POLITECNICO DI TORINO

Master Degree Course in Electronic Engineering

Master Degree Thesis

Design and Evaluation of Reconfigurable Systolic Arrays for Neural Networks

Advisors
Prof. Mario Roberto Casu
Dr. Edward Manca

Candidate
Sergio Ivano Fierro

Academic Year 2024-2025

Summary

In recent years, Deep Neural Networks (DNNs) have achieved unprecedented accuracy across a wide range of tasks. These gains, however, come with substantial increases in model complexity and per-inference computational cost. As a result, deploying DNNs presents new challenges, and devising efficient methods to execute their computations has become a central concern in research. Modern DNN workloads comprise many nested loops, with Multiply-and-Accumulate (MAC) operations dominating. In this context, Systolic Arrays (SAs) have emerged as an architecture that connects and coordinates large numbers of Processing Elements (PEs) operating in parallel. Usually, SAs are composed of PEs that communicate only with their direct neighbors. This neighbor-to-neighbor connectivity allows them to have low fan-out connections. Moreover, SAs supports dataflow organizations that enable operand reuse, and relax the back-pressure to the memory to feed the PEs with new values. Finally, their regular structure aligns well with lightweight control units, usually composed by counters. All these properties make them a good choice to compute the kernels of DNNs such as matrix multiplication and convolution. On the other hand, both the shape of the array — i.e., the number of elements along rows and columns — and the dataflow scheme the array is designed for — namely Output Stationary (OS), Weight Stationary (WS), or Input Stationary (IS) — have a significant impact on computational efficiency. Each combination of shape and dataflow determines a computation strategy that better fits a given algorithm and a different design point in the latency, silicon area, and power consumption analysis. The goal of this thesis is to explore and design reconfigurable SAs that support multiple shapes and/or multiple dataflows. To this end, I explored the design space given by SAs with different shapes and dataflows, over a class of selected DNN kernels - i.e. convolutions, linear, and attention layers. For each configuration I verified the correctness with RTL simulation tools. Moreover, I collected the number of clock cycles needed by the SA to complete the computation, and I synthesized them on a 28 nm digital library to also collect latency, silicon area, and power consumption results. Overall, these data allowed me to rank the various shapes and dataflows on an efficiency metric of OPC/mW, and to select the most efficient SA configurations. Once the best configurations have been selected, I designed and verified a reconfigurable SA supporting more than one configuration in the same design. Since the number of configurations to support directly influence the overhead coming from the reconfigurability, I explored architectures that implements two/three configurations at most in the same SA. This design has been simulated and synthesized on the same 28 nm technology library to validate its efficiency with the same OPC/mW metric. The study demonstrates the importance of architectural choices in the SA design process and proposes a path to have more efficient reconfigurable SAs that can optimally execute more than one DNN algorithm with the same SA structure. Looking forward, this approach may serve as a foundation to study and efficiently compute the algorithms of novel DNN layers, leveraging run-time reconfiguration.

Acknowledgements

I would like to sincerely thank Professor Mario Roberto Casu and Dr. Edward Manca, without whose expert and constant guidance this work would not have been possible. Also, a special acknowledgment goes to the entire research group whose availability, competence and constructive support provided a solid foundation for the development of this thesis.

I want to thank my family, whose constant support and encouragement not only made possible this academic journey but has always accompanied me at every stage of my life. Particularly, I am grateful to my brother, whose presence and constant support, even from afar, have been of inestimable value, especially during the most challenging times.

I am also deeply beholden to my significant other for her profound understanding, continuous encouragement, and for the meticulous contribution, especially for revising this manuscript, which improved its clarity and overall cohesion.

Finally, I would like to thank all my friends, whose constant presence and support accompanied me throughout my university journey, making it lighter, richer, and unforgettable.

Contents

Chapter 1

Introduction

1.1 Deep Learning Evolution and Efficiency Challenges

1.1.1 Evolution of deep learning

With the advent of the digital age, much of the data that was once accessible only in an "analog" form has been transferred to and made available on digital systems. In particular, today—considering the enormous amount of information produced, processed, and stored in scientific, industrial, and research contexts—we increasingly speak of a true "era of big data."

Consequently, traditional tools for the analysis and manipulation of collected information have had to evolve. It is within this context that Neural Network Learning emerged — a system designed to emulate the behavior of the human brain. When comparing a machine and a brain in terms of data storage and access, their differences are immediately clear: the human brain (which can roughly be compared to a computer's CPU) does not have a central memory unit where information is saved and later retrieved (as occurs in a computer's hard drive). Instead, the brain creates connections among accessible data, allowing it to retrieve past information through associative processes. The goal, in this regard, has always been to replicate such behavior in machines, avoiding the creation of massive, slow, and difficult-to-access data repositories.

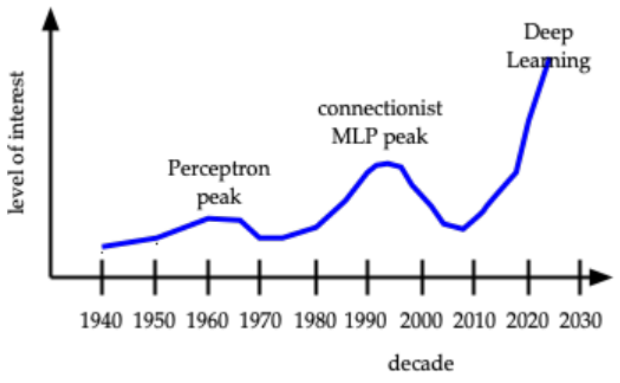

Figure 1.1: Three wave of neural network research[1].

This way of envisioning machines as networks of interconnected neurons has actually existed in research much longer beyond anything one could imagine. As reported in various studies, neural network learning has gone through three main waves of research activity. This trend is clear in Figure 1.1: the first wave dates back to the 1960s, with *Frank Rosenblatt*'s "Perceptron" [2], the first supervised learning algorithm capable of classifying linear patterns, *Bernard Widrow* and *Marcian Hoff*'s "MADALINE" [3], which introduced for the first time a multilayer network based on adaptive learning rules — laying the foundation for future neural architectures.

The second wave started in the 1980s, when John Hopfield showed that recurrent neural networks could address NP-hard combinatorial optimization problems . Around the same time, the introduction of the backpropagation algorithm made it possible to effectively train multilayer networks, sparking renewed interest in deep learning research

Finally, the third and most significant wave is the one we are experiencing today. It began around 2012, when deep convolutional neural networks were first used for image classification [4]. The fundamental difference between the early neural networks and those designed for deep learning lies in the number of hidden layers, i.e., the computational layers between the input and output nodes. In recent years, deep learning—thanks especially to its ability to process and learn from much larger volumes of data than previous systems—has become the reference paradigm within the field of machine learning.

1.1.2 Increase in model complexity

In recent years, the field of deep learning has experienced rapid growth not only in terms of application adoption but also—and above all—in the size and complexity of the models used. This evolution is justified by the empirical scaling laws, which show that increases in the number of parameters, the amount of training data, and the available computational resources lead to better performance. At the same time, there is a human drive to tackle increasingly complex tasks (such as computer vision and natural language processing) that require models capable of generating ever richer and more abstract representations.

A clear example is represented by language models—for instance, GPT-2 and related architectures— which have demonstrated that the generalization error (i.e., how well a learning model performs on unseen data) decreases according to a power law as the model size, training data, and computational budget grow. These studies [5] suggest that, to improve model quality, one of the most effective strategies is simply to scale up: more parameters, more data, and more computation.

In practice, models such as ResNet (Residual Networks) used in computer vision architectures, or attention-based models like Transformers used in natural language processing, have become extremely deep or contain billions of parameters. This design is motivated by the need for models capable of robustly understanding entities, relationships, and contexts in complex tasks such as image recognition, machine translation, or text generation. In this sense, model complexity is regarded as a necessary means to achieve the desired performance leap.

Two key factors have enabled the evolution of neural networks in this direction: the exponential increase in available data and the rapid development of specialized hardware (GPUs, TPUs, accelerators, systolic arrays, FPGAs), which can now train increasingly dense networks on a daily basis. As a result, large-scale models are today the rule rather than the exception in cutting-edge research. For instance, it has been estimated that the computational requirements of certain language models have increased by several orders of magnitude in just a few years—according to a recent analysis, by approximately 300,000$\times$ between 2012 and 2018 [6].

In summary, the growing complexity and scale of deep learning models stem from two main factors. On one side, the abundance of data, advances in hardware, and improved design expertise drive the development of larger architectures. On the other, empirical scaling laws and the quest for emergent capabilities make increasing model size more beneficial than focusing just on architectural improvements.

1.1.3 Computational and energy inefficiency in training and inference

Such progress has led to a significant increase in the resources required — both in terms of computational power and energy consumption — during both the training phase, in which the model processes large amounts of data to learn its parameters, and the inference phase, in which the already trained model is employed to generate predictions or decisions on new inputs. Both stages share a fundamental characteristic: the need to perform an extremely high number of mathematical operations, mainly multiplications and accumulations (MAC, Multiplication and Accumulation), whose volume grows proportionally with the model's complexity and depth.

During these operations, activations, weights and gradients must be continuously read from and written to memory with a constant exchange of information between different cache levels, main memory and compute arrays. These data flows entail considerable energy expenditure, not only for the arithmetic operations themselves but also for memory transfers and synchronization among the various hardware units. Overall efficiency therefore depends not only on the algorithmic design of the model but also on the hardware's ability to manage these operations efficiently, minimizing waste due to latency or redundancy.

From this perspective, hardware efficiency emerges as a critical factor. Although modern GPUs and specialized accelerators provide remarkable computational power for artificial intelligence tasks, they also entail significant energy demands, particularly during prolonged use or when training large-scale models.

Energy consumption during the inference phase should not be underestimated either, considering that these models are now used daily by millions of users. The study by Patterson et al.(2021) [7] shows that, for certain large-scale models, the total energy consumption resulting from distributed inference can even exceed that required for the initial training, particularly in contexts where the model is executed millions of times per day.

In summary, the issue of efficiency — both computational and energetic — represents one of the major challenges in neural network engineering today. It concerns not only the speed at which a model can be trained or deployed, but also directly affects the overall sustainability of the field and the possibility of making artificial intelligence a truly scalable, accessible, and responsible technology.

1.2 Contribution and Thesis Objectives

This thesis lies within the context of hardware accelerator design for Deep Neural Networks (DNNs), with a particular focus on architectural efficiency and execution flexibility. In recent years, the prevailing approach has been to design dedicated hardware optimized for the execution of a single algorithm, aiming to achieve the lowest possible latency and power consumption. However, it is often necessary to handle layers that exhibit very different computational characteristics — not only between the training and inference phases, but sometimes even within the same network, depending on the specific operation being executed at a given time step.

This exposes the fundamental problem that this work aims to address: it is not feasible to rely on a fixed hardware architecture that remains optimal under all possible conditions.

In this context, Systolic Arrays (SAs) have emerged as a highly regular and scalable computational paradigm, particularly well-suited for the execution of massively parallel workloads [8]. SAs differ from one another in terms of the organization and dataflow of their Processing Elements (PEs). At the same time, these very design choices — which define their operational flexibility — also represent their main limitation. Indeed, both the shape of the array and the dataflow scheme are fixed at design time. Since different classes of layers within the same neural network exhibit different computational patterns and reuse characteristics, one must select a single architectural configuration that represents a reasonable compromise across all layers, rather than the optimal solution for each individual case.

The scenario described above forms the starting point for this work, whose goal is to explore, in terms of area and power, the existing design space of systolic architectures and to propose a new approach based on architectural reconfigurability. The core idea is to enable the array to dynamically adjust its shape and dataflow in order to adapt to the computational requirements of the currently executed layer. In this way, by incurring only a modest area overhead (due to the additional reconfiguration circuitry), it becomes possible to balance power consumption and latency, achieving improved performance or energy efficiency depending on the desired operating point.

The main contribution of this thesis unfolds along two complementary directions. First, a systematic performance study was conducted to establish a solid reference design to be used for comparison with the proposed architecture. Several SA configurations were analyzed, varying both the shape and the dataflow, and their impact was evaluated across a representative set of neural network workloads. Second, a Systolic Array capable of supporting multiple shapes within a single hardware design was implemented and verified. The architecture is equipped with a control mechanism that could, in principle, modifies dynamically — even during the execution of a single layer — to the configuration that offers the best performance or energy trade-off.

Through this approach, the thesis aims to provide a general design and evaluation methodology applicable to a wide range of systolic accelerators, laying the groundwork for future research toward dynamic and adaptive architectures for efficient neural network computation.

1.3 Thesis Outline

The present thesis is organized as follows.

Chapter 2 provides the theoretical and architectural foundations required to understand the design decisions made in this work. It outlines the fundamental concepts of Deep Neural Networks (DNNs), detailing their core computational operations and data reuse patterns. The discussion then shifts to Systolic Array architectures, explaining their operating principles, the various dataflow strategies, and the key design trade-offs involving performance, area, and power consumption

Chapter 3 describes the design space exploration conducted on different configurations of Systolic Arrays. Several architectures with varying shapes and dataflows are analyzed and evaluated in terms of performance, area, and energy consumption. The results of this analysis are discussed to identify the configurations that achieve the highest efficiency according to a performance metric defined as "Operations Per Cycle" per Watt (OPC/W). In addition, a short discussion is included on how neural networks can be mapped and executed on Systolic Arrays, and why these architectures are particularly suitable for efficiently running such algorithms.

