Chair of Integrated Systems
Technical University of Munich

Department of Electronics
and Telecommunications
Politecnico di Torino

POLITECNICO
DI TORINO

A Top-Down Compiler for Spatially Distributed Computation of Convolutional Neural Networks

Master thesis

Author: Alessandro Di Gioia

Advisor: Nael Fasfous
Supervisor: Prof. Dr. Ing. Walter Stechele and Prof. Dr. Ing. Maurizio Martina

Submission date: December 9, 2020

Abstract

Convolutional Neural Networks (CNNs) are widely used in modern AI systems because of their superior accuracy, but they require highly parallel hardware structures to compute their operations efficiently. The computational complexity of convolution and the high amount of data movement are two key factors to be considered when designing a hardware accelerator for CNN.
Systolic/Spatial architectures represent an interesting choice for accelerating the filtering operation of the convolution as they do not require stochastic data movement, but rather a regular and deterministic transfer and access among the different storage hierarchies. Moreover, this kind of architecture is more suitable for dataflow processing and is commonly employed in ASIC and FPGA-based designs [1].
A good scheduling can exploit the deterministic CNN execution: in most deployment scenarios the neural networks do not change frequently, so a predetermined schedule would save many resource and control logic at runtime.
A coherent and efficient Instruction Set Architecture (ISA) is required to maintain the flexibility of creating such schedules for different CNNs: it consists in some instructions that are necessary to cover the data movement and the logic required by a particular dataflow.
This thesis will show: a flexible Compiler able to map the CNN execution into the hardware, by providing the schedule of the computations under different configurations of the input parameters; an ISA to encode the schedule in FPGA platforms and in *Versal*, a cutting-edge ASIC of Xilinx suitable to accelerate CNN; the testing of the algorithms and instructions and an evaluation of the latency on such architectures.

Acknowledgements

Throughout my academic studies I have received a great deal of support and assistance from many people.

First, I would like to pay my special regards to professor Stechele and professor Martina for the amazing opportunity I have been given and for the support during this thesis work.

I wish to express my sincere appreciation to my advisor Nael Fasfous for his valuable and constructive suggestions during the planning and development of this research work. Without his persistent help, the goal of this project would not have been realized.

I would also like to thank HAPPI-Net team and my external advisor Manoj Rohit Vemparala for the support received and their willingness to share their findings and knowledge.

I would like to thank the TUM and BMW group for providing me this engaging master thesis and the Italdesign group for giving me this Erasmus opportunity.

I am deeply thankful to my parents their support and love received always, without them none of my accomplishments would have been possible. You are always there for me. In addition, I would like to thank my brother, my source of inspiration.

Finally, my gratitude goes to all my friends in Turin, my dear colleagues of Politecnico, and to my eternal friends of my lovely hometown.

Contents

List of Figures

List of Tables

Listings

1. Introduction

1.1. Motivation

Deep Neural Networks (DNNs) are strongly adopted in a large number of fields as speech recognition, image classification and segmentation [2], cancer detection [3] autonomous driving, [4] and they are currently the foundation for many modern Artificial Intelligence (AI) applications [1], but the great accuracy and performance of DNNs comes at the cost of high computation and storage complexity that must be considered when designing a hardware platform to accelerate the execution of such networks.
In particular, Convolutional Neural Networks (CNNs) are a common form of DNNs composed of high-dimensional convolutions [1]. The CNN inference represents the main limitation in hardware accelerator due to energy efficiency problems and low throughput. In fact, to provide more accurate results, the size of the CNNs can grow by adding more neural network layers, thus the huge number of operations and parameters contribute to make the computational challenge for general purpose processors more hard.
Therefore, some hardware platforms such as Field Programmable Gate Array (FPGA), Graphic Processing Unit (GPU) and Application Specific Integrated Circuit (ASIC) have been employed to improve the inference of CNNs [5], generally by making the execution parallel, reducing the latency and the power consumption according to the adopted type of accelerator.
Other important parameters, to meet power and cost constraints, are memory bandwidth and buffer sizing [6]. Basically they can be significantly reduced with a good data reuse pattern that is determined by the *dataflow schedule* of computation, i.e. the amount of operations allowed to be partitioned and scheduled for a specific computation and the data movement in the memory hierarchy [7]. Indeed in spatial architectures several levels of local memory hierarchy are introduced in order to improve energy efficiency by providing a low-cost data accesses [1]. The considerable number of possible dataflows and the related choices of hardware implementation have created a large *architecture design space* [8].

1.2. Contribution and Organization

This work deals the fine-level estimation of CNN, so the execution of CONV layers in dataflow architecture, focusing in the *mapping process* for a given input layer shape. According to previous accelerators like [9, 10, 11], the *Output Stationary* (OS) dataflow provides good performance, so it was a good starting point for implementing a Compiler able to translate the high level specification into micro-instructions at HW level.
Indeed, the first contribution of this work was a scheduling algorithm, in order to obtain a schedule of atomic instructions that follow a regular pattern within the spatial array. To find the best schedule, loop scheduling techniques have been applied in a certain way to support the OS dataflow.
An ISA, studied on purpose for such schedule, was employed to get the instructions to be dispatched in the spatial array in order to execute a CNN correctly.
Once that the Compiler has been created and verified with proper test, the other main contribution was the HW implementation for a FPGA-based design, which allowed to have an overview of the HW resources utilization for some configurations. Moreover, the latency trend for two CONV layer shapes (AlexNet and ResNet) was measured, obtaining a quasi-ideal behavior of the HW and giving high fidelity to the designed Modeling tool.
The last part of this research was about an investigation on the implementation of the same schedule of operations, obtained for the previous design, in *Versal* platform, namely a new ASIC with high performance for accelerating CNN.

Below, the organization of the next chapters is explained.
The chapter *Background* gives a brief introduction of DNN, with focus on CNN and the inference task, with its HW solutions and challenges.

In the chapter *Related work*, the different dataflows covered in the literature and the approaches to the design flow of CNN accelerators are analyzed.

The chapter *Mapping of CNN into spatial architecture* provides a wide overview of the HW components employed in dataflow architectures for executing CNN, with an explanation of the most common loop scheduling methods like loop tiling, loop unrolling and loop interchange.

In the chapter *Modeling tool*, the functional blocks of the design flow are described, which comprise the Compiler, its Scheduling algorithm, the Program Code, the pattern of the data involved in the convolution and the final encoding stage through the ISA; multiple examples are introduced to show the type of instruction schedule.

The chapter *Hardware implementation* discuss the HW architecture that is possible to obtain with the proposed schedule for the OS dataflow, moreover this chapter provides the results obtained with this model.

The last chapter *Conclusion and Outlook* gives a summary of the entire design flow followed throughout this work and an evaluation of the results; the potential future works are also described.

2. Background

This chapter is divided in two sections: the first one provides an overview of CNN as main subfamily of DNN employed in computer vision tasks, the second introduces the state-of-the-art hardware architecture for the CNN inference and the data movement in such architecture.

2.1. Deep Neural Networks

The employment of DNNs in a wide range of applications is due to their ability to extract high-level features over a large amount of data to obtain an effective representation of the input space [1], reaching high performance and accuracy.
Neural Networks (NNs) are biologically inspired algorithms made up of neurons, the basic computational element, that receive input signals from dendrites and produce output signal on the axons (single element per each neuron), then the axon out of a neuron can connect with a synapse to a dendrite of other neurons. Input and output signals of a neuron are also called activations. The activation x_i can be scaled up or down through the synapse by a factor w_i, so all these products $x_i w_i$ can be added together with a possible addition of a bias b, following a non linear function $f(.)$ (or activation function) on the final sum to get the output signal y_i. The most popular activation function is the rectified linear unit (ReLU) that consists in a max operation with the threshold set to 0.

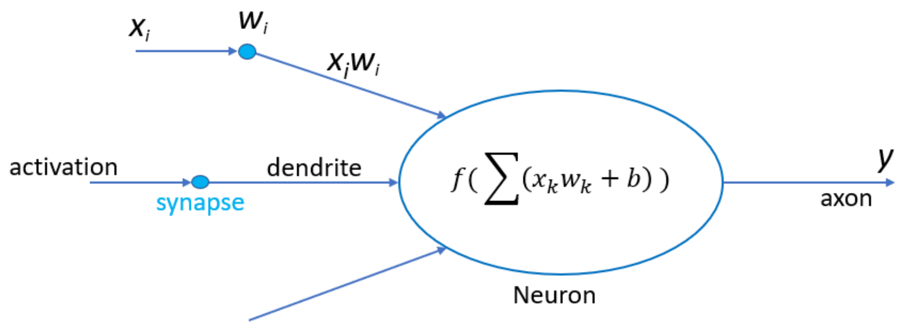

Figure 2.1.: Model of a neuron

The key characteristic is that the weight w_i can be changed to have a different response to the input, representing in this way the learning process. Hence the training or learning of a DNNs consists just in determining the values of weights and bias, while the inference is the running of the network with the trained weights. The analysis of the training process goes beyond the scope of this work.

To create a NN, clusters of neurons work together to create layers and a batch of layers are put in sequence to form the network. Sometime *pooling* (or subsampling) is employed to reduce the dimensions of the feature maps, hence reducing the overall amount of parameters in the network.
In literature there are 2 main categories of NN: feedforward networks, where there are sequences of computations on the outputs of previous layers, and recurrent networks, in which some intermediate outputs can be stored internally and used as next inputs. Anyway only feedforward networks will be considered from now on in this work. Middle layer of the NN is also frequently called "hidden layer".
The types of connections among layers that can be identified in a DNN are fully connected (FC) and sparsely connected layers, with the former denoting a situation where all outputs of a layer are connected to all inputs of the next layer, while in the latter not all the neurons of two adjacent layers are connected each other because some connections are removed by setting the corresponding weights to 0. Typically, the FC layer is the last one of the network, used for classifying the features extracted by the CONV layers.

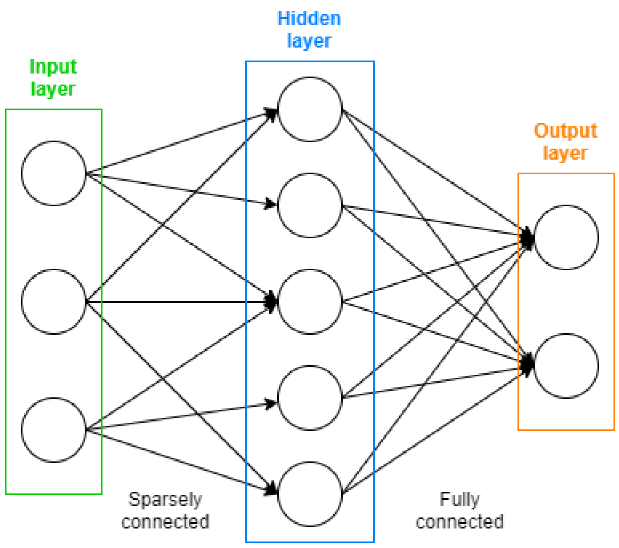

Figure 2.2.: Simple neural network.

In FC layers a significant amount of storage and computation is required because of the high number of weighted sum in a layer, so the global efficiency can be increased by using the same set of weights in the computation of every outputs of a layer. Convolutional Neural Networks are popular types of DNN that employ *weight sharing* and organize the computation as a convolution. A more detailed explanation of this kind of network is provided in the next section.

2.2. Convolutional layers

CONV layers are the basic elements of CNNs, employed in feature extraction and classification of the input image [12]. A CONV layer is composed by multiple sets of shared 4D filters of small size that convolve over the 3D input image (called input feature map or *ifmap*) in order to generate 3D output feature maps (*ofmap*).

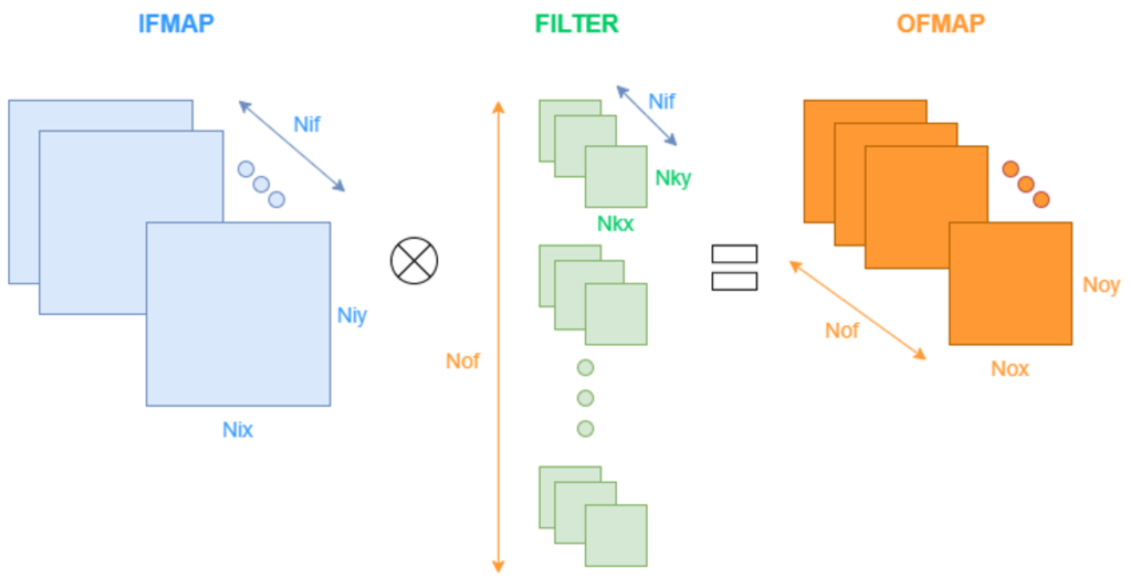

Figure 2.3.: CONV layer

Shape parameter	Description
Nif	input channels
Nix	ifmap width
Niy	ifmap height
Nkx	kernel width
Nky	kernel height
Nof	output channels
Nox	ofmap width
Noy	ofmap height

Table 2.1.: Notation of the layer shapes

Basically, the convolution operation consists in sliding a small window of $N_{ky}N_{kx}$ size (i.e. the kernel) over the input image of larger size ($N_{iy}N_{ix}$) to produce a partial sum ($psum$), which is accumulated over the remaining spatial dimension N_{if} to get the final output pixel of one channel. The same operation is repeated with a new set of filters to compute the ofmap of the other output channels along Nof direction. So, the main operation of the convolution is the multiply-accumulate (MAC), namely the element-wise multiplication between the input pixels and the weights, while the psums are added together to obtain a single value. This operation, starting from the top-left corner of the ifmap, is repeated again by moving the window by S strides at a time in one spatial direction (usually towards right) and, when the edge of the ifmap is reached, the sliding proceeds downwards until the ifmap is completely scanned [5]. The following example shows the convolution among a 3x3 kernel (on the left) and a portion of ifmap (in the middle), while the resulting psum is in a portion of ofmap plane on the right; the stride is 2, so the window (i.e. kernel) slides by 2 position on the right.

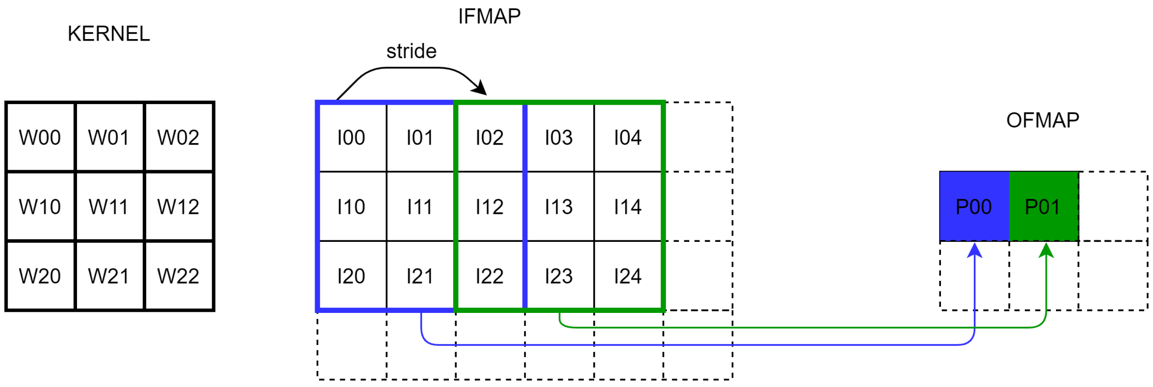

Figure 2.4.: Convolution with focus on two sliding windows. Different colors are associated to different sliding window (i.e. different psum on the ofmap)

$$\begin{aligned}P00 = {} & I00 \cdot W00 + I01 \cdot W01 + I02 \cdot W02 \\ & + I10 \cdot W10 + I11 \cdot W11 + I12 \cdot W12 \\ & + I20 \cdot W20 + I21 \cdot W21 + I22 \cdot W22\end{aligned}$$

$$\begin{aligned}P01 = {} & I02 \cdot W00 + I03 \cdot W01 + I04 \cdot W02 \\ & + I12 \cdot W10 + I13 \cdot W11 + I14 \cdot W12 \\ & + I22 \cdot W20 + I23 \cdot W21 + I24 \cdot W22\end{aligned}$$

The MACs required to obtain those 2 partial sums are listed above: the first index is about the column, the second is for the row.

The spatial dimensions of the output and input volume can be computed using the following formulas according to [13] and adopting the notation introduced in the table 2.1:

$$N_{ox} = \left\lfloor \frac{N_{ix} - N_{kx} + 2P_x}{S_x} + 1 \right\rfloor \quad , \quad N_{oy} = \left\lfloor \frac{N_{iy} - N_{ky} + 2P_y}{S_y} + 1 \right\rfloor \tag{2.1}$$

$$N_{ix} = (N_{ox} - 1)S_x + N_{kx} \quad , \quad N_{iy} = (N_{oy} - 1)S_y + N_{ky} \tag{2.2}$$

Thus the dimensions of the ofmap depend strictly on the stride (S_y and S_x) of the convolution and on the paddings (P_x and P_y) of the input; throughout this work the input dimensions are always considered already padded.

The canonical form of the CNN is evaluated by the 7D nested for-loops [14]:

```
for (ba=0; b<B; b++)
  for (no=0; no<Nof; no++)
    for (y=0; y<Noy; y++)
      for (x=0; x<Nox; x++)
        for (ni=0; ni<Nif; ni++)
          for (ky=0; ky<Nky; ky++)
            for (kx=0; kx<Nkx; kx++)
              psum[no; y, x] += ifmap[ni; y*S+ky, x*S+kx]
                                * weight[no, ni, ky, kx]
```

Listing 2.1: Nested convolutional loops

These nested loops can be broken down by applying loop optimization techniques [15] such as loop unrolling, tiling and interchange (or interleaving) leading to a large design space: there are different choices for implementing parallelism, sequencing the order of computations and especially partitioning the large amount of data into smaller chunks to fit a memory hierarchy level [14], with the purpose of improving the efficiency of the CNN inference step.
One main focus of this work, which is depicted in next chapters, is setting the nested loops of the convolution according to a particular dataflow, the Output Stationary, in order to have the best schedule of the computation and the best reuse of the involved data.
Indeed the inner loops of the convolution may iterate many times on the same portion of a volume of data, thus some data are required several times in different iterations, offering various *data reuse* opportunities and reducing as a consequence the communication between different levels of memory to exchange data.

2.3. Hardware model for CNN inference

In CNN, more than 90% of the operations involve convolutions [16], i.e. MACs, therefore acceleration schemes should focus on the management of parallel computations and the organization of data storage and access across multiple levels of

memories [14].
CNN algorithms involve a large amount of data, so a unique level of memory is not suitable to store all ifmaps, weights and psums. In a typical CNN accelerator there are 4 levels of storage hierarchy:

1. off-chip memory (usually a DRAM)
2. on-chip global buffer (GLB)
3. array (inter-PE communication)
4. register file (RF), or scratchpad (SPAD), associated to the processing element (PE)

Basically, chunks of data are fetched from the external memory to the GLB, then they are transferred to the RFs within the multiple processing elements through a Network on Chip (NoC). After that the computations are over, the results are sent back from the PEs to the GLB and also to the DRAM if necessary.
It is important to highlight that the energy cost, associated to a read/write operation, is higher at the DRAM than at the RF, so having an efficient accelerator in terms of energy consumption implies to optimize the number of accesses to the levels of storage. The multiple levels of memory hierarchy provides low cost data accesses: fetching data from RF or neighbour PEs will cost one or two orders of magnitude lower energy than from DRAM [1].

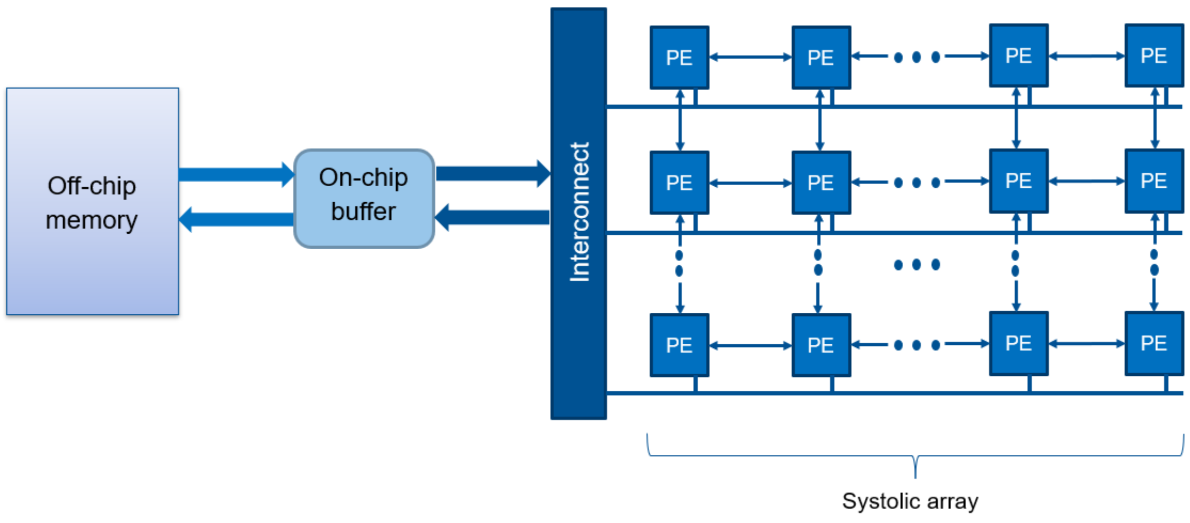

Figure 2.5.: Spatial architecture

Spatial architectures, or 2D Systolic array [17], is made up of an on-chip buffer and an array of PEs, connected to the GLB through Interconnects. Each PE has

a local scratchpad (RF) for storing data, an ALU datapath for MACs and other types of operations, a control logic for handling the data and for interfacing to the Interconnect.
The focus of this works is mainly in the PE array section of the accelerator and the aim is to develop, according to an offline compiler, a smart deployment of the instructions to the systolic array.

3. Related work

Prior works have proposed several hardware designs for CNN acceleration, providing simulation results [18, 19, 20, 9] and measured results from fabricated chips [12, 21, 22]. By now there are several variants of cutting-edge accelerators with multiple levels of storage hierarchy, sharing the same concepts as dataflow mapping, inter-layer scheduling (i.e. pipelined execution of layers) memory scheduling (defined also as loop blocking techniques) and differing in the underlying hardware platform and employed dataflow inside it.
These accelerators are based on analytical memory bandwidth models, to optimize the loop-nest algorithm for CNN and to find a suitable dataflow schedule targeting the lowest communication bandwidth according to the memory constraints. Having a reduction in the memory bandwidth means having also a reduction in the power consumption, because the total number of accesses to fetch data is lower.

CNN mapping methods in hardware architectures A flexible architecture has to be able to support different ways of scheduling operations and staging data on the same architecture [8]. The infrastructure *Timeloop* has evaluated and explored, in a systematic way, the architecture design space of DNN accelerator through a *model*, for describing performance and energy efficiency, and a *mapper* to configure the accelerator to the optimal feature. Conversely, other works have dealt the architecture design space problem within the scope of their specific design [12, 23, 24, 25].
Similarly, the *DNNWeaver* framework generates a custom synthesizable accelerator on FPGA, starting from the high level specification, through its novel ISA that represents a macro dataflow graph of the DNN [26]. The mentioned tool comprises: hand-optimized template designs to match the needs of the DNN and the available resources on the FPGA; algorithms to find the best tradeoff between schedule of the instructions and memory accesses.
The *Cambricon* ISA, inspired by the RISC-ISA design principles, has a comprehensive and descriptive capacity over a broad range of NN techniques, showing that the ISA design is a fundamental step that can limit both flexibility and efficiency of NN accelerators [27]. In fact, customizable hardware accelerators such as [9, 28, 29] adopt a more straightforward instructions set but can support a small set of NN techniques, because as the NN techniques grow up, the design complexity and the area/power overhead become unacceptable.
On the other hand, ASICs are also used as accelerator for CNN inference, but the majority of them targets single-hardware design point. The *Gemmini* framework is

an open source and agile systolic array generator, enabling systematic evaluations of deep-learning architectures [22] and targeting ASIC and FPGA implementations. It takes into account all the design parameters from software frontend and hardware backend: layer dimensions or model size impact strictly on the mapping and scheduling of a *workload* into a particular HW architecture [30]; instead, on the hardware side, the system-level integration parameters such as power, area and maximum clock frequency are affected by pipeline depth, employed dataflow, banking strategy, memories capacity and bandwidth. Therefore, a model-based methodology is useful in the HW/SW codesign process for deep learning workloads.

Dataflow categories Generally, the memory access represents the bottleneck for processing data. A dataflow can increase data reuse within the PE array, in order to reduce the number of accesses to memories and the energy consumption associated to them. It also takes into account the storage capacity of the low cost memories, which cannot fit all the data to be processed for a certain CNN. The taxonomy introduced in [1] can be used to classify DNN dataflows in works like [18, 28], based on their data handling characteristics [12]:

- *Weight Stationary (WS)* minimizes the energy consumption of reading weights by keeping them in the PE, whereas the ifmaps are broadcast to it for the convolution and the psums are spatially accumulated through the PE array.

- *Output Stationary (OS)*: each PE of the array holds a psum for the same output activation value, while the ifmaps are streamed across the array and the weights are broadcast to all PEs. Possible variants of OS dataflow can be adopted depending on the organization of the way the 7D nested loop algorithm: these techniques will be discussed in the next chapter P

- *No Local Reuse (NLR)*: no data is stationary in the PE and they are always fetched from the on-chip memory, which has a larger storage capacity.

- *Row Stationary (RS)* dataflow, proposed the first time by Eyeriss accelerator, maximizes the reuse and accumulation of all types of data at PE level, by keeping rows of filter weights in the RF and then the ifmaps are streamed into PEs of the array along the diagonal direction, implying an accumulation of the psums across the vertical direction.

The above taxonomy used techniques already mentioned, like loop interchange and unrolling, to get a different order of the CNN nested loops and way of iterating over the feature maps. Moreover, dataflows are meant to be static, namely a CNN layer has to be mapped using a particular dataflow, but the possibility of choosing a suitable dataflow for each layer could lead to a better execution of the layer itself.

