POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

A Python-based Hardware Generation Framework for Tensor Systolic Accelerators

Supervisors

Prof. Andrea CALIMERA

Antonio CIPOLLETTA

Candidate

Nicole DAI PRÀ

December 2021

Summary

Accelerating Deep Neural Networks (DNN) with custom hardware represents an attractive solution to meet stringent applications constraints, especially in mobile/IoT inference scenarios where energy and area efficiency are crucial. Custom hardware is commonly implemented using an iterative process during which the designers identify the main computational and memory patterns of DNN workloads, implement specific hardware structures, and assess the end-to-end performance. As new classes of DNNs are constantly developed and novel reconfigurable platforms, like FPGAs and CGRAs, allow the silicon to be customized after fabrication, agile automation tools are needed to quickly navigate the design space. To this end, in this work, a Python-based framework is proposed to generate tensor systolic arrays, a class of accelerators widely used to perform matrix multiplication, a key operation in DNN workloads.

The proposed framework leverages the metaprogramming capabilities of an HDL embedded in Python to minimize the design and verification effort. In fact, smart systolic array templates allow the user to focus on designing and verifying new processing elements, leaving the burden of creating the routing fabric, the control unit, and the integration tests to the generation framework. The proposed framework is used to perform a design space exploration on the *Zynq Ultrascale+ MPSoC ZCU104 Evaluation Board*, assessing the effect of several knobs, namely, array size, data bitwidth, PE structure, and sparsity support, on area occupation, power consumption, and latency. The obtained results reveal non-trivial trade-offs, motivating the need for such agile design tools to keep raising the efficiency of domain-specific accelerators.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Nowadays Artificial Intelligence (AI) is used in many different fields, from identifying diseases based on medical data to helping the user typing with its voice. One of the most known AI technique is machine learning, specifically one of its sub-classes, Deep Neural Networks (DNN). They typically suit applications based on image, video or speech processing because of their accuracy in recognizing specific objects. Neural network works in two phases: the first one is training, where the model learn to recognize the target of the application. The second one, instead, is the inference, which is the actual usage of a NN on the field. Training requires more computational power and resources than inference, but, while this phase can be executed on powerful machines, inference is commonly performed on edge devices. This implies meeting stringent constraints on computational and memory resources as well as on energy efficiency.

Neural networks are composed of layers. The first is the input one, the last is the output one while all the intermediate ones are called hidden layers. DNNs present many hidden layers of three main kinds: convolutional, pooling and fully-connected. The majority of the computations are performed in convolutional layers, which are based on the convolution operation. Straightforward convolution is not the most efficient operation to implement on hardware, since it is based on several nested loops dependent on the size of the operand matrices. Luckily, it can be mapped to a general matrix multiplication (GEMM), which was highly optimized during the years. To effectively speed up the processing, custom hardware accelerators are adopted, as they are able to effectively parallelize the computation reducing the run time. Spatial architectures are an example of such accelerators, especially systolic arrays. They are a sub-class of spatial architectures whose data flow recall the one of the convolution. As new solutions are constantly developed, in particular based on reconfigurable platforms like FPGAs, agile automated tools are needed to quickly explore the design space. To this end, this work presents a Python-based framework to generate tensor systolic arrays.

The framework exploits the metaprogramming capabilities of an HDL embedded in Python to reduce the effort required to design and verify the architecture. In fact, the whole structure of the array, the control units and the integration tests are automatically generated by the framework allowing the user to focus only on the development of new processing elements. The framework is used to perform a design space exploration showing the effects of different knobs like data bitwidth, array size, PE structure and sparsity support on resource occupation, power consumption and latency. Results reveal non-trivial trade-offs, motivating the need of developing agile automated tools to reduce design and verification efforts. This work is structured as follows:

- **Chapter 2** briefly explains neural networks structure and presents the convolution operation, detailing how to speed up its execution on CPU-based systems. Moreover, it shows the spatial architectures as a solution to execute NNs.
- **Chapter 3** focuses on systolic arrays, reporting several examples in both industry and academia.
- **Chapter 4** details the structure of the framework.
- **Chapter 5** assess the quality of the spatial architectures produced by the proposed framework, analyzing the results of a design space exploration.
- **Chapter 6** concludes this work presenting interesting future directions.

Chapter 2

Neural networks

Artificial intelligence (AI) is now part of the everyday life, so much that is used without thinking, even though it is the outcome of many years of research. AI includes all those techniques allowing a computer or a machine to mimic the decision-making capabilities of human beings. One of the most know AI branch is machine learning (ML), which uses data and algorithms to imitate the human learning process, gradually increasing its accuracy. Machine learning, in turn, is composed of many different sub-fields and Neural Networks (NN) is one of the most popular. NNs were first proposed by Warren McCullough and Walter Pitts in 1944, but, given the computation cost, the technology at that time was not able to afford it. Another issue was the lack of data sets required to train a NN. In the last decades, computers and, in general, technology improved a lot, making feasible the usage of those previously proposed models. Moreover, thanks to the "big-data phenomenon", data sets large enough to train a NN can be retrieved without much effort. That is why artificial intelligence in now at the basis of many fields and everyday tasks. Few examples of applications are:

- Speech recognition systems: it is the capability to process human speech into a written format.
- Image and video processing: computer vision aims at extracting meaningful information from an image of a video to take action based on the inputs.
- Recommendation engines: by processing the past behaviour data, the algorithm can extract trends useful to tune the results showed by the engine.
- Medical field: AI systems are able to efficiently process medical data like radiology imaging to detect possible diseases.

2.1 Training and inference

Before being able to return the expected results, neural networks have to be trained. *Training* is the first phase that a NN has to go through, while the second one is the *inference*, that is the actual prediction. As detailed in the following section, neural networks consist of many neurons performing operations on weighted inputs. Weights are not hard-coded, on the contrary, they have to be determined based on the application where the NN is used. Training aims at tuning the weights so that the final predictions are as accurate as possible. The accuracy of a NN can be assessed by a loss function, hence the problem of training a neural network can be seen as the problem of minimizing the loss function. The mean squared error (MSE) is usually adopted. Its equation is the following:

$$MSE = \frac{1}{2m}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2 \tag{2.1}$$

In equation 2.1, m is the number of samples, i is the index of the sample, $\hat{y}$ is the predicted outcome and y is the actual value.

The algorithm used to minimize the loss is *Gradient Descent*, which allows the model to choose the direction to take to reduce the error. By applying this algorithm, weights are updated at each iteration as follows:

$$w_j^{t+1} = w_j^t - \alpha\frac{\partial L}{\partial w_j} \tag{2.2}$$

The weight at iteration $t+1$ is computed by subtracting from the weight at iteration t the derivative of the loss function with respect to the weight itself, multiplied by the learning factor α. The partial derivatives are computationally heavy to calculate, hence they have to be estimated by computing the error attributable to each neuron composing the network. This is done by employing a backward propagation algorithm.

Weights are commonly computed using the floating-point representation but, due to the computational resources required to perform floating-point operations, they are usually converted to integer numbers. This final step is called *quantization*.

Training can be of different types depending on the target of the application and on the data set used.

- Supervised learning: it takes labeled data which are used to train the NN comparing the results with the labels.

- Unsupervised learning: data is unlabeled and it is mainly used to discover common patterns within the data set.

- Semi-supervised learning: it takes a data set composed of labeled and unlabeled data.

Due to the calculations required to find the NN weights, training is the most computationally heavy phase. Inference, instead, is a much lighter phase which takes the trained model and classifies the given inputs. If the NN is carefully tuned, it reaches good results in accuracy without much effort.

2.2 Neural Network structure overview

NNs are composed of many layers of neurons connected to each other. In general, there is an input layer, different kinds of hidden layers and the output layer, as figure 2.1 shows.

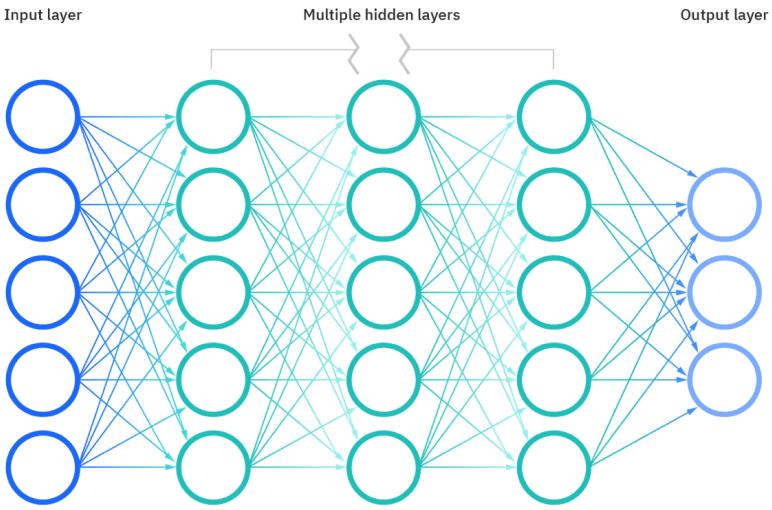

Figure 2.1: Structure of a Neural Network. Image taken from [1].

The structure of the hidden layers depends on the type of neural network. Three kinds are popular today:

- Multi-layer Perceptrons (MLP): each layer takes all the outputs of the previous one computing a non-linear function of their weighted sum.
- Convolutional Neural Networks (CNN): each layer is a set of non-linear functions of weighted sums of spatially nearby subsets of outputs of the previous layer.
- Recurrent Neural Networks (RNN): Each layer is a combination of non-linear function of the weighted sums of the outputs of the previous layer and the previous state.

NNs are inspired by the human brain as they consists of neurons, the nodes, and synapses, the connections between the nodes. Each neuron receives a set of weighted inputs from the previous layer, it performs the sum which goes though the activation function and then it sends the output to the subsequent layer. Figure 2.2 shows an overview of the neuron structure.

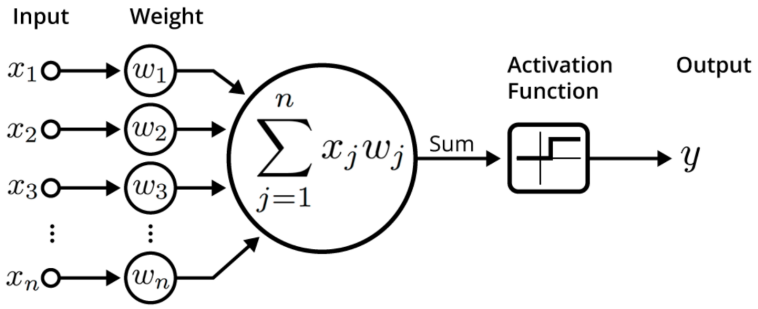

Figure 2.2: Structure overview of a neuron. Image taken from [2].

The activation function is used to determine the output, specifically, if the result of the sum exceeds a certain threshold, the neuron activates passing the data to the next layer. There are different types of activation functions, therefore only the commonly used ones are presented.

Threshold function. Threshold function outputs different values depending on whether or not the input signal is above a certain threshold. One example is the unit step function which returns the following output:

$$f(x) = \begin{cases} 1, & \text{if } x \geq 0 \\ 0, & \text{if } x < 0 \end{cases} \tag{2.3}$$

Sigmoid function. Sigmoid function accepts any input value but returns an output between 0 and 1. The mathematical definition is the following:

$$f(x) = \frac{1}{1 + e^{-x}} \tag{2.4}$$

Rectifier function (ReLU). Rectifier function returns the input value only if it is greater than 0, otherwise the outcome is 0. It is defined as follows:

$$f(x) = \max(0, x) \tag{2.5}$$

Hyperbolic Tangent function. Hyperbolic function is similar to sigmoid but its outputs values fall between -1 and 1. Its output is computed as follows:

$$f(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{2.6}$$

Neurons are the basic blocks composing neural networks and, as mentioned before, they are grouped in layers. While the input and the output layers are always present, the hidden ones may vary in kind and number depending on the type of the neural network. Since the focus of this work is on the Deep Neural Networks, the following section describes in details the hidden layers composing DNNs.

2.2.1 DNNs layers

Deep Neural Networks are one of the most known examples of neural networks. They slightly differ to the classical ones because of the depth of their hidden layers, in fact NNs are usually composed of fewer hidden layers than DNNs. There exist many types of deep neural networks, which are composed of different configurations of the intermediate layers. Besides this fact, hidden layers belong to three main categories:

- Convolutional layers
- Pooling layers
- Fully-connected layers

DNNs layers may be freely combined, but the first one is always a convolutional one, while the last is a fully-connected one. Convolutional and pooling layers may be preceeded and followed by any other type of layer, while the fully-connected one must be the last before the output. In general, the first layers are used to identify simple features, for example, in an image, colors or edges, while the closest to the output are used to detect larger objects.

Convolutional layers

Convolutional (CONV) layers are the core of the computation and where the majority of the operations are performed. CONV layers applies a series of filters to the input matrix to extract a feature map. A filter, also known as kernel, is moved across the input matrix so that, for each considered area, the dot product is computed, generating the corresponding output matrix element. Figure 2.3 shows an example of this operation.
In this case, the element in position (0,0) of the output matrix is computed as:

$$\begin{aligned} Output[0][0] &= (9*0) + (4*2) + (1*1) + (1*4) + (1*1) + (1*0) + (1*1) + (2*0) + (1*1) \\ &= 0 + 8 + 1 + 4 + 1 + 0 + 1 + 0 + 1 \\ &= 16 \end{aligned}$$

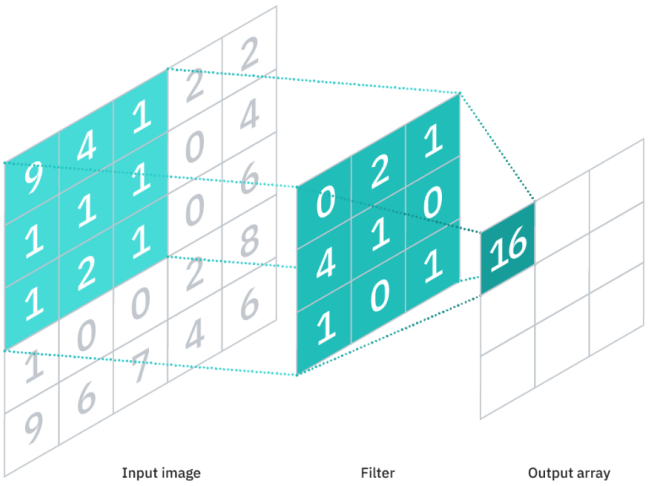

Figure 2.3: Convolution example. Image taken from [3].

A filter is then moved to compute another element until the whole output matrix is calculated. While weights values are adjusted during training phase, there are few parameters that are fixed before it. The first one is *stride*, that is the distance by which a filter moves over the input matrix. In the example before, kernel is moved by one position to compute element (0,1), that means stride is equal to 1. Then, there is the *number of filters* to be applied. This parameter defines the depth of output matrix. The last one specifies the *padding* strategy to adopt. When the filter size does not fit the input matrix, elements falling outside of the input matrix have to be set to some value, usually 0. There are three kinds of padding strategies:

- Same padding: it ensures that the output matrix has the same size of the input one
- Valid padding: no padding is applied. The output matrix is smaller than the input one.
- Full padding: zeros are added to the border of the input matrix so that the output one has a larger size.

Pooling layers

Pooling layers are in charge of reducing the size of the matrices to decrease the computational resources required to perform the operations. Similarly to the convolutional layers, a filter is moved across the input matrix, but, instead of being composed of weights, it applies an aggregation function. There are two types of pooling:

- Max pooling: from each patch covered by the filter it selects the maximum value to send to the output.
- Average pooling: it computes the output as the average of the values covered by the filter.

Fully-connected layers

As the name says, all the nodes are connected to the output ones as it happens with multi-layer perceptrons. Fully-connected layers consist of neurons that apply a non-linear transformation to the inputs through an activation function. In general, they are used to extract even more features from the results of the previous layers.

In conclusion, DNNs are composed of many layers of three main kinds: convolutional, pooling and fully-connected. Besides they target different objectives, convolutional and fully-connected layers both rely on convolution. Since it is a computationally heavy operation, the following sections describe it in detail providing several strategies to speed up the processing.

2.2.2 Convolution operation

The previous section provides an overview of the convolution operation, that is the basis of the actual computation performed by DNNs. Figure 2.4 shows a generalization of this operation, where both filters and inputs are 3D matrices instead of being 2D as depicted in figure 2.3. A kernel has the same number of channels (*depth*), as the input matrix, while, its height and width are smaller, typically like 3x3 or 5x5. As before, shifting a filter across the input matrix and performing the dot product generates the feature map. Moreover, more than one kernel are usually applied to the input image, hence each time a filter is applied, a new channel of the output matrix is generated. The result is a 3D matrix, where the depth corresponds to the number of kernels. The example shown in picture 2.4 presents an input tensor of size $H_{in} \times W_{in} \times C$, over which kernels of size $K \times K \times C$ are applied. The result of the convolution is an output matrix of size $H \times W \times N$, where N is the number of filters applied.

Starting from this representation, the convolution operation can be derived as a seven nested loop as shown in listing 2.1, where N is the number of batches of multiple inputs, C_{out} is the dimension called N in the example above and C_{in} corresponds to C.

```
1 for batch in 0..N
2   for filter in 0..C_out
3     for channel in 0..C_in
4       for out_h in 0..H
```

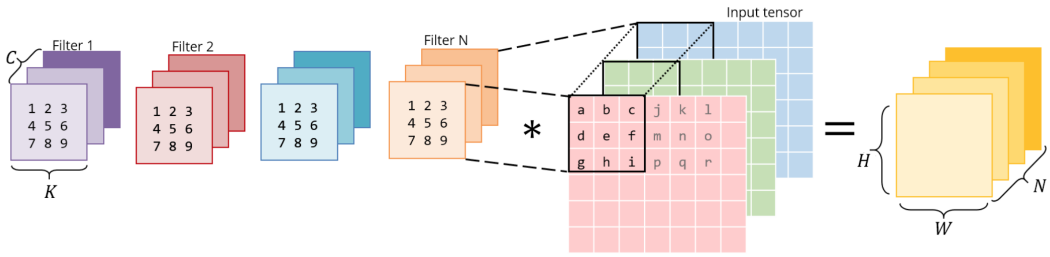

Figure 2.4: 3D convolution example with many filters. Image taken from [4].

```
5           for out_w in 0..W
6             for k_h in 0..K
7               for k_w in 0..K
8                 output[filter, out_h, out_w] +=
9                   kernel[filter, channel, k_h, k_w] *
10                  input[channel, out_h + k_h, out_w + k_w]
```

Listing 2.1: Naive convolution

Practically, the convolution operation can be implemented using different strategies. One of the most adopted exploits GEMM, the GEneral Matrix Multiplication algorithm, by mapping the convolution operation on a matrix multiplication which has been highly optimized over the years. A seven nested loop is not the most efficient way to implement this operation, as it prevents exploiting the data-reuse opportunities of the algorithm, which enhance the temporal and spatial locality of input and output operands. Specifically, the 2D convolution operator presents many properties that can be exploited to speed up the seven nested loop. For example, filters can be reused as the same kernel moves across the input matrix. Hence, by isolating the patches where the filter is applied, the computation can be parallelized. Another exploitable property is that, even if many different filters are applied, the patches considered are always the same. This leads to another reuse possibility which can speed up the whole computation. The architectures used to execute a convolution have to exploit as much as possible features like data reuse to ensure a fast processing of convolutional layers. In particular, when dealing with CPUs, the possible improvements rely on a wise usage of the cache and a favorable data layout in memory. The following sections present many possibilities to improve GEMM on a CPU after detailing the mapping of a convolution on GEMM.

2.2.3 From convolution to GEMM

Before analyzing how to map a convolution to GEMM and how to speed it up, a few details on the layout of the matrices are explained.

