POLITECNICO DI TORINO

Master Degree in Electronic Engineer

Master Degree Thesis

# Design of optimized architectures for non-linear convolutional neural networks

Supervisors

Prof. Maurizio MARTINA

Prof. Guido MASERA

Prof. Stefano MARSI

Candidate

Giuseppe AIELLO

December 2021

## Abstract

Nowadays, Machine Learning (ML) has become one of the most important topics of research because of the massive use into many applications such as self-driving cars, speech recognition, email spam recognition and in particular image recognition and processing.
The involving of the ML in digital image processing can be used for different target applications, among them for example there are: medical visualization (to improve the medical imaging), pattern recognition, noise reduction, image enhancement, and so on.
In this thesis a new type of neural network (NN) for image processing has been used. In fact, instead of using filters with fixed weights like in standard convolutional layers, this new NN uses space-variant coefficients. This new convolutional layer leads to better change its behaviour depending on the spatial characteristic of the input image. Since the spatial dependence introduces a non-linear behaviour to the layer, a Non-Linear-Convolution (NLC) replaces the standard linear convolution of a CNN.
Networks including NLC achieve performance that are comparable or better respect to the canonical Convolutional Networks, moreover, they require fewer layer and less input feature respect to the second one.
This thesis works focus on the implementation of the layer of a NLCN into field programmable gate arrays (FPGAs), which are one of the most important platforms to accelerate the ML inference. FPGAs, in fact, bring many advantages such as high parallelism, low power consumption, dedicated optimized hardware for digital signal processing (DSPs) and so on.
Unfortunately, all these advantages don't come without a price. In fact, while the fewer layers of the NLCN respect to the classical CNNs allows to reduce the number of features (which is an optimal solution for embedded accelerators which have very reduced resources), the overall complexity of the single layer of the NLCN is greater respect to the CNNs one.
So, the layer design has been designed to achieve a suitable trade-off between memory transaction and computation. Indeed, since the memory usage in the NLCN is mainly related to the space for internal computation, to store the parameters and to save the intermediate data, some techniques are adopted to find an optimal balance between off-chip memory transaction and computation. Among them, there are for example loop tiling and loop unrolling, in which the former allows to reduce the amount of on-chip memory required into FPGAs and the latter allows to speed up the execution of nested loops.

# Acknowledgements

Un primo ringraziamento va a i miei relatori, il professore Maurizio Martina, il professore Guido Masera e il professore Stefano Marsi per avermi dato la possibilità di condurre questo percorso di tesi con un sempre e presente supporto tecnico.

Un grazie enorme alla mia famiglia, mia mamma, mio papà e mia sorella che mi hanno sempre supportato nelle mie scelte e che mi hanno dato un conforto speciale anche nei momenti più difficili.

Infine un grazie speciale a tutti quegli amici che mi hanno accompagnato in questi anni di percorso universitario, e che hanno contribuito a rendere la mie giornate migliori, facendomi sentire sempre a casa.

# Table of Contents

# List of Tables

# List of Figures

# Acronyms

**AI** Artificial Intelligence

**AF** Activation Function

**ML** Machine Learning

**NN** Neural Network

**NLC** Non Linear Convolution

**CNN** Convolutional Neural Network

**NLCN** Non Linear Convolution Network

**FPGA** Field Programmable Gate Array

**ASIC** Application Specific Integrated Circuit

**GPU** Graphic Processing Unit

**CPU** Central Processing Unit

**DSP** Digital Signal Processing

**IC** Integrated Circuit

**MAC** Multiply-Accumulate

**PE** Processing Element

**BRAM** Block Random Access Memory

| | |
|---|---|
| **DDR** | Double Data Rate |
| **RAM** | Random Access Memory |
| **LUT** | Look-up Table |
| **CCR** | Computation to Communication Ratio |
| **ReLU** | Rectified Linear Unit |
| **Tanh** | Hyperbolic Tangent |

# Chapter 1

# Introduction

## 1.1 General principles

In the last years the growing of AI applications has been supported by an intense research work involved into the creation of new kinds of NN able to perform even better respect the previous ones. A neural network is a subset of the machine learning that imitates the behaviour of the human brain in order to solve common problems, recognize patterns and so on. In fact, an artificial neural network consists of different node layers, which mimic the way that biological neurons communicate with one another.

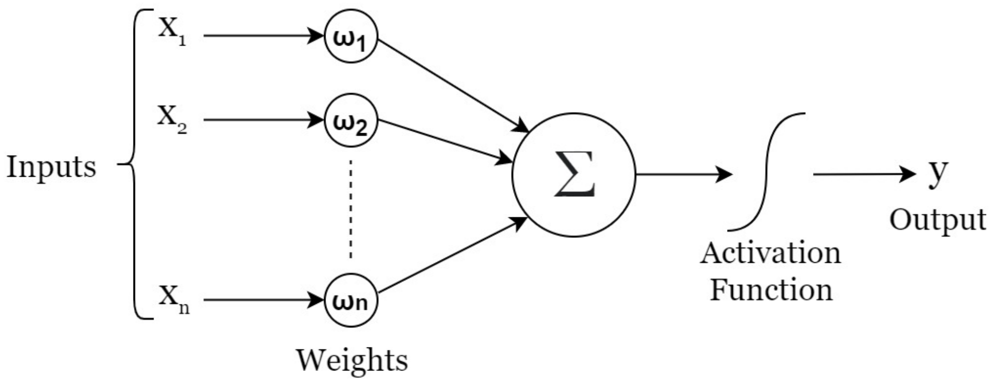


**Figure 1.1:** Structure of artificial neuron

For any node, or artificial neuron, is associated a set of weights and a threshold. In cases in which the output of an artificial neuron has a value above the threshold assigned, the data is propagated to the next nodes, otherwise no data is send.

Among the different types of neural networks, convolutional neural networks (CNNs) are usually utilized for pattern recognition, image recognitions, computer vision and so on.
Typically, a CNN is based on the implementation of:

- Convolutional layer
- Activation function
- Batch normalization
- Pooling layer
- Fully connected layer

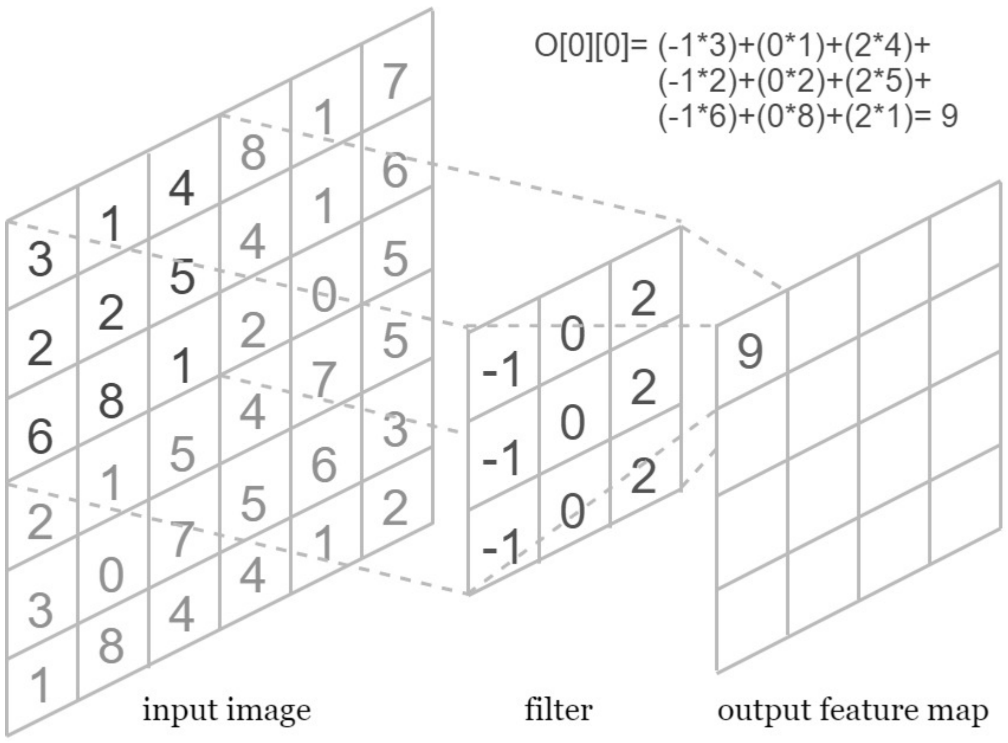


**Figure 1.2:** 2D convolution

Focusing on the convolutional layer, that is the cornerstone of CNNs, the main elements that characterized this type of layer are the input data or input feature map, the kernel or filter and the output feature map. Assuming that the input data is a matrix of pixels in 3D, as consequence it is has three dimensions: height, width and depth. Taking as example a 2D input data (Figure 1.2), the kernel is

applied to a portion of the input image, and a dot product between the pixels and the weights is performed. After that the filter slides along the images by a stride taking as input another portion of pixels and a new dot product operation is repeated. As final result, the overall dot products between weights and pixels fill an output map called output feature map.
In this thesis a new neural network for image processing [1] has been proposed. As novelty, concerning the convolutional layer, instead of using kernels with fixed coefficients, it uses, for a given input channel, space-variant weights that depend on the input data. This allows to better fit the behaviour of the network depending on the characteristic of the input data. In digital images, in fact, the pixels that are spatially near the one to each other have high correlated value. Furthermore, in digital natural images, non-linear functions can better achieve an inter-dependency respect to the simple convolution-based linear operators and as consequence they can better exploits the connections with adjacent pixels of an image.
With convolutional neural network (CNN), in fact, there is the necessity to implement several classical linear convolutional layers in order to perform complex tasks. Thereafter the output of each layer, in turn, typically goes through a point-wise non-linear function, such as Sigmoid, Hyperbolic Tangent or ReLU (Rectified Linear Unit).
On the other hand, the CNNs that use more complicated non-linearities notice an increase of the complexity of the layer, because on the increase of the number of parameters needed and on the increase of the execution time both on the training phase and in the inference phase. So, in the new layer proposed in [1], a non-linear convolution (NLC) replaces the standard linear convolution. In addition, to maintain a suitable dynamic range of the data between input and output features, a normalization of the spatially varying weights is computed.

## 1.2 Purpose of the thesis

The aim of this thesis is to study which are the possible optimized HW architectures of a NLC layer that could be implemented into FPGAs. In fact, this layer is the most computationally onerous layer of the Non-Linear Convolution Network (NLCN). Efforts have been made to reduce the number of access to the external memory connected with the FPGA during the computations. In fact, the exploitation of on-chip memory reduces the power consumption and the overall latency leading on a speed-up of the inference task. Unfortunately, since the amount of on-chip memory inside FPGAs is quite limited, one of the most problem is related to manage the huge amount of memory that is required to save the space-variant parameters necessary to perform the non-linear convolution.

## 1.3 Outline

The chapters of this thesis are organized as follow:

- **Chapter 2** gives a more deeply explanation of the architecture of a Non-Linear Convolutional Layer, explaining the mathematical details inside the layer. For the sake of simplicity, firstly it has been analyzed a one-channel case and then it has been generalized to a multi-input-output channel case.

- **Chapter 3** exposes an overview of the most important hardware platforms suitable for the NLC layer implementation, analysing the advantage and disadvantage for each of these ones. In particular, more attention has been done on the explanation of the internal architecture of contemporary FPGAs.

- **Chapter 4** explains the loop techniques used to optimize the implementation of a NLC layer in FPGAs. In particular, an overview of the loops variables which impact on the hardware requirement is listed.

- **Chapter 5** explains the implications of the loop variables on the latency, on the size of the on-chip memory and on the number of accesses to the off-chip memory.

- **Chapter 6** explains a possible scheme of HW accelerator of the NLC layer into FPGAs. In particular, most of the effort has been addressed on the design of the PE-Array, which is the computational core that is needed to perform both the first linear convolution and the second non-linear convolution of the layer. After that, a simulation of the designed convolutional core is performed to ensure a correct behaviour.

- **Chapter 7** provides a summary of the procedural steps followed to implement the NLC layer and proposes some steps for future developments.

# Chapter 2

# Non-Linear Convolutional Layer Architecture

## 2.1 Basic concepts

In the following subsections 2.1.1 and 2.1.2, it has been reported a briefly explanation of some basic concepts, in order to better understand the architecture of the NLC layer.

### 2.1.1 Classical Linear Convolutional layer

The classical linear convolution layer performs the convolution of the input data. In a convolution a dot-product is computed between the input image pixels and a matrix called kernel or filter. The values of this matrix are called weights. The filter slides along the input data by a stride factor, and the respective convolutions are then collected into an output array called output feature map. In addition, a bias term could be added to the output value.

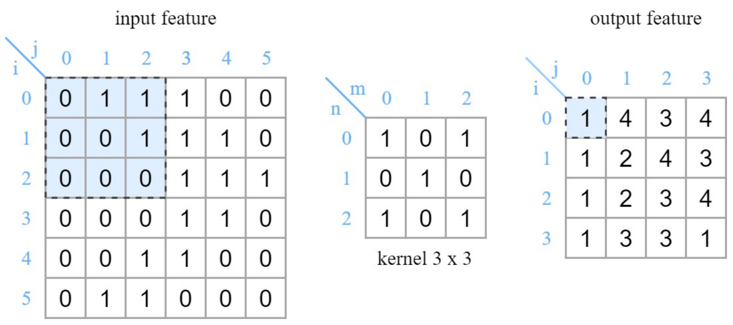


**Figure 2.1:** 2D convolution example with kernel 3x3 and stride 1

In a one-channel case the equation 2.1.1 describes the 2D convolution operation:

$$y(i,j) = \sum_{n=0}^{W} \sum_{m=0}^{W} x(i+n, j+m) \cdot v(n,m) \tag{2.1.1}$$

where:

- $x(i,j)$ is the input data
- $y(i,j)$ is the output data
- $W \times W$ is the kernel size
- $v(n,m)$ is the weight of the kernel $W \times W$

As in the equation 2.1.1 , the same principle could be extended to a multi-input-output channel case:

$$y(i,j,l) = \sum_{p=0}^{K} \sum_{n=0}^{W} \sum_{m=0}^{W} x(i+n, j+m, p) \cdot v_l(n,m,p) \tag{2.1.2}$$

where:

- K is the number of input channels
- l is the index that refers to the output channel, with L desired output channels

## Zero padding

By referring to figure 2.1, it can be noticed that the output feature has a smaller size respect to the input feature. In order to preserve the input dimensions, some strategies could be adopted to control the output image size. Among them, there is the zero-padding mechanism, in which zeros are added around the edges of the input matrix.

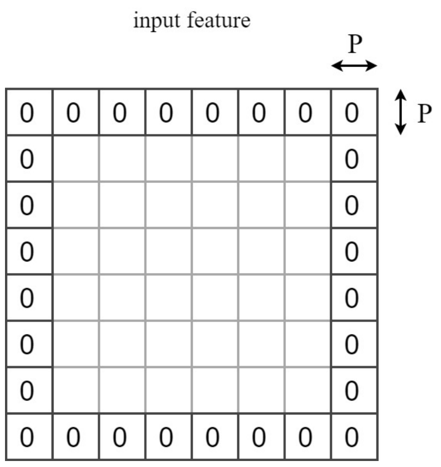


**Figure 2.2:** Zero padding example

## Size of the output volume of a convolutional layer

Assuming that $H_i$, $W_i$, and $K_i$ are respectively height, width and number of channels of the input features, the size of the output volume $H_o * W_o * K_o$ can be calculated as:

$$H_o = \frac{H_i - W + 2P}{S} + 1 \tag{2.1.3}$$

$$W_o = \frac{W_i - W + 2P}{S} + 1 \tag{2.1.4}$$

$$K_o = N_f \tag{2.1.5}$$

where:

- W is the width of a square filter
- P is the padding
- S is the stride
- $N_f$ is the number of filters

### 2.1.2 Activation functions

In order to add non-linearity to the convolutional neural networks, non-linear functions are added. In fact, without activation functions, a neural network is just a linear regression model that is not capable of learn the complex pattern of an input image. These functions are added after convolutional layers or fully connected layers. The most important activation functions are:

- Rectified Linear Unit (ReLU)

$$\begin{cases} 0 & if\ x \leq 0 \\ x & if\ x > 0 \end{cases}$$

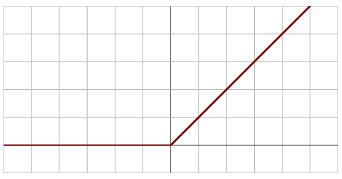
- Sigmond

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

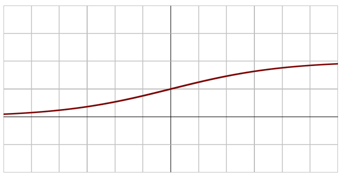
- Hyperbolic tangent

$$tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

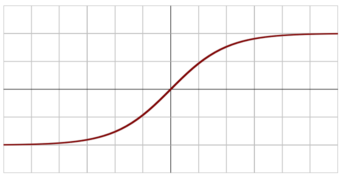
Among them, ReLU is one of the most used activation functions in CNNs because it has a low computational cost and also because it is capable to reduce the training time without significantly affecting the accuracy.



ReLU [2]



Sigmond [3]



Hyperbolic tangent [4]

**Figure 2.3:** Activation Functions

## 2.2 NLC layer

During the last years, many architectures of convolutional neural networks (CNNs) have been inspired by models of the primate visual system. This approach has led to better achieve state-of-the-art performance in the image classification field. Indeed, in neuroscience, it has been noticed that the response of the visual cells is based on non-linear operations. Usually, in CNNs, non-linearity is exploited through the implementation of activation functions, but in the literature, to better improve the model of the neuron response, different approaches have been used from the above mentioned.
Since classical convolutional layers are linear systems, their capabilities are quite limited, so, many research have tried to extend the classical convolution to a form of non-linear convolution. One of them, for example is by applying the Volterra's theory to CNNs by introducing the Volterra Kernels [5].
In [1], instead, a novel non-linear image-processing system is introduced through the implementation of a non-linear convolutional layer. Starting from a classical linear convolution, the main idea is related on deriving each weight $w_n$ of a filter through a further linear convolution, as it can be noticed from the equations 2.2.1 and 2.2.2:

$$y(i) = b' + \sum_{n=0}^{N} w_n \cdot x(i-n) \tag{2.2.1}$$

$$w_n = b''_n + \sum_{m=0}^{M} w'_{m,n} \cdot x(i-m) \tag{2.2.2}$$

Substituting equation 2.2.2 in equation 2.2.1, the overall result is expressed by the following equation:

$$y(i) = b' + \sum_{n=0}^{N} [b''_n + \sum_{m=0}^{M} w'_{m,n} \cdot x(i-m)] \cdot x(i-n) \tag{2.2.3}$$

Moreover, as can be seen, if M is forced to be equal to N, the equation 2.2.3 is equal to the series of N-size Volterra kernels limited to the second order:

$$y(i) = h_0 + \sum_{n=0}^{N} w_n \cdot x(i-n) + \sum_{n=0}^{N} \sum_{m=0}^{M} w_{m,n} \cdot x(i-m)x(i-n) + ... \tag{2.2.4}$$

So, at the end, the proposed non-linear convolutional layer differently from the standard convolutional layer used in CNNs is capable to change the parameters of the filters based on the local characteristic of the input image. This novelty acts in a better way, because it takes into account the very nature of the input data.

### 2.2.1 Single-input-output channel layer

To better understand the overall NLC layer architecture, this paragraph is focus on a straightforward example in which only one-input-output channel is considered. Considering a 2-dimensions input data $x(i, j)$ and output data $y(i, j)$ the non-linear convolution can be expressed with the following formula:

$$y(i,j) = \sum_{n=0}^{W_1} \sum_{m=0}^{W_1} x(i+n-o_1, j+m-o_1) \cdot v_{i,j}(n,m) \tag{2.2.5}$$

where:

- $o_1 = \frac{W_1-1}{2}$ is the padding added to maintain the output size constant respect to the input one

As it can be noticed, differently from the equation 2.1.1 of the classical 2D linear convolution, the weight $v_{i,j}(n, m)$ is space-variant, because his value change varying the index i and j. So, the weights are strictly related on the value of the pixels of the input data.
In order to compute the space-variant coefficients, each of the weight $v_{i,j}(n, m)$ of kernel size $W_1 \times W_1$ is computed with a linear classical convolution of the input feature. The convolution is performed with a kernel of odd size $W_2 \times W_2$ that is filled with fixed trained weights $u_{n,m}(r, s)$. Then the weights go through a non-linear point-wise activation function (i.e. ReLU):

$$\hat{v}_{i,j}(n,m) = AF(\sum_{r=0}^{W_2} \sum_{s=0}^{W_2} x(i+r-o_2, j+s-o_2) \cdot u_{n,m}(r,s)) \tag{2.2.6}$$

where:

- $o_2 = \frac{W_2-1}{2}$ is the padding added to maintain the output size constant respect to the input one

To get the final value of the space-variant weights, a normalization process is needed to guarantee that the dynamic range of the output feature is controlled, and that the system gain is suitable:

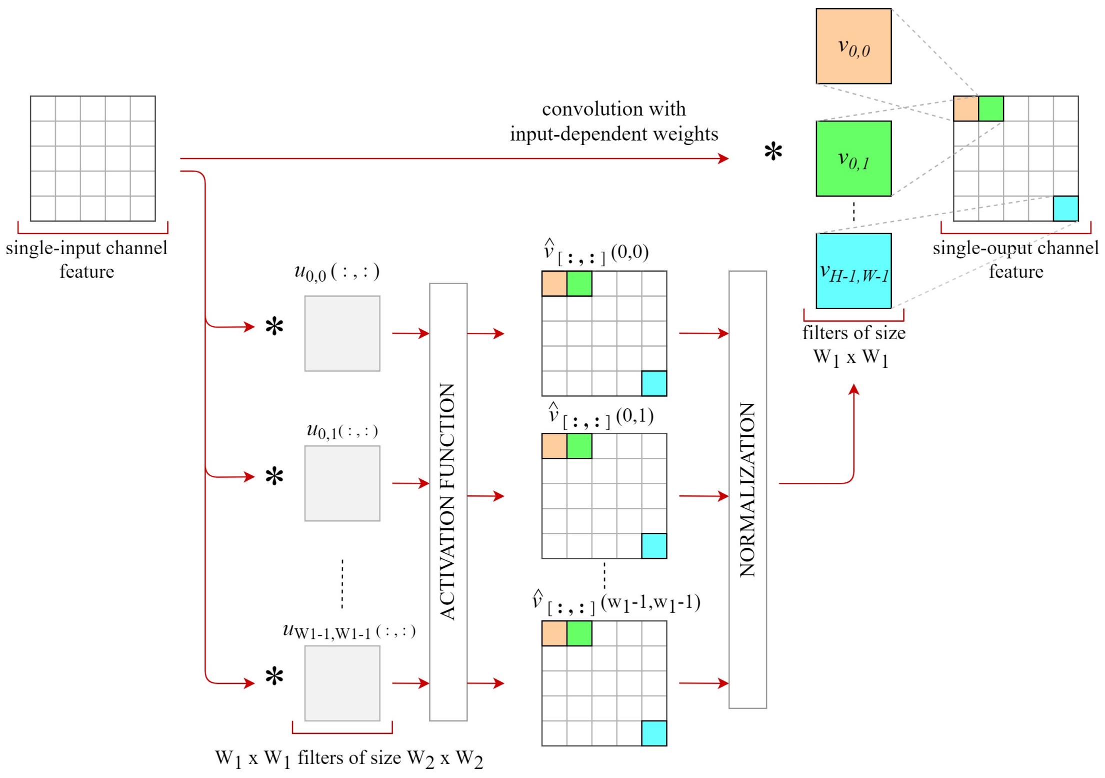
$$v_{i,j}(n,m) = \frac{\hat{v}_{i,j}(n,m)}{\sum_{n,m} \hat{v}_{i,j}(n,m) + \varepsilon} \tag{2.2.7}$$

or:

$$v_{i,j}(n,m) = \frac{\hat{v}_{i,j}(n,m)}{\sum_{n,m} |\hat{v}_{i,j}(n,m)| + \varepsilon} \tag{2.2.8}$$

where:

- $\varepsilon$ is a small value added to avoid that during the training process the system gets illegal values



convolution with input-dependent weights
single-input channel feature
single-ouput channel feature
$v_{0,0}$
$v_{0,1}$
$v_{H-1,W-1}$
filters of size $W_1$ x $W_1$
$u_{0,0}(:,:)$
$u_{0,1}(:,:)$
$u_{W1-1,W1-1}(:,:)$
ACTIVATION FUNCTION
NORMALIZATION
$\hat{v}_{[:,:]}(0,0)$
$\hat{v}_{[:,:]}(0,1)$
$\hat{v}_{[:,:]}(w_1-1,w_1-1)$
$W_1$ x $W_1$ filters of size $W_2$ x $W_2$

**Figure 2.4:** Example of single-input-output channel NLC layer

In Figure 2.4, a schematic of the single-input-output channel case of NLC layer is shown. As it can be noticed, on the bottom-left part of the picture the input data go through a convolutional stage that computes the input-dependent weights needed to perform the convolution with space-variant parameters (top-right of the picture). For the sake of simplicity, only a subset of the $W_1 \times W_1$ filters is shown. In particular, the orange kernel $v_{0,0}(:,:)$, the green kernel $v_{0,1}(:,:)$ and the cyan kernel $v_{H-1,W-1}(:,:)$ each have $W_1 \times W_1$ weights that are taken from the output feature (properly normalized) of the first convolution at the spatial coordinate $(i = 0, j = 0)$, $(i = 0, j = 1)$ and $(i = H - 1, j = W - 1)$ respectively.