Design flow of CNN accelerators What comes out from the above works is that the operation of DNN accelerators is analogous to that of general-purpose processors [7]: in a standard computer system, the compiler translates the program into machine-readable binary codes for the executable, given a specific HW architecture (x86 or ARM), while in the processing of DNNs the layer shape and size are translated by the mapper into a HW-compatible schedule of computations, given a specific dataflow. Similarly the HW architecture has to be flexible and adaptive to the layer shape configuration, rather that hardwired to process certain shapes, in order to improve throughput and energy efficiency.
Anyhow, selecting an energy-efficient scheduling for a CNN is challenging and requires an extensive search of loop schedules. What has been done in [31] and [32] addresses exactly the first steps of the mapping process of CNN:

- the optimization of data movement is carried out in the former work by a design space exploration framework, which provides some hardware metrics like communication schedules and memory statistics that can be used in the SW/HW codesign process; to determine efficient communication patterns for the memory usage (across different layer configurations), the framework exploited loop optimization techniques and inter-layer tiling and reuse strategies

- the HW requirements are considered in the latter work, which aims in finding the optimal resource binding strategy for a given network and architecture; in this case, an HW model provides a high level description of the main architectural parameters like number of cores and memory size, thus they are used in an energy and latency model to find optimized and valid solutions of mapping

So the first works deals the optimization methods for the off-chip communication, while the second one focuses more on the on-chip communication with the PE array, dealing both with strided convolution mostly and also GEMM algorithms.
This thesis represents a continuation of these works, therefore the main topic will be the schedule of the operations at the PE array level and the corresponding HW implementation, namely the last step of the design process for CNN accelerators. From now on, only strided convolutions will be considered.
The dispatch of data and instructions through the Network on Chip (NoC) to the PEs in the systolic array (SA) has been studied in parallel to this work [33].

4. Mapping of CNN into spatial architecture

As explained in previous chapter, this thesis is focused on providing a proper schedule of operations to execute a generic CNN layer on a PE array, thus it is related to the lowest part of the hierarchy in a systolic accelerator. It is important to plan the *design flow* from SW to HW, considering all the parameters that can affect the mapping. Before doing that, the dataflow within the systolic array needs to be analyzed in order to understand the correlation between HW architecture and high-level system parameters (section 4.2). This breakdown is carried out only for the OS dataflow, so the parameters that describe it are related only to this type of dataflow.
A detailed overview on the HW blocks in a systolic architecture is given in section 4.1. From the SW side, the loop blocking techniques can affect the HW implementation, so section 4.3 provides a review of the most common methods.
Then, everything is put together in section 4.4 in the mapping step of CNN into spatial array.

4.1. Spatial architecture

4.1.1. Overview on HW implementation

Generally there are two types of Systolic/Spatial accelerator: *coarse-grained* SAs (like multi-core processors), made up of tiled arrays of ALU-style PEs connected together through an on-chip network [34, 35], basically implemented in ASIC; *fine-grained* SAs (like SIMD processors) usually employed in FPGA.
The first type of SA has better performance than the second one, due to its HW uniformity and higher parallelism that can favour the execution of large number of MACs. Consequently, if from one side ASIC are more specialized to a PE array datapath only for CNN execution [10, 36], on the other side the FPGA can be also used to build a CNN accelerator by exploiting integrated DSP slices to construct the PE datapaths [18, 20]. Certainly in both type of design the main challenge is finding the right mapping of the CNN acceleration into the SA, in terms of latency and energy efficiency.

4.1.2. Processing Element

The Processing Element is the computational core of the array and it comprises a local small memory (i.e. RF), typically below 1kB, one or more ALU datapath able to perform MACs operation, a control logic to interface with the Interconnect and the neighbour PEs.
The RF can be exploited to have a low cost data reuse for the ALU; it can store all types of data, namely ifmap, weight and psum during the accumulation.

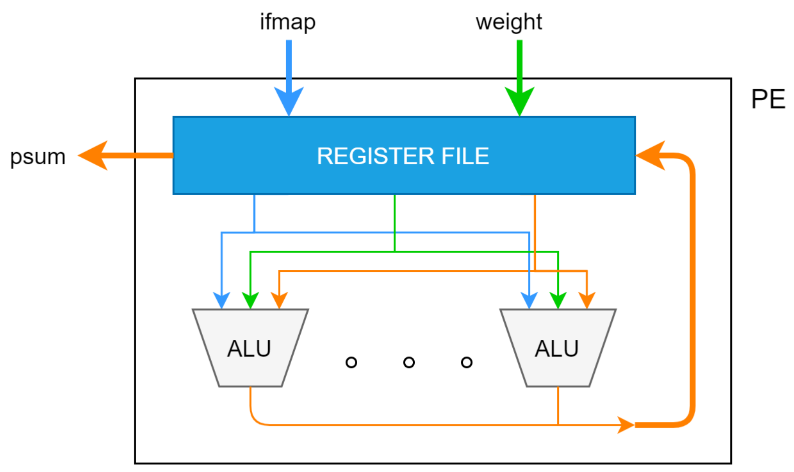

Figure 4.1.: Processing Element block scheme; the different data types are represented with different colors

4.1.3. Network on Chip

The NoC is in charge to dispatch instructions, which includes data and other parameters useful to perform the MACs, and to retrieve the final psums from every PE when the computations are over.
Thanks to the deterministic execution of CNN, the communication with the PE array does not require necessarily complex routing algorithms, usually supported by NoCs. A flexible and efficient on-chip Interconnection can be considered to support broadcast, multicast and unicast transfers to cover different deployment of the instructions. So, a *Time-Division Multiplexing (TDM) Interconnect* can be employed for this kind of usage, which is able to sustain a dataflow and reduce complexity in data delivery patterns compared to other more complex and expensive solutions like those of [37, 38].
In particular, the NoC comprises an array of vertical and horizontal Interconnect associated to every PE:

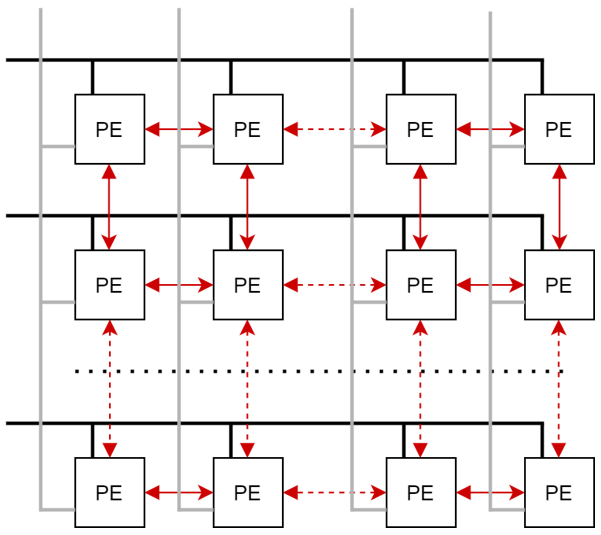

Figure 4.2.: The horizontal and vertical interconnects are depicted in black and grey respectively. In red, the n2n communication among neighbour PEs

4.1.4. PE array

The PE array allows to reach high compute parallelism by exploiting:

- communication between on-chip buffer and PE array through the Network on Chip

- point-to-point communication among adjacent PEs, called also *neighbour-to-neighbour communication* (n2n), which allows a direct interchange and flow of data between PEs without passing through the NoC

There is also some control logic and buffers used to interface correctly the PE array with the GLB and viceversa, but they are not covered in this work.
Each time the computation of a CONV layer starts, the Interconnect deploys *instructions* in the correct order to the target PEs.
Accessing to the global buffer has a certain energy cost, so it is better to minimize the number of accesses to have a low energy consumption associated to the data transfers. The n2n communication allows to reuse or accumulate data within the PE array, without taking them each time from the GLB and relaxing more the NoC when delivering a message.
Therefore the data reuse and the inter-PE communication needs to be considered in the scheduling of operations.

4.2. Data movement in the spatial array

The main challenge when accelerating a CNN is finding the right trade-off between energy consumption, throughput and number of computations that can be settled in the PE array. Although the large number of MACs can be easily parallelized, the throughput cannot be scaled up in the same way due to bandwidth constraints. Furthermore, the data movement can be more expensive than the computation itself [39, 40].
An efficient CNN processing is achieved by defining a proper dataflow that supports parallel processing, with the minimal data movement and the low cost memory level usage[41], without compromising the global performance.
The types of dataflows differ each other depending on:

- the way the 3 data types are stored in the storage hierarchy
- the data movement between GLB and array and in the array itself
- the direction of accumulation of the psum during the computation

Hence a dataflow directly affects the schedule of the operations required to compute the final output pixels of a certain CONV layer. In this work, the designed HW architecture is conceived to support the Output Stationary dataflow and two variants of it, according to loop unrolling and interleaving.

4.2.1. Output stationary dataflow

This type of dataflow minimizes the energy consumption associated to reading/writing operations of psum, since that the accumulation of an output activation happens locally in the RF of the PE and not spatially in the array.
There is a broadcast/multicast of the weights to the PEs processing the same input channels of the CONV layer, while the ifmap pixels are streamed along the spatial direction of the array in order to reuse them efficiently, reducing the number of accesses to the GLB.

The PE array can process ofmap along different dimensions according to a taxonomy defined in [41], which identifies 3 possible variations of the OS dataflow:

- OS Type-A (OS_A), suitable for CONV layers, focuses on processing a single plane of ofmaps at a time, maximizing the convolutional reuse, i.e. ifmap and kernel are reused several times within a given channel in different combinations.
- OS Type-B (OS_B) processes multiple ofmap belonging to different channels by using multiple pixels of the same ifmap at a time.
- OS Type-C (OS_C) is employed mainly for FC layers, because it computes multiple ofmaps but considering only one pixel at a time.

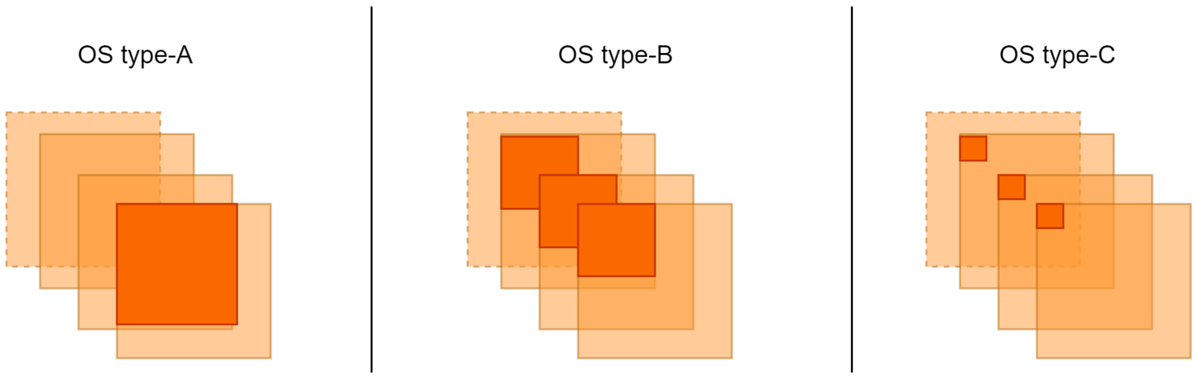

Figure 4.3.: Variations of OS dataflow. The darker part is the one being processed

Hence, the type of dataflow adopted in the HW architecture determines which pixel and weight of the CONV layer have to be allocated in the RF of every PE. In this work, the HW implementation supports OS_A and OS_B dataflow, giving more flexibility to the PE array.

4.3. Loop scheduling techniques

The problem of CNN mapping, given an input CONV layer, can be seen as finding the optimal architectural configuration of PEs in the array that can compute the psums efficiently. From an other point of view, this is a loop optimization problem [42, 43, 44, 45], so techniques like loop unrolling, tiling and reordering/interchange are applied to the nested convolutional loops to find the best configuration of PEs. Furthermore, these techniques modifies the data access pattern on the memory levels in the accelerator and influences also their minimum size, so a good loop schedule must optimize all these aspects.

4.3.1. Loop tiling

Applying loop tiling to the nested convolutional loop means dividing the execution into smaller loops, that receive new blocks of data smaller than the total loop dimension.

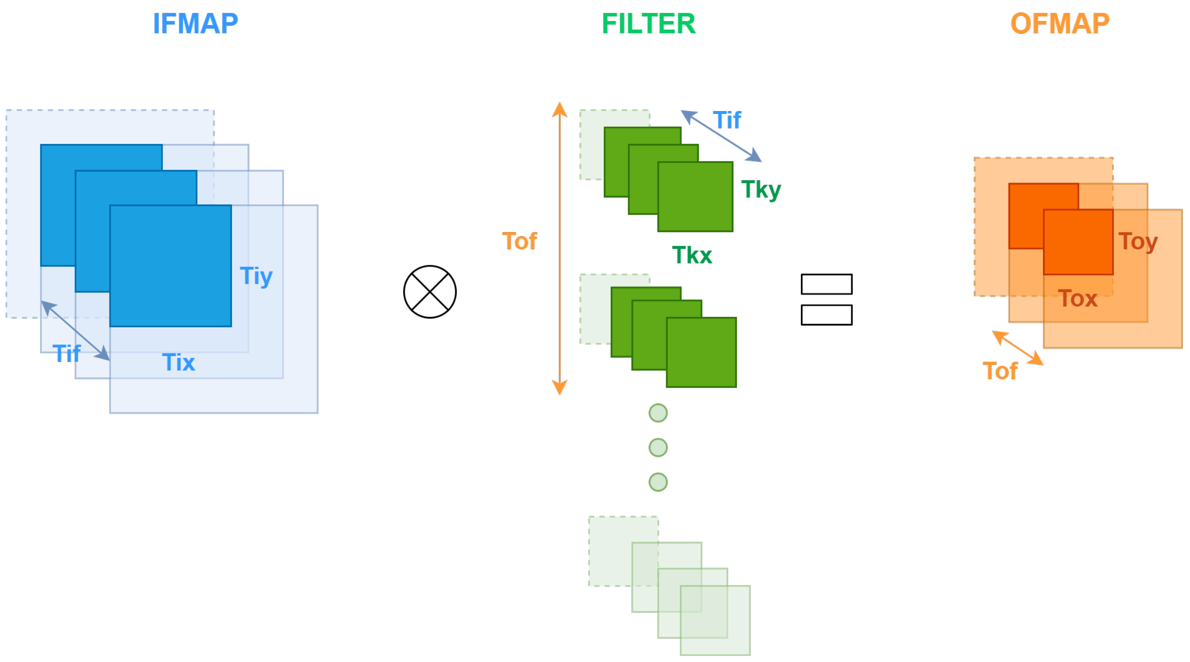

Figure 4.4.: Tiles in the layer shapes

ifmap	kernel	ofmap
Tix, Tiy, Tif	Tkx, Tky	Tox, Toy, Tof

Table 4.1.: Tiling sizes

Loop tiling divides the entire input volume into multiple blocks of data, in this way the on-chip buffer can accomodate tiles of data sent by the denser off-chip memory. External memory access happens when going from one tile to the next and are the most energy expensive, so the data transferred from DRAM to GLB must be reused as much as possible.
Loop tiling sets also the lower bound on the required on-chip size [14]:

$$\begin{aligned} GLB\_size &= ifmap\_volume + weight\_volume + ofmap\_volume \\ &= T_{ix} \cdot T_{iy} \cdot T_{if} \cdot T_b \cdot pixel\_datawidth \\ &+ T_{kx} \cdot T_{ky} \cdot T_{if} \cdot T_{of} \cdot weight\_datawidth \\ &+ T_{ox} \cdot T_{oy} \cdot T_{of} \cdot T_b \cdot pixel\_datawidth \end{aligned} \tag{4.1}$$

The kernel dimensions are never tiled since that are typically small, so $T_{kx,y} = N_{kx,y}$.

4.3.2. Loop unrolling and interleaving

Unrolling convolution loops leads to accelerate their execution with an hardware overhead. The different parallelization of computation affects the optimal PE architecture with respect to data reuse opportunities and memory access pattern [14].

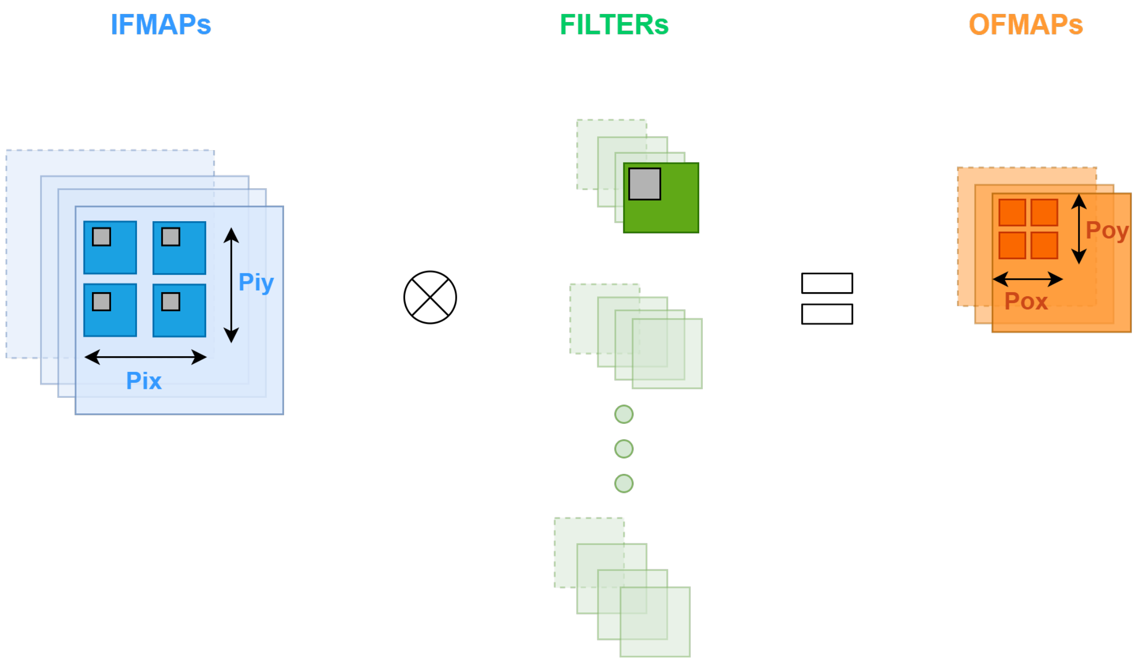

Figure 4.5.: Unrolling in the layer shapes for the OS dataflow; the darker areas are the effective ones processed in the PE array

Therefore, the unrolling variables determine the mapping and the schedule of the parallel MAC operations into the PE array.

$$1 \leq P \leq T \leq N$$

$$T_{ox,y} \rightarrow P_{ox,y} \quad , \quad T_{ix,y} \rightarrow P_{ix,y} = (P_{ox,y} - 1) \cdot S_{x,y} + T_{kx,y} \tag{4.2}$$

Loop interleaving [41] is applied to the tiling parameters T_{if} and T_{of}, obtaining q for the input channels and p for the output channels. In this way the computation in a single channel is halted and resumed along the channel dimension to maximally reuse data.

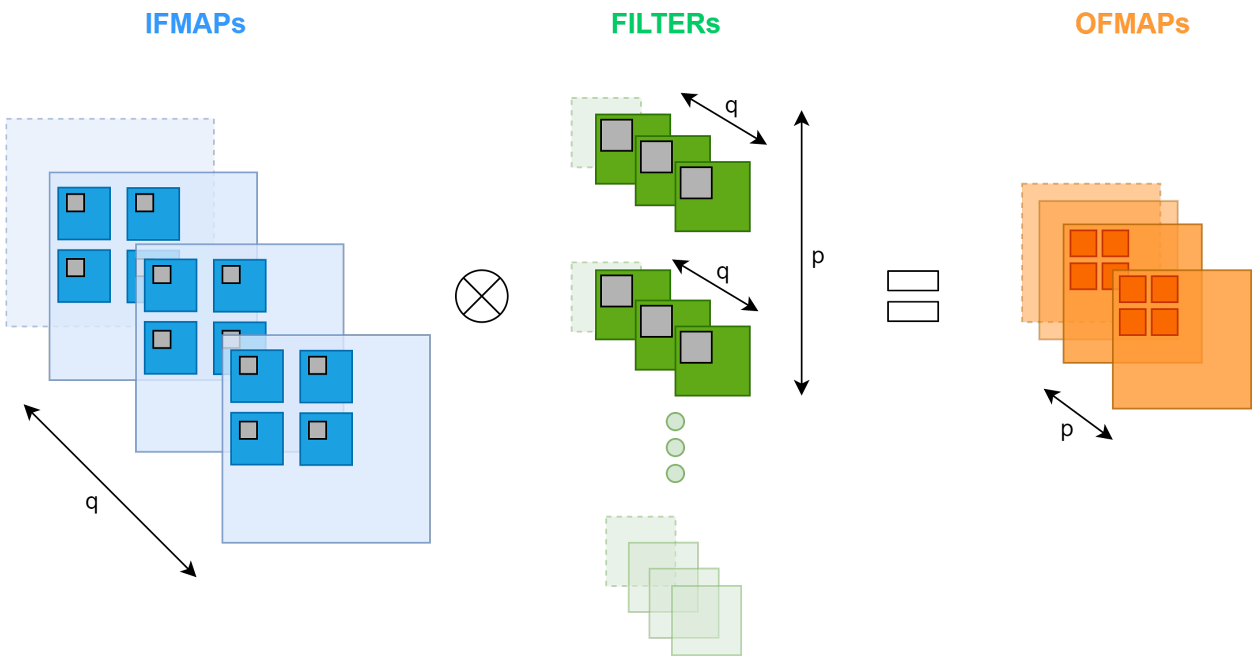

Figure 4.6.: Interleaving and unrolling depicted in the layer shape; the darker areas are the effective ones processed in the PE array, so it is clear that every PE needs more data

Furthermore, the usage of loop interleaving causes an increase of the SPAD size because a higher number of ifmap or weight needs to be hold at PE level. So, loop interleaving is only beneficial for the energy and not for the memory size.

4.3.3. Loop interchange

Loop interchange decides the processing order of the loop tiles and also the type of dataflow at the algorithm level, i.e. the pattern of data movement [14] between the different storage hierarchy.

4.4. Logical and physical mapping

The mapping process of a CNN model in the systolic architecture can be grouped in 3 steps:

1. finding a feasible and optimal configuration of tiling factors that can be allocated into the storage hierarchy
2. identifying a PE array configuration, according to the dataflow and unrolling factors, that can theoretically perform the MACs in a parallel way exploiting as much as possible a local reuse of data

3. mapping these configurations into the physical array, considering the HW constraints and the schedule of the operations for each PE

In literature [41], the first step is referred to as *logical mapping*, while the second one is called *physical mapping*: at the beginning the computations are mapped according only to the dataflow and layer shapes, then the HW parameters are taken into account and, if the PE array cannot fit all the operations, the schedule of the computation is serialized in order to find a correct HW implementation.
The mapping steps are done statically before runtime through an offline compiler; the one employed in this work is explained in next chapter.

4.4.1. PE set

Given a dataflow, a *PE set* is the minimal block of PEs that processes data in a parallel way, according to some unrolling parameters and to some logic. If the PE array size is large enough, multiple PE sets can be replicated in the array in order to further improve performance.

OS dataflow mapping In this type of dataflow, the accumulation of the psum is stationary in the PE, so a PE set is responsible for a portion of the ofmap; its width and height is determined by the unrolling factors P_{ox} and P_{oy} respectively.

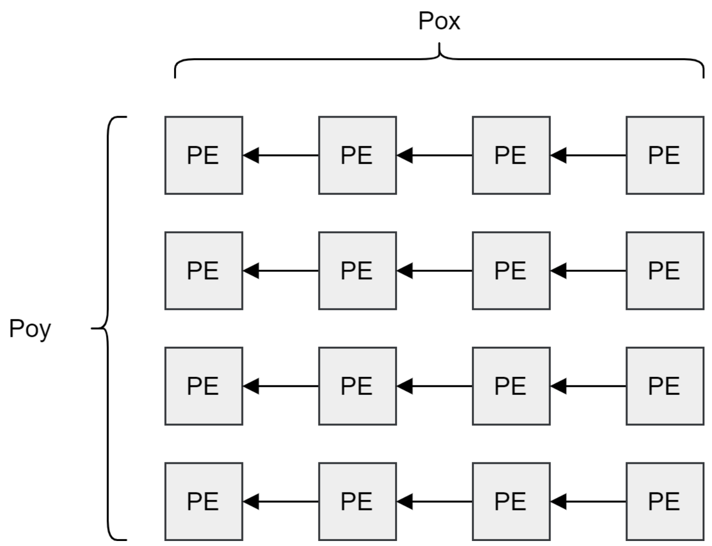

Figure 4.7.: Example of 4x4 PE set for the OS_A dataflow, with Pox=Poy=4

This first implementation is the OS_A dataflow, so the above PE set processes the output pixel of just one output channel, but it is possible to map in a PE set also multiple ofmap channels according to the interleaving parameter p, which leads to OS_B dataflow configuration because in one PE set there is an accumulation of psums belonging to different channels. In this way the ifmap reuse is favoured with respect to other data type.

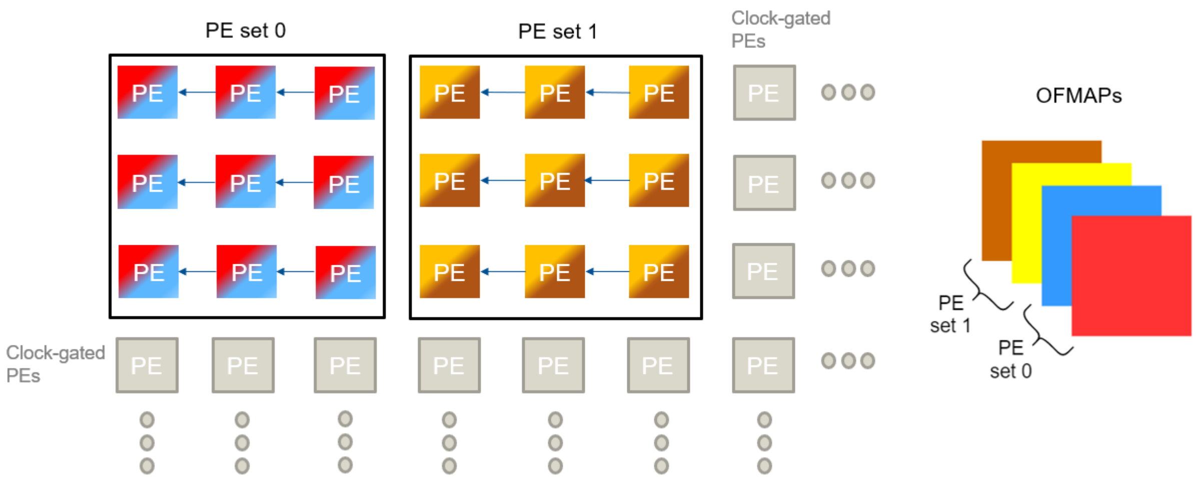

Figure 4.8.: Example of two 3x3 PE sets for the OS_B dataflow, with Pox=Poy=3 and p=2. Since that Tof=4, two PE sets are required to process 2 output channels each (different colors refer to different output channels). The PEs in grey are clock-gated to save energy

In this example, two PE sets are mapped in the array horizontally, so P_{of} *vertical* is 1 and P_{of} *horizontal* is 2: these new parameters are used to map the logical PE sets, necessary to cover all computations depending on the current dataflow, in the physical array that is characterized by a limited size.

