i) and added to a constant parameter (bias b_j) in order to give the output results.

$$y_i = b_j + \sum_{i=0}^{N}(x_i * w_i) \tag{2.1}$$

The weights are the mathematical way to model the synapses behavior. For example, if the weight $w_i > 1$, the input signal is amplified; otherwise with $w_i < 1$, the input is reduced [6]. The following picture shows the neurons structures:

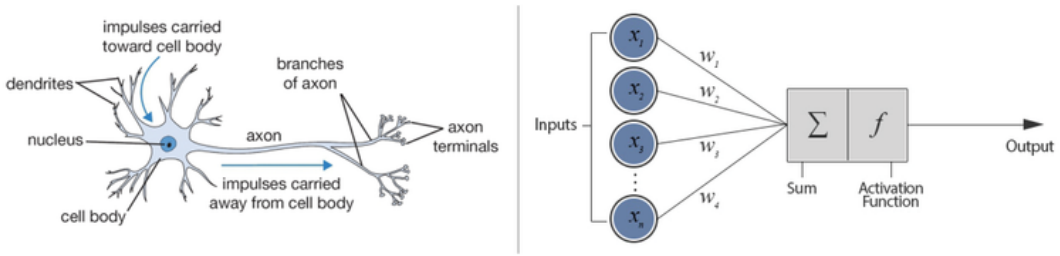


Figure 2.1: Biological and Artificial Neuron

An artificial neural network is composed of many different neurons layers. Only the neurons belonging to the same layer can be connected together. The following picture shows a trivial neural network:

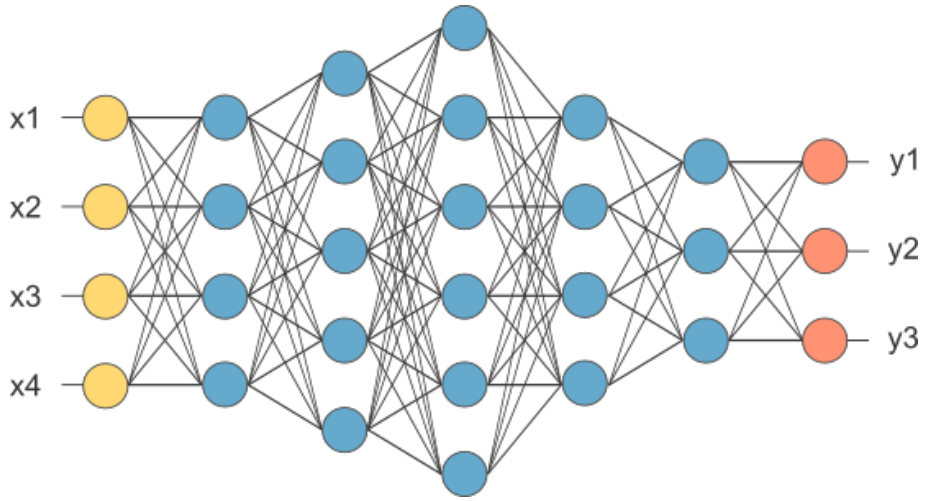


Figure 2.2: Artificial Neural Network structure [7]

In this case, the network has four neurons in the first layer and three at the output. A neural network has to learn in order to recognize and make predictions, just like a child. The neural network development is made up of two steps: training (learning) and validation.

Training

The training phase is crucial for developing a good neural network. Just like a human being learns from experiences and from samples, a neural network *learns* from samples. In this case samples are labeled (*supervised learning*), not labeled (*unsupervised learning*) or a mix of labeled and not labeled samples (*semi-supervised learning*). In neural networks the weights are the objects of the learning process. In the initial training phase, the weights are initialized with random parameters. The samples are sequentially presented to the network input layer (*forward*). The outputs are compared to a *ground truth labels* using a *loss function*. That is a metric for quantifying the distance between output and expected results. The main purpose of the training phase is to repeat the forwarding on all the training set (*epoch*) minimizing the loss function. Then a gradient is used to understand how weights influence the error. The gradient is computed backpropagating the output error through the network (backward). According to the gradient the weights are updated.
The training phase is an optimization loop composed of these three phases: forwarding, backwarding and updating.

Validation
After a while (epochs), the algorithm used for training converges. In order to extract the accuracy of the network, a validation phase is needed. This phase uses a set of inputs different from the ones used for training (*training set*): the validation set. The idea is to test the network on new samples that represent the real application samples for the considered network. Each input sample is submitted in the network for the forwarding phase. The outputs are compared to the real ones in order to understand the accuracy of the network.

2.1.2 Convolutional Neural Network architectures

Covolutional Neural Networks (CNN or ConvNet) represent a special class of deep learning algorithms mostly used for two-dimensional inputs (images, video) for image classification, object recognition tasks. A CNN consists of a set of layers, inputs, outputs and different hidden layers.
In a classic neural network, the neurons of each layer are fully connected. In image-related application, the inputs are pixels. In a small image (i.e 200x200) the number of processed inputs are 200x200x3. As a consequence, in a fully-connected layer, billions of weights are required. Not in this case, because for images we can exploit the locality of the information. For example, if we want to recognize an airplane in a picture, in the majority of the cases the airplane is in the center of the picture. The border pixel or the background pixels are not significant for that task. The core of the information, for image, stands in the neighbor relationship among near pixels. For example, the edges give us information about the shape of the object to detect [6]. This is the main reason we do not need all fully connected layers but we can replace most of them with convolutional ones.

Parameters The parameters involved in ConvNet are the input feature maps with $WxHxC$ size, where W stays for the width, H for the height and C represents the number of input channels. The filters have $KxKxC$ size, where Ks are the dimensions of the kernel and C the number of the channels. The number of filters represents the output channels size. Another important parameter is the bias B, its dimension is equal to the number of filters.

The layers in CNN are the following:

- **Convolutional:** this layer applies multi-dimensional convolution on a set of input feature maps with a set of filters (multi-dimensional kernels). The output of convolution is provided after computing the dot product between the weights and a local region on the input feature map.[6]

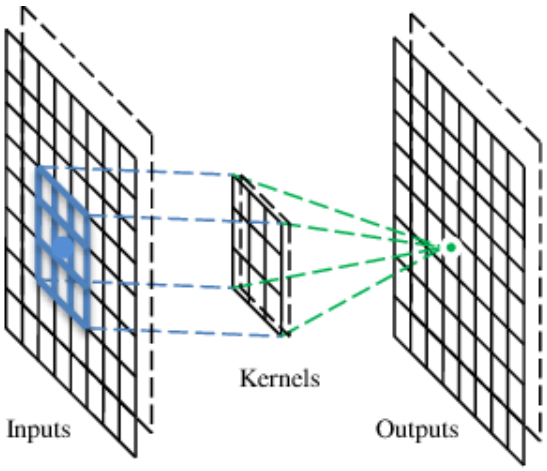


Figure 2.3: Convolution [8]

Figure 2.3 shows the convolution operation between a slice of the input feature map and a kernel. Sometimes the input feature map could be padded with zeros (or ones).

- **Pooling:** it is common to find a pooling layer among the convolutional layers. The pooling layer reduces the spatial dimension of the convolutional output and consequently it reduces the number of input parameters for the following convolutional layer. The most used pooling is the max pooling, which uses the maximum value from each cluster of neurons [9]. It works on the width and height dimensions leaving depth unchanged. Another form of common pooling is the *L2-norm pooling*. The following picture shows a max pooling example:

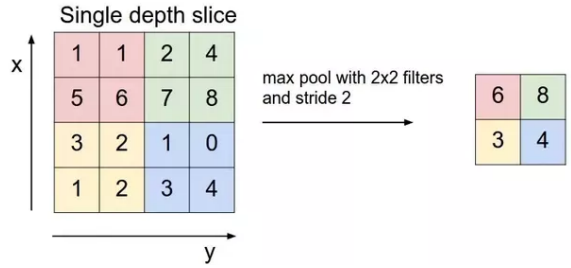


Figure 2.4: Max pooling example [6]

In Figure 2.4 a pooling kernel of size 2x2 and stride 2 is used to reduce input size.

- **Fully-connected:** as suggested by its name, a fully-connected layer connects every neuron in one layer to every neuron in another layer. It is the basic concept of a regular neural network. The computation is a matrix multiplication and bias addition [6].

- **Nonlinearity:** it is used to apply a non-linear activation function. Several activation functions are used in CNN; here the most common ones:
  - **Rectified Linear Unit (ReLu) [10]**: $f(x) = max(0, x)$ where x is the input pixel;
  - **Sigmoidal**: $f(x) = \frac{1}{1+e^x}$;
  - **Parameteric rectified linear unit (PReLU) [11]**: $f(\alpha, x) = max(\alpha * x, x)$ where α is another parameter to "learn".

- **Local Response Normalization (LRN):** this layer was introduced the first time in AlexNet [42]. The concept is related to the biological concept of "lateral inhibition". It is the capacity of a stimulated neuron to affect the neural activity of neighbors neurons. This mechanism creates a sort of competition among the neighbor neurons. In artificial neural networks the same effect can be reproduced by normalizing a neuron response compared to the neurons of the adjacent output channels.

- **Dropout:** the most popular technique used in order to avoid the *overfitting* [1]. It works by cutting ("dropping") some random connections in the training phase.

- **Softmax:** the classical *classifier*. It computes the final score for each class and it is usually used as the last layer in the most common CNN architectures.

Popular Neural Networks frameworks In the last few years, the software frameworks for designing, training and validating neural networks have drastically grown. Among others, the most used are Caffe [34], TensorFlow [14], Neural Net-

[1]It occurs when a model has poor predictive performance since the generated parameters fits "too much" the training set.

work Toolbox from Matlab [12], Theano [13]. The majority of these frameworks are both CPU and GPU compatible. For this work Caffe is used in order to extract the interesting parameters and to validate the network after applying optimization.

Popular network for Image Classification The popularity of CNNs, the diffusion of image classification contests and competitions contributed to the growth of the developed CNN. Here we describe the most popular ones:

- **AlexNet**[42]: the network used in this work. It is composed of five convolutional layers, three fully connected layers and about 60 million of parameters.

- **GoogLeNet** [15]: by Christian Szegedy et al. from Google, it is the state-of-the-art of CNN in image classification. It is a record-breaking network since it achieved 6.67% of error rate in ILSVRC with 1.2 a million of parameters. It is composed of 22 layers, only three of which are the convolutional layers. The strength of this network stands in the so called *Inception modules* that can reduce the numbers of channels.

- **VGG** [16]: stands for Visual Geometry Group from University of Oxford. This network has 19 convolutional layers with more than 150 million of parameters.

- **SqueezeNet** [17]: the design purpose of this network is different from the others, not aiming at the accuracy, but at parameters reduction. It obtains 50X less parameters than AlexNet, preserving the same accuracy.

2.2 Embedded Convolutional Neural Networks

In the previous chapter we described the state-of-the-art for ConvNet. In this chapter we describe the embedded solution adopted for CNN algorithms in terms of commercial and research platforms.

2.2.1 Potential Hardware Platforms

Embedded systems usually have more strict constrains, compared to the general purpose ones, in terms of area and energy. Moreover, in some cases those systems have process tasks with a constrained deadline (real-time application). These are the main reasons why designing algorithms for an embedded platform could be a non trivial task. The task becomes harder when the algorithms are CNNs that require a high-computational capability while preserving the strict constrains that derive from the embedded world. Nonetheless, there are still several options for developing CNNs on embedded platforms. The most popular ones are described below:

- **CPUs**: processors which are commonly used in general purpose systems. The high throughput required in CNN algorithms seems not to fit adequately this solution. High-parallel computing paradigms (SIMD) multi-cores architectures effectively address the computation requirement to achieve high throughput, energy consumption still remains high as data movement can be more expensive than computation [25].

- **GPUs**: originally designed for graphical applications, these platforms have now adopted a general purpose paradigm (GPGPUs). GPUs are multi-core platforms with a high-level of computing parallelism supporting OpenCL or CUDA software frameworks. A top gamma GPU like NVidia GeForce GTX Titan X [18] is composed of more than 3000 floating-point processing cores running at 1 GHz. While computing 6600 GFLOP/s, power consumption is up to 250W. The majority of GPUs are well suited for CNN applications, but the area overhead and the **high energy consumption** make GPUs not suited for an embedded environment.

- **FPGAs**: special ICs composed of an array of programmable logic blocks and a hierarchy of reconfigurable interconnections. This allows to modify the design after the manufacturing process. This characteristic makes FPGAs appealing in supporting CNN application. In [20], the authors evaluate emerging DNN algorithms on two generations of Intel FPGAs (Intel Arria10 and Intel Stratix 10) against the latest highest performance NVIDIA Titan X Pascal GPU. The results show that Intel Stratix 10 FPGA is 5.4x better in performance (TOPS/sec) than Titan X Pascal GPU on GEMMs for sparse, Int6, and binarized DNNs, respectively. On Ternary-ResNet, the Stratix 10 FPGA can deliver 60% better performance over Titan X Pascal GPU, while being 2.3x better in performance/watt [21]. So, power consumption is lower than GPUs but higher for an embedded environment.

- **ASICs**: apparently an adequate compromise between performance and energy consumption. Several solutions have been designed, which are suitable for CNN. The main problem with ASICs is that they require lots of effort in the design; plus, in the majority of cases they are not easily scalable. However, recent researches have achieved appreciable results. Some of them are described below.

2.2.2 Existing CNN Implementations on Embedded Platforms

In this section we introduce the most relevant works on CNN implementations with a special focus on ASIC solutions, since an ASIC is the architecture used as reference for our simulations.

The FPGA approach

A milestone for FPGA implementations is the approach of Zhang et. al. in 2015 [23]. After observing several FPGA implementation of CNNs, he realized the majority of those implementations did not maximize the exploitation of the available FPGA resources. That research group developed a polyhedral-based optimization framework for analyzing loops tiling in convolution. All the possible schedules were analyzed using the *"roofline model"*, which relates to computation and communication for the design exploration.

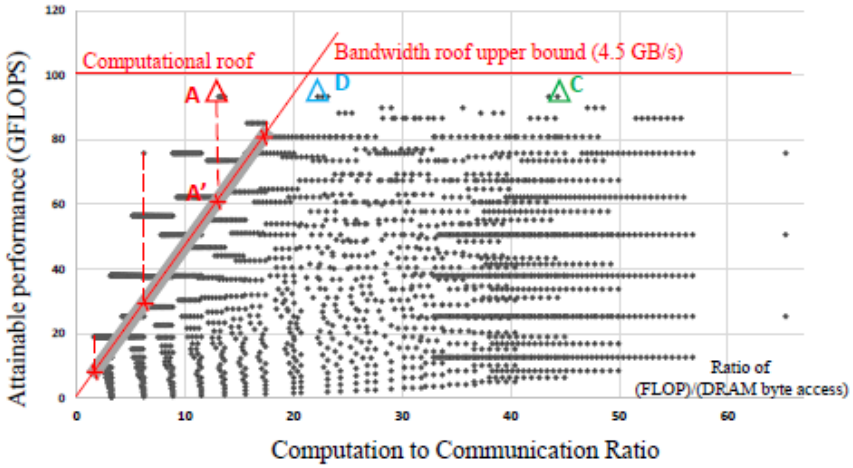


Figure 2.5: Design exploration in [23]

In Figure 2.5 we find the FLOPS vs Computation to Communication ratio curve for conv5 layer of AlexNet using Xilinx Virtex7 485T FPGA as platform. The possible implementations (varying the tiling factor) are represented in 2.5 as dots and the red lines indicate the platform constraints. In that design space, the best solutions can be chosen, C in this case.