### 2.2.2 Multiple-input-output channel layer

Considering an input data $x(i,j,k)$, where k refers to the selected channel of the input feature with $0 \leq k < K$, and an output data $y(i,j,l)$, where l refers to the output channel with $0 \leq l < L$, the same formula 2.2.5 could be extended considering a multi-input-output channel case:

$$y(i,j,l) = \sum_{p=0}^{K} \sum_{n=0}^{W_1} \sum_{m=0}^{W_1} x(i+n-o_1, j+m-o_1, p) \cdot v_{i,j,l}(n,m,p) \tag{2.2.9}$$

Like in a classical convolutional layer, the weights $v_{i,j,l}(n,m,p)$ depend on the channel k of the input image and on the desired output channel l. In addition, they also depend on the spatial coordinate i and j, because they are space-variant parameters.
As for the one channel case, each weight of the kernel of size $W_1 \times W_1 \times K$ is computed with a further classical convolution followed by an activation function:

$$\hat{v}_{i,j,l}(n,m,p) = AF(\sum_{q=0}^{K} \sum_{r=0}^{W_2} \sum_{s=0}^{W_2} x(i+r-o_2, j+s-o_2, q) \cdot u_{n,m,p,l}(r,s,q)) \tag{2.2.10}$$

Similarly, a normalization process is needed to keep under control the gain of the output image:

$$v_{i,j,l}(n,m,p) = \frac{\hat{v}_{i,j,l}(n,m,p)}{\sum_{n,m,p} \hat{v}_{i,j,l}(n,m,p) + \varepsilon} \tag{2.2.11}$$

or:

$$v_{i,j,l}(n,m,p) = \frac{\hat{v}_{i,j,l}(n,m,p)}{\sum_{n,m,p} |\hat{v}_{i,j,l}(n,m,p)| + \varepsilon} \tag{2.2.12}$$

Now, a further consideration about the implications related to the AF adopted is discussed below. In fact, with the implementation of an activation function like the ReLU, which returns only positive value of weights, the NLC layer provides a spatially variant low-pass version of the input data. So, a normalization process is necessary to keep unitary the sum of the weights in order to ensure that the dynamic range of the output image is comparable with the dynamic range of the input one. In some cases, it is convenient to use AF like $tanh()$ which brings out both positive and negative weights.

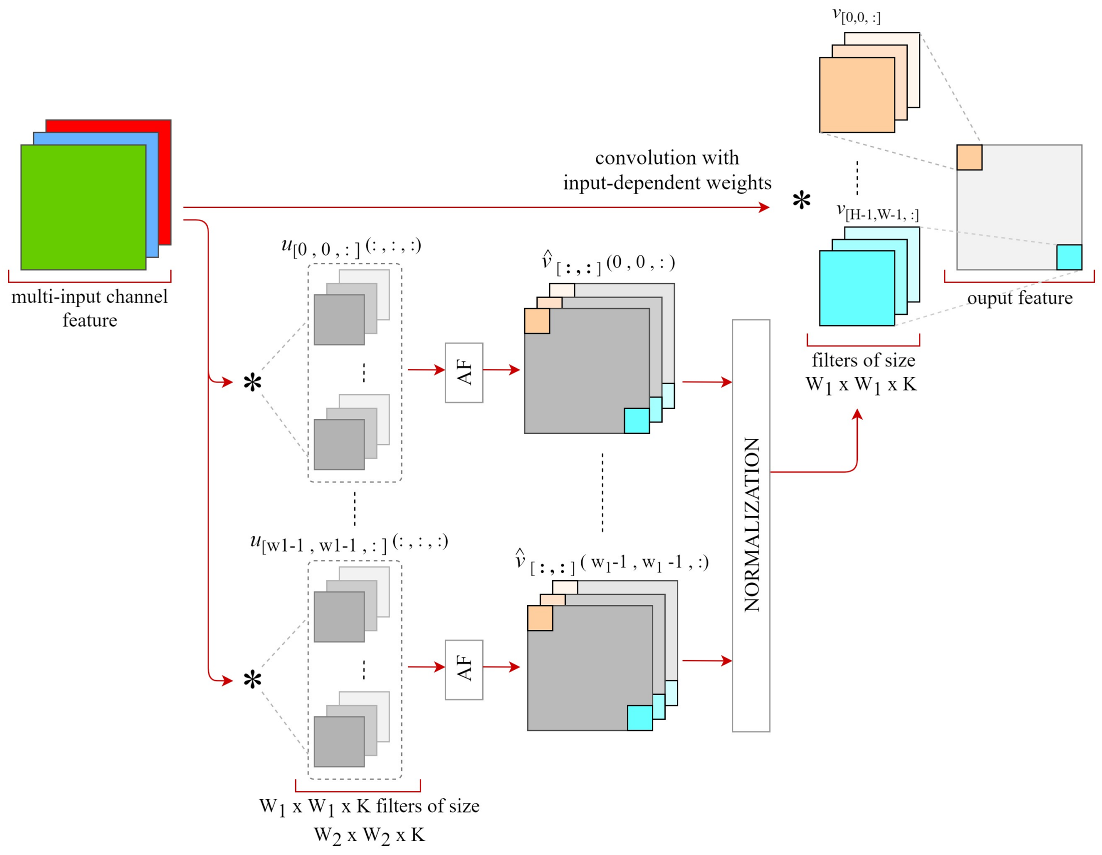

$v_{[0,0,:]}$
convolution with input-dependent weights
$v_{[H-1,W-1,:]}$
multi-input channel feature
$u_{[0,0,:]}(:,:,:)$
$\hat{v}_{[:,:]}(0,0,:)$
AF
NORMALIZATION
filters of size $W_1$ x $W_1$ x K
ouput feature
$u_{[w1-1,w1-1,:]}(:,:,:)$
$\hat{v}_{[:,:]}(w_1-1, w_1-1,:)$
AF
$W_1$ x $W_1$ x K filters of size $W_2$ x $W_2$ x K

**Figure 2.5:** Example of multi-input single-output channel NLC layer

Figure 2.5 shows an example of NLC layer that has as input an image with 3-channel depth. For the sake of simplicity, only one-output channel is computed. Viewing the schematic as for the Figure 2.4, on the bottom-left part of the picture the input-dependent weights are calculated, while on the top-right the non-linear convolution is performed.

## NLC layer vs standard-convolutional layer

Some of the differences between the NLC layer and the standard-convolutional layer are summarized in this subsection. In particular, table 2.1 makes a comparison on the number of fixed weights configured during the training phase, while figure 2.6 shows a comparison of the block scheme of the two proposed layers.

| | **Classical Convolutional Layer** | **Non-Linear Convolutional Layer** |
|---|---|---|
| Fixed Weights | $W_1^2 \times K \times L$ | $W_2^2 \times W_1^2 \times K^2 \times L$ |

**Table 2.1:** Number of fixed weights required by a NLC layer and a Convolutional layer

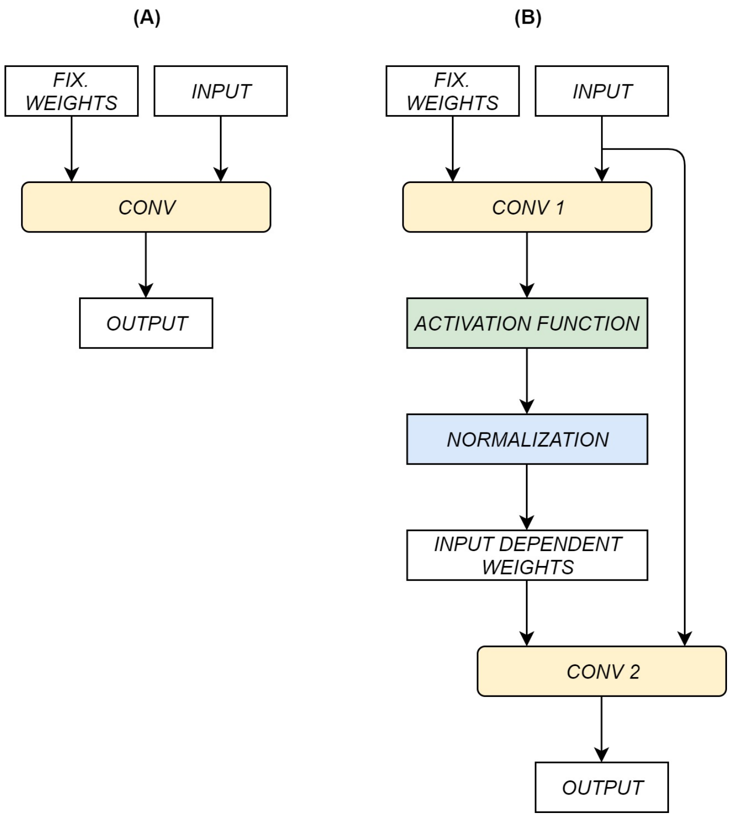


**Figure 2.6:** Comparison of the block scheme between the convolutional layer (A) and the non-linear convolutional layer (B)

## 2.3 NLCN Architectures

A convolutional neural network is based on some building-blocks such as convolutional layers, pooling layers, fully connected layers and so on. CNNs in order to perform complex real tasks, typically, involve the implementation of several of these building-blocks.
Usually, a stack of convolutional layers and pooling layer is followed by fully connected layers.

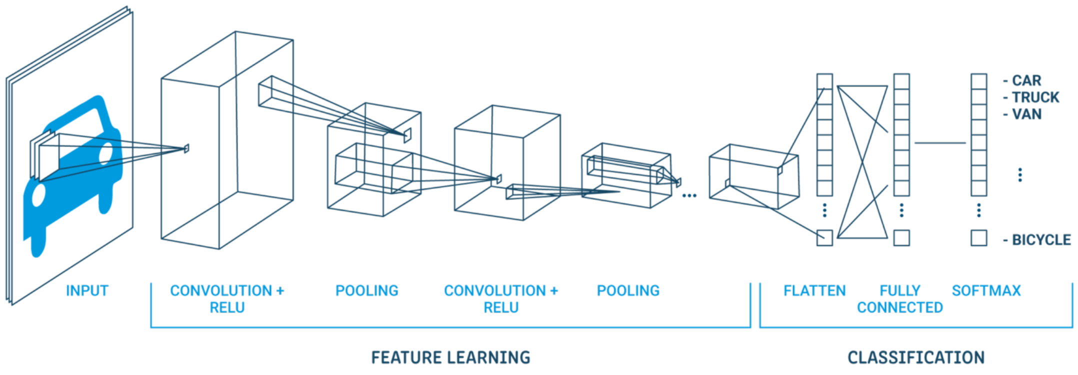


**Figure 2.7:** Example of architecture of a CNN [6]

In the same manner, non-linear convolution networks to complete complex problems need to combine different layers including in particular the NLC layer. Starting with the fact that NLC layers exploit a better ability of mimic the real behaviour of biological networks respect to classical convolutional layers, in NLCNs, the number of layers implemented can be much less respect to the ones implemented in standard CNNs. As it was reported in [1] , with only 5 layers it is possible to achieve performances comparable with CNNs that implement dozens of layers.
In the following subparagraph 2.3.1, a briefly overview on the possible strategies to combine layers in NLCNs has been explained.

### 2.3.1 Strategies to combine layers in NLCNs

#### Cascade

This is the most straightforward solution to combine layers, in which non-linear layers are serially connected on a stack. This strategy, applied in many CNNs, is affected by a problem called *vanishing gradient*, specially with deeper networks. Either way, due to the quadratic kernel and the limited number of layers implemented, this problem does not occur in NLCNs.

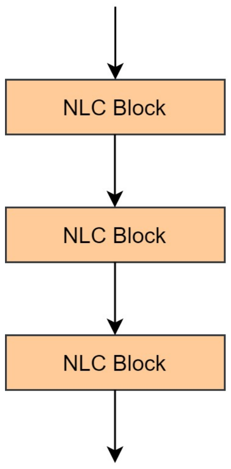


**Figure 2.8:** Cascade Architecture of a NLCN

## Cascade + bypass

To mitigate the problem of the vanishing gradient and the relative saturation problem, in literature many authors [7, 8, 9] add shortcuts connections to skip part of the network. As it can be notice in Figure 2.9, some direct paths are added to the cascade architecture. In this way, each layer performs with all the possible knowledge, because the incoming layer has as input the L output channels and the K input channels of the previous layer.

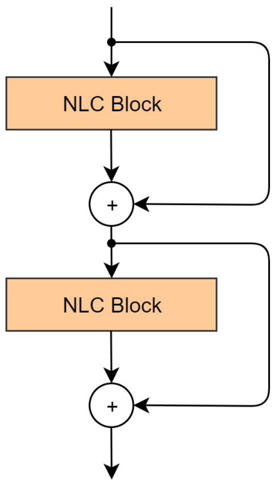


**Figure 2.9:** Cascade + Bypass Architecture of a NLCN

## Cascade + bypass + mixing layer

With the cascade plus bypass layer strategy, the layer of each stage required a greater number of parameters respect to the previous layer. This is related on the fact that at each stage the NLC layer has K + L input channels. In particular, by referring the table 2.1, the number of fixed weights required is quadratically dependent respect to the number of input channels. This could lead to the saturation of the computing resources. To prevent this problem, among the possible solutions a convolutional layer with kernel 1x1 is inserted before a NLC layer. The insertion of this layer acts as a mixer on the input features because it provides a suitable linear combination of them.

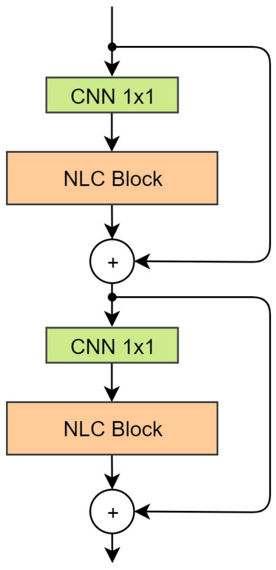


**Figure 2.10:** Cascade + Bypass + Mixing Layer Architecture of a NLCN

## Strategy to improve Receptive Field

To improve the output efficiency, one of the possible solutions is to enlarge the portion of input image to analyse. This input area analysed is called *receptive field*. Since, enlarging the kernel size the number of parameters grows exponentially as is described in table 2.1, one possible method is to use a parallel architecture approach. This architecture uses multiple NLC layers which perform in parallel. Each NLC layer has different dilation parameters. In fact, in dilated convolutions the filters are enlarged by a parameter D (Figure 2.11). Using this approach, it is possible to analyse portion of input features larger than the previous architectures.

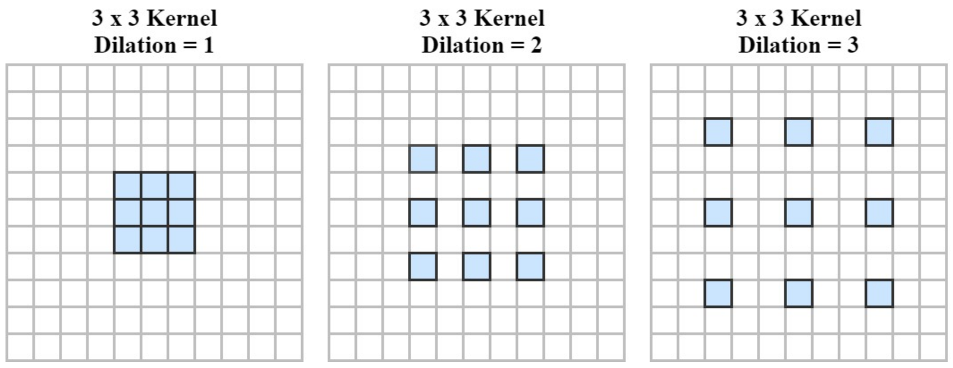


**Figure 2.11:** Example of kernels with different dilation rates

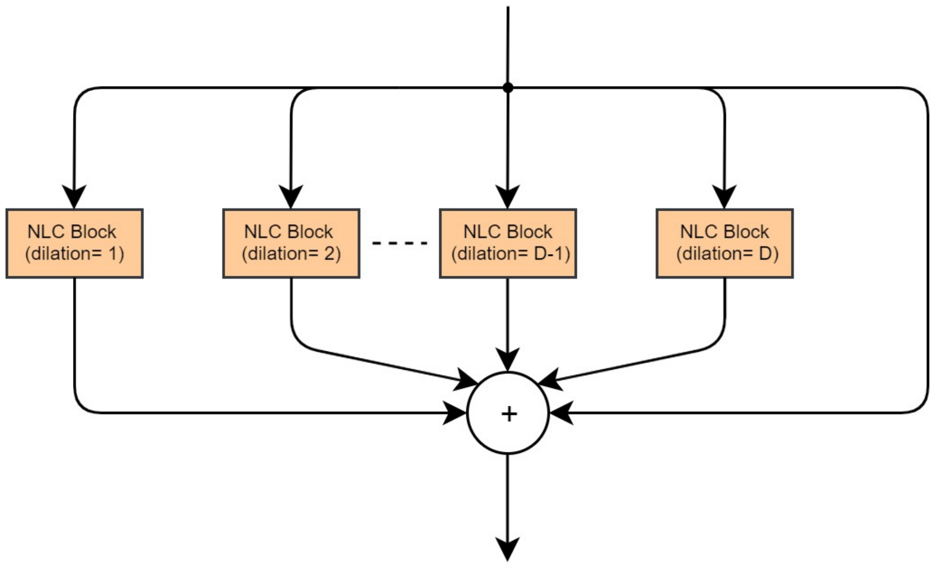


**Figure 2.12:** Parallel architecture to improve Receptive Field

# Chapter 3

# Hardware Platform for NLCN Processing

During the last years, in ML algorithms the amount of data to be processed is become greater and greater. For this reason, much research has been carried out to develop dedicated hardware accelerator for implementing the chosen NN architecture. In fact, this approach has led both to speed up the overall execution and to reduce the power consumption of the inference phase. Currently, in most of the cases, the training phase is performed by the Graphic Processing Units (GPUs), which are the most suitable hardware platform due to the flexibility of the instruction code execution and the high precision format computation. On the other side, FPGAs and ASICs have become the most important candidate to accelerate machine learning inference both in the edge and in the cloud. In fact, while these last ones allow to implement flexible architecture through the custom logic, CPUs and GPUs are general purpose processors with a fixed architecture.

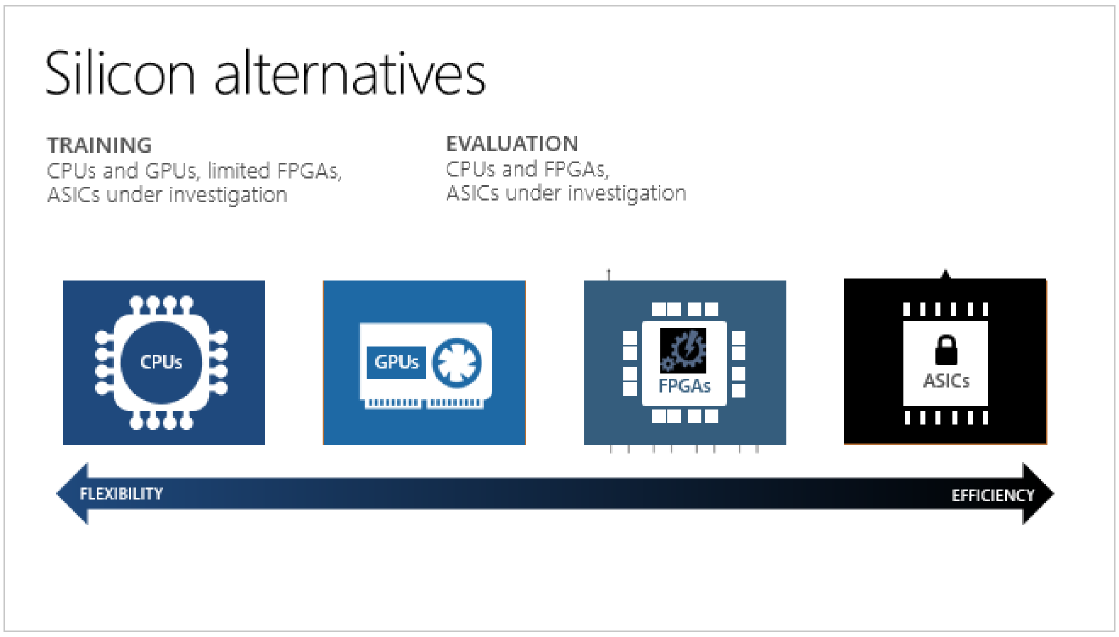

Silicon alternatives
TRAINING
CPUs and GPUs, limited FPGAs, ASICs under investigation
EVALUATION
CPUs and FPGAs, ASICs under investigation
CPUs
GPUs
FPGAs
ASICs
FLEXIBILITY
EFFICIENCY

**Figure 3.1:** Flexibility vs efficiency among different hardware platforms [10]

Clearly, each hardware platform has advantages and disadvantages respect to the other ones. In particular, looking at the flexibility and the efficiency, if the former grows the latter decrease and vice-versa (Figure 3.1):

- **CPU** (Central Processing Unit) is an integrated circuit generally composed of one or more cores that process a flow of instructions. If on one side this architecture allows the highest flexibility on the executed code, on the other side the high latency and high-power consumption make of this HW platform the least suitable regarding the ML inference.
- **GPU** (Graphic Processing Unit) is an integrated circuit composed of many cores, but differently from the CPUs, these are optimized to compute high precision floating point operations. Through the high parallelism and the high memory bandwidth available, it is possible to compute the inference with low latency. Unfortunately, the main disadvantage of the GPUs is related on the huge power consumption which makes inefficient their use in the edge computing.
- **FPGAs** (Field Programmable Gate Arrays) are a programmable integrated circuit that is capable to implement any designed digital architecture. So, if properly configured, this IC is able to perform ML inference. Moreover, it is possible to investigate different levels of parallelism trying to better optimize the data flow. However, the fixed available programmable logic, the on-chip

memory and the low bandwidth limit the capability of FPGAs to implement complex architectures. On the other side, the capability to program the internal logic brings to explore different optimization techniques. Furthermore, FPGAs have typically a lower power consumption respect to GPUs, which make them one the most important candidate for the ML inference both in the edge and in the cloud.

- **ASICs** (Application Specific Integrated Circuits) are an integrated circuited that is designed to perform a specific task. This hardware platform offers the maximum flexibility on the design phase allowing to reach an optimal speed and a reduced energy consumption. So, if on one side ASICs have low power consumption and the most optimized data flow for a specific application, on the other side their architecture is not re.programmable. So, this HW platform has the lowest flexibility in term of task reconfigurability respect to the other ones. Moreover, their high design cost makes them less competitive respect to FPGAs.

Some of the most useful criteria to choose the proper hardware platform are summarized in the table below:

| | **CPUs** | **GPUs** | **FPGAs** | **ASICs** |
|---|---|---|---|---|
| Power Consumption | high | very-high | low | very-low |
| Flexibility | very-high | high | medium | very-low |
| Latency | very-high | medium | low | very-low |
| Throughput | low | high | high | very-high |
| Time-to-market | low | medium | medium | very-high |
| Efficiency | low | medium | high | very-high |

**Table 3.1:** Comparison between CPUs, GPUs, FPGAs and ASICs

In this thesis, the FPGA has been chosen as the most suitable HW platform to implement the hardware accelerator of a NLC layer.

## 3.1 Architecture of a FPGA

Field Programmable Gate Arrays (FPGAs) are a reprogrammable integrated circuit that is typically organized on 2D array, in which inside there are some *Configurable Logic Blocks* (CLBs) that are the ones that contain the functional logic. After that there are the CLB interconnections that can be either local or global, depending on the fact that there is the necessity to connect portions of the FPGA that are located close or far the one to each other respectively. In the end, there are the *I/O Blocks* (IOBs) that are used for connecting the logic array with the outside world.

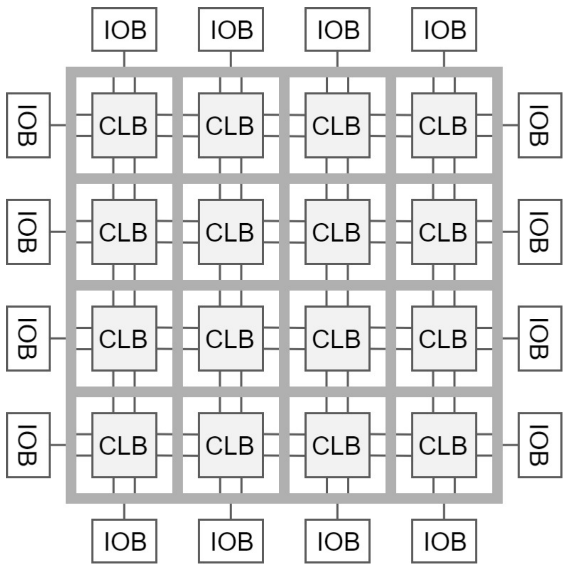


**Figure 3.2:** Architecture of a FPGA

CLBs are based on look-up tables, flip-flops, and multiplexers which allow to implement any kind of combinational or sequential logic. In particular, look-up tables are the basic kernel for the combinational logic inside FPGAs. As consequence, the number of available CLBs limits the design implementation, because the higher is the complexity of the design and the higher is the amount of hardware resources that is required. Moreover, the architecture of contemporary FPGAs includes additional logic that allows to better optimize and increase the computational capability. In Figure 3.3 it is shown an example of contemporary architecture of a Xilinx FPGA. Respect to the basic FPGA architecture, in fact, other logic blocks like BRAMs, off-chip memory controllers, DSP blocks, high-speed transceivers and phase-locked loops (PLLs) are added to the IC.

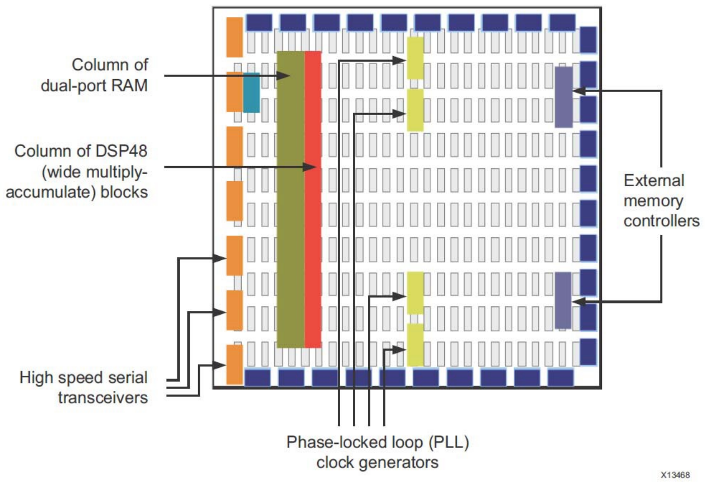

Column of dual-port RAM
Column of DSP48 (wide multiply-accumulate) blocks
High speed serial transceivers
External memory controllers
Phase-locked loop (PLL) clock generators
X13468

**Figure 3.3:** Architecture of a contemporary FPGA [11]

Block RAM (BRAM) is an on-chip storage element used as Random-Access Memory (RAM). In fact, in many applications such as signal processing and encryption it is required to store many data into local buffers which are then used to perform the dedicated algorithms. In particular, the BRAM can be used as dual-port RAM, which allows to read different data at the same clock cycle. In fact, this is a very useful feature used to implement data accumulators.