A further variation of the OS dataflow consists in considering the unrolling factor q as well as p, which means that the ifmap pixels are fully reused because they are fetched only one time from the GLB for a specific output pixel of the ofmap. Furthermore, the more the unrolling factors p and q are taken into account, the more the complexity and the RF size increase.

Data reuse In a convolution, multiple input pixel belonging to adjacent sliding windows can be reused within a PE set, if the stride is such that there is an overlap, by exploiting the n2n communication of neighbour PEs.
In the OS dataflow, the reuse happens only along the horizontal direction of the array from right to left because of the sliding movement of the kernel over the ifmap from left to right.

A *ping pong buffer* can be employed to avoid data conflict and dual-port memory in the RF. In particular, it is a type of double buffer useful when there is a I/O operation with data busy in a computation.
The reading/writing operations on the 2 buffers are handled with a switch that changes when the 2 blocks of data are completely processed, i.e. when the pointer to the buffer reaches the last filled location. Thus the usage of ping pong buffer increases the overall throughput and prevents eventual bottlenecks.

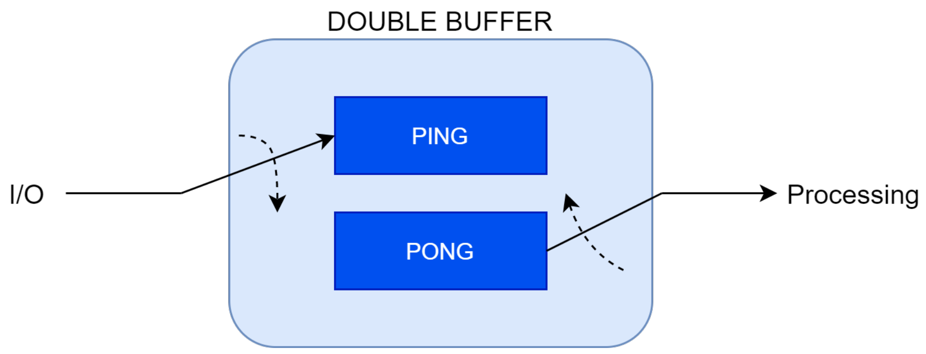

Figure 4.9.: Ping pong buffer block scheme

5. Modeling tool

The OS dataflow and the loop optimization techniques provide a particular configuration of the nested loops for CNN. The *Compiler* introduced in section 5.1 employs this type of configuration (5.1.1) to translate the high level specifications into the atomic operations at HW level, which enclose the main features of the dataflow.
Indeed the main purpose of this breakdown is to characterize the *figure of merit* (FOM) involved in the mapping process of a CNN layer.
So it is possible to identify a regular *Data pattern* (section 5.2) for the OS dataflow, which is used to make a proper *ISA* able to encode the atomic jobs into few instructions to dispatch in the PE array. The characteristics of this novel ISA are described in section 5.3. All these concepts are merged together to make a proper *Modeling tool*:

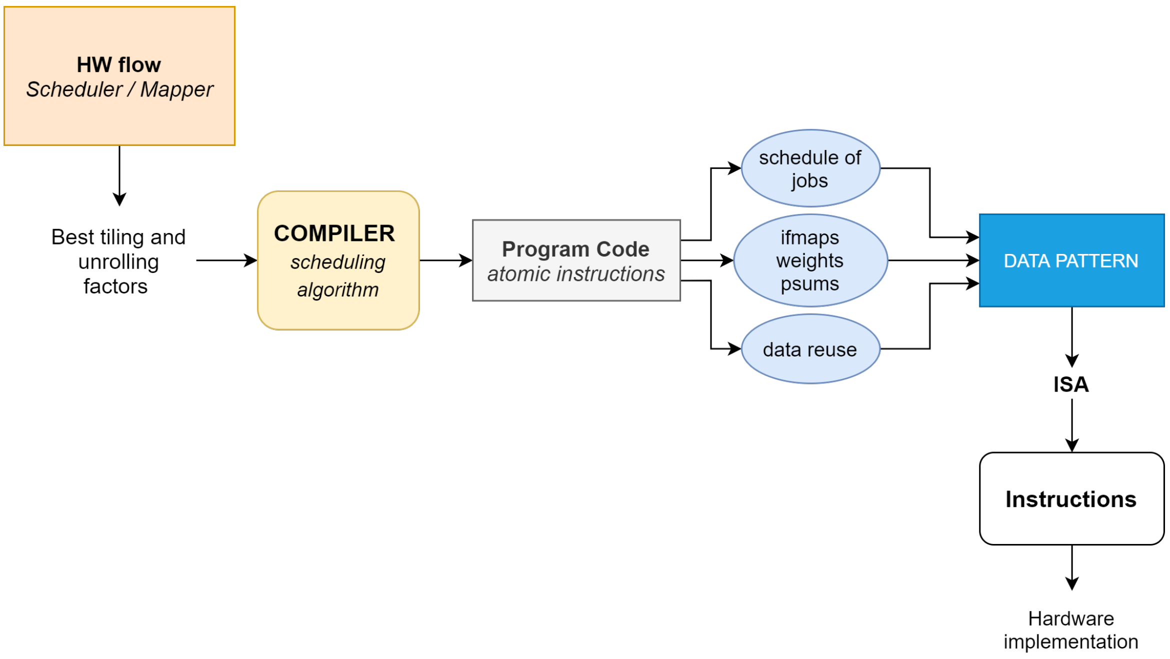

Figure 5.1.: Flow chart of the Modeling Tool

Each functional block of the tool is discussed in next sections.

5.1. Compiler

The inputs of the Compiler are the tiling and unrolling factors associated to the layer shape, which are given by the tools developed in prior works [32, 31].
Basically, in CNN there are 3 levels of estimation:

1. *Coarse level*: high level estimation of the CNN in terms of size of memory blocks, number of operations and parameters
2. *Mid level*: off-chip to on-chip movement, it is the estimation of the tiles of computation
3. *Fine level*: on-chip movement, namely in-parallel HW processing or spatial distributed computation

The CNN estimation is carried out in *Hardware flow*, a HW/SW codesign framework which breaks down the high-level computation graph of a neural network into a bunch of computation blocks that can be implemented in the hardware.
The Compiler of this work is about the fine level estimation of a CNN: it maps multiple computations in a parallel HW architecture, considering also the temporal parallelism such that the sequence of operations is correct and the CNN algorithm is not violated.

5.1.1. Scheduling algorithm

The algorithm of the compiler is based on the nested convolutional loop (2.2), but some changes need to be done by using loop blocking techniques in order to increase the parallelism, process an affordable number of data in memory, iterate in an efficient way over the layer shapes to have data reuse possibilities and use data in a smart way.

OS dataflow breakdown The canonical version of the CNN algorithm is already set to support OS dataflow, indeed the outer loops are about the ofmap volume and the psum is accumulated in the PE without moving during the computation, which means that, along the sequence of MAC operations, the index of the psum is fix and the indexes of the ifmap and weight are being updated as depicted in 2.4.
Being a fine level estimation of CNN, the algorithm of the compiler takes care only the tiles loaded in the on-chip buffer (table 4.1).
When applying loop unrolling, the input pixels processed at the same time is larger and it depends on P_{ix} and P_{iy}, i.e. the corresponding unrolling factors of P_{ox} and P_{oy}. The number of psums that can be computed in parallel is allocated spatially in a number of PEs according to the horizontal and vertical unrolling factors P_{ox} and P_{oy}:

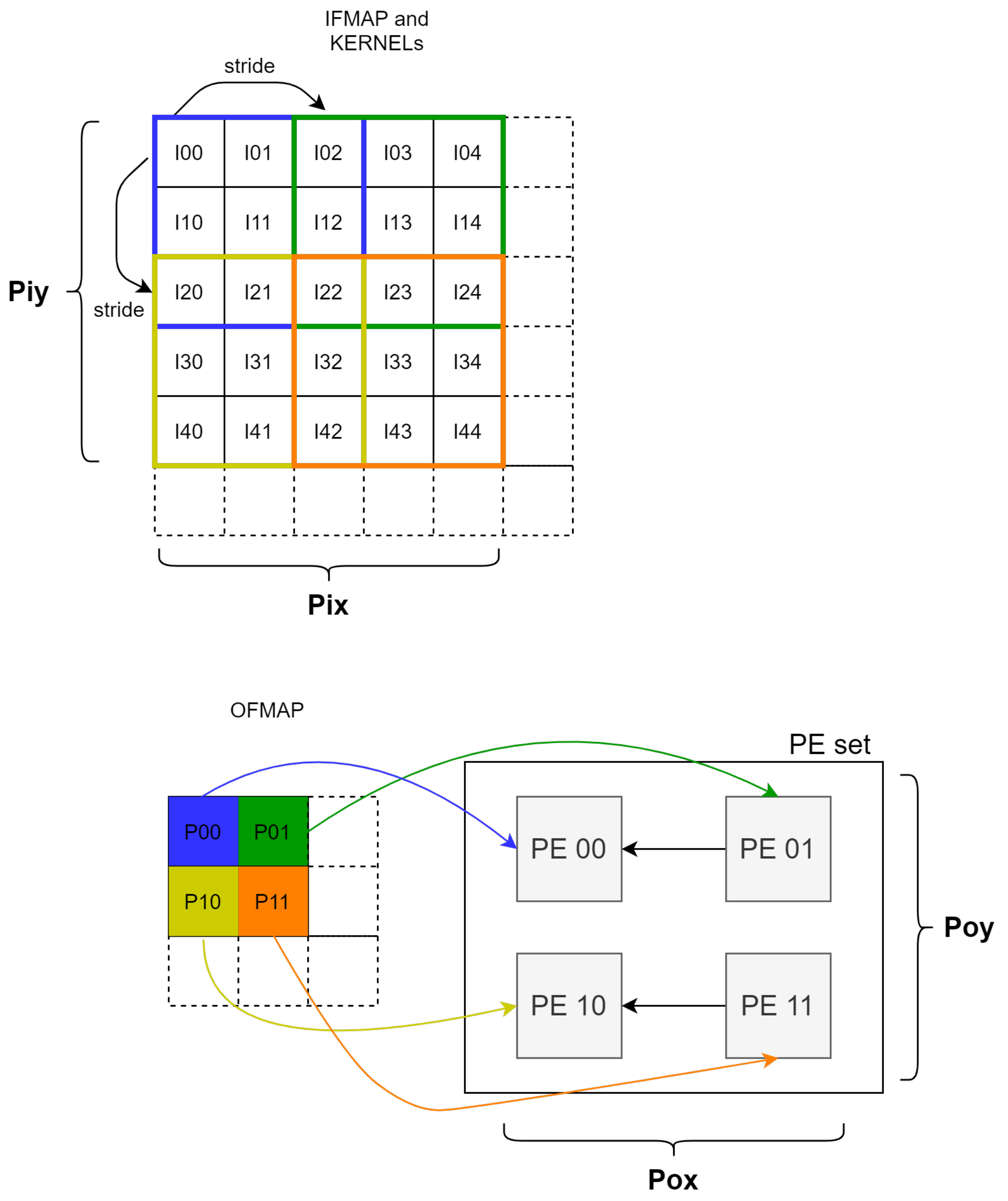

Figure 5.2.: Mapping of the computation in a 2x2 PE set (Pox=Poy=2). The different colors of the sliding windows (3x3 kernel) are associated to a different overlap with the ifmap, so the four psums of the ofmap

It is worth to highlight that there are 3 pixels that can be shared through the n2n communication among each couple of adjacent PEs in a row:

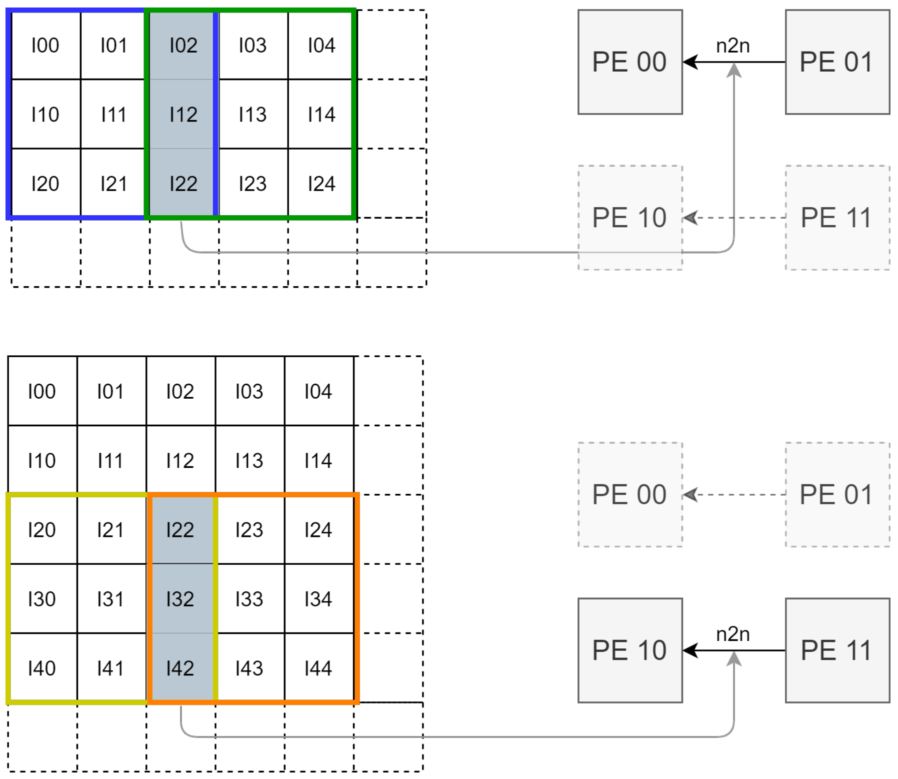

Figure 5.3.: The kernel size and the stride make sure that the second column of pixels in grey are in common between sliding windows of each row, thus PE01 and PE11 can send that pixel column to PE00 and PE10 respectively, through a direct point-to-point communication, without using the Interconnect

Hence, the *data dependency* is only along a row of PE set, while the vertical n2n connection is never used in the OS dataflow. In this case, the 1D convolution primitive [12] is strictly related to the horizontal unrolling factors (besides the kernel dimensions and the stride), which establishes the number of sliding windows (i.e. kernels) & corresponding ifmap beneath that can be settled into the PEs; then this schedule is repeated as many times as the vertical unrolling factor in order to fully exploit the HW resources in the spatial array.

If looking to the first sliding window of the PE set row in the example 5.3, it is possible to identify an important parameter and its complementary:

$$overlap = max\{0, T_{kx} - stride\} \tag{5.1}$$

$$not\_overlap = T_{kx} - max\{0, T_{kx} - stride\} \tag{5.2}$$

In that example, the *overlap* quantity is 2 while the *not_overlap* quantity is 1 and corresponds to the column of pixels that can be reused, by exploiting a n2n communication. The usage of these quantities will be more clear in next examples.

A further enhancement consists in mapping more than one output channel in a PE set, by interleaving the computation of a psum of a channel with other psums of next channels. In this way the ifmap reuse is maximized but the number of weights to be sent in a PE is higher because the PE needs multiple weights from different set of filters.
When considering the interleaving parameter p, the OS dataflow is called *OS Smart (OSS) dataflow*.

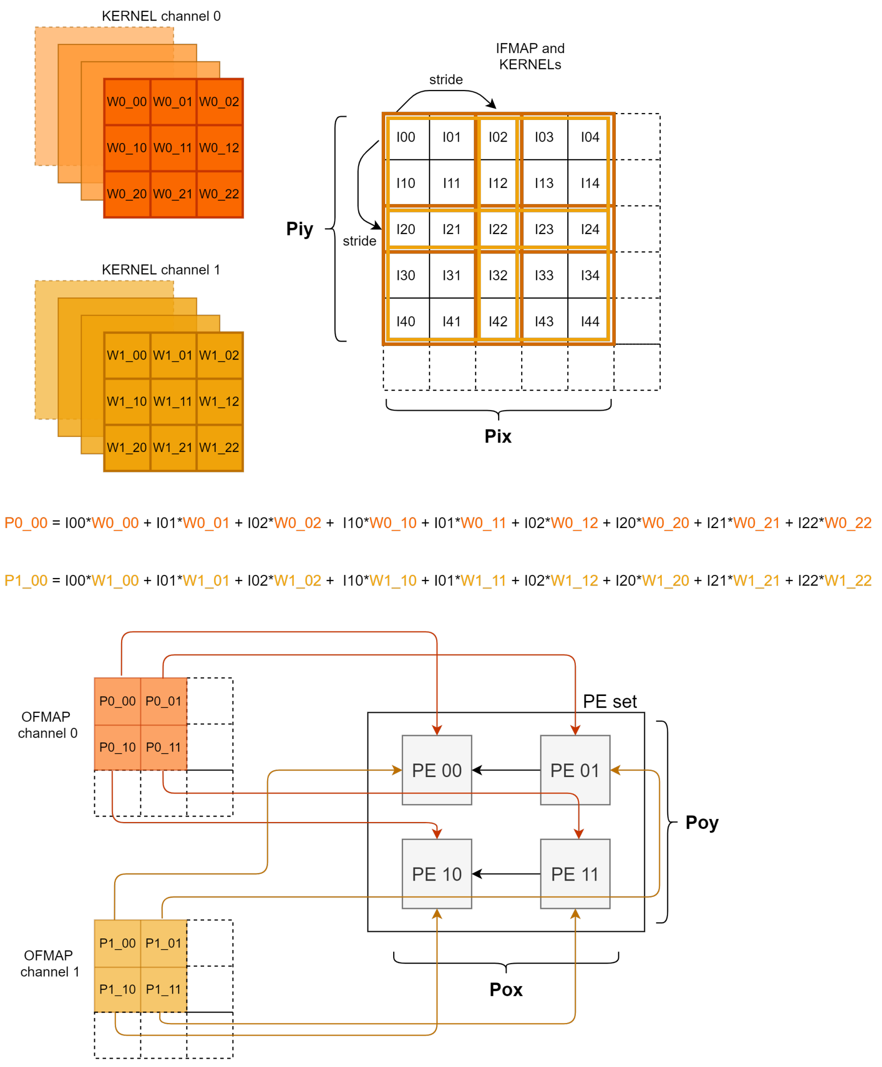

Figure 5.4.: Mapping of the computation with OSS dataflow.

In this example, the two colors are associated to different channels and the computation of two psums in PE00 is reported; the data written in the equations are meant to be in the corresponding RF. it is noticeable the reuse of input pixel when interleaving P0_00 (i.e. psum of channel 0 at position (0,0) in the ofmap) with P1_00, although the number of weights is doubled with respect to the standard case (i.e. computation of a single output channel).

Smart scan of CONV layer Once applied loop unrolling and loop tiling to the nested convolutional loop, it is already possible to obtain a schedule of jobs for the HW architecture. Furthermore, the list of data hold in every PEs needs to be scheduled in such a way that the control logic, at HW level, is not much complicated and allows a simple managing of the accesses to the RFs.
The ping pong buffer represents a suitable solution for handling n2n communication and ifmap reuse, since that one buffer is used to hold the pixels corresponding to the portion not overlapped with the adjacent sliding window, whereas the pixels of to the overlapped portion are sent to the other buffer.
This means that the PE processes first the pixel belonging the the not overlapped portion of the ifmap, then those of the overlapped section.
The next examples show the schedule of the jobs obtained from this *smart scan* of CONV layer for the OS dataflow. Every time, a PE set row is taken into account and all the reasoning on it can be applied in the same way to the next row of PEs. Different colors are associated to the allocation of the pixel in the ping or pong buffer.

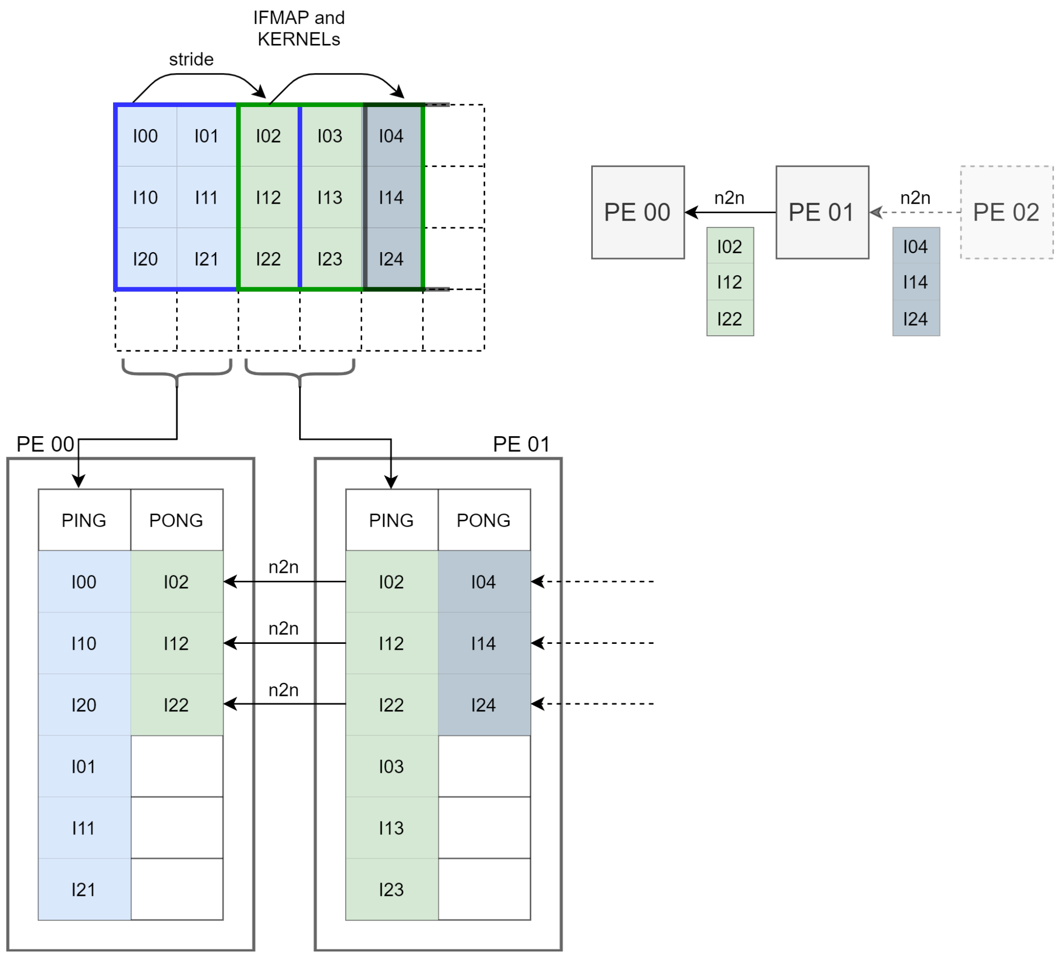

Figure 5.5.: Mapping of a 3xT_{ix} ifmap with a 3x3 kernel, stride = 2.

The number of the effective memory locations in the ping or pong buffer depends directly by the number of pixels in the columns of ifmap, which are: sent by the Interconnect before the computation starts; received from a neighbour PE.
With this mechanism, it is better to scan a ifmap column by column, rather than the usual row by row, and interleaving at every cycle the weights of different set of filters with the same ifmap in order to compute psums along the channel dimension. This type of iteration is critical especially when the stride is equal to 1, as depicted in the following example:

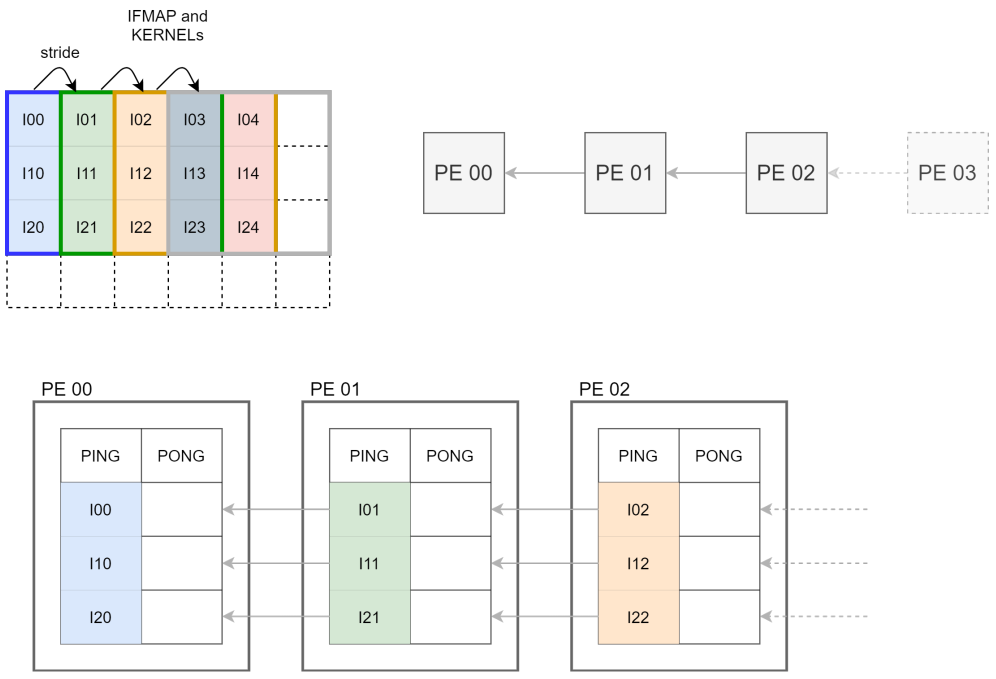

Figure 5.6.: Mapping of a 3xT_{ix} ifmap with a 3x3 kernel, stride = 1. Load step (i.e. n2n still not used)

Every ping buffer holds just 1 column of pixel that corresponds to the not overlap section, whereas the pong buffer is empty at the beginning. When the computation starts, the neighbour PEs fill up the pong buffer with the next column of pixel of the corresponding overlapped section:

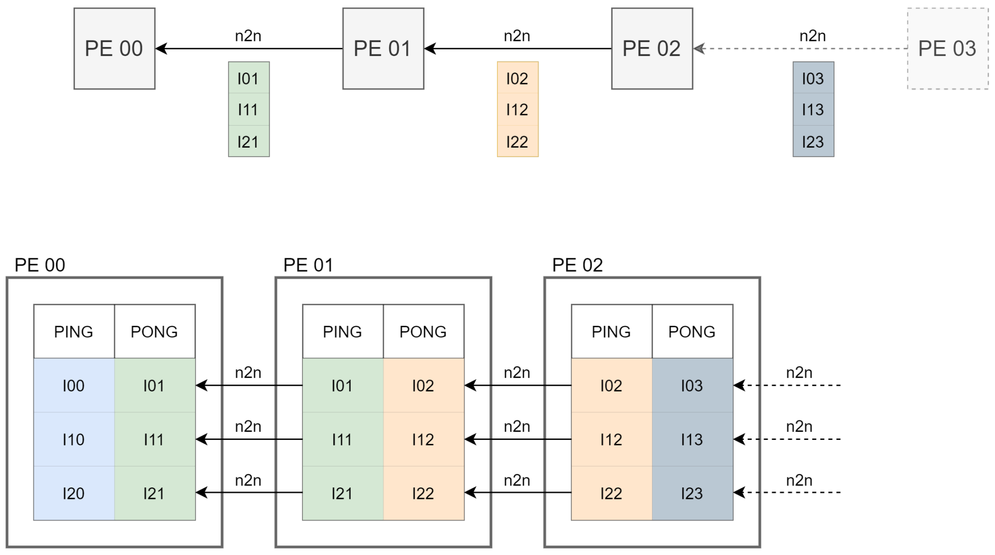

Figure 5.7.: The correct second columns of pixel are sent through n2n communication.