ASIC approaches

In recent literature, the custom accelerators for CNNs are uncountable. We try to present the state-of-the-art of both commercials and research in chronological order. The first work to mention is *DaDianNao* accelerator(2014) [22], used both for training and inference. It is a multi-chip system composed of 1 to 64 nodes and a large, shared off-chip central memory (DRAM). The synthesized results are up to 126.66x for energy saving and 300.04x faster than the GPU baseline.
Another relevant architecture was proposed in [25] in 2016. It is a heterogeneous platflorm for CNN.

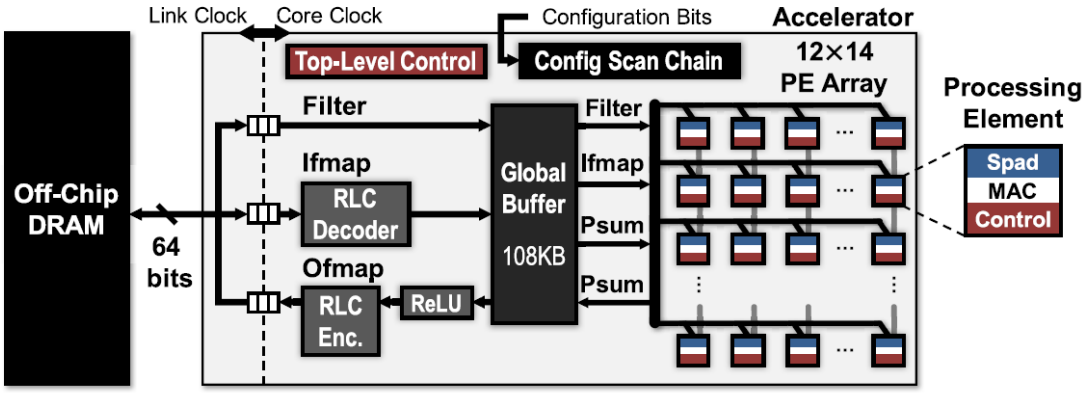


Figure 2.6: Eyeriss block diagram

In Figure 2.6, we can notice 168 processing elements composed of three parts: a computation, memory and control. A four memory hierarchy includes off-chip DRAM, on-chip Global Buffer, inter-Pe communications and register files. The energy efficient features in [25] are the RLC encoding for reducing DRAM accesses (left side in 2.6), zero detection unit[2] and the *Row-Stationary dataflow*. The basic idea of that dataflow is to break the high-dimensional convolution into a set of 1D primitives. Each primitive operates on one row of filter weights and one row of input feature map pixels, and generates one row of partial results [25]. Data are fetched from the different elements of the memory hierarchy.

The state-of-the-art for CNNs on ASIC is the Tensor Processing Unit used by Google in its datacenters [24].

[2]discussed in the following sections

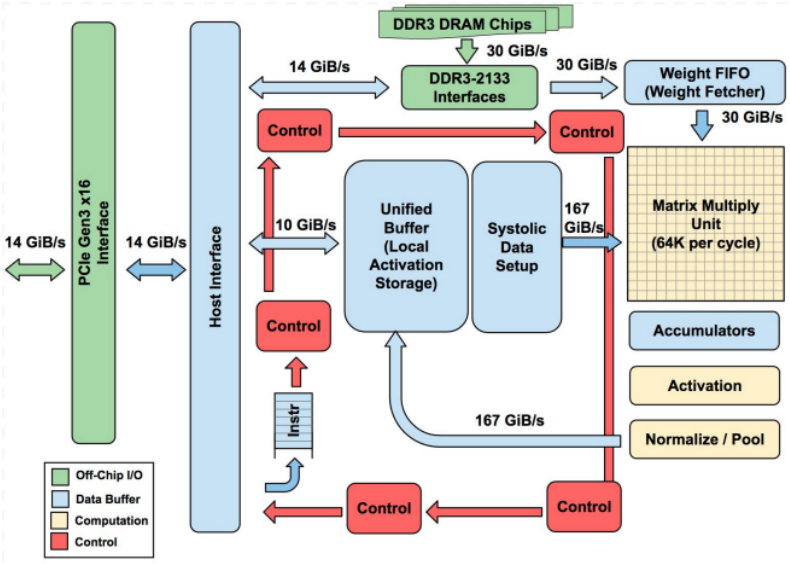


Figure 2.7: Google TPU block diagram

Figure 2.7 shows the block diagram of TPU. The core of TPU is the Matrix Multiply Unit (MMU), which is composed of 256x256 MACs that can perform 8-bit multiply-and-adds on signed or unsigned integers designed for dense matrices. The *Accumulators* are used to accumulate the results of MMU, one 256-element partial sum per clock cycle. The memory hierarchy is composed of a dedicate memory for loading/storing weights, *Weight Memory* (8Gb DRAM), a 24 Mb on-chip *Unified Buffer* for partial results[24]. The several benchmarks are used for performance evaluation of CNN, Multi-level-percetron etc.

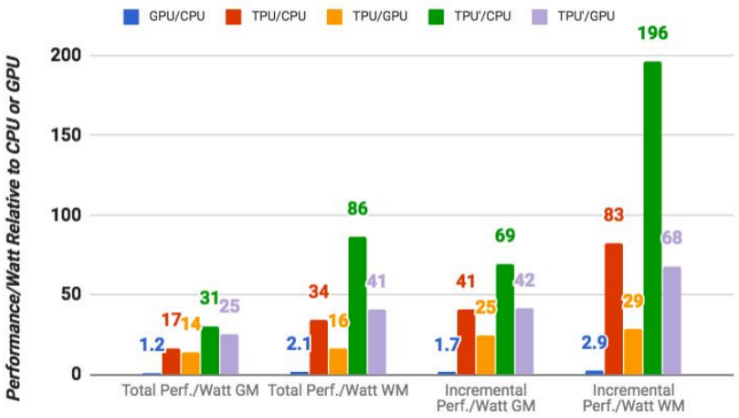


Figure 2.8: Google TPU block diagram

In 2.8, we find the relative performance/Watt ratio of TPU compared to CPU and GPU. TPU dominates the competitors. In 2.8, TPU' is an improved version with larger memory and higher operating frequency.

2.3 Dataflows Taxonomy

In the recent literature all dataflows can be clustered in the following classes [25]:

- Output Stationary (OS) Dataflow;
- Input Stationary (IS) Dataflow;
- No Local Reuse (NLR) Dataflow;
- Weight Stationary (WS) Dataflow.

2.3.1 Output Stationary Dataflow

The main feature of this family of dataflows is the accumulation of the partial sums in the same PE registers. At the same time each pixel should stay stationary in the PE to be accumulated with the purpose of minimizing the accumulation cost. Each dataflow is different depending on the choice of the accumulation plane:

- Accumulate the partial sums over the channels (multiple or single);
- Accumulate the partial sums considering one sub pixel plane (multiple or single);
- Mixed solution.

These sub-classes share the concept of accumulating data in the register file of the PEs and using an intra-PE data reuse, with the need of additional register memory. Some examples of the above described dataflows can be found here [27, 28, 29]

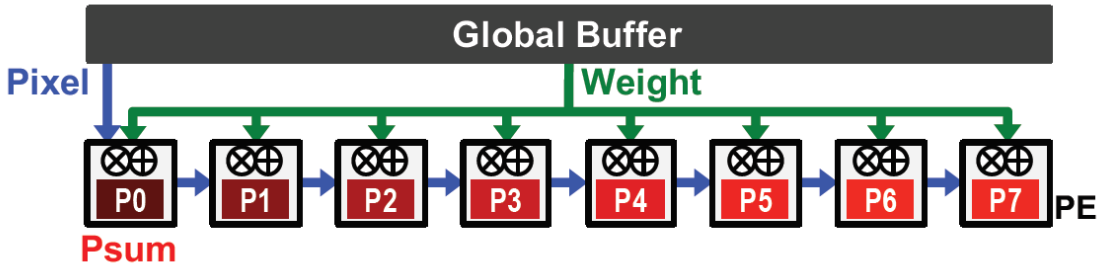


Figure 2.9: Output Stationary dataflow schema [25]

2.3.2 No Local Reuse Dataflow

In this kind of dataflow there is no data reuse at PE level, so the PEs are used only for computation and the global buffer could be larger than the previous cases. There is only an inter-PE reuse, array level. The datapath of each PE reads as input an input feature map element and a weight element and performs the MAC operation. The partial sums are accumulated in the global buffer as the input/output feature map in order to reduce the read/write operations from the main memory.
Some examples of the above described dataflows can be found here [30, 31]

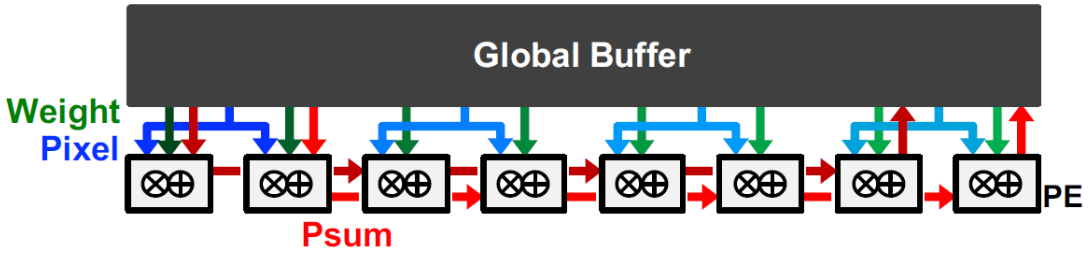


Figure 2.10: No Local reuse dataflow schema [25]

2.3.3 Weight Stationary Dataflow

For this class of dataflows the weights are stationary in registers of PEs in order to maximize the weights and convolutional reuse. Weights belonging to same filters are fetched from the DRAM and remain stationary in the register file. KxK weights are in registers and the input feature map is fetched from the buffer or directly from DRAM and shared among the PEs. Moreover, partial sums are not accumulated in the register file, so additional buffer traffic is required for computing them. The buffer size is critical for the performance: a small buffer could require additional on-chip transactions for the partial sums and off-chip transactions for swapping data from buffer to external memory.
Some examples of the above described dataflows can be found here [32, 33]
This work will present a variant of that class of dataflows.

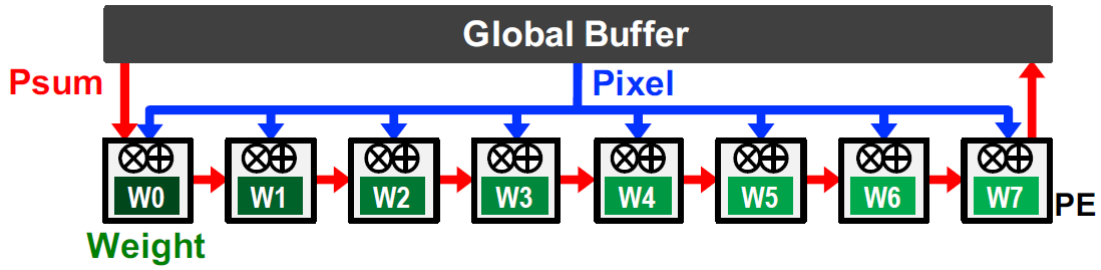


Figure 2.11: Weight Stationary dataflow schema [25]

2.4 Exploiting data statistics

In the previous chapters it was stated that the convolution operation is the most expensive in terms of energy. Data involved in this operation derive from two classes of input feature maps and filters. The most used methods are classified using two generals criteria:

- Reduce the number of the operations;
- Reduce the size of the operand;

Moreover, another classification can be done, based on the object of the optimization: input feature maps or filters. In the following chapter we summarize the most used optimization techniques and then we present the optimization developed in this work.

2.4.1 Operations reduction

In order to reduce the number of operations for convolution, different approaches have been developed at different points of the design:

1. **Network level:** pruning a set of CNN layers;
2. **Intra Layer compaction:** exploiting the input data (filters or input feature maps) statists to compress data.

Network level

The first approach involves the CNN designers. It consists of minimizing the number of convolutional layers in order to reduce the total number of operations. The main concept is to break the high-dimensionality of convolutional layers and decompose them in smaller layers. In this operation the trade-off is between network accuracy and number of operations. An example of decomposing filters is shown in the following picture:

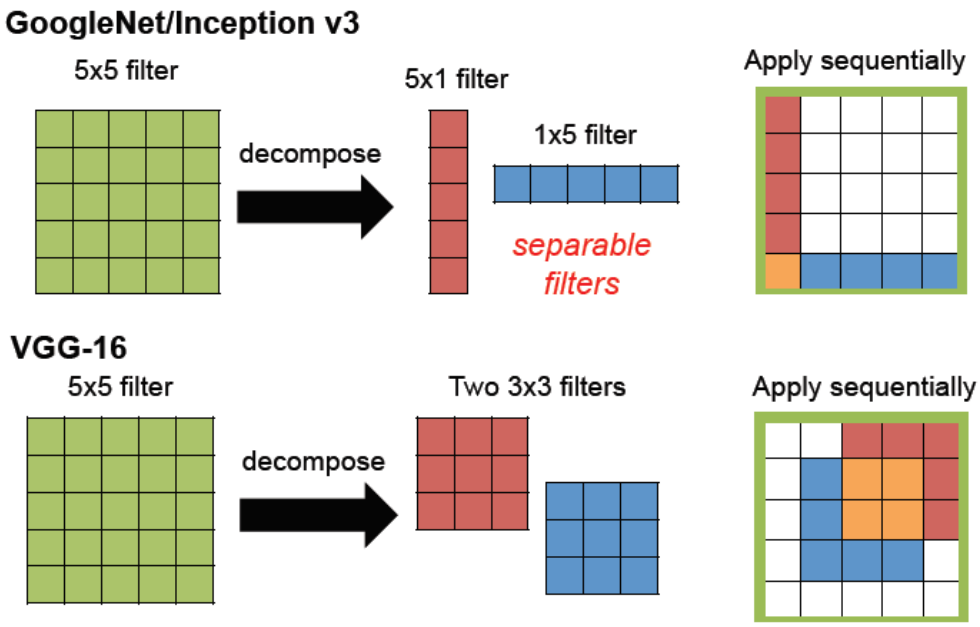


Figure 2.12: Filter design in GoogleNet and VGG-16 [37]

In Figure 2.12, we can notice that GoogleNet [15] uses separable filters, which are special filters that can be expressed as the outer product of two vectors. So, the original kernel can be written as a matrix product of a column and a row vector. Moreover, the convolution operation is associative so:

$$f * (v * h) = (f * v) * h \tag{2.2}$$

where v and h are the two vectors that compose the original kernel and f the other signals involved in convolution. In that way, considering HxW input feature map size and KxK the kernel sizes, the original complexity is $HWKK$; instead, with separable filters the computational complexity of convolution will be $HW(K+K)$. Moreover, the filters obtained after training are not usually separable.

Intra Layer compaction

Many examples of this kind of approaches are presented in recent literature, the most relevant ones are described in this work.