Chapter 4 presents the design and implementation of the proposed reconfigurable Systolic Array architecture, capable of supporting multiple shapes within the same hardware design. The architectural modifications required to support multiple configurations are described, along with the approach adopted to activate each configuration and to assess its impact in terms of area and power consumption. Furthermore, this chapter reports the experimental evaluation of the reconfigurable architecture, followed by an analysis of the proposed design and a comparison with the fixed-shape Systolic Arrays introduced in the previous chapters, highlighting the results obtained in terms of efficiency and flexibility. The reported data refers to different neural network workloads, using as basis for comparison the number of clock cycles, silicon area, and power consumption.

Chapter 5 reviews what has been achieved in this thesis, drawing attention to the main contributions. The discussion also looks ahead, suggesting how some of the ideas developed here could evolve into future applications or be explored under different design constraints.

Chapter 2

Conventional Systolic Arrays

The following chapter aims to explain the fundamental concepts necessary to justify and understand some of the design choices made during the experimental phase concerning the systolic architecture.

The first part introduces the operating principles, structure, and characteristics that make Systolic Arrays (SAs) particularly well suited for parallel computation, and, in this sense, for efficiently executing the algorithms at the core of neural networks.

The second part, on the other hand, provides an overview of the architectures available in the current state of the art, with the goal of contextualizing the evolution of SAs in the field of hardware acceleration for deep learning and identifying the existing solutions that inspired some of the design decisions presented in this thesis.

2.1 Systolic Array Fundamentals

2.1.1 Operating principles and general architecture

The concept of Systolic Arrays was first introduced by *Kung* and *Leiserson* in 1979 [8]. Their proposal described an architecture designed to achieve the highest possible degree of parallelism by exploiting a regular and highly organized structure composed of a number N of Processing Elements, referred to as PEs.

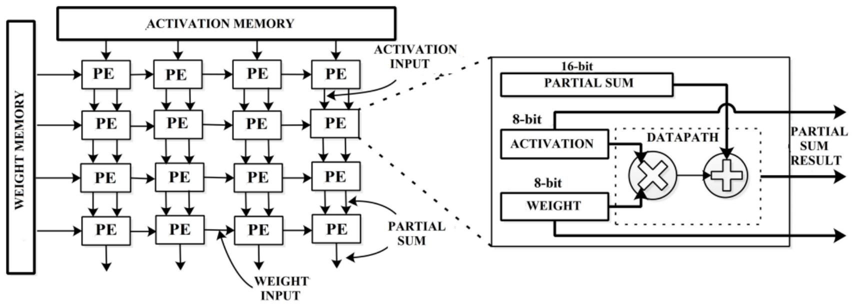

Figure 2.1: Implemented TPU's Systolic Array model[9].

The term systolic was chosen by *Kung* and *Leiserson* by analogy with the functioning of the human circulatory system. In the heart, during the systole phase, blood is pumped in a rhythmic and coordinated way through a network of vessels. Similarly, in a Systolic Array (SA), data flows through a network of Processing Elements following a regular and synchronized rhythm.

A systolic array is essentially a two-dimensional grid of PEs interconnected with one another. Since this architecture is not limited to a specific application domain, the internal behavior of the PEs may vary depending on the computational task the array is meant to perform. In the case analyzed here, each PE mainly performs Multiply-and-Accumulate (MAC) operations. In the baseline architecture studied by *Kung* and *Leiserson*, the PEs communicate only with their direct neighbors. This local communication reduces interconnection complexity and allows data to propagate through the array in a deterministic and deeply pipelined manner.

The operation of a systolic array can be easily understood by analyzing how matrix multiplication (MAT-MUL) can be mapped onto such an architecture. In this scenario, one of the matrices (e.g., A) is transmitted into the array from left to right, while the other matrix (e.g., B) is injected from the bottom and propagated vertically. Each PE—assuming, as in this case, that it performs MAC operations—computes a partial product and forwards it to the next PE, which adds it to its own result. As computation proceeds, partial sums move diagonally across the array until they reach the upper-right corner, where the final results are produced. This example highlights the key characteristics of Systolic Arrays:

- **Regularity**: structure is composed of identical PEs connected periodically.
- **Locality**: short and predictable interconnections between elements.
- **Parallelism**: many operations are performed simultaneously.

These properties make systolic architectures particularly suitable for compute-intensive tasks such as matrix multiplications, convolutions, and other linear algebra kernels that form the core of modern Deep Neural Networks (DNNs).

2.1.2 Key concepts: data reuse, spatial and temporal loops

One of the greatest advantages of systolic architectures is their ability to exploit, in an extremely efficient way, the data reuse properties that characterize Machine Learning (ML) algorithms. On the other hand the main bottlenecks in high-performance computing systems is the latency introduced by the countless accesses to the outer memory layers – generally slower than the processing system itself. Systolic Arrays (SAs) are designed to minimize the number of read–write cycles to external memories by ensuring that the same operands are reused multiple times within the array.

This efficient data reuse is made possible by the regular and synchronized propagation of information among the various PEs. Assuming that the PE at position (x, y) within the computation matrix needs, at time t+1, the same data d it had at time t, one can imagine that d does not move to its neighbor but instead circulates back to itself, thereby avoiding a second memory access.

In the context of systolic arrays, it is in fact possible to identify two distinct types of reuse: spatial and temporal.

- **Temporal reuse** occurs when data fetched by a PE is used by itself in the next clock cycles .
- **Spatial reuse** occurs when data fetched by a PE are transferred to neighbor PEs to be reused in the next clock cycles.

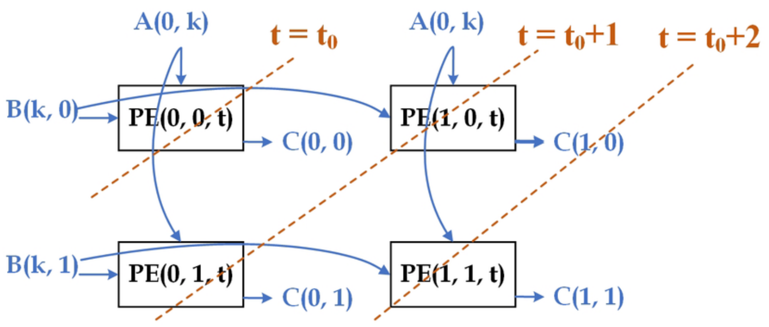

Figure 2.2: Spatial-temporal mapping onto a systolic array with its data access and data reuse.[10]

Properties shown in Figure 2.2 prove very useful when mapping ML algorithms, since the fundamental operations that compose such algorithms — and in particular deep neural networks (DNNs, CNNs, Transformers, etc.) — can be represented as a set of nested loops. These loops, which iterate over the dimensions of the input tensors, the weights, and the output features, describe how the data required for computation can be reused within the array. Furthermore, they algorithmically define the entire computational process: for each output element, sums and products are performed by iterating over the internal dimensions.

For example, a matrix multiplication — at the heart of many operations in ML models — can be formalized as:

$$C_{i,j} = \sum_{k=0}^{k-1} A_{i,k} \times B_{k,j} \tag{2.1}$$

that, in pseudo-code, is equal to three nested loop:

Algorithm 1 Matrix Multiplication (Nested Loops Representation)

```
1: for $i = 0$ to $M - 1$ do                                  ▷ Rows of $A$ and $C$
2:     for $j = 0$ to $N - 1$ do                               ▷ Columns of $B$ and $C$
3:         $C[i][j] \leftarrow 0$
4:         for $k = 0$ to $K - 1$ do                           ▷ Inner dimension loop
5:             $C[i][j] \leftarrow C[i][j] + A[i][k] \times B[k][j]$
6:         end for
7:     end for
8: end for
```

In the same way, convolutions in Convolutional Neural Networks (CNNs) or attention mechanisms in Transformers can be described as more complex loop structures, yet always based on the same principle. From the nested loops that describe matrix multiplication, which are directly mapped into hardware, it is possible to distinguish:

- **Spatial loops**: they represent operations that are executed simultaneously by different PEs, exploiting the parallelism of the array by distributing operations along the physical dimensions.
- **Temporal loops**: they represent operations that, over time, are executed by the same PE. In fact, each clock cycle introduces new input data, while the previous ones move to the subsequent PEs.

Returning to the example of matrix multiplication execution, each element of matrix A is transmitted horizontally along a row, while the elements of matrix B move vertically along a column. This means that a single element of A can be used by all PEs in the same row, and an element of B by all those in the same column. This scheme achieves extremely efficient spatial reuse and temporal reuse. An additional advantage is that the entire computation process takes place under very simple local control: each PE is governed by counters or regular synchronization signals, without the need for a centralized control unit or complex scheduling logic. This structural simplicity makes Systolic Arrays particularly scalable, both in terms of grid size and operating frequency. In the context of Deep Neural Networks (DNNs), this property becomes crucial.

2.1.3 Analysis of different dataflows

The execution time and efficiency of a Systolic Array (SA) in performing a given algorithm do not depend only on the number of Processing Elements (PEs) and their spatial organization, but also — and above all — on the logic underlying the interconnections, that is, on how data are moved and utilized within the structure.

From this, it is possible to introduce the concept of "dataflow," which refers to the strategy through which operands and intermediate results move within the array. In a Machine Learning algorithm, three different types of data can be distinguished: inputs, weights, offsets, and outputs, which, by drawing a correspondence with the previously discussed MAT-MUL case, correspond - respectively - to A, B, and C.

The dataflow simply specifies which of these data are kept stationary within the PEs and which, instead, are circulated among them. It is worth noting that these three data types necessarily have different bitwidths, and depending on how the layer is mapped onto the SA, the various dataflows may require a different number of memory accesses, that can negatively influence execution efficiency in terms of latency and power. In real systems, the choice of dataflow is closely related to the type of layer being accelerated, since each algorithm exhibits different data reuse patterns. The three most common types — widely employed in architectures designed for deep neural network acceleration — are:

- Output-Stationary (OS)
- Weight-Stationary (WS)
- Input-Stationary (IS)

Output-Stationary (OS)

In the Output-Stationary case, the partial results are kept locally within the PEs until the computation is completed. Inputs and weights, on the other hand, flow among the various nodes, contributing to the accumulation of the final result. Mathematically, the operation can be described as:

$$P_{(i,j)}^{(t+1)} = P_{(i,j)}^{(t)} + A_{(i,k)}^{(t)} \cdot W_{(k,j)}^{(t)} \tag{2.2}$$

In this case, $(C_{(i,j)})$ (the partial output) remains stationary within the PE until the computation of the entire element of the output matrix is completed. Only once the accumulation is finished the result is written to external memory. This approach minimizes the traffic of partial sums (in terms of bitwidth are the most significant), since only the final result leaves the corresponding PE. This type of dataflow is particularly suitable for Fully Connected (FC) layers — in these layers the number of weights is large and the final result depends on many intermediate products.

The OS case is also the simplest to implement, since it does not require diagonal paths for the propagation of partial sums: each PE locally manages its own accumulation. Therefore, the OS scheme significantly reduces the internal traffic of intermediate results and is particularly well-suited for highly dense layers, but it loses efficiency in the case of small computations.

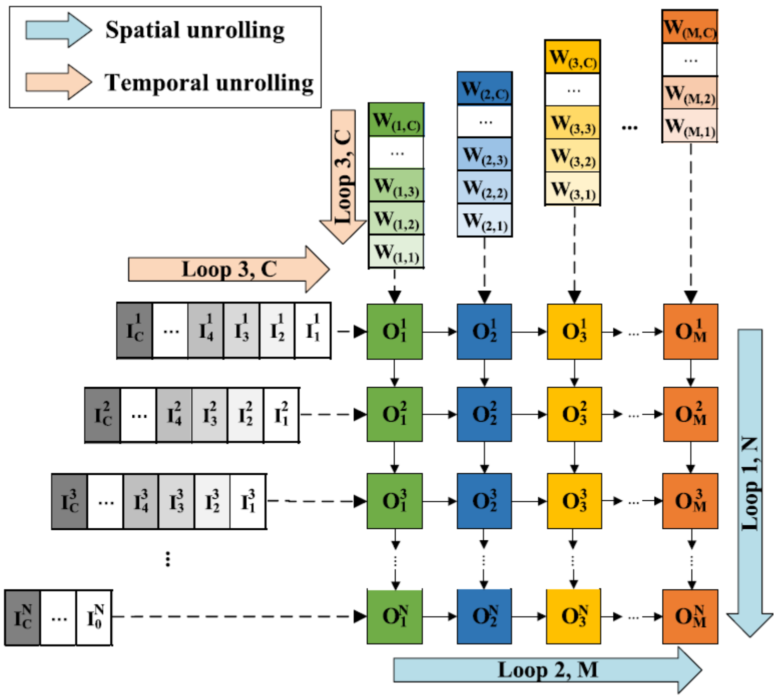

Figure 2.3: Systolic Array - Output-Stationary (OS) configuration[11]

Weight-Stationary (WS)

In the Weight-Stationary dataflow, the layer weights are preloaded and kept fixed within the PEs for the entire duration of the computation. Inputs (activations) and partial sums, instead, flow horizontally and vertically across the grid, interacting with the locally stored weights. Formally, each PE((i,j)) performs the operation:

$$P_{(i,j)}^{(t+1)} = P_{(i,j)}^{(t)} + A_{(i,k)}^{(t)} \cdot W_{(k,j)} \tag{2.3}$$

where:

- $(W_{(k,j)})$ is the weight kept fixed inside the PE
- $(A_{(i,k)}^{(t)})$ represents the input flowing horizontally
- $(P_{(i,j)}^{(t)})$ is the partial sum accumulated over time.