At this point the every PE need to process the third column of pixel: again, the PE receives this last column by its neighbour PE on the right. Since that the data in the ping buffer are already used, these incoming columns of pixel are stored in the ping buffer by overwriting the useless pixel:

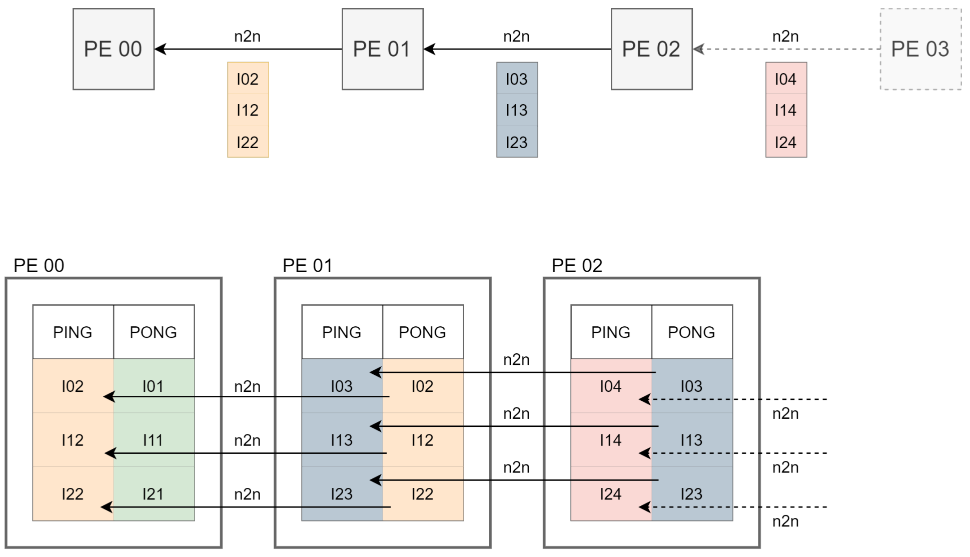

Figure 5.8.: Point-to-point communication for sending the last columns of data.

If the kernel was larger on its width, e.g. 3x6 instead of 3x3, the n2n communication would continue in the same way as before by sending columns of input pixel to the ping or pong buffer according to its switch position.

When the stride is 3, in the same example, the situation is easier because there is no overlap among adjacent sliding windows and the n2n communication never happens, so the schedule provides only MAC operations.

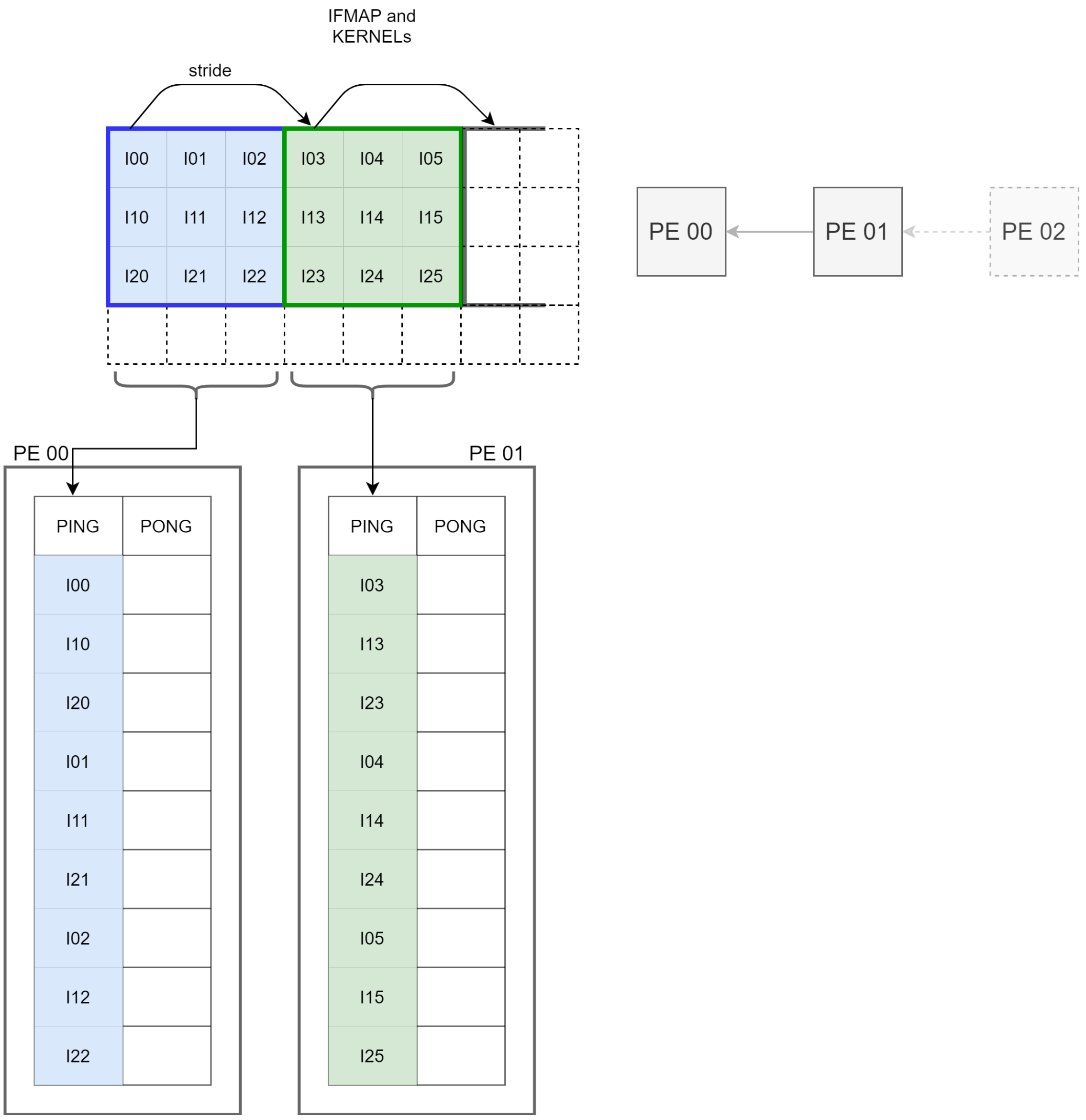

Figure 5.9.: Mapping of a 3xX ifmap with a 3x3 kernel, stride = 3. Load step

The scheduling algorithm, after applying loop scheduling techniques, is showed below:

```
for (no=0; no<Tof-(p-1); no+=p)
    init_PEset
```

```
for (y=0; y<Toy-(Poy-1); y+=Poy)
    for (x=0; x<Tox-(Pox-1); x+=Pox)
        for (ni=0; ni<Tif; ni++)
            load_data
            for (kx=0; kx<Tkx; kx++)
                for (ky=0; ky<Tky; ky++)
                    for (i=0; i<p; i++)
                        for (py=0; py<Poy; py++)
                            for (px=0; px<Pox; px++)
                                update_RF
                                MAC
            if (ni==Tif-1)
                send_output
            schedule_n2n
update_PEset_history
```

Listing 5.1: Scheduling algorithm pseudo-code

The concept of *PE set history*, i.e. reuse of a logical PE set, is explained in next sections.

RF dimensions The ifmap RF requires at least $T_{kx} \cdot T_{ky}$ locations in order to face all possible situations like in 5.9.
For what concerns the RFs for the weights and psums, the structure is always the same in all PEs:

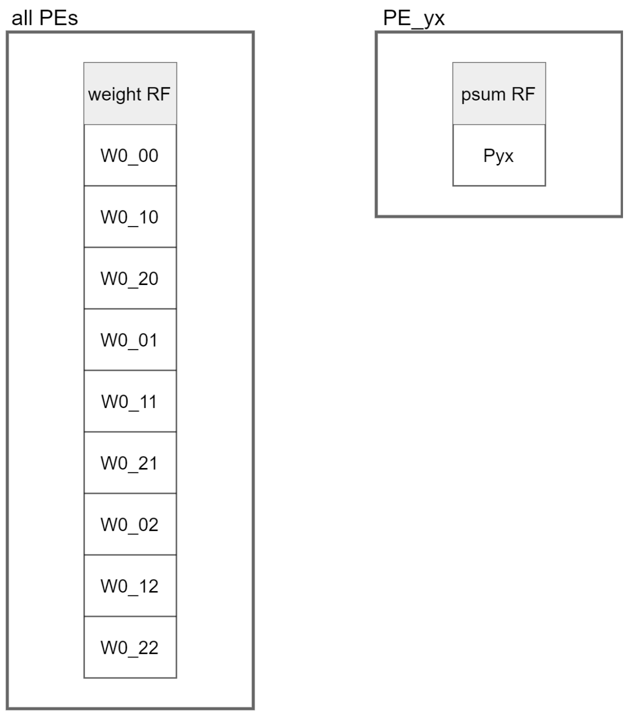

Figure 5.10.: RF size for the weights and psums (the interleaving parameter is equal to 1 in this case)

The weight RF holds all the weights in the 3x3 kernel used in the convolution considered in the PE array, so if $p > 1$ the RF grows up up to $p \cdot T_{kx} \cdot T_{ky}$.
The weights stored in this RF are the same in all PEs of the PE set, so the dispatch of the weights to the array can be accomplished through a Multicast, whereas the deliver of ifmap needs a Unicast because every RF stores input pixels different from each other.
The number of locations in the psum RF is at least equal to p.

OSS dataflow - version 2 To sum up, this algorithm provides the schedule of jobs supporting the OS_B dataflow. In this case the other interleaving parameter q, related to the input channels, is equal to 1, which means that PE sets processes tiles of ifmap one at a time.
Hence the communication between GLB and PE array is determined by T_{if}, because the NoC has to dispatch ifmap and kernel of a new channel at the beginning of the computation. Although the number of memory accesses could be very large if T_{if} is big, the size of the SPAD is small since that the ping pong buffer is limited to few ifmap pixels.
To reduce the accesses to the GLB, it is possible to consider $q > 1$ and load multiple input channels of kernels and ifmap into the SPAD.
The principle of the algorithm remains the same: the ifmap columns of different

channels are processed first, then the computation proceeds with the next set of columns.

```
for (no=0; no<Tof-(p-1); no+=p)
    init_PEset
    for (y=0; y<Toy-(Poy-1); y+=Poy)
        for (x=0; x<Tox-(Pox-1); x+=Pox)
            load_data
            for (kx=0; kx<Tkx; kx++)
                for (ni=0; ni<Tif; ni++)
                    for (ky=0; ky<Tky; ky++)
                        for (i=0; i<p; i++)
                            for (py=0; py<Poy; py++)
                                for (px=0; px<Pox; px++)
                                    update_RF
                                    MAC
            send_output
            schedule_n2n
    update_PEset_history
```

Listing 5.2: Second version of the Scheduling algorithm

The drawback of this new version of OS dataflow is that the RF size of ifmap and weight increases dramatically due to the introduction of T_{if} channels in both registers:

Virtual neighbour To have a homogenous movement of data inside the array in the point-to-point connection, the rightmost PE of the row in the PE set needs a *virtual neighbour* from which to receive columns of pixel. The horizontal Interconnect of that PE can be exploited for this job:

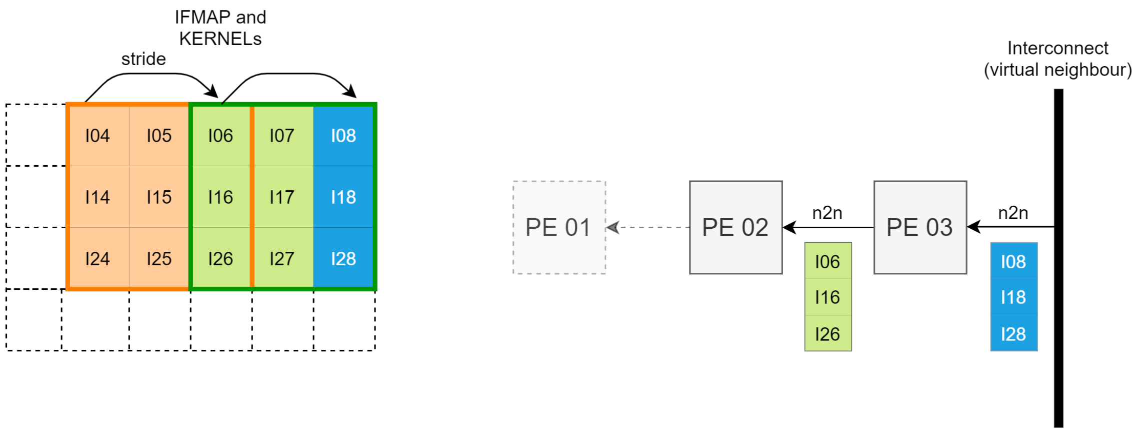

Figure 5.11.: Mapping of a 3x9 ifmap with a 3x3 kernel, stride = 2. The sliding windows reached the column edge.

In this way all the PEs, in the last column on the right in the PE set, can have the same pattern of data and RF requirements of the other PEs.

Program code generation The scheduling algorithm determines the type of iteration over the input layer shapes, compliant with the OSS dataflow, to map the computation in a PE array.
Moreover, the Compiler needs to translate the necessary operations into *jobs* or *atomic instructions* that implements the CNN execution at HW level. To do that, some commands are introduced in the algorithm in order to generate a proper Program Code, just as showed in the pseudo codes of the algorithms (5.1.1 and 5.1.1), which represents a raw version of the instructions for the HW architecture. It is important to notice that the commands are placed in specific locations of the scheduling algorithm according to its HW meaning, so the initialization and update phase of a logical PE set happens at specific moment of the execution to provide the right schedule of micro-instructions.

Cycle count (MAC)	**west n2n**	**PE 01**	**east n2n**
		load_data	
cycle 0	sendI[0,0,2]	O[0,0,1]+=I[0,0,2]*W[0,0,0,0]	receiveI[0,0,4]
cycle 1	sendI[0,1,2]	O[0,0,1]+=I[0,1,2]*W[0,0,1,0]	receiveI[0,1,4]
cycle 2	sendI[0,2,2]	O[0,0,1]+=I[0,2,2]*W[0,0,2,0]	receiveI[0,2,4]
cycle 3		O[0,0,1]+=I[0,0,3]*W[0,0,0,1]	
cycle 4		O[0,0,1]+=I[0,1,3]*W[0,0,1,1]	
...		...	
		(send_output)	

Table 5.1.: Schedule of the atomic instructions of 5.5

Cycle count (MAC)	**west n2n**	**PE 01**	**east n2n**
		load_data	
cycle 0	sendI[0,0,1]	O[0,0,1]+=I[0,0,1]*W[0,0,0,0]	receiveI[0,0,2]
cycle 1	sendI[0,1,1]	O[0,0,1]+=I[0,1,1]*W[0,0,1,0]	receiveI[0,1,2]
cycle 2	sendI[0,2,1]	O[0,0,1]+=I[0,2,1]*W[0,0,2,0]	receiveI[0,2,2]
cycle 3	sendI[0,0,2]	O[0,0,1]+=I[0,0,2]*W[0,0,0,1]	receiveI[0,0,3]
cycle 4	sendI[0,1,2]	O[0,0,1]+=I[0,1,2]*W[0,0,1,1]	receiveI[0,1,3]
cycle 5	sendI[0,2,2]	O[0,0,1]+=I[0,2,2]*W[0,0,2,1]	receiveI[0,2,3]
cycle 6		O[0,0,1]+=I[0,0,3]*W[0,0,0,2]	
...		...	
		(send_output)	

Table 5.2.: Schedule of the atomic instructions of 5.6

Cycle count (MAC)	west n2n	PE 01	east n2n
		load_data	
cycle 0		O[0,0,1]+=I[0,0,3]*W[0,0,0,0]	
cycle 1		O[0,0,1]+=I[0,1,3]*W[0,0,1,0]	
cycle 2		O[0,0,1]+=I[0,2,3]*W[0,0,2,0]	
cycle 3		O[0,0,1]+=I[0,0,4]*W[0,0,0,1]	
...		...	
		(send_output)	

Table 5.3.: Schedule of the atomic instructions of 5.9

Schedule of operations Previous examples show that the execution of the atomic instructions, e.g. MAC or send/receive data through n2n, requires a synchronization between all PEs in the PE set rows. Indeed there is data dependency horizontally in the spatial array due to data reuse opportunities.
This uniform movement of data in the spatial, for the OS dataflow, brings to a schedule of jobs that requires a regular pattern of MAC and n2n operation.

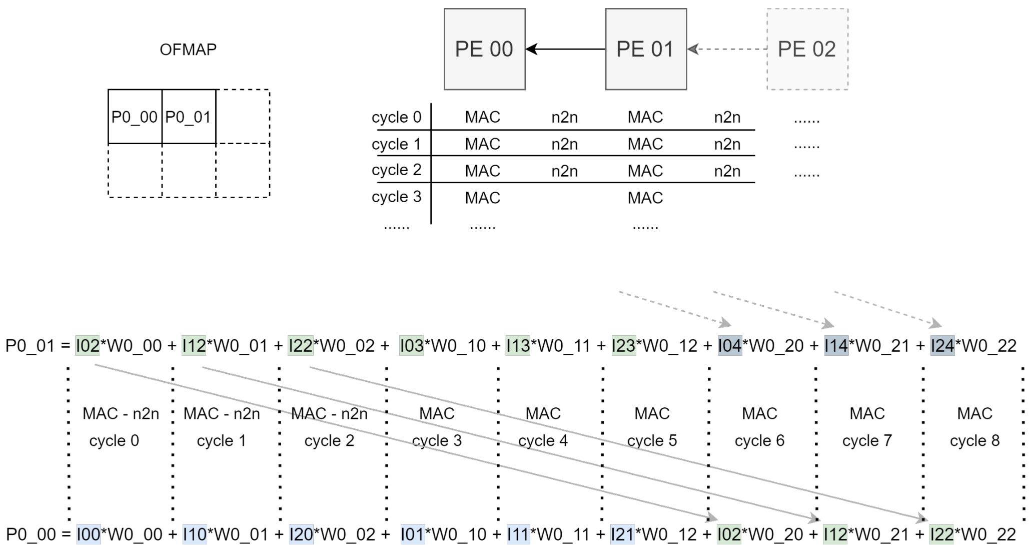
OFMAP
P0_00
P0_01
PE 00
PE 01
PE 02
cycle 0 MAC n2n MAC n2n ......
cycle 1 MAC n2n MAC n2n ......
cycle 2 MAC n2n MAC n2n ......
cycle 3 MAC MAC
....
P0_01 = I02*W0_00 + I12*W0_01 + I22*W0_02 + I03*W0_10 + I13*W0_11 + I23*W0_12 + I04*W0_20 + I14*W0_21 + I24*W0_22
MAC - n2n cycle 0
MAC - n2n cycle 1
MAC - n2n cycle 2
MAC cycle 3
MAC cycle 4
MAC cycle 5
MAC cycle 6
MAC cycle 7
MAC cycle 8
P0_00 = I00*W0_00 + I10*W0_01 + I20*W0_02 + I01*W0_10 + I11*W0_11 + I21*W0_12 + I02*W0_20 + I12*W0_21 + I22*W0_22

Figure 5.12.: Schedule of 5.5

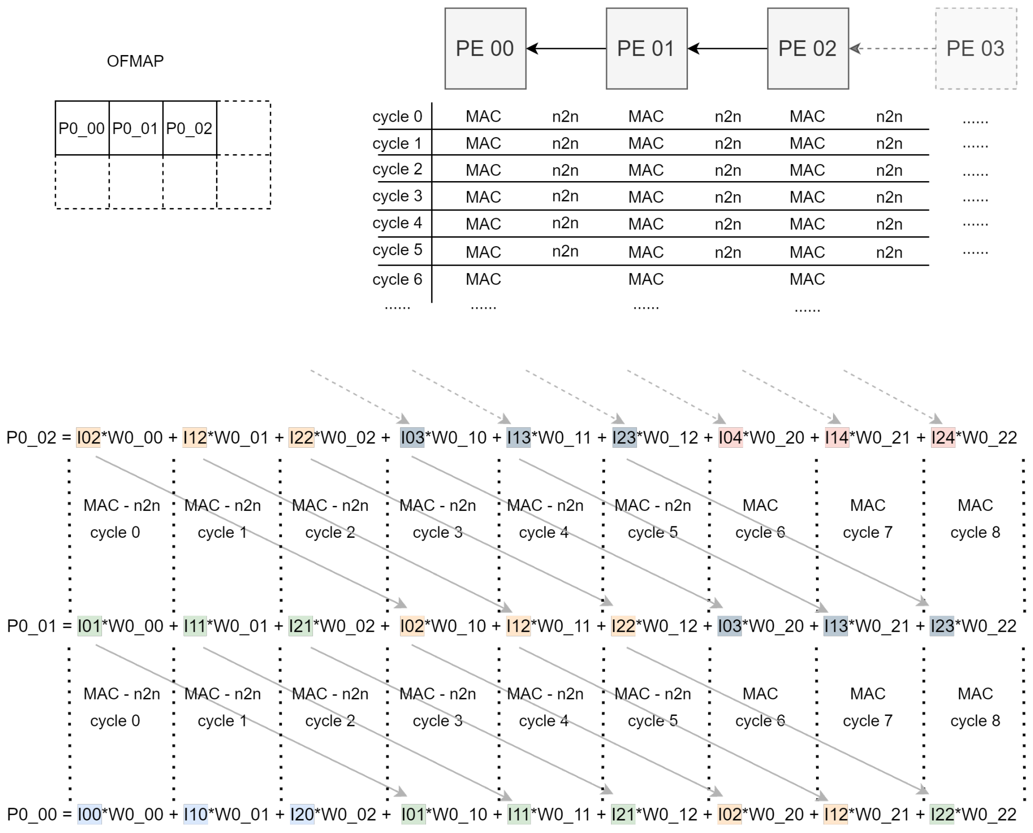
OFMAP
P0_00
P0_01
P0_02
PE 00
PE 01
PE 02
PE 03
cycle 0
cycle 1
cycle 2
cycle 3
cycle 4
cycle 5
cycle 6
MAC
n2n
P0_02 = I02*W0_00 + I12*W0_01 + I22*W0_02 + I03*W0_10 + I13*W0_11 + I23*W0_12 + I04*W0_20 + I14*W0_21 + I24*W0_22
P0_01 = I01*W0_00 + I11*W0_01 + I21*W0_02 + I02*W0_10 + I12*W0_11 + I22*W0_12 + I03*W0_20 + I13*W0_21 + I23*W0_22
P0_00 = I00*W0_00 + I10*W0_01 + I20*W0_02 + I01*W0_10 + I11*W0_11 + I21*W0_12 + I02*W0_20 + I12*W0_21 + I22*W0_22
MAC - n2n
cycle 7
cycle 8

Figure 5.13.: Schedule of 5.6

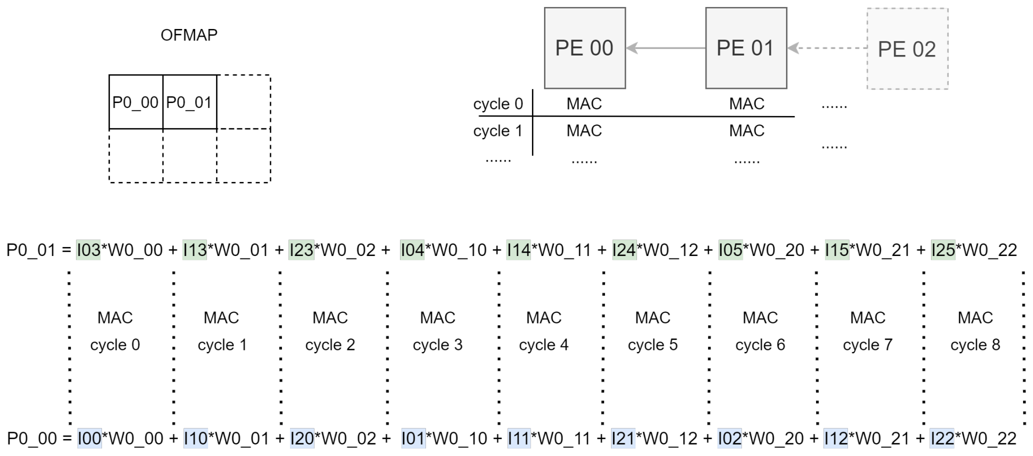
OFMAP
P0_00
P0_01
PE 00
PE 01
PE 02
cycle 0
cycle 1
MAC
P0_01 = I03*W0_00 + I13*W0_01 + I23*W0_02 + I04*W0_10 + I14*W0_11 + I24*W0_12 + I05*W0_20 + I15*W0_21 + I25*W0_22
P0_00 = I00*W0_00 + I10*W0_01 + I20*W0_02 + I01*W0_10 + I11*W0_11 + I21*W0_12 + I02*W0_20 + I12*W0_21 + I22*W0_22
cycle 2
cycle 3
cycle 4
cycle 5
cycle 6
cycle 7
cycle 8

Figure 5.14.: Schedule of 5.9

In these examples, the interleaving parameter p does not occur but, if it does, the main change is in the schedule of n2n:

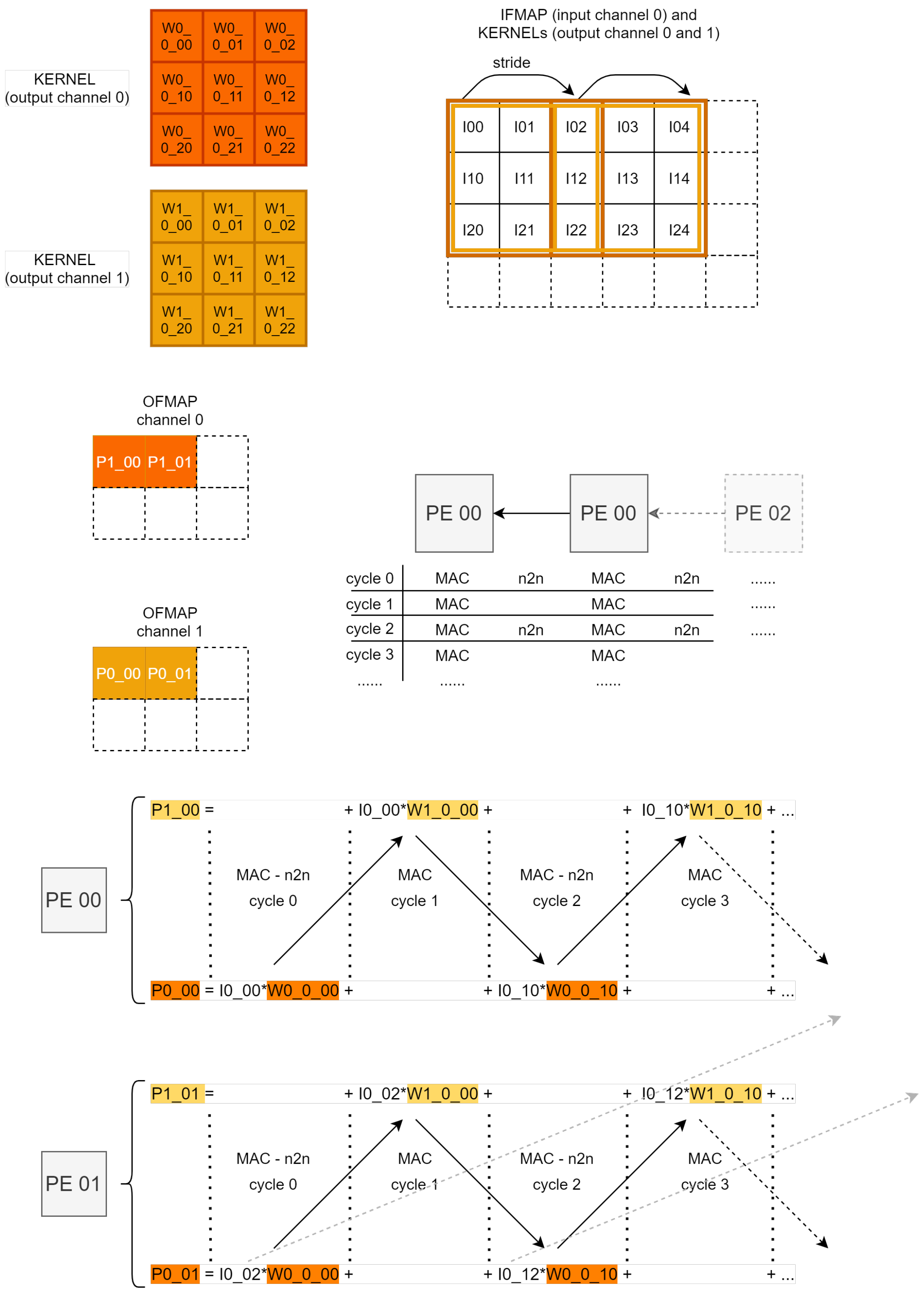

Figure 5.15.: Schedule of 5.5 with p=2. Here different colors refer to different output channels

The computation is interleaved at every cycle with psums of different output channels. In this way the ifmap is reused immediately and it can be overwritten by the data received on n2n in the next cycles.