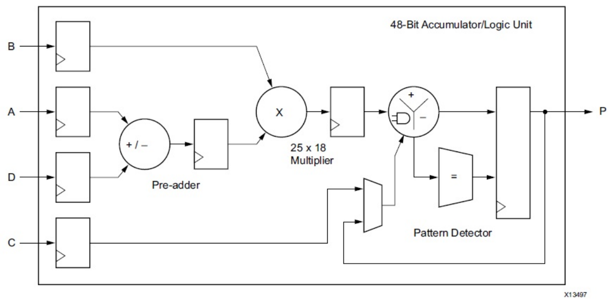


**Figure 3.4:** Architecture of a Xilinx DSP slice [11]

Another important additional in contemporary FPGAs is the availability of a cert limited number of DSPs slices, which are a dedicated and optimized logic that perform different type of functions. In particular DSPs perform the MAC operation in an efficient way. Since ML algorithms are based on the massive execution of MAC operations, the use of DSPs increases the overall performances because they have low latency and low power consumption.

# Chapter 4

# Optimization techniques for convolution loops in NLC layers

## 4.1 A generic scheme of hardware accelerator of a NLC layer

Compared with a classical CNN, the smaller number of layers of a NLCN provides to decrease the number of features, leading to reduce the total memory required. So, if on one side these advantages enhance the implementation of NLCN in embedded accelerators, on the other side NLC layers require a larger memory addressed to store the trained parameters and the intermediate results. In fact, even if the number of layers of a NLCN is fewer respect to a standard CNN, the overall complexity of a single NLC layer is higher if compared to a standard convolutional layer.

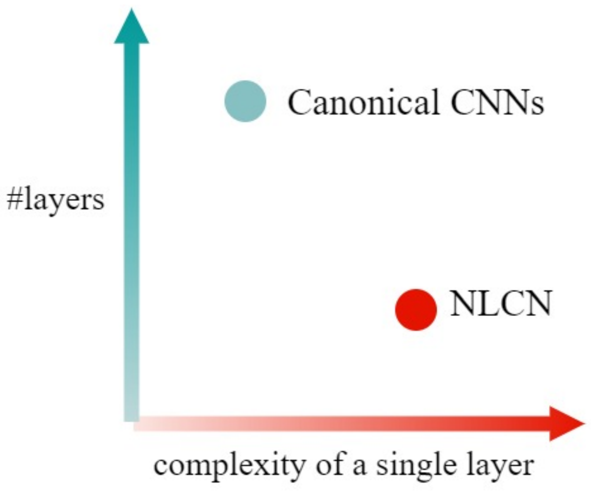


**Figure 4.1:** Comparison of the computational complexity between NLCN and standard CNN

In particular, a comparison of the number of trained parameters to store for each layer is shown below:

$$\#Parameters_{NLCN} = W1^2 \times W2^2 \times K^2 \times L \tag{4.1.1}$$

$$\#Parameters_{CNN} = ((W1^2 \times K) + 1) \times L \tag{4.1.2}$$

where:

- $W_1^2$ and $W_2^2$ are the kernel size
- $K$ is the number of input feature maps
- $L$ is the number of output feature maps
- '+1' refers to the bias term

In fact, apart from the term of the second kernel of size $W_2 \times W_2$, it can be noticed that $\#Parameters_{NLCN}$ respect to $\#Parameters_{CNN}$ has a quadratically dependence respect to the number of input feature K. This implies that the size of on-chip memory inside the FPGAs is not big enough to store both data and weights. In order to better understand where these are actually allocated, in Figure 4.2 it is shown a typical architecture of hardware accelerator for convolutional layers.

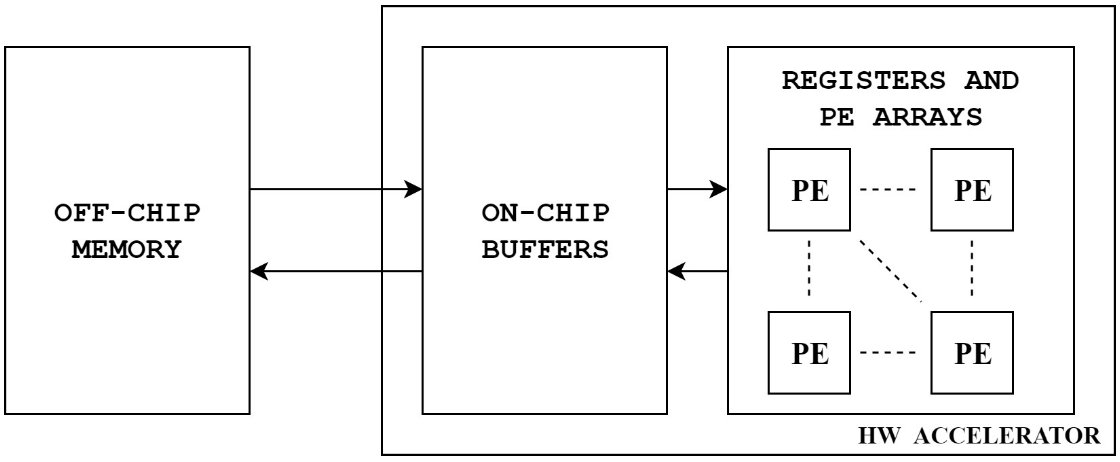


**Figure 4.2:** Example of HW accelerator for convolutional layers

As it can be noticed, the memory hierarchy inside HW accelerators can be subdivided in three main levels:

- Registers
- On-chip buffers
- Off-chip memory

In convolutional layers and NLC layers, most of the computationally effort is related on the product between data and weights and on the accumulation of the relative result on storage elements. So, the main factors that impact directly on the energy required per each single MAC operation are related on:

1. the choice of the storage element used to fetch the data
2. the way how efficiently data are reused, and partial sums (psums) are accumulated

So, the purpose of an energy-efficient hardware accelerator should be to compute the internal operations choosing the data movement with the lowest energy cost.

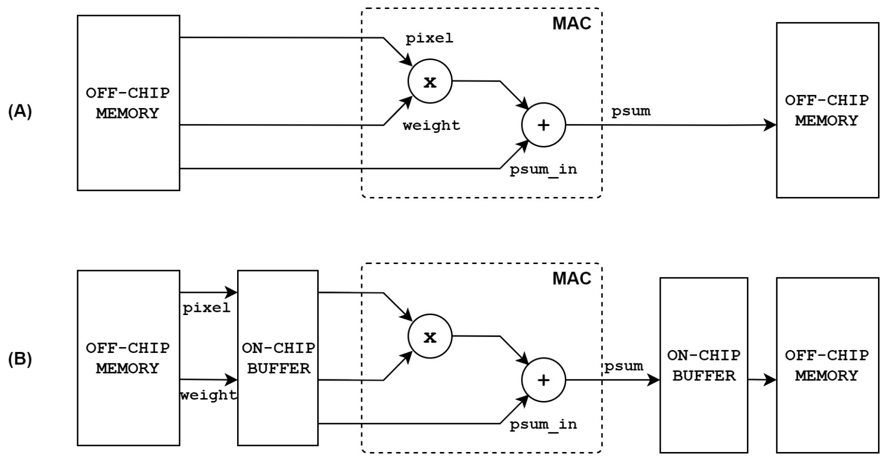


**Figure 4.3:** comparison between read and write of data for MAC operation with no memory hierarchy (A) and memory hierarchy (B)

Usually, the off-chip memory contains all the data and weights necessary to perform the inference task. Since, the size of the on-chip memory is quite limited, only a certain amount of data and weights is fetched from off-chip to on-chip buffers to compute the convolution operation. These on-chip buffers are then used to feed the registers and the relative processing elements with the proper data and weights. So, the main idea is to use on-chip buffers to increase the data reuse, storing the partial results of the computational logic (Figure 4.3). Once that the processing elements have completed the assigned tasks, the output results are then stored into the off-chip memory. As mentioned before, in fact, the introduction of a memory hierarchy lowers the energy cost required for data movement.

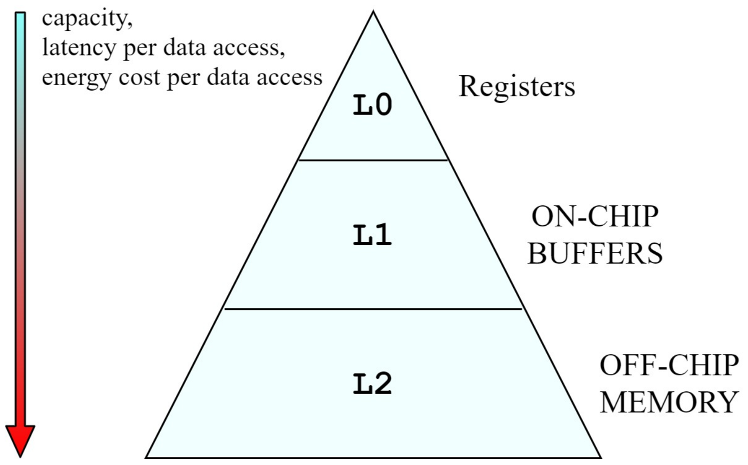
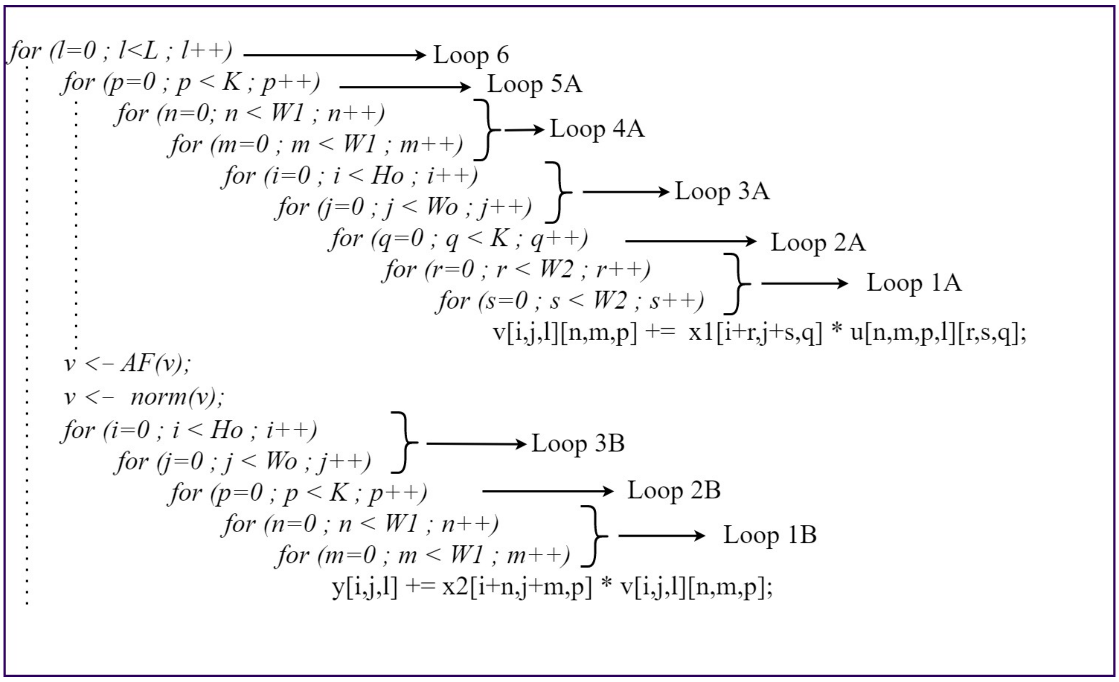


**Figure 4.4:** Generic memory hierarchy in HW accelerators

Generally, off-chip memory is a large DRAM, while on-chip buffer is a small SRAM. In particular, the first has long latency and high-power consumption per access and the second has small latency and low-power consumption per access. In [12], it has been compared different models of DRAM and SRAM and it has been shown that the first has an energy consumption per access that is two or three order of magnitude higher respect to the second. So, the use of the on-chip buffers with a proper size combined with some loop optimization strategies lead to increase the overall performances of the inference task, lowering the latency and the power consumption.

## 4.2 Convolution loops in NLC layers

As for standard CNNs, the most recurrent operation that is performed in NLCNs is the convolution. This operation, in turn, consist of several multiply-and-accumulate (MAC) operations between the input feature data and the weights. Starting from the equation 2.2.9, which describe the mathematical behaviour of the NLC layer, in Figure 4.5 it has been written a pseudocode that indicates all the nested loops needed to compute the L output feature maps.



```
for (l=0 ; l<L ; l++)                                  ⟶ Loop 6
  for (p=0 ; p < K ; p++)                              ⟶ Loop 5A
    for (n=0; n < W1 ; n++)                            } ⟶ Loop 4A
      for (m=0 ; m < W1 ; m++)                         }
        for (i=0 ; i < Ho ; i++)                       } ⟶ Loop 3A
          for (j=0 ; j < Wo ; j++)                     }
            for (q=0 ; q < K ; q++)                    ⟶ Loop 2A
              for (r=0 ; r < W2 ; r++)                 } ⟶ Loop 1A
                for (s=0 ; s < W2 ; s++)               }
                  v[i,j,l][n,m,p] += x1[i+r,j+s,q] * u[n,m,p,l][r,s,q];
  v <- AF(v);
  v <- norm(v);
  for (i=0 ; i < Ho ; i++)                             } ⟶ Loop 3B
    for (j=0 ; j < Wo ; j++)                           }
      for (p=0 ; p < K ; p++)                          ⟶ Loop 2B
        for (n=0 ; n < W1 ; n++)                       } ⟶ Loop 1B
          for (m=0 ; m < W1 ; m++)                     }
            y[i,j,l] += x2[i+n,j+m,p] * v[i,j,l][n,m,p];
```

**Figure 4.5:** Pseudocode of the NLC layer

As it can be noticed, in Figure 4.5 the convolutional loops can be grouped in different macro loops. In particular, the loops with the suffix 'A' refer to the first convolution that compute the space-variant weights, while the loops that end with 'B' refer to the second convolution that use space-variant parameters instead of fixed weighs. Since zero-padding technique is used to keep constant the output size, both the first and the second convolution take the input feature maps filled with zeros around the edge, with a padding factor that depends on the kernel size $W_1$ and $W_2$ respectively. As previously said, the NLC layers spent most of the time to perform the convolution operation, so, in this thesis main of the effort has been done to optimize the first and the second convolutional loop. To better understand the overall nested loops, in the following figures, from a graphic point of view it has been explained the execution flow through a coloured arrow of the same colour of the relative loop.

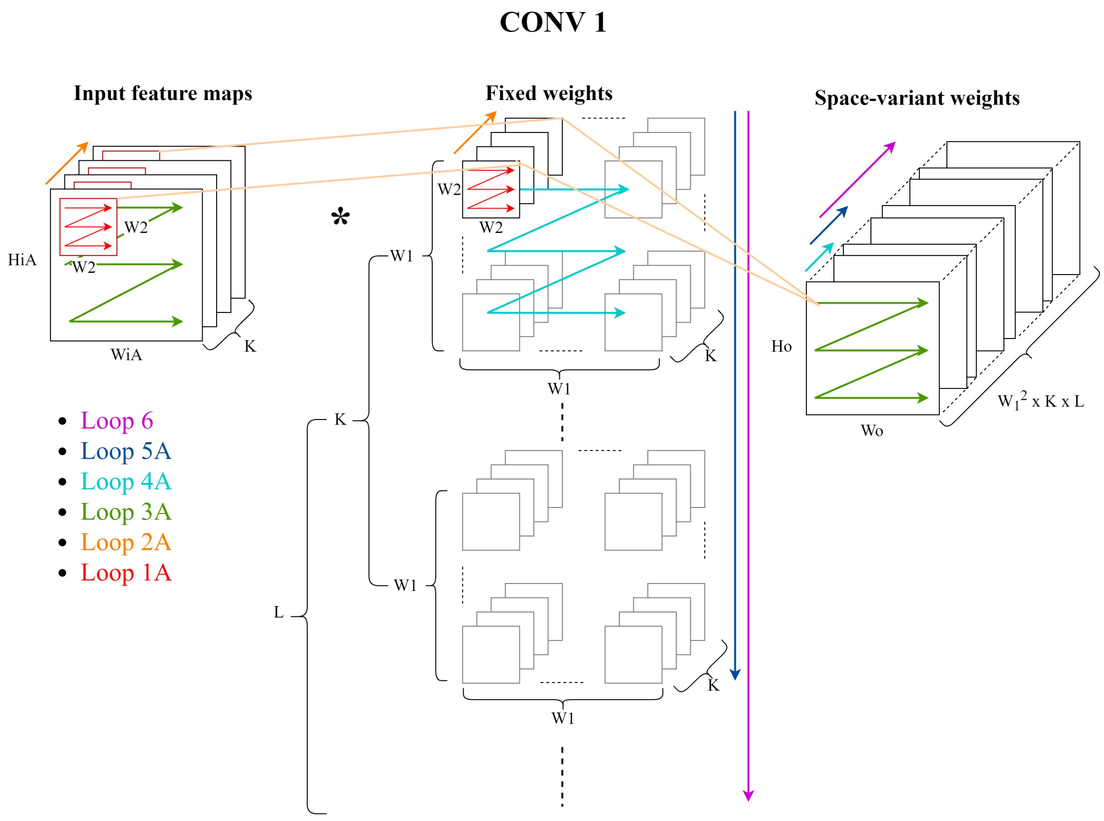


**Figure 4.6:** Nested loops of the first convolution

In Figure 4.6, it has been shown the first convolution which performs the space-variant weights. Starting from the left to the right they can be recognized the input feature maps, the fixed trained weights, and the space-variant parameters respectively. Looking at the related nested loops:

- Loop 1A performs the MAC operation between the input feature map and the kernel of size $W_2 \times W_2$
- Loop 2A moves across the channels of the filter and of the input feature maps
- Loop 3A slides along the i and j directions inside one channel of the input feature map of size HiA and WiA
- Loop 4A moves across the filters to compute the space-variant weights of a kernel of size $W_1 \times W_1$, used then for a specific input channel
- Loop 5A slides across the fixed filters to compute the matrix of space-variant weights for different input channels
- Loop 6 moves across the fixed filters to compute the input dependent parameters used then to elaborate the different output feature maps

As for the first convolution, in figure 4.7 it has been shown graphically the nested loops of the second convolution, which performs the output feature maps through a non-linear convolution operation.

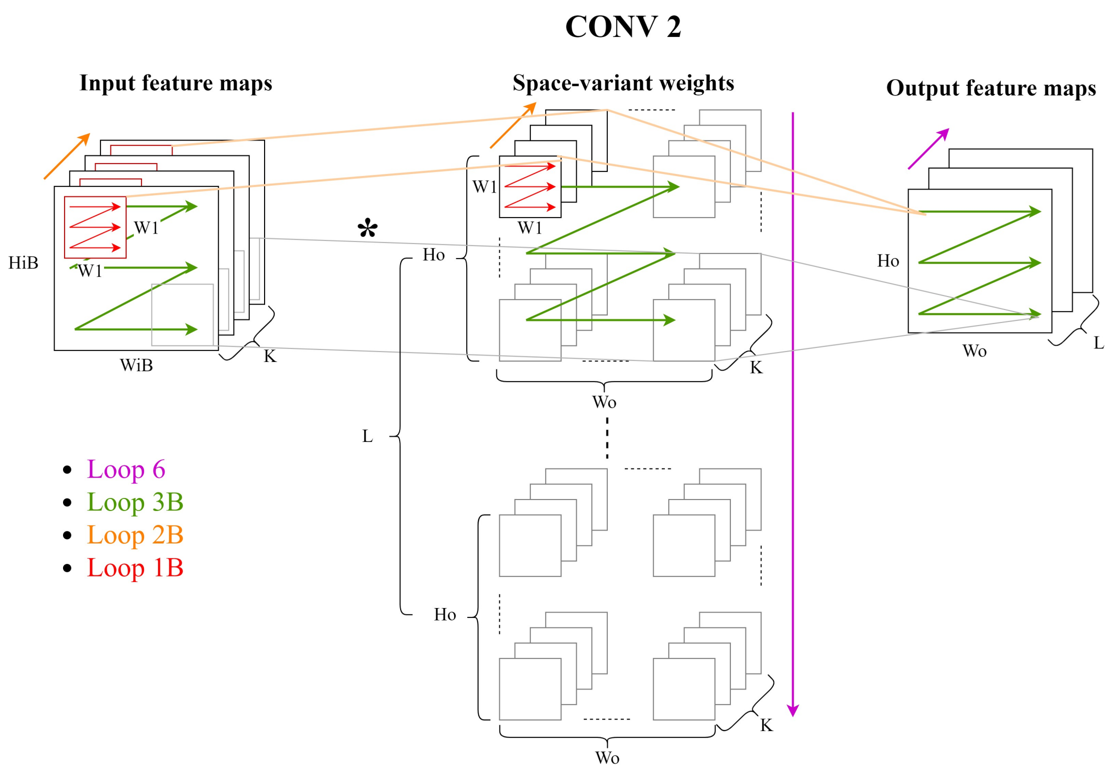


**Figure 4.7:** Nested loops of the second convolution

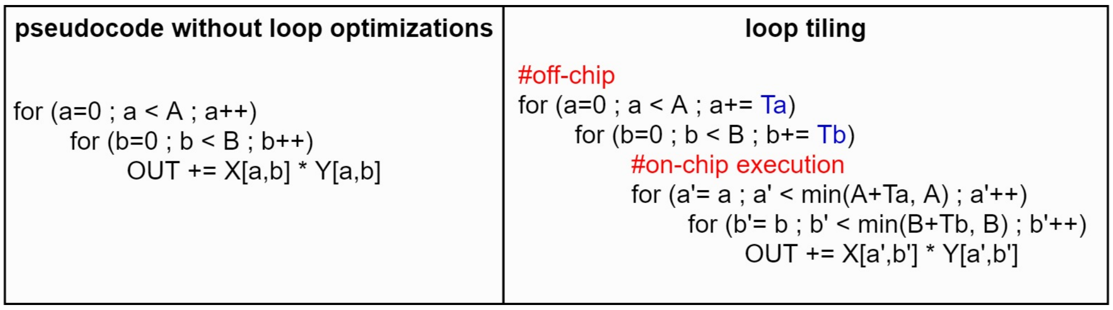
Looking at the related nested loops:

- Loop 1B performs the MAC operation between the input feature map and the kernel of size $W_1 \times W_1$
- Loop 2B moves across the channels of the filter and of the input data
- Loop 3B slides along the i and j directions inside one channel of the input feature maps of size HiB and WiB. Moreover, since this is a non-linear convolution, it also moves along the space-variant weights.
- Loop 6 moves across the L output feature maps

## 4.3 Loop optimization techniques

As in [13, 14], focusing on the NLC layer, three loop optimization techniques are investigated to optimize the dataflow:

- **loop tiling**: since on-chip memory on FPGAs is limited, this technique allows to load on-chip only a portion of the overall data and weights, and for this reason the data are subdivided in '*tiles*'. However, the tiling section must be chosen in a way that doesn't lower the performances of the PEs execution.



**pseudocode without loop optimizations**

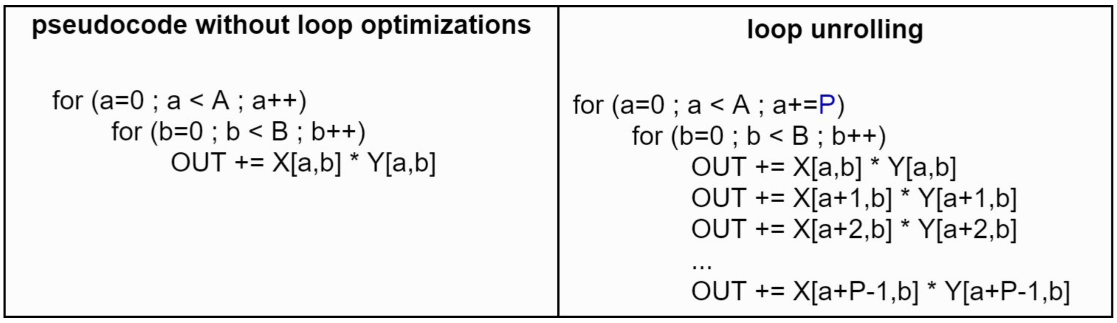
```
for (a=0 ; a < A ; a++)
    for (b=0 ; b < B ; b++)
        OUT += X[a,b] * Y[a,b]
```

**loop tiling**

```
#off-chip
for (a=0 ; a < A ; a+= Ta)
    for (b=0 ; b < B ; b+= Tb)
        #on-chip execution
        for (a'= a ; a' < min(A+Ta, A) ; a'++)
            for (b'= b ; b' < min(B+Tb, B) ; b'++)
                OUT += X[a',b'] * Y[a',b']
```

**Figure 4.8:** Loop tiling example

- **loop unrolling**: in order to speed up the execution of the convolutional loops, this technique exploits the parallel execution inside loops, reducing the overall iterations. So, an investigation of the hardware resources available on board is needed to guarantee the maximum parallelization achievable.



**pseudocode without loop optimizations**

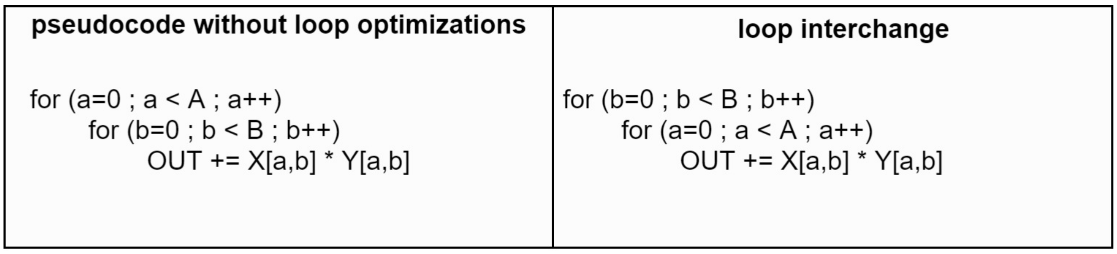
```
for (a=0 ; a < A ; a++)
    for (b=0 ; b < B ; b++)
        OUT += X[a,b] * Y[a,b]
```

**loop unrolling**

```
for (a=0 ; a < A ; a+=P)
    for (b=0 ; b < B ; b++)
        OUT += X[a,b] * Y[a,b]
        OUT += X[a+1,b] * Y[a+1,b]
        OUT += X[a+2,b] * Y[a+2,b]
        ...
        OUT += X[a+P-1,b] * Y[a+P-1,b]
```

**Figure 4.9:** Loop unrolling example

- **loop interchange**: this loop optimization technique consists on interchanging the order of execution of the nested loops. This technique combined with the previous ones allows to exploit the maximum data reused, leading on a reduction of data movements between off-chip and on-chip memory.