Zero handling

The data involved in convolution, filters and input feature maps are full with zero values. Moreover, after the ReLU layer, other zero values are added in the output feature maps. There are two ways of exploiting this fact for reducing the multiply and add operation. The first one is to use a zero detector for skipping the MAC operations; the second one is to exploit this data characteristic to use a sparse representation for matrices or to use data encoding such as RLC (Run-Length Compression) or smarter ones, to compress data [25]. The former can be easily applied using a "zero detector". An example can be found in the following picture:

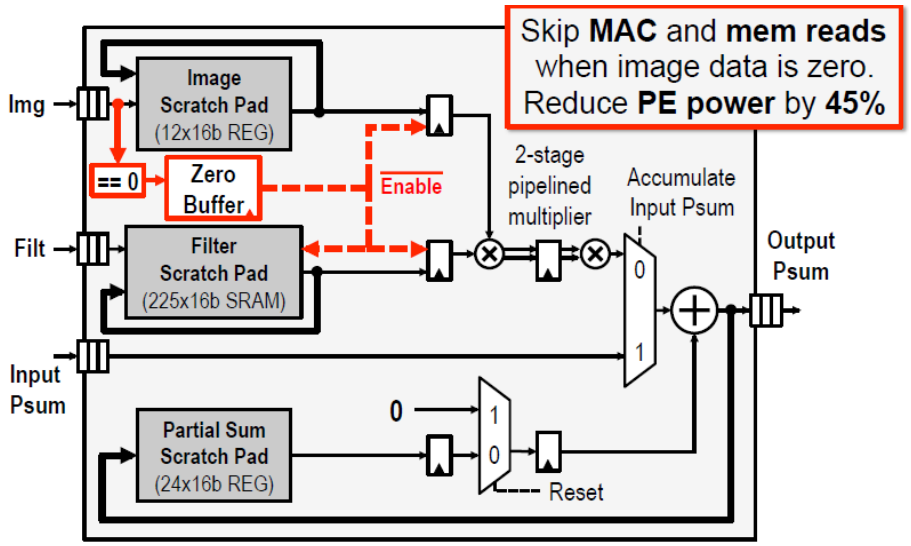


Figure 2.13: Zero detection in [37]

In Figure 2.13, the architecture presents a "zero detection" mechanism that allows to recognize zero elements in input feature maps while reading them from the dedicated scratchpad memory. This mechanism guarantees an estimated power saving of 45%.

Another way to prune the weight is described in [36]. In that work, weights are pruned during the training phase. This pruning mechanism is based on **weights magnitude**. This technique in 2.14 is composed of three steps:

- A standard training for learning the connectivity;
- All connections with weights below a certain threshold are pruned;
- Retraining for learning the new weights.

The following picture shows the dataflow adopted in [36] :

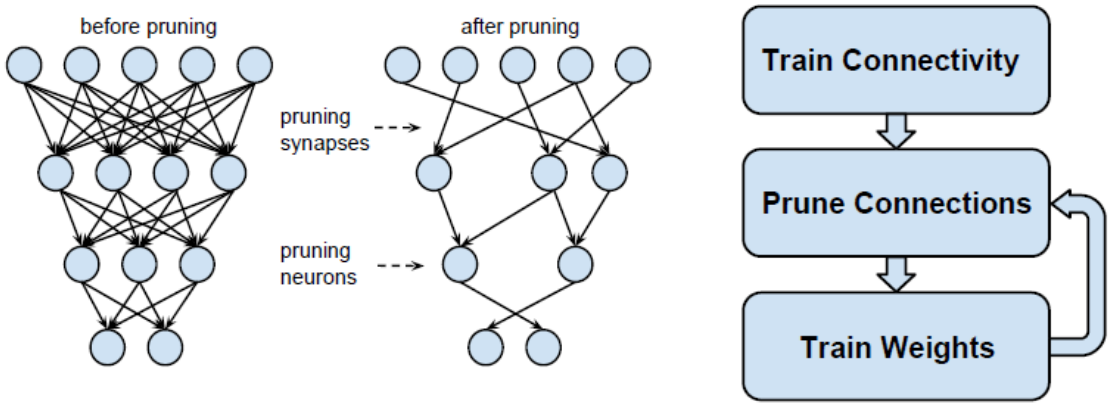


Figure 2.14: Weights pruning in [36]

In Figure 2.14, on the left there is a schematic representation of the original network before and after the pruning. This technique is also used on fully connected layers with an average speed up of 3.2x on GPU, 3x on CPU [36]. Moreover, in this technique we do not find an energy aware pruning. Some examples of energy-aware methods can be found here [38].

2.4.2 Operand size reduction

The operand size reduction is one of the most exploiting optimization area. In this work only few relevant approaches are reported. Two are the main possible optimizations:

- Bit-width reduction and data representation;
- Data quantization;

Bit-width reduction and data representation

The values involved in convolution are usually floting-point values. The first approach can be to change representation from floating point to fixed-point representation. In [39], we can find an example of dynamic and static fixed-point. The dynamic fixed-point representation allows to customize the data representation of each layer of the network in order to reduce the data size and meanwhile minimize the accuracy loss. The following technique is composed of the steps shown in the following picture:

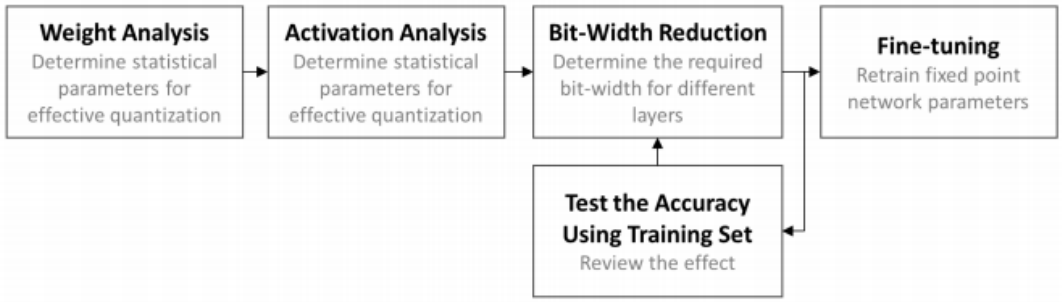


Figure 2.15: Approximation flow in [39]

In Figure 2.15, the work-flow is shown. A first analysis on the weights allows to properly choose the quantization parameters. Then, a similar analysis is done on the connections. After these two steps, the required bit-width for each different layer is found. The network was tested on the test set in order to evaluate the accuracy. If the accuracy is the desired one, the optimization algorithm stops. Otherwise, the bit-width configurations previously chosen for the layers are changed and re-tested. Once a good trade-off between small number representation and classification accuracy is found, the resulting network can be fine-tuned. In that phase, the network was retrained with the new weights and the gradient function

in back propagation computes the error between the floating point weights and the quantized ones.
The results in terms of network accuracy are shown in the following picture:

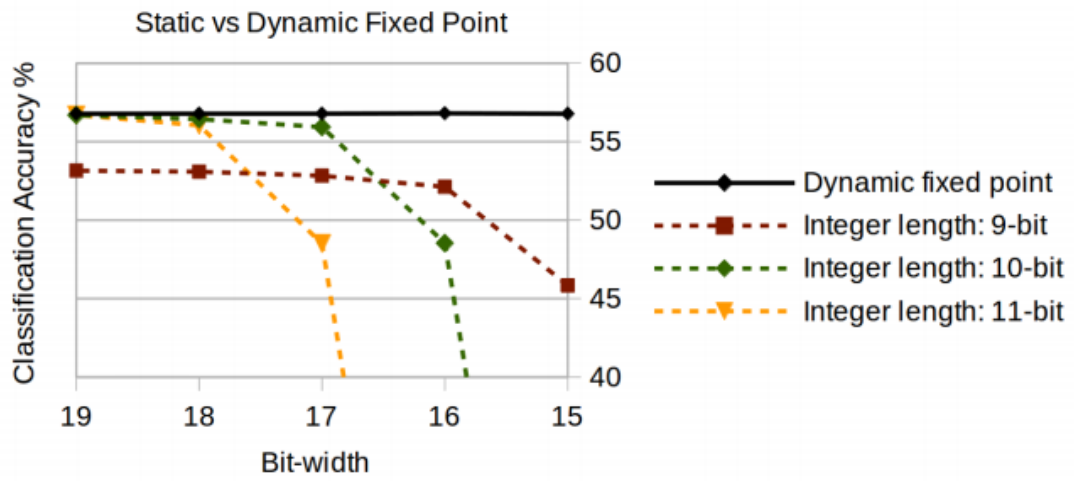


Figure 2.16: Dynamic vs static fixed-point: Top-1 accuracy for CaffeNet on ILSVRC 2014 validation dataset

In Figure 2.16, we can find the accuracy comparison between dynamic and static fixed-point of different bit-widths. What emerges is the better performance of dynamic fixed point.

Data quantization

Quantization means remapping data from a given set to a different one reducing the number of levels (bits) of representation. In this case the main purpose is improving performances (enegy/speed) while preserving a certain accuracy. An example of quantization applied to weights can be found in [40]. In this case the new domain is logarithmic.

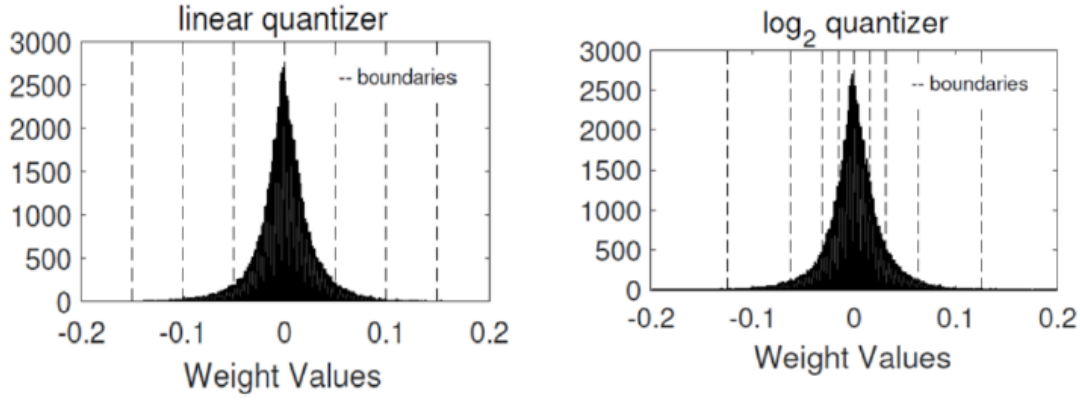


Figure 2.17: Weights distribution in the original and logarithmic domain

In Figure 2.17, there are the representations of weights distribution in linear and logarithmic domain. The main advantage to use the log domain is that the product in this domain is just a shift. So, the most used operation in convolution (MAC) is replaced by a shift and add operation, faster and less energy consuming. Two approaches are used: log domain for activations only or for both weights and activations. The results, using weights: 5-bits for CONV, 4-bit for FC; activations: 4-bits, imply just a 3.2% of accuracy loss for AlexNet [40].

3

Energy-Efficient Dataflows: modelling and simulation

This chapter is the core of the work. It provides a detailed description of the designed dataflows and the adopted simulation framework. Then, an optimization technique is presented.

3.1 Dataflows description

This section illustrates the core of the work. Firstly, it presents the reference architecture on which we simulate the implemented dataflows; then, the simulation framework is described. Finally, the two implemented dataflows are detailed.

3.1.1 Reference Architecture

In the previous chapters there is a description of the several proposed architectures, ASIC or on FPGA. The adopted architecture is a heterogeneous one and it is composed of :

- Main memory (DRAM);
- Global buffer;
- PE plan: composed of NxN processing elements with register file;
- Inter-PE buffer shared among PEs.

The following picture shows the schematic of the architecture :

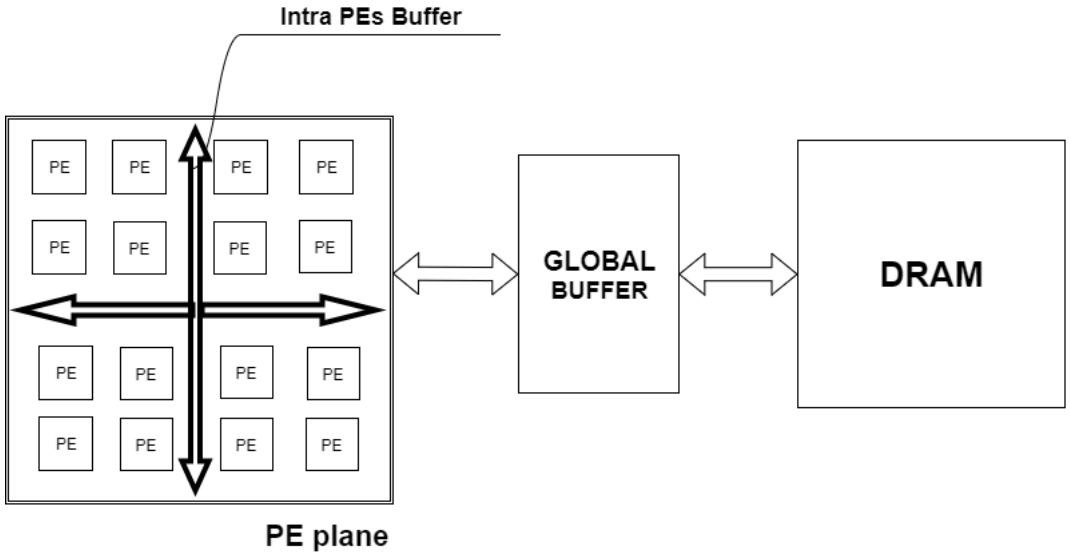


Figure 3.1: Reference architecture schema

The architecture in Figure 3.1 has four levels of storage hierarchy: DRAM, global buffer, array (inter-PE communication) and RF, sorted by their energy cost for data accesses from high to low [25]. The main memory is the off-chip where data are stored: input feature map and filters. The main buffer is an on-chip memory. The buffer usage depends on the implemented dataflow. The PE plan is the chip where there are NxN processing elements. All PEs share a memory buffer used in both the implemented dataflows, for storing partial sums.
The following picture shows the schematic of the internal structure of a Processing Element:

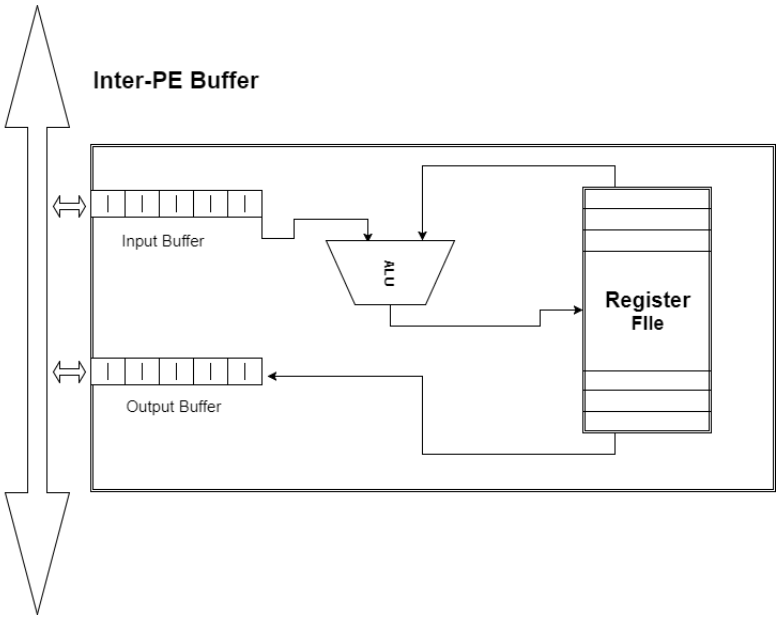


Figure 3.2: Processing element schema

The processing element structure in Figure 3.2 conceptually has an ALU that includes a Multiplier and Accumulate (MAC) ,two FIFOs queue for data exchanging to/from inter-PEs buffer and a register file. As for the implemented dataflow, the size of this register file (i.e the number of registers) and the number of PEs change.

The main purpose of this architecture is that of supporting an energy efficient dataflow. Hence, a memories hierarchy is used for reducing the memory accesses to an off-chip memory. Moreover, the PE plan guarantees an high parallel computation, required for the convolution operation.