5.1.2. Corner cases

The unrolling factors Pox and Poy allow to make parallel the execution of multiple psums by exploiting the available HW resources. Nevertheless, the maximum and affordable number of PEs is limited to a certain amount depending on the HW constraints and, in some cases, in the PE array there might be several PE sets with few PEs that cannot compute a large ofmap.
The Scheduling algorithm must recognize and handle this unusual situation in which the sequence of operations is no more regular and the mapping results being different. The following conditions detects possible corner cases in the mapping process:

1. $P_{ox,y} < T_{ox,y}$: the logical PE set is reused more than one time with different data since that the number of available PEs in the PE set cannot process the entire ofmap at the same time.

2. $T_{ox,y} \% P_{ox,y} \neq 0$: when the sliding window of the convolution reaches the edge along the row/column of the ifmap, not all PEs are working at the same time; this requires to introduce some stalls in the schedule in order to turn-off the functionality of some PEs when this corner case occurs.

3. $T_{of} \% p \neq 0$ The interleaving factor considered in the schedule of the computation is not anymore equal to p but to 1: this may happen in the PE set responsible for the computation of the last output channel, so there are also less MACs to do than the standard case

When some of these situations comes out, the starting and ending indexes employed in the scheduling algorithm are changed to new indexes that resume correctly to computation from the position that has been halted.

Edge cases on the 2D ofmap The first and second type corner cases, mentioned above, happens in a situation like the following:

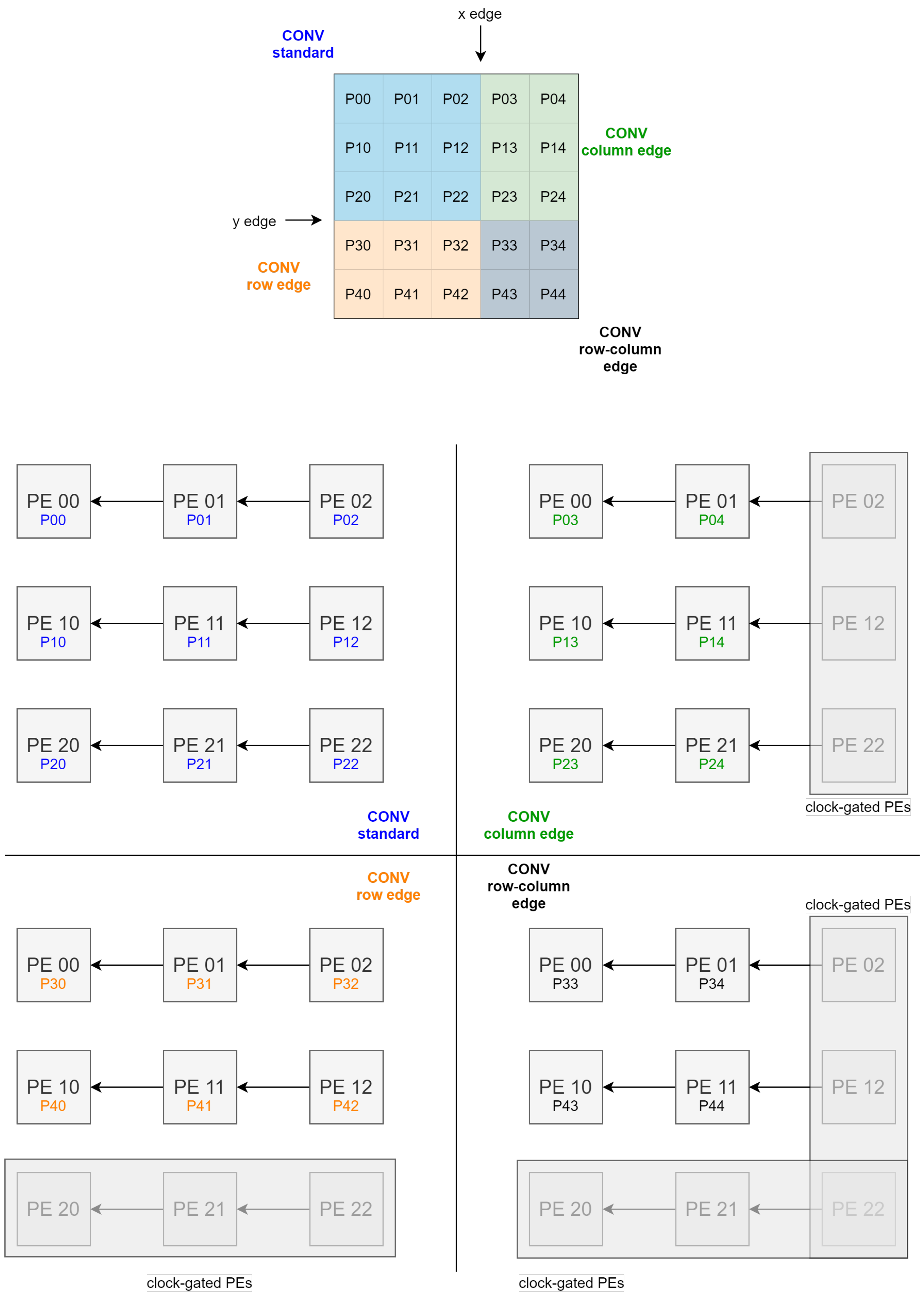
x edge
CONV standard
CONV column edge
y edge
CONV row edge
CONV row-column edge
P00 P01 P02 P03 P04
P10 P11 P12 P13 P14
P20 P21 P22 P23 P24
P30 P31 P32 P33 P34
P40 P41 P42 P43 P44
PE 00 P00, PE 01 P01, PE 02 P02
PE 10 P10, PE 11 P11, PE 12 P12
PE 20 P20, PE 21 P21, PE 22 P22
PE 00 P03, PE 01 P04, PE 02
PE 10 P13, PE 11 P14, PE 12
PE 20 P23, PE 21 P24, PE 22
PE 00 P30, PE 01 P31, PE 02 P32
PE 10 P40, PE 11 P41, PE 12 P42
PE 20, PE 21, PE 22
PE 00 P33, PE 01 P34, PE 02
PE 10 P43, PE 11 P44, PE 12
PE 20, PE 21, PE 22
clock-gated PEs

Figure 5.16.: Coverage of an ofmap and handling of all corner cases in the PE set

The example under consideration takes a 11x11 ifmap, 3x3 kernel, stride=2 and tries to map the computation of the 5x5 ofmap in a 3x3 PE set (Pox=Poy=3).
What happens is that, when the computation approaches the edge on the row or column of the ofmap, there is a row or column of PEs that can be turned off since that there is no psum to map on them.
This situations can be generalized to larger PE sets (i.e. higher unrolling factors and bigger ofmap) where there might be more than one row or column turned off, or just one instead that three corner cases.

Edge cases on the ofmap channels In the third type of corner cases, there is no more uniformity of schedule in all PE sets because the last one deals only the last output channel, so p is forced to 1 and the number of processed data is lower.

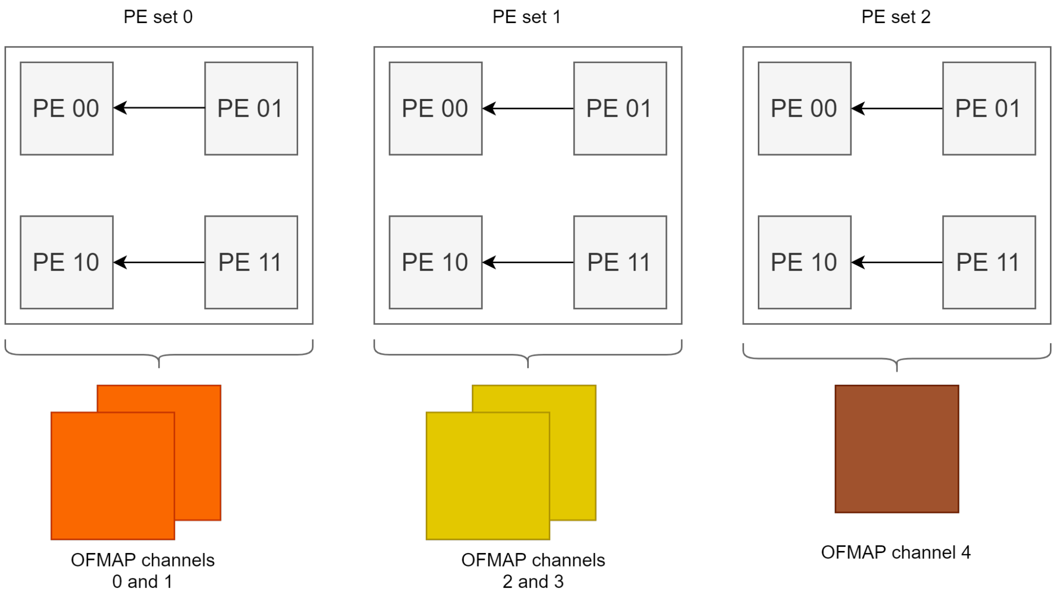

Figure 5.17.: Mapping of 5 output channels in 3 logical PE sets, having p=2. Here the colors are associated to the same PE set that computes the output channels.

Reuse of PE set A *PE set history* is a data structure that holds the schedule of the convolution corresponding to the number of times the same PE set is reused, because of the process of the input channels or the corner cases. Indeed, when a PE set is reused, it needs new data to process even if the sequence of operations is pretty much the same.
For instance, in 5.16 the PE set history is equal to 4 because the same logical PE set is reused 4 times. It is obvious that, if the number of physical PE sets in the is 4, each PE set of the history can be mapped in the array. The concept of history is only a way to group logical PE sets evenly.

5.1.3. Physical mapping in the array

From 5.17 it comes out that the number of logical PE sets is equal to $\lfloor T_{of}/p \rfloor + T_{of}\%p$. Then the schedule for these ideal PE sets needs to be mapped into the real PE array, which could not fit all the PE sets at the same time because of HW resource limitations.
The physical mapping exploits the parameters P_{of} *vertical* and P_{of} *horizontal* provided by HWflow to organize the schedule of the instructions of every PE sets in a sequential order. Hence, the possible physical number of PE sets in the spatial array is equal to the number of logical PE sets divided by (Pofh + Pofv).

$$logical\_PEset = \lfloor T_{of}/p \rfloor + T_{of}\%p \tag{5.3}$$

$$physical\_PEset = \frac{logical\_PEset}{P_{ofh} + P_{ofv}} \tag{5.4}$$

5.2. Data Pattern

The *Figure of Merit* provided by the Scheduling algorithm of the Compiler are the following ones:

1. Schedule of jobs or atomic instructions for every PE
2. All data in the RF involved in the convolution operation
3. Data reused in the neighbour-to-neighbour communication

From this information it is possible to identify a specific *data pattern*, which determines the way each data is presented to the PE:

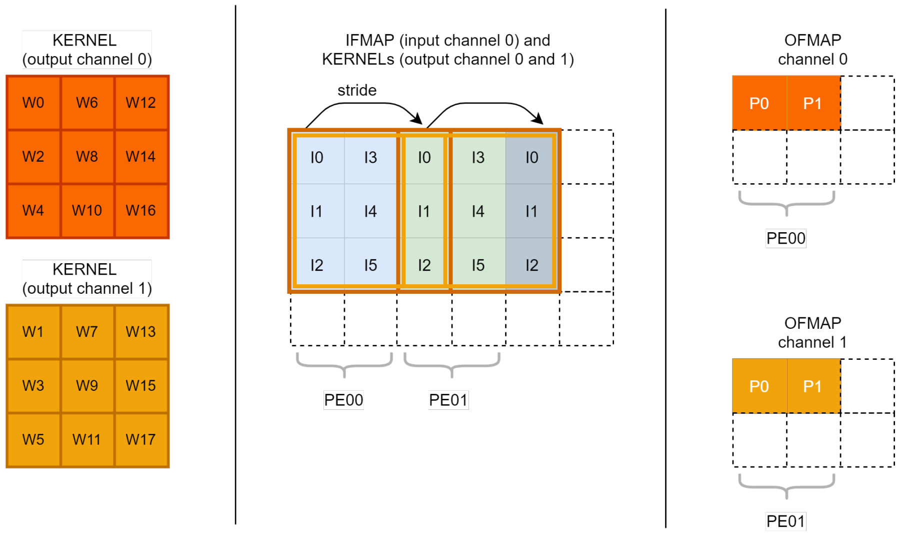

Figure 5.18.: Flattening of data of 5.5 but with p=2.

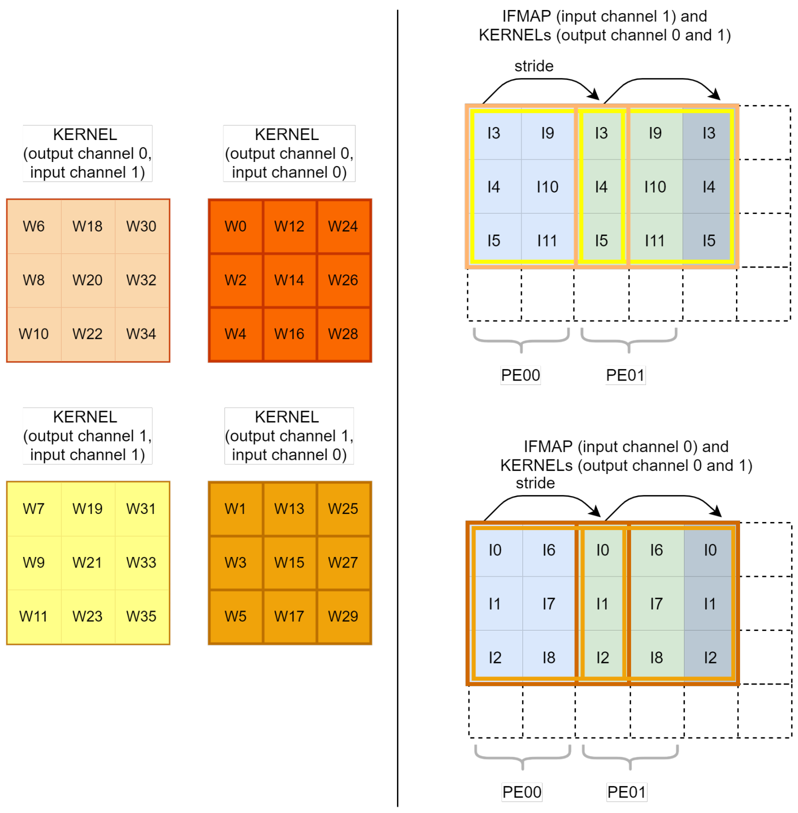
KERNEL (output channel 0, input channel 1)
W6 W18 W30
W8 W20 W32
W10 W22 W34
KERNEL (output channel 0, input channel 0)
W0 W12 W24
W2 W14 W26
W4 W16 W28
KERNEL (output channel 1, input channel 1)
W7 W19 W31
W9 W21 W33
W11 W23 W35
KERNEL (output channel 1, input channel 0)
W1 W13 W25
W3 W15 W27
W5 W17 W29
IFMAP (input channel 1) and KERNELs (output channel 0 and 1)
stride
I3 I9 I3 I9 I3
I4 I10 I4 I10 I4
I5 I11 I5 I11 I5
PE00
PE01
IFMAP (input channel 0) and KERNELs (output channel 0 and 1)
stride
I0 I6 I0 I6 I0
I1 I7 I1 I7 I1
I2 I8 I2 I8 I2
PE00
PE01

Figure 5.19.: Flattening of data of 5.5 but with p=2 and q=2.

Therefore, the Scheduling algorithm flattens all data types in 1 dimension, putting them in a sequential order in every RF. In this way, the control logic at runtime is simplified because the further dimensions of the input data are not anymore considered during the processing step in the HW.
The drawback of having a specific data pattern for the OS dataflow is that the program code is not general purpose, so a compilation step is necessary every time the CNN changes.

Visualization of the OS Dataflow The regular and uniform data movement in a PE set row can be visualized in some charts, which reports the usage of the data sitting in the RF.
It is interesting to focus more on the ifmap data pattern, since that are the type of data involved in n2n communication. The data pattern for the psums and weights is pretty much the same in all the examples.

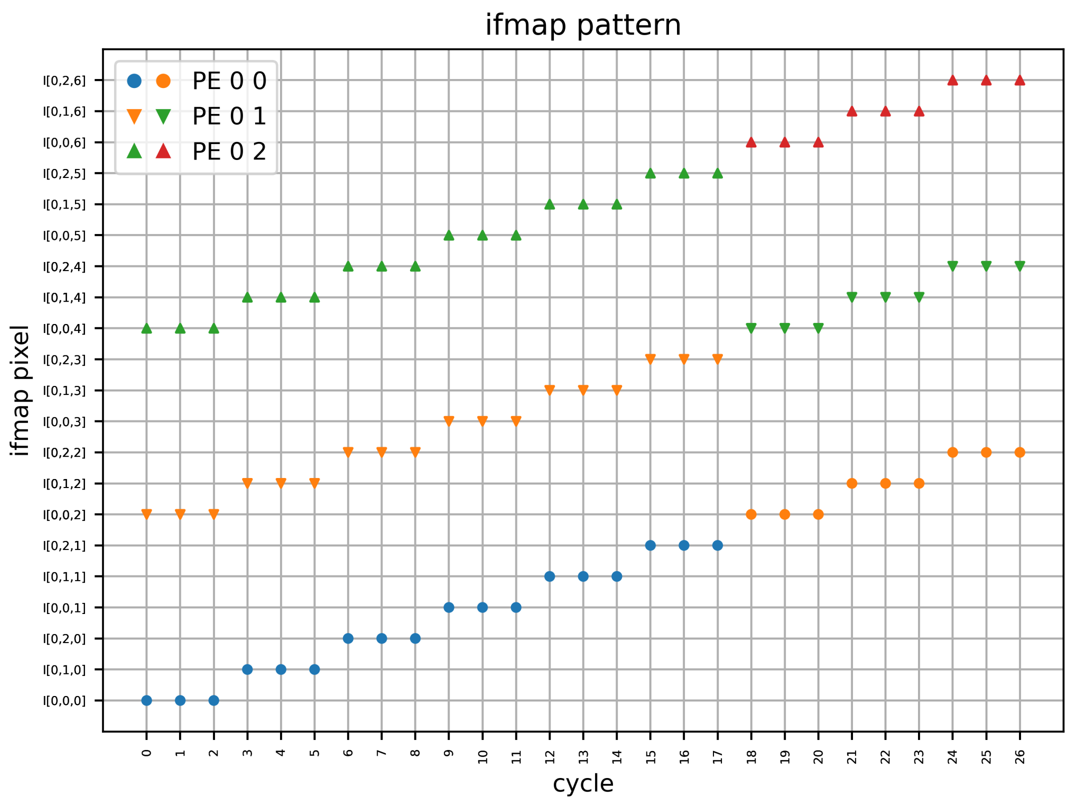

Figure 5.20.: Ifmap data pattern of 5.5 with p=3 and Pox=3. Different colors are associated to columns of ifmap pixel received from the Interconnect in the load stage, while different markers points out which PE of the PE set row is processing the data

If looking on the horizontal axis, the PEs of the PE set row retrieve the pixels in a sequential way from the RF and process them in parallel according to Pox: in this case, on each vertical line of the grid there are 3 pixels because the horizontal unrolling factor is equal to 3. The same pixel is also reused as many times as the parameter p, because the interleaving happens at every cycle.
On the vertical axis it is reported which pixel is processed at which cycle: the pattern follows the schedule of 5.12 but with p=3. Moreover it is possible to notice the n2n communication by looking at the pixels with same colors on the same row. The *overlap* quantity is 1, so the number of pixel sent through n2n is $overlap \cdot T_{ky} = 3$ just as showed in the plot.
The PE02 has the Interconnect as a virtual neighbour, thus the last 3 pixels have different colors with respect to the other ones of its same pattern (red instead that green), which means that they were not originally sent by the Interconnect in the load stage.

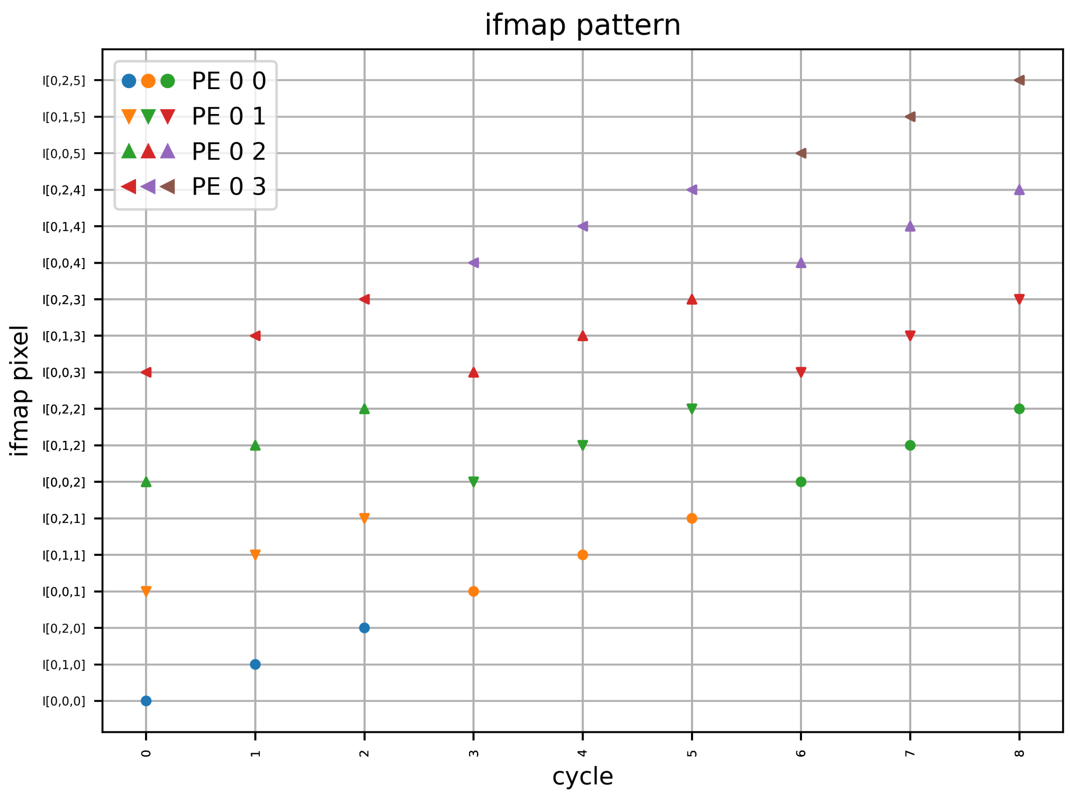

Figure 5.21.: Ifmap data pattern of 5.6 with p=1 and Pox=4.

The same previous considerations are valid for the example above, where the *overlap* quantity is 2 and there are multiple n2n communications one after the other.