**pseudocode without loop optimizations**

```
for (a=0 ; a < A ; a++)
    for (b=0 ; b < B ; b++)
        OUT += X[a,b] * Y[a,b]
```

**loop interchange**

```
for (b=0 ; b < B ; b++)
    for (a=0 ; a < A ; a++)
        OUT += X[a,b] * Y[a,b]
```

**Figure 4.10:** Loop interchange example

However, it is important to underline that these techniques do not change the behaviour of the algorithm. Referring on the pseudocode reported in Figure 4.5, in order to apply the different loop optimization techniques, it is convenient to assign some design parameters for each of the nested loops. By changing these parameters, the implemented layer will change its hardware requirement and the relative hardware footprint. Moreover, these parameters will directly affect the time of execution of the inference task.

Since the number of nested loops is high, it has been decided to list the parameters for the loops of the first convolution and the loops of the second convolution into two different tables:

| | ***Filter (Height, Width)*** | ***# Input Feature Maps*** | ***Input Feature Map Size (Height , Width)*** | ***Output Feature Map Size*** | ***# Output Feature Maps*** | | |
|---|---|---|---|---|---|---|---|
| *Conv. 1 Loops* | *Loop 1A* | *Loop 2A* | *Loop 3A* | *Loop 3A* | *Loop 4A* | *Loop 5A* | *Loop 6* |
| *Loop Parameters* | *W2 , W2* | *K* | *HiA , WiA* | *Ho , Wo* | *W1 , W1* | *K* | *L* |
| *Loop Tiling (T)* | *Tr, Ts* | *Tq* | *THiA , TWiA* | *THo , TWo* | *TnA , TmA* | *TpA* | *TL* |
| *Loop Unrolling (P)* | *Pr , Ps* | *Pq* | *PHiA , PWiA* | *PHo , PWo* | *PnA , PmA* | *PpA* | *PL* |

**Table 4.1:** Conv. 1 nested loop parameters

| | ***Filter (Height, Width)*** | ***# Input Feature Maps*** | ***Input Feature Map Size (Height , Width)*** | ***Output Feature Map Size*** | ***# Output Feature Maps*** |
|---|---|---|---|---|---|
| *Conv. 2 Loops* | *Loop 1B* | *Loop 2B* | *Loop 3B* | *Loop 3B* | *Loop 6* |
| *Loop Parameters* | *W1 , W1* | *K* | *HiB , WiB* | *Ho , Wo* | *L* |
| *Loop Tiling (T)* | *TnB, TmB* | *TpB* | *THiB , TWiB* | *THo , TWo* | *TL* |
| *Loop Unrolling (P)* | *PnB , PmB* | *PpB* | *PHiB , PWiB* | *PHo , PWo* | *PL* |

**Table 4.2:** Conv. 2 nested loop parameters

Looking at the tables 4.1 and 4.2, for each loop it has been assigned a tiling and unrolling variable that is characterized by the T and P prefixes respectively. So, for instance, looking at the table 4.2 the unrolling variables (PnB, PmB), PpB, (PHiB, PWiB), (PHo, PWo) and PL will determine the number of parallel operations that are performed. While the Tiling variable (TnB, TmB), TpB, (THiB, TWiB), (THo, TWo) and TL will determine the on-chip memory required to store the input data, the partial results, and the output data. The same principle can be applied to the variables listed in table 4.1. However, for each nested loop, both tiling and unrolling variables follow the relation reported below:

$$1 \leqslant P_{parameter} \leqslant T_{parameter} \leqslant Parameter \tag{4.3.1}$$

For example, in Table 4.1 looking at the height and the width of the filter for the Loop 1A:

$$1 \leqslant Pr \leqslant Tr \leqslant W_2 \tag{4.3.2}$$

$$1 \leqslant Ps \leqslant Ts \leqslant W_2 \tag{4.3.3}$$

As described in the equations 2.2.9 and 2.2.10, the NLC layer uses the zero-padding technique to keep constant the output size respect to the input one. So, the parameters (HiA, WiA) and (HiB, WiB), which are the sizes of the input feature maps for conv. 1 and conv. 2 respectively, already includes the padding coefficient. Further, between these proposed variables, some constrains must be applied. In fact, both on Loop 3A and loop 3B the input and the output feature maps are related as follow:

| | **Loop Size** | **Loop Tiling (T)** | **Loop Unrolling (P)** |
|---|---|---|---|
| ***Loop 3A*** | HiA = Ho + W2 - 1<br>WiA = Wo + W2 − 1 | THiA = THo + W2 -1<br>TWiA = TWo + W2 -1 | PHiA = PHo<br>PWiA = PWo |
| ***Loop 3B*** | HiB = Ho + W1 - 1<br>WiB = Wo + W1 − 1 | THiB = THo + W1 -1<br>TWiB = TWo + W1 -1 | PHiB = PHo<br>PWiB = PWo |

**Table 4.3:** Variable constraints on Loop 3A and Loop 3B

## 4.3.1 Loop Unrolling

To speed-up the time of execution of the NLC layer, a parallelization of the operations performed is needed. Given a loop, the unroll factor indicates the number of parallel computations performed at each time. As consequence, by varying the unroll factor, the hardware resources required will change accordingly with this one.

- **Unroll of the Loop 1A**:

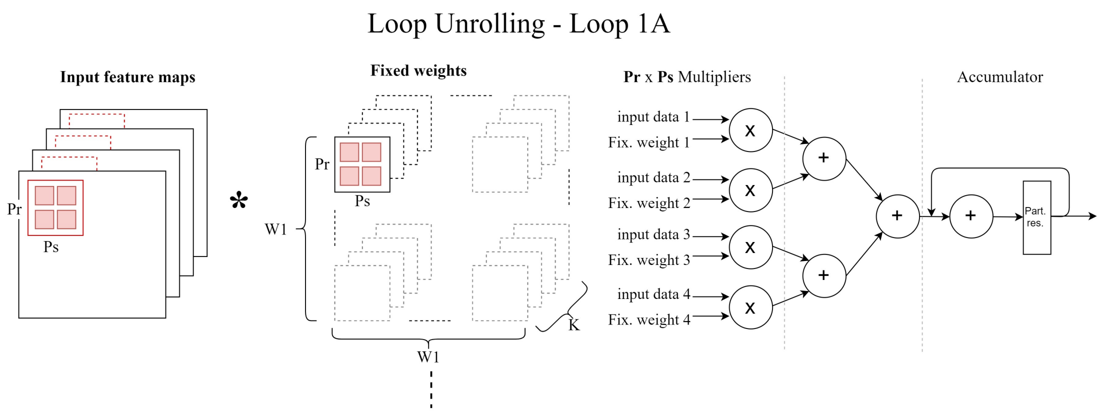


**Figure 4.11:** Unroll of the Loop 1A

The unroll of the Loop 1A allows to perform in parallel the product between the weights and the input data within the filter of size $W_2 \times W_2$. Since the unroll variable are Pr and Ps, the number of required multipliers is $Pr \times Ps$. Moreover, an adder tree is needed to add all the partial products. Finally, an accumulator is needed to accumulate the partial result at each cycle.

- **Unroll of the Loop 2A**:

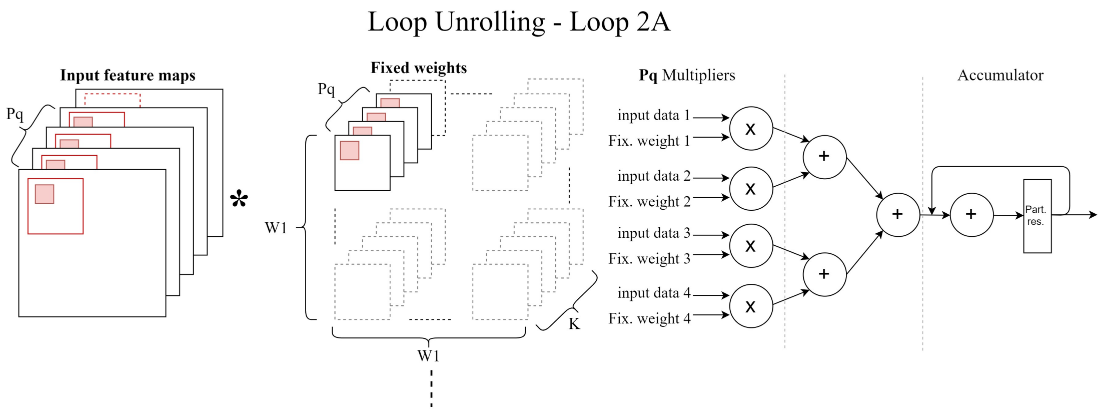


**Figure 4.12:** Unroll of the Loop 2A

The unroll of the loop 2A allows to perform in parallel the product between the input data and the weights of different input channels. Given the unroll variables Pq, the number of multipliers required is Pq. As for the previous case, an adder tree and an accumulator are required.

- **Unroll of the Loop 3A**:

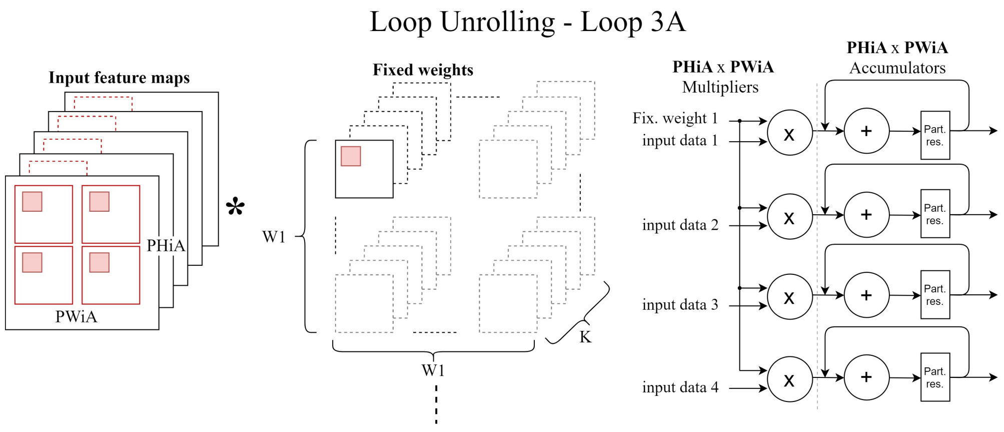


**Figure 4.13:** Unroll of the Loop 3A

The unroll of the loop 3A allows to perform in parallel multiple s-v weights of a specific output channel. In order to calculate the $PWiA \times PHiA$ output data, one fixed weight is multiplied for different input data of the same input channel. In fact, with this architecture one single weight is reused for $PWiA \times PHiA$ times. Moreover, in order to perform the final output values, $PWiA \times PHiA$ accumulators are needed to store the partial sums on-chip.

- **Unroll of the Loop 4A**:

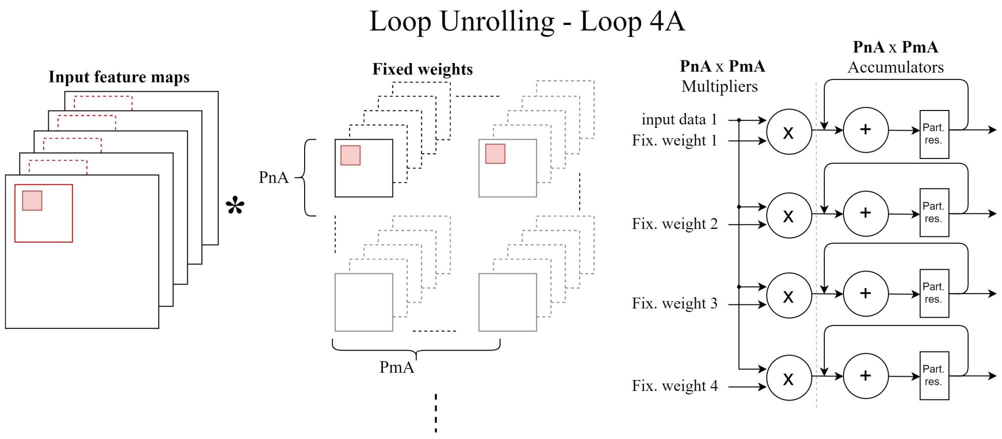


**Figure 4.14:** Unroll of the Loop 4A

The unroll of the loop 4A allows to perform in parallel data along different output channels. With this architecture are computed simultaneously the space-variant weights that are then used to be multiplied with the input data of a defined input channel. The structure of the architecture is similar with the previous one, the only difference is that in this case the input data is reused $PnA \times PmA$ times. For the hardware requirement, the number of multipliers and accumulators required is equal to $PnA \times PmA$.

- **Unroll of the Loop 5A**: The unroll of the loop 5A allows to compute in parallel the space-variant weights that are then used to be multiplied with the data of different input channels. As before, the scheme of the architecture is the same. In this case the required multipliers and accumulators are given by the unroll variable PpA.

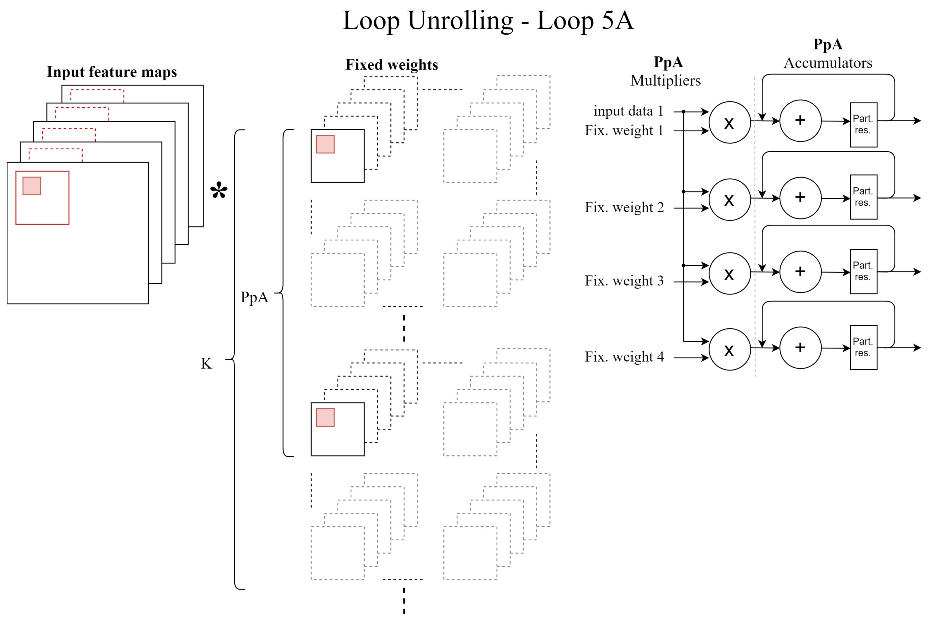


**Figure 4.15:** Unroll of the Loop 5A

- **Unroll of the Loop 6 – Conv. 1**:

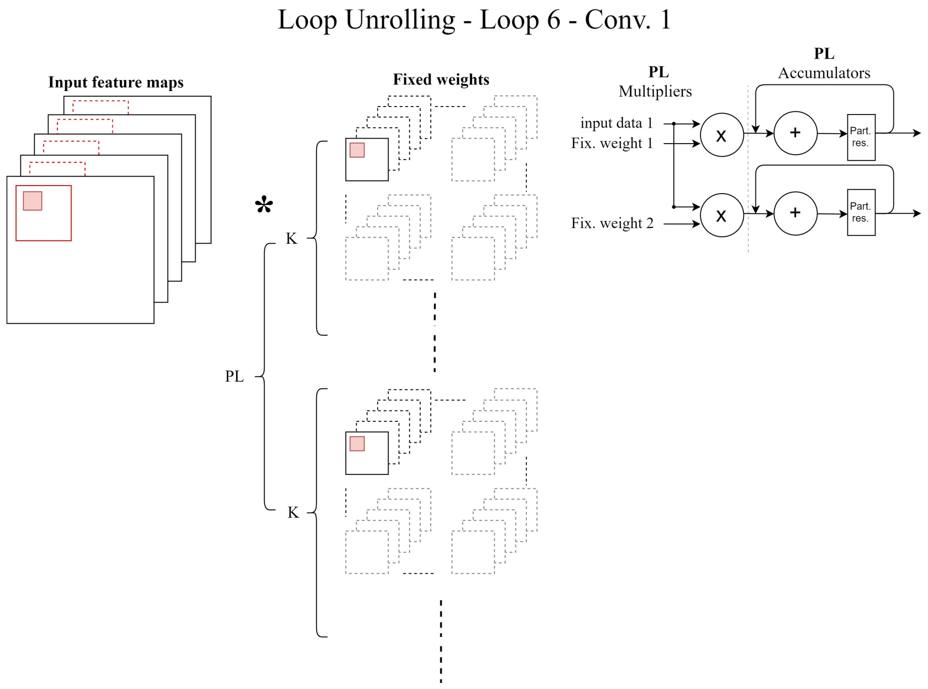


**Figure 4.16:** Unroll of the Loop 6 – Conv. 1

The unroll of the loop 6 for the conv. 1 allows to compute in parallel the s-v weights used to compute the data of different output channels. The number of required multipliers and accumulators is equal to the unroll variable PL.

- **Unroll of the Loop 1B**:

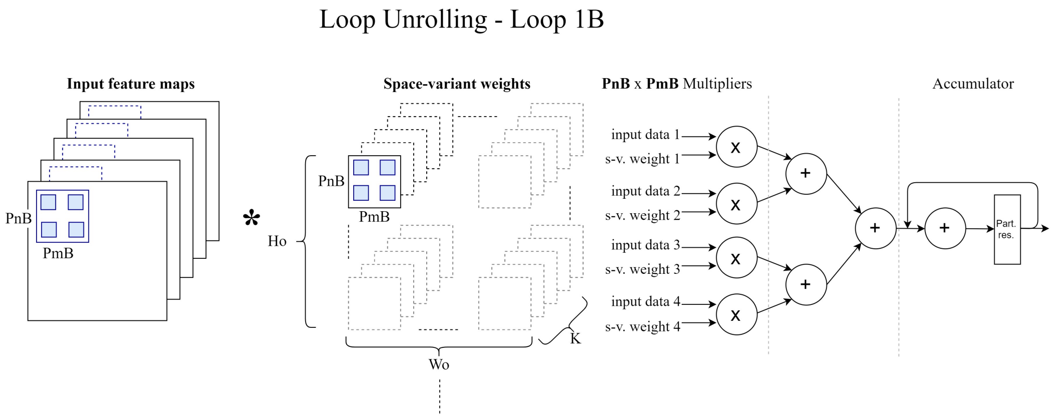


**Figure 4.17:** Unroll of the Loop 1B

The unroll of the loop 1B allows to perform in parallel the product between the space-variant weights and the input pixels within the same kernel of size $W_1 \times W_1$. The number of multipliers required is equal to $PnB \times PmB$. All these partial products are then added together through an adder tree. Finally, in order to compute the output data, an accumulator is needed to store the partial results.

- **Unroll of the Loop 2B**:

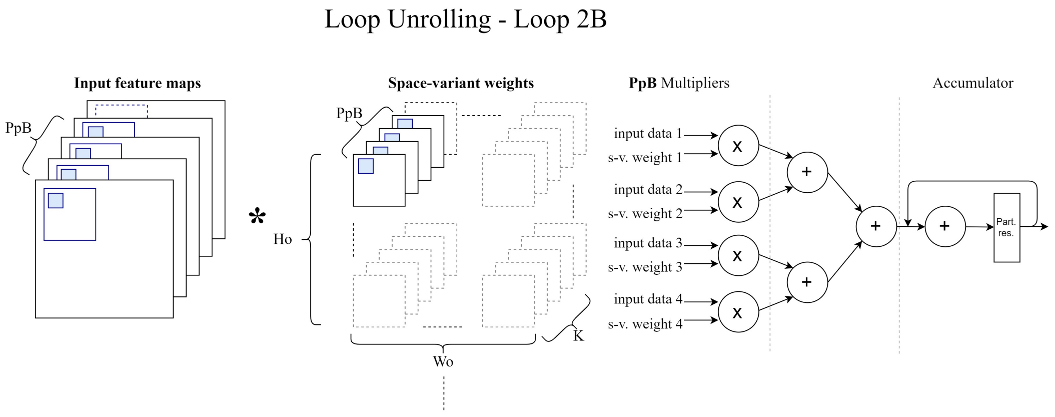


**Figure 4.18:** Unroll of the Loop 2B

At every cycle, with the unroll of the loop 2B, an output pixel is performed with the multiplication between the input data and the s-v weights of different channels. As it can be noticed, the architecture is the same as the previous one.

- **Unroll of the Loop 3B**:

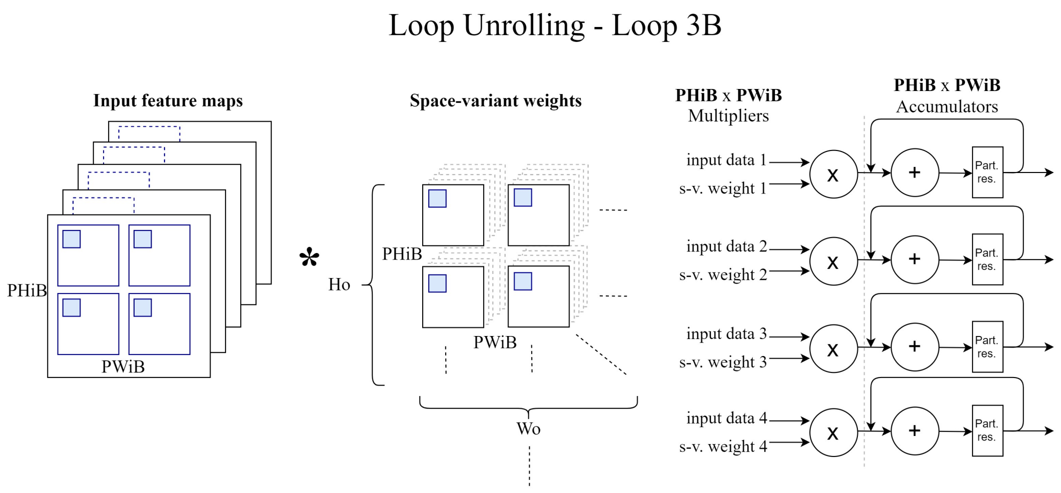


**Figure 4.19:** Unroll of the Loop 3B

Differently from the Loop 3A, the unroll of the loop 3B does not allow to reuse the same weight to perform simultaneously different output pixels. This is due to the fact that the second convolution uses space-variant weights instead of fixed weights. In this architecture the $PHiB \times PWiB$ multipliers compute the partial products that are then stored inside $PHiB \times PWiB$ accumulators.

- **Unroll of the loop 6 – conv. 2**: Finally, the unroll of the loop 6 for the conv. 2 allows to compute in parallel the output pixels of different output channels. As it can be noticed, in this architecture the input data can be reused PL times.

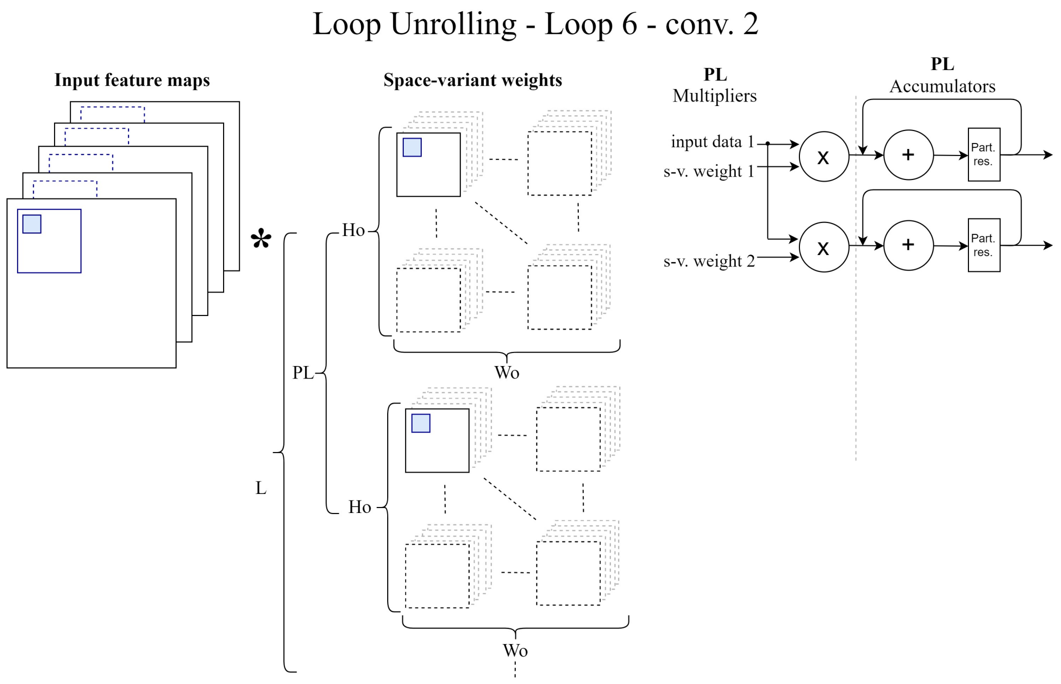


**Figure 4.20:** Unroll of the loop 6 – conv. 2

Finally, after viewing all the possible parallel combination both for conv.1 and conv.2, it can be possible to compute the number of multipliers (#Mul) required to compute the MAC operations. In fact, #Mul is given by multiplying the unroll variables listed in table 4.1 and table 4.2.

$$\#Mul\_conv1 = P_r \cdot P_s \cdot P_q \cdot P_{HiA} \cdot P_{WiB} \cdot P_{nA} \cdot P_{mA} \cdot P_{pA} \cdot P_L \tag{4.3.4}$$

$$\#Mul\_conv2 = P_{nB} \cdot P_{mB} \cdot P_{pB} \cdot P_{HiB} \cdot P_{WiB} \cdot P_L \tag{4.3.5}$$

### 4.3.2 Loop Tiling

Since the size of the on-chip memory is quite limited, generally, it is not possible to store all the data and the weights necessary to compute all the output feature maps of one NLC layer. So, this brings to store only a portion of data and weights on-chip. With the loop tiling technique, the volume of the data is subdivided in 'blocks' which are transferred from off-chip to on-chip and vice-versa. However the choice of the tiling variables for the different loop parameters determines at least the minimum size of the on-chip memory required inside the FPGA. In fact, the on-chip memory is needed to store the input data, the fixed weights, the s-v weights and the output data. In the following figures are shown the tiling section of data that are buffered on-chip. All the tiling variables refer to the tables 4.1 and 4.2:

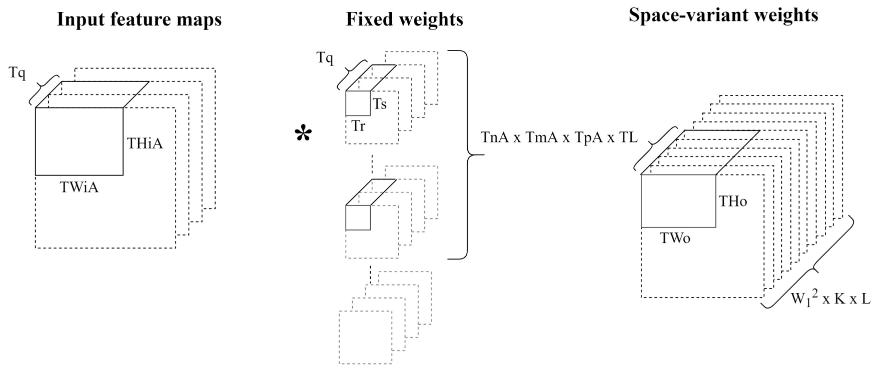


**Figure 4.21:** Loop tiling of conv.1

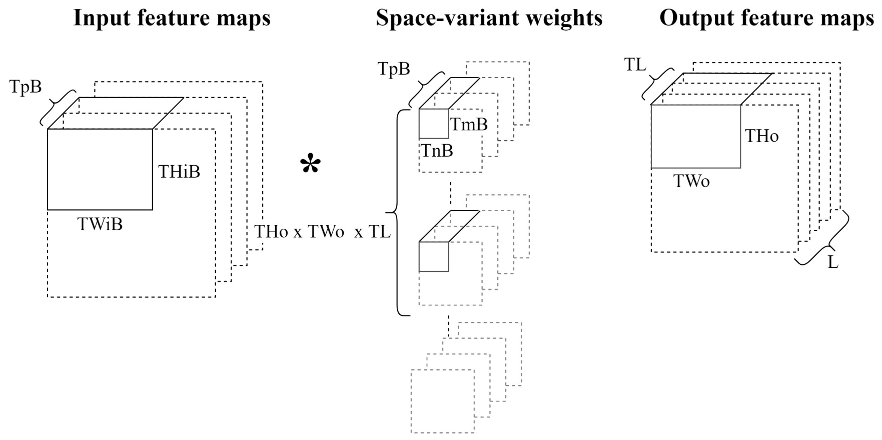


**Figure 4.22:** Loop tiling of conv.2

In the following tables are listed the minimum size of on-chip memory required to store the data required to perform the assigned task:

| | **Conv. 1** |
|---|---|
| *Data Input mem.* | TWiA x THiA x Tq x (data_in_datawidth) |
| *Input f.w mem.* | Tr x Ts x Tq x TnA x TmA x TpA x TL x (f.w_datawidth) |
| *Data Output mem.* | TWo x THo x TnA x TmA x TpA x TL x (s-v_w_datawidth) |

**Table 4.4:** Buffer size for conv.1

| | **Conv. 2** |
|---|---|
| *Data Input mem.* | THiB x TWiB x TpB x (data_in_datawidth) |
| *Input s-v w mem.* | TnB x TmB x TpB x THo x TWo x TL x (s-v_w_datawidth) |
| *Data Output mem.* | TWo x THo x TL x (data_out_datawidth) |

**Table 4.5:** Buffer size for conv.2

### 4.3.3 Loop Interchange

Differently from standard linear convolutional layers, the NLC layer consists of a first convolution that computes s-v weights, a point-wise AF, a normalization process, and a second convolution that uses the s-v weights instead of the fixed weights. This different structure leads on a more complicated management of the blocks of data to move. Furtherly, another important question is related on the on-chip memory that is required to store the s-v weights computed with conv. 1. In fact, if for the sake of simplicity a multi-input single-output NLC layer is considered, the data volume of the s-v weights computed is equal to $Ho \times Wo \times W_1^2 \times K$. Moreover, looking at the equation 2.2.11, to have the normalized weights, the normalization process requires that for a given $v_{i,j,l}(n,m,p)$ all the s-v weights $\hat{v}_{i,j,l}(n,m,p)$ with index $\hat{v}_{i,j,l}(:,:,:)$ must be previously computed. So, due to the data dependencies, this last condition imposes a constraint on the minimum amount of data $(W_1^2 \times K)$ to be processed before the execution of the conv.2. So, given a NLC layer with a number of input feature maps and output feature maps, it can be possible that for the chosen architecture, the input-dependents weights required cannot be stored all on-chip. For this reason, the order in which the nested loops are executed has a big impact on the data reuse and on the data dependencies issue. This leads to two different main categories of possible architectures [15]:

(A) **s-v weights stored off-chip**: with this architectural solution, part of the s-v weights is stored off-chip. In fact, in addition to the data and fixed weights, the data-bus is involved in the transaction of the s-v weights that increases the data movement between on-chip and off-chip memory. So, this choice leads to a lower computation to communication ratio (CCR) coefficient, which is the ratio between the computational effort and the data transaction between the off-chip and on-chip memory.

(B) **s-v weights stored on-chip**: with this architectural solution, all the portion of s-v weights that is needed to ensure the normalization process is stored on-chip. With this choice, the data transaction between the on-chip and off-chip memory is only related on the loading and saving of the input pixels, the fixed trained weights and the output pixels. So, under the same computational effort, with this architecture a higher CCR is achieved respect to the previous case, leading on better performances.

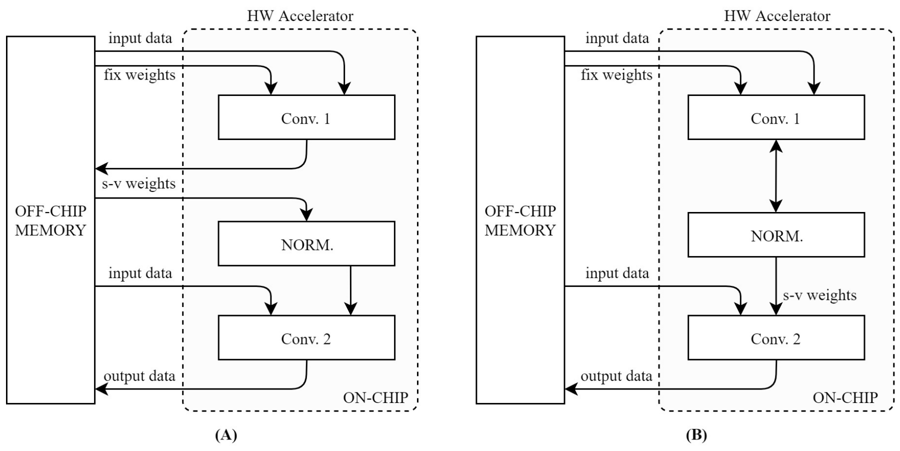


**Figure 4.23:** s-v weights stored off-chip (A) vs s-v weights stored on-chip (B)

# Chapter 5

# Analysis and impact of the loop variables on a NLC layer

In this chapter, an analysis on how the principal design variables change by varying the tiling and unrolling variables is conducted.

## 5.1 Latency of the System

Referring on the tables 4.1 and 4.2, the number of multiplications that must be performed in each NLC layer is given by the following equation:

$$\#TOT\_M = M\_CONV1 + M\_AF\_NORM + M\_CONV2 \tag{5.1.1}$$

Focusing on the multiplications ($M\_CONV1$) of the first convolution, which performs the input dependent weights, and on the multiplications ($M\_CONV2$) of the non linear convolution, these two terms can be expressed as:

$$M\_CONV1 = L \cdot Ho \cdot Wo \cdot W_1^2 \cdot K^2 \cdot W_2^2 \tag{5.1.2}$$

$$M\_CONV2 = L \cdot Ho \cdot Wo \cdot W_1^2 \cdot K \tag{5.1.3}$$

So, related on conv.1 and conv.2, the number of executed cycles can be calculated as the product between the number of inter-tile cycles and the intra-tile cycles. In particular the inter-tile loop order represents how data are transferred from off-chip to on-chip, while the intra-tile loop order represent how data are transferred form on-chip to the PEs.

$$\begin{aligned} \#intertile\_cycles\_conv1 = & \left\lceil \frac{\mathrm{L}}{\mathrm{T_L}} \right\rceil \left\lceil \frac{\mathrm{H_o}}{\mathrm{T_{Ho}}} \right\rceil \left\lceil \frac{\mathrm{W_o}}{\mathrm{T_{Wo}}} \right\rceil \left\lceil \frac{\mathrm{W_1}}{\mathrm{T_{nA}}} \right\rceil \left\lceil \frac{\mathrm{W_1}}{\mathrm{T_{mA}}} \right\rceil \\ & \times \left\lceil \frac{\mathrm{K}}{\mathrm{T_q}} \right\rceil \left\lceil \frac{\mathrm{K}}{\mathrm{T_pA}} \right\rceil \left\lceil \frac{\mathrm{W_2}}{\mathrm{T_r}} \right\rceil \left\lceil \frac{\mathrm{W_2}}{\mathrm{T_s}} \right\rceil \end{aligned} \tag{5.1.4}$$

$$\begin{aligned} \#intratile\_cycles\_conv1 = & \left\lceil \frac{\mathrm{T_L}}{\mathrm{P_L}} \right\rceil \left\lceil \frac{\mathrm{T_{Ho}}}{\mathrm{P_{Ho}}} \right\rceil \left\lceil \frac{\mathrm{T_{Wo}}}{\mathrm{P_{Wo}}} \right\rceil \left\lceil \frac{\mathrm{T_{nA}}}{\mathrm{P_{nA}}} \right\rceil \left\lceil \frac{\mathrm{T_{mA}}}{\mathrm{P_{mA}}} \right\rceil \\ & \times \left\lceil \frac{\mathrm{T_q}}{\mathrm{P_q}} \right\rceil \left\lceil \frac{\mathrm{T_pA}}{\mathrm{P_pA}} \right\rceil \left\lceil \frac{\mathrm{T_r}}{\mathrm{T_r}} \right\rceil \left\lceil \frac{\mathrm{T_s}}{\mathrm{P_s}} \right\rceil \end{aligned} \tag{5.1.5}$$

$$\#cycles\_conv1 = \#intertile\_cycles\_conv1 \times \#intratile\_cycles\_conv1 \tag{5.1.6}$$

$$\begin{aligned} \#intertile\_cycles\_conv2 = & \left\lceil \frac{\mathrm{L}}{\mathrm{T_L}} \right\rceil \left\lceil \frac{\mathrm{Ho}}{\mathrm{T_{Ho}}} \right\rceil \left\lceil \frac{\mathrm{Wo}}{\mathrm{T_{Wo}}} \right\rceil \left\lceil \frac{\mathrm{W_1}}{\mathrm{T_{nB}}} \right\rceil \left\lceil \frac{\mathrm{W_1}}{\mathrm{T_{mB}}} \right\rceil \\ & \times \left\lceil \frac{\mathrm{K}}{\mathrm{T_{pB}}} \right\rceil \end{aligned} \tag{5.1.7}$$

$$\begin{aligned} \#intratile\_cycles\_conv2 = & \left\lceil \frac{\mathrm{T_L}}{\mathrm{P_L}} \right\rceil \left\lceil \frac{\mathrm{T_{Ho}}}{\mathrm{P_{Ho}}} \right\rceil \left\lceil \frac{\mathrm{T_{Wo}}}{\mathrm{P_{Wo}}} \right\rceil \left\lceil \frac{\mathrm{T_{nB}}}{\mathrm{P_{nB}}} \right\rceil \left\lceil \frac{\mathrm{T_{mB}}}{\mathrm{P_{mB}}} \right\rceil \\ & \times \left\lceil \frac{\mathrm{T_{pB}}}{\mathrm{P_{pB}}} \right\rceil \end{aligned} \tag{5.1.8}$$

$$\#cycles\_conv2 = \#intertile\_cycles\_conv2 \times \#intratile\_cycles\_conv2 \tag{5.1.9}$$

As can be seen, if the ratio between the variables is not writable as an integer, the multipliers and the memory are not completely used. So, for this reason, in order to minimize the latency it is convenient to impose the unroll and tiling variables such that:

$$min \left( \left\lceil \frac{parameter}{T_{parameter}} \right\rceil - \frac{parameter}{T_{parameter}} \right) \tag{5.1.10}$$

$$min\left(\left\lceil\frac{T_{parameter}}{P_{parameter}}\right\rceil - \frac{T_{parameter}}{P_{parameter}}\right) \tag{5.1.11}$$

However, it must be underlined that these equations do not take into account the other further contributes. One of them is as example the delay introduced per each data transaction between the off-chip and on-chip memory.

## 5.2 Size of the on-chip memory

The choice of the target FPGA imposes a constraint on the maximum size of on-chip memory available. In fact, this last one must be large enough to store the input pixels, the weights, the spatially variant parameters and the output pixels needed during the execution of the task. Relating on conv.1 and conv.2, the formula which is used to compute the size of the on-chip memory can be expressed as:

$$on\_chip\_size = I\_PXL + F\_W + SV\_W + O\_PXL \tag{5.2.1}$$

where:

- **I_PXL** is the contribute of the input pixels. Under the hypothesis of execute one convolution between conv.1 and conv.2 at the time, this memory must be big enough to store the entire data block of input feature maps needed to perform either conv.1 or conv.2. Referring on the tables 4.4 and 4.5, the size of this buffer must the maximum between the sizes of the input buffers required to compute the two convolutions:

$$\begin{aligned} I\_PXL = MAX(&TWiA \cdot THiA \cdot Tq \cdot (data\_in\_datawidth), \\ &THiB \cdot TWiB \cdot TpB \cdot (data\_in\_datawidth)) \end{aligned} \tag{5.2.2}$$

- **F_W** is the contribute of the fixed weights. The storage element must be big enough to store the tiling block of fixed weights used in conv.1. This size is taken from table 4.4:

$$F\_W = Tr \cdot Ts \cdot Tq \cdot TnA \cdot TmA \cdot TpA \cdot TL \cdot (fw\_datawidth) \tag{5.2.3}$$

- **SV_W** is the contribute of the on-chip memory needed to store the spatially variant weights needed to perform the conv.2. As discussed in 4.3.3, due to the data dependencies introduced by the normalization process, $W_1^2 \times K$ s-v weights must be calculated to compute a single output pixel. Based on this last requirement, if the inter-tile loop 3A and loop 3B are computed as first the memory required is the following:

$$SV\_W = TWo \cdot THo \cdot W_1^2 \cdot K \cdot TL \cdot (sv\_w\_datawidth) \tag{5.2.4}$$

While, for the other combinations of order of execution of inter-tile loops:

$$SV\_W = Wo \cdot Ho \cdot W_1^2 \cdot K \cdot TL \cdot (sv\_w\_datawidth) \tag{5.2.5}$$

As it can be noticed the second solution is a sub-optimal solution, because it requires a huge amount of on-chip memory. To solve this problem, a possible solution could be to store part of the performed s-v weights off-chip. Unfortunately, as consequence, this approach increases the number of accesses to the off-chip memory, lowering the performances.

**O_PXL** is the contribute of the memory needed to store the partial sums regarding the conv.2. These last ones are the partial results of the product between the s-v weights and the input pixels which are accumulated to compute the output pixels. By referring on the table 4.5:

$$O\_PXL = TWo \cdot THo \cdot TL \cdot (data\_out\_datawidth) \tag{5.2.6}$$

## 5.3 Number of accesses to the off-chip memory

With the hypothesis of using an architectural solution in which all the data and partial sums are stored on-chip, this section deals with the estimation of the number of accesses to the off-chip memory. As it was discussed in 4.1, by reducing the accesses to the off-chip memory, the overall performances increase. For this reason a data reuse strategy combined with a proper order of execution of the nested loops is essential to keep the accesses to the off-chip memory as lower as possible. Since the spatially-variants parameters are all stored on-chip, the accesses to the off-chip memory are only addresses to fetch the input pixels required to perform the conv.1 and conv.2 and to fetch the fixed weights used in conv.1:

$$\#off\_chip\_acc = \#off\_chip\_px + \#off\_chip\_fw \tag{5.3.1}$$

Regarding the first contribute, its value can be expressed as:

$$\#off\_chip\_px = \#off\_chip\_px\_conv1 + \#off\_chip\_px\_conv2 \tag{5.3.2}$$

where:

- **#off_chip_px_conv1** represents the number of accesses to the off-chip memory needed to fetch the pixels used in conv.1. The following table shows a subset of the possible values assumed by $\#off\_chip\_px\_conv1$ by varying the order of execution of the inter-tile nested loops.

| **loop executed before L3A or L2A** | **#off_chip_px_conv1** |
|---|---|
| none | $\left\lceil\frac{Ho}{THo}\right\rceil\cdot\left\lceil\frac{Wo}{TWo}\right\rceil\cdot\left\lceil\frac{K}{Tq}\right\rceil$ |
| L6 | $\left\lceil\frac{Ho}{THo}\right\rceil\cdot\left\lceil\frac{Wo}{TWo}\right\rceil\cdot\left\lceil\frac{K}{Tq}\right\rceil\cdot\left\lceil\frac{L}{TL}\right\rceil$ |
| L6, L5A | $\left\lceil\frac{Ho}{THo}\right\rceil\cdot\left\lceil\frac{Wo}{TWo}\right\rceil\cdot\left\lceil\frac{K}{Tq}\right\rceil\cdot\left\lceil\frac{L}{TL}\right\rceil\cdot\left\lceil\frac{K}{TpA}\right\rceil$ |
| L6, L4A | $\left\lceil\frac{Ho}{THo}\right\rceil\cdot\left\lceil\frac{Wo}{TWo}\right\rceil\cdot\left\lceil\frac{K}{Tq}\right\rceil\cdot\left\lceil\frac{L}{TL}\right\rceil\cdot\left\lceil\frac{W1}{TnA}\right\rceil\cdot\left\lceil\frac{W1}{TmA}\right\rceil$ |
| L6, L1A | $\left\lceil\frac{Ho}{THo}\right\rceil\cdot\left\lceil\frac{Wo}{TWo}\right\rceil\cdot\left\lceil\frac{K}{Tq}\right\rceil\cdot\left\lceil\frac{L}{TL}\right\rceil\cdot\left\lceil\frac{W2}{Tr}\right\rceil\cdot\left\lceil\frac{W2}{Ts}\right\rceil$ |
| L6, L5A, L4A | $\left\lceil\frac{Ho}{THo}\right\rceil\cdot\left\lceil\frac{Wo}{TWo}\right\rceil\cdot\left\lceil\frac{K}{Tq}\right\rceil\cdot\left\lceil\frac{L}{TL}\right\rceil\cdot\left\lceil\frac{K}{TpA}\right\rceil\cdot\left\lceil\frac{W1}{TnA}\right\rceil\cdot\left\lceil\frac{W1}{TmA}\right\rceil$ |
| L6, L5A, L1A | $\left\lceil\frac{Ho}{THo}\right\rceil\cdot\left\lceil\frac{Wo}{TWo}\right\rceil\cdot\left\lceil\frac{K}{Tq}\right\rceil\cdot\left\lceil\frac{L}{TL}\right\rceil\cdot\left\lceil\frac{K}{TpA}\right\rceil\cdot\left\lceil\frac{W2}{Tr}\right\rceil\cdot\left\lceil\frac{W2}{Ts}\right\rceil$ |
| L6, L4A, L1A | $\left\lceil\frac{Ho}{THo}\right\rceil\cdot\left\lceil\frac{Wo}{TWo}\right\rceil\cdot\left\lceil\frac{K}{Tq}\right\rceil\cdot\left\lceil\frac{L}{TL}\right\rceil\cdot\left\lceil\frac{W1}{TnA}\right\rceil\cdot\left\lceil\frac{W1}{TmA}\right\rceil\cdot\left\lceil\frac{W2}{Tr}\right\rceil\cdot\left\lceil\frac{W2}{Ts}\right\rceil$ |
| L6, L5A, L4A, L1A | $\left\lceil\frac{Ho}{THo}\right\rceil\cdot\left\lceil\frac{Wo}{TWo}\right\rceil\cdot\left\lceil\frac{K}{Tq}\right\rceil\cdot\left\lceil\frac{L}{TL}\right\rceil\cdot\left\lceil\frac{K}{TpA}\right\rceil\cdot\left\lceil\frac{W1}{TnA}\right\rceil\cdot\left\lceil\frac{W1}{TmA}\right\rceil\cdot\left\lceil\frac{W2}{Tr}\right\rceil\cdot\left\lceil\frac{W2}{Ts}\right\rceil$ |

**Table 5.1:** Number of accesses to the off-chip memory required to fetch the pixels needed to perform conv.1 for different order of execution of inter-tile loops

As it can be noticed, the earlier execution of the loops 3A and 2A leads on an appreciable reduction of the accesses to the off-chip memory.

- **#off_chip_px_conv2** represents the number of accesses to the off-chip memory required to fetch the pixels used in conv.2. Since it is assumed that the required input dependent weights are stored on-chip, $\#off\_chip\_px\_conv2$ can be expressed as:

$$\#off\_chip\_px\_conv2 = \left\lceil\frac{Ho}{THo}\right\rceil\cdot\left\lceil\frac{Wo}{TWo}\right\rceil\cdot\left\lceil\frac{K}{TpB}\right\rceil\cdot\left\lceil\frac{L}{TL}\right\rceil \qquad (5.3.3)$$

By referring to the equation 5.3.1, the second contribute $\#off\_chip\_fw$ represents the number of accesses to the off-chip memory needed to fetch the fixed weights used in conv.1. As previously, the following table shows how $\#off\_chip\_fw$ changes by varying the order of execution of the inter-tile nested loops.

| **loop executed before L6 or L5A or L4A or L2A or L1A** | **#off_chip_fw** |
|---|---|
| none | $\left\lceil \frac{L}{TL} \right\rceil \cdot \left\lceil \frac{K}{TpA} \right\rceil \cdot \left\lceil \frac{W1}{TnA} \right\rceil \cdot \left\lceil \frac{W1}{TmA} \right\rceil \cdot \left\lceil \frac{K}{Tq} \right\rceil \cdot \left\lceil \frac{W2}{Tr} \right\rceil \cdot \left\lceil \frac{W2}{Ts} \right\rceil$ |
| L3A | $\left\lceil \frac{L}{TL} \right\rceil \cdot \left\lceil \frac{K}{TpA} \right\rceil \cdot \left\lceil \frac{W1}{TnA} \right\rceil \cdot \left\lceil \frac{W1}{TmA} \right\rceil \cdot \left\lceil \frac{K}{Tq} \right\rceil \cdot \left\lceil \frac{W2}{Tr} \right\rceil \cdot \left\lceil \frac{W2}{Ts} \right\rceil \cdot \left\lceil \frac{Ho}{THo} \right\rceil \cdot \left\lceil \frac{Wo}{TWo} \right\rceil$ |

**Table 5.2:** Number of accesses to the off-chip memory required to fetch the weights needed to perform conv.1 for different order of execution of inter-tile loops

As in the previous case, the earlier execution of the loops 6, 5A, 4A, 2A and 1A leads on a reduction on the number of accesses needed to fetch the fixed weights. In conclusion, it can be noticed that by interchanging the order of execution of the inter-tile loops, if on one side $\#off\_chip\_px$ decreases, $\#off\_chip\_fw$ increases.

## 5.4 Optimal design space exploration for a NCL layer

The reduction of the accesses to the off-chip memory leads on many advantages, such as a lower power consumption and a lower latency of the overall execution. Unfortunately, usually a lower number of accesses leads to increase the required on-chip memory. So, a trade-off between these two needs to be found to optimize the design process of a NLC layer.
In order to compute the couple of values $(on\_chip\_size, \#off\_chip\_acc)$ for different orders of execution of inter-tile loop and for different values of tiling parameters, a python script has been written. The script can be found in appendix A.

During the execution of the code, given the specific of the NLC layer, several nested *For* loops generate various combinations of the order of execution of the nested loops and different combinations of tiling variables with range $1 \leq T_{parameter} \leq Parameter$. So, at each combination of the previous ones, the functions $'OnChipMem()'$ and $'TotAccOffChipMEM()'$ compute the size of the on-chip memory required and the number of accesses to the off-chip memory respectively. These functions are based on the theory explained in 5.2 and 5.3. The couples $(on\_chip\_size, \#off\_chip\_acc)$ generated are then saved on a .csv file. Finally, these points are plotted in a 2D graph, in which on the y axis and on the x axis are reported the $\#off\_chip\_acc$ and the $on\_chip\_size$ respectively.
As example, it has been supposed to explore the design space for a NLC layer with the following parameters:

| | **Input feature size (Height x Width x depth)** | **Filter W1 (Height x Width)** | **Filter W2 (Height x Width)** | **Output channels L** |
|---|---|---|---|---|
| NCL layer | 512 x 512 x 3 | 3 x 3 | 3 x 3 | 6 |

**Table 5.3:** Parameters tested for the optimal design space exploration of a NLC layer

While, for what concern the datawidth of the input pixels, of the fixed weights, of the s-v weights and of the output pixels, it has been assumed:

| | **input pixel** | **fix. weight** | **s.v weight** | **output pixel** |
|---|---|---|---|---|
| datawidth | 8 bit | 16 bit | 16 bit | 8 bit |

**Table 5.4:** Datawidth for the data used during the design space exploration

### 5.4.1 #accesses to off-chip mem. vs on-chip mem. size

In figure 5.1 it has been plotted the #off_chip_acc vs on_chip_size graph for an optimized and not optimized order of execution of inter-tile loops. As it was previously discussed, in 5.2 the on-chip memory addressed to store the s-v weights (SV_W) depends on the loop order. As consequence, fixed the #off_chip_acc, the blue dots compared with the orange ones require a higher size of on-chip memory.