The key benefit of this approach is the maximization of weight reuse: each weight element ($W_{(k,j)}$) can serve multiple input values without requiring repeated reads from external memory. This substantially reduces the energy expenditure linked to weight retrievals, which can constitute a considerable fraction of the overall power consumption in these systems

In this architectures, each PE contains a small local memory (often just a simple register) - for WS dataflow used to store weights -, which are loaded only once per input batch. The WS approach proves particularly advantageous when each filter (i.e., a subset of weights) is reused to compute multiple "regions" of the input tensor. The drawback, on the other hand, is the intrinsic need to constantly move partial sums, which considerably increases internal data traffic.

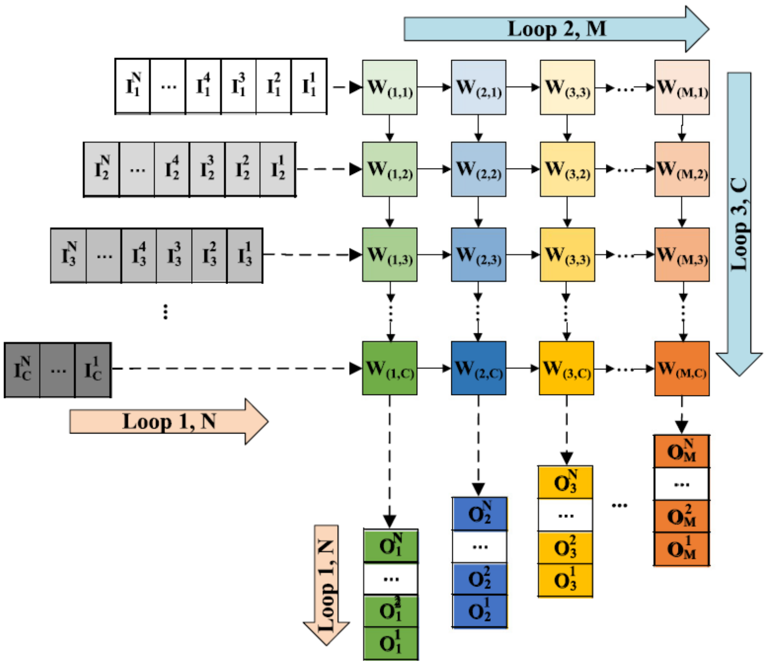

Figure 2.4: Systolic Array - Weight-Stationary (WS) configuration[11]

Input-Stationary (IS)

In the Input-Stationary dataflow, the input data remain fixed within the PEs, while the weights and partial sums are propagated across the array. The operation performed by each PE can be represented as:

$$P_{(i,j)}^{(t+1)} = P_{(i,j)}^{(t)} + A_{(i,k)} \cdot W_{(k,j)}^{(t)} \tag{2.4}$$

In this scheme, each input element ($A_{(i,k)}$) is reused for multiple combinations of weights, thus maximizing its spatial and temporal reuse. The result is a reduced number of input memory reads, which can be beneficial in terms of energy consumption.

The IS approach is beneficial whenever the same input is involved in generating multiple outputs, as seen in convolutional layers or attention blocks in Transformer models. However, this also results in increased bandwidth demands for weight transfers, since weights must be repeatedly transmitted across the computation array.

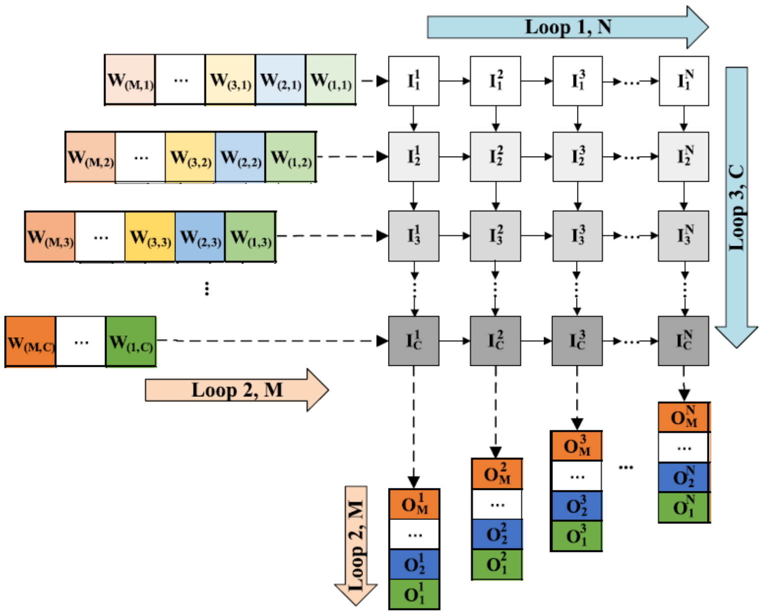

Figure 2.5: Systolic Array - Input-Stationary (IS) configuration[11]

2.2 State of the Art in SA Architectures

In recent years, Systolic Arrays have become one of the most widely adopted architectural models — both in industry and academia — when discussing accelerators for neural networks. Their widespread use is not only the result of the simplicity of the architecture — originate from the regularity of computation and the straightforward dataflow, as discussed in previous sections — but, above all, due to their versatility. As observed in recent years, SAs adapt very well to the increasingly diverse range of modern DNNs, which feature layers with different numerical and dimensional characteristics. Today, with the evolution of the systolic paradigm, designers are progressively moving away from the traditional square shape and fixed dataflow, techniques that are making SAs extremely efficient systems.

The most recent studies on these architectures have shown that their success is closely tied to the ease with which these systems scale. SAs can be replicated, aggregated, and/or organized into subgroups, forming very large computation matrices while still maintaining predictable behavior and a highly localized communication pattern. These characteristics make them ideal for optimization in terms of throughput per mm^2 (crucial aspect in advanced technology nodes).

The adoption of these systems in industrial accelerators has raised the emergence of increasingly specialized variants. Asymmetric arrays have been introduced to match the shapes and sizes of matrices typical of Transformers, and pod-based structures have been designed to improve horizontal scalability. There is also a growing integration of systolic pipelines within GPU-like architectures, and increasingly often, support for multiple numerical formats is provided. This variety of configurations demonstrates that the SA model is no longer bound to its historical form, but is instead becoming a building block that designers can customize according to constraints on area, bandwidth, and data reuse.

Very often, such accelerators are designed with the premise that the computational part is less restrictive than the data-movement model, for this reason they are adopted in data-heavy context like NN computation. In fact, the modularity of systolic arrays facilitates pairing with dedicated buffers and in some of the most modern architectural solutions, the shape of the array is designed based on available bandwidth rather than the opposite, as was common in older systems.

Considering all these aspects, the role of systolic arrays in modern architectures is no longer limited to the regular computation of matrix products. These systems are becoming a flexible paradigm that can be reinterpreted in various ways to respond to the increasing complexity of deep learning models. The following sections will explore how these solutions have materialized in industrial and academic applications, highlighting the differences among the various approaches currently available on the market.

Differences between SA and GPU-based computation

In the domain of neural-network accelerators, systolic arrays are often compared to GPUs, which today represent the reference platform for training and inference of most existing models. Both architectures aim for massive parallelism, but they differ substantially in the way data is managed, and these differences impact performance, efficiency, and predictability.

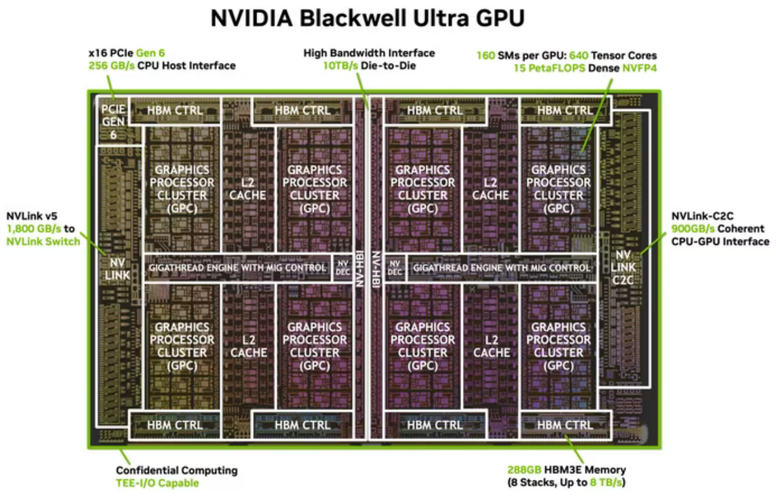

Figure 2.6: CUDA Cores - NVIDIA GPU[12]

GPUs (Figure 2.6) are general-purpose systems designed to maximize throughput by exploiting a very large number of independent cores. In these systems, operations are split into thousands of threads, each dynamically scheduled according to resource availability. Such a system is extremely flexible, allowing GPUs to be used in a wide range of fields. However, this flexibility comes at a cost: the control structure, the memory system, and the complex interconnection fabric must sustain a high level of concurrency across computing units. The result is an extremely powerful architecture, but one with a significant energy cost.

SAs, on the other hand, follow a fundamentally different approach. Computation is not divided into independent threads but is distributed across a network of identical units. Data flows through the PEs, which compute a partial result and propagate it to the next node. This organization eliminates the need for complex control and drastically reduces dependence on external memory thanks to the high reusability of operands at the local level. The consequence is clear: the ratio between energy consumed and operations performed is far more favorable, making this architecture ideal where efficiency is a fundamental requirement.

Another significant difference lies in temporal predictability. In GPU-based systems, data latency can vary substantially depending on core saturation and the state of the memory system supporting the compute units. In systolic arrays, the dataflow is evaluable regardless of external conditions, allowing the global computation latency to be determined in advance—an essential feature in real-time or embedded systems where timing stability is a strict requirement.

Finally, while GPUs rely heavily on processing large batches in parallel — experiencing notable efficiency loss when required workloads are small — systolic arrays maintain stable performance even on significantly smaller batch sizes, since their efficiency depends on the match between the shape of the array and the matrices being processed. This peculiarity makes them particularly suitable for inference workloads, where memory systems are a major limitation (especially for large batch sizes).

In TPUs[13], for example, the systolic array becomes the structure that concentrates the computational capability, while the rest of the chip is organized to continuously feed it with weights and activations. This model has significantly contributed to the widespread adoption of SAs in subsequent designs, demonstrating how a highly specialized implementation can outperform general-purpose solutions in terms of energy efficiency and throughput predictability.

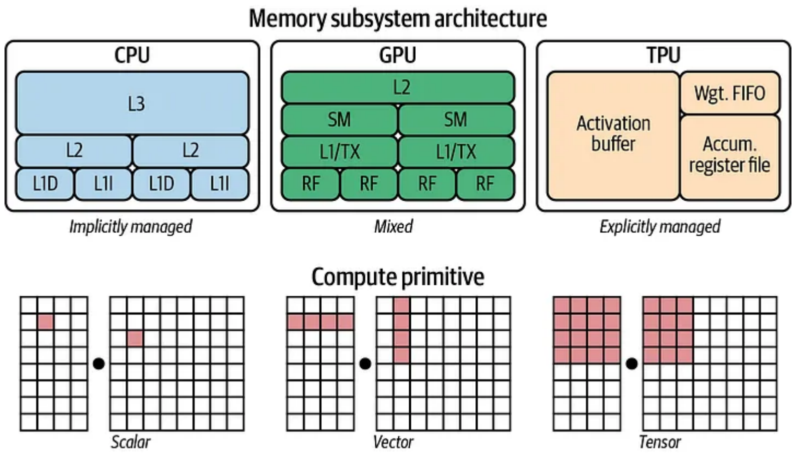

Figure 2.7: CPU vs GPU vs TPU[14]

Overall, even though GPUs and SAs are often directly compared, they represent two complementary solution spaces. The former are designed to maximize flexibility and raw computational power, while the latter focus on efficiency and predictability. For this reason, the current state of the art shows an increasingly tight integration between these models. It is not uncommon to find GPU-based systems that incorporate systolic elements, as well as SAs that inherit configuration features typical of general-purpose GPUs.

SA based architecture - Google TPU v1-v3

The first generation of TPU developed by Google represents one of the most significant cases in which a systolic array has been employed on a large scale in an industrial product. In the official technical blog[13], Google describes the TPU v1 as an accelerator built “around a systolic array” (Figure 2.8) and specifies that the computational core of the chip is a 256 $\times$ 256 matrix of 8-bit MAC units, for a total of 65,536 multiply–accumulate operations per cycle. The array operates at a frequency of 700 MHz, achieving a theoretical throughput of approximately 92 Tera-MAC/s (value computed as 65,536 x 700 MHz).