Since memories are a linear one-dimensional space, multi-dimensional data cannot be stored as is. Usually, they are saved selecting which dimension is kept contiguous and which not. When dealing with 2D data, two are the possible fashions: *row-major* and *column-major*. Typically, in this kind of applications, the elements are saved following a *row major approach*. When dealing with tensors, as it happens in CNNs, there are 2 more dimensions to deal with, the depth of the matrix and the number of output channels. For this reason, two are the possible ordering of the elements in memory: *NCHW* and *NHWC*. As picture 2.5 shows, considering a tensor composed by N blocks, with C channels of $H \cdot W$ 2D matrices, in the former case the elements belonging to the same block are saved sequentially. In the latter, the 2D matrices on the same channel are saved in sequence.

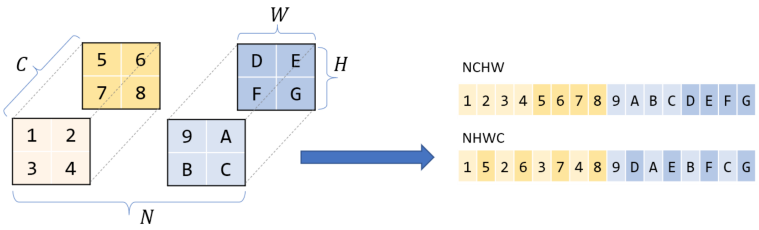

Figure 2.5: Tensor layout in memory. Image taken from [4].

Convolution is, after all, a dot-product between the filters and the input matrix. If the kernels and the input data are laid out in a 2D space as in figure 2.6, the result of the matrix multiplication between the two matrices below is the one of the convolution. The first operand matrix is composed by flattening each filter on a single row. The first channel is laid on the first section of the row and so on. The second operand, instead, is generated by taking the input tensor, identifying the patches covered by the filters and organizing them in columns. Each column is composed by the same patch coming from different channels. The mapping from convolution to this representation is called *im2col*, as it takes an image, an input matrix, and encodes it into columns. The result of the matrix multiplication of these two matrices is the output matrix produced by the convolution. In this way, convolution can be transformed into a matrix multiplication.

2.2.4 General Matrix Multiplication

The general matrix multiplication is defined as $C_{M\times N} \mathrel{+}= A_{M\times K} * B_{K\times N}$, whose pseudo-code is presented in listing 2.2.

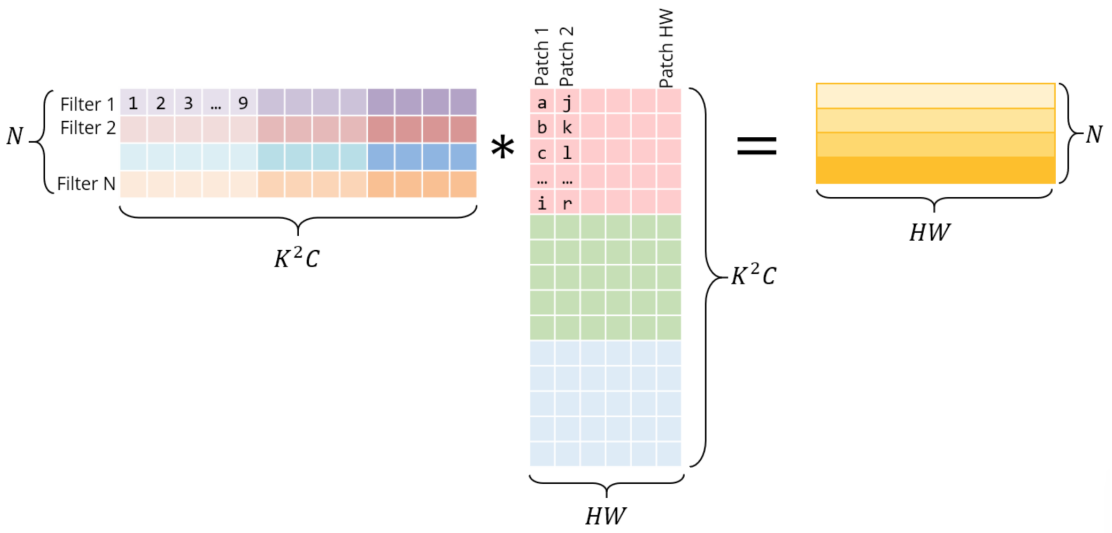

Figure 2.6: Im2col representation. Image taken from [4].

```
for i in 0..M
  for j in 0..N
    for k in 0..K
        C[i, j] += A[i, k] * B[k, j]
```

Listing 2.2: Matrix multiplication.

This nested loop performs the innermost operation $M * N * K$ times. This computation requires not only the ability to perform this kind of operation fast, but also a way to retrieve data as fast as they are processed.

Locality

Usually, storage systems have a hierarchical structure which presents fast but small and expensive memories, like caches, followed by larger and cheaper but slower memories, like RAMs. To exploit at most the fastest memories, data reuse have to be maximized as much as possible. One technique commonly used is *loop reordering*. When working with nested loops, two iteration variables can be swapped without changing the final output but improving data reuse. As an example, consider a matrix multiplication executed on a CPU where cache memories are available. Caches load an entire row at a time, hence, to exploit all of it, a matrix should be stored in memory so that all the elements belonging to that row are used in the computation before other data is loaded. Consider the layout represented in figure 2.6 and the algorithm in 2.2, where A corresponds to the filters and B is the input matrix. At each iteration, the matrix multiplication algorithm multiplies a row of A by a column of B, accumulating the results to compute the corresponding output

matrix element. If data in A is stored in row-major, each time i is incremented, is likely that the element $A[i+1,k]$ has already been loaded in cache. This does not hold for the input matrix B, since it has to be traversed column by column. Figure 2.7 shows a visual representation of this procedure.

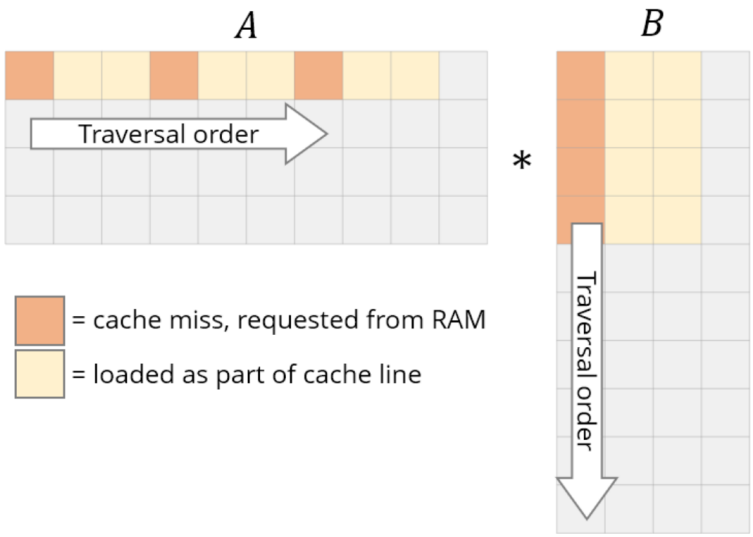

Figure 2.7: Row-major storage and cache utilization. Image taken from [4].

Therefore, a first improvement is rearranging the loops to iterate over all the element of the input matrix loaded in a row. The algorithm is presented in listing 2.3.

```
for i in 0..M
  for k in 0..K
    for j in 0..J
        C[i, j] += A[i, k] * B[k, j]
```

Listing 2.3: Matrix multiplication with loop reordering.

Tiling

Another issue given by the size of the cache is that only few data belonging to a matrix's row can be loaded at a time. Since in a matrix multiplication the same rows and columns are used many times, data previously loaded may be evicted before another iteration of the operation uses them again. *Tiling* can be used to solve this problem. Instead of working on the whole output matrix, the computation is performed on submatrices, so that all data required to fully compute one of them can be loaded in cache. The corresponding algorithm together with a visual representation are provided in listing 2.4 and in picture 2.8.

```
1
2  for i_t in 0..N/TileI:
3    for k_t in 0..K/TileK:
4      for j_t in 0..J/TileJ:
5        for i in 0..TileI:
6          for k in 0..TileK:
7            for j in 0..TileJ:
8              C[i_t*TileI+i ,j_t*TileJ+j] +=
9              A[i_t*TileI+i, k_t*TileK + k] *
10             B[k_t*TileK + k, j_t*TileJ+j]
```

Listing 2.4: GEMM exploiting tiling.

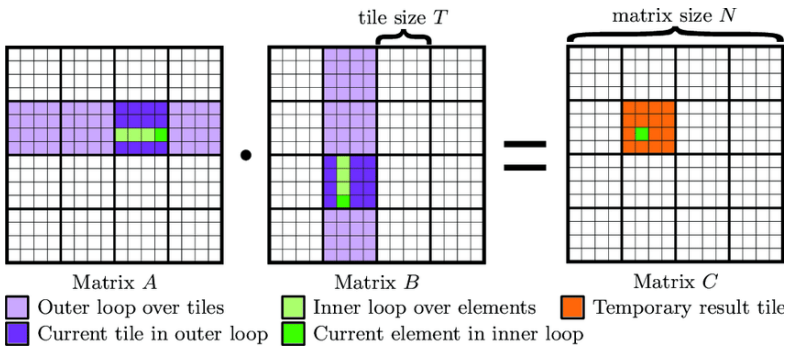

Figure 2.8: GEMM using tiling. Image taken from [5].

Vectorization

Exploiting efficiently small memories like caches is not the only way to improve GEMM. In fact, arithmetic intensive operations like matrix multiplication can be decomposed into vector operations. In DNNs, the same operation is performed on different data. Instead of computing element by element, input data can be vectorized transforming a scalar operation into a vector one. Figure 2.9 shows an example. Instead of computing a scalar operation on 4 data in 4 CPU cycles, with vectorized data only 1 CPU cycle is required to compute a vector operation. This transformation helps saving CPU cycles.

Deep Neural Networks require an arithmetic-intensive processing of thousands of data, therefore CPUs may not be the best platform choice. Lately, custom hardware accelerators were developed to afford the computational requirements of DNNs. An example are spatial architectures. They are composed of a mesh of many processing elements guaranteeing data reuse though an internal network able to parallelize the computation. The following sections better detail their structure.

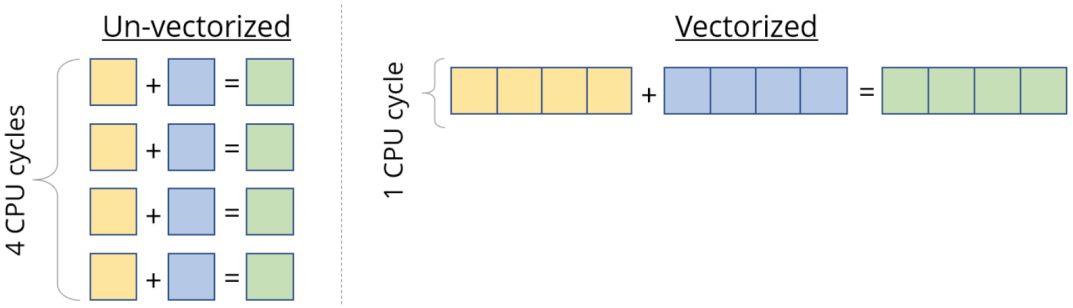

Figure 2.9: Data vectorization. Image taken from [4].

2.3 Hardware acceleration

2.3.1 Spatial Architectures

CNNs are composed of many layers, but, in most of the cases, the convolutional ones account for the the majority of the overall operations count. Therefore, their processing has to be extremely efficient. To do so, two aspects of the computation have to be carefully considered. The first is that, due to the structure of a convolutional layer, the same input data have to be shared among different computations. In fact, the same filters are used over the whole input matrix, adjacent windows share some input elements if the stride is smaller that the filter size and the same patch is covered by many filters generating output elements on different channels. Hence, there has to be a mechanism to share data previously loaded without accessing the memories each time an element is required. Moreover, the partial sums computed have to be forwarded to the next processing element to continue the computation. The second, instead, regards the shape of the data processed. Different layers normally work with matrices with different parameters, like spatial size or number of input and output channels. Therefore, the accelerator has to be flexible enough to support many possible configurations. Spatial architectures are one of the adopted solutions because they can provide a high compute parallelism thanks to the direct communication between the processing elements composing an array. Moreover, they can be programmed to support different algorithms mapped as different dataflows.

Spatial architectures are a class of accelerators designed as an array of simple processing elements (PEs) connected in such a way to provide a high compute parallelism. To speed up the computation and to not depend too much on the memory latency, a distributed memory system is exploited. Moreover, PEs are interconnected with a cheap distributed routing fabric. There are two types of spatial architectures: fine-grained and coarse-grained. The first one is typically in

the form of an FPGA, while the second one is composed of a set of PEs connected with an on-chip network. The coarse-grained spatial architectures are the ones commonly used because of their organization into multiple PEs which parallelizes the computation, while their interconnections makes possible to efficiently forward data to be reused or further elaborated.

A spatial architecture is usually composed of a global buffer and an array of processing engines (PEs), wich can be designed to support different algorithms. Data is moved to and from the array through FIFOs, which communicates with the off-chip DRAM and the global buffer. This last component is used to exploit data reuse to hide the DRAM access latency. When dealing with CNNs, the PEs are programmed as ALUs connected together via on-chip networks. The array internal interconnection is effective since the partial sums can be passed from a PE to another and the inputs can be shared among all of them. The datapath of the PE performs multiply-and-accumulate (MAC) operations, it includes a register file used as scratchpad, and a FIFO to control the data going in and out of the PE. Figure 2.10 shows the block diagram of a generic spatial architecture.

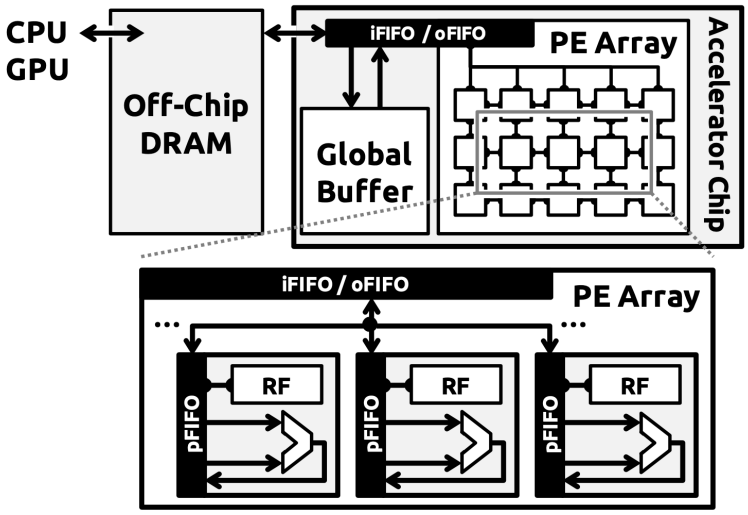

Figure 2.10: Spatial architecture's block diagram. Image taken from [6].

2.3.2 Design-space parameters

There are several different DNNs accelerators, especially since the systolic arrays have regained interest in the latest years.

For example, NeuFlow [7] designed a systolic array that relies on a mesh of processing elements, each of them communicating with its neighbors through FIFOs.

The architecture presented also provides a configuration bus to reconfigure the hardware efficiently at run time, to optimize the execution of a wide variety of neural networks.

DianNao [8] accelerator, instead, is composed of custom inner-product units. The target of the research team was developing an accelerator where memory transfers are minimized and performed as efficiently as possible.

Eyeriss team [6] chose to focus on the dataflow, highlighting its importance and proposing a new one, called row-stationary. This new dataflow aims at optimizing the energy consumption of all types of data movements.

Interstellar's [9] researchers, adopted a different approach by exploiting Halide [10], a domain-specific language for high-performance operator development. They extended the compiler to generate different hardware architectures by simply changing the Halide schedule associated with the same high-level program specification. The design space exploration performed with the generated accelerators showed that dataflow choice is not so important as long as data reuse and resource utilization are maximized. Moreover, they highlighted the importance of optimizing memory hierarchy as it directly influences energy efficiency.

Each of the previously mentioned solutions explores a different section of the design-space, demonstrating how the parameters considered affect the efficiency and the performance of the architectures. Specifically, the design-space that has been widely studied is composed of the following parameters:

- Dataflow
- Resource Allocation
- Loop blocking

DNN and CNN accelerators usually exploit the parallel execution of the operations on different processing elements to speed up the processing. The *dataflow* scheme dictates how units communicate with each other and how data is accessed. In general, a dataflow aims at reducing as much as possible the accesses to the slowest memories by carefully controlling data forwarding. A more detailed description of the possible dataflow schemes is provided in the following section.

Performance and efficiency of an accelerator depends on resource allocation, too. The size of the PE array and memory hierarchy affect the final throughput and cost in terms of energy and latency to access memory. Since the cost of each memory access grows with the memory size, it is important to design them as small as possible, but, if a memory is too small, the eviction rate increases. Therefore, they have to be large enough to guarantee that all data required is ready for processing, but not larger than what workload requires, otherwise energy is wasted.

Given the cost of accessing memories, it is better to schedule the computation to maximize the data reuse. This can be done by reorganizing the nested loop shown

in 2.1, to exploit data loaded into the smaller memories as much as possible before evicting them. In other words, the same principles applied to the high performance GEMM in section 2.2.4 can be exploited to improve memory usage in a spatial architecture.

Dataflow schemes

As previously mentioned, one of the characteristics of a DNN accelerator is dataflow, which describes how the computation is performed and how data is forwarded. When performing a convolution, there are three main data reuse possibilities. First, the same filter is shifted over the input matrix and each computation window generates an output matrix element. Second, adjacent windows share some input elements if the stride is smaller than the filter size. Third, when computing the output elements in different channels, the same input window is used for different filters. Thus, three main dataflow strategies can be identified based on which data stay stationary at the PE level:

- Output stationary
- Weight stationary
- Input stationary

Figure 2.11 shows the output stationary scheme. Each processing element is in charge of computing a single output element. Matrices' elements are fed from the edges of the array and then internally forwarded to the other PEs. Once the computation is done, results have to be shifted out. In the meantime, no other calculation can be done. This approach aims at minimizing the accumulation cost since the partial sums are not moved between the PEs. It is particularly convenient when data is asymmetric, i.e., when the accumulation is performed on more bits than the input ones.

The weight stationary scheme is represented in figure 2.12. The array is pre-filled with weights that remain stationary throughout the whole computation, maximizing filter reuse. Elements of the input matrix come from the left edge, in this way, each PE generates a partial sum at every clock cycle. The generated partial sums are then reduced along each column in parallel to generate one output element per column. If tiling strategy is applied, an external accumulator may be required to compute the final output.

The input stationary scheme is quite similar to the weight stationary one, but the inputs are pre-loaded into the array instead of the filters' weights. The array is pre-filled in such a way that each row has the elements required to compute a given output element.

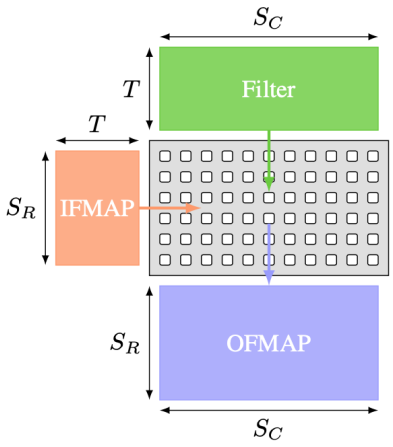

Figure 2.11: Output stationary dataflow. Image taken from [11].