3.1.2 Simulation Framework

The simulation framework is the main core of the work since it allows to evaluate the energy and latency of the developed dataflows. It was totally written in Python and it uses the most common libraries used for image processing and data science as *Numpy, Scipy etc* ...
The following picture shows the overall simulation framework:

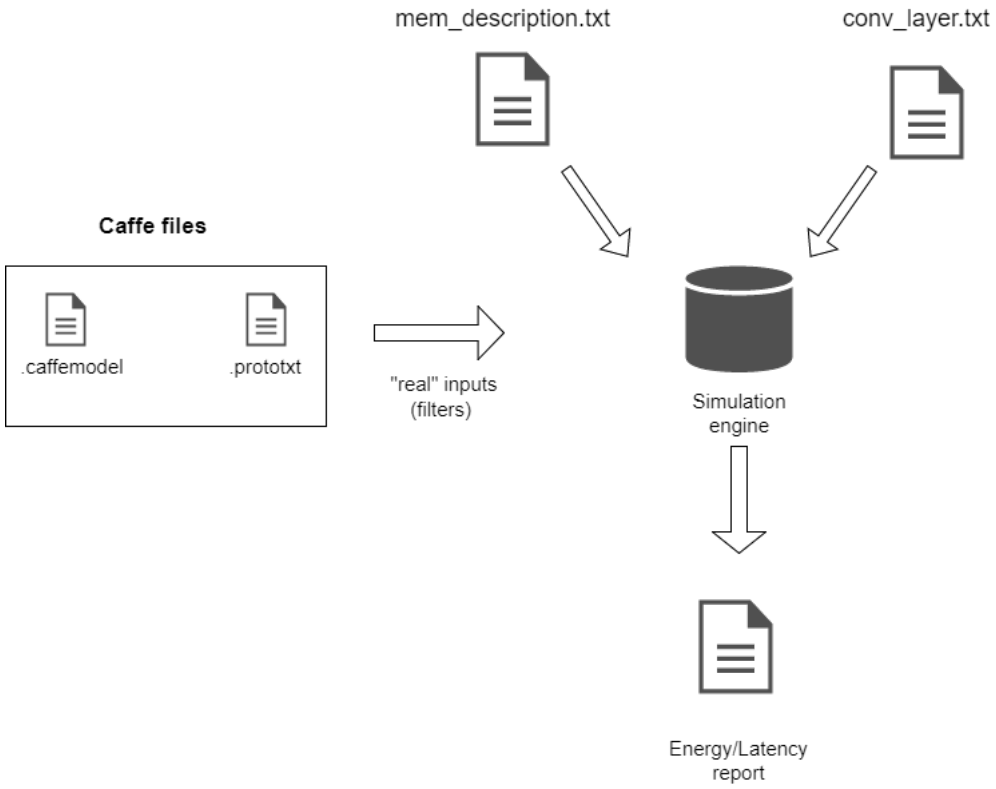


Figure 3.3: Simulation framework schema

In Figure 3.3, it is described the entire software architecture. On the left side of the image, there are Caffe files used in our framework. "Caffe is a deep learning framework made with expression, speed, and modularity in mind" [34]. Caffe uses file with extension `.prototxt` [1] to describe the CNN networks with their layers. Moreover, a `.caffemodel` [2] is used to store the model obtained after training phase. Both protocols are Google Protocol Buffer.
From `.caffemodel` and `.prototxt` file, we extract the real parameters (filters and biases) used in convolutional layers of the selected network. The additional input files describe the memory hierarchy and convolutional layer parameters shapes in the adopted custom protocol. The main parameters for memory specifications are: number of words, energy access cost, just to cite the most relevant. On the other hand, the parameters for the architecture are the number of PEs, registers per PE,

[1] Plaintext protocol buffer.
[2] Binary protocol buffer.

inter-PE buffer size, type, ALU cost etc...

mem.txt ×

```
DRAM {nwords=10485760,energy_access=200,rw_lat=1}
BUFFER {nwords=3072000,energy_access=6,rw_lat=1}
cgra {X=11,Y=11,nregs=32,energy_access=1,bnwords=1024,bpower_access=2,brw_lat=1}
```

Figure 3.4: Memory description file example

In Figure 3.4, an example of configuration is showed. In 3.4, by crga we mean PE plan and the inter-PE buffer. In this case we use 11x11 PEs, each of them having 32 registers with an unitary energy access cost and an inter-PE buffer with 1024 words with an access cost of 2 and a read/write with unitary latency.
The outputs of the simulations are the characteristics of the convolutional layer (number of MACs operation, input/output feature map size), **energy** and **latency**. The software framework is composed of Python classes for memories and PE description, utility classes for extracting operation statistics, given the input parameter shapes.

3.1.3 Window Stationary Dataflow

The implemented dataflow belongs to a weights stationary family since it shares all the concepts described at the beginning of this chapter, but it takes some changes in order to achieve better performance in terms of energy saving. First of all, we try to resume the main concepts related to the weights stationary family:

- Pros:
  - Weights are stationary in Register File;
  - Maximization of weights and convolutional reuse;
- Cons:
  - Partial sums are NOT (always) accumulated in the PEs;
  - Additional bus traffic due to continuous memory-to-memory transfer;

This work aims at taking the main advantages of this solution and operating on the drawbacks.
The *Window Stationary Dataflow* takes its name from the fact that we store in the PEs register file at least one complete filter: a $K\ x\ K\ x\ C$ where K is the width/height of a square kernel and C the kernel channels. The intermediate buffer must contain a complete input feature map ($W\ x\ H\ x\ C$: width, height and channels) and at least one kernel ($K\ x\ K\ x\ C$) depending on the considered configuration. In that way we have no need of fetching data directly from the main memory (off-chip transaction). The PE plan is able to compute at least one window each C clock cycle, where C are the considered kernel/image channels. In that way the MAC results are **accumulated in the PE register file**. For the partial sums we use an intra-PE buffer in order to compute the additions using PE in a sort of tree of adders trying to exploit efficiently all the computation capability available.
The main dataflow could be splitted in two parts: configuration and storage of the input parameters, convolution.

Part one: configuration and storage of the input parameters

In the following picture we can find the **part one** description: configuration and storage handling.

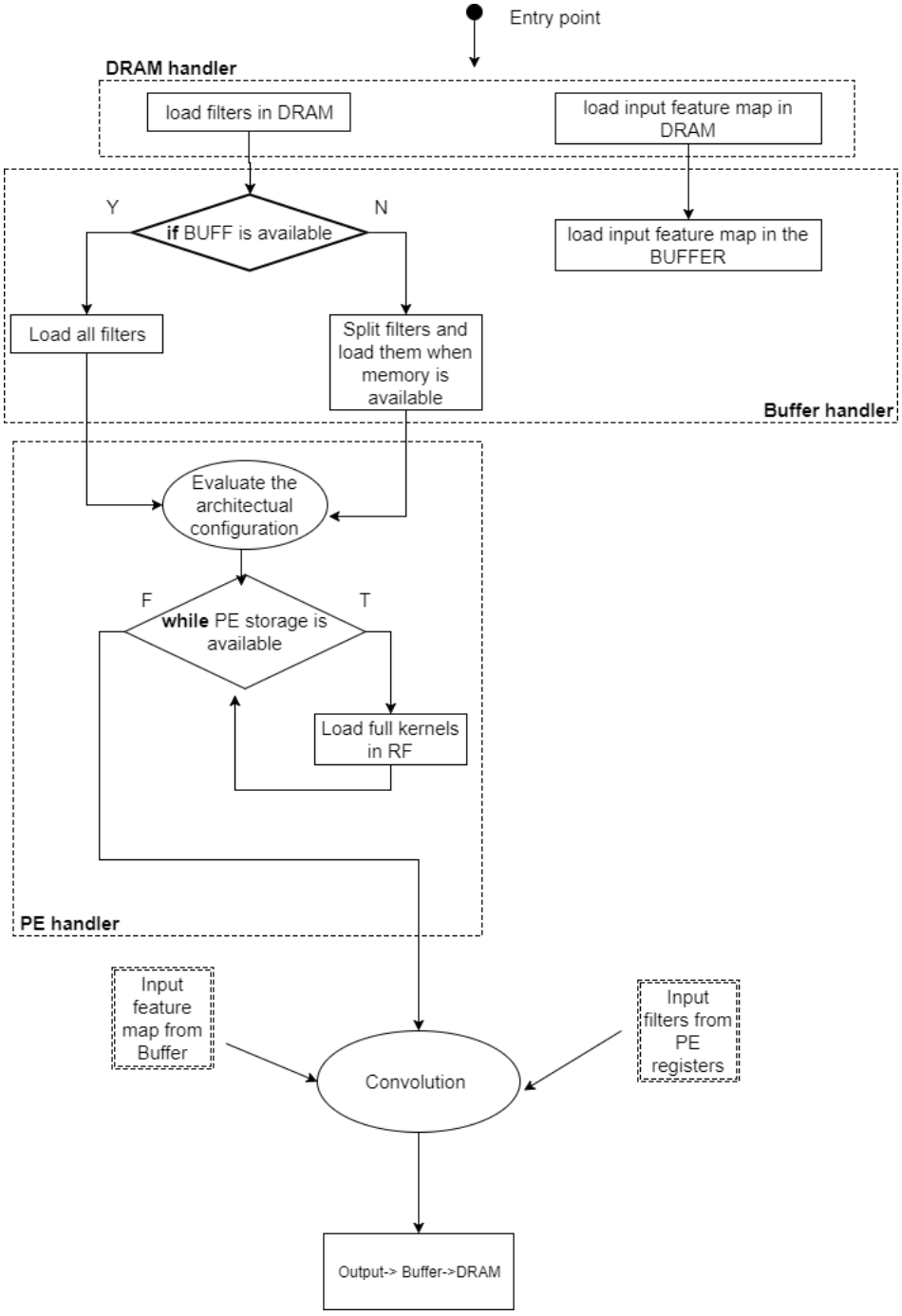


Figure 3.5: Configuration and storage handling

This dataflow in Figure 3.5 performs the following steps:

1. Input feature maps and kernels are loaded in the main memory.

2. The buffer must contain all input feature maps and at least one complete filter. So input feature maps are completely loaded in the buffer. Then, if there is any available space in the buffer, all space available is filled with a certain number of filters. The filters that are not in the buffer yet are loaded in a second moment, when that part of filters, which are in the PE register file, is no longer useful for the computation.

3. According to the PE plan configuration (i.e. number of PE and register file size of each PE), a subset of filters is stored in the RF. The PE plan must contain at least one complete kernel at time.

4. At that point, the convolution is performed with input filters from register file and input feature map from buffer. In this way we use the **nearest memory** for the **most frequent memory accesses** (reads of kernel elements).

Part two: convolution

The convolutional layer is the most significative in terms of computation. Now we analyze the convolution operation and how we perform it in our dataflow.

"Multidimensional discrete convolution refers to the mathematical operation between two functions f and g on an n-dimensional lattice that produces a third function, also of n-dimensions"[35].

In order to illustrate the adopted operation, in the following table we define the parameters shape.

Parameter	Description
M	Number of multidimensional filters and output feature map channels
C	Number of input feature map and filter channels
H/W	Input feature map height/width
K	Filters height/width
OH/OW	Output feature map height/width
S	Stride

Table 3.1: Parameters shape

Using the parameters in Table 3.1, we can describe the used convolution with the equation (3.1), where $0 \leq u \leq M$, $0 \leq y \leq OH$ (with $OH = (H - K + S)/S$) and $0 \leq x \leq OW$ (with $OW = (W - K + S)/S$).

$$\mathbf{Outfmap}[u][x][y] = \sum_{z=0}^{C-1} \sum_{i=0}^{K-1} \sum_{j=0}^{K-1} \mathbf{I}[z][Sx+i][Sy+j] x \mathbf{W}[u][z][i][j] \tag{3.1}$$

For sake of clarity, we attach also a C-like pseudo-code for a naïve implementation of convolution with nested for loops:

```
for (u=0;u< M; x++){
   for (y=0;y< OH; y++){
      for (x=0;x< OW; x++){
               for ( i=0;i< K; i++){
                  for ( j=0;j< K; j++){
                           for (z=0;z< C; z++){
            output[u][x][y]+=I[z][Sx+i][Sy+j]*W[u][z][i][j]
                           }
                        }
                  }
               }
    }
}
```

The pseudo code representation allows us to understand the computational complexity of this operation in terms of number of MACs operations. Moreover, we can gather the information that the loops are independent from each other so the loop order can be chosen properly.
In order to better describe our dataflow behavior, in the next line we propose a simple example of how the computation happens in on-chip part of our architecture.
Example:

In our example we analyze the convolution of a $3x3x3$ kernel with an input feature map describing the computational behavior and the main benefits of this approach.

Starting point : All required data are in the main buffer waiting to be distributed in the PE register file.
The following picture shows the filter used in this example and the corresponding three channels C0, C1, C2:

$$\begin{bmatrix} 0 & -1 & -1 \\ 1 & -1 & 0 \\ 0 & 1 & -1 \end{bmatrix} \begin{bmatrix} 1 & -1 & -1 \\ 0 & 0 & 1 \\ 0 & -1 & -1 \end{bmatrix} \begin{bmatrix} 0 & 1 & 1 \\ 0 & -1 & 0 \\ 0 & 1 & 1 \end{bmatrix} \tag{3.2}$$

Figure 3.6: An example of $3x3x3$ filter where C0, C1, C2 are respectively the filter channels.

In the following table we summarize the shape of the parameters described in Table 3.1:

Parameter	Value
M	1
C	3
H/W	Not relevant
K	3
OH/OW	Not relevant
S	1
PE plan	3x3 with 4 regs

Table 3.2: Parameters example

For this example we use one filter (M in Table 3.2) and a PE plan composed of 9 PEs each of them with 4 registers. After setting the parameters, the following picture shows in a visual way all the parameters involved in convolution:

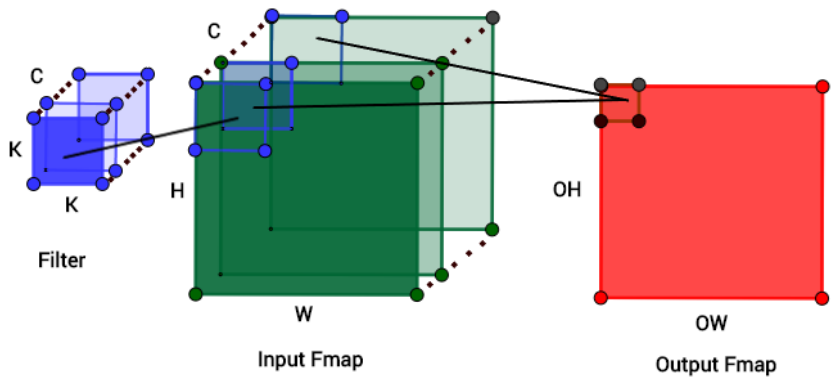


Figure 3.7: Convolution computation

In Figure 3.7, what is shown is the convolution between an input filter with all its channel C, on a slice of the input feature map of size WxH giving as results an output feature map element.