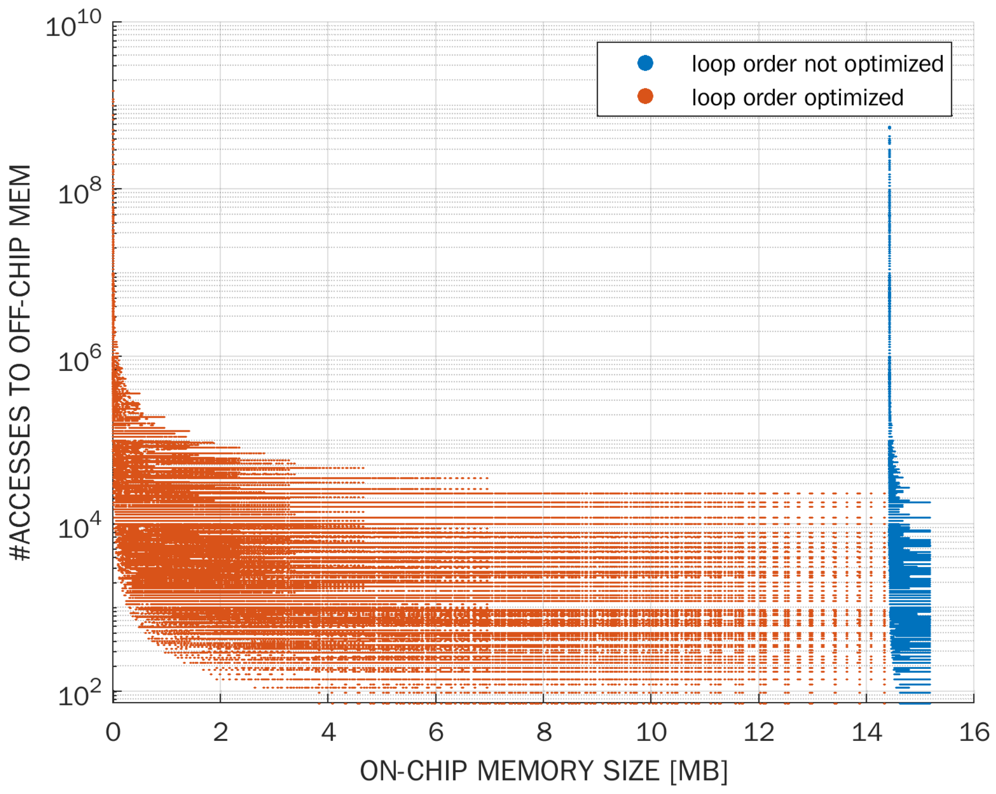


**Figure 5.1:** #off_chip_acc vs on_chip_size for an optimized and not optimized order of execution of inter-tile loops

In the following figures, for the sake of simplicity, they have been plotted only the couples $(on\_chip\_size, \#off\_chip\_acc)$ which belong to a optimized loop order as viewed in figure 5.1.
In figure 5.2 it has been plotted #off_chip_acc vs on_chip_size in function of $Tr \leq W_2$ and $Ts \leq W_2$. As it can be noticed, higher values of $Tr \times Ts$ lead to reduce the accesses to the off-chip memory.

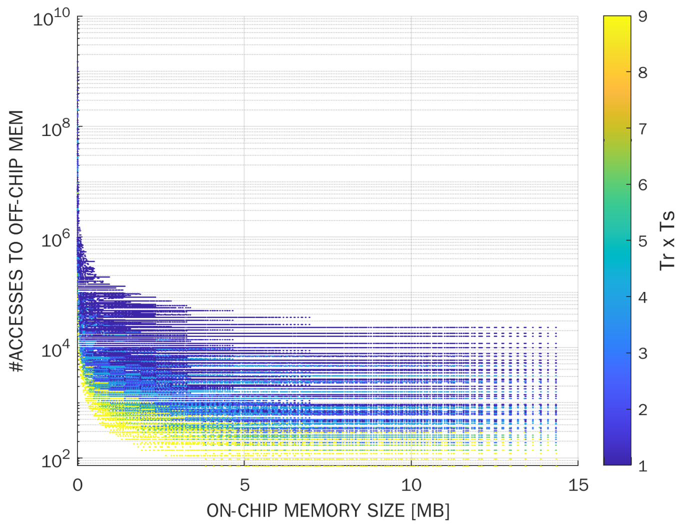


**Figure 5.2:** #off_chip_acc vs on_chip_size in function of $Tr \leq W_2$ and $Ts \leq W_2$

In figure 5.3 it has been plotted #off_chip_acc vs on_chip_size in function of $Tq \leq K$ . As expected, it can be noticed that higher values of input channels $Tq$ loaded lead to reduce the accesses to the off-chip memory.

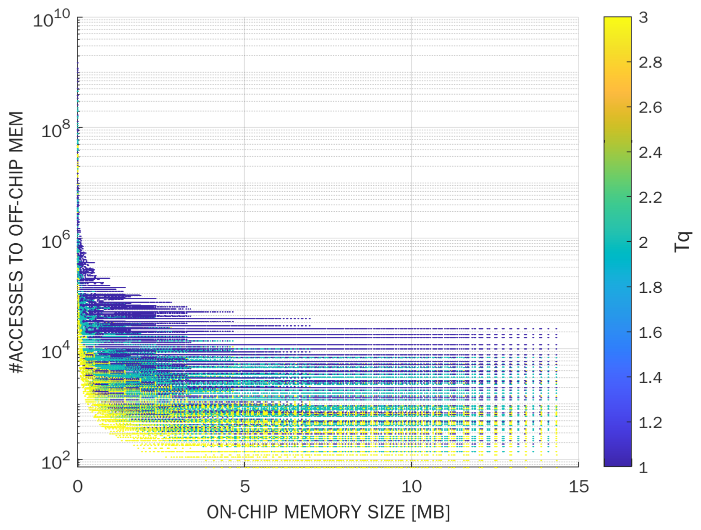


**Figure 5.3:** #off_chip_acc vs on_chip_size in function of $Tq \leq K$

In figure 5.4 it has been plotted #off_chip_acc vs on_chip_size in function of $TWo \leq Wo$ and $THo \leq Ho$ . As expected, it can be noticed that higher values of input data loaded to perform the output $TWo \times THo$ lead to reduce the accesses to the off-chip memory, but also they lead to increase significantly the on-chip memory required. So, a trade-off is needed to guarantee that the target FPGA is capable to store the overall data.

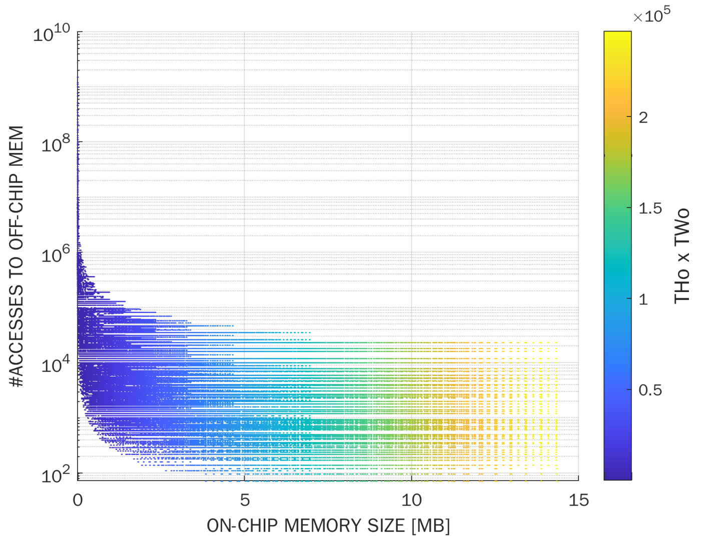


**Figure 5.4:** #off_chip_acc vs on_chip_size in function of $TWo \leq Wo$ and $THo \leq Ho$

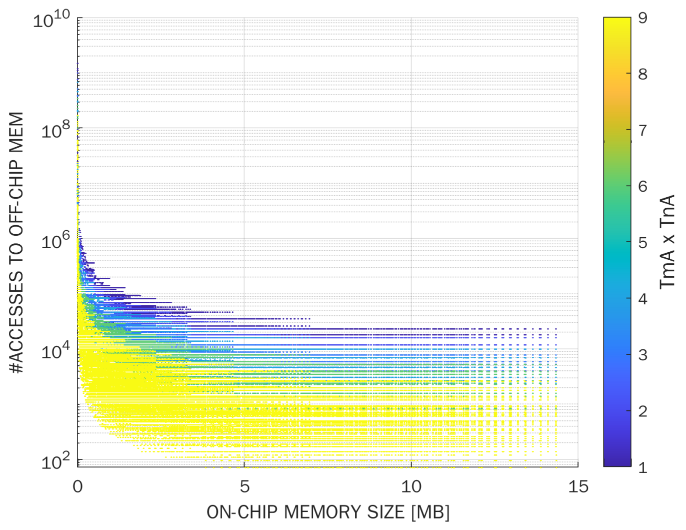


**Figure 5.5:** #off_chip_acc vs on_chip_size in function of $TnA \leq W_1$ and $TmA \leq W_1$

In figure 5.5 it has been plotted #off_chip_acc vs on_chip_size in function of $TnA \leq W_1$ and $TmA \leq W_1$ . As expected, since $TnA \times TmA$ is a variable of #off_chip_acc (tables 5.1 and 5.2 ),it can be noticed that higher values of loaded fixed weights and loaded pixels needed to perform the space variant weights lead to reduce the accesses to the off-chip memory.

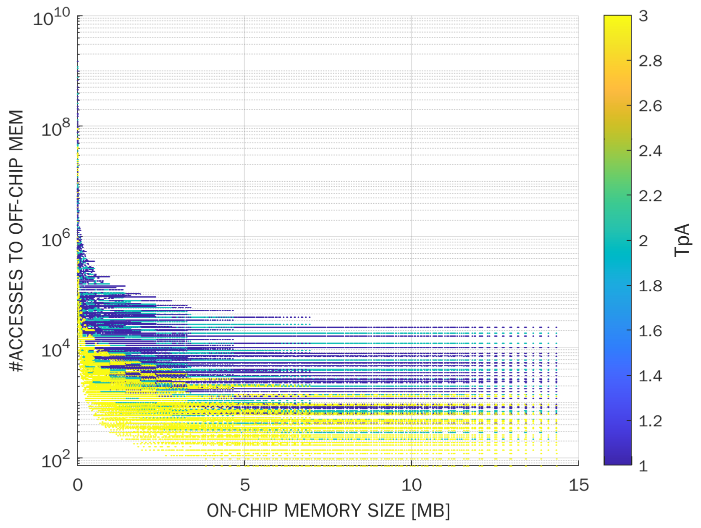


**Figure 5.6:** #off_chip_acc vs on_chip_size in function of $TpA \leq K$

In figure 5.6 it has been plotted #off_chip_acc vs on_chip_size in function of of $TpA \leq K$ . As in the previous case, since $TpA$ is a variable of #off_chip_acc (tables 5.1 and 5.2 ) , it can be noticed that higher values of loaded fixed weights and loaded pixels needed to perform the space variant weights lead to reduce the accesses to the off-chip memory.

In conclusion, the best optimized solution should be the solution which has as minimum both the on-chip memory size and the number of accesses to the off-chip memory [14]. However, since on-chip memory for the target FPGA is fixed, a possible choice could be to fix an upper limit to the maximum on-chip memory available. After that, the optimal solution will be the solution with the lowest number of accesses to the off-chip memory among the possible solutions. For instance, by referring on the previously simulation with the input parameters listed in 5.3 and 5.4, they have been listed the optimal values of tiling variables in function of different maximum sizes of on-chip memory:

| Max. on-chip memory size | #accesses to off-chip mem. | Tr x Ts | Tq | TWo x THo | TmA x TnA | TpA | TpB | TL |
|---|---|---|---|---|---|---|---|---|
| 100 KB | 3.2E+03 | 9 | 3 | 1617 | 9 | 3 | 3 | 1 |
| 256 KB | 1.2E+03 | 9 | 3 | 4257 | 9 | 3 | 3 | 1 |
| 500 KB | 5.8E+02 | 9 | 3 | 8481 | 9 | 3 | 3 | 1 |
| 1.0 MB | 2.9E+02 | 9 | 3 | 16705 | 9 | 3 | 3 | 1 |
| 1.5 MB | 2.2E+02 | 9 | 3 | 24929 | 9 | 3 | 3 | 1 |
| 2.0 MB | 1.4E+02 | 9 | 3 | 33153 | 9 | 3 | 3 | 1 |

**Table 5.5:** Optimal tiling variables for different maximum size of on-chip memory available

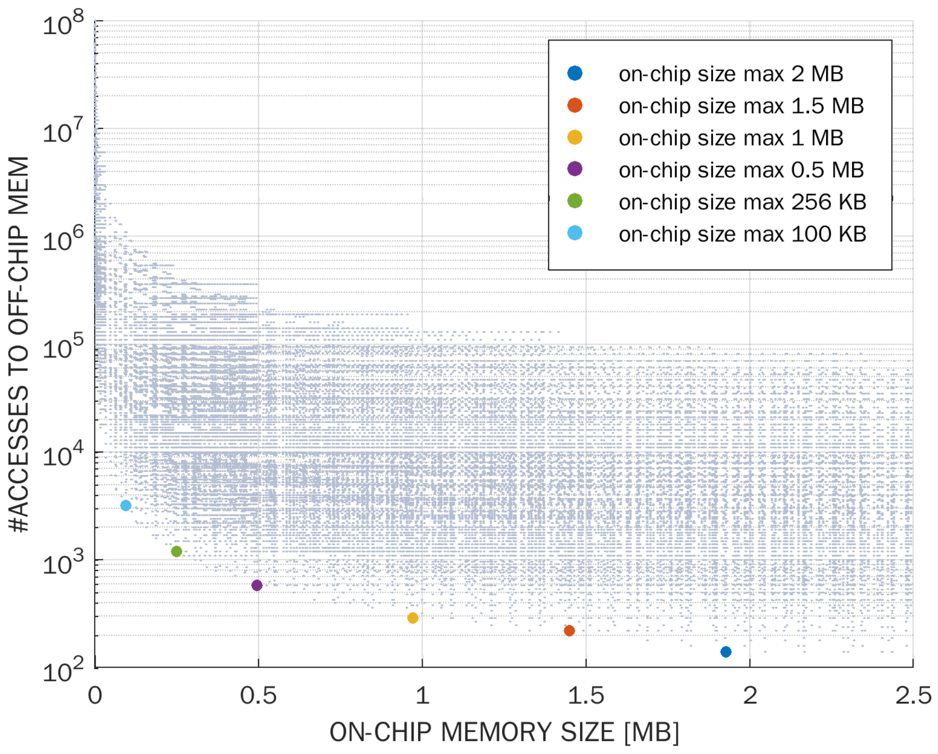


**Figure 5.7:** Optimal design of a NLC layer for different maximum size of on-chip memory available

As it can be noticed on table 5.5 and figure 5.7, among the optimal solutions, $TWo \times THo$ is the only tuneable parameter. In fact, if this last one term increases the number of accesses to off-chip memory decreases as consequence, because more data can be stored and reused on-chip.

Chapter 6

# Proposed NLC layer Accelerator

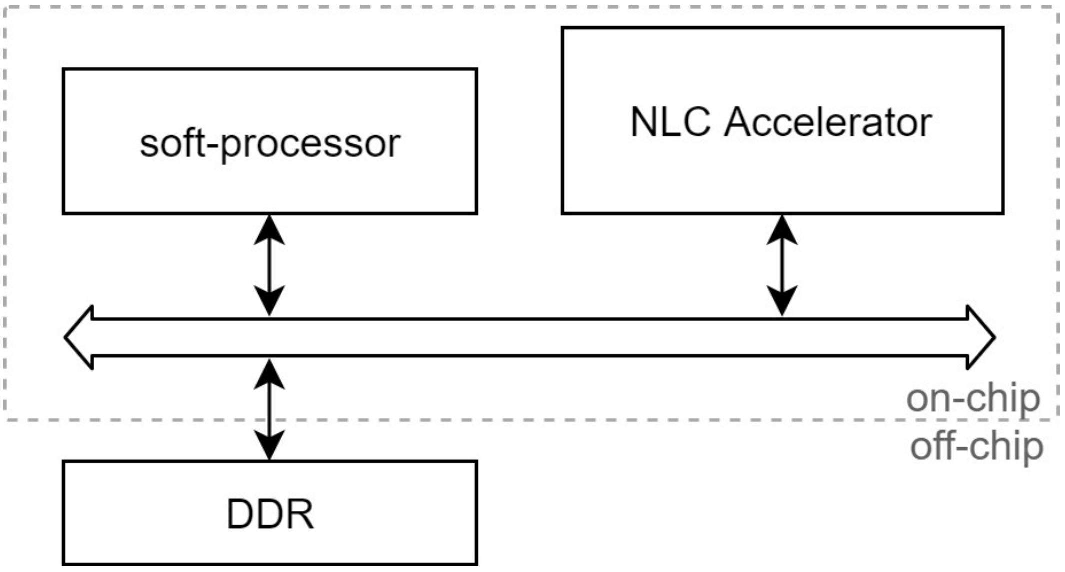


**Figure 6.1:** Generic scheme of NLCN HW accelerator

In this chapter, a proposed NLC layer accelerator is explained. Since the execution of a NLC layer involves a more complex dataflow respect to the classical convolutional layer, the use of a soft-processor could benefit on the design process. In fact, this last one could be addressed to manage the data transaction between off-chip and on-chip memory (inter-tile loops). With this design strategy, the *NLC Accelerator block* will be only addressed on the execution of the convolutions tasks and normalization process (intra-tile loops).

## 6.1 HW description of a NLC Accelerator block

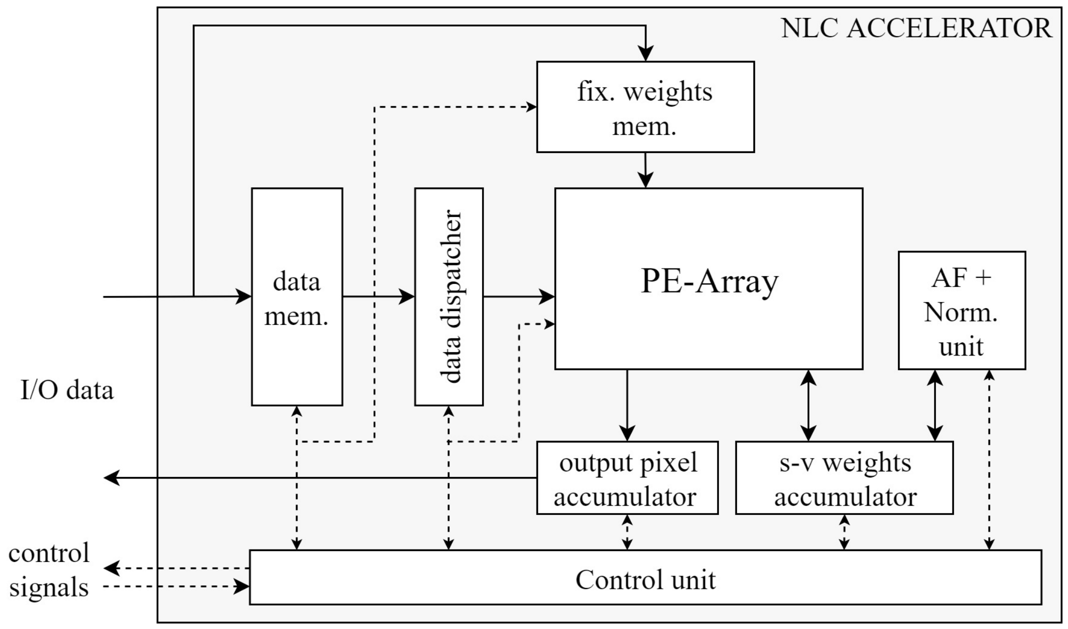


**Figure 6.2:** Proposed scheme of NLC accelerator

In figure 6.2 it is shown an example of NLC accelerator block. In particular, this block deals with the computation of the intra-tile loops described in the pseudocode 4.5. As hypothesis, this architecture is supposed to store on-chip all the input dependent weights needed to perform the normalization task and the non-linear convolution. In turn, this HW block is composed of several sub-blocks:

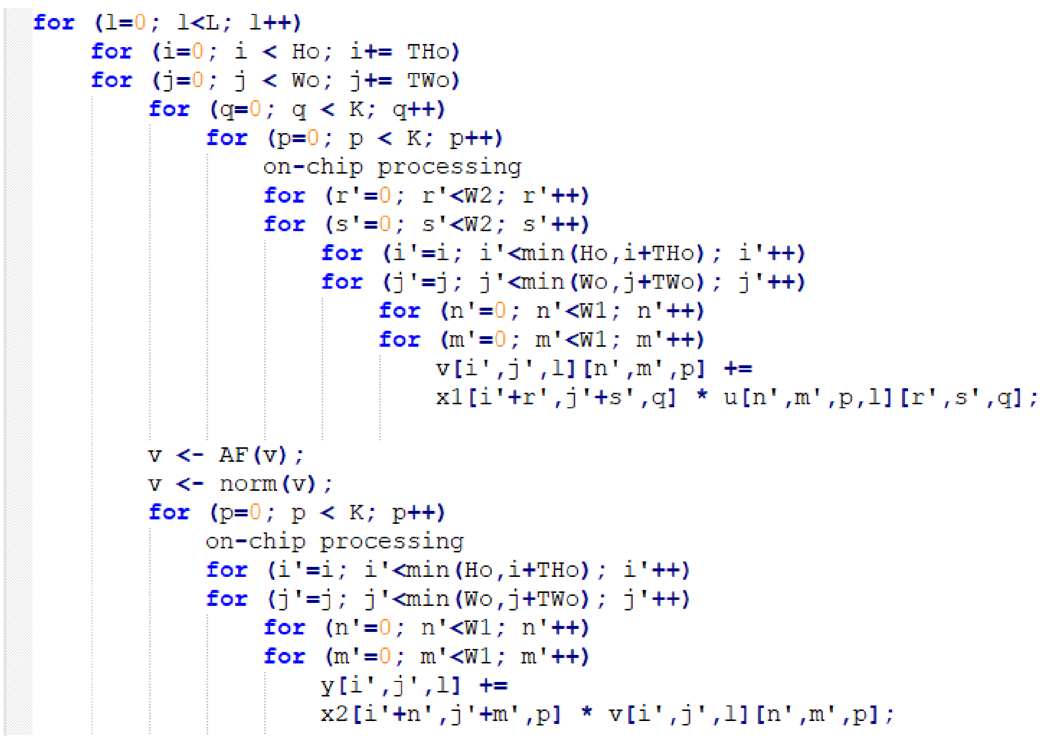
- **Data Memory** is the memory which stores on-chip the input pixels needed to perform either the first linear convolution or the second non-linear convolution.
- **Fixed Weights Memory** is the memory which stores on-chip the portion of fixed weights needed to perform the first linear convolution.
- **Data Dispatcher** is the logic block which is addressed to reschedule the dataflow of the pixels stored on the on-chip memory. In fact, to perform the convolution operation, the PE-Array needs a proper dataflow of the input data.
- **PE-Array** is the logic-block which performs the convolution operation both for the first linear convolution and for the second non-linear convolution. However, since only one convolution task at time is performed, to reduce the

number of PEs as much as possible, it could be possible to share part of the hardware resources (multipliers and adders).

- **S-V Weights Accumulator** is the block which accumulates all the partial sums needed to perform the input dependent weights. Further, this accumulator must be big enough to store all the s-v weights required to perform the normalization of these without additional data transaction between on-chip and off-chip memory.
- **AF + Normalization Unit** is the block which performs the point-wise AF and the normalization of the space variant weights stored in the *s-v weights accumulator* block as described in section 2.2.2
- **Output Pixel Accumulator** is the block which accumulates all the partial sums needed to perform the output pixels. These output pixels are computed by accumulating the partial products which are performed between the input pixels and the s-v weights, inside the *PE-Array* block. Once that the value of the output pixels is valid, these are transferred to the off-chip memory.
- **Control Unit** is the logic block which is addressed to control the execution of the tasks inside the *NLC Accelerator* module. This block communicates with the soft-processor in order to ensure the correct execution of the assigned tasks. In particular, a start signal and a done signal are needed. In fact, these indicates respectively the start condition and the done condition for the assigned task. Further, to make simply the control of the different tasks, it is convenient to organize the control unit in a hierarchical structure. In this way, each sub-control unit will ensure the correct execution of the assigned task.

In particular, the proposed thesis work has been focused on the design of the PE-Array which is the computational core of the HW accelerator, used to perform both the first linear convolution (conv.1) and the second non-linear convolution (conv.2).

## 6.2 Proposed HW blocks for Intra-Tile Execution



```
for (l=0; l<L; l++)
    for (i=0; i < Ho; i+= THo)
    for (j=0; j < Wo; j+= TWo)
        for (q=0; q < K; q++)
            for (p=0; p < K; p++)
                on-chip processing
                for (r'=0; r'<W2; r'++)
                for (s'=0; s'<W2; s'++)
                    for (i'=i; i'<min(Ho,i+THo); i'++)
                    for (j'=j; j'<min(Wo,j+TWo); j'++)
                        for (n'=0; n'<W1; n'++)
                        for (m'=0; m'<W1; m'++)
                            v[i',j',l][n',m',p] +=
                            x1[i'+r',j'+s',q] * u[n',m',p,l][r',s',q];

        v <- AF(v);
        v <- norm(v);
        for (p=0; p < K; p++)
            on-chip processing
            for (i'=i; i'<min(Ho,i+THo); i'++)
            for (j'=j; j'<min(Wo,j+TWo); j'++)
                for (n'=0; n'<W1; n'++)
                for (m'=0; m'<W1; m'++)
                    y[i',j',l] +=
                    x2[i'+n',j'+m',p] * v[i',j',l][n',m',p];
```

**Figure 6.3:** Loop tiling and interchange applied to the pseudocode of a NLC Layer

In figure 6.3, loop tiling and loop interchange techniques have been applied to the pseudocode of a NLC layer. This loop order allows to reuse the input feature map multiple times, reducing the accesses to the off-chip memory. Differently from the ideal and optimized inter-tile loops which use the tiling variables shown on table 5.5, this architecture sets the tiling variables $TpA$, $TL$, $TpB$ and $Tq$ equal to one, in order to reduce the architectural complexity as much as possible.