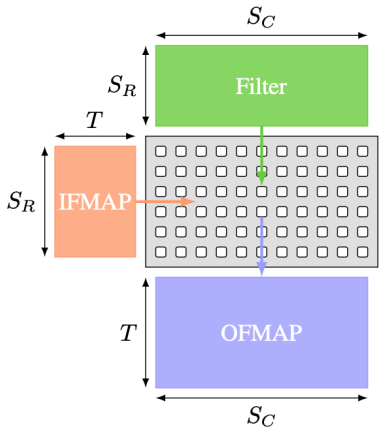

Figure 2.12: Weight stationary dataflow. Image taken from [11].

Other more complex mappings of a convolution on a spatial architecture have been lately proposed to exploit additional reuse. One example is the row stationary dataflow adopted in Eyeriss [6] accelerator, which aims at optimizing all types of data movement energy consumption. In particular row stationary maximize filters and feature map reuse while minimizing partial sum accumulation costs.

It was possible by minimizing as much as possible all data movements through the exploitation of PE local storage and the internal interconnections. With this architecture, the research team was able to achieve better results in terms of energy consumption when running configurations of AlexNet [12]. However, more complex dataflow like row stationary complicates the overall system. In fact, the memory hierarchy has to adapt to the complexity of the dataflow, the routing fabric

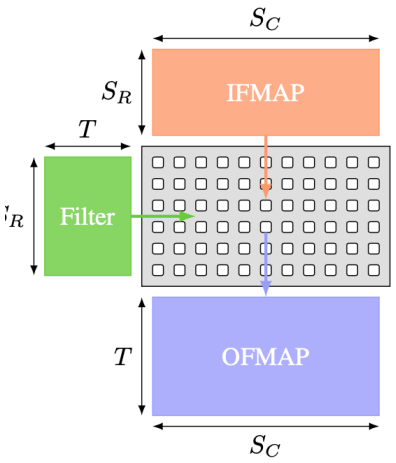

Figure 2.13: Input stationary dataflow. Image taken from [11].

has to be more flexible and the control units have to handle a more complicated synchronization of the computation.

However, as highlighted in [9], dataflow choice is not crucial as long as data reuse and resource utilization are maximized. Instead, memory hierarchy and the effective number of computational resources actually affect energy efficiency.

Chapter 3

Systolic Arrays

Systolic arrays belongs to the class of the spatial architecture accelerators. They offer a simple, regular and modular structure which is easily adjustable to the desired application reducing the design cost. Moreover, a systolic array is composed of many processing elements parallelizing the work and it provides a simple internal communication network which does not require complex control logic to manage the synchronization of the computation. These features make systolic arrays an interesting solution on which GEMM, and consequently convolution, can be efficiently mapped.

3.1 Systolic array overview

As a spatial architecture, a systolic array is composed of a mesh of processing element performing multiply-and-accumulate operations. Each PE communicates with its neighbors to forward the inputs and the computed data. As the name suggests, data flows in the array in a systolic way, which means that, at each clock cycle, data is moved from a PE to another, from the memory to the PEs directly connected to it and from the last PEs to the memory again. Figure 3.1 shows how data flows in a set of PEs. It is first loaded in the left-most unit. At the next clock cycle it moves in the right PE an so on, until it is shifted out from the last processing element and goes back in memory. Data like weights in the picture below, moves with the same mechanism but from the top of the array to the bottom.

An example of this behaviour is represented in figure 3.2, which shows the computation on a systolic array adopting the output stationary dataflow. During a clock cycle, each PE performs the MAC operation and then forward the data to the neighbors. Inputs and filters are fed from the edges of the array following a skewed approach. This means that, in the first clock cycle of the computation, the

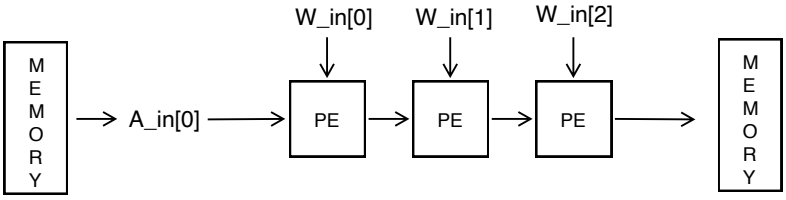

Figure 3.1: Data movement.

elements going into the top-left PE are sent to the array. At the following clock cycle, the top-left PE continues to be fed with data, together with the bottom and right neighbors, and so on.

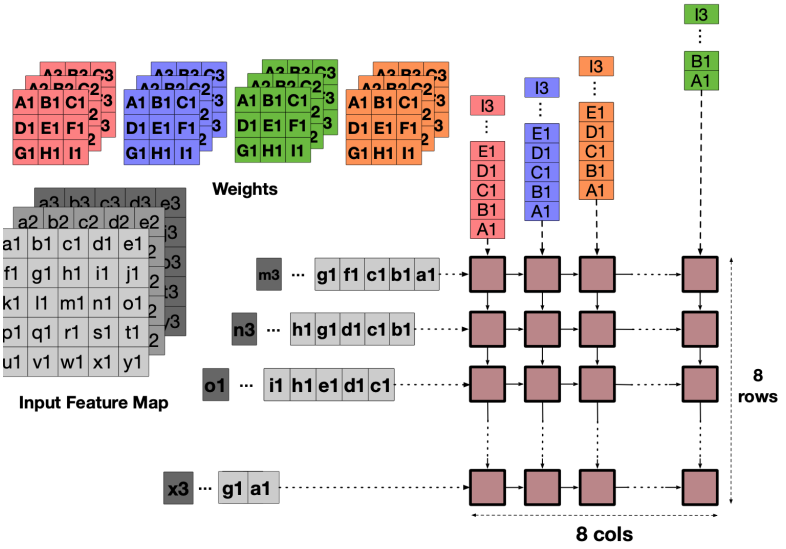

Figure 3.2: Skewing mechanism on a output stationary scheme. Image taken from [11].

3.2 Analytical model

To understand the effectiveness of this kind of architecture, SCALE-SIM's [11] research team proposed the Systolic Array Simulator (SCALE-SIM). SCALE-SIM is a simulator developed to allow designers to quickly explore the design-space searching for the optimization points based on the chosen architectural parameters. It provides cycle-accurate timing, power/energy, memory bandwidth and trace results for a specified accelerator configuration and neural network architecture,

letting users to customize various micro-architectural features such as array size, scratchpad memory size, dataflow mapping strategy, as well as system integration parameters such as memory bandwidth. Figure 3.3 shows a schematic of SCALE-SIM with an example of inputs and outputs.

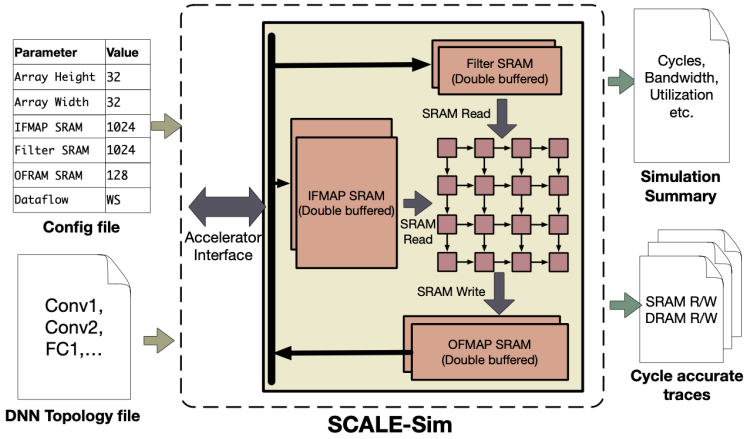

Figure 3.3: Schematic of SCALE-SIM showing and example of inputs and outputs. Image taken from [13].

The choice of these parameters is crucial to obtain an efficient accelerator. In particular, the results obtained by using SCALE-SIM focus on three main aspects:

- Effect of dataflow
- Effect of memory size
- Effect of shapes

As previously detailed, a dataflow scheme dictates the mapping strategy of the computation. Researchers were able to demonstrate that the output stationary scheme is the best choice in terms of computation cycles and energy consumed, as long as the OS does not show evidence of stalls. Moreover, the cost of logic within the accelerator is assumed to be the same for the three dataflows, which may not be true.

Data reuse is a key feature in CNNs, hence providing enough on-chip memory can reduce the number of accesses to the off-chip memory. However, memory cost increases with its size. The results show that, in general, there is a threshold of bandwidth above which the gain no longer increases, which depends on the workload considered.

When talking about the efficiency of an accelerator, the shape of a systolic array also matters. It was shown that this aspect, together with dataflow, affects the

results. For example, a short-wide array is more efficient if coupled with output stationary or weight stationary schemes. On the contrary, square shapes are better exploited by the input stationary scheme.

From SCALE-SIM the analytical model is derived. A systolic array deals with operand matrices of size $S_R \times T$ and $T \times S_C$ respectively, where S_R and S_C are the the number of rows and columns of the input matrices, while T is the number of elements in a convolutional window. Considering the ideal case where the array size corresponds to the output matrix size, that means having an unlimited number of PEs, the computation follows the procedure presented in figure 3.4.

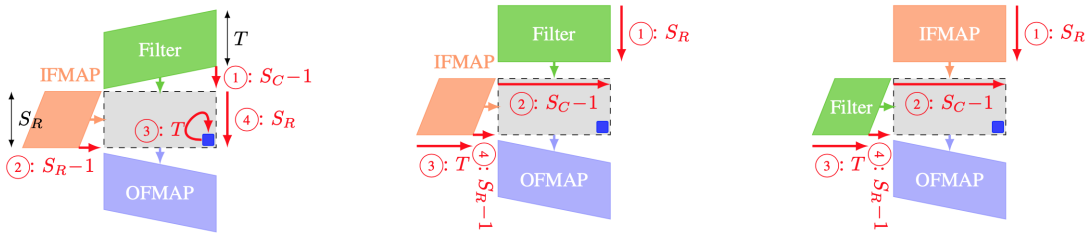

Figure 3.4: Runtime representation for each dataflow. From the left: output stationary, weight stationary and input stationary. Image taken from [11].

Since data comes from the left and top edges, the bottom-right PE is the last one to receive the data. In particular, this happens after $S_R + S_C - 2$ cycles. Each processing element requires T cycles to generate the final output, so the last PE ends its computation after $S_R + S_C + T - 2$ cycles. Moreover, other S_R cycles are required to download the outcome of the computation from the array. Therefore, the total number of cycles required is:

$$\tau_{min} = 2S_R + S_C + T - 2 \tag{3.1}$$

This reasoning holds for all the three dataflow schemes presented in the previous section.

Having a large enough systolic array to cover all the output matrix is quite often not feasible, so computation has to be split in chunks. As figure 3.5 shows, S_R and S_C dimensions are sliced, and each tile of the matrix is computed sequentially.

Therefore, a single systolic array of size $R \times C$ computes the whole output matrix in $\left\lceil \frac{S_R}{R} \right\rceil \cdot \left\lceil \frac{S_C}{C} \right\rceil$ iterations. By taking the previous analysis for the computation of two matrices of size $S_R \times T$ and $T \times S_C$ respectively on a systolic array of size $R \times C$, with $R = S_R$ and $C = S_C$, the resulting runtime is:

$$\tau = (2S_R + S_C + T - 2)\left\lceil \frac{S_R}{R} \right\rceil \left\lceil \frac{S_C}{C} \right\rceil$$

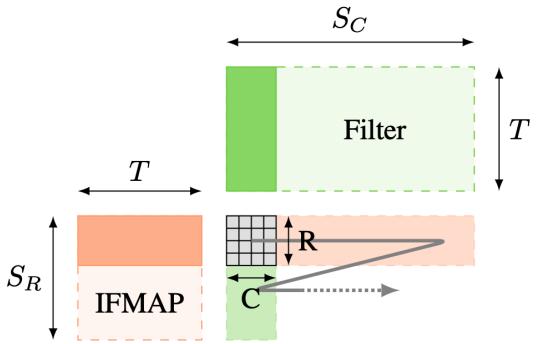

Figure 3.5: Computation sequence. Image taken from [11].

3.3 Systolic arrays in industry and academia

Systolic arrays have been deeply studied over the latest four decades and they gained attention both in industry and academia. This section reports two accelerators belonging to the two aforementioned fields.

3.3.1 Tensor Processing Unit

In 2015, Google made the first step toward the usage of these kind of architectures to run a DNN: the Tensor Processing Unit (TPU) [14]. The TPU was designed as a co-processor in charge of running the inference of production-level ML models, such that the interaction with the host CPU was reduced to the minimum. In fact, instead of making the TPU fetch the instructions, the host sends them to the TPU. Its main unit is the Matrix Multiply Unit, a 256x256 weight-stationary systolic-array. The inputs of the unit come from the Weight FIFO and the Unified Buffer, while the computed products are collected into the Accumulators buffer. Figure 3.6 shows the TPU block diagram.

Tensor Processing Unit leverages the reduction in energy and area with respect to the GPU to which the TPU was compared to. In particular, the key features that help reaching such results reside in the structure itself: the size of the matrix-multiplication unit, the software-controlled on chip memory, and the ability to run the inference of a model minimizing the host CPU intervention.

3.3.2 Gemmini

Gemmini [15, 16] is an open-source modular and flexible generator of systolic array accelerators, which supports multiple dataflows and targets ASIC implementations. The ability to generate parameterized architecture allows the user to easily explore the design space, and so efficient hardware and software co-design. It does not

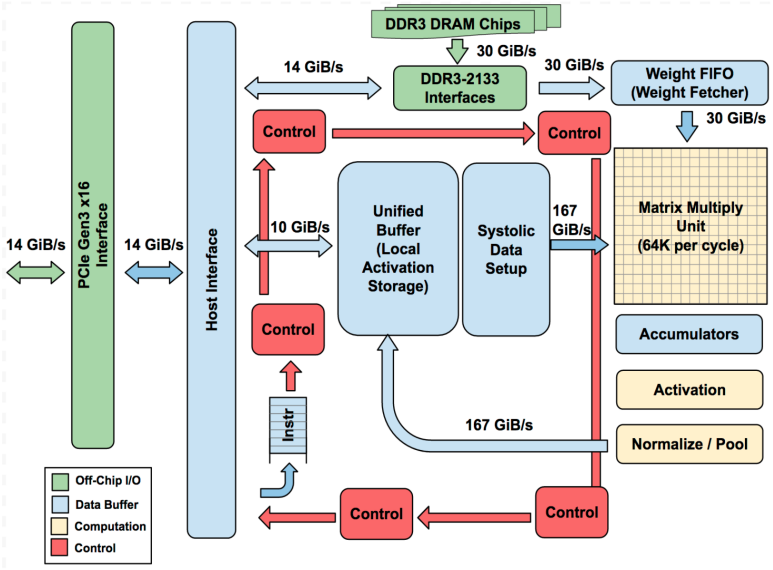

Figure 3.6: TPU block diagram. Image taken from [14].

focus on the systolic array only, but it takes also in consideration the whole system around the array, focusing as well on the interaction with the host CPU. Many parameters are customizable and they are listed below:

- Dataflow: weight stationary or output stationary
- Array dimension: 16x16 or 32x32
- Data bitwidth: 8 bit input 32 bit result or 32 bit input 32 bit result
- Pipeline depth: fully pipelined or fully combinational
- Memory capacity: 64 KiB or 256 KiB
- Number of memory banks: 5 or 33
- Bus width: 128 bits or 64 bits
- Host processor: rocket or BOOM

Figure 3.7 shows an overview of the architectural template.

Researchers demonstrated that specific designs generated by Gemmini were able to achieve two to three orders of magnitude speedup on DNN inference comparing to a CPU implementation. They also highlight the importance of a full-system evaluation, by showing that no significant improvement can be achieved if the entire DNN is not efficiently mapped into the whole system, i.e., host and accelerator.

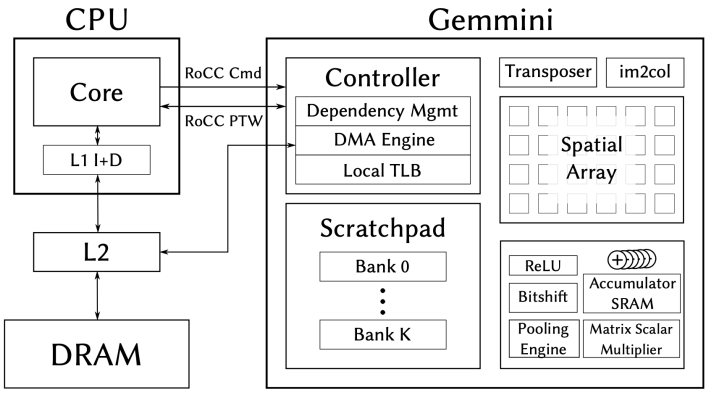

Figure 3.7: General schematic of Gemmini. Image taken from [15].

3.4 Tensor Systolic Array

Tensor Systolic Array (TSA) is a generalization of the classic systolic array composed of more complex processing elements called Tensor Processing Element (TPE). Each TPE takes a tensor of weights and a tensor of activations as inputs to performa small matrix multiplication, instead of a simple MAC between two elements. Specifically, the input matrices have size $A \times C$ and the operation performed is a B-way dotproduct. Figure 3.8 shows an example of a TSA of size 2×4×2_2×2. The size of the array is in the form: $A \times B \times C\_M \times N$, where $M \times N$ is the size of the array, while $A \times B \times C$ is the size of the TPE, in particular B is the dimension of a DP{B} unit. The example shown in figure 3.8 considers a TPE, the red box, which takes two tensors as operands and performs a small matrix multiplication at each clock cycle. DP4 elements, instead, perform 4-way dot-product accumulating into a single register.

The structure of a TPE increases, with respect to the classic SA, the so called intra-TPE operand reuse, which represent the data reuse inside a processing element. The increase of this parameter further amortize data movement. In fact, a whole matrix multiplication is performed inside a TPE. Another important metric is the inter-TPE operand reuse, which represent the array MACs to array input operands ratio. It indicates how much the cost of reading operands from SRAM is amortized. Inter-TPE and intra-TPE operands reuse parameters may drive the choice of a dense tensor systolic array instead of, for example, a sparse systolic array, which is detailed later in this chaper.

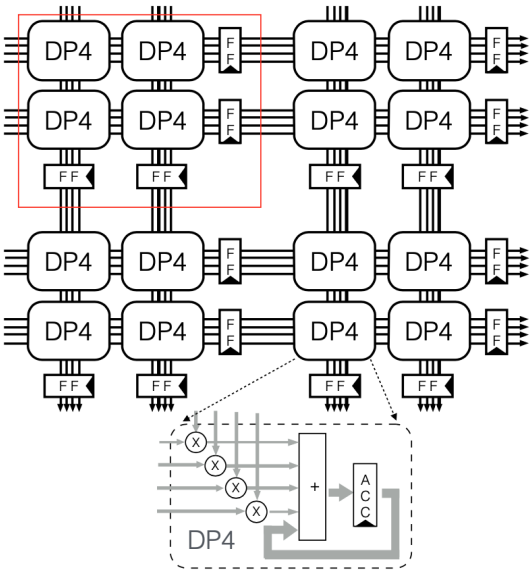

Figure 3.8: Systolic tensor array of size 2×4×2_2×2. Image taken from [17].

3.5 Accelerator/Model CoDesign

With the increasing usage of DNNs, the inference is moved from the cloud to the edge devices that present strict power and compute requirements. To match constraints of this kind of devices, DNNs models have been adapted to execute the inference phase using a lower precision of weights and activations. Thanks to NNs resilience to approximations, their accuracy does not significantly decrease. Moreover, their computation time and resources required drop, allowing the inference to be executed on edge devices. Many solutions were proposed, like quantizing NNs parameters or exploiting data sparsity.