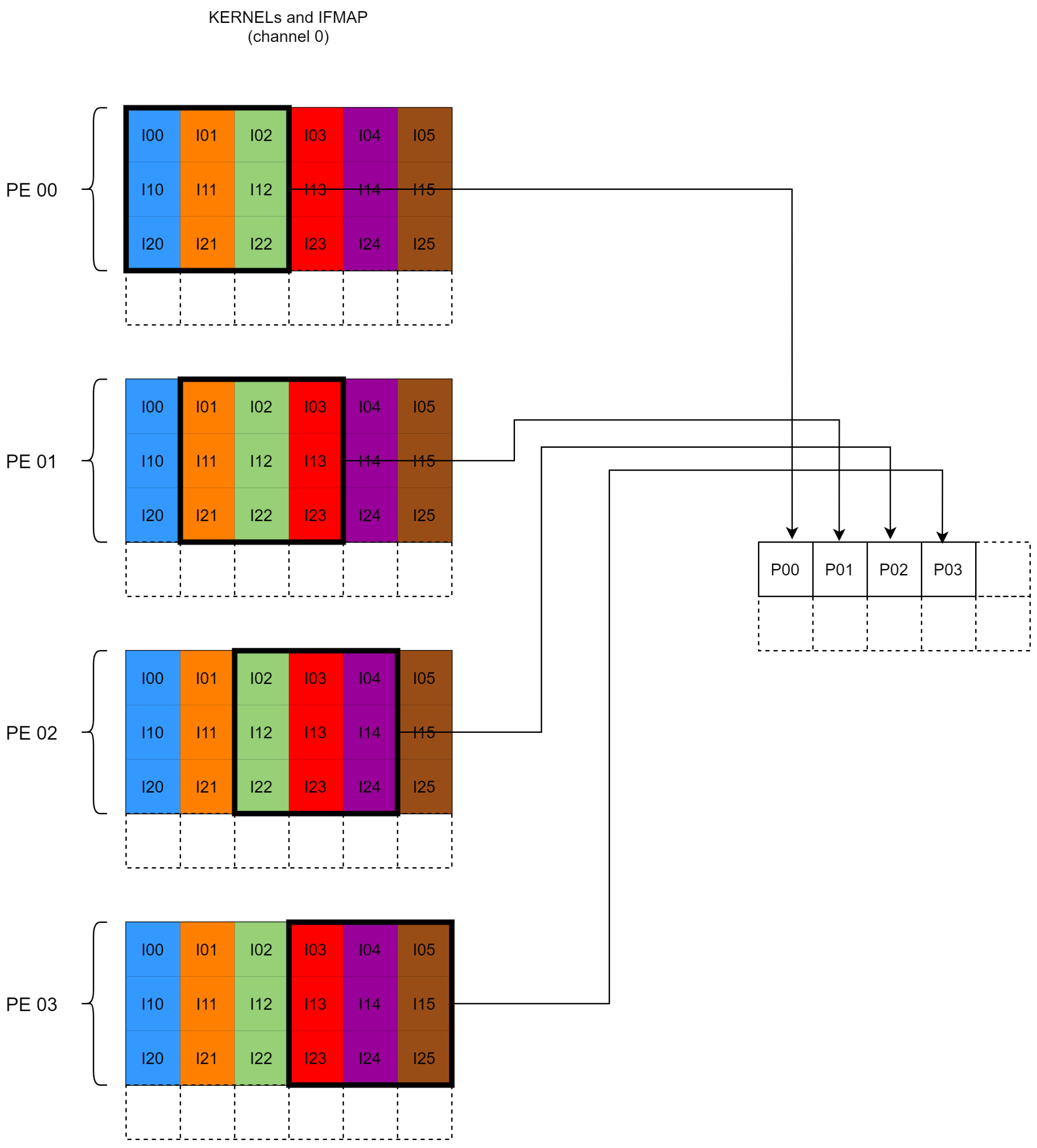

Figure 5.22.: Mapping of the sliding windows on each PE of the first PE set row for 5.6

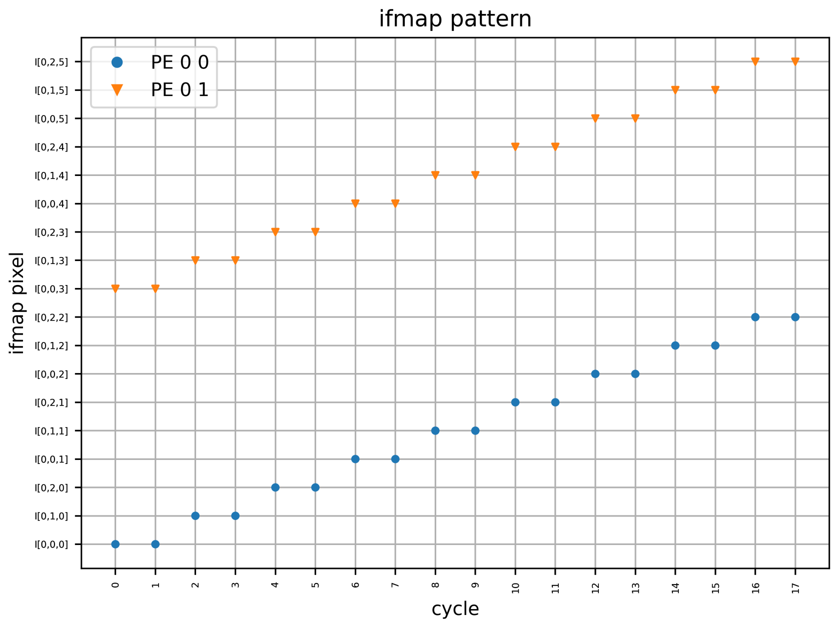

Figure 5.23.: Ifmap data pattern of 5.9 with p=3 and Pox=3.

This is the most simple data pattern because n2n never occurs.

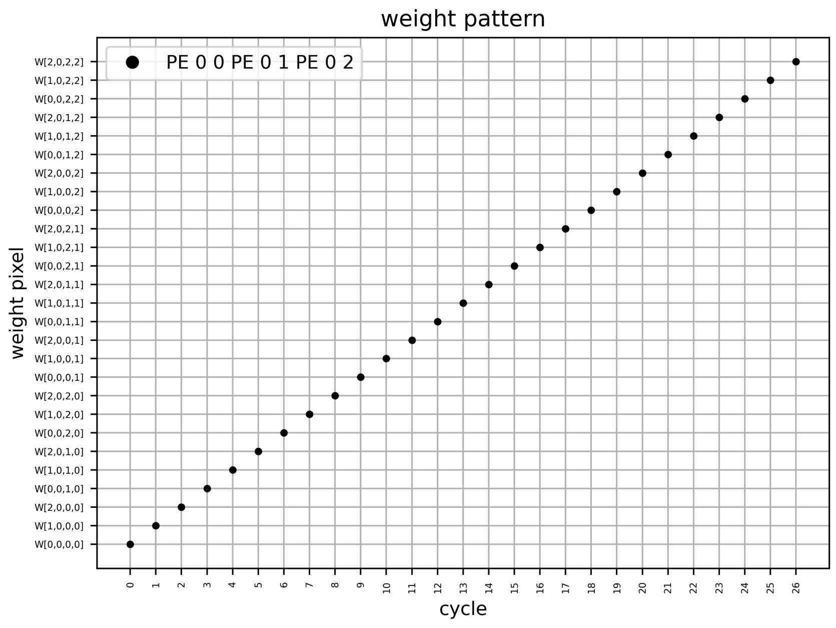

Figure 5.24.: Weight data pattern of 5.5 with p=2 and Pox=2.

The data pattern for the weight is just a sequence of points sitting on a straight line, showing the linear behavior of the fetching operation from the RF, which behaves as a FIFO.

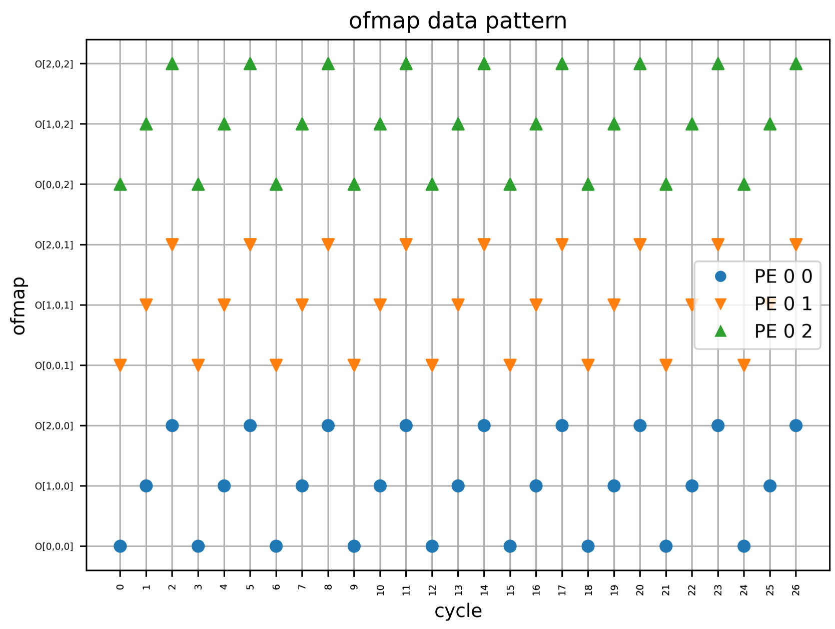

Figure 5.25.: Psum data pattern of 5.5 with p=3 and Pox=3.

The choice of the interleaving at every cycle causes this "zig-zag" trend of the ofmap pattern belonging to a PE. Being Pox=3, there are also 3 psums accumulated in parallel in the PE set row.

5.2.1. Categories of schedule

The data pattern of the OS dataflow (both version 1 and 2) establishes 3 types of schedule of atomic instructions that strictly depend on the kernel width and stride:

1. The *overlap* quantity is 0, which means that $Tkx \leq s$ and there is no n2n; every PE does the assigned number of MACs independently on the other ones, so data dependency along one row is not anymore relevant

2. $Tkx > s$ and there is just one overlapped column ($overlap = 1$), so the n2n communication happens just one time

3. $Tkx > s$ but there are several overlapped columns ($overlap > 1$), so the n2n communication occurs more than one time and there is an intensive usage of the ping pong buffer, synchronized with the ongoing MAC

PICTURES FOR ALL OF THE 3 CASES??

5.3. Instruction Set Architecture

The program code is made up of single instruction for each data and describes the sequence of jobs necessary to accomplish the convolution.
Anyway, it is not possible to directly employ these atomic instructions in the HW architecture, otherwise it would be expensive to dispatch so many instructions through a TDM interconnect and it would require a high bandwidth with the on-chip memory, which would bring a large power consumption.
Hence an encoding step at compile time is required to get an ISA able to unpack the jobs with as few as possible instructions (called also *messages*). By having an efficient ISA, a smart deployment in the spatial array can be also achieved.

5.3.1. Instruction encoding

The FOM identified by the data pattern describe the OS dataflow and the processing of data. One key feature of this pattern is that the scheduling algorithm puts the data in a sequential order in each register file of the PE, so it is possible to leverage this for considering only the information about 1 dimension in the implementation. In this way, all the MAC operations of one PE can be grouped in one global MAC instruction, which establishes the maximum number of computation with those data.

The main information enclosed by the data pattern are represented by the following ISA:

Type of command	Command	Type of data	Input data	Max iteration	Step range	Data reuse	Virtual n2n	Send output

It is possible to notice the similarity with a VLIW (Very Long Instruction Word) ISA. Indeed, just as in a VLIW processor architecture, in this implementation the Compiler has a full information about everything and the identification of parallel instructions is done statically before runtime.
Nevertheless the main drawback of having a such specific ISA is the no portability of the generated code.

Below it is explained the meaning of each field:

- *Type of command* identifies the category of command, which can be either an instruction for loading *data* or an operational (*OP*) instruction with information about the MACs.

- *Command* specifies the command itself, namely the main operations supported by a PE that are *MAC* and *load*.

- *Type of data* establishes if the input data belongs to ifmap, weight or psum; in OS dataflow the psum type is never used because it is available only at the output after the computation.

- *Input data* is a list of all pixel values currently in the message.

- *Max iteration* can be the total number of MACs or the number of pixels to be loaded in the RF.

- *Step range* corresponds to the interleaving factor, i.e. the amount of cycles in which the same ifmap pixel is reused across different filters.

- *Data reuse* is the number of ifmap pixel in common, between 2 adjacent sliding windows, that can be reused in neighbour PEs.

- *Virtual n2n* determines whether the PE has a virtual or real neighbour, i.e. Interconnect or east n2n connection.

- *Send output* enables the final write back of the psum in the GLB.

These fields are used to get 2 types of instructions, one for loading the necessary data and the other for computing the psum in the correct way. In the instruction there are some unused fields that are ignored in the decode phase.

Load instruction This instruction is the first that the PE needs to receive since that it holds all the data required to compute a psum. Generally, the minimum numbers of messages with this type of instruction is two (one for the weights and one for the ifmap ideally), but there might be more than 2 messages to deliver all the data because of the limited number of data transferred by the Interconnect in just one instruction.

Type of command	DATA
Command	load
Type of data	ifmap / weight
Input data	I[ni; iy, ix], ... / W[no; ni, ky, kx], . . .
Max iteration	q*Tky*(Tkx-max{0, Tkx-s}) / q*p*Tky*Tkx

Table 5.4.: Load instruction

In theory, the compression of the atomic instructions is of 1 instruction/data_type instead of 1 instruction/data.

MAC instruction The sequence of MACs between ifmap pixel and weight can be wrapped efficiently by using just 1 instruction/convolution instead of $p \cdot q \cdot T_{ky} \cdot T_{kx}$ instructions/convolution:

Type of command	OP
Command	mac
Max iteration	q*p*Tky*Tkx
Step range	p
Data reuse	q*Tky*max{0, (Tkx-s)}
Virtual n2n	0/1
Send output	0/1

Table 5.5.: MAC instruction

The usage of those field can be seen in another way in the following pseudo-code:

```
for (j=0; j<instr.max_iteration; j+=instr.step_range)
      for (i=0; i<instr.step_range; i++)
            psum[i] += ifmap[j]*weight[i+j];

if (send_output==1)
    commit_stage;
```

Listing 5.3: Pseudo code of the execution stage

```
for (i=0; i<instr.data_reuse; i++)
    if (virtual_n2n==1)
        receive_from_virtual_neighbour;
    else
        receive_from_eastN2N;
    send_n2n_data
```

Listing 5.4: Pseudo code of the n2n operation

Thus the *step range* field is necessary especially when interleaving occurs, whereas the number of cycles where the N2N communication is working is determined by the *data reuse* field.

5.4. Tester

Given the layer shape, the Modeling tool provides a suitable data pattern for the OS dataflow and a list of instructions encoded with the current ISA. After the compile phase, a *Tester* verifies the correctness of the program code and final instructions: it builds a *Golden Model* of the convolution, according to the input tiling and unrolling factors provided by *HW flow*, and it compares the correct output volume with that obtained from a SW emulation which follows the schedule of the Modeling tool.

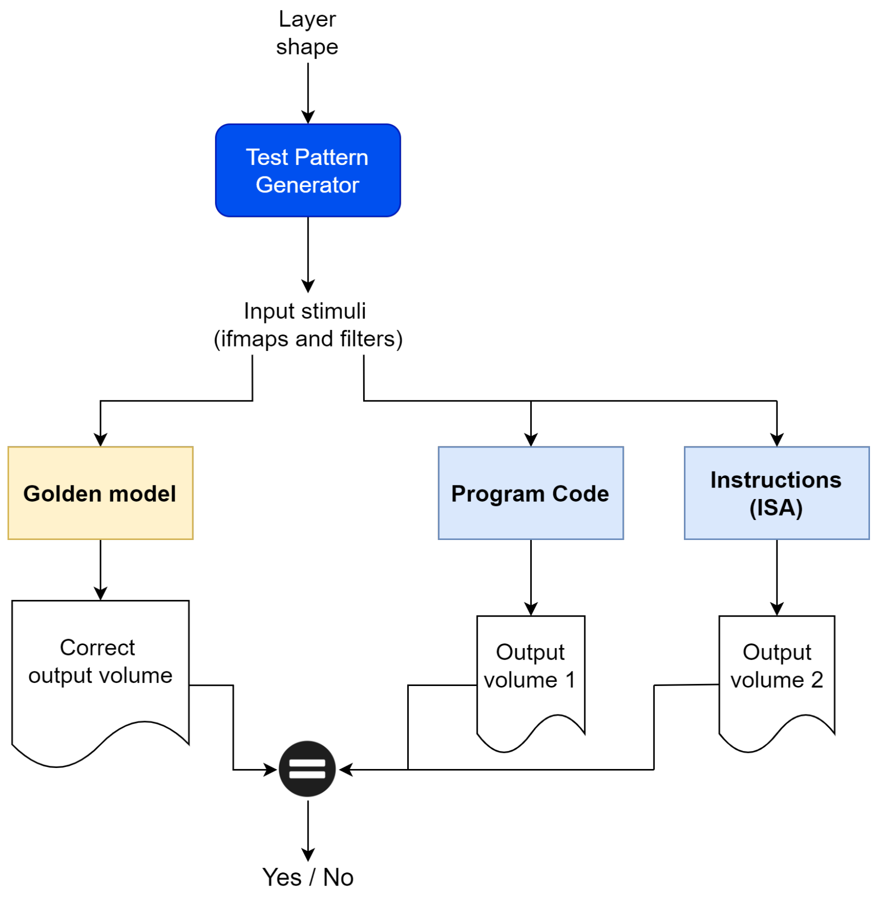

Figure 5.26.: Working principle of the Tester. The Unit Under Test (UUT) is compared with the Golden model.

Doing that, the program code and the ISA are validated and the sequence of instructions can be used correctly in the next HW implementation.

6. Hardware implementation

The previous chapters gave a detailed explanation of the scheduling algorithm and mapping techniques employed by the Modeling tool, which basically provides the schedule of instruction for a given layer shape. The next step in the design flow of a CNN accelerator is to build the spatial array, starting from the single processing element.
The HW implementation has been done in VHDL in *Vivado Design Suite*.
Section 6.1 describes the implementation of a PE and explains the functionality of each building block inside it, then an overview of the complete PE array with all the connections is given in section 6.2.
Once having the HW architecture, the post-synthesis results of Vivado are reported in section 6.3, for a given PE array configuration.
This HW implementation is then compared with the ideal one, i.e. fully pipelined, by running some simulations and measuring the number of cycles to process a determined input layer shape provided by *HWflow*.
What discussed so far is about a FPGA-based design, but, in the last section of this chapter, the same schedule of instructions is implemented in *Versal ACAP* (Adaptive Compute Acceleration Platform), namely a new cutting-edge ASIC of Xilinx for accelerating CNN. In this case the spatial array has been programmed in C++ by using the corresponding SW tools.

6.1. Microarchitecture of the Processing element

The high-level block diagram of a PE is depicted below:

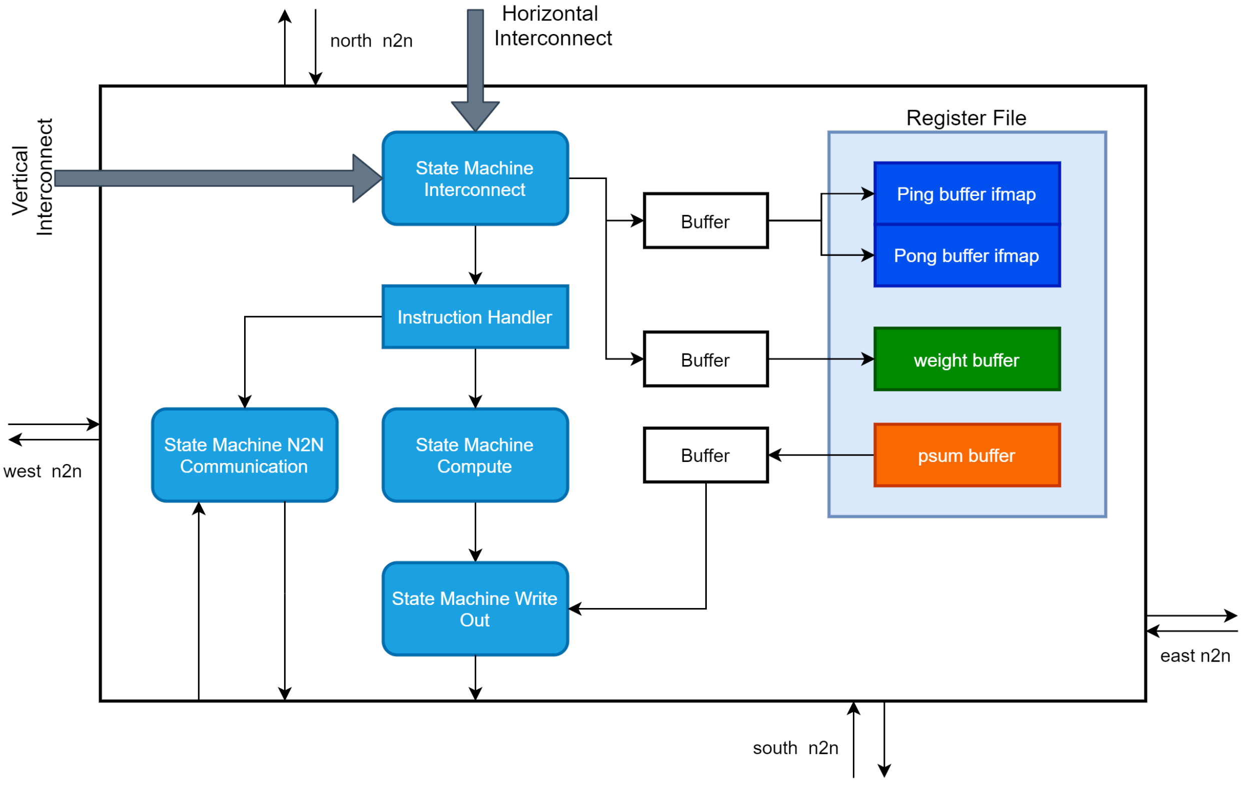

Figure 6.1.: Microarchitecture of a PE

Starting from the outside, there are all the point-to-point connections with neighbour PEs, according to the literature of systolic array, even if the real connections used in the OS dataflow are the incoming east n2n and the outcoming.
Moreover, each Interconnect is made up of the instruction, encoded with the current ISA, and additional signals:

1. *en* (enable signal) to indicate that the interconnect has a new message.
2. *conv_en* (convolution enable signal), used to start the convolution inside the PE
3. *req_en* (request enable signal) and *req_ack* (request acknowledge signal) employed to implement a *handshake* among interconnect and PE in the final commit stage

The output is the *request* of the PE which holds the output pixels and the locations of the GLB for storing them.

The register file is made up of 4 block RAM (BRAM), one for the weights, the psums, the ping and pong buffer of the ifmap; the RAM has on asynchronous read port and one synchronous write port.
However, a detailed description of the functionality of each building block is provided below, considering the typical instruction cycle of a CPU: fetch-decode-execute-commit.

Fetch In this first stage the data are fetched from the on-chip memory and, after being through some logic meant to form the instruction, they are dispatched to the target PE.

Decode First the *State Machine Interconnect* listens over the horizontal and vertical interconnect and so, if there is an enable on one of them, it means that there is an new incoming instruction.
Based on the *tran_type* field, if there is an OP instruction, the state machine sends directly it to the *Instruction Handler*, otherwise if there is a LOAD (5.4) the state machine looks at the *data_type* and decides in which location store the payload of the message. In this last case, the storing phase is handled with the help of two circular buffers, one for the weight RF and the other for the ifmap RF (i.e. ping pong buffer), to interface correctly with the memory elements. In fact the data packet in the instruction comprises multiple data, so a buffer is required to send one data per cycle to the correct location, one after the other.
Basically the RF behaves as a FIFO, so all the data are put in a sequential order in the correct memory location. For what concerns the ifmap RF, the ping buffer is always filled up with the data received by the Interconnect, while the pong buffer is involved later for the n2n communication, as explained in several examples of previous chapter.
On the other hand, the Instruction Handler has an internal FIFO to hold the OP instruction, so in this case just one location is sufficient because there is always one MAC instruction for every PE, nothing more. In other types of dataflow it would be necessary to make larger this FIFO.
When *conv_en* is set to 1, the Instruction Handler starts to unpack the instruction in its FIFO, selecting the useful fields of the ISA to get the MAC instruction (5.5), so it also triggers the state machines involved in the Execute stage.

Execute This stage comprises the *Compute State Machine* and the *N2N Communication State Machine*, both running in parallel. Indeed, as soon as the enable to start the MAC is set, the n2n state machine prepares to receive and send data through horizontal point-to-point connections with the correct timing. If the current PE belongs to the right-most column of the PE set, it needs to receive data from the Interconnect (behaving as a virtual neighbour) and not from the east n2n connection; this distinction is done by using the *virtual_n2n* signal.

Usually the N2N state machine finishes its job after a small number of cycles and then it goes back to the idle state, whereas the compute state machine continues with the accumulation of psum.
When the computation is over, the compute state machine checks if the *send_output* field is set, which means that the psums are ready to go back to the GLB; in the opposite case, the psum needs to be further accumulated and all the state machines go in an idle state, waiting to receive new data to process.

Commit If *send_output* becomes high, the *Write Out State Machine* starts the final commit stage. The output pixels are retrieved from the RF by using a circular buffer, which reads a psum at a time and, once that all psums are in the buffer, they are used to make the output *request* and the *req_en* signal is set.
At this point the state machine waits until the Interconnects takes the request with the psums, which happens when the Interconnect raises up the *req_ack* signal. When this event takes place, the state machine lowers the *req_en* signal and the commit stage is done.
If the interleaving parameter is large enough, there might be more than one commit stage to send all psums over the Interconnect. It is a symmetrical situation of the loading phase.

6.2. PE array implementation

The processing elements are put all together in a PE array, where several PE sets can be allocated depending on the type of logical and physical mapping of the schedule. The north, west, south and east point-to-point connections in both directions are made, even if the PE employs only the horizontal n2n connections in the OS dataflow.
Instead of having routers, which are computational expensive, for handling the communication between GLB and PE array, there are horizontal and vertical interconnects connected to a Scheduler that decides what data send over the interconnect. Thus, the PEs do not need to talk with a router because a PE already knows how to work thanks to the ISA.

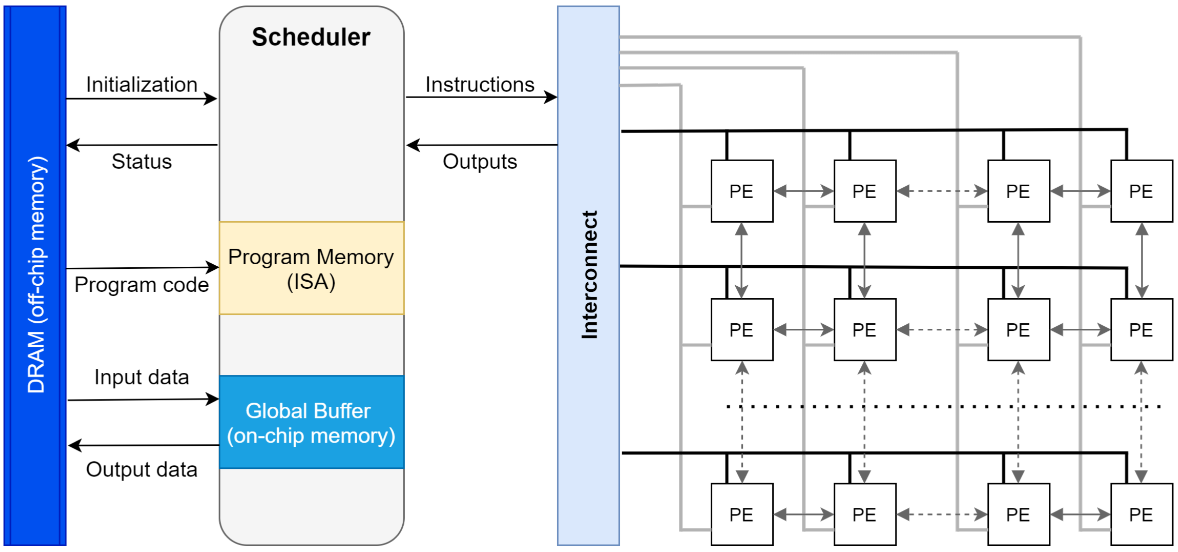

Figure 6.2.: Systolic accelerator with the Scheduler

The overall architecture is parametric, so it is possible to define all the wanted HW parameters. In this work, the following parameters have been considered:

- data width = 32 bit, i.e. the number of bit supported by the Interconnect for a data
- ifmap bits = weight bits = 16
- psum bits = 32
- burst size = 10, i.e. the number of data supported by the Interconnect
- psum RF depth = 16
- ifmap ping or pong buffer depth = 12
- weight RF depth = 224

Particularly, all these variables are given by *HWflow*, which ensures the validity of them for specific tiles of data and unrolling/interleaving factors.