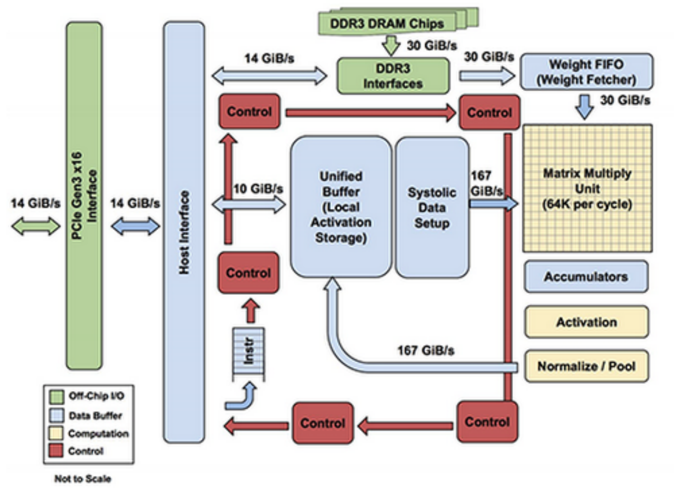

Figure 2.8: Internal MXU structure - Google TPU v1[13].

The chip, fabricated in 28 nm CMOS, consumes approximately 40 W, placing it in a very favorable energy-efficiency regime compared to contemporary CPU and GPU solutions. According to data published by Google, TPU v1 achieved improvements from 15× to 30× in absolute performance and from 30× to 80× in performance per watt compared to the general-purpose processors of the time (Figure 2.9).

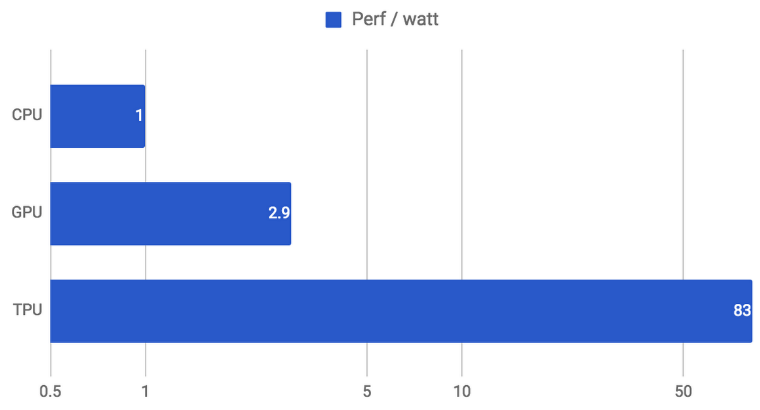

Figure 2.9: Performance comparison between TPU v1, CPU and GPU on different workloads[13].

Dataflow within the chip is an important aspect of understanding the systolic array's role. As pointed out by Google, once the weights are loaded into the MXU matrix, they are reused locally during the whole duration of the operation without any further access to external memory. This fully aligns with the systolic paradigm:

computation progresses by "streaming" operands through adjacent nodes to reduce heavy data movement and minimize dependence on off-chip bandwidth. The fact that the TPU uses a PCIe Gen3 ×16 link providing only 12.5 GB/s bandwidth also shows that the architecture is designed under the assumption of high internal data reuse.

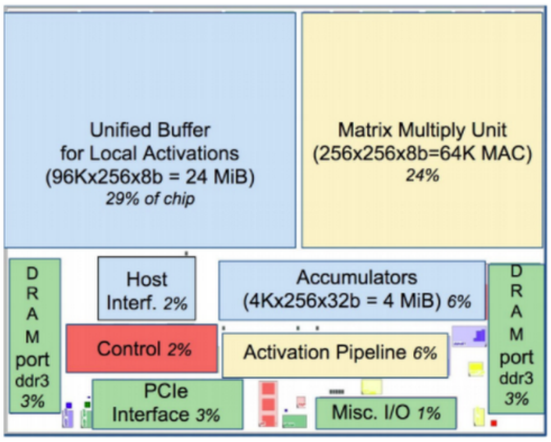

Figure 2.10: Google TPU v1 internal floorplan[13].

Another feature that characterizes the TPU v1 is the simplicity of its control logic. According to their datas, the internal control circuitry occupies less than 2% of the die area (Figure 2.10), with most of the silicon dedicated to pure computation, and is designed to support an extremely regular dataflow without sophisticated scheduling structures or parallelism-management mechanisms. That is one of the key differences with a GPU, in which much of the complexity arises from coordinating thousands of independent threads. The performance results validate the approach: for real latency, Google reports speedups up to 71× over reference CPUs on convolutional workloads (CNN1), besides more predictable temporal behavior. The systolic structure enables not only high throughput but also more predictable execution times, a key feature for distributed inference systems and large-scale cloud services.

Tesla FSD (Full Self-Driving) Chip

The Tesla FSD (Full Self-Driving) Chip introduced with the "HW 3.0" platform represents one of the most important industrial implementations of an architecture specifically devoted to real-time neural network execution. The device is designed specifically to meet automotive constraints that require predictable latencies, a low power consumption, and a highly regular computation flow. Inside the chip, there are two neural processing units (NPUs), each integrating a 96 × 96 array of MAC units optimized for 8-bit operations with 32-bit accumulation. According to the technical analysis reported by WikiChip Fuse[15], this structure allows each NPU to reach a theoretical throughput of about 36 TOPS.

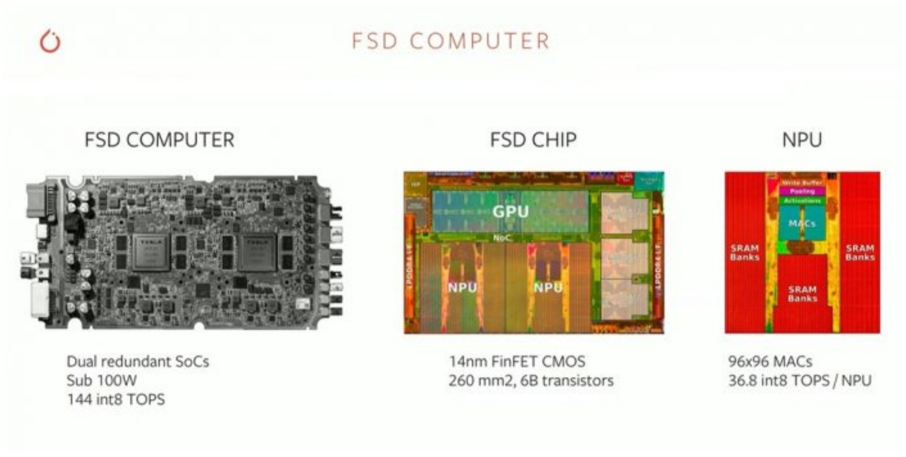

Figure 2.11: Tesla FSD chip[16].

A particularly relevant feature for comparison with systolic arrays is the way the 96 × 96 array is fed. In the FSD chip's NPUs, the 32 MiB local memory is subdivided into numerous independent banks and can sustain up to 256 bytes of activations and 128 bytes of weights per cycle. This corresponds to a maximum bandwidth of about 786 GB/s-a value that is necessary both to keep the utilization rate of the MAC array high and to reduce access to external DRAM[15]. This physical proximity between the array and the local buffer is a fundamental architectural element and reflects one of the main features of systolic arrays: reduction of the "distance of data movement," which is essential for keeping energy consumption under control.

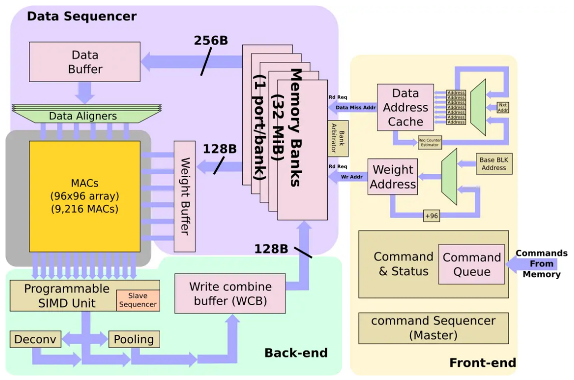

Figure 2.12: Tesla FSD chip internal floorplan[13].

From an internal organization point of view, the dataflow of Tesla's array is not formally described as a classical systolic pipeline; it exhibits similar characteristics since data is delivered to the elements of the array according to regular patterns by using horizontal and vertical broadcast mechanisms and propagation paths that maximize operand reuse[15]. Due to this regularity of data movement, together with the grid topology, Tesla's NPU gets closer to the behavior of systolic arrays, although there are some operational differences driven by the necessity of supporting highly heterogeneous network models and real-time tasks.

It can also be perceived that the FSD chip is part of a much larger system including a general-purpose processor, video controllers, and high-speed interfaces. These NPUs are utilized as dedicated engines for the heavy steps of inference using a pipeline devised to ensure consistent latency. The architectural approach here shows that, even in application domains quite different from training and datacenter-scale workloads, the use of regular and predictable arrays remains an effective strategy for high throughput and deterministic behavior. The Tesla FSD Chip is an interesting example of such an industrial accelerator because it adopts a large MAC array, a substantial local memory hierarchy, and regular dataflows-elements that clearly put it into the same conceptual family as systolic arrays, even if it does not follow the classical formulation exactly.

Where this thesis takes place?

As it clearly comes out from the survey of the analyzed architectures, regular structures for accelerating neural-network workloads represent an established choice both in industrial and academic domains. Solutions such as Google's TPU v1, Tesla FSD Chip and other similar projects share a computation strongly organized around MAC units arranged in regular grids, fed by local memory hierarchies optimized to maximize data reuse. These examples clearly illustrate the effectiveness of the systolic — or systolic-like — approach when the workload is dominated by dense and repetitive operations.

At the same time, these architectures share a common trait: while being extremely effective in a specific domain of application, the systolic array shape and dataflow remain essentially fixed. Solutions presented here tend to prefer regular structures deeply optimized for specific use scenarios, leaving little room for dynamically changing the array structure, data propagation mode, or PEs internal organization. This design choice — totally coherent with an industrial goal of predictability and maximum efficiency — on the other hand reduces their possibility to adapt to models that have various geometric requirements or to kernels that present less regular structures and dependencies.

It is in this framework that the present work finds its place, investigating the possibility of designing reconfigurable systolic arrays capable of changing their shape or data flow in order to adapt to more types of operations and a wider range of layers. It is not intended to supplant the existing architectures but to propose an alternative that brings more flexibility within the same paradigm. This research takes a complementary approach with respect to what has been done so far: identifying reconfigurability as a potential answer to increasing model heterogeneity in deep learning while preserving the advantages of regularity, predictability, and locality that made systolic arrays one of the most robust and efficient solutions for neural computation.

Chapter 3

Efficiency of Conventional SAs in executing NNs

In this chapter, the main transformations used to execute DNN layers on systolic architectures will be introduced. This is followed by an overview of the metrics adopted for analyzing, in terms of efficiency, the non-reconfigurable architectures currently present in the state of the art. The discussion examines how the array shape, dataflow, and data reuse strategies influence throughput and energy consumption across different types of layers. The goal is to quantitatively assess the limitations of fixed-shape solutions and to motivate the need for more flexible approaches.

3.1 Mapping Neural Networks to Systolic Arrays

The properties that make hardware acceleration of neural networks through Systolic Arrays (SAs) extremely efficient is the ability to reduce, through appropriate transformations, different types of layers to a limited set of computational primitives.

The most widely used of these is the transformation of layers into dense matrix multiplications, or GEMM (General Matrix-Matrix Multiplication). A large portion of the computation in a Deep Neural Network (DNN) can be expressed as a matrix product of the form ($C = A \times B$), where each element of the product is a matrix — properly derived and conditioned — from the input tensors.

These types of transformations are particularly advantageous in the context of SAs because, by their very operating principle, they are naturally optimized for GEMM-type operations, which consist of a series of regular, pipelined, and highly parallelizable steps. For this reason, most software frameworks and the majority of hardware accelerators (such as TPUs and GPUs) adopt a common strategy: they "map" high-level layers (FC, CNN, Attention) onto one or more GEMM operations.

In the following sections, the main mapping techniques will be analyzed depending on the type of layer being considered. The layers can be classified as follows:

- **Fully-Connected (FC) layers**: have a 1:1 mapping with GEMM, meaning they can be directly expressed as a matrix multiplication;
- **Convolutional layers (CNNs)**: require transformations such as `im2col` or its variants;
- **Attention blocks in Transformers**: are composed by Q/K/V projections and attention scores. Those computations can also be reduced to matrix-matrix multiplications.

3.1.1 Fully-Connected Layers

Fully-connected (or dense) layers represent the simplest and most direct linear operator within deep neural networks. Despite their apparent simplicity, they play a central role in numerous models, such as multilayer perceptrons, final classifiers in CNNs, and — even more significantly — the MLP blocks in Transformers. From a computational standpoint, FC layers are particularly well-suited for execution on systolic architectures, since their structure perfectly matches the form of a dense matrix multiplication (GEMM).