Regarding the intra-tile execution, in order to accelerate the execution of the first linear convolution and of the second non-linear convolution, it has been chosen to fully unroll the intra-tile loops 1A (Fig. 4.11), 4A (Fig. 4.14 ) and 1B (Fig. 4.17). Focusing on the first linear convolution, the fully unrolling of loop 1A allows to perform in parallel the product between the weights and the input pixels within the filter of size $W_2 \times W_2$, while the fully unrolling of loop 4A allows to reuse the input pixels to compute in parallel the $W_1 \times W_1$ un-normalized space-variant weights. Focusing on the second non-linear convolution, with intra-tile loop 1B fully unrolled, the product between the normalized space-variant weights and the input pixels of the same input channel is performed in parallel. As consequence the PEs required to perform the MAC operation needed to execute conv.1 and conv.2 can be derived by the equations 4.3.4 and 4.3.5 as:

$$\begin{aligned}\#MAC\_conv.1 = P_r \times P_s \times P_{mA} \times P_{nA} = \\ 3 \times 3 \times 3 \times 3 = 81\end{aligned} \tag{6.2.1}$$

$$\#MAC\_conv.2 = P_{mB} \times P_{nB} = 3 \times 3 = 9 \tag{6.2.2}$$

Further, since the first convolution and the second convolution are not performed in parallel due to the data dependencies, the number of of MAC units can be reused. As consequence, the number of required MAC unit is equal to the maximum between the number of MAC units needed to perform conv.1 and conv.2:

$$\#MAC = MAX(\#MAC\_conv.1, \#MAC\_conv.2) = 81 \tag{6.2.3}$$

### 6.2.1 PE-Array Design (Conv.1 )

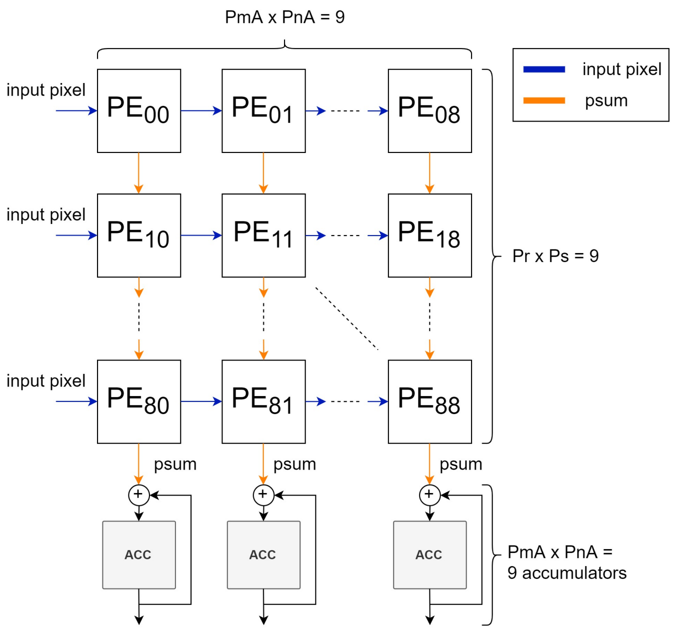


**Figure 6.4:** PE-Array $9 \times 9$ for conv.1

Relating on the first linear convolution, the portion of input pixels that is stored on-chip is bring to the PE-Array to perform the convolution operation. At each clock cycle, the input pixels must be reschedule in a proper order to compute the required task correctly. While, the 81 fixed weights are stored and kept fixed in the registers inside the PEs. Looking at the PE-Array, each column of the array performs the product between the fixed weights and the input pixels of one input channel. So, each column computes one of the $W_1 \times W_1$ un-normalized s-v weights. Since loop 1A and 5A are fully unrolled, the PE array has $W_2 \times W_2$ rows and $W_1 \times W_1$ columns.

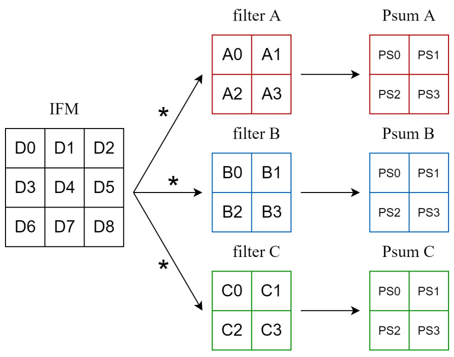


**Figure 6.5:** Example of convolution with multiple output channels

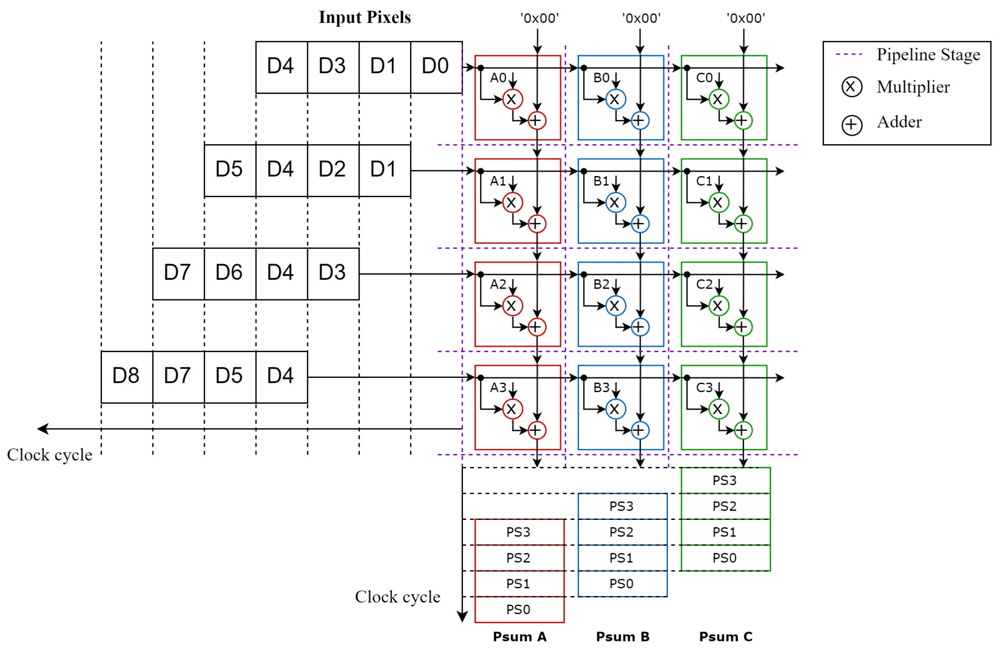


**Figure 6.6:** Example of convolution with a PE-Array $4 \times 3$

To better understand the data flow of the PE-Array in conv.1, in figure 6.5 it is shown a straightforward case. In this case, only for graphic reason, they have been used 3 filters of size $2 \times 2$, instead of the 9 filters of size $3 \times 3$, and an input map of size $3 \times 3$. In figure 6.6 at each clock cycle, looking from the left to the right, the input pixels are send and propagated along the rows of the PE array, while, the partial sums of each output features are moved downward.

Each of the red, blue and green squares represents one PE. Horizontal and vertical pipeline stages are added to reduce the critical path. In figure 6.7, it is shown the schematic of the processing element used to perform the MAC operation. Looking at the schematic:

- the multiplier MPL performs the product between the input pixel IN_PXL and the weights W_IN

- the product OUT_MPL is added with the input partial sum PS_IN and the generated result PS_OUT is then propagated to the next processing element.

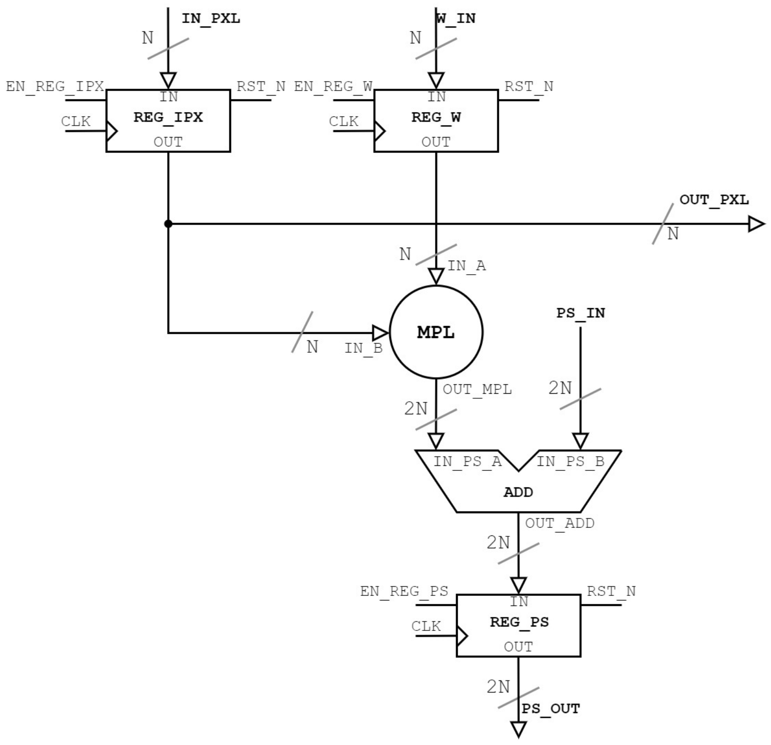


**Figure 6.7:** Processing element of PE-Array

### 6.2.2 S-V Weights Accumulator Design (Conv.1 )

Referring on figure 6.6, the partial sums that are computed must be accumulated to perform the un-normalized weights. Since the PE-Array is composed of $W_1 \times W_1$ columns, 9 accumulators of size $THo \times TWo \times K$ are needed.

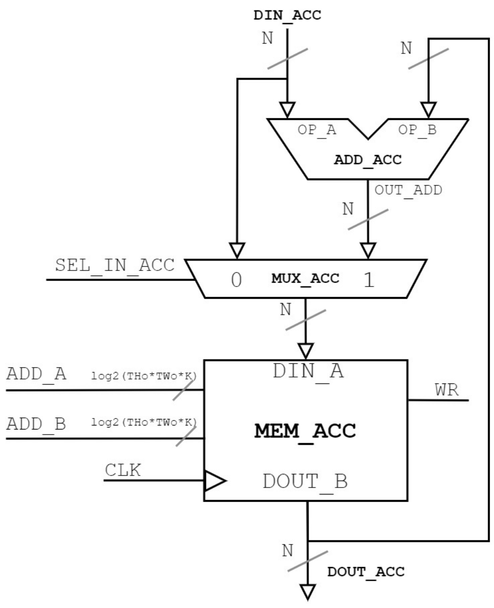


**Figure 6.8:** Accumulator needed to compute and store the s-v weights

Each accumulator is composed of an adder (ADD_ACC) which upgrades the partial sum stored in the on-chip memory (MEM_ACC). A multiplexer (MUX_ACC) is needed to save the correct first partial sum associated to the un-normalized spatial variant weight. Since the $TWo \times THo$ partial sums are provided by the PE-Array consecutively, the base address needed to store the data is equal to product $TWo \cdot THo \cdot p$, with $0 \leq p < K$.

### 6.2.3 PE-Array Design (Conv.2)

Relating on the second non-linear convolution, the portion of input pixels that is stored on-chip is bring to the PE-array to perform the non-linear convolution operation. At each clock cycle, the input pixels must be reschedule in a proper order in the same way as for the first linear convolution. However, differently from the previous case, no further weight is loaded to compute the $TWo \times THo$ output pixels, since the $TWo \times THo \times K$ normalized weights are already stored on-chip.

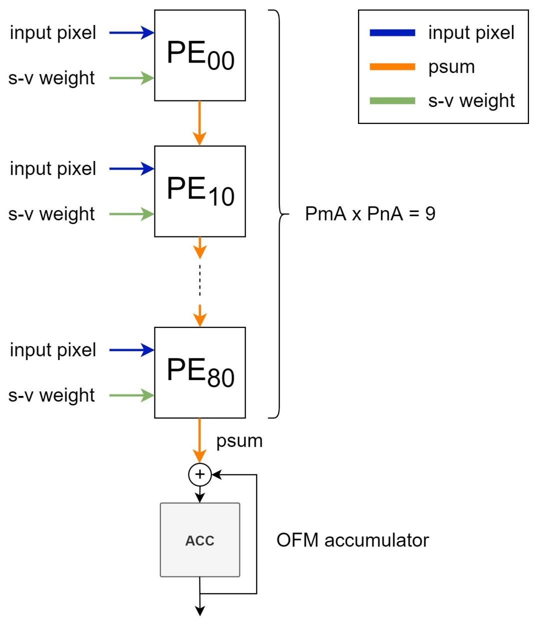


**Figure 6.9:** PE-Array $9 \times 1$ for conv.2

Looking at the PE-Array in figure 6.9, it is composed of $W_1 \times W_1$ rows and one column, since it is performed only one output feature. Further, in comparison to the previous case, the weights are not kept fixed for each clock cycle during the execution of the task. This is due to the fact that this PE-Array performs a non-linear convolution operation.

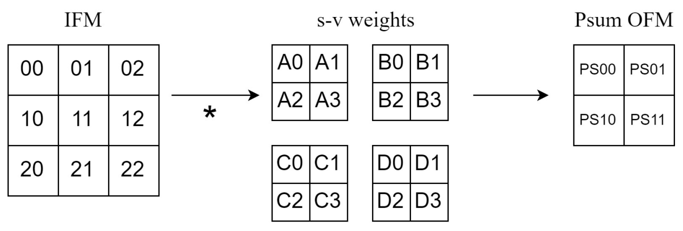


**Figure 6.10:** Example of non-linear convolution

To better understand the data flow of this PE-Array, like for the previous case, in figure 6.10 it is shown a straightforward case. In this example, only for graphic reason, it has been used a filter of size $2 \times 2$, instead of a filter of size $3 \times 3$, and a input map of size $3 \times 3$. In figure 6.11, at each clock cycle, looking from the left to the right, the input pixel and the associated s-v weights are send to the relative PE, while, the partial sum of each output pixel is moved downward. As mentioned in the previous sections, since the computational core of the PE is equal both for conv.1 and conv.2, part of the PE-Array $9 \times 9$ used for the linear convolution can be shared to compute the non-linear convolution.

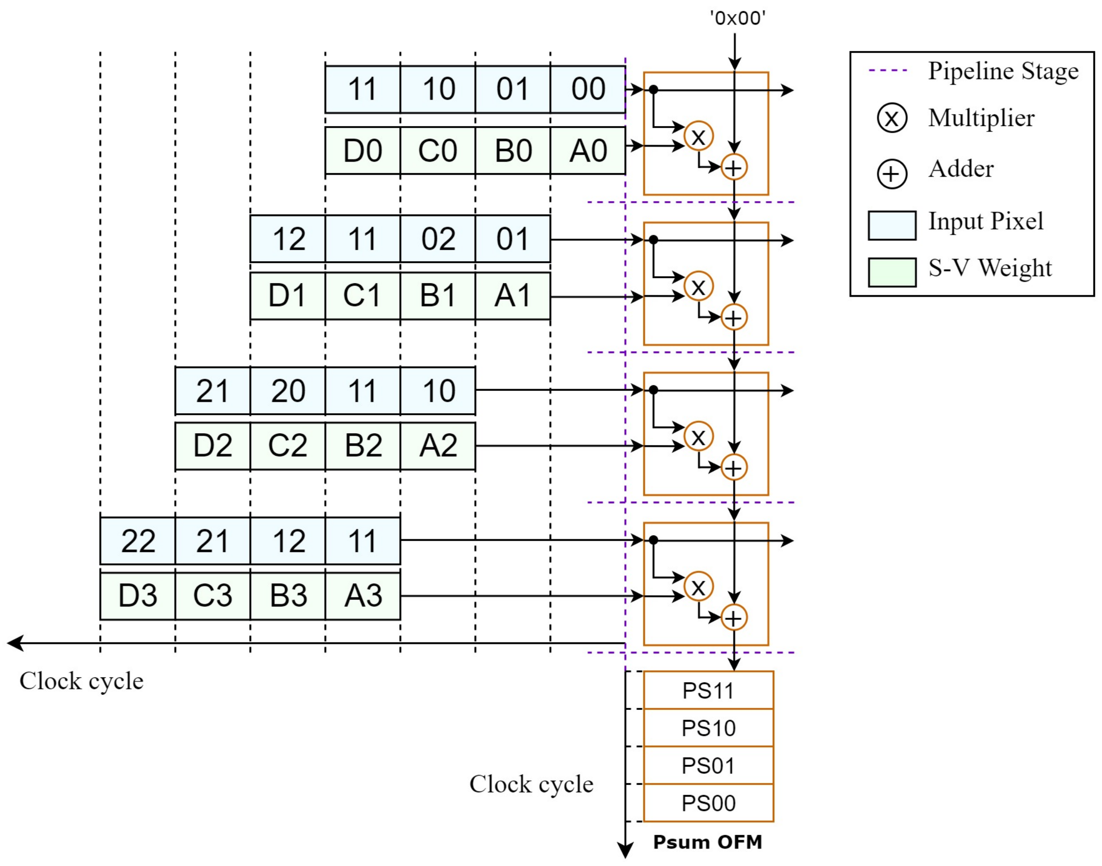


**Figure 6.11:** Example of non-linear convolution with PE-Array $4 \times 1$

### 6.2.4 OFM Accumulator

In order to accumulate all the partial sums needed to compute the final output pixels, an accumulator of size $TWo \times THo$ is needed. The schematic is the same as the previous described in subsection 6.2.2. In fact, the adder (ADD_ACC) upgrades the partial sum stored in the on-chip memory (MEM_ACC), while the multiplexer (MUX_ACC) allows to correctly save the first valid data.

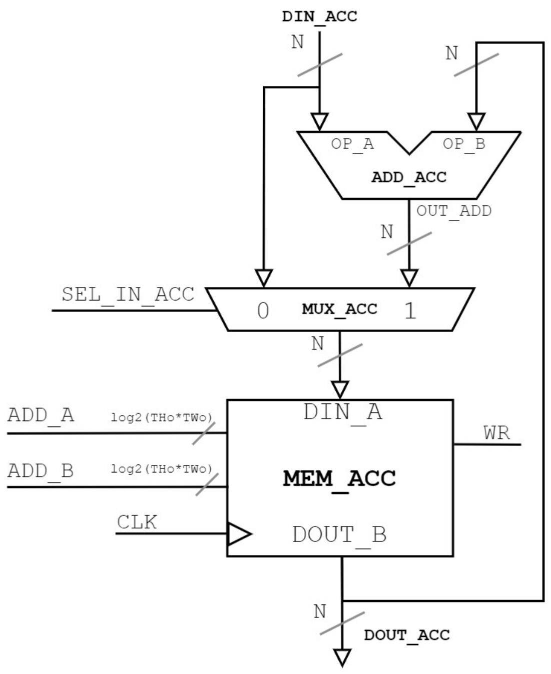


**Figure 6.12:** Accumulator needed to compute the $TWo \times THo$ output pixels

## 6.3 Designed NLC Accelerator block Simulation

To test the correct behaviour of the PE-Array both for the first linear convolution and for the second non-linear convolution, the developed logic blocks have been simulated with *ModelSim*, which is a HDL simulator environment.

### 6.3.1 First linear convolution Simulation

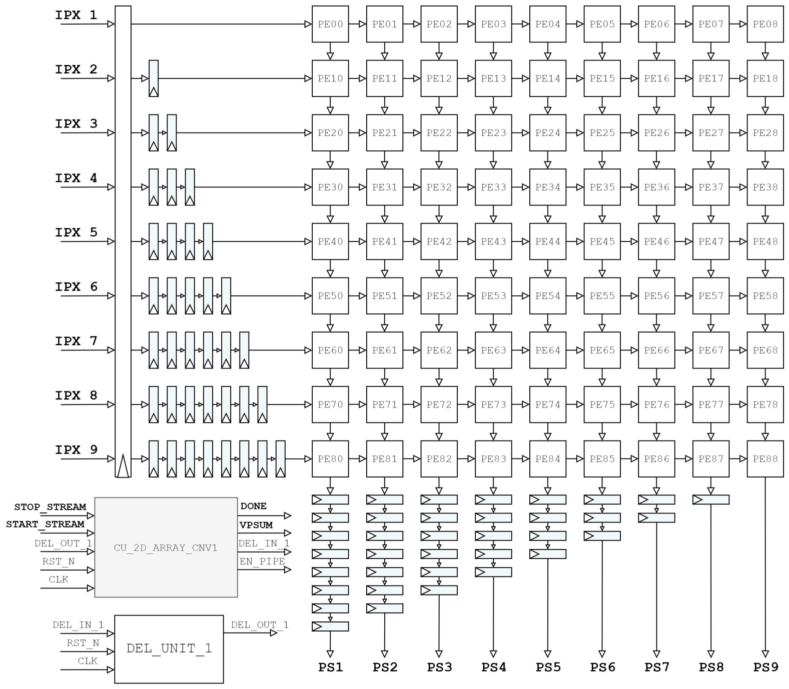

IPX 1
IPX 2
IPX 3
IPX 4
IPX 5
IPX 6
IPX 7
IPX 8
IPX 9
PE00 PE01 PE02 PE03 PE04 PE05 PE06 PE07 PE08
PE10 PE11 PE12 PE13 PE14 PE15 PE16 PE17 PE18
PE20 PE21 PE22 PE23 PE24 PE25 PE26 PE27 PE28
PE30 PE31 PE32 PE33 PE34 PE35 PE36 PE37 PE38
PE40 PE41 PE42 PE43 PE44 PE45 PE46 PE47 PE48
PE50 PE51 PE52 PE53 PE54 PE55 PE56 PE57 PE58
PE60 PE61 PE62 PE63 PE64 PE65 PE66 PE67 PE68
PE70 PE71 PE72 PE73 PE74 PE75 PE76 PE77 PE78
PE80 PE81 PE82 PE83 PE84 PE85 PE86 PE87 PE88
STOP_STREAM
START_STREAM
DEL_OUT_1
RST_N
CLK
CU_2D_ARRAY_CNV1
DONE
VPSUM
DEL_IN_1
EN_PIPE
DEL_IN_1
RST_N
CLK
DEL_UNIT_1
DEL_OUT_1
PS1 PS2 PS3 PS4 PS5 PS6 PS7 PS8 PS9

**Figure 6.13:** Architecture of the PE-Array used to test the intra-tile loops of conv.1

To test the intra-tile loops of conv.1, it has been developed the architecture show in figure 6.13. As it can been noticed some registers are added to the input and to the output of the PE-Array in order to ensure a correct data-flow. In fact, as it was discussed with the example shown in figure 6.6, the input pixels must be progressive delayed. Similarly, since the output psum are not aligned, they have been added a proper number of registers under each PEs column. Clearly, to store the data performed, the PS1.. PS9 signals must be connected to the *S-V Weights Accumulator*. To manage the correct data-flow and synchronize the proposed block with the execution of the other HW blocks implemented in the HW Accelerator, it has been developed the control unit *CU_2D_ARRAY_CNV1* (Figure 6.14).

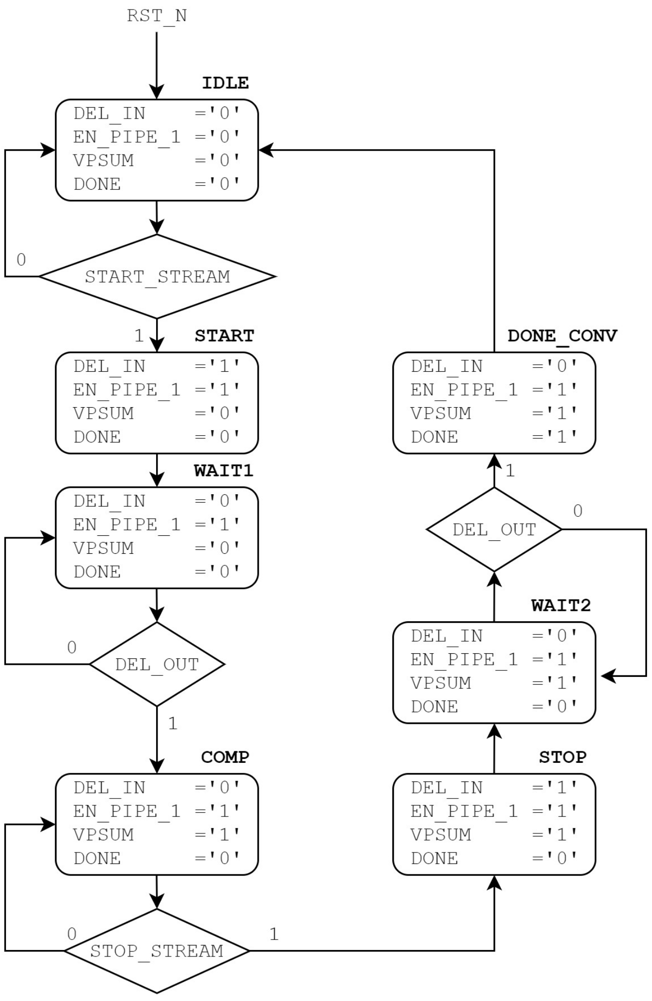


**Figure 6.14:** CU_2D_ARRAY_CNV1 ASM chart

To correctly start the convolutional operation, an input signal START_STREAM indicates that the input pixels are ready to be processed. After that the control unit changes its state from IDLE to START. In this state the internal pipe register of the PE-Array and the delay unit are enabled. In fact, the delay unit DEL_UNIT_1 is needed to proper synchronize the control unit to enable the data-out valid signal VPSUM in correspondence of the correct output partial sums performed. In COMP the PE-Array performs the convolution operation until the signal STOP_STREAM is asserted. After that, the control unit changes its state from COMP to STOP. where the delay unit is enabled once more. As before, the delay unit is used to proper enable the output signal VPSUM for the remaining output data. Finally, in correspondence of the last valid partial sums the control unit change its state from WAIT2 to DONE_CONV, where is asserted the DONE signal. In figure 6.15

it is shown a simulation of the proposed architecture. In the testbench, the clock has been set to 100 MHz. For this simulation, it has been chosen an input map of 18 × 18 pixels.