All the configurations are done at that moment. Each PE has in its register file one (in this case, or more, depending on the configuration) weight for each filter channel. In this way each PE can perform the MACs operations along the channels accumulating the results in the additional register available. All the PEs can elaborate the whole window in parallel, in a $C\ x\ MAC\ latency$ clock cycle; after that, the results are written in the inter-PE register file. This feature **improves the computational throughput** by exploiting all the computational power available. Moreover, the partial sums are accumulated in the inter-PE register file that allows to exploit the nearest (free) memory storage **improving the energy efficiency** of

the computation.
The following picture is a snapshot of the PE plan and the respective register file:

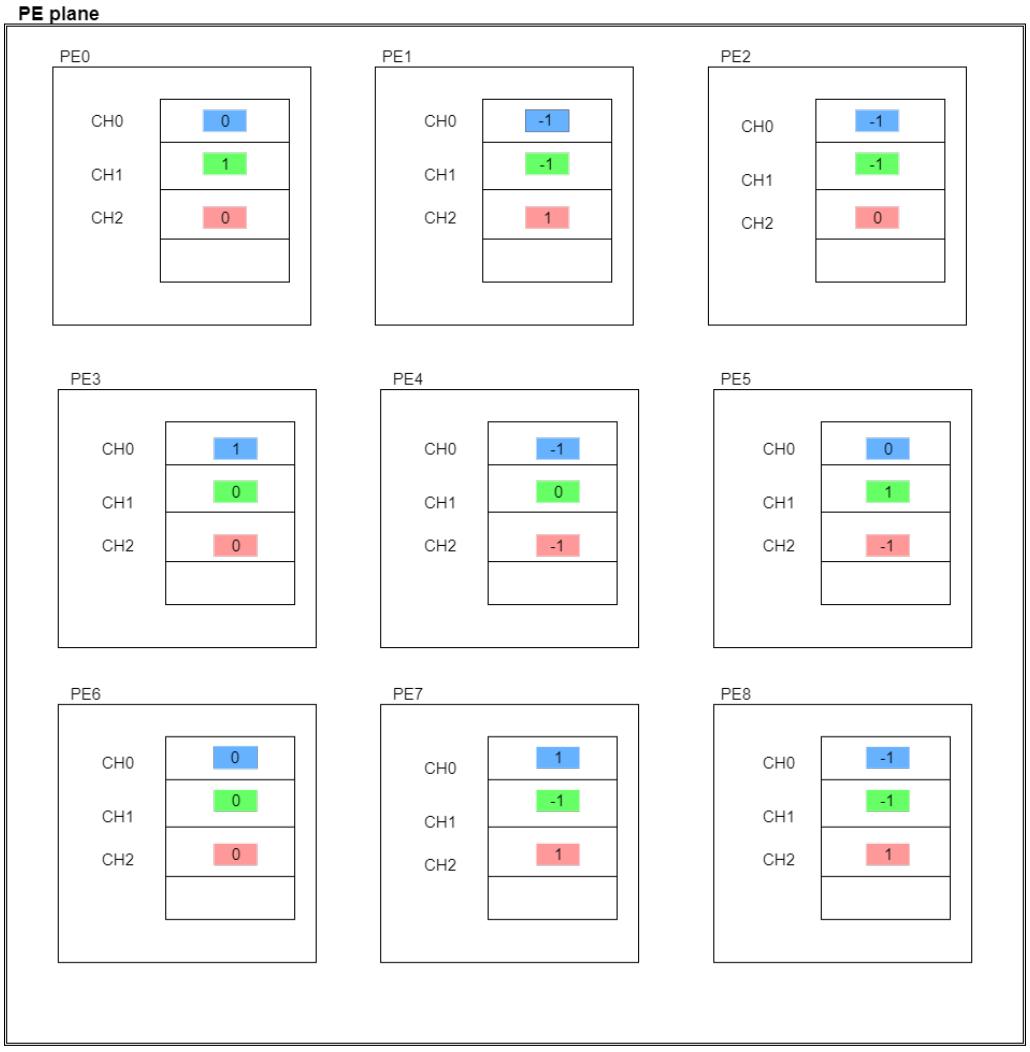


Figure 3.8: PE plan after initial weights assignment

In Figure 3.8, we can see the distribution of weights in the PEs register files. The blue values belong to the channel C0 of the filter in 3.6, the green ones to C1 and the pink ones to C2. The filter mapping in Figure 3.8 allows to have the values from different channels that belong to the same filter in the register file of a PE. The additional register is used for accumulating temporary MAC results. In these examples, after the MACs operations, we have nine values to be summed together. This phase is shown in the following picture:

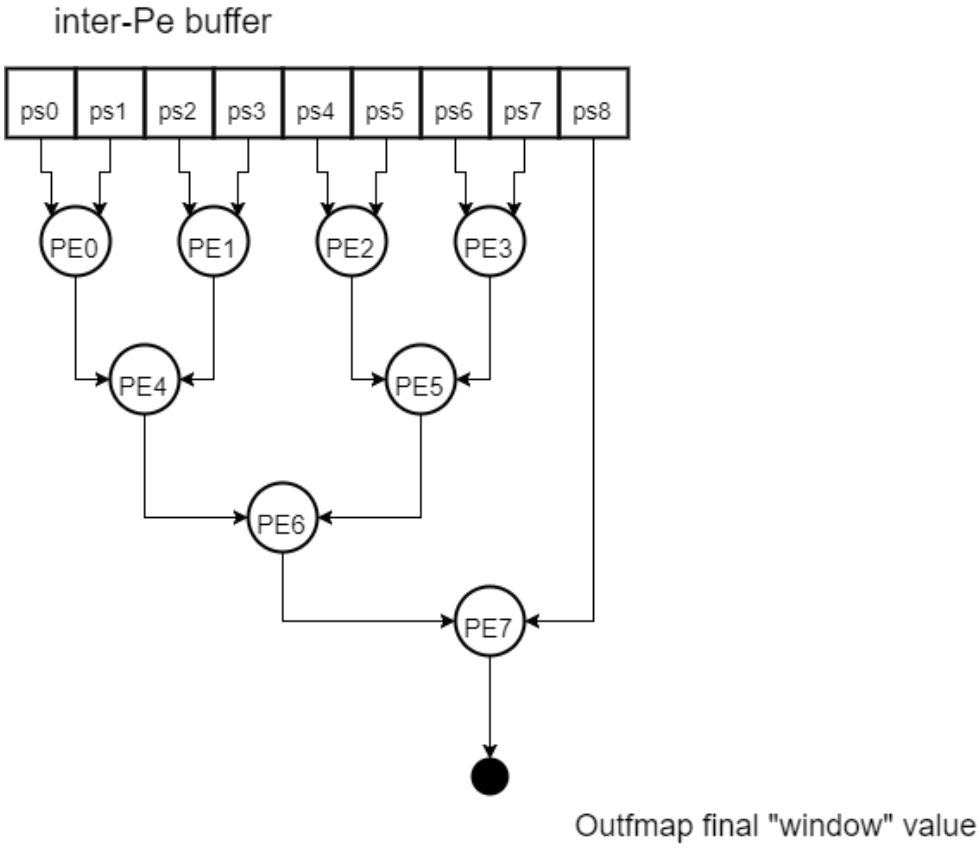


Figure 3.9: Tree of addition configuration

We can see the nine values resulting from the previous phase to be added on the top of the Figure 3.9, in the inter-PE buffer. At this point the PEs are configured to form a tree of adder. The adders in PEs are used to compute the final sum loading/storing partial sums to/from inter-PE buffer...
This mechanism improves the throughput and the energy efficiency when reading values in the nearest memory location (lower access cost).
The main benefits of using this dataflow are the following:

- Maximization of the weight and convolutional reuse;
- Exploitation of the nearest memory available to load/store data in order to minimize the energy consumption due to the data movements;
- Reuse of the computational capability of each PE also to maximize the throughput of partial sums.

3.1.4 No Local Reuse

This dataflow is used as baseline for the performance comparison. This dataflow belongs to the homonym dataflow family. The main features are :

- No local RFs are used to accumulate data;
- The global buffer is usually big (>2Mbyte) in order to store the whole dataset (input feature maps and filters);
- The accumulation of partial sums is in the inter-PE buffer;

The main dataflow is described in the following picture:

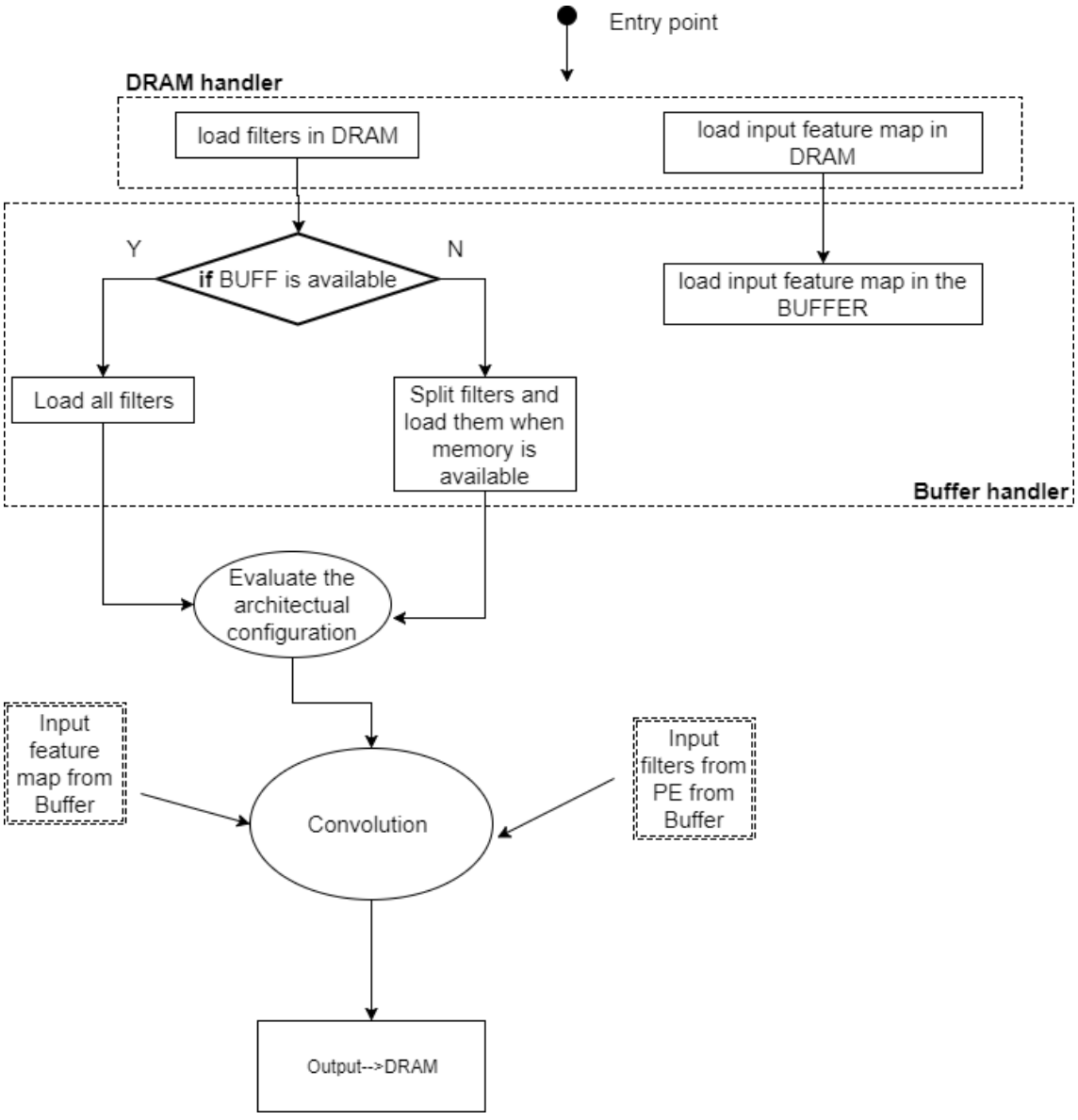


Figure 3.10: NLR dataflow

The dataflow in Figure 3.10 performs the following steps:

1. Input feature maps and kernels are loaded in the main memory.

2. The buffer must contain all input feature maps and at least one complete filter. So input feature maps are completely loaded in the buffer. Then, if there is any available space in the buffer, all space available is filled with a certain number of filters. The filters that are not in the buffer yet are loaded in a second moment, when that part of filters is no longer useful for the computation. For that kind of dataflow it is preferable to have the whole input dataset in the global buffer.

3. At that point the convolution is performed with input filters and input feature maps fetched from buffer.

In the following chapter we will find a comparative analysis between the two dataflows that underlines the main differences in terms of performance.

3.2 Average "pooling" optimization

In the CNN, the input parameters of convolutional layers are input feature maps and filters. This work presents an optimization strategy for input feature maps.
The main idea is to work on the number of operations involved in convolution. In order to do that, we apply an average pooling on the input feature map. The choice of average comes from an accurate evaluation after the comparison between average and max pooling. In order to do that, we selected a subset of images from ILSVRC2012 validation set[3] [41] , applied transformation of this image set and used the Euclidean distance as metric for evaluating the similarity between the original images and the transformed ones.
If $\mathbf{p} = (p_1, p_1, ..., p_n)$ and $\mathbf{q} = (q_1, q_1, ..., q_n)$ are two points in an Euclidean n-space, then the distance (d) from p to q is given by the following formula:

$$d(\mathbf{q}, \mathbf{p}) = \sqrt{\sum_{i=1}^{n}(q_i - p_i)^2} \tag{3.3}$$

The results are shown in the following table:

Technique	**W**	**Euclidean dististance (average value)**
Max "pooling"	2	40.13
	3	56.70
Average "pooling"	2	24.62
	3	30.31

Table 3.3: Max vs Average "pooling"

The Table 3.3 shows that average "pooling" has a better results in terms of similarity, that is the reason why we chose this method.
For sake of clarity, the implemented technique is **improperly** called average "pooling" since, as described in the Chapter 2., a pooling layer reduces the size of the input feature map. In this case, the feature map **preserves the original dimensions**.

The steps involved in that kind of optimization are the following:

1. Choosing a window size (**W**) according to the granularity of the pooling;

2. Parsing the input feature map with a step related to the selected window size.

3. All the elements of selected window are accumulated and the average value is computed.

4. All the elements belonging to the selected window are replaced with their average value.

[3] In the next chapter this choice will be motivated

A practical example is shown in the following pictures:

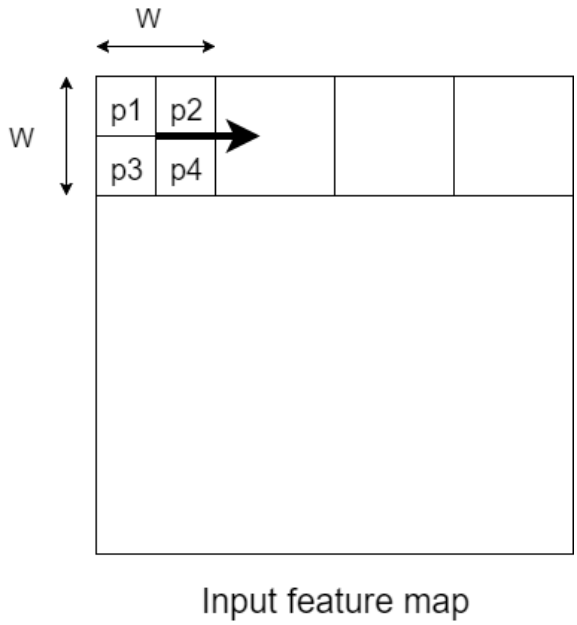


Figure 3.11: Step 1 and 2 of average "pooling" algorithm

In Figure 3.11, we can see the input feature map and the selected window (**W**). In this example W=2. After that we have to compute the average value of the data in that window.

$$P_{avg} = \frac{p1 + p2 + p3 + p4}{4} \tag{3.4}$$

Finally, the average values is assigned to the entire window:

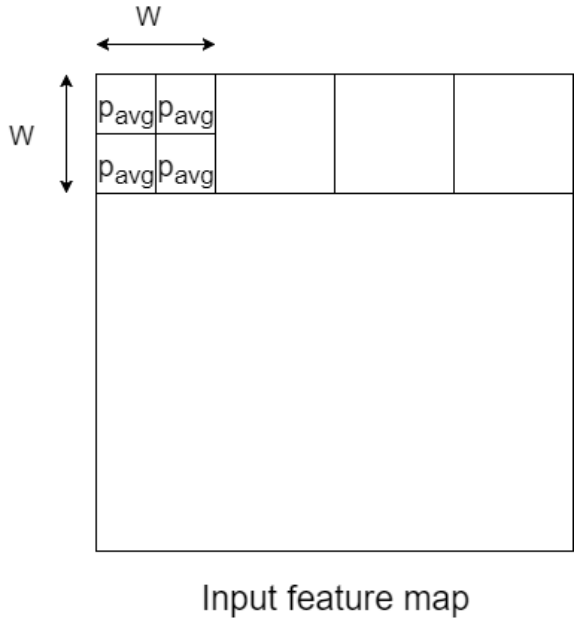
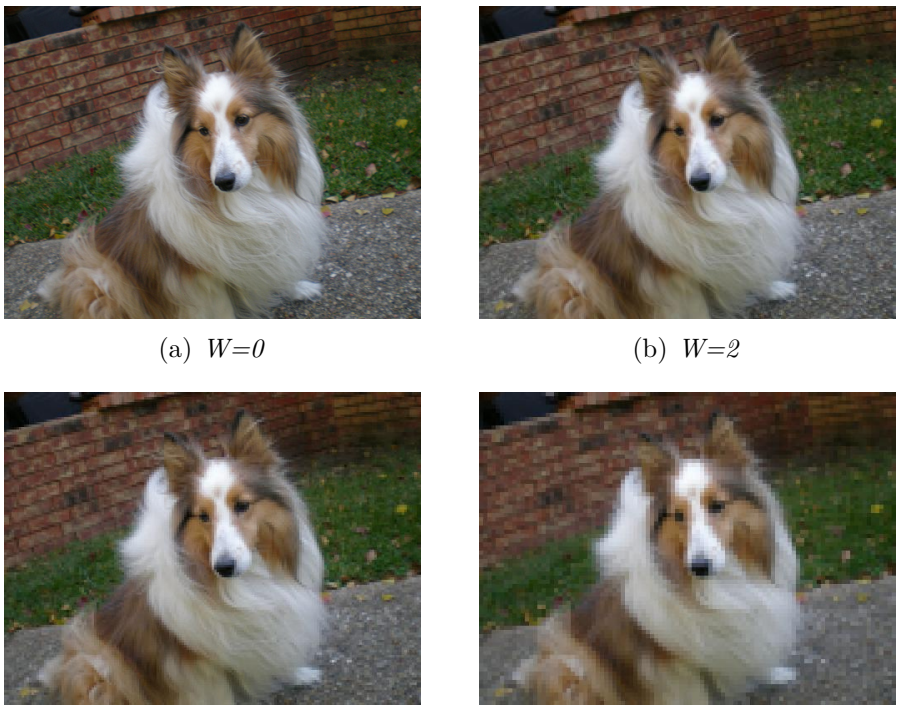


Figure 3.12: Step 4 of average "pooling" algorithm

A visual representation of the results is provided in order to understand the effect of this technique on the input feature maps:



(a) *W=0* (b) *W=2*

(c) *W=3* (d) *W=5*

Figure 3.13: Image before and after average pooling

In Figure 3.13, we can see the visual effect of our technique on a possible example of input image. When W is increased, larger regions have the same average value. The visual effect is the so called "pixel effect", that worsens the initial image quality. The euclidean distance between the original and transformed image grows along with the window size. If these are the drawbacks, what the benefits? We use an example to show them:

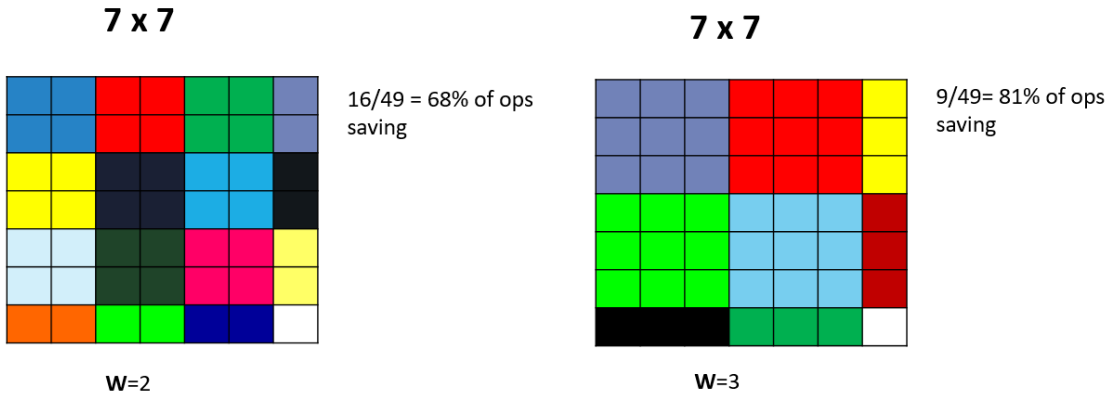


Figure 3.14: Average pooling on 7x7 image window

In Figure 3.14, the area with the same color represents regions of the image with the same pixel value (the average one). A mechanism which can detect a sequence of equal values can be used to skip the operations that have the same input feature map value. In this way we can reduce the number of operations according to the window size. As for the adopted example, we can see the number of operations saving in 3.14. Moreover, we have not only a significant reduction in terms of number of operations, but we can store (and transfer) less data than the original image.
The main benefits of this method are the following:

- Reducing the number of operations;
- Reducing the **effective** dimension of the input feature map.

In the following chapter the impact of this optimization on the developed dataflows will be described.

4
Evaluation and Results

In this chapter we analyze the performances of the dataflows described in the previous chapter and the optimization developed and applied on those dataflows. The chapter is divided in three sections. In the first section we show the model for estimating performances. Then, the performances of the two dataflows. Finally, the results concerning the application of the average pooling on the developed dataflows.

4.1 Experimental setup

In this section we will describe the performance model for our simulations and all the configurations and setting used for the developed dataflow simulation.

4.1.1 Energy/Latency model

The main purpose of this work is to describe an energy model in order to test a designed dataflow. Moreover, we know that data transfer in the memory hierarchy and MAC operations are the main contributions to the energy consumption of a convolutional layer. For the energy characterization we use data from [25]:

Memory	**Norm. Energy cost**
DRAM	200
Global Buffer (>100kB)	6
Inter-PE buffer (1-2mm)	2
Register File	1

Table 4.1: Normalized energy cost

The values in Table 4.1 are related to works [25, 43, 44], based on 65nm technology. Those reference values are used to compute the overall energy cost with the following equation.

Total Energy = # access x (Norm. Energy access cost)
= #dram access x (DRAM cost) +#buffer access x (BUFFER cost)
+#inter-pe buffer x (I-PE buff cost) +#RF access x (RF cost)

Moreover, for latency we count memory accesses and operations weighted for their normalized cost in terms of clock cycles. For the proposed results this cost is unitary. In the following section we describe the CNN used in our experiments.

4.1.2 AlexNet

AlexNet [42] is a convolutional neural network compared the first time in the ImageNet Large Scale Visual Recognition Challenge in 2012, for the classification task. ILSVRC evaluates algorithms for object detection and image classification on large scale. One high level motivation is to allow researchers to compare progress in detection across a wider variety of objects taking advantage of the quite expensive labeling effort. Another motivation is to measure the progress of computer vision for large scale image indexing for retrieval and annotation [45].
The following picture shows the AlexNet structure:

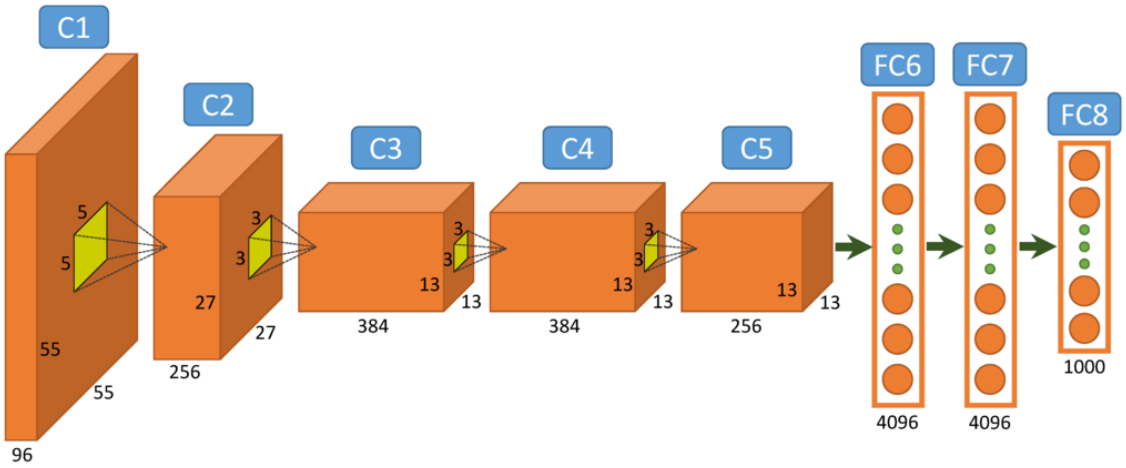


Figure 4.1: AlexNet structure [42]

In Figure 4.1, there are AlexNet layers, five Conv Layers and three Fully-Connected layers. It was implemented in CUDA for GPUs and in a second moment for CPUs. A pre-trained model is available for the Caffe framework. This model obtains a top-1 accuracy 57.1% and a top-5 accuracy 80.2% on the validation set, using only the center crop [1]. The Caffe implementation of this model is our reference model [1]. The Conv Layers are the most interesting ones for our work. The following table shows the parameters involved in each convolutional layer:

	M	W	H	K	C	OH	OK	S
CONV1	96	227	227	11	3	55	55	4
CONV2	256	27	27	5	96	27	27	1
CONV3	384	13	13	3	256	13	13	1
CONV4	384	13	13	3	384	13	13	1
CONV5	256	13	13	3	384	13	13	1

Table 4.2: AlexNet convolutional layers

[1] `https://github.com/BVLC/caffe/tree/master/models/bvlc_alexnet`

4.2 Dataflow results

In the previous section we have described the performance model and the convolutional layers involved in our experiments. The last configuration is the architectural configuration. We want to find a suitable configuration for both dataflows and for all CONV layers. In Window Stationary Dataflow, as a project decision, at least one complete filter must be in the buffer, so the total storage area (#PE x RF_size) $\geq$ C x K x K . In order to do that, we have to analyze the parameters involved in convolution for each layer:

	K	**C**	**Total**
CONV1	11	3	363
CONV2	5	96	900
CONV3	3	256	2304
CONV4	3	384	3456
CONV5	3	384	2304

Table 4.3: Filter size for each Conv layer: Channels and Kernel width/height

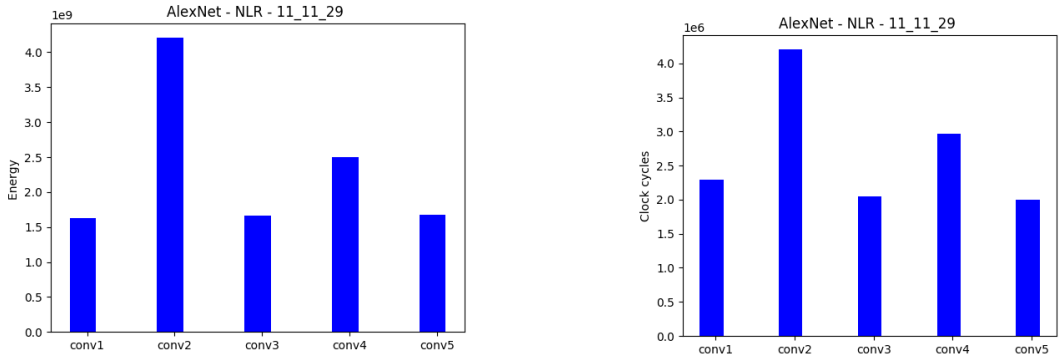
We chose the number of PEs in order to fit the largest kernel (11 x 11). So choosing the PE plan size equal to 121, the resulting number of registers per PE is 29. In the following table architectural parameters are summarized:

Parameter	**Value**
DRAM	Not relevant
Buffer	600kB
Inter-PE buffer	1kB
#PE	121
#regs per PE	29

Table 4.4: Configuration for simulation experiments

4.2.1 NLR Results

After describing the simulation framework and the configuration, we can see the results. In the following picture we can find the energy and latency of AlexNet Conv layers:



(a) *Energy* (b) *Latency*

Figure 4.2: NLR - Performance

The energy/latency distribution in Figure 4.2 is related to the sizes of the convolutional layers. The most energy consumption one is the layer that processes the 96 input feature maps and 256 filters 5 x 5. In the other layers the input feature maps and/or filters are smaller. In the following picture we can find the energy distribution:

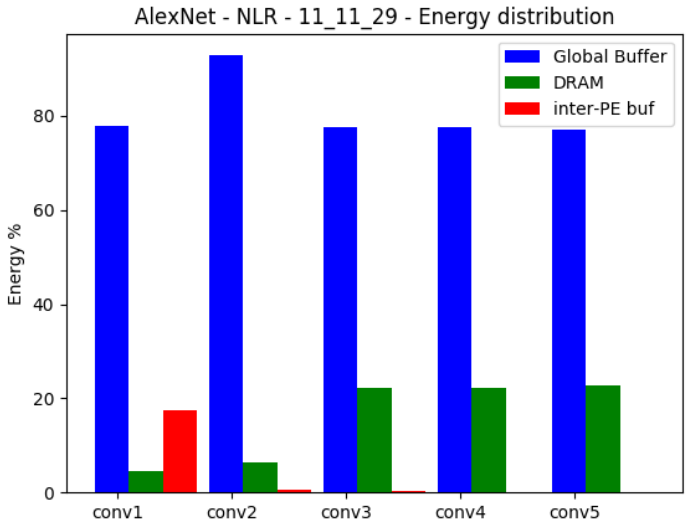


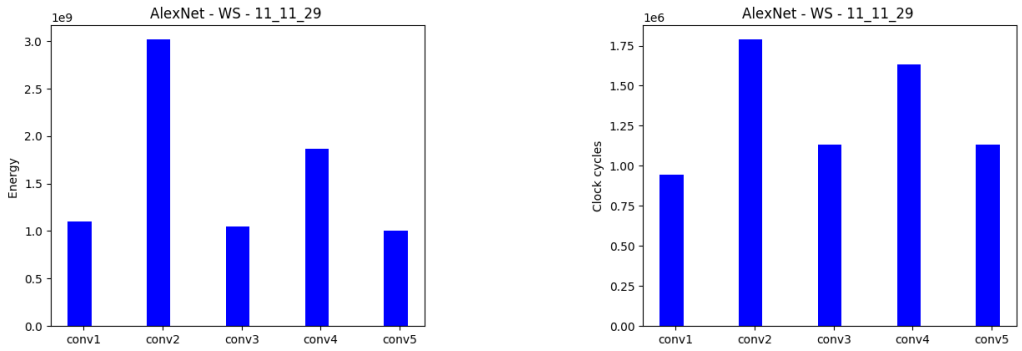
Figure 4.3: NLR - Energy distribution

From the results in Figure 4.3, for this dataflow the buffer is the memory element used the most, since both input feature maps and filters are fetched from the buffer

for convolution.

4.2.2 WS Results

Now, we analyze the WS dataflow. The following picture shows the results:



(a) *Energy* (b) *Latency*

Figure 4.4: WS - Performance

The pattern of the performance in Figure 4.4 is similar to the previous dataflow in 4.2.