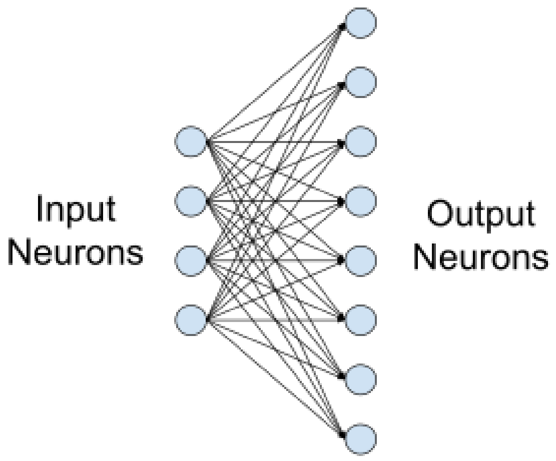

Figure 3.1: Example of a small fully-connected layer with four input and eight output neurons[17].

They are called "fully connected" because every input neuron can influence every output neuron. In the context of FC networks, all three phases of network training can be rewritten as GEMM operations.

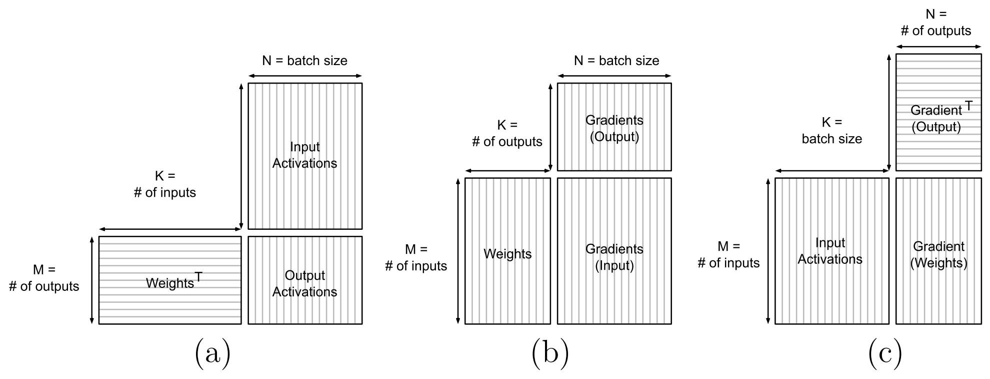

Figure 3.2: Dimensions of equivalent GEMMs for (a) forward propagation, (b) activation gradient, and (c) weight gradient computations of a fully-connected layer[17].

Mathematical Formulation of the Fully Connected Layer

A fully-connected layer performs a linear transformation:

$$y = Wx + b \tag{3.1}$$

where:

- $(x \in \mathbb{R}^{d_{\text{in}}})$ is the input vector,
- $(y \in \mathbb{R}^{d_{\text{out}}})$ is the output vector,
- $(W \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}})$ is the weight matrix,
- $(b \in \mathbb{R}^{d_{\text{out}}})$ is the bias vector.

When processing multiple samples in parallel (batch size (B)), the inputs are organized as:

$$X \in \mathbb{R}^{d_{\text{in}} \times B} \tag{3.2}$$

and the layer becomes:

$$Y = WX + b1_B^\top \tag{3.3}$$

where $b1_B^\top$ is simply a matrix composed by the vector B repeated b_{out} times. It's the mathematical way to say that the same bias is applied to the entire computation. The dominant operation, in previous equation, is the matrix multiplication:

$$Y = WX \tag{3.4}$$

which corresponds to a GEMM with dimensions:

$$M = d_{\text{out}}, \quad K = d_{\text{in}}, \quad N = B \tag{3.5}$$

Computational Complexity

The total number of operations (MACs) is:

$$\text{MACs} = d_{\text{out}} \cdot d_{\text{in}} \cdot B \tag{3.6}$$

This makes FC layers extremely costly in models such as Transformers, where MLP blocks typically cover 40–60% of the total computational cost. Also in Vision-based models, with very large fully-connected layers, using multiple batches makes this value increases rapidly.

Data Reuse Patterns

The fully-connected layer represents the most favorable case for systolic arrays in terms of data reuse. In fact:

- **Weight reuse**: each row of (W) is reused for all input vectors in the batch and also with large batch sizes, each weight is reused dozens or even hundreds of times (optimal for WS dataflows)
- **Input reuse**: each column of (X) is reused for all rows of (W). This makes Input-Stationary (IS) dataflow appealing.
- **Output Local Accumulation**: each element $(Y_{(i,j)})$ is obtained through accumulation which can be performed entirely within each PE. This behaviour is ideal for Output-Stationary (OS) dataflow.

In conclusion, all three dataflows are reasonably applicable to FC layers. However, the optimal choice depends on the shape of the SA and the batch size.

3.1.2 Convolutions (CNN) - GEMM via im2col

Two-dimensional convolutions represent one of the most fundamental operators in Convolutional Neural Networks (CNNs). Although the convolution operation is not, in its native form, a matrix multiplication, it can be transformed into a GEMM through a reorganization of the input tensors known as im2col ("image-to-column"). This transformation is widely adopted in major frameworks (such as cuDNN, PyTorch, and TensorFlow) and in dedicated hardware accelerators, as it enables the use of highly optimized GEMM implementations.

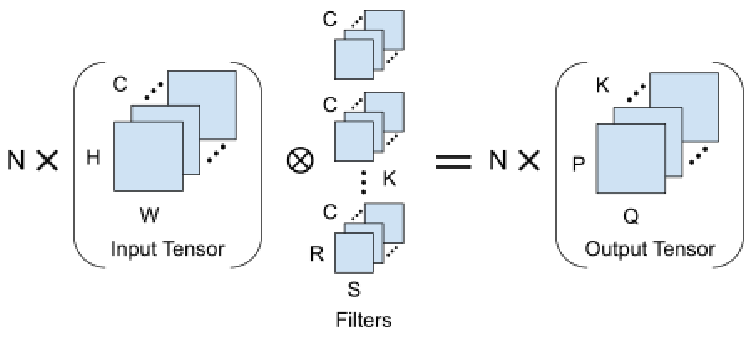

Figure 3.3: Convolution of an NCHW input tensor with a KCRS weight tensor, producing a NKPQ output[17].

A 2D convolution is a mathematical operation where a smaller matrix called the "filter" or "kernel" is slid over an input matrix (often an image) to extract meaningful features. The process of convolution can be described step by step as follows:

1. The kernel is placed over a specific region of the input matrix.
2. The two matrices are multiplied element-wise.
3. The resulting products are summed to compute the output value for the current position (pixel).
4. The filter is then shifted to the next position.

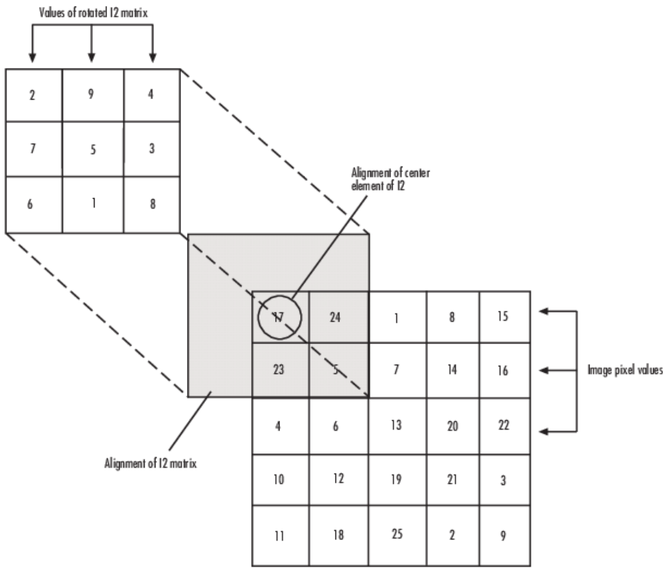

Figure 3.4: 2D discrete convolution of two input matrices[18].

This process produces an output matrix that captures certain patterns depending on the filter used, such as edges, textures, or shapes. In a 2D convolution, two parameters are very important: stride and padding. Stride determines how far the filter moves at every step, in terms of coordinates. Different DNNs use different stride values depending on the desired behavior for example to control the spatial resolution of the output.

Padding is necessary when the filter size is not an integer submultiple of the input matrix. In those cases, the filter may exceed the matrix boundaries during the edge iterations inducing computational errors. A common solution is to add rows and/or columns of zeros around the input matrix such that the filter remains within valid bounds ensuring that the original data are not distorted.

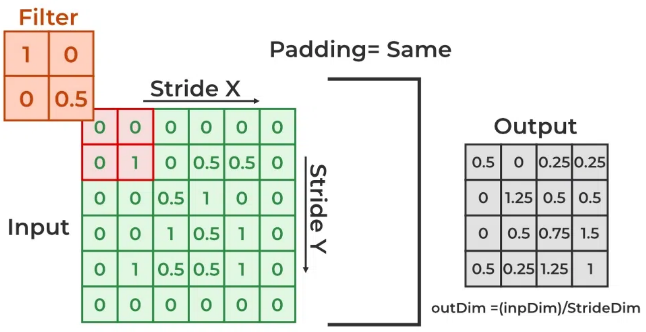

Figure 3.5: Stride and Padding in 2D-Convolution[19].

In the following mathematical discussion, stride will be fixed at 1 for simplicity.

Filter size is an integer submultiple of input so padding will be not required.

2D Convolution Mathemathical Fomulation

Start considering a basic 2D convolution between a three-dimensional input tensor:

$$X \in \mathbb{R}^{C_{\text{in}} \times H \times W} \tag{3.7}$$

and a set of filters:

$$K \in \mathbb{R}^{C_{\text{out}} \times C_{\text{in}} \times R \times S} \tag{3.8}$$

The output of the convolution is given by:

$$Y_{(c_o,i,j)} = \sum_{c_i=0}^{C_{\text{in}}-1} \sum_{r=0}^{R-1} \sum_{s=0}^{S-1} K_{(c_o,c_i,r,s)} \cdot X_{(c_i,,i+r,,j+s)} \tag{3.9}$$

The coordinates ((i, j)) span all spatial positions where the kernel can be applied, for a total of:

$$H_{\text{out}} = H - R + 1, \quad W_{\text{out}} = W - S + 1 \tag{3.10}$$

Thus, the final output has dimensions:

$$Y \in \mathbb{R}^{C_{\text{out}} \times H_{\text{out}} \times W_{\text{out}}} \tag{3.11}$$

This operation involves a triple nested loop over channels and spatial coordinates — which is inefficient to compute directly in hardware.

The Idea Behind im2col

The goal of im2col is to convert the convolution into a single large matrix–matrix multiplication. Each application of the $(R \times S)$ kernel over a region of the input can be viewed as a dot product between a vector containing the kernel values, and a vector obtained by "flattening" the corresponding image patch. Each filter is flattened into a vector of length:

$$K_{\text{flat}} = C_{\text{in}} \cdot R \cdot S \tag{3.12}$$

By stacking all filters as rows, the weight matrix of the GEMM is obtained:

$$K_{\text{mat}} \in \mathbb{R}^{C_{\text{out}} \times (C_{\text{in}} RS)} \tag{3.13}$$

For each valid position of the input tensor ((i, j)) the corresponding patch is extracted and then is flattened into a vector of length $(C_{\text{in}} RS)$. Each of these vectors is then appended as a column, forming:

$$X_{\text{col}} \in \mathbb{R}^{(C_{\text{in}} RS) \times (H_{\text{out}} W_{\text{out}})}. \tag{3.14}$$

Once these two matrices are constructed, the convolution operator becomes the simply GEMM that is previously analyzed in Fully Connected Layers (subsection 3.1.1):

$$Y_{\text{mat}} = K_{\text{mat}} \cdot X_{\text{col}} \tag{3.15}$$

where:

$$Y_{\text{mat}} \in \mathbb{R}^{C_{\text{out}} \times (H_{\text{out}} W_{\text{out}})} \tag{3.16}$$

which can then be reshaped back into its three-dimensional form:

$$Y \in \mathbb{R}^{C_{\text{out}} \times H_{\text{out}} \times W_{\text{out}}} \tag{3.17}$$

In summary: the input image is a tensor, but to make it suitable for processing within a Systolic Array, it must be converted into a matrix. To do so, a “patch” (or small region) of the tensor is extracted and flattened along its third dimension, producing a vector. By placing all such vectors side by side, the GEMM input matrix is obtained. Similarly, each filter is also flattened into a vector, forming the weight matrix. The resulting matrix multiplication is thus equivalent to evaluating — in parallel — the dot product of every filter with every subregion of the image.

3.1.3 Transformer and Self-Attention

The self-attention mechanism is the core element of Transformer architectures and represents one of the most computationally significant kernels in modern deep learning. Transformers currently define the state of the art in the domain of Large Language Models (LLMs). A full Transformer model consists of an encoder and a decoder, both implemented as deep neural networks with multiple layers of attention and feed-forward components.