Testbench Just as the previous Tester 5.26, a test of the PE and spatial architecture is carried out to check the respect of the design specifications and interface, the correct functionality of the HW implementation, the validity of the schedule provided by the Modeling tool.
A wrong dispatch of data/instructions to the PEs, a missed synchronization between Compute and N2N state machines during the Execution stage, or an incorrect interface with the Interconnect in the final Commit stage, are all errors that can be

detected through a suitable testbench.
Basically this HW testbench feeds the Unit Under Test (UUT), i.e. the PE array, with the schedule of instructions obtained from the input stimuli and check whether the output pixels are the same computed by the Reference model.
The test takes into account also the dispatch of the data in several instructions, according to the burst size of the interconnect, and the correct retrieve of the final output pixel computed by every PE.

6.3. Experimental results

This section shows the validation of the Modeling tool, namely the schedule of the instructions provided by its Compiler, for 2 input layer shapes given by the Scheduler/Mapper of *HWflow* (see the Appendix for the dimensions). As remarked several times, the OS dataflow is the only one considered in this work, which is quite expensive in energy but not so bad in latency and it keeps all the PEs busy (i.e. 100% of utilization of the array) typically. According to *HWflow*, it provides better results than WS dataflow.
For what concerns the HW implementation, the focus is on 3 configurations of the PE array, which have been synthesized in Vivado, in a Xilinx Zynq UltraScale+ MPSoC (active device xczu9eg-ffvb1156-2-e), and employed in logic simulation to measure the latency and make some comparisons with the ideal HW architecture.

6.3.1. Post-synthesis results

The HW implementation of the single PE and PE array has been synthesized in "out-of-context" mode, so preventing the insertion of I/O buffers.
The only design constrain considered was the clock of 200 MHz, which ensured no timing violations (i.e. setup and hold time) in this design.
The post-synthesis result of one PE is depicted below:

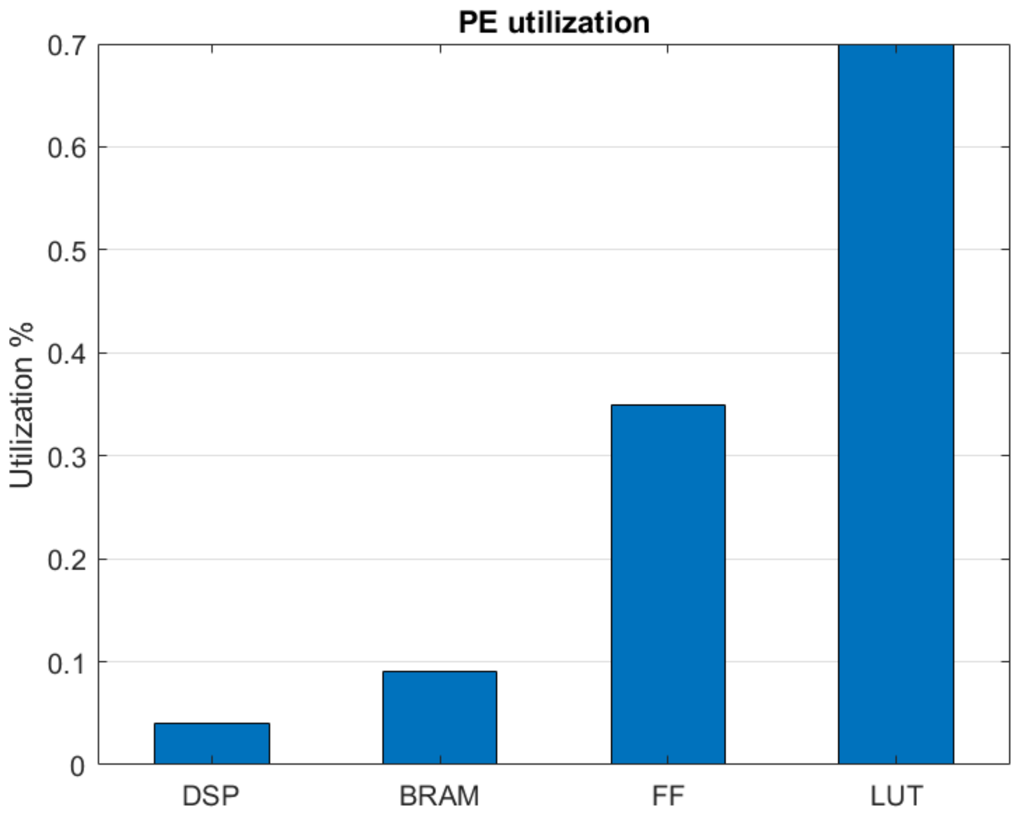

Figure 6.3.: Utilization summary of one Processing Element; the HW resources considered are Digital Signal Processors, BRAMs, Flip-flops, Look-up-tables

The HW utilization is below 1%, so the result is reasonable. The LUT utilization is the higher because of the high control logic employed in the PE, in order to handle correctly the state machines correctly. The DSP used by one PE, for the MAC operation, is just 1.

This PE is employed to build three configurations of PE array : 4x4, 6x6 and 8x8 array.

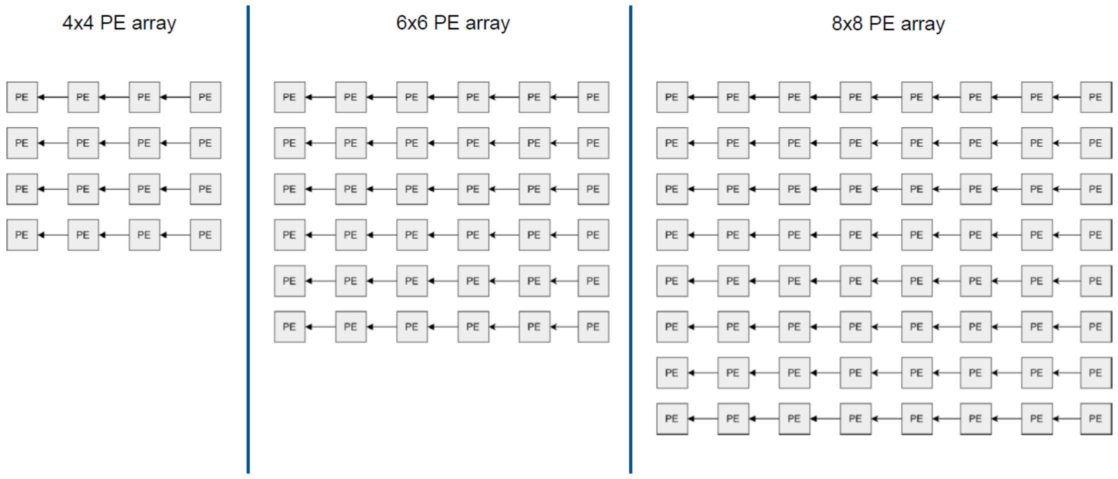

Figure 6.4.: Overview of the 3 configurations considered for the synthesis

After the synthesis, the results are the following ones:

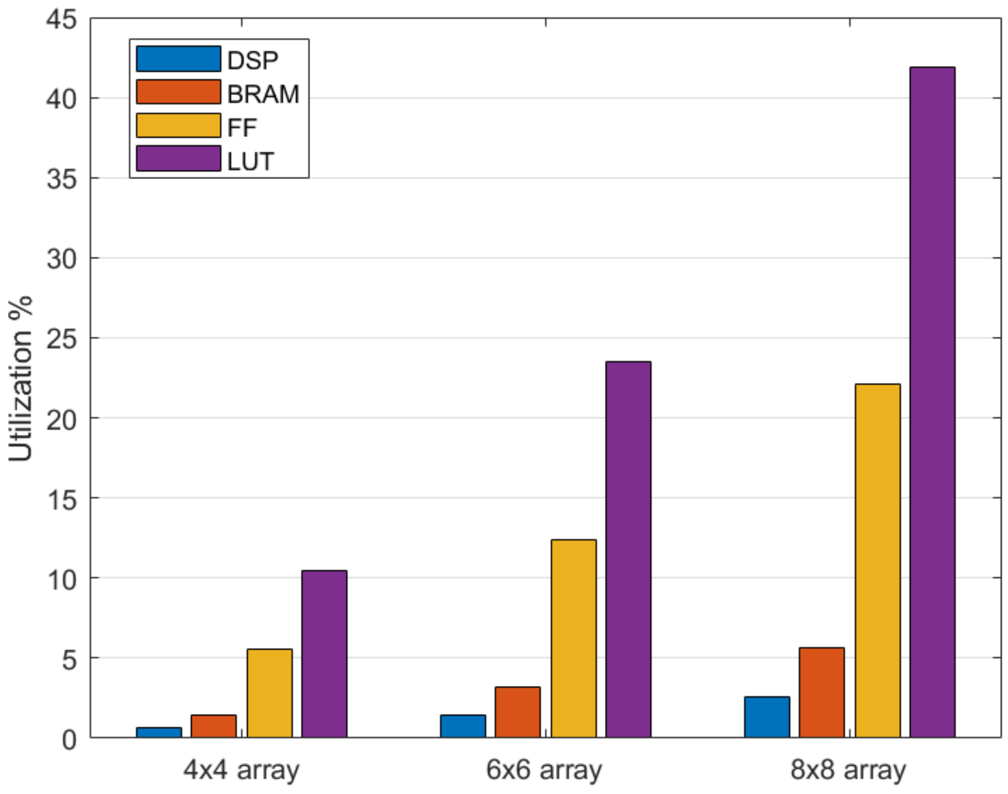

Figure 6.5.: Utilization summary of the 3 configurations for the spatial array

As expected there is a general increase of the HW resources as the PE array grows up. The trend of the 4 types of HW resources is the same in all configurations, with a pretty high LUT utilization reaching almost 45% in the 8x8 array, whereas the other two configurations have a overall HW utilization that is low.

6.3.2. Logic simulations

The two input layer shapes considered are AlexNet and ResNet20; moreover, for each of it, *HWflow* provides the schedule of tiling and unrolling factors for the 3 PE array configurations.
Generally, the number of multiplication operations for a layer shape [14], without considering loop tiling and unrolling, are:

$$N_{mul} = N_{if} \cdot N_{ky} \cdot N_{kx} \cdot N_{of} \cdot N_{oy} \cdot N_{ox} \tag{6.1}$$

This value can be compared with the number of computing cycles, under different loop unrolling and tiling dimension, necessary in the spatial array for a specific layer shape employing the OS dataflow:

$$ComputingCyles\_ideal = T_{if} \cdot T_{ky} \cdot T_{kx} \cdot T_{of} \cdot \left\lceil \frac{T_{oy}}{P_{oy}} \right\rceil \cdot \left\lceil \frac{T_{ox}}{P_{ox}} \right\rceil \tag{6.2}$$

This quantity is nothing else that the required number of cycles to process the data stored in the GLB through the PE array.
It is worth to point out how the unrolling parameters strongly affect this quantity due to their presence at denominator, while the interleaving parameter is beneficial only for the energy consumption and it does not affect that quantity.

The number of computing cycles per each layer, in the HW implementation of this work, is given by:

$$ComputingCyles\_real = T_{if} \cdot (T_{ky} \cdot T_{kx} \cdot T_{of} + 4) \cdot \left\lceil \frac{T_{oy}}{P_{oy}} \right\rceil \cdot \left\lceil \frac{T_{ox}}{P_{ox}} \right\rceil \tag{6.3}$$

The introduction of that constant is due to:

- 3 cycles at the beginning of the computation, of which 2 cycles required to unpack the instruction from the Instruction Handler + 1 cycle to start the MAC and the n2n communication
- 1 cycle at the end of the computation for having the psums ready to be sent back to the GLB

A high fidelity of the Modeling tool must ensure a similar trend between ideal and real number of computing cycles. The next paragraphs provide and estimation of these quantites for 19 CONV layers of ResNet20 and 5 CONV layers of AlexNet.

ResNet20 The order of magnitude of the total number of multiplications, without applying loop unrolling and tiling, is around 10^6:

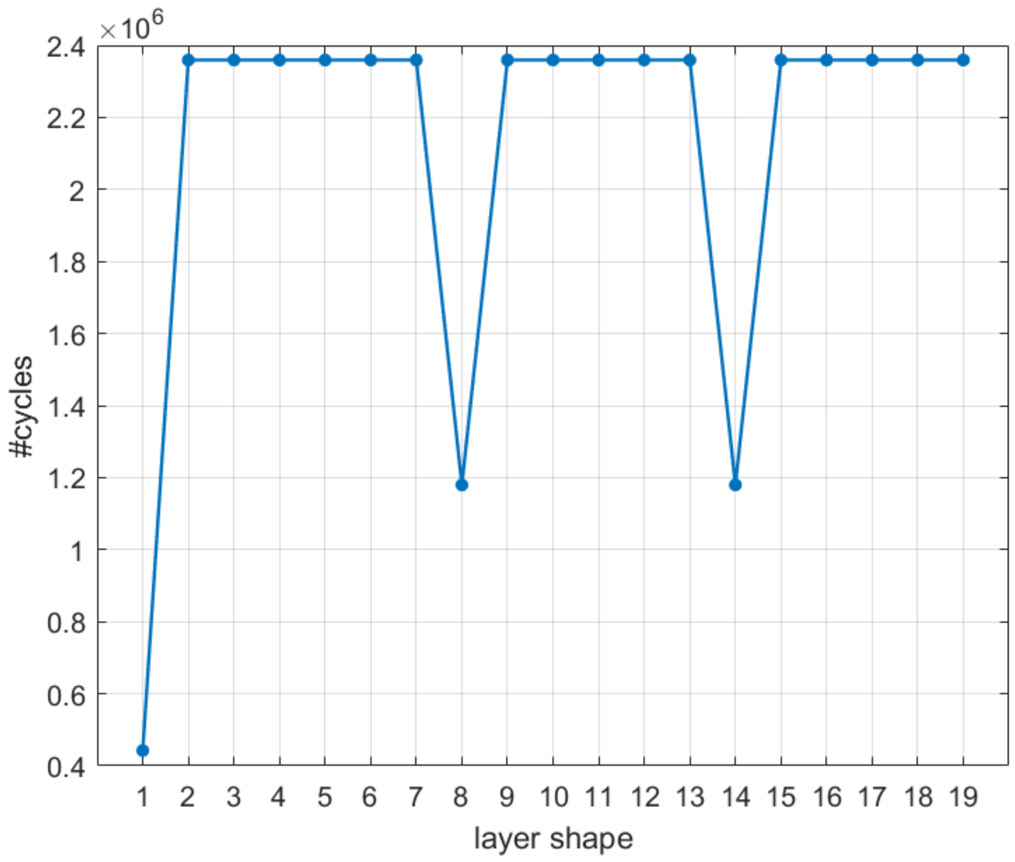

Figure 6.6.: N_{mul} of ResNet20

The adoption of loop scheduling techniques must provide a reduction of this quantity:

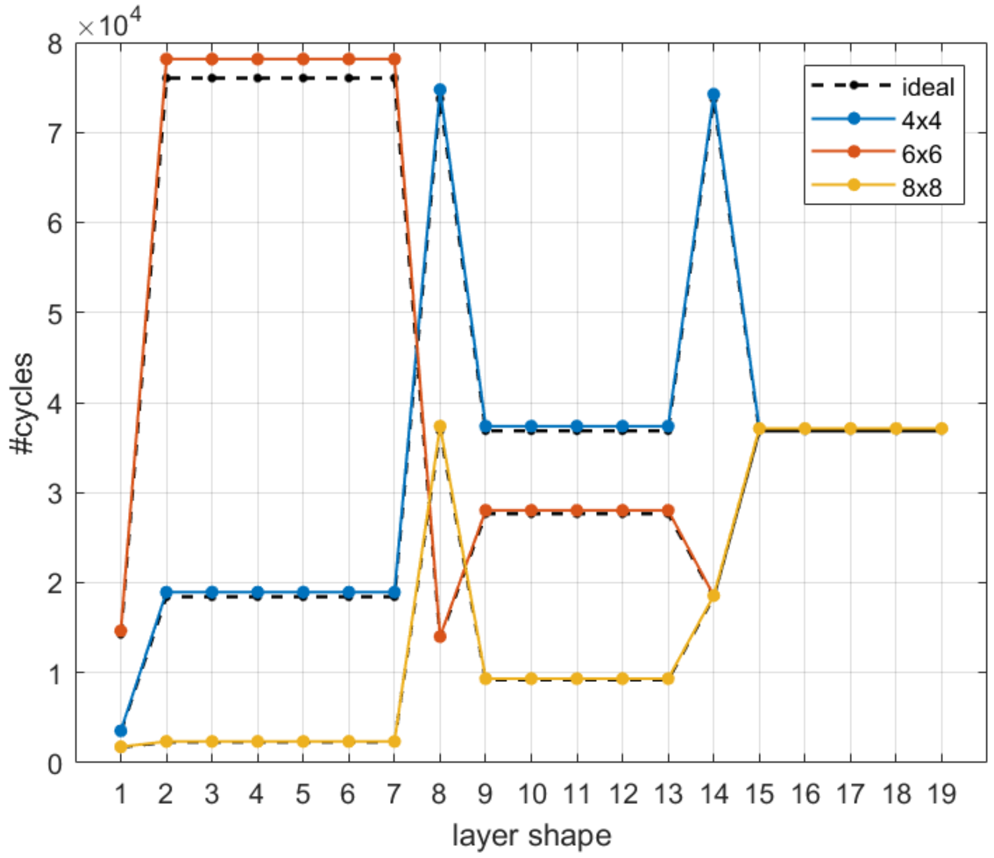

Figure 6.7.: ResNet20 trend of the computing cycles; the dashed line is the ideal number of computation, while the colored lines are for the real trend of the three PE array configurations

Now, the maximum number of computing cycles is around $8 \cdot 10^4$, so there is a reduction of 2 orders of magnitude. The trends of the real and ideal number of computations are very similar each other, in fact the average relative change is of 1.68% in every PE array configuration.

AlexNet The overall number of multiplications required for these CONV layers are around 10^8:

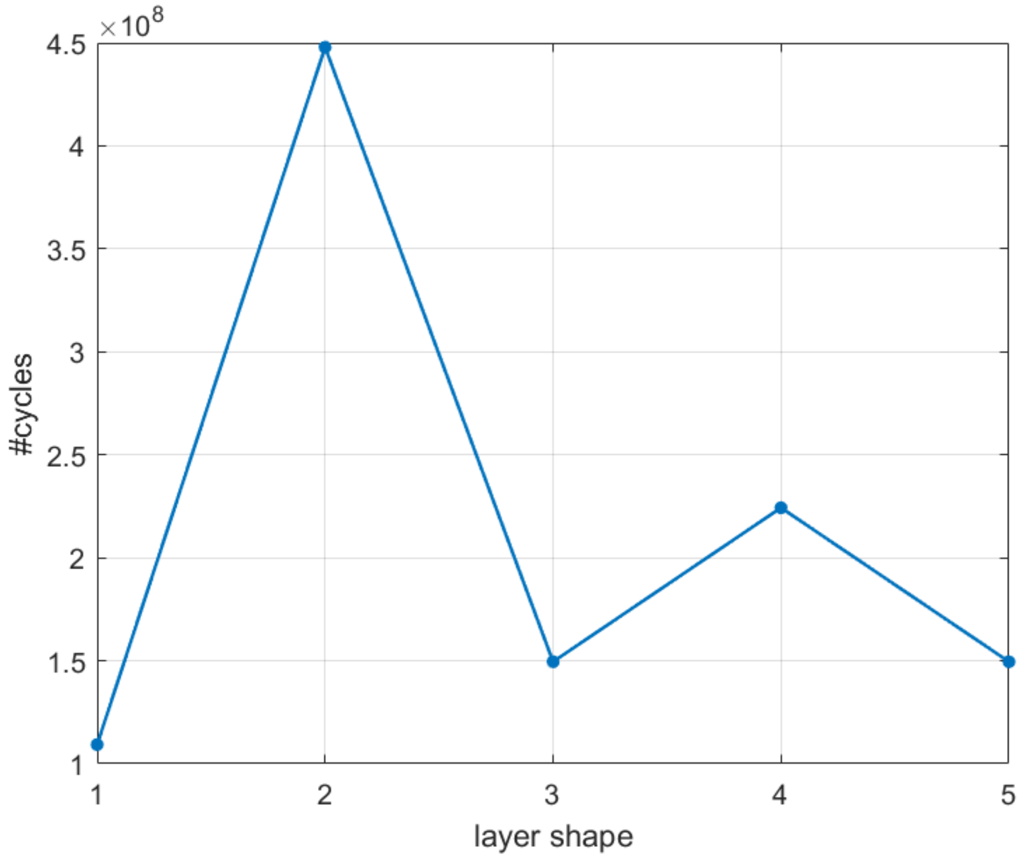

Figure 6.8.: N_{mul} of AlexNet

Also in this case, there is a reduction of 2 orders of magnitude in the computing cycles after applying loop scheduling techniques:

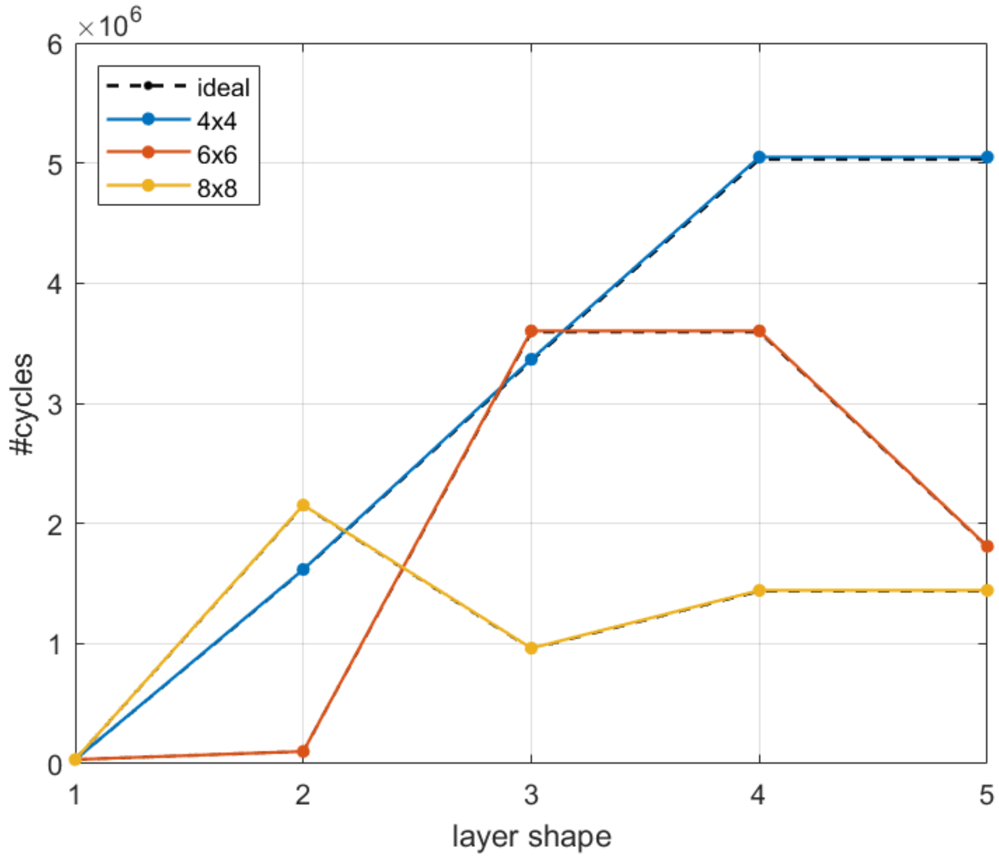

Figure 6.9.: AlexNet trend of the computing cycles; the dashed line is indistinguishable from the colored lines because ideal and real behaviour are nearly coincident

These trends are almost equal to the ideal ones: the average relative changes are below 1%, so negligible for all configurations.

Remarks The Execute stage of the PE has the particular feature of having the n2n and compute state machines running in parallel, which allows to have a HW model as similar as possible to a fully pipeline HW, where there is no additional latency during the execution.
Although the 8x8 PE array has the highest resource utilization, it provides the lower number of computing cycles per layer with respect to the other configurations, reaching a large parallel execution of CONV layer through its 64 PEs.
Conversely the 4x4 array provides the lowest HW utilization but a latency trend high enough. On the other hand, the 6x6 array represents a good trade-off for latency and HW resources utilization.
Therefore, depending on the type of target application, a PE array configuration

could be more suitable than the others.

6.4. Versal design

The Xilinx *Versal* ACAP (Adaptive Compute Acceleration Platform) is a new heterogeneous compute architecture [46] that comprises Scalar/Vector processing elements tightly coupled with a programmable logic. This platform is SW programmable in C or C++ with proper frameworks.