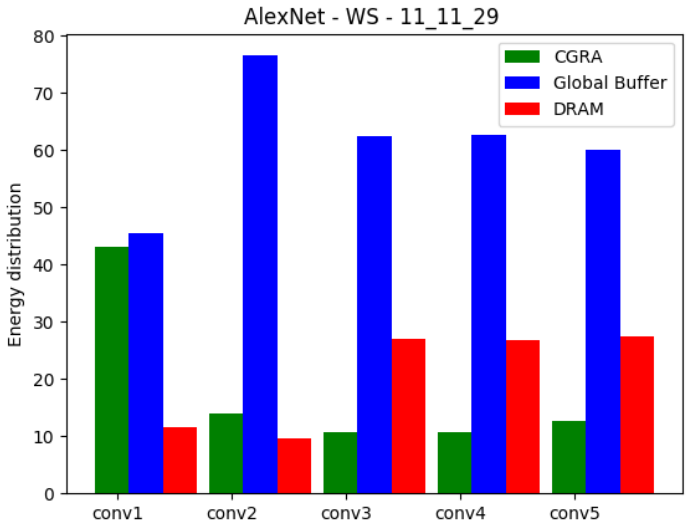


Figure 4.5: WS - Energy distribution

For CGRA we mean the sum of inter-PE buffer and local register files. In this dataflow, compared to NLR, the energy distribution changes. The buffer is still the most used memory element, but the usage of RF and inter-PE plays a significant role in the energy distribution. Moreover, we can observe an anomalous usage of inter-pe buffer in the CONV1 layer. We have to remember that inter-PE buffer is used to accumulate partial sums after the MACs computation. So, the number

of accesses depends on the kernel size, 11x11 in that case: for each "window" the number of accesses is 11x11, the biggest size among CONV layers.

4.2.3 Dataflows comparison

At this point of the work we have evaluated the performance of each dataflow and now it is possible to compare them.

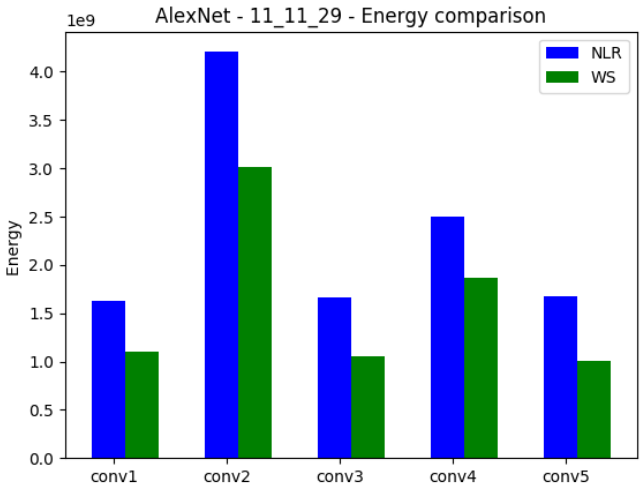


Figure 4.6: WS vs NLR - Energy comparison

From Figure 4.6 the differences between the two dataflow in terms of energy emerge. The fact that in WS dataflow we fetch data from the register file and not from the buffer implies a significant energy saving.

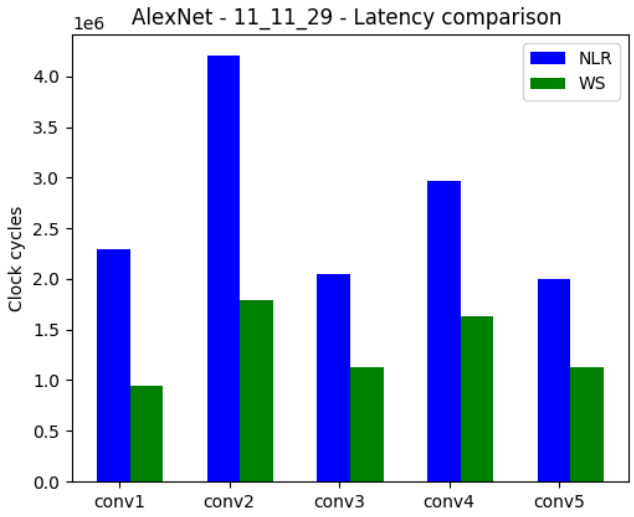


Figure 4.7: WS vs NLR - Latency comparison

In terms of latency, the factor that allows us to speed up the operations lies in the fact that we do not need to fetch weights from buffer each cycle of computation, but they are stationary in the register files of PE. In the following table we quantify the amount of performance saving:

	Energy saving (%)	**Latency saving (%)**
CONV1	32.61	58.89
CONV2	28.24	57.47
CONV3	37.04	44.81
CONV4	25.34	44.89
CONV5	39.79	43.37
Average	32.60	49.89

Table 4.5: Energy/latency saving comparison

From Figure 4.6, 4.7 and Table 4.5 what emerge are the energy/latency differences between the two dataflows. The average energy saving is the 32.60% with a corresponding latency saving of 49%.

In WS dataflow the factors that allow to save energy/latency are:

- Maximization of convolutional and weights reuse due to the usage of PE register files and the exploitation of the inter-PE buffer to accumulate partial sums.
- The usage of a tree of adders to speed up the partial sums.

4.3 Average "pooling" results

In this section we analyze the results that come from the application of average pooling technique on our network. The first consideration concerns understanding the impact of average pooling on the number of MAC operations. In the following picture we can find the MAC operations per Conv layer:

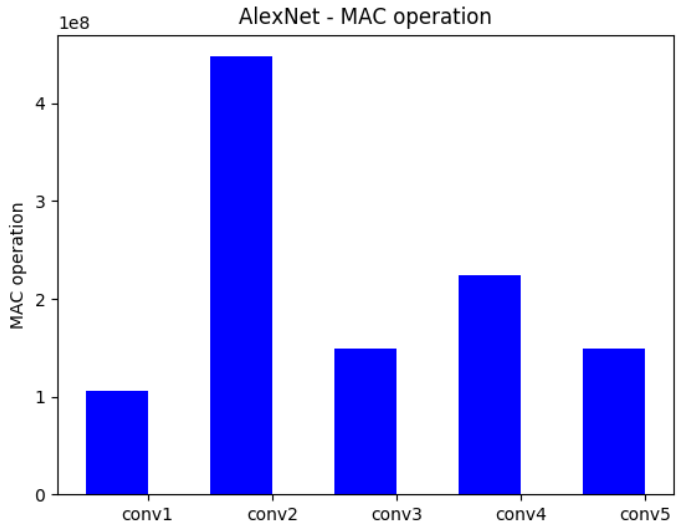


Figure 4.8: MAC operations

From Figure 4.8 it emerges that some layers present more MAC operations than others. In order to understand the goodness of our approach and at the same time evaluate the drawbacks, we analyze the first convolutional layer of AlexNet.

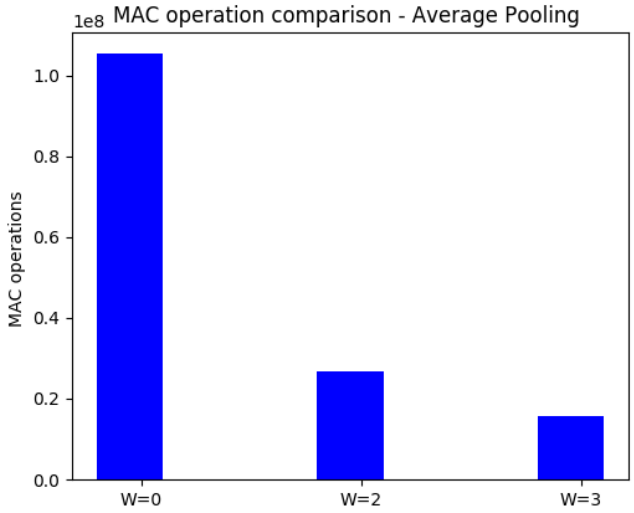
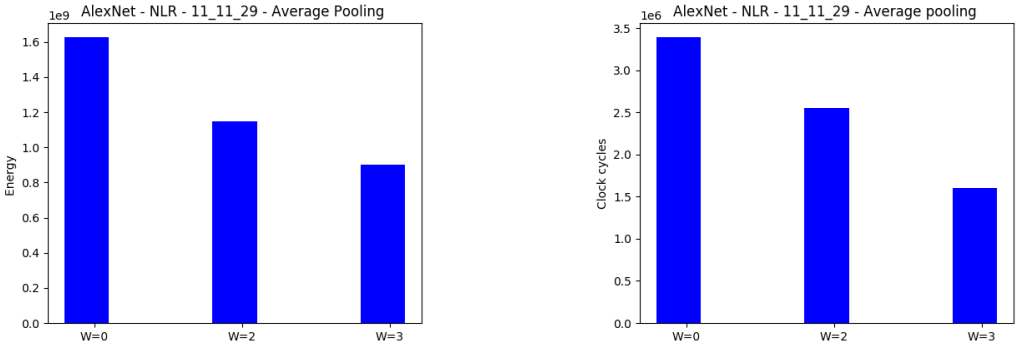


Figure 4.9: MACs operation comparison

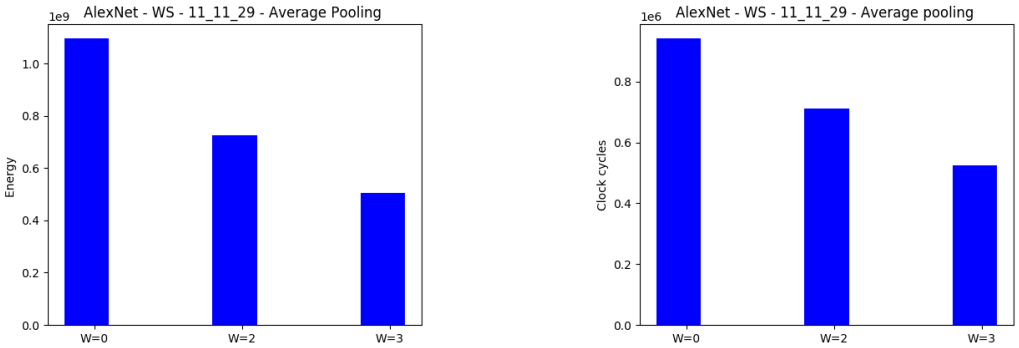
In Figure 4.9, we can see the effect of average pooling in terms of operations reduction on the CONV1 layer, the input image. The equal pixels recognition in a selected window has an impact on MAC operation reduction. Moreover, we are interested in evaluating the impact of that optimization on the developed dataflows.



(a) *Energy* (b) *Latency*

Figure 4.10: NLR - Average Pooling - Performance

The impact on the performance on the NLR dataflow is perfectly aligned with what we expected. In 4.10 we can see a 36.94% energy saving and 38.75% for latency saving.



(a) *Energy* (b) *Latency*

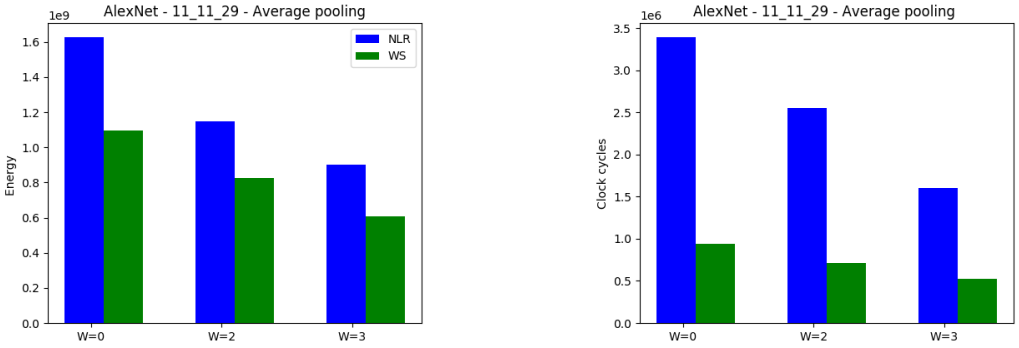
Figure 4.11: WS - Average Pooling - Performance

WS dataflow performances patterns follow the NLR one in 4.10. In the following table we resume the average savings.

	Energy saving (avg%)	**Latency saving (avg%)**
W=2	~31	~28
W=3	~48	~45

Table 4.6: Average energy and latency savings for the two dataflows

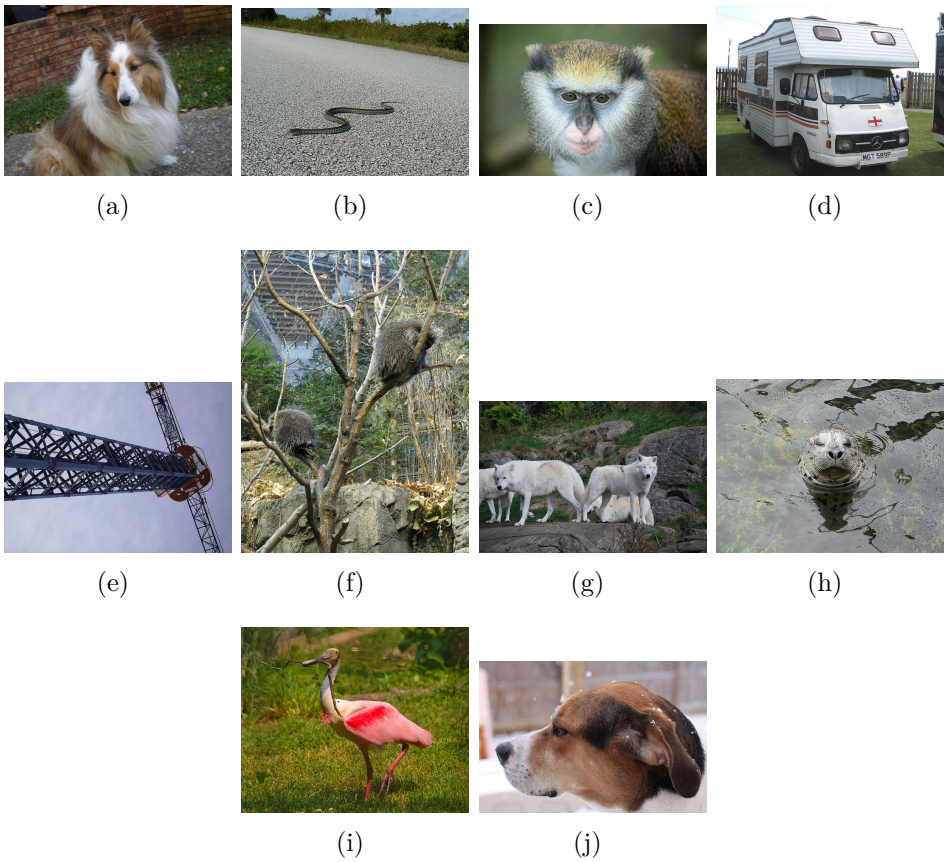
In Table 4.6, we can find the average energy/latency saving between the two dataflows varying the window size. The following picture shows the difference of performances between the two dataflows that remain constant.



(a) *Energy* (b) *Latency*

Figure 4.12: Dataflow - Average Pooling - Comparison

In the engineering field "there is no free lunch". The drawback of this technique is the accuracy loss of the overall network since the transformed image is more different than the original, varying the windows size. In order to estimate the AlexNet accuracy, we use the ILSVRC2012 validation set on classification task.



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

(e) (f) (g) (h)

(i) (j)

Figure 4.13: Image samples from ILSVRC2012 validation set

In Figure 4.13, there are some samples from ILSVRC2012 validation set used for estimating the accuracy of our network. The following validation flow is used:

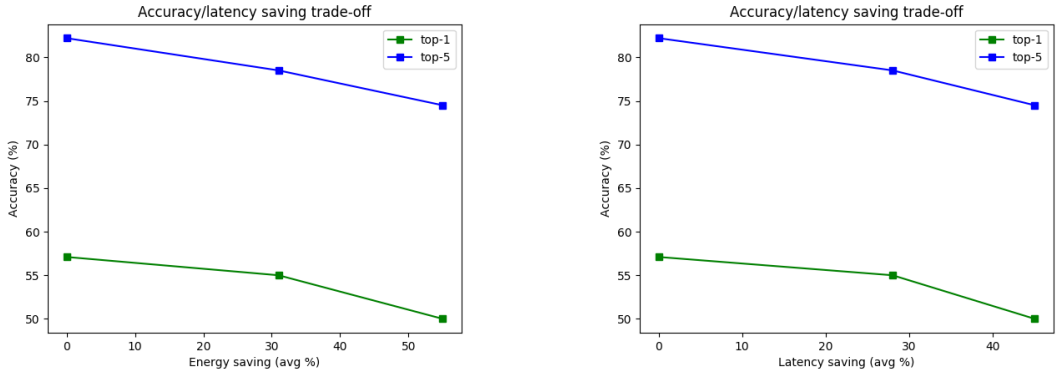
- Fetch and transform original image using average pooling;
- Perform classification;
- Compare output label with the correct one and accumulate the result.