3.5.1 Quantization

Quantization is one of the most common way used to decrease the computational time and energy consumption of neural networks, however it introduces additional noise which may decrease the overall accuracy. The basic idea is that, while training is usually performed working of 32-bit floating point data, inference can be executed considering NNs parameters with lower precision like 8-bit integers. This leads to a faster execution, which reduces memory access cost and increases the computation efficiency. Specifically, lower-bit data requires less data movement which reduces memory bandwith saving energy. Moreover, 8-bit integer arithmetic consumes way less energy than floating point operations. There are two main ways to quantize data:

- Post-training: it trains the NN using 32-bit floating point weights and inputs, then it quantizes weights.

- Quantization-aware training: it quantizes weights during training.

3.5.2 Sparsity

Since CNNs imply heavy computations, exploiting properties like data sparsity would be of great help. In general, CNNs exhibit 50–70% zeros but the saving in computation cost gained by the sparse matrix multiplication (sGEMM) is balanced by the overhead introduced to handle the sparseness. Moreover, zeros are typically distributed randomly. This leads to two issues. First, the non-zero elements have to be indexed explicitly, increasing the overhead. Second, randomness complicates the load balancing mechanism, but this step is necessary to improve the hardware utilization. To speed up the learning process of a neural network, few constraints can be set. One example is constraining the distribution of zero elements. *Block Sparsity* is a way to deal with sparseness, where zero and non-zero elements are grouped into different blocks, like in figure 3.9 (b). This strategy reduces indexing overhead and makes load balancing easier than random sparsity. However, CNNs accuracy decreases as the block size increases. Another alternative is *Density Bound Block* (DBB), which shows the advantages of block sparsity while maintaining CNN accuracy. Density bound block relaxes the constraints of block sparsity since it does not require the zero elements to be contiguous. Specifically, it considers data grouped in blocks and sets a maximum number of elements which can be different from zero. Figure 3.9 shows a visual representation of different sparsity possibilities.

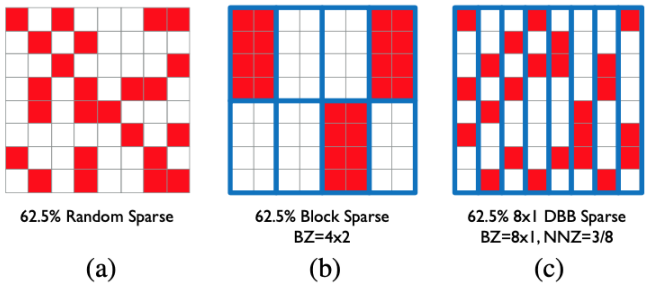

Figure 3.9: Data sparsity. (a) represents random sparsity, (b) shows bleck sparsity while in (c) DBB is presented. Image taken from [17].

This bound also leads to a limitation, that is the sparseness is fixed at design time. In this way, only specific CNN models presenting that percentage of data sparsity are able to fully exploit the architecture.

Even though DBB may be limited by the fixed-sparsity ratio, recent works [18, 17] demonstrated that it can be efficiently exploited in sparse Neurual Networks. Contrary to activations, weights are known in advance and so their sparseness. This can be exploited designing a DBB sparse systolic array exactly tuned for the application considered, where the sparsity ratio corresponds to the sparsity present in weights. As mentioned before, density bound block introduces a constraint on NNZ. A weight tensor is decomposed depthwise, like the blue-highlighted elements in figure 3.10, which compose a block. Each block is then compressed keeping only the non-zero elements and a bitmask is generated, where each index that refers to a non-zero element is set to 1 while the others are set to 0.

Figure 3.10 shows how a tensor of weights is compressed to only store the non-zero elements.

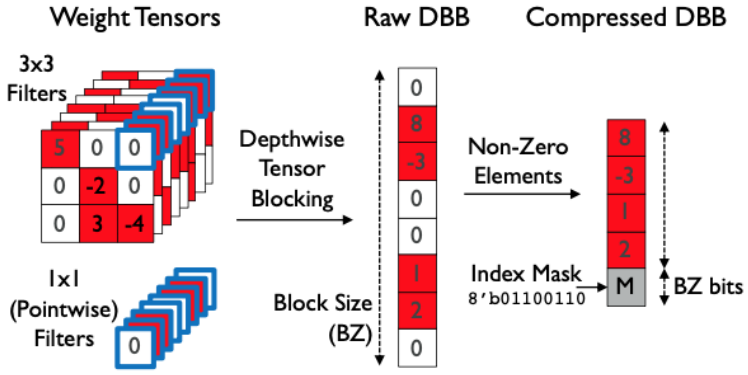

Figure 3.10: How filter tensors are compressed when DBB is used. Image taken from [17].

Even if DBB can be exploited for weights, its sparseness cannot be changed at run time. A more favorable strategy is to develop a variable density block bound (VDBB) approach and this is what the same researchers who studied DBB for filters' data sparsity did. For further details refer to [18, 17]. Figure 3.11 (a) shows a conventional dense datapath, where 8 weight-activation multiplications are performed. When dealing with random sparsity, figure 3.11 (b), it is sufficient to add a clock gating mechanism which detects the zero weights to reduce the energy consumption of the whole architecture. However, many multipliers are not fully exploited, reducing the utilization of the hardware. DBB instead, figure 3.11 (c), works only with NNZ multipliers whose inputs are multiplexed. This is possible thanks to the imposed bound on the non-zero elements in a weight block. The bitmap saved together with data is used as multiplexers' selection signal, so that the correct activations are multiplied to the corresponding weights. Since NNZ is fixed at design time, having a higher sparsity leads to under utilization of the

hardware, as shown in figure 3.11 (d).

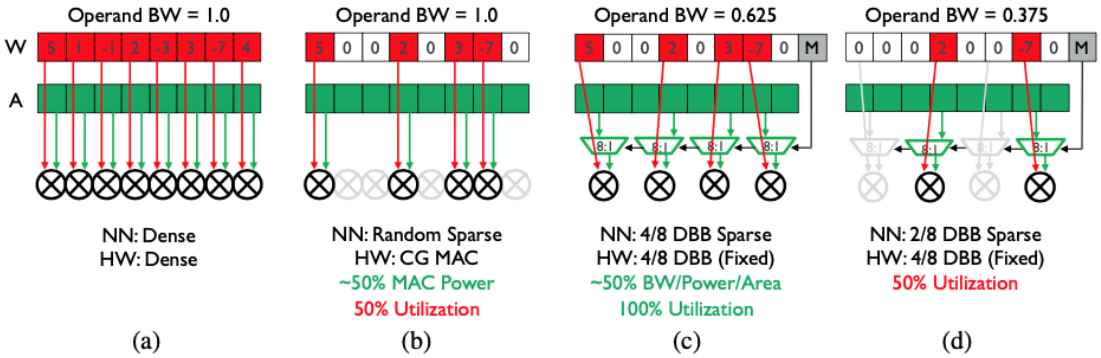

Figure 3.11: Spatially unrolled datapaths. Image taken from [17].

A sparse Systolic Tensor array with DBB support is shown in figure 3.12. The external structure is the same as the one of a systolic tensor array, what changes is the implementation of the TPE. Instead of being composed of DP4 blocks, considering the examples, it consists of 4×4-input 2-way Sparse Dot Product (S4DP2) units, which implements what figure 3.11 (c) shows. In this case the sparsity ratio is set to 50%.

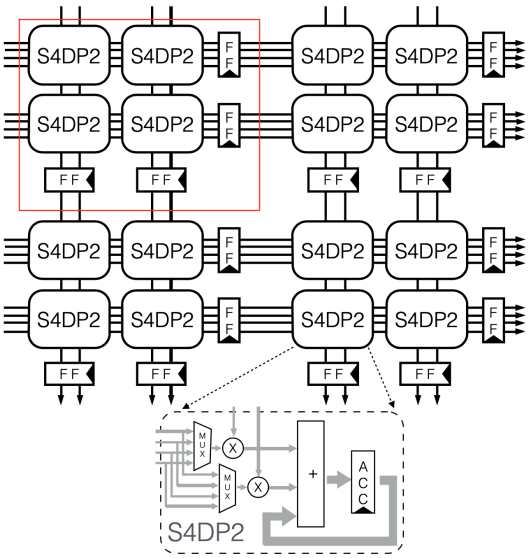

Figure 3.12: Systolic tensor array of size 2×4×2_2×2 with DBB support. Image taken from [17].

Even if DBB presents a fixed-sparsity ratio which may not be suitable to all the

applications, it is quite easy to implement and it increases energy efficiency with sparsity. Section 4.5.1 describes how a sparse systolic tensor array supporting DBB may be implemented exploiting the presented framework.

Chapter 4

Hardware generation framework

Systolic arrays are extremely modular structures that can be parameterized as required by the application. The framework presented in this chapter implements an easy way to build a systolic array and to extend its functionalities. It leverages the metaprogramming capabilities of an HDL language (*Magma*) and a verification framework (*Fault*) embedded in Python, to minimize the design and the verification effort. Specifically, the framework consists of smart systolic array templates, which allow the user to focus on designing and verifying new processing elements, leaving the burden of creating the routing fabric, the control unit, and the integration tests to the generation framework. Before detailing the framework, a briefly overview of magma is presented.

Magma [19] is a hardware construction language embedded in Python. It provides the capabilities of a high level programming language to describe complex, parameterizable circuits generators producing synthetizable Verilog. Its basic abstraction is the circuit, which is a set of components wired together. From this point of view, it is similar to Verilog, but what makes magma a powerful language is the possibility to write programs (generators) which write other programs, in other words, its metaprogramming capabilities. In particular, it allows a designer to describe a structure dependent on the features of an inner module, so that, when the inner module is defined, the whole structure is automatically created adapting to the module specification. For example, in the presented framework, the processing element is the basic block of a systolic array. When a PE is specified, for example as a scalar PE or a TPE, the corresponding systolic array is automatically generated adapting the interconnections interface to the PE specification.

The framework provides smart templates for the generation of different types of PEs and, consequently, of the corresponding systolic arrays. Specifically, the systolic

arrays proposed adopt an output stationary dataflow, which is not a limitation since other dataflows may be supported by simply writing other similar smart templates. Moreover, the framework provides the control units managing the computation on the systolic array generated.

In the following sections, the structure of the framework will be presented in details together with a test suite which is automatically set up when a new design is generated.

4.1 Systolic Array

As already explained in section 4.1, a systolic array is a matrix of *Processing Elements* (PEs). Each of them is capable of performing a specific tensor operation.

This framework is designed to easily generate systolic arrays customizable in the following aspects:

- Number of rows.
- Number of columns.
- PE structure.
- Data bitwidth.

Once a PE is generated, it is used as argument of the function `SysArrayXX`, which requires also the number of rows and columns to build the whole systolic array. `XX` can be `Scalar`, `ACTiling`, `BTiling`, `STA` or `S2TA` depending on the array type.

4.1.1 Data plane

Data is fed from the edges, in particular, the input matrix comes from the left side, while the weights arrive from the top. Each PE performs an operation and forwards data to neighbor PEs on its right and bottom sides. Before the computation, if there is a bias, it is pre-loaded in each PE from the top edge, as shown in figure 4.1.

The systolic array is fed following a skewed approach. Considering an array that takes one element per PE at a time, only data processed by $PE[0][0]$ is sent at the i-th clock cycle. At the next clock cycle, $PE[0][0]$ continues to receive data, while $PE[0][1]$ and $PE[1][0]$ start processing their data and so on, as figure 4.2 shows.

This mechanism can be generalized for a tensor systolic array. With respect to the previous situation, a PE takes a set of data instead of a single element. The skewing process works similarly, delaying the set of data instead of the single

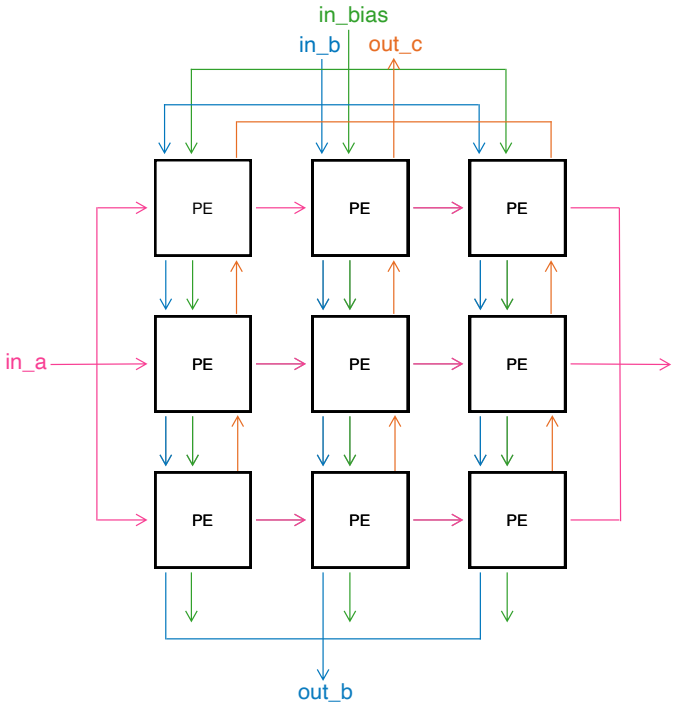

Figure 4.1: Systolic array data flow.

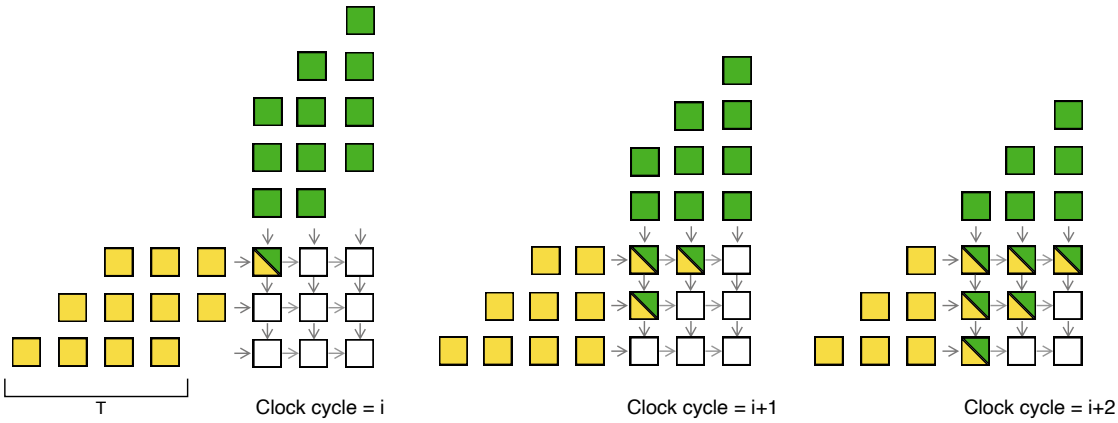

Figure 4.2: Skewed data.

element. The delay is not applied anymore to subsequent elements, but sets. Each input is then forwarded to the neighbor PE on the same row or column. Once the systolic array has performed the computation, the results have to be moved from the array to the external memories. To this end, each PE has a pass through connection that allows data stored in a PE to be forwarded to the next one. The computed output is available starting from the top-left element and the downloading phase

will follow a skewed approach flowing from the bottom to the top. This mechanism allow the new bias to be loaded while data is shifted out from the array avoiding idle cycles between different computations. The overlapping of the two phases requires two different networks to forward bias to the next PE and to move a result through the current PE.

4.1.2 Control plane

In order to correctly handle the computation, a systolic array requires also a control network, which is able to forward the control signals to the neighbor PEs as it happens with the data. The control plane presented is designed to scale efficiently with the size of the architecture. In fact, there is no control signal which is broadcast, instead they all follow a systolic flow, which results in a low cost scaling.

Figure 4.3 shows the connections required for loading the input matrices. They are enable signals, used to sample the data into the PE input and output registers. Further details about the PE will be given in Sec. 4.2.

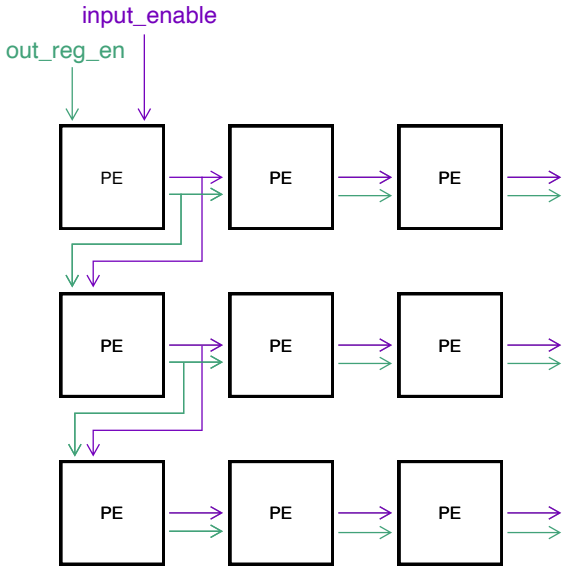

Figure 4.3: Load data control signals.

`load_bias` network was designed as shown in figure 4.4 to correctly drive the PEs internal multiplexers when loading a bias matrix.

A bias matrix comes from the top edge and, because of the skew, the leftmost row will be the first one to be filled. Once a row is full, PEs do not have to store other data as bias. In order to stop the column from sampling what comes from bias port, a signal for each row is required. For example, considering that a PE

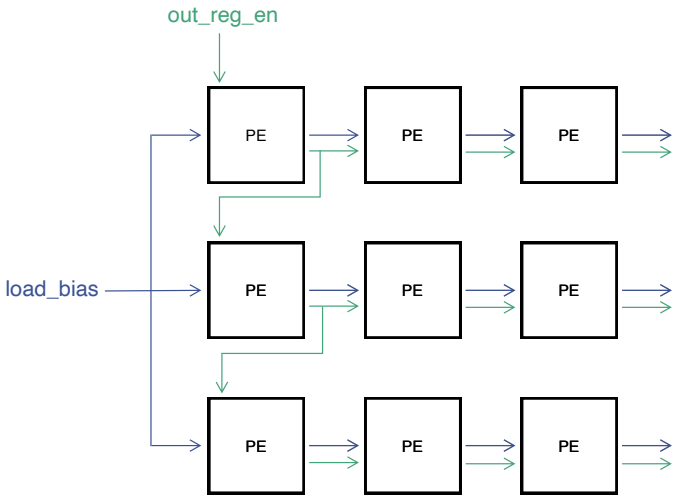

Figure 4.4: Load bias control signals.

samples the bias with `load_bias[i]` set to 1, when dealing with a matrix with three rows, the `load_bias` signal is:

	t_{i+3}	t_{i+2}	t_{i+1}	t_i
r0	0	1	1	1
r1	0	1	1	0
r2	0	1	0	0

Table 4.1: Load bias signal.

where t_i correspond to the first clock cycle in which the signal is used. This pattern can be generalized for any number of rows setting to 1 the signal belonging to the row which follows the one previously set to 1. Taking as an example table 4.1, if row 0 is set to 1 at time t_i, row 1 is set to 1 at the following clock cycle and so on.

After the computation, the resulting matrix is moved from the systolic array to an external memory. As described before, data is shifted out in a skewed way, starting from the top-left PE. Next, the neighbors on the same column and on the same row send their results to the output port, therefore the top-left PE should be bypassed. This mechanism is implemented by exploiting an additional register to store the data coming from the neighbor PE on the same column. For this reason, each PE needs an enable signal for the bypass register and a selection signal for the multiplexer that drives the output port. Figure 4.5 shows the network exploited to download the resulting matrix.