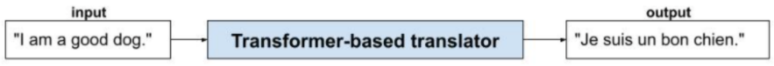

Figure 3.6: A Transformer-based application that translates from English to French[20].

As described by Google[20]:
“For example, in a translator:

- The encoder processes the input text (for example, an English sentence) into some intermediate representation.
- The decoder converts that intermediate representation into output text (for example, the equivalent French sentence).

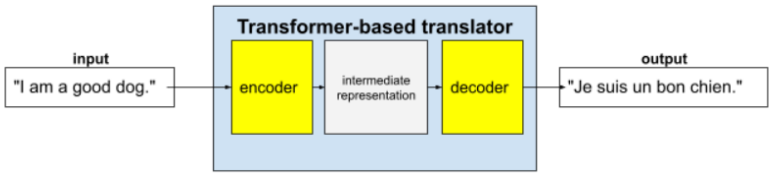

Figure 3.7: A full Transformer contains both an encoder and a decoder[20].

The 'self' in 'self-attention' refers to the input sequence. Some attention mechanisms weigh relations of input tokens to tokens in an output sequence, like in translation, or to tokens in some other sequence. But self-attention only weighs the importance of relations between tokens in the input sequence."

Although the behavior of the attention mechanism is more complex than that of a fully connected or convolutional layer, from a mathematical and computational standpoint it can be largely expressed as a sequence of dense matrix multiplications. This makes it particularly well-suited for execution on systolic array architectures, which are optimized for regular, parallel GEMM operations. In this section, the main operations involved in the Attention block are analyzed, with a particular focus on data reuse patterns and computational implications.

Mathematical Formulation of Attention Block in Transformer Layer

Given an input tensor:

$$X \in \mathbb{R}^{T \times d_{\text{model}}} \tag{3.18}$$

where (T) represents the sequence length and (d_{model}) the model dimensionality, the multi-head attention mechanism constructs three linear projections — query, key, and value — as follows:

$$Q = XW_Q, \quad K = XW_K, \quad V = XW_V \tag{3.19}$$

with:

$$W_Q, W_K, W_V \in \mathbb{R}^{d_{\text{model}} \times d_{\text{head}}}, \quad d_{\text{head}} = \frac{d_{\text{model}}}{h} \tag{3.20}$$

where (h) is the number of heads, meaning the number of different subspaces (or "perspectives") the model attends to simultaneously when making a decision.

Each transformation is a GEMM of the form:

$$(T \times d_{\text{model}})(d_{\text{model}} \times d_{\text{head}}) \tag{3.21}$$

In this formulation are evident that high data reuse can be exploited. The same inputs (X) are used three times and each weight matrix (W) is reused (T) times, once for each token in the sequence. Thus, the construction of (Q), (K), and (V) is computationally very similar to a fully-connected layer.

Another step that requires GEMM calculation is the computation of Attention Scores: ($S = QK^\top$). The core of the attention mechanism is the compatibility matrix:

$$S = QK^\top, \quad S \in \mathbb{R}^{T \times T} \tag{3.22}$$

Each element is defined as:

$$S_{(i,j)} = \langle Q_i, K_j \rangle \tag{3.23}$$

which measures how relevant token (j) is to token (i). From a computational point of view the product ($QK^\top$) is a square GEMM of dimension:

$$(T \times d_{\text{head}})(d_{\text{head}} \times T) \tag{3.24}$$

Also in this case the algorithm has an high data reuse, in fact, each row of (Q) and (K) are reused (T) times. Hence, ($QK^\top$) represents an almost ideal case for a Systolic Array since dense tensors with highly regular computation pattern are organized in square matrix structure (high parallelism potential) that exploit extensive data reuse along both rows and columns. Last step is the computation of the Weighted Output ($O = \tilde{S}V$). The final output of the attention mechanism, for a single head, is:

$$O = \tilde{S}V, \quad O \in \mathbb{R}^{T \times d_{\text{head}}} \tag{3.25}$$

where ($\tilde{S}$) is the normalized attention score matrix (after applying softmax). Also in this case is a rectangular GEMM:

$$(T \times T)(T \times d_{\text{head}}) \tag{3.26}$$

The high and regular reuse pattern makes this computation very suitable for systolic array architectures. In this case each row of ($\tilde{S}$) is reused (d_{head}) times and represents a probability distribution. Instead, each row of (V) is reused (T) times. In conclusion, the entire Attention block (for a single head) is dominated by five GEMM operations:

1. Three GEMMs to produce (Q), (K), and (V).
2. One square GEMM for ($QK^\top$).
3. One rectangular GEMM for ($\tilde{S}V$).

Each of these operations maps efficiently onto systolic array architectures, highlighting why Transformers — despite their apparent algorithmic complexity — can be effectively accelerated through GEMM-optimized hardware.

3.2 Evaluation Methodology

To evaluate the efficiency of the different systolic array configurations, a two-phase methodology was adopted.

First, an RTL-level analysis was performed to estimate the number of clock cycles required to fully execute specific DNN kernels.

Next, the corresponding architecture was synthesized using a 28 nm technology node, which was employed to estimate area and power consumption at a fixed operating frequency of 500 MHz. This frequency value is widely adopted in the literature related to DNN accelerators implemented in this or comparable technology nodes. Works such as MAGNet by NVIDIA[21], NullHop[22], and SHARP[23] use target frequencies in the range of 400–600 MHz for post-synthesis evaluations. Operating within this frequency range allows for reliable estimations of area and power while ensuring a fair comparison among the different configurations under analysis. It remains true, however, that the architecture — as well as the synthesis scripts — is fully modular. As a result, obtaining the same results at a different operating frequency becomes extremely straightforward.

3.2.1 Performance estimation method

For each array configuration, the total number of clock cycles, the size of the local buffers, and the bandwidth required by the array to execute a given kernel were estimated. The goal was to obtain a latency model that did not depend on memory-access constraints but focused exclusively on the intrinsic properties of the architecture. The analysis is based on a Python function implementing an analytical model of the array's behavior, which, unlike full RTL simulation, allows for rapid exploration of the configuration space and helps narrow down the number of candidates in a first approximation. The results obtained for the proposed architectures were compared against RTL simulation runs to verify the correctness of the analytical estimates.

Bandwidth estimantion

The first step consists of estimating the local resources and the required bandwidth. For each dimension of the array — identified as D1, D0, and S — all parameters necessary for sizing the local memory are determined, such as the amount of data to store, the number of access ports, and the theoretical bandwidth required to feed the PEs. This phase quantifies the communication overhead, which is useful for assessing the integrability of the array in memory-constrained systems, but it does not affect the latency computation, which remains idealized.

Wavefront-Steps Definition

A characteristic aspect of systolic architectures is the initial latency needed for the data stream to propagate through the entire array. This latency depends on the dimensions of the array and is modeled as:

```
wavefront_steps = arr_shape_D1 + arr_shape_D0 - 1
```

This value corresponds to the number of cycles required for the wavefront to reach the last PE in the array. During this interval, the PEs do not yet operate at full throughput, but this phase is essential to obtain a fully filled pipeline.

Per-Tile latency and total number of tiles

DNN kernels are divided into submatrices, or "tiles", defined by:

- number of submatrices along D1 (`N_tiles_D1`)
- number of submatrices along D0 (`N_tiles_D0`)
- number of submatrices along dimension S (`N_tiles_S`)

From that is possible to derive the total number of tiles such:

```
N_tiles = N_tiles_D1 * N_tiles_D0 * N_tiles_S
```

Each tile requires a number of cycles equal to:

```
steps_per_tile = arr_shape_S
```

since the accumulation along this dimension determines the number of steps needed to complete the partial computation associated with the tile.

Computation Latency Model

The total number of clock cycles is therefore given by:

```
tot_CC = (steps_per_tile * N_tiles) + wavefront_steps - 1
```

The first term represents the effective computation time of the tiles, while the second accounts for the delay introduced by filling the wavefront and draining the pipeline at the end of the computation. This formula — derived directly from the synchronization model of the array — matches, within the granularity of the ideal analytical model, the behavior observed in RTL simulations.

Mapping Function - Fully Connected

To compare the performance of different SA configurations, each layer of every neural network must be expressed in a form compatible with the computational model of the array. As discussed earlier, modern DNNs are far from being simple GEMMs, but with the appropriate transformations the problem can be reduced to a form that systolic arrays can efficiently handle. To uniformly evaluate latency and throughput across configurations, it is therefore necessary to have a function that maps each layer to a standard three-dimensional domain ((D1, D0, S)), representing the cube of MAC iterations required to complete its computation. Given the shapes of tensors a layer needs to compute, a python function was built to return the boundaries of the for-loops of the c-code that would perform the computation of the layer, reordering the boundaries based on the dataflow under analysis (i.e., Output-Stationary, Weight-Stationary, or Input-Stationary).

In the case of a fully-connected layer, the computation is simply the multiplication between an activation matrix and a weight matrix. The function used to complete this task only assigns the three dimensions (D1), (D0), and (S) to the relevant quantities of the layer depending on the dataflow:

- Output Stationary (OS): Output elements remain resident in the PEs.

  ```
  D1 = out_rows, D0 = out_cols, S = input_cols
  ```

- Weight Stationary (WS): Weights remain fixed in the PEs, inputs flow through the array.

  ```
  D1 = weight_rows, D0 = weight_cols, S = input_rows
  ```

- Input Stationary (IS): Activations remain fixed, weights propagate across the array.

  ```
  D1 = input_cols, D0 = input_rows, S = weight_rows
  ```

This conversion enables each fully-connected layer to be modeled as a uniform three-dimensional domain, directly compatible with the systolic array latency model.

Mapping Function - CNN

Convolutional layers must also be mapped into the same domain. The convolution is interpreted as a matrix–matrix product by flattening both patch tensors and filter tensors. For the CNN case, another function is needed to transform the layer. In brief, the difference between the FC and the CNN case is one step in which the im2col algorithm previously presented is applied. This function builds the triplet

((D1, D0, S)) by relating the number of spatial output positions, the number of output channels and the internal kernel dimension (input channels × filter area) following the organization of the data vector imposed by the chosen dataflow. In particular:

- **Output Stationary** (OS):

  ```
  D1 = N * H_out* W_out, D0 = C_out, S = C_in * K_h * K_w
  ```

- **Weight Stationary** (WS):

  ```
  D1 = C_out, D0 = C_in * K_h * K_w, S = N * H_out * W_out
  ```

- **Input Stationary** (IS):

  ```
  D1 = C_in * K_h * K_w, D0 = N * H_out * W_out, S = C_out
  ```

This recoding makes it possible to process convolutional layers using exactly the same tiling mechanism as other operators. Moreover, since everything has been reduced to a matrix multiplication, the same analytical latency model can be used for performance estimation.

Definition of Efficiency Metric

Identify the boundaries of the for-loops both for FC and CNN layers is an essential step to also figure out the number of MAC operations. Moreover, in order to compare the different configurations of the systolic array in a homogeneous way, an efficiency metric was adopted that combines computational capability and power consumption into a single value.

The metric Operations Per Cycle (OPC) is defined as the ratio between the total number of MAC operations actually executed and the overall number of clock cycles required to complete them. To incorporate power consumption into this value, the result is then normalized with respect to the dissipated power. In other words, the throughput is first measured in terms of MAC per cycle, obtained by accumulating the total number of operations performed and the corresponding cycles used across the entire sub-layer under consideration. This value is then divided by the average power, expressed in milliwatts, obtained from the design synthesis. The result is a metric expressed as OPC/mW, which describes how much effective work the architecture can perform per unit of energy.

This metric, in addition to being easy to interpret, makes it possible to compare configurations with different array shapes, different dataflows, or different levels of parallelism, highlighting which solutions offer the best compromise between performance and power consumption.

3.3 Performance Results

In this paragraph, the performance of the various SA configurations considered during the design space exploration will be evaluated systematically. Since performance, in terms of efficiency, varies depending on the specific dataflow scheme and array shape, five models — each composed of two layers — were selected to cover operation types that are representative of modern deep neural networks.

For each layer, a plot will be presented summarizing the efficiency of the different configurations according to the OPC/mW metric, together with observations on latency, area, and dynamic power consumption. The aim is not to examine every collected value in detail, but to highlight the combinations of shapes and dataflows that prove to be the most efficient for the structure of the layer under analysis.

This approach made it possible to identify relationships between the type of layer being executed and the optimal architecture, which are crucial for selecting the most versatile configurations. These selected configurations will later be integrated into the reconfigurable architecture developed in the second part of the work.

Model	**Layer 0**	**Layer 1**	**Category**
GPT-2	Self-attention	Feed-forward	Transformer
ViT-L/16	Self-attention	Feed-forward	Transformer
ResNet-152	Bottleneck B2	Bottleneck B4	Deep CNN
TinyBERT	Self-attention	Feed-forward	Transformer
MobileNetV2	Inverted residual	Inverted residual	CNN

Table 3.1: Overview of the neural network models and corresponding layers considered in the performance evaluation.