With this flow we can extract the accuracy of AlexNet after average pooling. The following table resumes the results:

	top-1 accuracy (%)	**top-5 accuracy (%)**
Original	57.1	80.2
W=2	55.0	78.5
W=3	50.6	74.5

Table 4.7: Accuracy comparison

In Table 4.7, there is the AlexNet accuracy applying average "pooling" on the same validation set with different windows size. We can notice a 2.1% of accuracy loss with W=2 and 6.5% with W=3. The following pictures show the trend of performances saving / accuracy trade-off.



(a) *Accuracy/Energy* (b) *Accuracy/Latency*

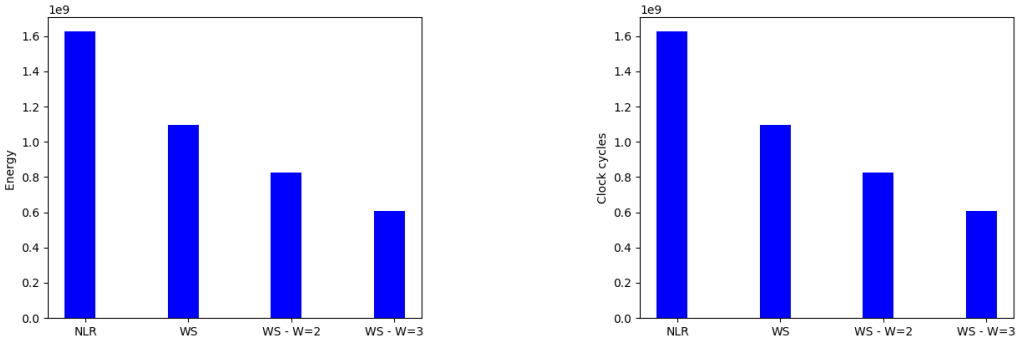
Figure 4.14: Accuracy/Performances

In Figure 4.14, it is shown the relationship between accuracy and energy saving. The trend is almost equal for both dataflow.
The following table resumes the ratio between performance saving and accuracy lost.

		Accuracy loss (%)	**Energy saving (%)**	**Latency saving (%)**
top-1	W=2	2.1	31	28
		1.7	48	45
top-5	W=3	6.5	31	28
		5.7	48	45

Table 4.8: Accuracy loss/Energy saving

Finally, this technique guarantees a good balancing between accuracy and performance according to data in Table 4.8.
In the last analysis we use NLR dataflow as reference and we compare the energy performance of that dataflow with the WS dataflow with and without average pooling.



(a) *Energy* (b) *Accuracy/Latency*

Figure 4.15: Latency

In this analysis we want to evaluate the performance benefit of using WS optimized dataflow. From 4.15 emerges that the achieved energy saving (from NLR to optimize WS) is 49.2% when W=2 and 62.74% with W=3.

5
Conclusion

This chapter outlines the achievements of this work and provides some directions for future studies.

5.1 Achievement

The main reason behind this work is to present a dataflows analysis for ConvNet on a heterogeneous platform. This work offers a detailed description of the dataflows and the reasons behind our design choices. The strengths of the Window Stationary dataflow lie in:

- The maximum exploitation of all hardware resources available for convolutional and weights reuse;

- The maximum exploitation of near-memory computing in the usage of the nearest memory resource to accumulate MAC results and partial sums.

Those features allow to have an energy saving (compared to the baseline dataflow) of 33% (on average) and latency saving of 49% (on average).
The custom simulation framework has been developed in order to be integrated to deep learning existing tools (i.e Caffe, Tensorflow, Theano). It uses textual description for the architectural configuration and the ConvNet layer parameters. From those tools we extract the main parameters used as inputs in the performance simulation framework.
A further analysis has been done in order to exploit the data statistic of the convolutional inputs. The adopted approach can be integrated in our simulation engine written in Python. An energy saving / accuracy loss trade-off emerges from the experimental results. An accuracy loss of 2.1% (compared to the baseline dataflow) leads to an energy saving of 31% (on average).

5.2 Future Work

The results are promising but further studies are required in order to obtain a complete and reusable simulation framework. The author believes the following suggestions may improve this work:

1. The current simulation software is not completely modular. Most of the software routines and classes are reusable. But the core of the dataflow modeling is dataflow depending. Further efforts are required to implement a new dataflow using this framework. This problem is due to the facts that dataflows could be very different from each other in the architectural exploitation and in data elaboration.

2. The developed optimization technique is tested on the first layer of AlexNet since the accuracy evaluation is easy with the existing tools. Further experiments on intermediate outputs would be interesting to understand the overall optimizations.

3. The reference architecture used in our work perfectly addresses a stationary oriented filter mapping. Moreover, the inter-PE buffer allows the near-memory computation locality. However, it would be interesting to test the developed dataflows on different architectures in order to evaluate the performances.

4. The choice of using a dataflow that belongs to the window stationary dataflow class is due to the fact that this is the dataflow that presented the widest margin of improvement compared to the other classes. But it could be interesting to implement other dataflows that belong to other classes, in order to complete the performance analysis.

Bibliography

[1] Artificial neural network, `https://en.wikipedia.org/wiki/Artificial_neural_network`

[2] W. S. McCulloch and W. Pitts, *"A logical calculus of the ideas immanent in nervous activity"*, The bulletin of mathematical biophysics, vol. 5, no. 4, pp. 115–133, 1943

[3] Y. LeCun, Y. Bengio, and G. Hinton, *"Deep learning"*, Nature, vol. 521, no. 7553, 2015.

[4] Rosenblatt, F. (1958). *"The Perceptron: A Probabilistic Model For Information Storage And Organization In The Brain"*. Psychological Review. 65 (6): 386–408.

[5] Ivakhnenko, A. G.; Grigor'evich Lapa, Valentin (1967). *" Cybernetics and forecasting techniques"*. American Elsevier Pub. Co.

[6] A. Karpathy. (2016). Stanford University *CS231n: Convolutional Neural Networks for Visual Recognition*. `http://cs231n.github.io`.

[7] *Open neural network: High Performance library for Advanced Analytics* , `http://www.opennn.net/`

[8] Guo, Yanming and Liu, Yu and Oerlemans, Ard and Lao, Song-Yang and Wu, Song and S. Lew, Michael. (2015). *Deep learning for visual understanding: A review. Neurocomputing*. 187. . 10.1016/j.neucom.2015.09.116.

[9] Ciresan, Dan; Meier, Ueli; Schmidhuber, Jürgen (June 2012). *"Multi-column deep neural networks for image classification"*.IEEE Conference on Computer Vision and Pattern Recognition. New York, NY: Institute of Electrical and Electronics Engineers (IEEE), 2012.

[10] Nair, Vinod; Hinton, Geoffrey E. (2010), *"Rectified Linear Units Improve Restricted Boltzmann Machines"*, 27th International Conference on International Conference on Machine Learning, ICML'10, USA: Omnipress, pp. 807–814, ISBN 9781605589077.

[11] Xu, Bing; Wang, Naiyan; Chen, Tianqi; Li, Mu. *"Empirical Evaluation of Rectified Activations in Convolutional Network"*. arXiv, 2015.

[12] The MathWorks, Inc. (2016). Neural Network Toolbox - MATLAB, [Online]. Available:`http://mathworks.com/products/neural-network`.

[13] R. Al-Rfou, G. Alain, A. Almahairi, et al., *"Theano: A python framework for fast computation of mathematical expressions"*, CoRR, vol. abs/1605.02688, 2016.

[14] : Martin Abadi, Ashish Agarwal, Paul Barham, et al., "TensorFlow: Large-scale machine learning on heterogeneous systems", Software available from `tensorflow.org`, 2015

[15] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed and D. Anguelov, D. Erhan, V. Vanhoucke and A. Rabinovich, *"Going Deeper with Convolutions"*,CoRR,abs/1409.4842, 2014.

[16] K. Simonyan and A. Zisserman, *"Very deep convolutional networks for large-scale image recognition"*, ArXiv preprint arXiv:1409.1556, 2014.

[17] F. N. Iandola, M. W. Moskewicz, K. Ashraf, et al., *"SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1mb model size"*, ArXiv:1602.07360, 2016.

[18] NVIDIA Corporation. (2016). NVIDIA GeForce GTX Titan X Secifications. Available: `http://www.geforce.com/hardware/desktop-gpu/geforce-gtx-titan-x/` specifications

[19] David Gschwend,*"ZynqNet: An FPGA-Accelerated Embedded Convolutional Neural Network"*, Master Thesis, August 2016, ETH Zürich.

[20] Nurvitadhi, Eriko and Venkatesh, Ganesh and Sim, Jaewoong and Marr, Debbie and Huang, Randy and Ong Gee Hock, Jason and Liew, Yeong Tat and Srivatsan, Krishnan and Moss, Duncan and Subhaschandra, Suchit and Boudoukh, Guy; *"Can FPGAs Beat GPUs in Accelerating Next-Generation Deep Neural Networks?"*, Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA '17.

[21] TheNextPlatform, *"Can FPGAs Beat GPUs in Accelerating Next-Generation Deep Learning?"*, `https://www.nextplatform.com/2017/03/21/can-fpgas-beat-gpus-accelerating-next-generation-deep-learning/`, 2017.

[22] Y. Chen, T. Luo, S. Liu, et al., *"DaDianNao: A machine-learning supercomputer"*, in Proceedings of the 47th Annual IEEE/ACM International

Symposium on Microarchitecture, IEEE Computer Society, 2014.

[23] Zhang, Chen and Li, Peng and Sun, Guangyu and Guan, Yijin and Xiao, Bingjun and Cong, Jason, *"Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks"*, Proceedings of the 2015 ACM/SIGDA ISFGA.

[24] Jouppi, Norman P. and Young, Cliff and Patil, Nishant and Patterson, David and Agrawal, Gaurav and Bajwa, Raminder and Bates, Sarah and Bhatia, Suresh and Boden, Nan and Borchers, Al and Boyle, Rick and Cantin, Pierre-luc and Chao, Clifford and Clark, Chris and Coriell, Jeremy and Daley, Mike and Dau, Matt and Dean, Jeffrey and Gelb, Ben and Ghaemmaghami, Tara Vazir and Gottipati, Rajendra and Gulland, William and Hagmann, Robert and Ho, C. Richard and Hogberg, Doug and Hu, John and Hundt, Robert and Hurt, Dan and Ibarz, Julian and Jaffey, Aaron and Jaworski, Alek and Kaplan, Alexander and Khaitan, Harshit and Killebrew, Daniel and Koch, Andy and Kumar, Naveen and Lacy, Steve and Laudon, James and Law, James and Le, Diemthu and Leary, Chris and Liu, Zhuyuan and Lucke, Kyle and Lundin, Alan and MacKean, Gordon and Maggiore, Adriana and Mahony, Maire and Miller, Kieran and Nagarajan, Rahul and Narayanaswami, Ravi and Ni, Ray and Nix, Kathy and Norrie, Thomas and Omernick, Mark and Penukonda, Narayana and Phelps, Andy and Ross, Jonathan and Ross, Matt and Salek, Amir and Samadiani, Emad and Severn, Chris and Sizikov, Gregory and Snelham, Matthew and Souter, Jed and Steinberg, Dan and Swing, Andy and Tan, Mercedes and Thorson, Gregory and Tian, Bo and Toma, Horia and Tuttle, Erick and Vasudevan, Vijay and Walter, Richard and Wang, Walter and Wilcox, Eric and Yoon, Doe Hyun, *"In-Datacenter Performance Analysis of a Tensor Processing Unit"*, ISCA 2017

[25] Chen, YuHsin and Krishna, Tushar and Emer, Joel and Sze, Vivienne, *"Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks"*, IEEE International SolidState Circuits Conference, ISSCC 2016, Digest of Technical Papers, 2016,262-263.

[26] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, *"Understanding Sources of Inefficiency in General-purpose Chips"*, ISCA, 2010.

[27] M. Peemen, A. A. A. Setio, B. Mesman, and H. Corporaal, *"Memory-centric accelerator design for Convolutional Neural Networks"*, IEEE ICCD, 2013.

[28] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, *"Deep Learning with Limited Numerical Precision"*, CoRR, vol. abs/1502.02551, 2015.

[29] 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"* ,

ISCA, 2015.

[30] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, *"Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks"*, FPGA, 2015.

[31] 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"*, MICRO, 2014

[32] S. Park, K. Bong, D. Shin, J. Lee, S. Choi, and H.-J. Yoo, *"A 1.93TOPS/W scalable deep learning/inference processor with tetra-parallel MIMD architecture for big-data applications"*, IEEE ISSCC, 2015.

[33] L. Cavigelli, D. Gschwend, C. Mayer, S. Willi, B. Muheim, and L. Benini, *"Origami: A Convolutional Network Accelerator"*, GLSVLSI, 2015.

[34] Jia, Yangqing and Shelhamer, Evan and Donahue, Jeff and Karayev, Sergey and Long, Jonathan and Girshick, Ross and Guadarrama, Sergio and Darrell, Trevor, "*Caffe: Convolutional Architecture for Fast Feature Embedding*", arXiv preprint arXiv:1408.5093, 2014.

[35] Wikipedia: Multidimensional discrete convolution `https://en.wikipedia.org/wiki/Multidimensional_discrete_convolution`.

[36] S. Han, J. Pool, J. Tran, W. J. Dally, *"Learning both Weights and Connections for Efficient Neural Networks"*,arXiv:1506.02626v3 [cs.NE], 30 Oct 2015.

[37] Joel Emer, Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, *"DNN Model and Hardware Co-Design"*, ISCA Tutorial 2017.

[38] Tien-Ju Yang, Y.n Chen, V. Sze, "Designing Energy-Efficient Convolutional Neural Networks using Energy-Aware Pruning", ISCA 2017.

[39] Gysel, Philipp and Motamedi, Mohammad and Ghiasi, Soheil, textit"Hardware-oriented Approximation of Convolutional Neural Networks", arXiv preprint arXiv:1604.03168, 2016.

[40] E. H. Lee, D.Miyashita, E. Chai, B. Murmann, S. S. Wong, *LogNet: Energy-efficient neural networks using logarithmic computation*, ICASSP 2017.

[41] Deng, J. and Dong, W. and Socher, R. and Li, L.-J. and Li, K. and Fei-Fei, L., *"ImageNet: A Large-Scale Hierarchical Image Database"*, CVPR09, 2009.

[42] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, *"ImageNet*

Large Scale Visual Recognition Challenge", IJCV, vol. 115, no. 3, pp. 211–252, 2015.

[43] M. Horowitz, *"Computing's energy problem (and what we can do about it)"*, IEEE ISSCC, 2014

[44] K. T. Malladi, B. C. Lee, F. A. Nothaft, C. Kozyrakis, K. Periyathambi, and M. Horowitz, *"Towards energy-proportional datacenter memory with mobile dram"*, ISCA, 2012.

[45] ImageNet Large Scale Visual Recognition Challenge, `http://www.image-net.org/challenges/LSVRC/`