Control logic elements like registers or multiplexers can be easily introduced where needed by specifying their instantiation as an high level algorithm written

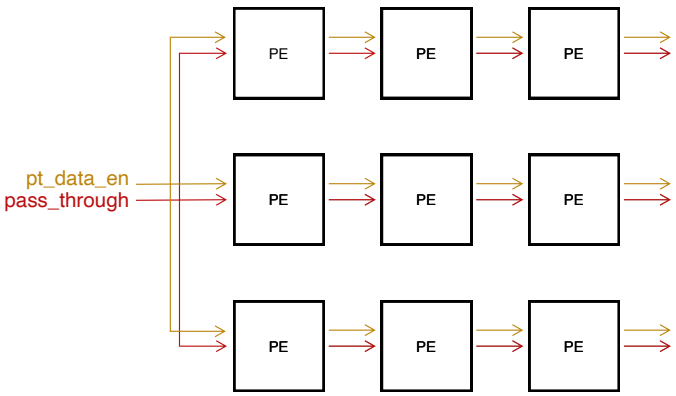

Figure 4.5: Pass through control signals.

in Python.

Table 4.2 shows how to set the enable signal to sample the forwarded data using, as an example, a systolic array with three rows. In the first clock cycle, every PE belonging to the first column except for the last one have to sample what comes from their bottom neighbor. In the second clock cycle, the same is done by all the PEs except for the last two ones, and so on. The last row enable signal can be ignored since the PEs do not have bottom neighbors.

	t_{i+2}	t_{i+1}	t_i
r0	0	1	1
r1	0	0	1
r2	0	0	0

Table 4.2: Pass through registers' enable signal.

In table 4.3, instead, the timing sequence of `pass_through` signal is reported. In the first clock cycle the top-left PE output comes from the internal accumulator register. On the contrary, in the following clock cycle PEs belonging to the first column have to forward data coming from the bottom neighbor. This is done by setting the selection signal to all 1s, where `pass_through[i]` at 1 means that a PE outputs the content of the bypass register. By doing so, the result computed by $PE[1][0]$, stored in $PE[0][0]$ bypass register, can be downloaded. Moreover, the data coming from $PE[2][0]$, previously stored in $PE[1][0]$ bypass register, will be sampled by $PE[0][0]$, so that it will be downloaded at the following clock cycle. In order to shift out the whole matrix, pass through signal has to be kept to all 1s, except for the last row, whose selection signal does not matter.

	t_{i+2}	t_{i+1}	t_i
r0	1	1	0
r1	1	1	0
r2	0	0	0

Table 4.3: Processing element output selection signal.

4.1.3 Control units

The presented framework provides not only the structure of a systolic array, but also the control units required to manage it. In this way, the generated module requires only an interface to deal with the memories and a few control signals to program the control units, which have to be generated by the accelerator decode unit where the systolic array is used.

Three phases characterize the computation of a full matrix-matrix multiplication:

1. Loading of the bias matrix.
2. Actual computation.
3. Download of the resulting matrix.

For each phase there is a different control unit in charge of handle it.

Load bias control unit

Load bias control unit is in charge of loading bias matrix into the systolic array. It is designed in order to handle control signals, as explained in the previous section, and to generate addresses used to retrieve the data from the memory. The sequence of `load_bias` signals is stored in a ROM whose entries are read sequentially to output the correct control signal at the right clock cycle. Before generating the addresses, the control unit requires to know the number of active rows of the systolic array, then it starts generating the proper control signals. At each clock cycle, it generates a set of addresses used to retrieve the matrix' elements in a skewed fashion and the corresponding read enable signals. The `out_reg_en` enable signal is driven accordingly to the proceeding of the computation.

Load data control unit

The actual computation is handled by the load data control unit. There are two instances of this control unit, one to manage the loading of matrix B, the other for matrix A. A load data CU is in charge of generating the addresses and the read enable signals to access the memories where the input data is stored. Before

starting the computation, it requires to know the reduction dimension of the input matrices. Together with the addresses, it drives also the input and output enable signals required by the systolic array.

Pass through control unit

The pass through control unit handles the signals used to download the resulting matrix, in particular `pt_data_en` and the `pass_through` selection signal. Since they depend only on the size of the systolic array, their pattern is fixed and so they are stored in a ROM, which returns, at each clock cycle, the right sequence of control signals.

Thanks to the metaprogramming capabilities of magma, the user does not require to manually adapt the control units to the array. In fact, when creating the systolic array, the smart template in charge of it extracts all the required parameters from the PE specification and then generates the control units accordingly, without any other user effort. If any changes to the control units are required, user can find their specification by looking at functions `LoadBiasCU`, `LoadDataCU` and `ShiftOutCU`.

4.2 Processing Element

The *processing element* is essentially a Multiply-and-Accumulate (MAC) unit in charge of performing the actual computation. It is designed in a modular way, with a clear division between the integration logic and the computational one, so that operations implemented can be customized as the user wishes. This section presents the template of the processing element using the process followed for the development as a walk-through to better understand the final meta-structure of the tensor PE.

4.2.1 Scalar Processing Element

The basic version of a processing element is a scalar PE, which is used to provide a detailed explanation of its structure which is the same independently on the operations implemented. A processing element is composed of two main blocks: one used for the integration in the systolic array and the other actually implementing operations. This choice derives from the need of designing a highly customizable PE. In this way, the performed operation can be changed by acting on a single module that implements it, instead of modifying the whole processing element. This may seem similar to a Verilog black box modeling, but the main difference is that the capabilities of a high level programming language like Python make the injection of new functionalities way easier. For example, the operations performed by a scalar

PE and a TPE are different, but a single change in the function describing the operation is enough to switch from one version to another. Moreover, if a new PE is designed, the integration in the systolic array is performed automatically, thanks the so called *high order functions* provided by magma. These are functions that takes as arguments circuit instances and return new circuit instances. An example is reported in listing 4.1. The code snippet shows how the integration of a PE is performed. It can be seen that the first call to `braid` function generates the rows of the array. The second one instead, takes the rows previously generated and connect them together to build the whole array. `braid` is a powerful function that allows the user to specify circuit instances that compose the final circuit and, through the definition of additional parameters, connect them in the desired way only specifying the name of the ports independently on the data type.

```
# generate the rows of the array
rows = []
for i in range(0, self.nr):
  rows.append(
    m.braid([PEScalar(self.in_bitwidth,
                      self.out_bitwidth,
                      self.guard_bits)()
             for i in range(0, self.nc)],
            joinargs=['in_b', 'bias', 'out_b', 'out_c'],
            foldargs={'in_a': 'out_a',
                      'load_bias_in': 'load_bias_out',
                      'pt_data_en_in': 'pt_data_en_out',
                      'pass_through_in': 'pass_through_out'},
            scanargs={'out_reg_en_in': 'out_reg_en_out',
                      'input_enable_in': 'input_enable_out'}))


# generate the whole array
sys_array = m.braid(rows,
                    joinargs=['in_a', 'out_a',
                              'load_bias_in',
                              'load_bias_out',
                              'pt_data_en_in',
                              'pt_data_en_out',
                              'pass_through_in',
                              'pass_through_out'],
                    foldargs={'in_b': 'out_b',
                              'bias': 'bias_out',
                              'out_c': 'pass_through_data'})
```

Listing 4.1: PE integration in a systolic array.

Figure 4.6 shows a schematic of the PE structure, where the elements required for the integration are grouped in the *Control logic*, while the ones implementing the actual operation are represented by the datapath.

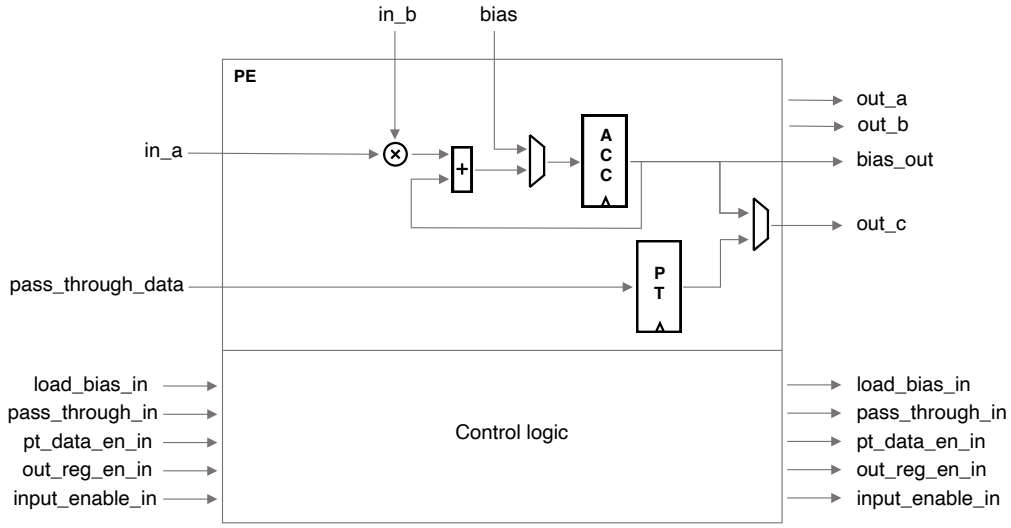

Figure 4.6: Processing element structure.

The control logic is mainly composed by registers used to handle the internal enable and selection signals and to forward the incoming systolic array's control signals to the neighbor PEs.

In the computation logic, instead, the actual operation is performed. For example, a scalar PE generally implements a MAC. The result saturates if it does not fit the bitwidth of the output register. The quantization strategy is implemented in a separate function so that it can be changed as the user desires. As it can be seen in listing 4.2, the saturation operation is described as an algorithm and the compiler is instructed to translate the if condition as a combinational circuit by the annotation over the definition of `if_cond` function. If the quantization strategy implemented is not suitable for the application, it can be completely modified by describing another algorithm and calling the corresponding function.

```
1  def saturate_output(in_bitwidth, out_bitwidth, guard_bits, io,
       tmp_sum, out_reg, idx_out_reg=-1):
2      sat_var_neg = m.sint(m.concat(
3          m.repeat(0, int(out_bitwidth) - 1), m.bits(True, 1)))
4      sat_var_pos = m.sint(m.concat(
5          m.repeat(1, int(out_bitwidth) - 1), m.bits(False, 1)))
6
7      out_c = m.Bits[out_bitwidth]()
8
9      @m.inline_combinational()
10     def if_cond():
11         if m.reduce(operator.and_,
12                     tmp_sum[int(2*in_bitwidth):
13                             int(out_bitwidth)]) \
```

```
                and (tmp_sum[int(out_bitwidth) - 1]):
            # no ovf: negative number
            out_c @= tmp_sum[:int(out_bitwidth)]
        elif not (m.reduce(operator.or_,
                           tmp_sum[int(2*in_bitwidth):
                                   int(out_bitwidth)])) \
                and not (tmp_sum[int(out_bitwidth) - 1]):
            # no ovf: positive number
            out_c @= tmp_sum[:int(out_bitwidth)]
        else:
            if tmp_sum[-1]:
                # negative overflow
                out_c @= sat_var_neg
            else:
                # positive overflow
                out_c @= sat_var_pos

    ...
    out_reg.I @= m.Mux(2,
                    m.SInt[out_bitwidth])()(out_c, io.bias,
                                            io.load_bias_in)
```

Listing 4.2: Naive convolution.

Finally, the accumulator register is used not only to store the output, but also to load an initial bias. In order to bypass the PE when shifting out the results, a pass through register is required, so that data coming from the neighbor PE can be stored and then forwarded.

A scalar PE is generated by the `PEScalar` function, that takes as arguments the following parameters:

- `in_bitwidth`: input data bitwidth
- `out_bitwidth`: output data bitwidth
- `guard_bits`: guard bits used in the accumulator

4.2.2 Tensorial Processing Element

The ultimate goal was to develop a tensor systolic array, which is composed by processing elements able to deal with tensors, not only with scalar elements. As previously mentioned in chapter 2, a matrix multiplication can be optimized first by reordering the loops describing it, secondly by applying tiling. While loop reordering corresponds to swapping different iterations of the loop, tiling refers to the process of performing a matrix multiplication on sub-matrices smaller than the original ones. The input matrices have three dimensions: N and M which correspond to the number of matrix A rows and matrix B columns, respectively,

and K that is the reduction dimension. The tiling on N is called $TileA$, the one on M is $TileC$, while the tiling on K is referred as $TileB$. Listing 4.3 presents the pseudo-code of the matrix multiplication exploiting loop reordering and tiling.

```
for n_t in 0..N/TileA:
  for  k_t in 0..K/TileB:
    for m_t in 0..M/TileC:
      for n in 0.. TileA:
        for k in 0.. TileB:
          for m in 0.. TileC:
            C[n_t*TileA+n ,m_t*TileC+j] +=
            A[n_t*TileA+n, k_t*TileB + k] *
            B[k_t*TileB + k, m_t*TileB+m]
```

Listing 4.3: Convolution after loop reordering and tiling.

A tensor processing element (TPE) processes $TileA$ rows from matrix A, $TileC$ columns from matrix B and $TileB$ elements from the same row of A and column of B, to perform a small matrix multiplication that generates a tile of the output matrix. A visual representation of the TPE processing is shown in figure 4.7. From now on $TileA$ is also referred as a_tiling, $TileB$ as b_tiling and $TileC$ as c_tiling.

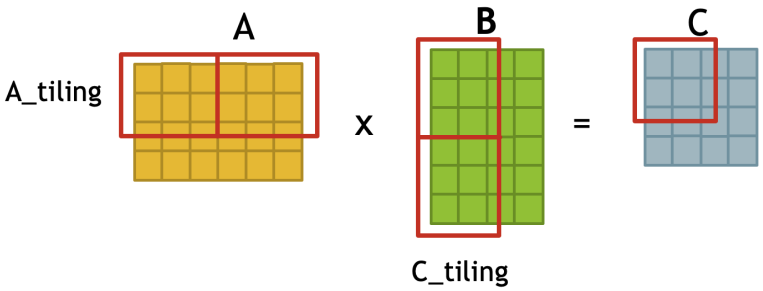

Figure 4.7: Tensors tiling.

Since each PE computes more than one element of the output matrix, the size of the systolic array required to perform the whole matrix multiplication is reduced. Moreover, since more than one element per row (column) is processed at a time, specifically $TileB$, the number of clock cycles required for the entire computation is smaller.

The PE is designed so that there are $a\_tiling * c\_tiling$ MACs. Each of them elaborates b_tiling elements coming from the matrices A and B. An example with $a\_tiling = c\_tiling = 2$ and $b\_tiling = 3$ is shown in figure 4.8

TPEs are generated using `TPE` function which, as `ScalarPE`, requires input and output data bitwidth and guard bits as arguments. Moreover, it needs to know

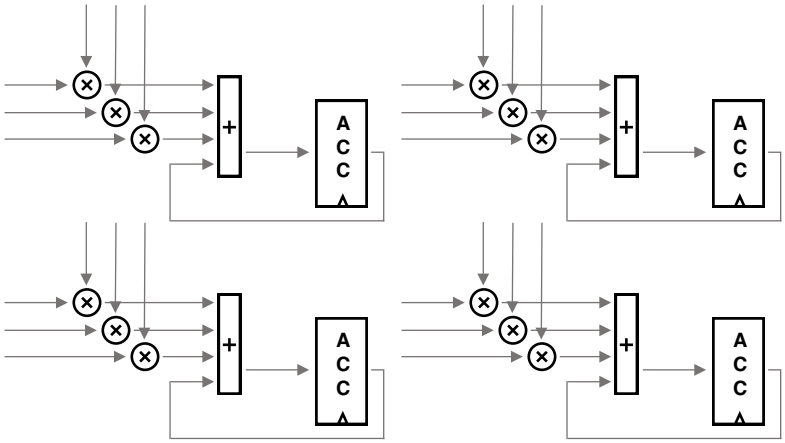

Figure 4.8: Tensor processing element.

`a_tiling`, `b_tiling` and `c_tiling`.

To better understand the computation of a tensor processing element, two corner cases can be analyzed. The first one considers that `b_tiling` is set to 1, so `a_tiling` rows and `c_tiling` columns of the input matrices are read element by element by each processing element. In the second one, instead, each PE process one row (column) considering `a_tiling` and `c_tiling` set to 1 and `b_tiling` elements form each row (column) at a time.

Figure 4.9 shows a visual representation of the first scenario. Supposing to take `a_tiling` rows from matrix A and `c_tiling` columns from matrix B, the corresponding PE will store $a\_tiling * c\_tiling$ elements of the resulting matrix.

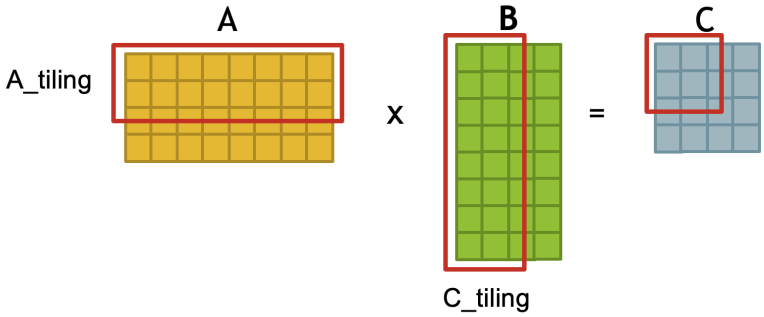

Figure 4.9: AxC tiling.

Taking many rows and columns at once means that the systolic array's size will be smaller than in the scalar version. However, no gain will be obtained in the number of clock cycles required for a computation, since only one element from

each row and column will be passed to a PE.

The second scenario, instead, is shown in figure 4.10, where a PE takes `b_tiling` elements from each row of A and from each column of B.

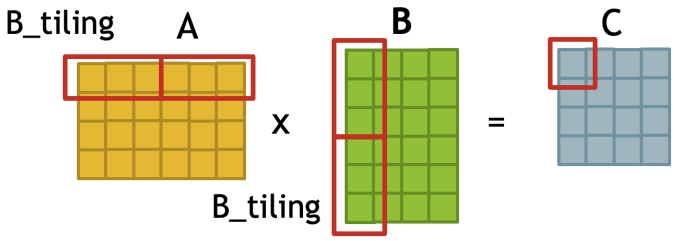

Figure 4.10: B tiling

In this case, the size of the systolic array is the same as the one of the output matrix, while the number of clock cycles required to compute the result is smaller.

As already mentioned, the differences between the PE versions reside only in the computation logic, where different kind of operations are performed. With the first approach, a PE elaborates many rows and columns at once, hence several MACs are needed. An example with $a\_tiling = 2$ and $c\_tiling = 2$ is shown in figure 4.11, where there are $a\_tiling * c\_tiling = 4$ MACs. Data coming from a matrix' row (column) is forwarded to every MAC present on the same row (column), so that each accumulator will store a different element of the output matrix. In the following example $A[i]$ is the element from the i-th row of matrix A, while $B[j]$ is the j-th element of matrix B. Considering the picture below, while the top-left MAC is computing $A[i] \cdot B[j]$, $A[i]$ is forwarded to the top-right accumulator which calculates $A[i] \cdot B[j+1]$. The bottom row does the same, resulting in $A[i+i] \cdot B[j]$ in the left-most accumulator and $A[i+1] \cdot B[j+1]$ in the other one.

The second approach, instead, does not require a PE with more than one MAC. On the contrary, there have to be many multipliers that compute the products between the input elements and an adder that sums them together with the previously accumulated result. An example of such a design is shown in figure 4.12.

Consider a row of matrix A as $a_k a_{k-1} ... a_2 a_1 a_0$ and a column of matrix B as $b_k b_{k-1} ... b_2 b_1 b_0$. At each clock cycle, the MAC takes $b\_tiling = 3$ elements from each row and column and performs element-wise multiplication. The resulting $a_i \cdot b_i$, $a_{i+1} \cdot b_{i+1}$ and $a_{i+2} \cdot b_{i+2}$ are summed together with the accumulator value.