AI engine One of the main feature is the presence of a dataflow architecture for parallel computation of massive data. Indeed, the *AI engine* (AIE) corresponds to the PE of the previous implementation and multiple AI engines are grouped together in a AIE array, so it is possible to implement the OS dataflow in this type of architecture considering the current data pattern.
The AIE is suitable for vector-based algorithms, providing flexible custom compute and data movement [47]. The main characteristics are:

- single instruction multiple data (SIMD) processor
- VLIW ISA up to 6-way instruction parallelism

Scheduling algorithm mapping in the AIE The scheduling algorithm of the Compiler changes the data representation by handling just 1 index:

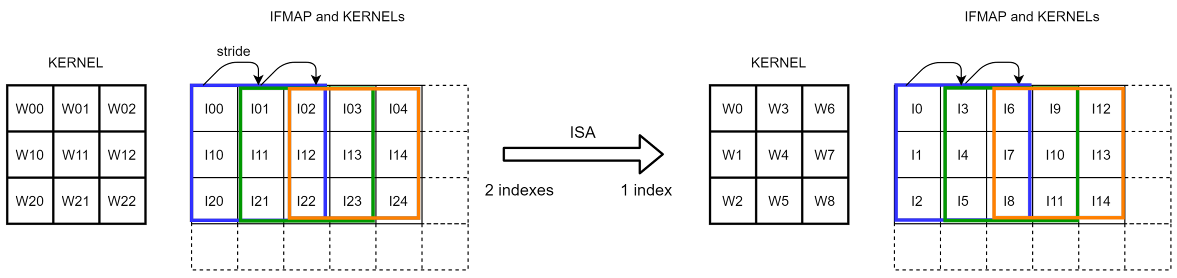

Figure 6.10.: Flattening of 5.6, achieved through the ISA, with focus on 3 sliding windows

The computation required by those 3 sliding windows can be easily mapped in three PEs (Pox=3) laying on 1 row of the PE set. If looking the ifmap and the kernel in 1D, the result is the following:

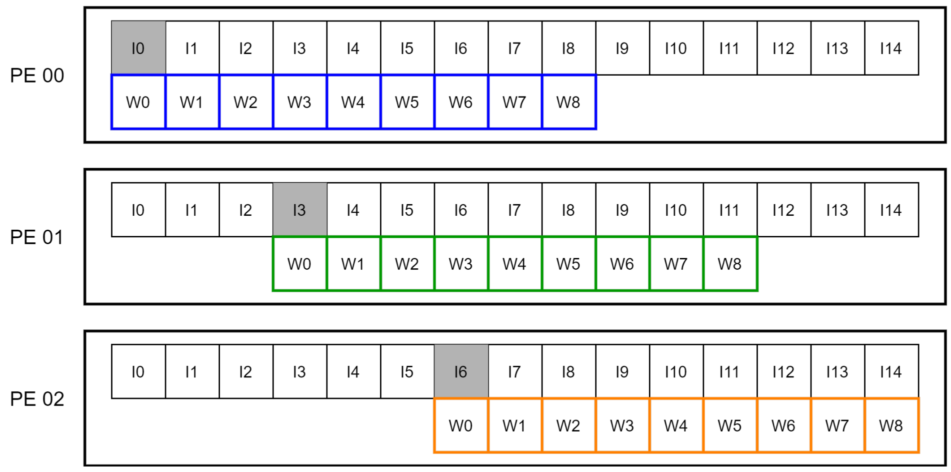

Figure 6.11.: 1D representation of the ifmap and kernel, the pixel in grey is the starting one of the computation

Hence, what happens is that the the weight array slides over the ifmap by shifting of $overlap \cdot T_{ky} = 3$ positions at every PE. The equation of the psums computed with this pattern is given by:

$$psum[k] = \sum_{k=0}^{Pox-1} \sum_{i=0}^{N-1} = weight[i] \cdot pixel[i + k \cdot C] \tag{6.4}$$

It is worth to highlight the similarity of it with the FIR (Finite Impulse Response) filter equation:

$$y[n] = \sum_{k=0}^{N} c[k] \cdot d[n + k] \tag{6.5}$$

Hence the data samples are the ifmap pixels, the coefficients of the filter are the weights, while the step at every cycle is not anymore by 1 but it is $C = overlap \cdot T_{ky}$, which is a constant depending on the stride and the kernel height. By exploiting the scalar/vector processor of the AI engine, it is possible to schedule the computations of 6.5 in just one AIE, obtaining:

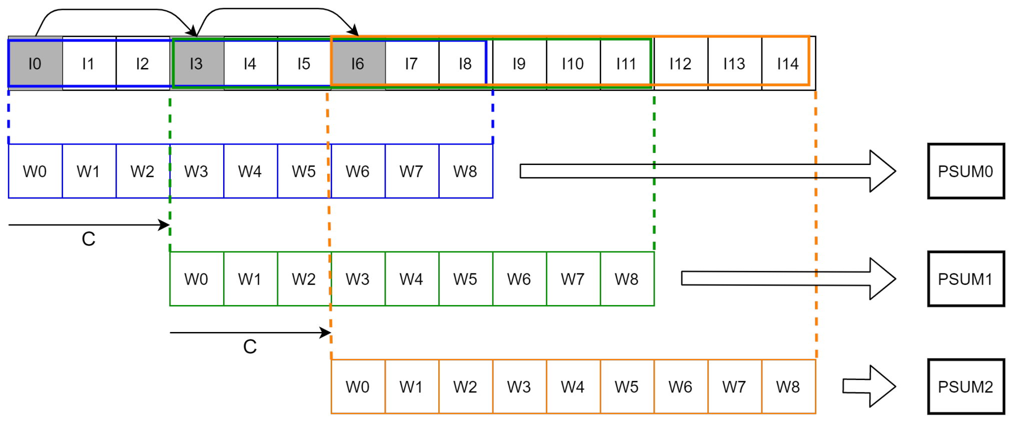

Figure 6.12.: Schedule of 3 psums in one AIE, showing the sliding window at every iteration over the ifmap

This type of schedule allows to map a PE set row, made up of Pox PEs, in just one AI engine, achieving the best reuse of ifmap pixel since that the n2n communication happens within the engine:

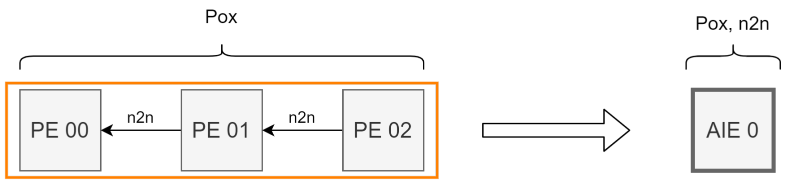

Figure 6.13.: New type of mapping in the AIE

These considerations can be applied when having larger unrolling factors, achieving a high compression of the PEs inside one AIE.
Nevertheless the limitations come out when dealing with greater shift of the sliding window, like with a stride=2: the offset applied at the ifmap at every iteration, to get the right overlap with the sliding window, is not always straightforward to apply to the buffer that holds the data. When things start complicating, the scalar process of the AIE can be employed to handle more complex offset schemes, even though the vector processor is definitely more efficient with parallelizable computations.
Moreover, the maximum number of PEs in one AIE depends on the number of data that can be stored in the registers inside the AIE, which are fixed to certain amounts. In fact, it is better to employ $q = 1$ and $p = 1$ in this application, in order to keep low the number of necessary data in one AIE.

7. Conclusion and Outlook

7.1. Summary

The mapping of the CNN execution in dataflow architecture is a challenging task that requires an extensive analysis, which takes into account all possible SW and HW parameters affecting the performance of the accelerator. The HW/SW codesign process of CNN accelerator has started in *HWflow* framework and this work proposed mainly a Compiler that provides a correct schedule for a possible HW implementation.
The breakdown of the OS dataflow helps in identifying all the relevant features of the architecture, leading to a homogeneous and regular data pattern in the spatial array. This uniformity in the data movement contributed to make an efficient ISA to encode all the micro-instructions, required by the dataflow, in only 2 types of instructions, basically one for loading the necessary data and the other for the effective computation. Moreover, the schedule obtained through this ISA does not require a high usage of the horizontal and vertical Interconnect, just as wanted from the beginning of the design, which is oriented to a low power implementation.
All these concepts are grouped together in a sole Modeling tool, able to link SW and HW implementation of CNN. However, this work dealt only the OS dataflow, so the program code and the schedule of instructions generated at compile time is not general purpose but they can be employed for this particular dataflow, which showed good performance in the prior evaluations of *HWflow*.
Furthermore, the HW implementation showed an almost ideal behaviour of the Execute stage, giving a high fidelity to the Modeling tool. The post-synhtesis results and the utilization summaries provide an idea on what HW configuration is better to adopt for the target application.
Finally, the same Compiler and type of schedule is adopted in a new design with the *Versal* platform. The scalar processor of the AI engine provides a flexible implementation of the schedule, especially when the CONV layer shape is irregular, whereas the vector processor is suitable for a more regular layer and leads to a high improvement of the performance.
To wrap up, it is important to highlight that suitable testbenches were prepared to check the correctness of each design step, especially the final HW implementation.

7.2. Future works

The approach followed in this works, to analyze the OS dataflow and use it in the mapping process, can be extended to the other types of dataflow like WS, IS and RS. In fact *HWflow* provides the schedule also for those dataflows, so a final comparison of the trends and HW utilization can be made. To implement an other type of dataflows it is sufficient applying loop ordering technique to get the right schedule of computations, and employing the unrolling and interleaving factors suggested by *HWflow*.
Regarding the implementation carried out in this work, the HW can be easily changed to obtain the wanted spatial architecture that is more suitable for a given CNN execution. Thus this custom PE array can be integrated with the other HW components of the systolic accelerator, such as the horizontal and vertical Interconnect, the GLB, the DRAM and also some additional logic to improve the dispatch of the instructions in the array.
An interesting outlook is the usage of the AIE array of *Versal* for accelerating large CNN layers, by using the same type of Compiler and data pattern. Moreover, the vector processor of the AI engine is suitable to process in parallel the psums scheduled for a PE set row of the OS dataflow, providing a high acceleration ratio and a smart schedule of n2n within the engine.

Appendices

A. Layer shapes

This appendix contains all the dimensions about the layer shapes employed for the HW evaluation.

A.1. AlexNet

Layer	**Nif**	**Nix**	**Niy**	**Nkx**	**Nky**	**Nof**	**Px**	**Py**	**Nox**	**Noy**	**stride**
1	3	224	224	11	11	96	5	5	56	56	4
2	96	27	27	5	5	256	2	2	27	27	1
3	256	13	13	3	3	384	1	1	13	13	1
4	384	13	13	3	3	384	1	1	13	13	1
5	384	13	13	3	3	256	1	1	13	13	1
6	256	1	1	1	1	4096	0	0	1	1	1
7	4096	1	1	1	1	4096	0	0	1	1	1
8	4096	1	1	1	1	1000	0	0	1	1	1

Table A.1.: Layer shape and computing cycles

Layer	**Tif**	**Tix**	**Tiy**	**Tkx**	**Tky**	**Tof**	**Tox**	**Toy**	**stride**
1	3	15	15	11	11	96	2	2	4
2	48	7	31	5	5	64	3	27	1
3	32	15	15	3	3	128	13	13	1
4	48	15	15	3	3	128	13	13	1
5	48	15	15	3	3	128	13	13	1

Table A.2.: Tiles for a 4x4 spatial array

Layer	Poy	Pox	p	Pofv	Pofh
1	2	2	1	2	2
2	4	1	8	1	4
3	1	2	16	4	2
4	1	2	16	4	2
5	1	2	16	4	2

Table A.3.: Unrolling factors for a 4x4 spatial array

Layer	Tif	Tix	Tiy	Tkx	Tky	Tof	Tox	Toy	stride
1	3	15	15	11	11	96	2	2	4
2	32	7	7	5	5	128	3	3	1
3	32	15	15	3	3	192	13	13	1
4	32	15	15	3	3	192	13	13	1
5	16	15	15	3	3	256	13	13	1

Table A.4.: Tiles for a 6x6 spatial array

Layer	Poy	Pox	p	Pofv	Pofh
1	2	2	1	3	3
2	3	3	8	2	2
3	1	3	16	6	2
4	1	3	16	6	2
5	2	2	15	3	3

Table A.5.: Unrolling factors for a 6x6 spatial array

Layer	Tif	Tix	Tiy	Tkx	Tky	Tof	Tox	Toy	stride
1	3	15	15	11	11	96	2	2	4
2	32	7	31	5	5	128	3	27	1
3	32	15	15	3	3	128	13	13	1
4	48	15	15	3	3	128	13	13	1
5	48	15	15	3	3	128	13	13	1

Table A.6.: Tiles for a 8x8 spatial array

Layer	Poy	Pox	p	Pofv	Pofh
1	2	2	1	4	4
2	4	1	8	2	8
3	1	7	16	8	1
4	1	7	16	8	2
5	1	7	16	8	1

Table A.7.: Unrolling factors for a 8x8 spatial array

A.2. ResNet20

Layer	Nif	Nix	Niy	Nkx	Nky	Nof	Px	Py	Nox	Noy	stride
1	3	32	32	3	3	16	1	1	32	32	1
2	16	32	32	3	3	16	1	1	32	32	1
3	16	32	32	3	3	16	1	1	32	32	1
4	16	32	32	3	3	16	1	1	32	32	1
5	16	32	32	3	3	16	1	1	32	32	1
6	16	32	32	3	3	16	1	1	32	32	1
7	16	32	32	3	3	16	1	1	32	32	1
8	16	32	32	3	3	32	1	1	16	16	2
9	32	16	16	3	3	32	1	1	16	16	1
10	32	16	16	3	3	32	1	1	16	16	1
11	32	16	16	3	3	32	1	1	16	16	1
12	32	16	16	3	3	32	1	1	16	16	1
13	32	16	16	3	3	32	1	1	16	16	1
14	32	16	16	3	3	64	1	1	8	8	2
15	64	8	8	3	3	64	1	1	8	8	1
16	64	8	8	3	3	64	1	1	8	8	1
17	64	8	8	3	3	64	1	1	8	8	1
18	64	8	8	3	3	64	1	1	8	8	1
19	64	8	8	3	3	64	1	1	8	8	1
20	64	1	1	3	3	10	0	0	1	1	1

Table A.8.: Layer shape and computing cycles

Layer	**Tif**	**Tix**	**Tiy**	**Tkx**	**Tky**	**Tof**	**Tox**	**Toy**	**stride**
1	3	6	34	3	3	16	4	32	1
2	16	6	34	3	3	16	4	32	1
3	16	6	34	3	3	16	4	32	1
4	16	6	34	3	3	16	4	32	1
5	16	6	34	3	3	16	4	32	1
6	16	6	34	3	3	16	4	32	1
7	16	6	34	3	3	16	4	32	1
8	16	33	33	3	3	32	16	16	2
9	32	4	18	3	3	32	2	16	1
10	32	4	18	3	3	32	2	16	1
11	32	4	18	3	3	32	2	16	1
12	32	4	18	3	3	32	2	16	1
13	32	4	18	3	3	32	2	16	1
14	32	9	17	3	3	64	4	8	2
15	64	4	6	3	3	64	2	4	1
16	64	4	6	3	3	64	2	4	1
17	64	4	6	3	3	64	2	4	1
18	64	4	6	3	3	64	2	4	1
19	64	4	6	3	3	64	2	4	1
20	64	1	1	3	3	10	1	1	1

Table A.9.: Tiles for a 4x4 spatial array

Layer	Poy	Pox	p	Pofv	Pofh
1	3	6	8	2	1
2	3	6	8	2	1
3	3	6	8	2	1
4	3	6	8	2	1
5	3	6	8	2	1
6	3	6	8	2	1
7	3	6	8	2	1
8	6	2	6	1	3
9	6	2	6	1	3
10	6	2	6	1	3
11	6	2	6	1	3
12	6	2	6	1	3
13	6	2	6	1	3
14	2	2	4	3	3
15	2	2	4	3	3
16	2	2	4	3	3
17	2	2	4	3	3
18	2	2	4	3	3
19	2	2	4	3	3
20	4	4	16	1	1

Table A.10.: Unrolling factors for a 4x4 spatial array

Layer	Tif	Tix	Tiy	Tkx	Tky	Tof	Tox	Toy	stride
1	3	18	34	3	3	16	16	32	1
2	16	18	34	3	3	16	16	32	1
3	16	18	34	3	3	16	16	32	1
4	16	18	34	3	3	16	16	32	1
5	16	18	34	3	3	16	16	32	1
6	16	18	34	3	3	16	16	32	1
7	16	18	34	3	3	16	16	32	1
8	16	5	33	3	3	32	2	16	2
9	32	4	18	3	3	32	2	16	1
10	32	4	18	3	3	32	2	16	1
11	32	4	18	3	3	32	2	16	1
12	32	4	18	3	3	32	2	16	1
13	32	4	18	3	3	32	2	16	1
14	32	4	5	3	3	64	1	2	2
15	64	4	4	3	3	64	2	2	1
16	64	4	4	3	3	64	2	2	1
17	64	4	4	3	3	64	2	2	1
18	64	4	4	3	3	64	2	2	1
19	64	4	4	3	3	64	2	2	1
20	64	1	1	3	3	10	1	1	1

Table A.11.: Tiles for a 6x6 spatial array

Layer	**Poy**	**Pox**	**p**	**Pofv**	**Pofh**
1	3	6	8	2	1
2	3	6	8	2	1
3	3	6	8	2	1
4	3	6	8	2	1
5	3	6	8	2	1
6	3	6	8	2	1
7	3	6	8	2	1
8	6	2	6	1	3
9	6	2	6	1	3
10	6	2	6	1	3
11	6	2	6	1	3
12	6	2	6	1	3
13	6	2	6	1	3
14	2	2	4	3	3
15	2	2	4	3	3
16	2	2	4	3	3
17	2	2	4	3	3
18	2	2	4	3	3
19	2	2	4	3	3
20	4	4	16	1	1

Table A.12.: Unrolling factors for a 6x6 spatial array

Layer	**Tif**	**Tix**	**Tiy**	**Tkx**	**Tky**	**Tof**	**Tox**	**Toy**	**stride**
1	3	10	34	3	3	16	8	32	1
2	16	10	10	3	3	16	8	8	1
3	16	10	10	3	3	16	8	8	1
4	16	10	10	3	3	16	8	8	1
5	16	10	10	3	3	16	8	8	1
6	16	10	10	3	3	16	8	8	1
7	16	10	10	3	3	16	8	8	1
8	16	33	33	3	3	32	16	16	2
9	32	6	10	3	3	32	4	8	1
10	32	6	10	3	3	32	4	8	1
11	32	6	10	3	3	32	4	8	1
12	32	6	10	3	3	32	4	8	1
13	32	6	10	3	3	32	4	8	1
14	32	9	17	3	3	64	4	8	2
15	64	4	10	3	3	64	2	8	1
16	64	4	10	3	3	64	2	8	1
17	64	4	10	3	3	64	2	8	1
18	64	4	10	3	3	64	2	8	1
19	64	4	10	3	3	64	2	8	1
20	64	1	1	3	3	10	1	1	1

Table A.13.: Tiles for a 8x8 spatial array

Layer	**Poy**	**Pox**	**p**	**Pofv**	**Pofh**
1	8	8	16	1	1
2	8	8	16	1	1
3	8	8	16	1	1
4	8	8	16	1	1
5	8	8	16	1	1
6	8	8	16	1	1
7	8	8	16	1	1
8	4	8	16	2	1
9	8	4	16	1	2
10	8	4	16	1	2
11	8	4	16	1	2
12	8	4	16	1	2
13	8	4	16	1	2
14	8	4	16	1	2
15	8	2	16	1	4
16	8	2	16	1	4
17	8	2	16	1	4
18	8	2	16	1	4
19	8	2	16	1	4
20	4	4	16	1	1

Table A.14.: Unrolling factors for a 8x8 spatial array

B. HW resource utilization

This appendix shows the post-synthesis results of the FPGA-based design.

Resource	Available utilization
LUT	274080
LUTRAM	144000
FF	548160
DSP	2520

Table B.1.: Available HW resource utilization of the device used in the synthesis

	Utilization			
Resource	**PE**	**4x4 array**	**6x6 array**	**8x8 array**
LUT	1929	28642	64501	114707
LUTRAM	128	2048	4608	8192
FF	1892	30272	68112	121088
DSP	1	16	36	64

Table B.2.: Utilization summary

	Utilization %			
Resource	**PE**	**4x4 array**	**6x6 array**	**8x8 array**
LUT	0,70	10.45	23.53	41.85
LUTRAM	0,09	1.42	3.20	5.69
FF	0,35	5.52	12.43	22.09
DSP	0,04	0.63	1.43	2.54

Table B.3.: Percentage utilization summary

Bibliography

[1] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[2] K. Abdelouahab, M. Pelcat, J. Serot, and F. Berry, "Accelerating cnn inference on fpgas: A survey," *arXiv preprint arXiv:1806.01683*, 2018.

[3] A. Esteva, B. Kuprel, R. A. Novoa, J. Ko, S. M. Swetter, H. M. Blau, and S. Thrun, "Dermatologist-level classification of skin cancer with deep neural networks," *nature*, vol. 542, no. 7639, pp. 115–118, 2017.

[4] C. Chen, A. Seff, A. Kornhauser, and J. Xiao, "Deepdriving: Learning affordance for direct perception in autonomous driving," in *Proceedings of the IEEE International Conference on Computer Vision*, pp. 2722–2730, 2015.

[5] A. Shawahna, S. M. Sait, and A. El-Maleh, "Fpga-based accelerators of deep learning networks for learning and classification: A review," *IEEE Access*, vol. 7, pp. 7823–7859, 2018.

[6] A. Stoutchinin, F. Conti, and L. Benini, "Optimally scheduling cnn convolutions for efficient memory access," *arXiv preprint arXiv:1902.01492*, 2019.

[7] Y.-H. Chen, J. Emer, and V. Sze, "Using dataflow to optimize energy efficiency of deep neural network accelerators," *IEEE Micro*, vol. 37, no. 3, pp. 12–21, 2017.

[8] A. Parashar, P. Raina, Y. S. Shao, Y.-H. Chen, V. A. Ying, A. Mukkara, R. Venkatesan, B. Khailany, S. W. Keckler, and J. Emer, "Timeloop: A systematic approach to dnn accelerator evaluation," in *2019 IEEE international symposium on performance analysis of systems and software (ISPASS)*, pp. 304–315, IEEE, 2019.

[9] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 1, pp. 269–284, 2014.

[10] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "Shidiannao: Shifting vision processing closer to the sensor," in

Proceedings of the 42nd Annual International Symposium on Computer Architecture, pp. 92–104, 2015.

[11] M. Peemen, A. A. Setio, B. Mesman, and H. Corporaal, “Memory-centric accelerator design for convolutional neural networks,” in *2013 IEEE 31st International Conference on Computer Design (ICCD)*, pp. 13–19, IEEE, 2013.

[12] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, “Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks,” *IEEE journal of solid-state circuits*, vol. 52, no. 1, pp. 127–138, 2016.

[13] V. Dumoulin and F. Visin, “A guide to convolution arithmetic for deep learning,” *arXiv preprint arXiv:1603.07285*, 2016.

[14] Y. Ma, Y. Cao, S. Vrudhula, and J.-s. Seo, “Optimizing the convolution operation to accelerate deep neural networks on fpga,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 7, pp. 1354–1367, 2018.

[15] D. F. Bacon, S. L. Graham, and O. J. Sharp, “Compiler transformations for high-performance computing,” *ACM Computing Surveys (CSUR)*, vol. 26, no. 4, pp. 345–420, 1994.

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

[17] X. Wei, C. H. Yu, P. Zhang, Y. Chen, Y. Wang, H. Hu, Y. Liang, and J. Cong, “Automated systolic array architecture synthesis for high throughput cnn inference on fpgas,” in *Proceedings of the 54th Annual Design Automation Conference 2017*, pp. 1–6, 2017.

[18] M. Sankaradas, V. Jakkula, S. Cadambi, S. Chakradhar, I. Durdanovic, E. Cosatto, and H. P. Graf, “A massively parallel coprocessor for convolutional neural networks,” in *2009 20th IEEE International Conference on Application-specific Systems, Architectures and Processors*, pp. 53–60, IEEE, 2009.

[19] V. Sriram, D. Cox, K. H. Tsoi, and W. Luk, “Towards an embedded biologically-inspired machine vision processor,” in *2010 International Conference on Field-Programmable Technology*, pp. 273–278, IEEE, 2010.

[20] S. Chakradhar, M. Sankaradas, V. Jakkula, and S. Cadambi, “A dynamically configurable coprocessor for convolutional neural networks,” in *Proceedings of the 37th annual international symposium on Computer architecture*, pp. 247–257, 2010.

[21] L. Cavigelli, D. Gschwend, C. Mayer, S. Willi, B. Muheim, and L. Benini, "Origami: A convolutional network accelerator," in *Proceedings of the 25th edition on Great Lakes Symposium on VLSI*, pp. 199–204, 2015.

[22] H. Genc, A. Haj-Ali, V. Iyer, A. Amid, H. Mao, J. Wright, C. Schmidt, J. Zhao, A. Ou, M. Banister, *et al.*, "Gemmini: An agile systolic array generator enabling systematic evaluations of deep-learning architectures," *arXiv preprint arXiv:1911.09925*, 2019.

[23] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, "Tetris: Scalable and efficient neural network acceleration with 3d memory," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 751–764, 2017.

[24] H. Kwon, A. Samajdar, and T. Krishna, "Maeri: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects," *ACM SIGPLAN Notices*, vol. 53, no. 2, pp. 461–475, 2018.

[25] M. Motamedi, P. Gysel, V. Akella, and S. Ghiasi, "Design space exploration of fpga-based deep convolutional neural networks," in *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 575–580, IEEE, 2016.

[26] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh, "From high-level deep neural models to fpgas," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–12, IEEE, 2016.

[27] S. Liu, Z. Du, J. Tao, D. Han, T. Luo, Y. Xie, Y. Chen, and T. Chen, "Cambricon: An instruction set architecture for neural networks," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 393–405, IEEE, 2016.

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

[29] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 449–460, IEEE, 2012.

[30] K. Kwon, A. Amid, A. Gholami, B. Wu, K. Asanovic, and K. Keutzer, "Co-design of deep neural nets and neural net accelerators for embedded vision applications," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pp. 1–6, IEEE, 2018.

[31] E. Valpreda, "Optimizing off-chip data movement using layer fusion and loop blocking strategies," 2019.

[32] C. Manfredi, "Efficient hardware dataflow for convolutional neural networks," 2019.

[33] G. M. Caddeo, "Flexible on-chip networks for convolutional neural network accelerators," 2020.

[34] E. Mirsky, A. DeHon, *et al.*, "Matrix: a reconfigurable computing architecture with configurable instruction distribution and deployable resources.," in *FCCM*, vol. 96, pp. 17–19, 1996.

[35] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "Adres: An architecture with tightly coupled vliw processor and coarse-grained reconfigurable matrix," in *International Conference on Field Programmable Logic and Applications*, pp. 61–70, Springer, 2003.

[36] H.-J. Yoo, S. Park, K. Bong, D. Shin, J. Lee, and S. Choi, "A 1.93 tops/w scalable deep learning/inference processor with tetra-parallel mimd architecture for big data applications," in *IEEE international solid-state circuits conference*, pp. 80–81, IEEE, 2015.

[37] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, *et al.*, "A 48-core ia-32 message-passing processor with dvfs in 45nm cmos," in *2010 IEEE International Solid-State Circuits Conference-(ISSCC)*, pp. 108–109, IEEE, 2010.

[38] B. K. Daya, C.-H. O. Chen, S. Subramanian, W.-C. Kwon, S. Park, T. Krishna, J. Holt, A. P. Chandrakasan, and L.-S. Peh, "Scorpio: a 36-core research chip demonstrating snoopy coherence on a scalable mesh noc with in-network ordering," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pp. 25–36, IEEE, 2014.

[39] B. Dally, "Power, programmability, and granularity: The challenges of exascale computing," in *2011 IEEE International Test Conference*, pp. 12–12, IEEE Computer Society, 2011.

[40] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, "Understanding sources of inefficiency in general-purpose chips," in *Proceedings of the 37th annual international symposium on Computer architecture*, pp. 37–47, 2010.

[41] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 367–379, 2016.

[42] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. 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*, pp. 161–170, 2015.

[43] N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J.-s. Seo, and Y. Cao, “Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural networks,” in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 16–25, 2016.

[44] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, *et al.*, “Going deeper with embedded fpga platform for convolutional neural network,” in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 26–35, 2016.

[45] C. Zhang, D. Wu, J. Sun, G. Sun, G. Luo, and J. Cong, “Energy-efficient cnn implementation on a deeply pipelined fpga cluster,” in *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*, pp. 326–331, 2016.

[46] Xilinx, “Wp505 (v1.1.1),” 2020.

[47] Xilinx, “Wp506 (v1.1),” 2020.

Confirmation

Herewith I, Alessandro Di Gioia, confirm that I independently prepared this work. No further references or auxiliary means except those declared in this document have been used.

Munich, December 9, 2020

. . . .
Alessandro Di Gioia