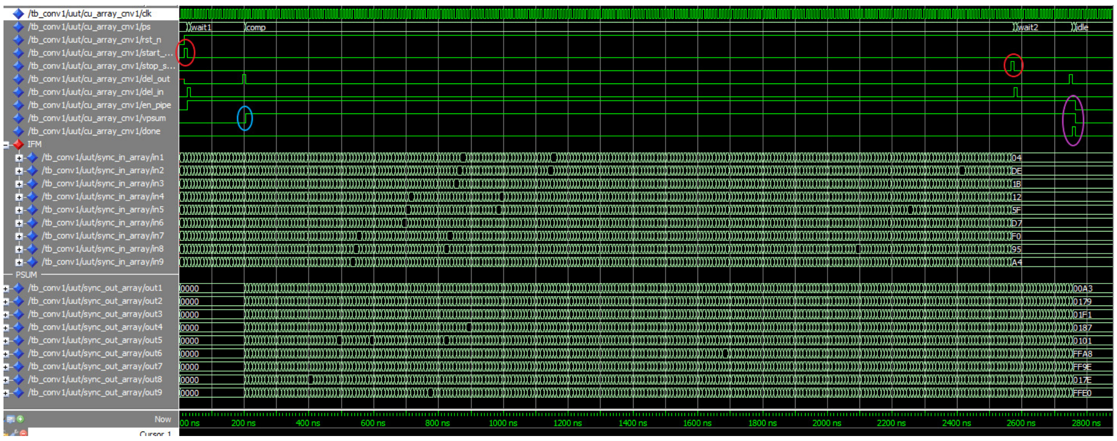

/tb_conv1/uut/cu_array_cnv1/clk
/tb_conv1/uut/cu_array_cnv1/ps
/tb_conv1/uut/cu_array_cnv1/rst_n
/tb_conv1/uut/cu_array_cnv1/start_...
/tb_conv1/uut/cu_array_cnv1/stop_s...
/tb_conv1/uut/cu_array_cnv1/del_out
/tb_conv1/uut/cu_array_cnv1/del_in
/tb_conv1/uut/cu_array_cnv1/en_pipe
/tb_conv1/uut/cu_array_cnv1/vpsum
/tb_conv1/uut/cu_array_cnv1/done
IFM
/tb_conv1/uut/sync_in_array/in1
/tb_conv1/uut/sync_in_array/in2
/tb_conv1/uut/sync_in_array/in3
/tb_conv1/uut/sync_in_array/in4
/tb_conv1/uut/sync_in_array/in5
/tb_conv1/uut/sync_in_array/in6
/tb_conv1/uut/sync_in_array/in7
/tb_conv1/uut/sync_in_array/in8
/tb_conv1/uut/sync_in_array/in9
PSUM
/tb_conv1/uut/sync_out_array/out1
/tb_conv1/uut/sync_out_array/out2
/tb_conv1/uut/sync_out_array/out3
/tb_conv1/uut/sync_out_array/out4
/tb_conv1/uut/sync_out_array/out5
/tb_conv1/uut/sync_out_array/out6
/tb_conv1/uut/sync_out_array/out7
/tb_conv1/uut/sync_out_array/out8
/tb_conv1/uut/sync_out_array/out9
wait1
comp
wait2
idle
04
DE
1B
12
5F
D7
F0
95
A4
0000
00A3
0179
01F1
0187
0101
FFA8
FF9E
017E
FFE0
Now
Cursor 1
00 ns
200 ns
400 ns
600 ns
800 ns
1000 ns
1200 ns
1400 ns
1600 ns
1800 ns
2000 ns
2200 ns
2400 ns
2600 ns
2800 ns

**Figure 6.15:** ModelSim simulation of Conv.1

Looking from the left to the right, when START_STREAM signal is asserted (red circle) the first 9 input pixels are available (IN1, IN2.. IN9). After some clock cycles the first $PmA \times PnA = 9$ partial sums are computed and the VPSUM is asserted (blue circle). After that the control unit remains in the COMP state until the last input data are received. Accordingly, in correspondence of the last 9 input pixels the input signal STOP_STREAM is asserted (red circle on the right) and the last data are elaborated. After some clock cycles in correspondence of the last 9 partial sums the DONE signal is asserted (violet circle). After that the control unit changes its state from DONE_CONV to IDLE.

### 6.3.2 Second non-linear convolution Simulation

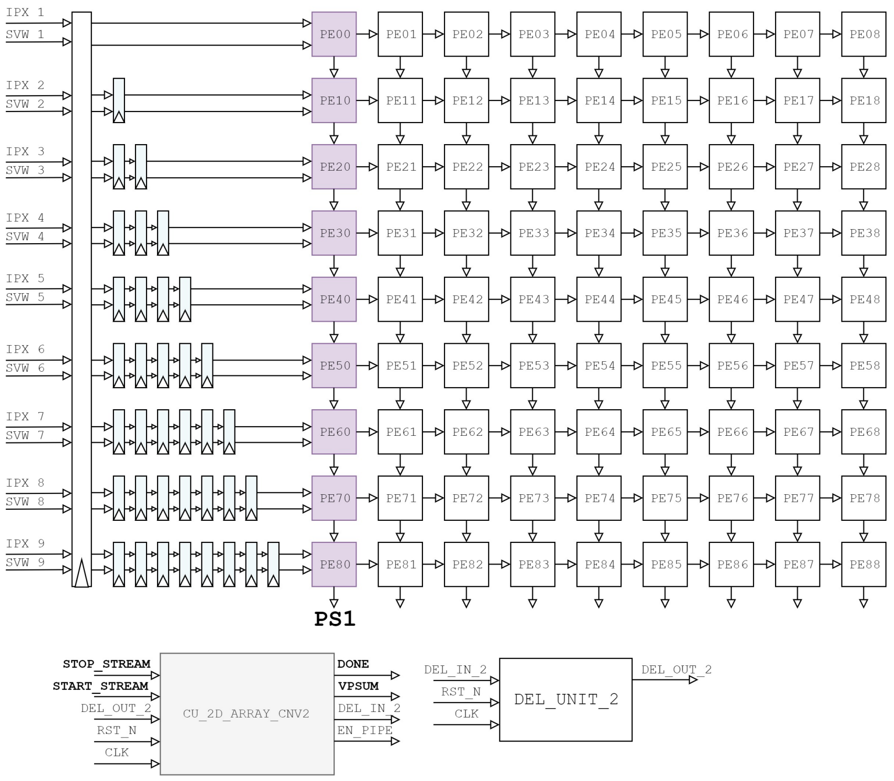


**Figure 6.16:** Architecture of the PE-Array used to test the intra-tile loops of conv.2

To test the intra-tile loops of the non-linear convolution, it has been developed the architecture shown in figure 6.16. Since conv.1 and conv.2 share the same PE-Array, the first column is used to perform the $PnB \times PmB = 9$ MAC operation between the input pixels and the normalized s-v weights. Differently from the previous case where the weights were fixed during the execution of the task, in this case, since the proposed non-linear convolution uses space-variant parameters, at each clock cycles $PnB \times PmB = 9$ s-v weights are updated (SVW_1, SVW_2 ... SVW_9) together with the relative input pixels (IPX_1, IPX_2 ... IPX_9). This time, no output registers are needed to align the output partial sums, because only one column of the array is used. However a delay unit DEL_UNIT_2 is still used, because the control unit CU_2D_ARRAY_CNV2 needs a synchronization signal to indicate the correct valid partial sums (VPSUM) referred to the output pixels. Clearly, to store the data performed, the PS1 signal must be connected to the *OFM Accumulator*.

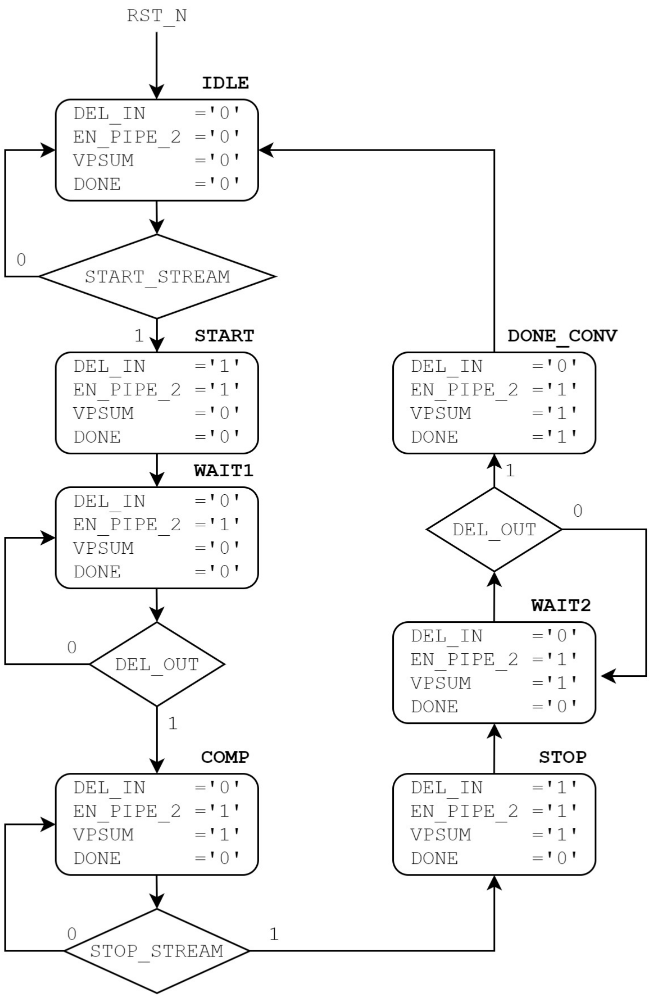


**Figure 6.17:** CU_2D_ARRAY_CNV2 ASM chart

The proposed control unit in figure 6.17 is the same of the one proposed for the first linear convolution. However, the only difference is related on the different delay unit used to synchronize the execution.
In figure 6.18, it is shown a simulation of the proposed architecture. In the testbench, the clock has been set to 100 MHz. As for the previous simulation, it has been chosen an input map of 18 × 18 pixels. Looking from the left to the right, when START_STREAM signal is asserted (red circle) the first 9 input pixels (IN1, IN2 ... IN9) and the 9 s-v weights (SVW1, SVW2 ... SVW9) are available. As it can be noticed, differently from the simulation of the first linear convolution, 9 weights are upgraded at each clock cycles, because these have a spatial dependence. After some clock cycles the first partial sum (cyan signal named out1) is available and the signal VPSUM is asserted (orange circle). After that the control unit remains in the

COMP state until the last input data are received. As for the previous simulation the control unit changes its state in the same manner. In fact, in correspondence of the last 9 input pixels and 9 s-v weights the input signal STOP_STREAM is asserted (red circle on the right) and the remaining data are elaborated. After some clock cycles in correspondence of the last partial sum the DONE signal is asserted (blue circle). After that the control unit changes its state from DONE_CONV to IDLE.

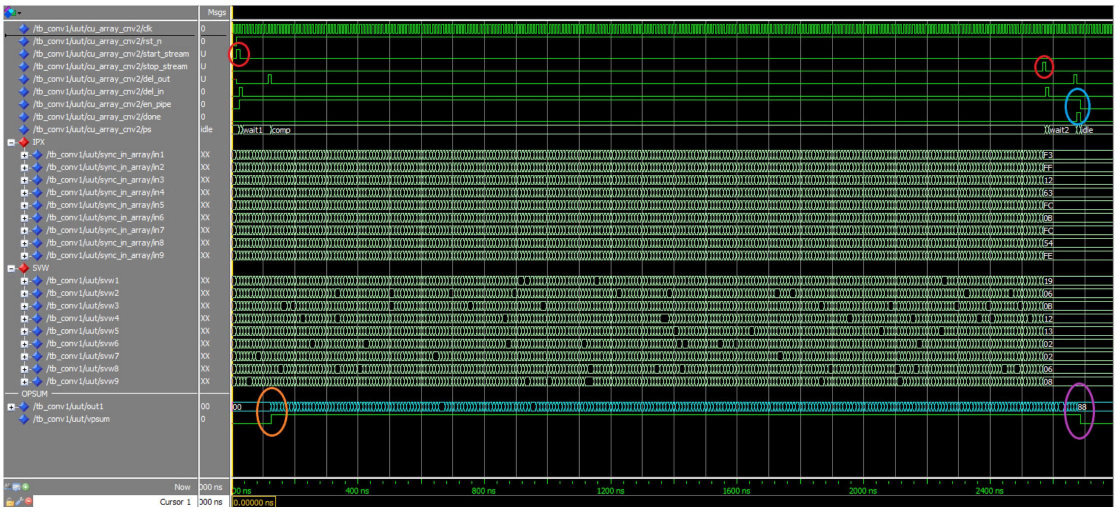

Msgs
/tb_conv1/uut/cu_array_cnv2/clk
/tb_conv1/uut/cu_array_cnv2/rst_n
/tb_conv1/uut/cu_array_cnv2/start_stream
/tb_conv1/uut/cu_array_cnv2/stop_stream
/tb_conv1/uut/cu_array_cnv2/del_out
/tb_conv1/uut/cu_array_cnv2/del_in
/tb_conv1/uut/cu_array_cnv2/en_pipe
/tb_conv1/uut/cu_array_cnv2/done
/tb_conv1/uut/cu_array_cnv2/ps
IPX
/tb_conv1/uut/sync_in_array/in1
/tb_conv1/uut/sync_in_array/in2
/tb_conv1/uut/sync_in_array/in3
/tb_conv1/uut/sync_in_array/in4
/tb_conv1/uut/sync_in_array/in5
/tb_conv1/uut/sync_in_array/in6
/tb_conv1/uut/sync_in_array/in7
/tb_conv1/uut/sync_in_array/in8
/tb_conv1/uut/sync_in_array/in9
SVW
/tb_conv1/uut/svw1
/tb_conv1/uut/svw2
/tb_conv1/uut/svw3
/tb_conv1/uut/svw4
/tb_conv1/uut/svw5
/tb_conv1/uut/svw6
/tb_conv1/uut/svw7
/tb_conv1/uut/svw8
/tb_conv1/uut/svw9
OPSUM
/tb_conv1/uut/out1
/tb_conv1/uut/vpsum
wait1
comp
wait2
idle
Now
Cursor 1
400 ns
800 ns
1200 ns
1600 ns
2000 ns
2400 ns

**Figure 6.18:** ModelSim simulation of Conv.2

# Chapter 7

# Conclusions

In this thesis work, differently from standard convolutional layer, a new kind of non-linear convolutional layer has been used. The aim of this thesis has been addressed on the study of the possible optimized HW architectures of a NLC layer that could be implemented into FPGAs. In particular, in the first part of the thesis, starting from the pseudocode of the NLC layer, different loop optimization techniques have been investigated. These, in fact, allow to optimize both the time of execution and the data transaction between off-chip memory and on-chip memory. So, as first step it has been developed an analytical model of the layer in order to explore the principal design variables such as the number of DSP slices used, the latency of the system, the size of the on-chip memory and the number of accesses to the off-chip memory in function of the different loop variables. In particular, a python script has been developed to investigate among the huge space of solutions which are the most suitable design solutions that minimized both the number of accesses to the off-chip memory and the size of the on-chip memory. In the last part, a proposed HW architecture of the NLC layer has been explained. In particular, most of the effort has been addressed on the design of the PE-Array which is the computational core of the HW accelerator.
In this thesis a first study of the possible HW implementation has been done. however, as future work, one of the possible step could be to implement the remaining modules and test the overall system on a target FPGA in order to test the real performances in comparison of the theoretical ones. In particular, great care must be addressed on the HW module which performs the normalization of the input dependent parameters.

# Appendix A

# A Python script for the Design Space Exploration of a NLC layer

```
''' import libraries'''
from gen_seq_loop import LoopCombNLC
from LoopInfo import TotAccOffChipMEM, OnChipMem
import csv

''' NCL layer parameters'''
Ho=512 #height of output feature
Wo=512 #width of output feature
W1=3 #kernel size conv2
W2=3 #kernel size conv1
K=3  #input channels
L=6  #output channels

''' Datawidth'''
data_in_datawidth=8
fw_datawidth=16
sv_datawidth=16
out_datawidth=8

'''generation of the possible order
of execution of the inter-tile loops'''
loop_comb= LoopCombNLC()

out=[]
for index in range(0,len(loop_comb),1):
    for TL in range(1,L+1,1):
        for TmA in range(1,W1+1,1):
            for TnA in range(1,W1+1,1):
                for THo in range(1,Ho+1,16):
                    for TWo in range(1,Wo+1,16):
                        for TpA in range(1,K+1,1):
                            for Tq in range(1,K+1,1):
                                for Tr in range(1,W2+1,1):
                                    for Ts in range(1,W2+1,1):
                                        for TpB in range(1,K+1,1):\
                TWi=TWo+W1-1
                THi=THo+W1-1
                acc= TotAccOffChipMEM(loop_comb[index], L, TL, W1, TmA, \
                    TnA, Ho, Wo, THo, TWo, K, TpA, Tq, W2, Tr, Ts, TpB)
                size=OnChipMem(TWi,THi,Tq,TpB,data_in_datawidth,\
                      Tr,Ts, TnA, TmA, TpA, TL, fw_datawidth,\
                      loop_comb[index],Ho,Wo,W1,K,TWo,THo,sv_datawidth,\
                      out_datawidth )
                size=size/(8*10**6)
                x_axis=size
                y_axis=format(acc, ',.1E')
                coordinate=[x_axis,y_axis,Tr*Ts,Tq,THo*TWo, TmA*TnA,TpA,
    TpB,TL]
                out.append(coordinate)


with open('loop_comb.csv', 'w') as f:
    write = csv.writer(f,delimiter=';',lineterminator='\n')
    Details=['MEM_SIZE',' MEM_ACC','TrxTs', 'Tq','TWoxTHo','TmAxTnA',
    'TpA','TpB','TL']
    write.writerow(Details)
    write.writerows(out)
```

```
''' import libraries'''
import math

'''ON-CHIP MEM SIZE'''
def OnChipMem(TWi,THi,Tq,TpB,data_in_datawidth,\
               Tr,Ts, TnA, TmA, TpA, TL, fw_datawidth,\
               loop_order,Ho,Wo,W1,K,TWo,THo,sv_datawidth,\
               out_datawidth ):
    '''
    this function computes the on-chip memory required
    '''
    out = I_PXL(TWi,THi,Tq,TpB,data_in_datawidth) + \
   F_W(Tr, Ts, Tq, TnA, TmA, TpA, TL, fw_datawidth) + \
    PsumSize(loop_order,Ho,Wo,K,W1,TWo,THo,TL,sv_datawidth) +\
    OutBuffer(loop_order,Ho,Wo,THo,TWo,TL,out_datawidth)

    return out


def I_PXL(TWi,THi,Tq,TpB,data_in_datawidth):
    buff1=TWi * THi * Tq *data_in_datawidth
    buff2=TWi * THi * TpB *data_in_datawidth
    return _max_(buff1, buff2)

def F_W(Tr, Ts, Tq, TnA, TmA, TpA, TL, fw_datawidth):
    return Tr * Ts* Tq* TnA * TmA * TpA * TL * fw_datawidth

def PsumSize(loop_order,Ho,Wo,K,W1,TWo,THo,TL,sv_datawidth):
    if loop_order[0][0] == 'L3A' and loop_order[1][0] == 'L3B':
        out_size = THo*TWo*K*W1*W1*TL*sv_datawidth
    else:
        out_size = Ho*Wo*K*W1*W1*TL*sv_datawidth
    return out_size

def OutBuffer(loop_order,Ho,Wo,THo,TWo,TL,out_datawidth):
    if loop_order[1][0]=='L3B' and loop_order[0][0] == 'L3A':
        out_size = THo * TWo * TL * out_datawidth
    else:
        out_size = Wo*Ho*TL* out_datawidth
    return  out_size

'''#ACCESSES TO OFF-CHIP MEM'''
def TotAccOffChipMEM(loop_order,L,TL,W1,TmA,TnA,Ho,Wo,THo,TWo,K,TpA,
    Tq,W2,Tr,Ts,TpB):
    '''
    this function computes the number of accesses to the off-chip
    memory
    '''
    pass
    tot_acc=AccOffChipFW(loop_order,L,TL,W1,TmA,TnA,Ho,Wo,THo,TWo,K,
    TpA,Tq,W2,Tr,Ts)+\
    AccOffChipPX(loop_order,L,TL,W1,TmA,TnA,Ho,Wo,THo,TWo,K,TpA,TpB,
    Tq,W2,Tr,Ts)
    return tot_acc


def AccOffChipFW(loop_order,L,TL,W1,TmA,TnA,Ho,Wo,THo,TWo,K,TpA,Tq,W2
    ,Tr,Ts):
    p={'L6':math.ceil(L/TL),'L5A': math.ceil(K/TpA),'L4A':math.ceil(
    W1/TnA)*math.ceil(W1/TmA),'L3A': \
        math.ceil(Ho/THo)*math.ceil(Wo/TWo),'L2A':math.ceil(K/Tq),'L1A
    ':math.ceil(W2/Tr)*math.ceil(W2/Ts)}
    Acc_Fw = p['L6']*p['L5A']*p['L4A']*p['L2A']*p['L1A']
    flag_L5A = False
    flag_L4A = False
    flag_L2A = False
    flag_L1A = False
    for loop in loop_order[0]:
        if flag_L5A == False or flag_L4A == False or flag_L2A == False or flag_L1A == False :
            if loop == 'L5A':
                flag_L5A = True
            elif loop == 'L4A':
                flag_L4A = True
            elif loop == 'L2A':
                flag_L2A = True
            elif loop == 'L1A':
                flag_L1A = True
            else:
                Acc_Fw=Acc_Fw * p[loop]
    return Acc_Fw

def AccOffChipPX(loop_order,L,TL,W1,TmA,TnA,Ho,Wo,THo,TWo,K,TpA,TpB,Tq,W2,Tr,Ts):
    p={'L6':math.ceil(L/TL),'L5A': math.ceil(K/TpA),'L4A':math.ceil(W1/TnA)*math.ceil(W1/TmA),'L3A': \
        math.ceil(Ho/THo)*math.ceil(Wo/TWo),'L2A':math.ceil(K/Tq),'L1A':math.ceil(W2/Tr)*math.ceil(W2/Ts)}
    Acc_PX_conv1 = p['L6']*p['L3A']*p['L2A']
    flag_L3A = False
    flag_L2A = False
    for loop in loop_order[0]:
        if flag_L3A == False or flag_L2A == False:
            if loop == 'L2A':
                flag_L2A = True
            elif loop == 'L3A':
                flag_L3A = True
            else:
                Acc_PX_conv1=Acc_PX_conv1 * p[loop]

    Acc_PX_conv2= math.ceil(L/TL)*math.ceil(Ho/THo)*math.ceil(Wo/TWo)*math.ceil(K/TpB)
    Acc_PX = Acc_PX_conv1 + Acc_PX_conv2
    return Acc_PX

def _max_(a,b):
    if a>b:
        out_=a
    else:
        out_=b
    return out_
```

```
''' import libraries'''
import csv

def CheckDiff(G):
    '''
    This function checks if the items of the array
     are all different the one with each other
    '''
    diff = False
    equal = 0
    for fir in G:
        for sec in G:
            if fir == sec:
                equal = equal + 1
    if equal == len(G):
        diff =  True
    return diff

def LoopOrder(Loops,n=-1,index=[],out_=[]):
    '''
    This function creates a list of the possible
    combination of the loops listed in the input list Loops
    '''
    if n == -1:
        out_=[]
        n = len(Loops)
        index = [None]*n
    if n > 0:
        for index[n-1] in range(len(Loops)):
            LoopOrder(Loops,n-1,index,out_)
    else:
        if CheckDiff(index):
            out=[]

            out.clear()
            for row in index:
                out.append(Loops[row])
            out_.append(out)
    return out_

def LoopCombNLC():
    '''
    This function creates a list of the possible
    combination of the loops listed in Nest_loop_A
    and Nest_Loop_B
    '''
    Nest_Loop_A=['L5A','L4A','L3A','L2A','L1A']
    Nest_Loop_B=['L3B','L2B','L1B']
    Loop_Comb_A=LoopOrder(Nest_Loop_A)
    Loop_Comb_B=LoopOrder(Nest_Loop_B)
    Tot_Seq_Loop=[]
    for Loop_Seq_A in Loop_Comb_A:
        for Loop_Seq_B in Loop_Comb_B:
            Loop_AB= [Loop_Seq_A, Loop_Seq_B]
            Tot_Seq_Loop.append(Loop_AB)
    return Tot_Seq_Loop
```

# Bibliography

[1] Stefano Marsi, Jhilik Bhattacharya, Romina Molina, and Giovanni Ramponi. «A Non-Linear Convolution Network for Image Processing». In: *Electronics* 10.2 (2021). ISSN: 2079-9292. DOI: 10.3390/electronics10020201. URL: https://www.mdpi.com/2079-9292/10/2/201 (cit. on pp. 3, 9, 15).

[2] Laughsinthestocks. *A plot of the rectified linear activation function.* URL: https://en.wikipedia.org/wiki/Activation_function#/media/File:Activation_rectified_linear.svg (cit. on p. 8).

[3] Laughsinthestocks. *A plot of the logistic activation function.* URL: https://en.wikipedia.org/wiki/Activation_function#/media/File:Activation_logistic.svg (cit. on p. 8).

[4] Laughsinthestocks. *A plot of the tanh activation function.* URL: https://en.wikipedia.org/wiki/Activation_function#/media/File:Activation_tanh.svg (cit. on p. 8).

[5] Georgios Zoumpourlis, Alexandros Doumanoglou, Nicholas Vretos, and Petros Daras. «Non-linear Convolution Filters for CNN-Based Learning». In: *2017 IEEE International Conference on Computer Vision (ICCV).* 2017, pp. 4771–4779. DOI: 10.1109/ICCV.2017.510 (cit. on p. 9).

[6] URL: https://www.run.ai/wp-content/uploads/2021/01/diagrams-Dec-2020_F-1536x593.png (cit. on p. 15).

[7] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. «Deep Residual Learning for Image Recognition». In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR).* 2016, pp. 770–778. DOI: 10.1109/CVPR.2016.90 (cit. on p. 16).

[8] Yulun Zhang, Yapeng Tian, Yu Kong, Bineng Zhong, and Yun Fu. *Residual Dense Network for Image Super-Resolution.* 2018. arXiv: 1802.08797 [cs.CV] (cit. on p. 16).

[9] Yuqing Liu, Xinfeng Zhang, Shanshe Wang, Siwei Ma, and Wen Gao. *Progressive Multi-Scale Residual Network for Single Image Super-Resolution.* 2020. arXiv: 2007.09552 [eess.IV] (cit. on p. 16).

[10] URL: https://docs.microsoft.com/it-it/azure/machine-learning/media/how-to-deploy-fpga-web-service/azure-machine-learning-fpga-comparison.png (cit. on p. 20).

[11] URL: https://www.xilinx.com/support/documentation/sw_manuals/ug1207-sdaccel-performance-optimization.pdf (cit. on p. 23).

[12] Xiaowei Xu, Yukun Ding, Sharon Xiaobo Hu, Michael Niemier, Jason Cong, Yu Hu, and Yiyu Shi. «Scaling for edge inference of deep neural networks». In: *Nature Electronics* 1.4 (Apr. 2018), pp. 216–222. ISSN: 2520-1131. DOI: 10.1038/s41928-018-0059-3. URL: https://doi.org/10.1038/s41928-018-0059-3 (cit. on p. 28).

[13] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. «Optimizing FPGA-Based Accelerator Design for Deep Convolutional Neural Networks». In: *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '15. Monterey, California, USA: Association for Computing Machinery, 2015, pp. 161–170. ISBN: 9781450333153. DOI: 10.1145/2684746.2689060. URL: https://doi.org/10.1145/2684746.2689060 (cit. on p. 31).

[14] Yufei Ma, Yu Cao, Sarma Vrudhula, and Jae-sun Seo. «Optimizing the Convolution Operation to Accelerate Deep Neural Networks on FPGA». In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 26.7 (2018), pp. 1354–1367. DOI: 10.1109/TVLSI.2018.2815603 (cit. on pp. 31, 54).

[15] Shaoxia Fang, Shulin Zeng, and Yu Wang. «Optimizing CNN Accelerator With Improved Roofline Model». In: *2020 IEEE 33rd International System-on-Chip Conference (SOCC)*. 2020, pp. 90–95. DOI: 10.1109/SOCC49529.2020.9524754 (cit. on p. 42).