Corner cases can be generated as well, exploiting functions `PE_AxC_Tiling` and `PE_B_Tiling`. They both requires input and output data bitwidth and guard bits as input parameters. In addition, `PE_AxC_Tiling` needs to know `a_tiling` and

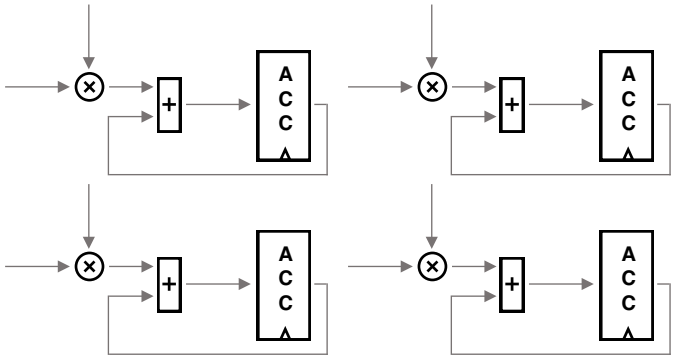

Figure 4.11: AxC tiling processing element.

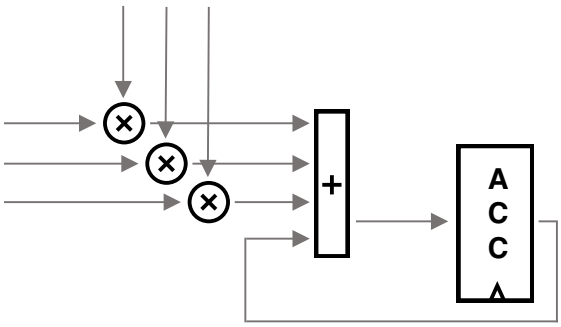

Figure 4.12: B tiling processing element.

`c_tiling` while `PE_B_Tiling` takes `b_tiling` as arguments.

The combination of the two corner cases produces the structure of a tensor processing element shown in figure 4.8.

4.3 Verification Suite

The presented framework provides not only a systolic array generator, but also a verification infrastructure that fits the customizable aspects of a systolic array. In particular, tests easily adapt to arrays of different sizes and also to different data bitwidth. The validation tests exploit *fault*, a domain-specific language capable of creating verification components and providing the same metaprogramming facilities found in a hardware construction language like magma. By using fault, a test is executed in two different stages. Once a designer has written a test program, the first stage constructs the test specification, while the second one invokes a runtime that executes the previously constructed specification. This concept in which the user constructs a program that constructs another program is called *staged metaprogramming*. Moreover, fault metaprogramming capabilities allow the

designer to write a test component as functions of the design under test, making the whole test automatically adapt to the component to be validated.

Thanks to the modularity of the generator, the PEs composing the systolic array can be modified to perform different kind of operations. In particular, when tests are required, most of them can be reused. In fact, the systolic array behaviour does not change even if different PEs are utilized. What has to be changed is the testbench that is used for the PE itself, since the operations performed may be different from the one already implemented in the framework.

As already explained in the previous sections, the framework generates a systolic array with its own control units. To validate the design, the systolic array is connected to several data buffer to simulate a real system with memories, even if their management is ideal considering that they have enough space to store the whole set of data. Each test - whose functions can be found in the `test_beh_xx` files, where `xx` stands for the array variants - is composed of a testbench which validates the correct behaviour when a computation is performed, and another testbench to check whether the bias is handled correctly or not. Since the systolic array is not included in a complete system, they both present a function used to manually load all the data required for the computation in the data buffers. Once data is stored in the buffers, the tests can be executed. The testbench executing a computation starts and programs the control units managing the data buffer, then it waits for the computation to end. When validating the whole computation, there is no bias loading phase. Once control units have finished, the testbench starts the CU in charge of shifting out the resulting data and checks the elements that come out of the array. Similarly, the second testbench starts the control unit in charge of loading bias in the systolic array. Once data is loaded, it simply shifts it out and check the correctness of the values.

As regards the PEs, the tests just check each functionality implemented. When extending the framework with a new processing element's design, these tests may not hold anymore, hence they have to be adapted to the new implementation. An example of this kind will be presented in section 4.5.

As already mentioned, the testbenches were developed exploiting *fault*. To construct the fault test component, a testbench uses the `Tester` object with the magma circuit to be tested as argument. Then, this object is used to record a series of actions to be performed exploiting Python constructs and data types. In this way, data can be manipulated as desired within the test procedure, leaving the burden of actually translating it into a hardware testbench to the compiler. Fault not only simulates the hardware architecture returning the outcome of the simulation, but it also allow the user to generate the actual SystemVerilog testbench file. In this way, taking as an example the aforementioned tests, a designer can create a golden model using the matrix multiplication implemented by *numpy* and check the architecture against the ooutput of the numpy function. When the compiler

has to translate this kind of procedure in SystemVerilog, it converts each Python data structure in simple raw data assigned to the signals. For example, a loop is unrolled and the resulting testbench is composed by all the iterations in sequence where the signals are assigned with what was defined in the original loop. This is a powerful ability that allow the designer to simply describe a validation test exploiting a high level program language like Python and automatically generating the corresponding SystemVerilog testbench, which can be used as a post-synthesis test like what is detailed in chapter 5.

Each of the previously presented tests validating the behaviour of the whole systolic array can be reused by simply changing the generation parameters of the array itself. Eventually, if a specific test has to be executed, one has also to change data loaded into the buffers since they are randomly generated.

4.4 Framework advantages

Given the framework structure, there are multiple exploitable advantages, related not only to how the systolic array generator was implemented, but also to the provided test suite and to the potential that a hardware construction language such as *Magma* can offer. The most important ones are summarized below:

- Systolic arrays are highly customizable.
- The framework can be extended with custom PEs.
- The modular structure provides an easy way to test the design.
- The generated Verilog is ready to be synthesized.
- The generated Verilog is readable.

As already presented, systolic arrays can be generated with different sizes and data bitwidth without much effort. It is sufficient to change the parameters at creation time. PEs that compose the array can be designed by the user, since the whole structure does not change depending on the processing element's implementation. What is important is that the interface is compliant to the one described in the framework. As regards testability, given the modularity of the whole design, the provided test suite can be completely reused without many changes. At most, the PE tests validating the processing element functionalities have to be developed if they are customized. The ones testing the whole systolic array are generated automatically. Moreover, there are the advantages that comes with Magma. Specifically, it generates a highly readable Verilog code that is also directly synthesizable without any change.

4.5 Use case: Sparse Systolic Tensor Array

So far, the considered systolic arrays do not take into account DNN weights sparsity. In particular, this property can be exploited to speedup the whole computation. As already explained in section 3.5.2, a traditional tensor systolic array can be modified in order to take advantage of the weight sparsity.

This section presents the design of a sparse systolic tensor array, where the weight sparsity is exploited. This is meant to be a demonstration of how the framework can be extended to implement a variation of the dense tensor systolic array.

4.5.1 Extension of the Framework

Design Effort

Only few changes are required to design a *Sparse Systolic Tensor Array* (S2TA), in fact its structure is basically the same as the one provided with the framework. Specifically, together with the data signals already presented, there has to be one more input, the *bitmap*. This is a signal used to select which elements from matrix A are multiplied to non-zero weights. As bitmap moves like the weights matrix, its specification can be included in `in_b` network, as shown in figure 4.1. In this way, the systolic array smart template does not require any changes. On the contrary, it is already able to automatically generate the whole structure supporting sparsity.

The major change has to be applied to the PE's structure. Indeed, the tensor processing element has to be extended in order to support data sparsity. This property comes with the addition of multiplexers driven by the bitmap, that choose the right elements belonging to matrix A.

As before, each PE is composed of $a\_tiling \cdot c\_tiling$ MACs, which now elaborate `sparsity` elements at a time, instead of `b_tiling`. The sparsity of a design is given by $1 - \frac{sparsity}{b\_tiling}$, since `sparsity` defines how many non-zero elements are present. As regards the inputs coming from matrix A, `b_tiling` of them are loaded at a time. Since $sparsity < b\_tiling$, they are multiplexed so that each multiplier takes the element corresponding to a non-zero weight, based on the selection signal derived from the bitmap. An example of the structure is shown in figure 4.13, where there are $sparsity = 2$ multipliers.

In order to drive the multiplexers, a priority encoder is required. Bitmap is supposed to be a vector where a bit at index i set to 1 means that the i-th element of the input A is multiplied to a non-zero weight. Given that there are `sparsity` bits different from zero, the selection signals are encoded one at a time. The original bitmap is processed by the priority encoder, which is able to identify the index of the first digit set to 1. The index just found is used to drive one of the multiplexers. The bitmap is then masked so that the resulting signal does not have that digit

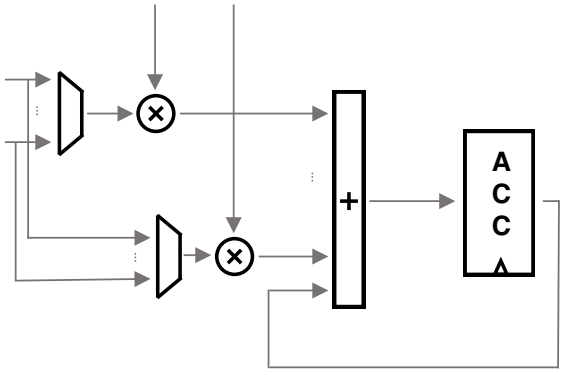

Figure 4.13: Structure of a sparse MAC.

set to 1 anymore. This identifies the following index to be used as selection signal. The aforementioned procedure is applied until all the non-zero bits of the original bitmap are found.

Besides these changes, nothing else belonging to the tensor PE was modified.

Verification Effort

As mentioned in section 4.3, if the processing element is changed, the testbench provided with the framework may not be valid anymore. In the case of the PE that composes the S2TA, most of it can be reused, hence, with slightly changes applied mainly because of the handling of bitmap signal, it was easily adapted. Because of this additional signal, the systolic array's test has changed as well. The main adjustment is that the testbench drives also the bitmap signals, as it happens for the PE test.

Functions overview

Before generating the whole array, a processing element has to be specified. Function `SparseTPE` generates a tensor processing element which support sparsity. `SparseTPE` takes the following parameters as arguments:

- `in_bitwidth`: input data bitwidth
- `out_bitwidth`: output data bitwidth
- `guard_bits`: number of accumulator guard bits
- `a_tiling`: tiling on matrix A rows
- `b_tiling`: tiling on matrix reduction dimension

- `c_tiling`: tiling on matrix B columns
- `sparsity`: number of non-zero elements

A Sparse Systolic Tensor Array is then generated by `SysArrayS2TA`, which takes as arguments the number of rows and columns and the PE generator produced by `SparseTPE`.

Validation tests, instead, are contained in `test_beh_s2ta` file.

Chapter 5

Experimental Results

The proposed framework is used to perform a design space exploration, targeting the *Zynq Ultrascale+ MPSoC ZCU104 Evaluation Board*, to assess the performances of different designs in terms of resources occupied, power consumed and processing time.

5.1 Experimental Setup

Figure 5.1 shows the flow followed to design, simulate and synthesise the architectures, in particular, it is composed of these steps:

- Python-based design.
- Python-based tests.
- RTL design generation.
- Logic synthesis and implementation.
- Post-implementation simulation.
- Power and area estimation.

Hardware Generation and validation

The presented framework provides functions to generate and validate a hardware design. There is a function `SynXX` - where `XX` can be Scalar, AC, B, TSA or S2TA depending on the desired array - which generates the hardware of a systolic array together with its control units with the specified parameters. It requires generators of the desired systolic array and of a bias control unit, together with the address

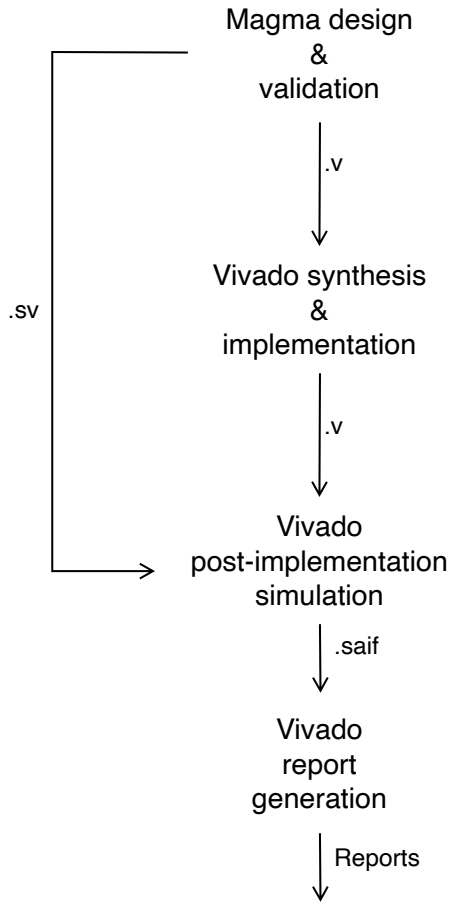

Figure 5.1: Design flow.

bitwidth which control units have to use to encode memory addresses. The user can also specify a name for the architecture if the default one is not suitable for the application. The systolic array generator, passed as parameter to `SynXX`, requires the number of rows, columns and the PE generator. A processing element, as a systolic array, can be of different types. These are:

- Scalar
- Vectorial with tiling on the rows/columns, called ACTiling
- Vectorial with tiling on reduction dimension of input matrices, called BTiling
- Tensorial
- Tensorial supporting sparsity

Each variant has its own function which requires different parameters. The ones in common are the following:

- `in_bitwidth`: bitwidth of the input data
- `out_bitwidth`: bitwidth of the output data
- `guard_bits`: guard bits used for the accumulator

Depending on the processing element required, the following parameters have to be passed as well:

- `a_tiling`: tiling on the rows
- `b_tiling`: tiling on the reduction dimension
- `c_tiling`: tiling on the columns

In case the processing element supports data sparsity, `sparsity` has to be specified, too.

The bias control unit generator, instead, requires the number of rows and columns of the systolic array, `a_tiling` and `c_tiling`. If the desired array is the scalar version they can be simply set to 1. It also requires the address bitwidth to encode the addresses and the size of the register used to store the number of elements on the reduction dimension.

Once everything is specified, `SynXX` is used to generate the hardware design ready to be validated. The presented framework provides several tests for the generated architectures which validates the behaviour of a systolic array together with the control units as if they were in a system including also memories. In fact, each design is first connected to data buffers, through the functions `DatapathXX`, and then validated. For further details on the tests structure refer to section 4.3.

Logic synthesis and implementation

After the validation phase, the architecture is almost ready to be synthesised. In this case, synthesis and place and route are followed by a post-implementation simulation to better estimate the switching activity, therefore, a SystemVerilog testbench is required. Each variant of the systolic array has its own script, `tb_generator_xx`. They generate the RTL netlist of a set of architectures, the ones used to perform the design space exploration, together with the SystemVerilog testbenches for the simulation.

The architectures may then be synthesized. A synthesizer generally optimizes, in terms of resource occupation and power consumed, the mapping of a netlist

to the FPGA, not necessarily exploiting all the possible resources on a board. For example, the *Zynq Ultrascale+ MPSoC ZCU104 Evaluation Board* provides not only LUTs but also DSPs, which are efficient cells particularly useful when performing operations like MAC. They allow complex operations to reach good performances without consuming much power. However, using this kind of cells leads to waste LUTs for connections, hence, the resulting mapping may not be the best one from the point of view of the synthesizer. In fact, in this case, the synthesised architectures are hardly ever mapped on DSPs.
Since the aim of the presented design space exploration is to assess the power consumed and the resources occupied while providing the best performance possible, the synthesizer has to be instructed to exploit DSPs. Then, it is sufficient to annotate the Verilog source code of each architecture with `(* use_dsp48 = "yes" *)` in correspondence of the modules which have to be mapped on DSPs.

RTL netlists are now ready to be synthesised. Synthesis and implementation are done by means of TCL scripts to automate the process. Each script requires the source code to be organized is a specific way: Verilog netlists are grouped in the `src` folder, testbenches are put in the `tb` folder and constraints are contained in the `constr` folder.

The constraints .xdc files have to be created by Vivado. To do that, open a new project and import a RTL netlist of the desired architecture. Run a synthesis. Once it is done, open the *Constraints Wizard*. A new window appears, select the target file where all the constraints will be written and confirm the choice. Open the wizard again so that a new window is opened. Going through all the windows of this tool allow the user to set the desired constraints like clock frequency, input and output delays. In the presented case, the following constraints are considered:

- Clock period = 22 ns
- Maximum input delay = 12 ns
- Minimum input delay = 10 ns
- Setup time = 2 ns
- Hold time = 0 ns

Clock period changes for the sparse architectures since they are not as efficient as the others. In particular it is set to 24 ns. Input delays changes accordingly to be ± 1 ns the edge of the clock.

Once all the architectures have their own constraints, the provided scripts can be used to perform a synthesis. Each of them follows the same structure that first creates a new Vivado project, setting as target board the Zynq Ultrascale+ MPSoC ZCU104 Evaluation Board. Then the script loads the Verilog source code of an

architecture together with the corresponding constraints. Before synthesising the design, the following properties are set:

- `STEPS.SYNTH_DESIGN.ARGS.MORE OPTIONS -value {-mode out_of_context}`
- `{xsim.simulate.runtime} -value {15000ns}`
- `{xsim.simulate.saif} -value {sw_act.saif}`

The first one is a synthesis property which declare how to synthesise the design, specifically `out_of_context`. Since the systolic array is not meant to be used alone in a FPGA but it should be contained into a larger system interfacing it, the number of I/O required is larger than the ones provided by the board. To prevent the synthesizer from mapping systolic array inputs and outputs on I/O ports, `out_of_context` mode is specified. The second and the third properties are related to simulation. `xsim.simulate.runtime` sets the runtime of the simulation to a value larger than the actual simulation time, so that the simulations certainly ends. `xsim.simulate.saif`, instead, specifies the name of the switching activity file.

After specifying the aforementioned properties, synthesis and place and route are run.
The following step is the post-implementation simulation, so, first the tesbench is imported, then the simulation is started.
Once it finishes, the script prints timing, power and occupation reports. Before generating power report, the switching activity file is read.

5.2 Design Space Exploration

The proposed framework is used to perform a design space exploration targeting the *Zynq Ultrascale+MPSoC ZCU104 Evaluation Board*. The generated hardware architectures are utilized to assess the effects of the following knobs on energy consumption, resources occupation and latency:

- **Array size:** 8x8, 16x16, 32x32
- **Data bitwidth:** 4-bit in and 16-bit out, 6-bit in and 20-bit out, 8-bit in and 24-bit out, 8-bit in and 32-bit out
- **PE structure:** scalar PE or tensor PE, including corner cases
- **Sparsity support:** yes or no

The first experiment assess how data bitwidth affects the energy consumption and the resources occupied. Considering that performing operations on more bits consumes more power and resources, the total energy and resources utilization is expected to increase with larger data bitwidth. Next, increasing a systolic array size should lead to more power consumed and resources occupied. The second experiment assesses how these two metrics are affected by the actual size of the array. Moreover, an interesting scenario takes in consideration all the possible configurations of a systolic array which perform the same number of MAC operations per cycle. The third experiment shows how the PE structure affect the energy consumed and the resources utilized by these set of systolic arrays. Finally, the support of sparsity is considered showing how energy and resources depend on the sparsity ratio. Since the generated architectures reach different maximum clock frequencies, they are compared to show the trade-offs depending on different configurations. Lastly, the overall performances are analyzed when executing different workloads.

The architectures used for the design space exploration are always considered with 6-bit inputs and 20-bit outputs and 8-bit inputs and 24-bit outputs, except for the experiment that assess the data bitwidth effects. These are commonly used configurations for inference. Moreover, guard bits used in PE saturation logic are always set to 8, which is an acceptable number of bits to keep the numerical stability of a neural network. Finally, since static power consumed by the architecture is platform dependent and it is always the same, it is not considered in the energy consumption metric. This fact was experimentally verified.