TinyBERT

TinyBERT[24] is a compact version of BERT obtained through knowledge distillation, proposed by *Huawei* to significantly reduce the computational complexity of the original model while maintaining competitive performance on natural language tasks[24]. The architecture follows the Transformer paradigm[25], dividing each block into two main components. A self-attention module, consisting of the QKV projection operations, the QK product, the V projection, and the final output projection and a feed-forward network (FFN) module, composed of two linear transformations.

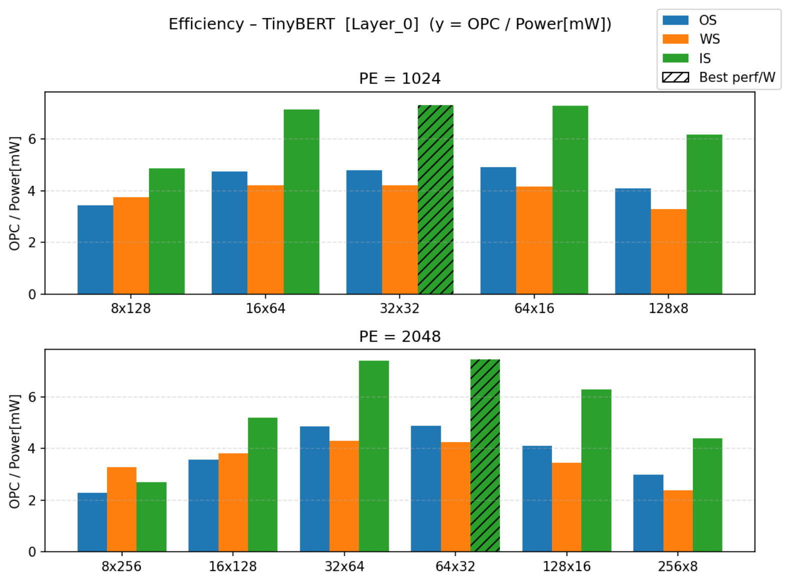

Figure 3.8: TinyBERT[24] - Self-attention Layer - results.

The first set of plots (Figure 3.8) analyzes the self-attention block across the selected systolic array configurations, combining multiple shapes with the three dataflows (OS, WS, IS). The batch size used in the tests is 4. TinyBERT operates on short computation sequences and with a reduced number of parameters compared to the standard versions of BERT, and for this reason the use of very large batch sizes is unnecessary. In these compact models, using a batch size of 4 represents a reasonable compromise between saturating the compute engine and maintaining a steady dataflow. Moreover, for architectural analysis, using a batch size that is not excessively large makes it possible to isolate more precisely the effects introduced by the structure of the array, without adding artificial improvements due to batching.

From a practical standpoint, using a batch size of 4 means that, during the inference phase, the model processes four sequences in parallel. In practice, instead of processing one sentence at a time, it groups four inputs and executes them simultaneously — an execution scenario that is not far from real-world usage, especially when considering large-scale deployments of such models. From the experimental results:

- OS and WS are competitive in certain configurations, but never dominant in this block.
- IS dataflow achieves, in most shapes and array sizes, the highest OPC/mW efficiency
- Generally the symmetric or near-symmetric shapes archive better efficiency score.

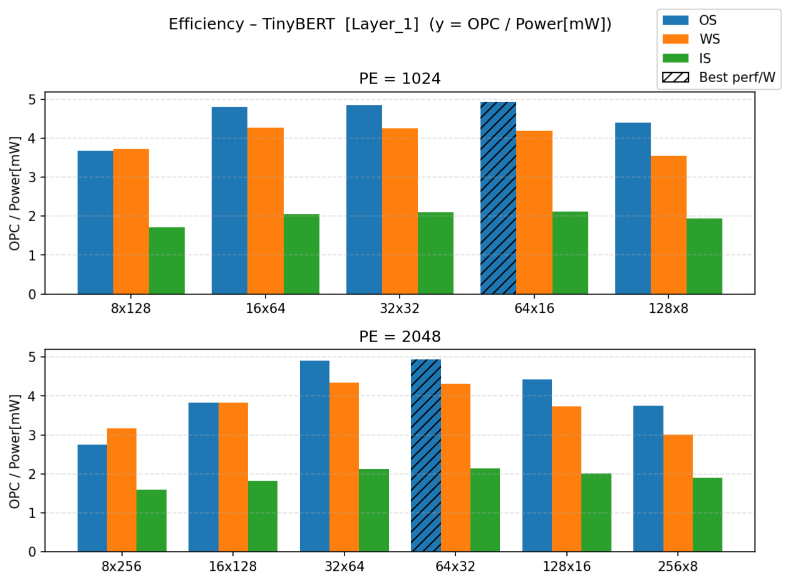

Figure 3.9: TinyBERT[24] - Feed-forward Layer - results.

In the second block (Figure 3.9), corresponding to the FFN, the behavior changes:

- OS and WS show performance comparable to - or even higher - than IS across nearly all shapes. WS is highly competitive in rectangular shapes, where the large internal FFN projection amplifies weight reuse.
- IS remains effective in some configurations, but does not reach the performance levels observed in the attention block.
- In this case, the square array, is not the overall best configuration.

This contrast between the two sub-blocks highlights how the same model exhibits heterogeneous computational behaviors, and how the interaction between dataflow and layer structure strongly influences the efficiency of the systolic array.

GPT-2

GPT-2[26] is an autoregressive Transformer model proposed by OpenAI[26], designed for text generation and based on decoder blocks. Each block is composed of two components:

- a masked self-attention module, use three linear projections Q, K, and V and their correlation (QK^T). Also, a fourth linear projection is applied to the attention output. This stage is dominated by matrix–matrix multiplications.
- a feed-forward (MLP) module, an high-dimensional linear transformation followed by a second projection. This module expands the internal dimensionality so there is a high weight reuse and very high computational intensity.

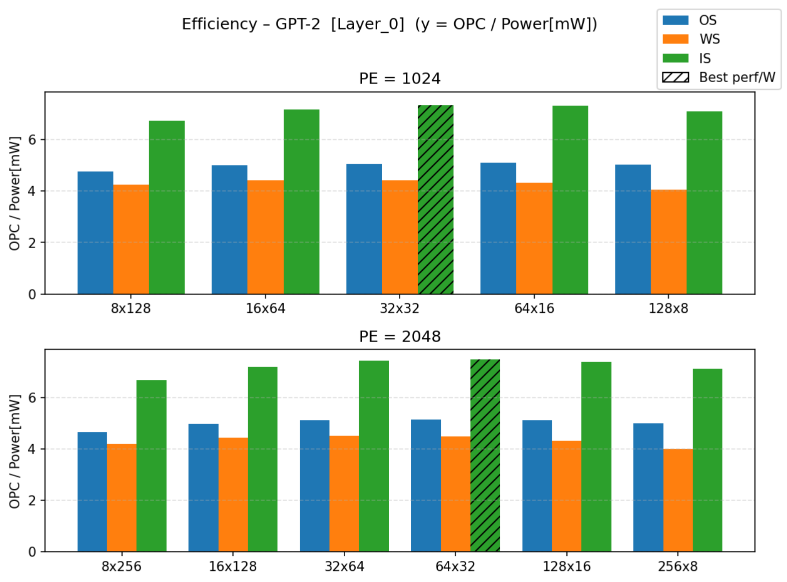

Figure 3.10: GPT-2[26] - Self-attention Layer - results.

In this case, as well, the first plot (Figure 3.10) analyzes the efficiency of the various combinations of shapes and dataflows for the self-attention block. The layer has to be considered as the sum of the QKV projections, the QK product, the V projection, and the final output projection. The batch size is set to 4, as in the TinyBERT analysis. GPT-2 is a model very similar to TinyBERT so the previous motivation about batching still remain valid. From the experimental evaluation, the following key observations emerge:

- IS is, in most configurations, the most efficient dataflow, particularly in the more symmetric shapes.
- OS shows competitive values across nearly all shapes but still remains noticeably below IS.
- WS is the least effective dataflow for the attention block.
- Symmetric and quasi-symmetric configurations are the ones with best efficiency.

As in the case of TinyBERT, the observed behavior aligns with the nature of attention operations: high reuse of inputs and very large output tensors, making the input-stationary approach particularly effective for computing this layer.

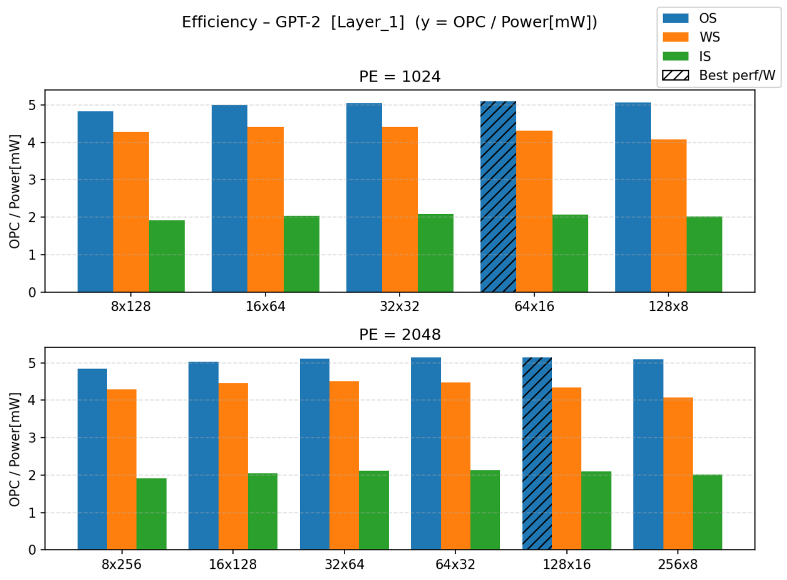

Figure 3.11: GPT-2[26] - Feed-forward Layer - results.

In the second block (FFN - Figure 3.11), consisting of the layers `GPT_2_FFN_L1` and `GPT_2_FFN_L2`, the behavior of the model changes noticeably. From the plots, the following trends emerge:

- OS becomes significantly more competitive than in the attention block, achieving high efficiency in various shapes — especially in the less symmetric ones.
- WS remains stable and shows good performance in symmetric or moderately unbalanced shapes, making it a valid alternative to OS.
- IS, while still functional, is generally less efficient than the other dataflows, with lower values in most shapes.

This behavior is driven by the different computational nature of the FFN compared to the attention block. The FFN features a very large internal dimension and produces a high number of outputs. In this context, keeping partial sums inside the PEs is advantageous, as it reduces the cost of writing intermediate results back to external memory.

MobileNet

MobileNet[27] family of networks was proposed with the aim of dramatically reducing the computational complexity compared to traditional convolutional models, while preserving competitive accuracy. The straightforward notion of this architecture relies on its systematic adoption of depthwise separable convolutions, which factor the convolution operation into two different steps, reducing the number of MACs needed by a wide margin. Thanks to this design choice, MobileNet established itself as one of the most efficient solutions for inference in resource-constrained platforms such as smartphones and embedded systems. With time, such a family evolved into a number of variants, such as MobileNetV2 and MobileNetV3, each targeting a better trade-off between accuracy, latency, and energy consumption, hence becoming a well-established reference for developing "edge-friendly" models. Unlike the previous cases, the batch size used for this model is equal to 1. MobileNet is a model designed for image processing in embedded systems; therefore, although parallel processing is a possible scenario, such cases have been excluded. By observing the graphs, one will immediately notice a rather drastic drop in performance compared to the previous cases. This behavior is justified by the fact that the array is excessively large relative to those on which MobileNet is intended to be executed, resulting in a significant underutilization of the PEs. However, for a fair comparison with the other cases, it was nonetheless decided to keep the same array.

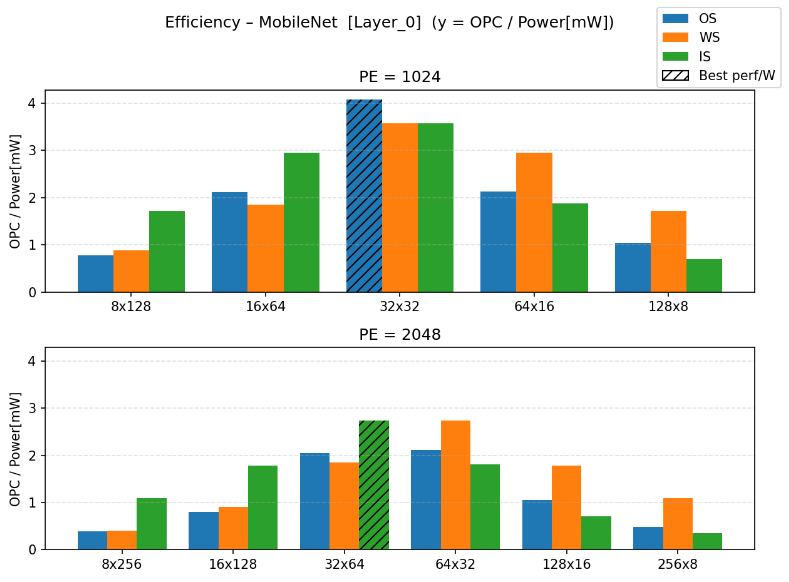

Figure 3.12: MobileNetV2[27] - Inverted residual L0 - results.