5.2.1 Resource occupation and power estimation with different data sizes

The first assessment regards how the data bitwidth affect the power consumption and the resource occupation of a systolic array, specifically a 16x16 scalar array. For these measurements, the bitwidth considered are the following:

- 4-bit inputs, 16-bit output
- 6-bit inputs, 20-bit output
- 8-bit inputs, 24-bit output
- 8-bit inputs, 32-bit output

Figure 5.2 shows the differences between various configurations. The left y-axis is used for the dynamic power consumption, while the right one keeps track of the resource number, expressed in thousands of units. As it can be seen, the

resource count grows mainly because of the registers required to store the data. The number of DSPs is constant except for the first configuration. DSPs are used to implement the PE multiplier, and, since the array size is 16x16, there are 256 DSPs. Vivado does not use DSPs to map the multipliers when the input bitwidth is 4 bits. The reason may be that they are too small to take effective advantage in using such resources. Multipliers, instead are mapped on a combination of CARRY8 cells and LUTs. CARRY8 is a primitive present on the ZCU104 board which is used in conjunction to LUTs to implement fast multipliers and adders, hence this increase the resources count. However, the power consumed by this larger number of resources is easily surpassed by the energy required by DSPs used in the other configurations. Therefore, the overall power consumption grows linearly with the data bitwidth.

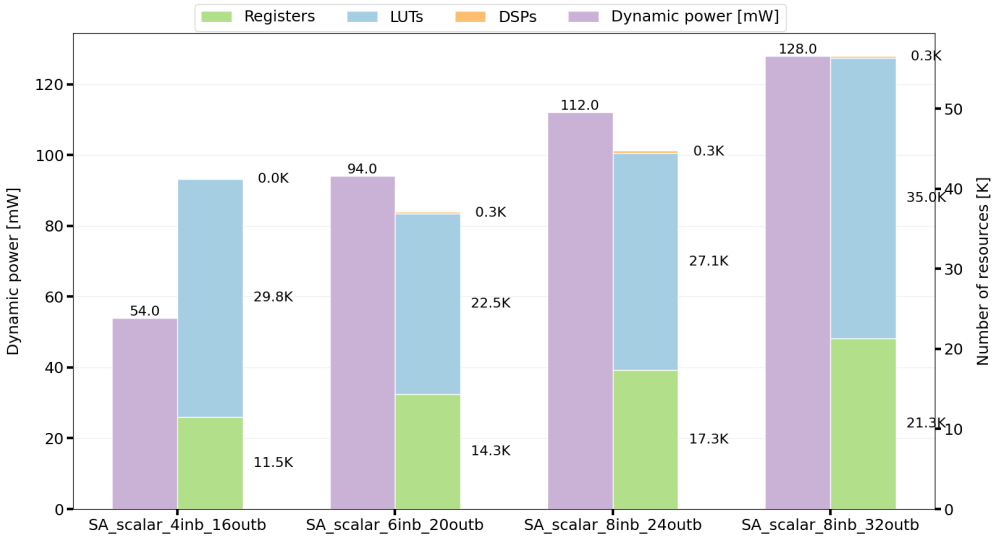

Figure 5.2: Resource utilization vs. dynamic power consumed of a 16x16 scalar systolic array with data of different bitwidth.

5.2.2 Scalar systolic arrays of different sizes

Figures 5.3 and 5.4 show how the resources occupation and the dynamic power increase with the size of a scalar systolic array. The former presents the results of an array whose inputs and outputs sizes are 6 bits and 20 bits, respectively. The latter considers 8-bit inputs and 24-bit outputs. The systolic arrays increasing size leads to an increment of the resources required to map the architecture on the

FPGA. Consequently, the dynamic power consumed increases as well. Specifically, it increases by 3-4 times each step, almost as much as the size of the array. The results follow the same trend independently on the data bitwidth, in fact the differences reside only in the absolute values obtained.

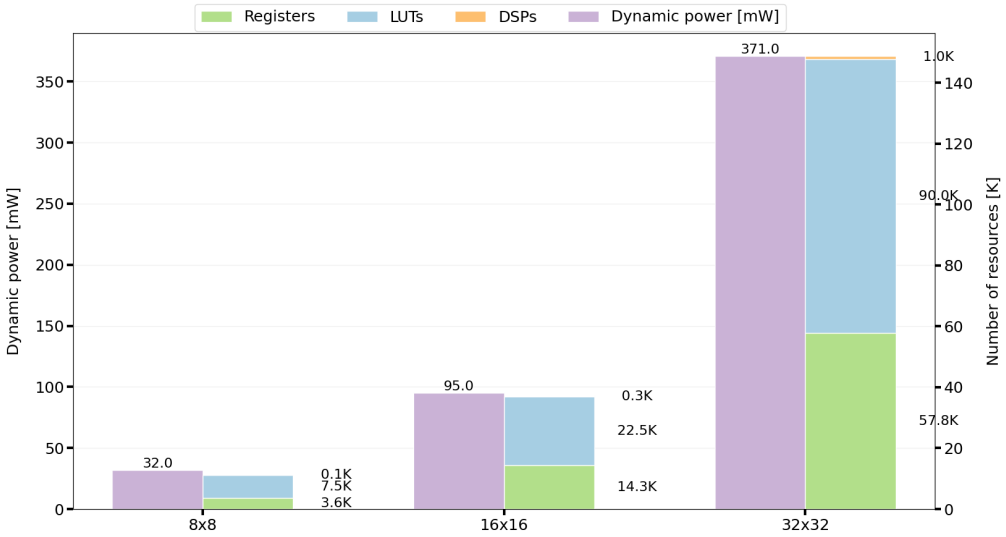

Figure 5.3: Resource utilization vs dynamic power consumed when scalar arrays of different sizes are considered. Inputs data size is 6 bits, outputs one is 20 bits.

A more detailed analysis of the dynamic power consumption is shown in figure 5.5. As it can be seen, the systolic array power prevails over the control units one, as expected. This is due to the difference in complexity, since the control unit is a quite simple FSM, while the systolic array is composed by many processing elements each implementing a MAC. The results show a trend for which the size of the array increases the power consumed, as expected. This happens for both the data bitwidth considered. The 8x8 ring may seem strange since the shown percentage is 0.0%, but this is due to the precision of the power reports that is not enough to estimate the power consumption. The control unit contribution on the power consumed is too small to be reported.

As a matter of comparison, table 5.1 shows the absolute values of the power consumed by the 8-bit inputs and 24-bit outputs systolic arrays. It can be seen that, as the number of rows increases by 2 times for each configuration (that is, 8, 16 and 32). As the two CUs depend on this parameter, the control units power consumption grow 4 times, since its size depends on the number of rows of the systolic array.

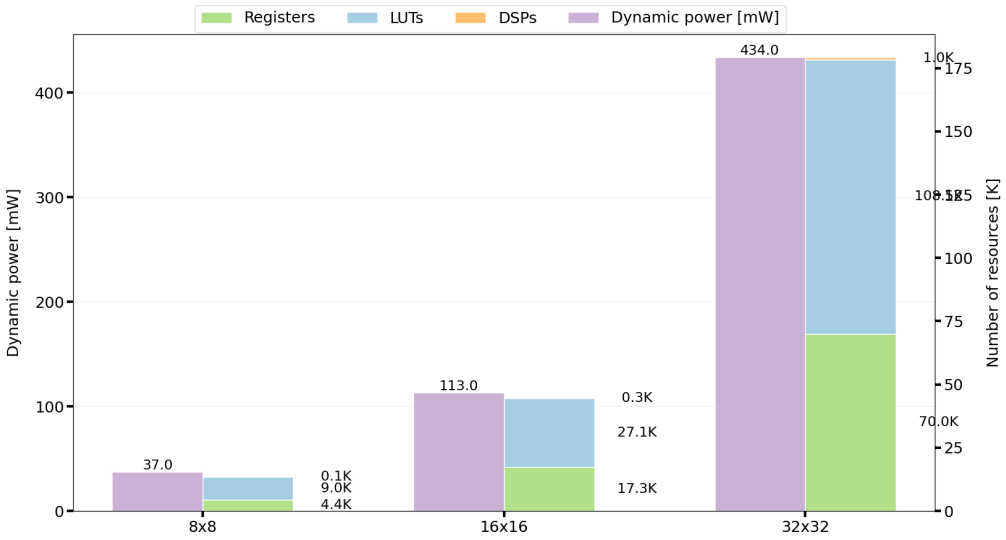

Figure 5.4: Resource utilization vs dynamic power consumed when scalar arrays of different sizes are considered. Inputs data size is 8 bits, outputs one is 24 bits.

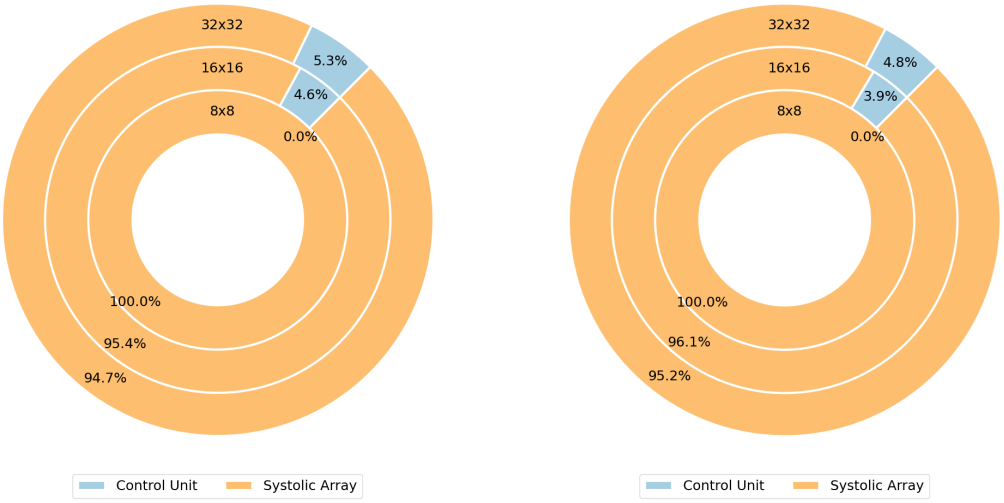

Figure 5.5: Power breakdown of a systolic array with 6-bit inputs and 20-bit outputs (left) and 8-bit inputs and 24-bit outputs (right).

Array size	Array power [mW]	Control unit power [mW]
8x8	0.037	~ 0.000
16x16	0.109	0.004
32x32	0.419	0.015

Table 5.1: Power breakdown 8-bit inputs 24-bit outputs systolic arrays.

5.2.3 Fixed number of MACs per cycle

Figure 5.6 shows the results in terms of power consumption and resource utilization of different architectures with 8-bit inputs and 24-bit outputs performing the same number of MAC operations per clock cycle, specifically 1024. The plot reports, for each design, the power consumption and the resources occupation as a function of the scalar systolic array which is taken as a baseline. Scalar systolic array and tensor systolic arrays are compared to show different effects caused by more complex, but also more powerful structures like TSA with respect to a classic systolic array. TSA corner cases are considered, too.

TSAs performing $A \times C$ MAC per cycle with a *TileB* equal to 1 are the architectures returning the worst results. An increasing of the tiling size lead to a larger data reuse inside a processing element, resulting in smaller systolic arrays with a smaller total number of intermediate registers between PEs. Even if the registers utilized decrease, the LUTs present consume much more than a scalar systolic array. The expected behaviour would show a decreasing in the number of registers and in the number of LUTs occupied as the tiling increases, that actually happens except for the 4Ax4C case. This last configuration, after synthesis and implementation, shows an anomalous increase of the LUTs required as registers with no apparent reason. Moreover, the differences in power with respect to the baseline are justified by the increasing switching activity detected in the post-implementation simulation. As it can be seen, designs which perform a B-way dot product lead to the best results. In particular, they present a lower resource utilization than the baseline. One of the reason is the fewer registers required to perform the accumulation, indeed B multiplications are accumulated on a single register, instead of only 2 as a normal MAC. Another reason is that a scalar systolic array of size 32x32 is four times larger than a 16x16 TSA. The resulting TSA LUTs count is smaller, despite the higher number required to perform a more complex accumulation operation in a TPE. The best scenario of resources occupation and power consumption is reached by STA_4Ax16Bx4C_2x2 and STA_2Ax16Bx2C_4x4, which consume as much power as SA_4B_16x16 requiring fewer resources. Moreover, by comparing SA_4B_16x16 to its counterparts, where tiling on rows and columns is introduced, it is possible to observe a lower resources utilization but a higher power consumption.

From these considerations it can be deduced that operating on *TileB* leads to better results in terms of power consumption with respect to *TileA* and *TileC*. On the contrary, increasing *TileA* and *TileC* helps saving resources at the cost of power consumed by the design. This reasoning holds also if the architectures considered take 6-bit inputs and return 20-bit outputs.

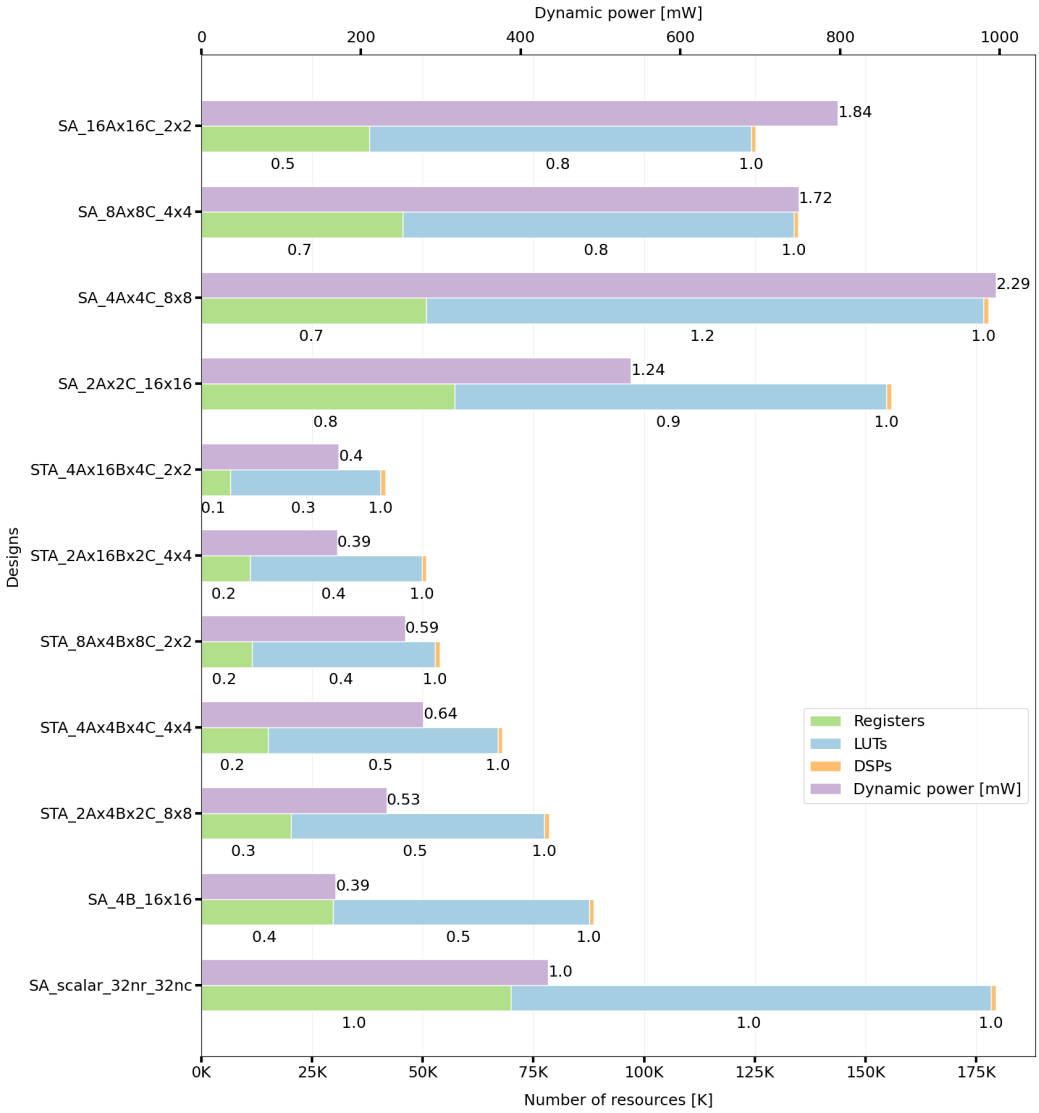

Figure 5.6: Power vs area with a fixed number of MAC operations per cycle. The considered designs have 8-bit inputs and 24-bit outputs.

5.2.4 S2TA: different data sparsity values

The Sparse Systolic Tensor Array is characterized as well. In particular, dynamic power consumption and resources occupation are assessed. As before, data size does not have a high impact on the trends of final outcomes, so results presentation consider designs with 8-bit inputs and 24-bit outputs only. Figure 5.7 shows the results obtained with a S2TA of size 2x8x2_4x4. As expected, the power consumed and the resources utilized decrease when the sparsity increases. Data sparsity directly affects the LUTs and DSPs count since less elements result in less multipliers, mapped on the DSPs, and less LUTs to implement the adders.

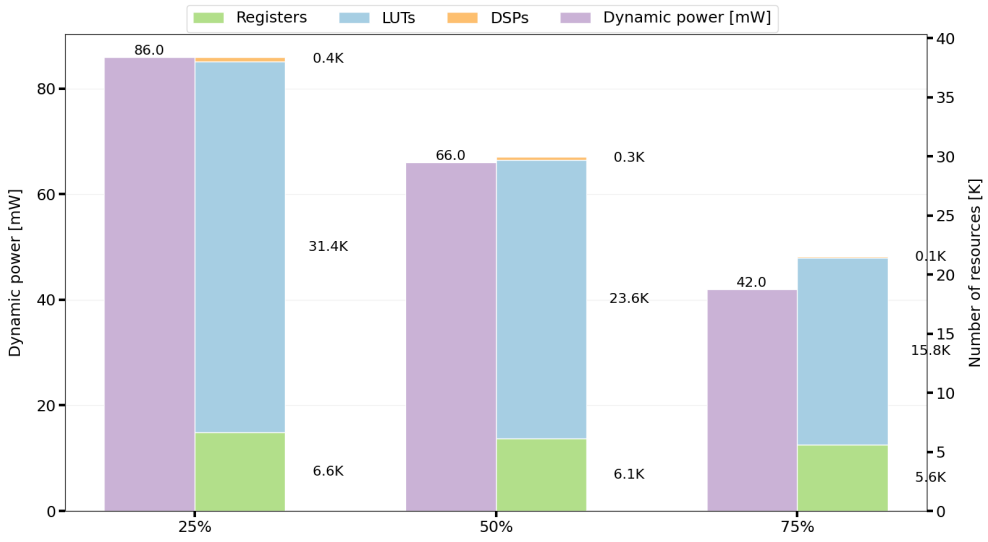

Figure 5.7: Dynamic power consumed and resources utilized by a Sparse Systolic Tensor Array of size 2x8x2_4x4, with 8-bit inputs and 24-bit output.

However, as table 5.2 reports, the maximum frequency obtained is lower than the one reachable by the same architecture without the support for sparsity. In particular, depending on the sparsity ratio, architectures supporting sparsity reach half the frequency of their dense counterpart.

One source of high inefficiency is the priority encoder used to decode the bitmap and set the multiplexer selection signals. In fact, it is implemented as a chain of multiplexers. In particular, with a sparsity of 50%, it increases the LUTs count of about 60% while worsening the latency of about 80%. This worsening depends on sparsity ratio, since the number of non-zero elements directly affects the complexity of the priority encoder. Besides this fact, sparsity helps saving resources and power if compared to a dense systolic tensor array of same size.

Architecture	Registers	LUTs	DSPs	Dynamic power	$Tclk_{min}$
2x8x2_4x4 dense	7.0K	22.0K	0.5K	254.0 mW	12 ns
2x8x2_4x4 25%	6.6K	31.0K	0.4K	86.0 mW	24 ns
2x8x2_4x4 50%	6K	24K	0.2K	103.0 mW	18 ns
2x8x2_4x4 75%	5.5K	16K	0.1K	72.0 mW	14 ns

Table 5.2: Comparison between a systolic tensor arrays with and without sparsity support.

5.3 Max Frequency Analysis

The aforementioned designs, which perform the same number of MAC per clock cycle, present various PE complexities, resulting in different maximum reachable frequencies. Table 5.3 reports, for each considered design, the minimum clock period, the power consumed and the resources utilized at its maximum frequency. DSPs are not shown because their number does not change with respect to the one reported in the previous analysis. Since differences between architectures with 8-bit inputs and 24-bit outputs and the ones with 6-bit inputs and 20-bit outputs do not influence the trends, the analysis is performed only on the first ones.

The best design in terms of latency is the scalar systolic array, thanks to a simple structure of the processing elements which allow the clock period to reach 8 ns. However, a 32x32 scalar systolic array is the largest design that utilizes more resources than all the others. Results confirm that, admitting a larger clock period than 8 ns, systolic tensor arrays characterized by *TileA* and *TileB* only save resources but consume a significant higher amount of power with respect to a scalar SA. The best results in terms of power consumption are obtained by systolic tensor arrays implementing a B-way dot product. However, *TileB* affects the adder trees present in the TPEs increasing the overall latency. STA_2x16x2_4x4 and STA_4x16x4_2x2 are examples. They are about 2.5 times slower than a scalar SA but they obtain the best results in terms of power consumption and resource occupation. A good trade-off between the three metrics may be reached by a TSAs presenting tiling on rows and columns and a not too large *TileB* which affect the latency. Architectures like STA_4x4x4_4x4 exploits the advantages of *TileA* and *TileC* in terms of resource occupation, and the power reduction that comes with *TileB* balances the power increasing introduced by rows and columns tiling. Unfortunately, these gains come at the cost of latency.

One last dimension to consider is the workload executed by a systolic array. Depending on the structure of a SA, the clock cycles required for the execution differs. These variations combined with the array characterization in terms of

latency, power consumed and resources occupied affect the total power and resources required by a whole computation on a specific systolic array. Next section analyzes this consideration.

Architecture	$Tclk_{min}$	F_{max}	Power	LUTs	Registers
STA 16x1x16_2x2	16 ns	62.5 MHz	1,100 mW	86 K	38 K
STA 8x1x8_4x4	12 ns	83.3 MHz	1,300 mW	88 K	45 K
STA 4x1x4_8x8	10 ns	100 MHz	2,000 mW	126 K	51 K
STA 2x1x2_16x16	10 ns	100 MHz	1,170 mW	97 K	57 K
STA 4x16x4_2x2	22 ns	45.45 MHz	172 mW	**34 K**	**7 K**
STA 2x16x2_4x4	22 ns	45.45 MHz	**170 mW**	38 K	11 K
STA 8x4x8_2x2	12 ns	83.3 MHz	457 mW	41 K	11 K
STA 4x4x4_4x4	11 ns	90.9 MHz	517 mW	52 K	15 K
STA 2x4x2_8x8	10 ns	100 MHz	498 mW	57 K	20K
STA 1x4x1_16x16	10 ns	100 MHz	365 mW	58 K	30 K
Scalar 32x32	**8 ns**	**125 MHz**	470 mW	108 K	70 K

Table 5.3: Maximum frequencies reached by designs performing the same number of MAC per cycle, specifically 1024.

5.4 Performance Analysis

5.4.1 Analytical Model

In a systolic array, input data is fed from the left and top edges. Once the computation in done, the resulting matrix is downloaded from the top-edge. Since the download behaviour is not the same as the one presented in chapter 3, the resulting analytical model is different, too.

As previously described in chapter 4, the download phase starts from the top-left PE, then it proceeds with the right and bottom neighbors and so on. Theoretically, once the top-left PE has finished its computation, data is ready to be downloaded. This mechanism would work since, at each following clock cycle, the neighbors finish the computation as well. Therefore, the results would be ready to be downloaded. The time of computation would correspond to the time required by a PE to process all the data, which is $\lceil \frac{K}{B} \rceil$, where K is the reduction dimension of the input matrices and B is the tiling on that dimension. Practically, given the design of the control units, before downloading the results, the CUs have to finish loading the data. The time required by a CU to end the computation is shown in equation 5.1, where *n_pe* is the number of processing element on the edge controlled by the CU. It

corresponds to *n_rows_array* for the control unit loading the input matrix and *n_cols_array* for the one loading the filters.

$$\tau(cu) = n\_pe + \left\lceil \frac{K}{B} \right\rceil - 1 \tag{5.1}$$

If the systolic array is not a square, the computation ends when the larger control unit finishes the processing. Moreover, in order to start the CUs, two more clock cycles are required. The first one to actually start it, the second one to program it. Considering *nr_a* and *nc_a* the number of rows and columns of the array, respectively, the resulting computation time is:

$$\tau(computation) = \max(nc\_a, nr\_a) + \left\lceil \frac{K}{B} \right\rceil + 1$$

Once the computation is finished, the download phase can be started. The time required to download the whole matrix is $nc\_a + nc\_a - 1$ plus the time needed to start the control unit which takes 1 clock cycle. The resulting download time is:

$$\tau(download) = nc\_a + nc\_a$$

As previously detailed in chapter 3, when processing large input matrices, many iterations on the same systolic array are required. The effective number now depends also on the tiling since the analysis is done considering systolic tensor arrays. Therefore, the number of iterations is $\lceil \frac{N}{nr\_a \cdot A} \rceil$ on the rows and $\lceil \frac{M}{nc\_a \cdot C} \rceil$ on the columns, where N and M are the rows and columns of the resulting matrix. A and B, instead, are the tiling parameters on the row and on the columns, respectively. Moreover, since the CUs can be programmed once before starting the whole computation and not at each iteration, $\tau(computation)$ can be reduces by 1, counting it only once at the beginning. The resulting clock cycles required perform a whole computation is:

$$\tau = \left(\max(nc\_a, nr\_a) + \left\lceil \frac{K}{B} \right\rceil + nc\_a + nr\_a \right) \left\lceil \frac{N}{nr\_a \cdot A} \right\rceil \left\lceil \frac{M}{nc\_a \cdot C} \right\rceil + 1$$

5.4.2 Experimental Results

Results obtained in previous experiments showed the effects of different knobs on power consumption, resources occupation and latency. The last assessment regards the effects of the executed workload on the total execution time.

Workloads are selected layers from two neural networks: EfficientNet [20] and ResNet [21]. The first two belongs to EfficientNet while the others to ResNet50 Table 5.4 presents the size of the matrices processed by each convolutional layer and the corresponding mapping on GEMM. The reported sizes are:

- H: output matrix height
- W: output matrix width
- K_{filter}: kernel size
- C_{in}: input matrix channels
- C_{out}: output matrix channels
- N: output matrix rows
- M: output matrix columns
- K: inputs matrices reduction dimension

To map a convolution on a GEMM, each filter has to be flattened in a row of the first operand matrix, generating a $C_{out}K^2_{filter} \times C_{in}$ matrix. The second operand matrix is created by composing each column as a depthwise sequence of patches from the input matrix, as figure 2.6 shows. The resulting matrix has size $K^2_{filter}C \times HW$. Performing a matrix multiplication between these two operands leads to an output matrix of size $C_{out} \times HW$, where HW corresponds to M and C_{out} to N. K instead indicates the reduction dimension $C_{out}K^2_{filter}$.

	Operands size					GEMM		
Workloads	H	W	K_{filter}	C_{in}	C_{out}	N	M	K
wl1	28	28	3	48	384	384	784	432
wl2	38	38	3	56	448	448	1444	504
wl3	28	28	3	128	128	128	784	1152
wl4	14	14	3	256	256	256	196	2304

Table 5.4: Size of the different workloads.

Table 5.5 shows the resulting clock cycles required to execute each workload. The results are obtained considering the size of the workloads and the equation derived in the previous section to compute the required clock cycles to execute a workload. It can be seen that a balanced tiling executes workloads in fewer clock cycles with respect to the other architectures. The exception is represented by workload 4, which performs better on a design processing a larger number of element on the reduction dimension at each time. This is due to the workload 4 K, which is way larger than the other sizes, affecting the total cycles count. For this reason, processing 16 elements at a time shows a higher gain with respect to the other designs.

Architecture	Workload 1	Workload 2	Workload 3	Workload 4
Scalar 32x32	158401	386401	125801	134401
STA 2x16x2_4x4	183457	440916	131713	118401
STA 4x16x4_2x2	155233	380101	122304	**116801**
STA 2x4x2_8x8	155233	382201	122305	121473
STA 4x4x4_4x4	**141121**	**351625**	**117601**	122305

Table 5.5: Total clock cycles required by different designs to perform the whole computation of each workload.

The best architecture in terms of clock cycles is STA_4x4x4_4x4, but clock frequency is not considered yet. Table 5.6 shows the execution time required by each design, considering the maximum frequency of each architecture reported in table 5.3 and combining it with the required clock cycles presented in table 5.5. As a matter of execution time, the best results are obtained by a 32x32 scalar systolic array. However, as previously showed, it comes at the cost of power consumption and resources occupation.

Architecture	F_{max}	wl1	wl2	wl3	wl4
Scalar 32x32	125 MHz	**1,270 ms**	**3,090 ms**	**1,010 ms**	**1,070 ms**
STA 2x16x2_4x4	45.45 MHz	4,040 ms	9,700 ms	2,900 ms	2,600 ms
STA 4x16x4_2x2	45.45 MHz	3,420 ms	8,360 ms	2,700 ms	2,570 ms
STA 2x4x2_8x8	100 MHz	1,550 ms	3,820 ms	1,220 ms	1,220 ms
STA 4x4x4_4x4	90.9 MHz	1,550 ms	3,870 ms	1,290 ms	1,350 ms

Table 5.6: Execution time of different designs at their maximum clock frequency processing different workloads.

To actually assess the overall performances of each architecture, other two metrics are considered: GOPs/s and GOP/mW. Table 5.7 shows the theoretical peaks of each architecture in terms of GOPs/s and GOPs/mW. From the results obtained, a scalar systolic array of size 32x32 seems to be the best choice in terms of theoretical operations performed in a second and for GOPs/mW.

However, when looking at the effective peak performance obtained after executing each workload, it is not the most efficient variant in terms of actual utilization. In fact, workloads 1, 2 and 3 better exploit STA_4x4x4_4x4, reaching a utilization factor of 90% or more. On the contrary, when executing workload 4, designs with larger dot products like STA_2x16x2_4x4 and STA_4x16x4_2x2 provide a utilization of 95%.

Architecture	OPs	F_{max} [MHz]	Power [mW]	GOPs/s	GOPs/mW
Scalar 32x32	2048	125	470	256	0.55
STA 2x16x2_4x4	2048	45.45	172	93	0.54
STA 4x16x4_2x2	2048	45.45	170	93	0.55
STA 2x4x2_8x8	2048	100	498	205	0.41
STA 4x4x4_4x4	2048	90.9	517	186	0.36

Table 5.7: Theoretical peak GOPs/s and GOPs/mW.

Architecture		WL 1	WL 2	WL 3	WL 4
Scalar 32x32	GOPs/s	205	210	230	215
	Utilization %	80%	82%	89%	84%
STA 2x16x2_4x4	GOPs/s	64	67	80	88
	Utilization %	69%	72%	86%	**95%**
STA 4x16x4_2x2	GOPs/s	76	77	86	89
	Utilization %	82%	83%	92%	**95%**
STA 2x4x2_8x8	GOPs/s	168	171	189	190
	Utilization %	82%	83%	92%	93%
STA 4x4x4_4x4	GOPs/s	168	169	179	171
	Utilization %	**90%**	**91%**	**96%**	92%

Table 5.8: Effective peak GOPs/s.

Chapter 6

Conclusions

The usage of deep neural networks in different fields increased so much that inference was moved from the cloud to edge devices. However, common edge devices present stringent constraints in terms of power consumption and resource occupation. This led to the development of custom architectures used to accelerate DNNs. Usually, to design this kind of architectures, a DNN workload is first analyzed, then the accelerator is developed and finally it is used to assess the end-to-end performance. As new DNNs and new accelerators are constantly developed, in particular on reconfigurable platforms like FPGAs, agile automation tools are needed to quickly navigate the design space.

The presented framework exploits the metaprogramming capabilities of magma, a hardware construction language embedded in Python, to generate custom systolic tensor arrays minimizing design and verification efforts. It provides smart templates which automatically build the whole structure of a systolic array and the corresponding control units starting from the specification of a processing element. The framework was used to explore the design space and to assess the effects of different knobs like data bitwidth, array size, PE structure and sparsity support, on resource occupation, power consumption and latency. The obtained results show that, as expected, a four times increasing size of the arrays correspond to a $3-4$ times growth of area occupied and power consumed. In addition, increasing the data bitwidth leads to a linear increase of power consumption and area occupation. Moreover, they reveal a non-trivial trade-off introduced by tensor systolic arrays characterized by tiling on input matrix rows and columns and on the reduction dimension. Considering a fixed number of MAC per cycle, designs employing tiling on rows and columns show a saving of about $10-20\%$ in terms of resource occupation, at the cost of power consumption, with an increase of almost 80% and a worsening in latency. On the other hand, TPEs accumulating more than two operands on the same register show a reduction in both area occupation and power consumption of about 50% and 40%, respectively. This gain comes at the

cost of latency, since their complexity affect the maximum clock frequency reach by the architecture. In addition, different workloads benefit of different systolic array configurations, showing a clear gain when the power consumption introduced by rows and columns tiling is balanced by the one reduced when performing high b_tiling-way dot products. However, the execution time depends also on the maximum clock frequency of each architecture. Another way to assess the performance of an architecture is estimate the GOPs/s and GOPs/mW. The considered 32x32 scalar array is the best choice in terms of GOPs/s with an average of 215. STAs reach an average total amount of 175 GOPs/s while STAs with high B-way dot products are the worst with 78 GOPs per second. However, STAs are the best designs in terms of effective utilization reaching an average of 90% of the theoretical estimation.

The revealed trade-offs demonstrate the need to quickly navigate the design space to find a suitable solution for the desired DNN application. That is why agile automation tools are required to keep raising the efficiency of domain-specific accelerators.

Although this framework is a useful tool to reduce design and verification effort, it may be improved even more. For example, it may be extended with additional smart templates to support other dataflows, not only output stationary. Moreover, control units managing the computation introduce bubbles delaying the operations. Therefore, they may be better designed to reduce the number of clock cycles required to perform a whole computation. Lastly, the sparse systolic tensor array generator could be improved, with a strong focus on the logic decoding the bitmap, which is not efficient at all.

Bibliography

[1] *Neural Networks*. IBM Cloud Education. URL: https://www.ibm.com/cloud/learn/neural-networks (visited on 11/14/2021) (cit. on p. 5).

[2] Nick McCullum. *Deep Learning Neural Networks Explained in Plain English*. URL: https://www.freecodecamp.org/news/deep-learning-neural-networks-explained-in-plain-english/ (visited on 11/14/2021) (cit. on p. 6).

[3] *Convolutional Neural Networks*. IBM Cloud Education. URL: https://www.ibm.com/cloud/learn/convolutional-neural-networks (visited on 11/05/2021) (cit. on p. 8).

[4] Manas Sahni. *Anatomy of a High-Speed Convolution*. URL: https://sahnimanas.github.io/post/anatomy-of-a-high-performance-convolution/ (visited on 11/05/2021) (cit. on pp. 10–13, 15).

[5] Alexander Matthes, Rene Widera, Erik Zenker, Benjamin Worpitz, Axel Huebl, and Michael Bussmann. «Tuning and Optimization for a Variety of Many-Core Architectures Without Changing a Single Line of Implementation Code Using the Alpaka Library». In: Oct. 2017, pp. 496–514. ISBN: 978-3-319-67629-6. DOI: 10.1007/978-3-319-67630-2_36 (cit. on p. 14).

[6] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. «Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks». In: *ACM SIGARCH Computer Architecture News* 44.3 (2016), pp. 367–379 (cit. on pp. 16, 17, 19).

[7] Clément Farabet, Berin Martini, Benoit Corda, Polina Akselrod, Eugenio Culurciello, and Yann LeCun. «Neuflow: A runtime reconfigurable dataflow processor for vision». In: *Cvpr 2011 Workshops*. IEEE. 2011, pp. 109–116 (cit. on p. 16).

[8] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. «Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning». In: *ACM SIGARCH Computer Architecture News* 42.1 (2014), pp. 269–284 (cit. on p. 17).

[9] Xuan Yang et al. «Interstellar: Using halide's scheduling language to analyze dnn accelerators». In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 2020, pp. 369–383 (cit. on pp. 17, 20).

[10] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. «Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines». In: *Acm Sigplan Notices* 48.6 (2013), pp. 519–530 (cit. on p. 17).

[11] A. Samajdar, J.M. Joseph, Y. Zhu, P. Whatmough, M. Mattina, and T. Krishna. «A Systematic Methodology for Characterizing Scalability of DNN Accelerators using SCALE-Sim». In: 2020 (cit. on pp. 19, 20, 22, 24, 25).

[12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. «Imagenet classification with deep convolutional neural networks». In: *Advances in neural information processing systems* 25 (2012), pp. 1097–1105 (cit. on p. 19).

[13] Ananda Samajdar, Yuhao Zhu, Paul Whatmough, Matthew Mattina, and Tushar Krishna. «SCALE-Sim: Systolic CNN Accelerator Simulator». In: (2019). arXiv: 1811.02883 [cs.DC] (cit. on p. 23).

[14] Norman P Jouppi et al. «In-datacenter performance analysis of a tensor processing unit». In: *Proceedings of the 44th annual international symposium on computer architecture*. 2017, pp. 1–12 (cit. on pp. 25, 26).

[15] Hasan Genc et al. «Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration». In: *Proceedings of the 58th Annual Design Automation Conference (DAC)*. 2021 (cit. on pp. 25, 27).

[16] Hasan Genc et al. «Gemmini: An agile systolic array generator enabling systematic evaluations of deep-learning architectures». In: *arXiv preprint arXiv:1911.09925* 3 (2019) (cit. on p. 25).

[17] Zhi-Gang Liu, Paul N. Whatmough, and Matthew Mattina. *Sparse Systolic Tensor Array for Efficient CNN Hardware Acceleration*. 2020. arXiv: 2009.02381 [cs.AR] (cit. on pp. 28–31).

[18] Zhi-Gang Liu, Paul N. Whatmough, Yuhao Zhu, and Matthew Mattina. *S2TA: Exploiting Structured Sparsity for Energy-Efficient Mobile CNN Acceleration*. 2021. arXiv: 2107.07983 [cs.AR] (cit. on p. 30).

[19] Pat Hanrahan. *Magma github*. URL: https://github.com/phanrahan/magma (visited on 11/23/2021) (cit. on p. 33).

[20] Suyog Gupta and Mingxing Tan. *EfficientNet-EdgeTPU: Creating Accelerator-Optimized Neural Networks with AutoML*. URL: https://ai.googleblog.com/2019/08/efficientnet-edgetpu-creating.html (visited on 11/24/2021) (cit. on p. 67).

[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. «Deep residual learning for image recognition». In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778 (cit. on p. 67).