The trends in the two graphs (Figure 3.12) show a behavior that is quite consistent with a layer dominated by standard convolution with a 3×3 kernel and a relatively high number of input channels compared to the deeper layers. The literature for this kind of operation suggests that there is no single "universal" combination of shape and dataflow that is optimal, as the result strongly depends on the balance between horizontal/vertical parallelism and the amount of weight reuse that is achievable in the specific layer. From Figure 3.12 this trends emerge:

- OS shows an overall stable and competitive behavior. In particular, symmetric or quasi-symmetric configurations achieve the best values in terms of efficiency. This configuration performs well because it is able to exploit the spatial locality of the output which, in layer 0, is still quite significant. Highly unbalanced configurations, not fitting the input matrices optimally, are the worst.
- WS exhibits a more irregular, yet sometimes competitive, trend. Generally less efficient on the first layer since the relatively deep filters do not allow for a weight reuse sufficient to justify a greater transfer on the inputs. Interesting peaks appear in fairly unbalanced shapes when the alignment between kernel and weights improves.
- IS is the most variable dataflow among those analyzed but also the one that reaches the highest efficiency peak. In this case, it is even the quasi-symmetric configurations that achieve the best results. This dataflow is still able to exploit activation locality effectively. As the activations in layer 0 still have significant dimensions, and, once again, it suffers from highly unbalanced shapes with drastic efficiency drops in extreme configurations.

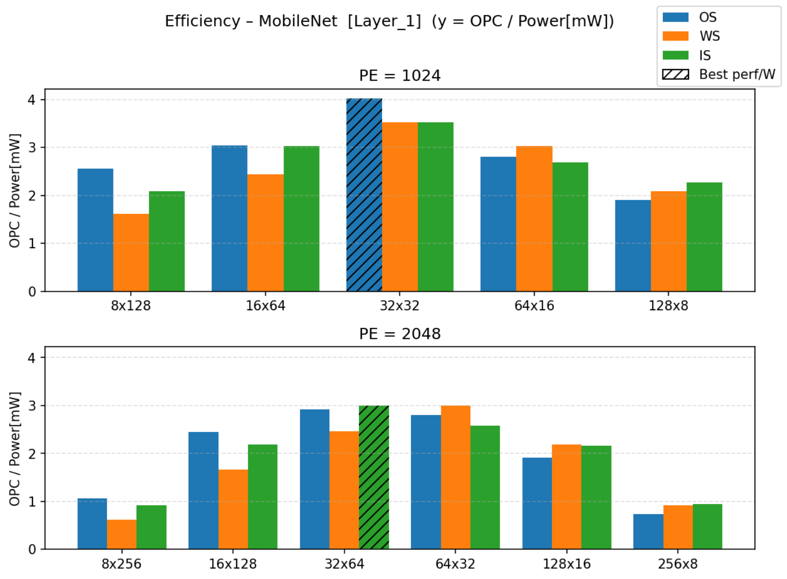

Figure 3.13: MobileNetV2[27] - Inverted residual L1 - results.

From the experimental data, in the Layer 1 of MobileNet (Figure 3.13):

- OS also shows good results, with the best values on square shapes. In this case as well, with the increase in the number of channels, output reuse proves to be the most advantageous. Very narrow shapes (8×128, 128×8) show a decrease in efficiency, indicating that the output flow struggles to saturate the array.

- WS improves compared to the previous case due to the increased number of channels, which leads to significantly more substantial weight reuse. WS becomes comparable to the OS case, even surpassing it in some configurations (e.g., 64×16 for 1024 PEs). It is never the absolute best case, but it remains stable in rectangular configurations.
- IS is once again the most performant dataflow overall, consistent with the high number of activations. It is the most sensitive to the shape of the array and also in the most unbalanced configurations, efficiency drops rapidly.

ResNet

The ResNet[28] family was introduced with the aim of overcoming some limitations of traditional deep networks, in particular the degradation phenomenon, whereby adding more layers does not improve performance and can even reduce it. The idea behind ResNet consists in the introduction of residual blocks in which information can bypass one or more convolutional transformations (Figure 3.14(, thanks to skip connections. This simple yet effective strategy facilitates gradient flow and makes it possible to train much deeper models without encountering instability. With time, ResNet has established itself as one of the reference architectures in computer vision thanks to its regular and easily scalable structure, and its ability to maintain a good balance between accuracy and computational cost.

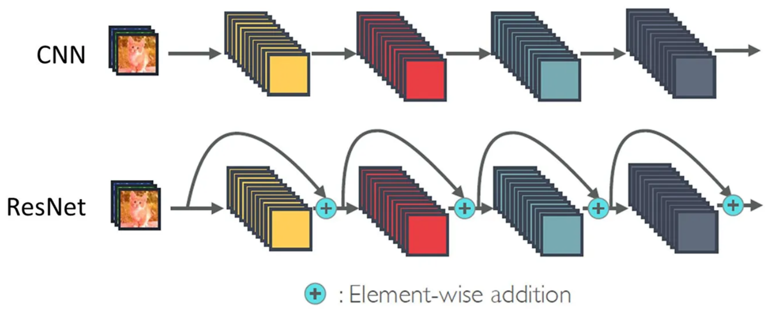

Figure 3.14: ResNet residual connection[29].

To do the experiments on ResNet, a batch size of 256 was chosen, which is especially suitable when working with a model much heavier than MobileNet. In practice, this means that 256 images are processed by the network in parallel on every computation step. Such a large batch size allows the hardware parallelism to be fully exploited and yields much more stable throughput measurements amortizing kernel activation time, since the variability in individual steps decreases. Moreover, ResNet is often used in high-volume training and inference scenarios where large batches are usually the norm and thus a realistic load for evaluating the behavior of such architecture.

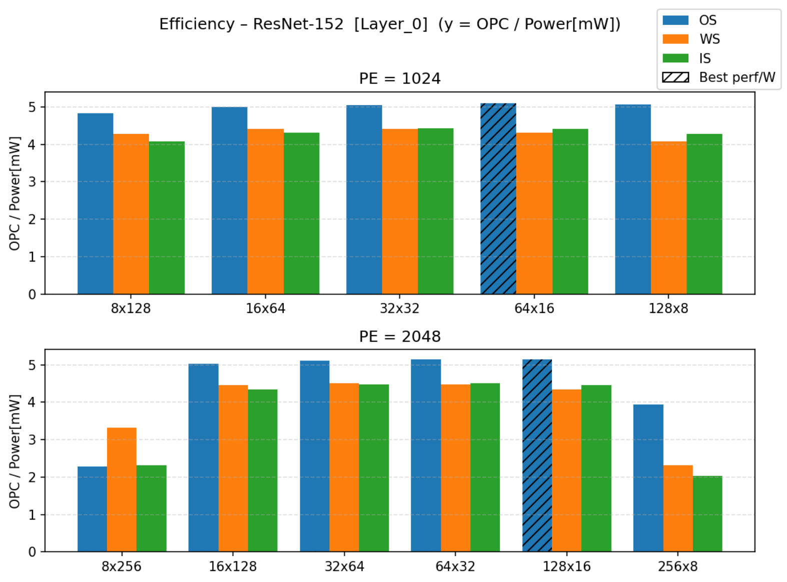

Figure 3.15: ResNet-152[28] - Bottleneck B2 - results.

The behavior of ResNet-152 (Figure 3.15) is extremely different from the trends observed so far, as the structure of the network's initial layer is completely different. ResNet implements a 7×7 convolution with stride 2, which is extremely dense and generates a very high reuse of both outputs and weights. As result, the analyzed arrays tend to remain constantly saturated, reducing the differences that were previously observed among the various shapes. The three types of dataflow also become more competitive, since the reuse of the 7×7 kernel masks part of the geometric inefficiency of the less symmetric shapes.

- OS is almost always the most efficient dataflow in this layer. The high number of accumulations, required by the 7×7 kernel, naturally favors output stationarity. The local accumulation of outputs makes it possible to minimize internal traffic and maximize PE utilization. In the not-symmetric configurations (like 64X16 and 128x16), OS reaches the highest values in the entire analysis and maintains solid performance even as the number of PEs increases.

- WS, in this layer, benefits from the very high weight reuse enabled by the kernel, which represents an ideal condition for WS. It still cannot match the performance of OS, but it remains competitive in regular shapes and maintains an efficiency level surprisingly close to the best case.

- IS is naturally penalized in this type of network. Activation reuse is less critical than the stationarity of accumulations and the reuse of weights, which is why it maintains low values across all configurations. It becomes comparable only on symmetric shapes.

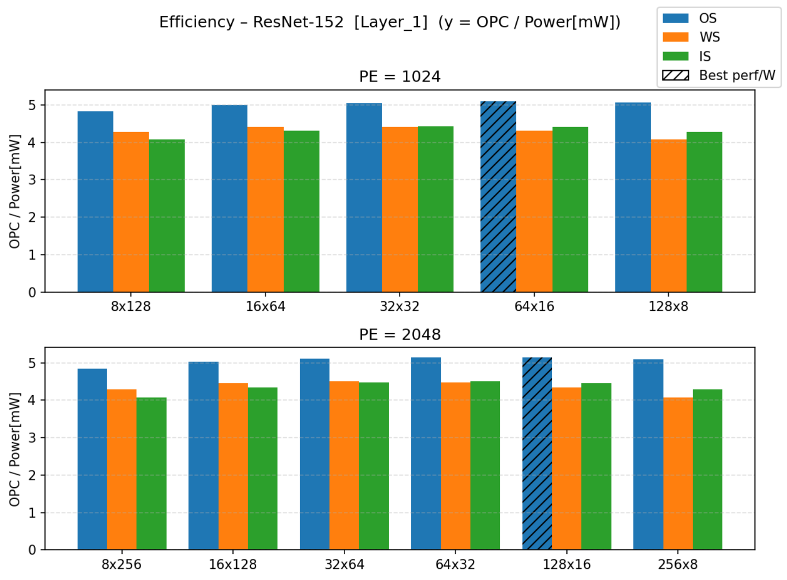

Figure 3.16: ResNet-152[28] - Bottleneck B4 - results.

Results in Layer 1 (Figure 3.16) of ResNet-152 appear very "flat" again, with reduced differences both for the various array shapes and across the dataflows. This behavior is fully consistent with the initial structure of ResNet: it follows the same pattern of Layer 0 with a high compute-dense operation (typically a 3×3 with many channels), which maintains high data reuse and allows the array to stay uniformly saturated. Thus, regular shapes like 32×32, 32×64, 64×32, or 128×16 yield very similar efficiencies, while even the more unbalanced shapes degrade less than what is observed in lighter architectures. Therefore, the trend is fully compatible with a highly computation-intensive layer.

- OS, exactly as in Layer 0, shows a clear advantage of the OS configuration over the others due to the extremely high number of accumulations caused by the initial convolution. For this reason, regular shapes such as 32×32, 32×64, 64×32, or 128×16 exhibit very similar efficiencies, and the performance degradation in more unbalanced structures is less severe.

- WS remains consistently stable, with values comparable to the other configurations yet never reaching the optimal performance of the OS case. It does not show significant drops as the array shape varies, again due to the substantial number of channels.

- IS loses a few points of efficiency compared to the other two cases while still maintaining high and steady performance. The array remains well saturated with good input reuse. The IS case tends to perform better on more symmetric shapes, losing some efficiency when shifting toward more extreme configurations.

ViT

ViT[30] represents an important change in computer vision because it applies the Transformer model - originally developed for language processing - to images. The idea is very simple: instead of using convolutions as in traditional CNNs, the image is divided into small regular patches that are treated as a sequence of "tokens" analogous to words in a text. Each patch is then transformed into a vector and enriched with positional information so that the model can preserve the spatial structure of the image.

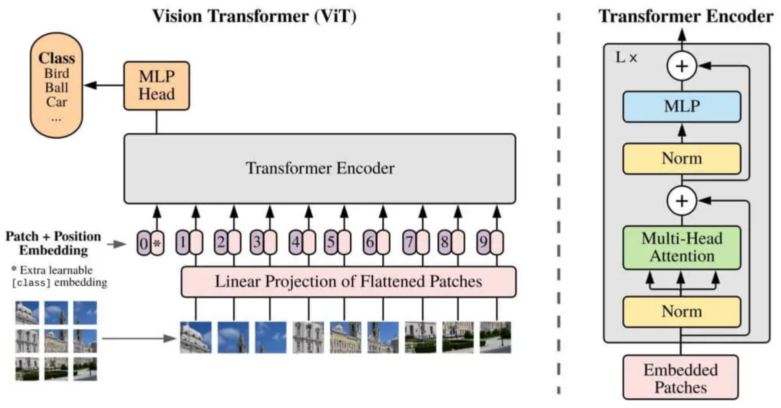

Figure 3.17: ViT transformer[31]

At this point, the self-attention mechanism comes into play, allowing the ViT to relate any patch to all the others, hence capturing global dependencies from the very first layers. This approach, reduces the typical constraints of CNNs - locality and fixed-size kernels - and enables the model to learn more complex structures, especially when it is trained on very large datasets. It is exactly this ability to reason over the whole image that has made the ViTs competitive or even superior to convolutional architectures in several tasks.

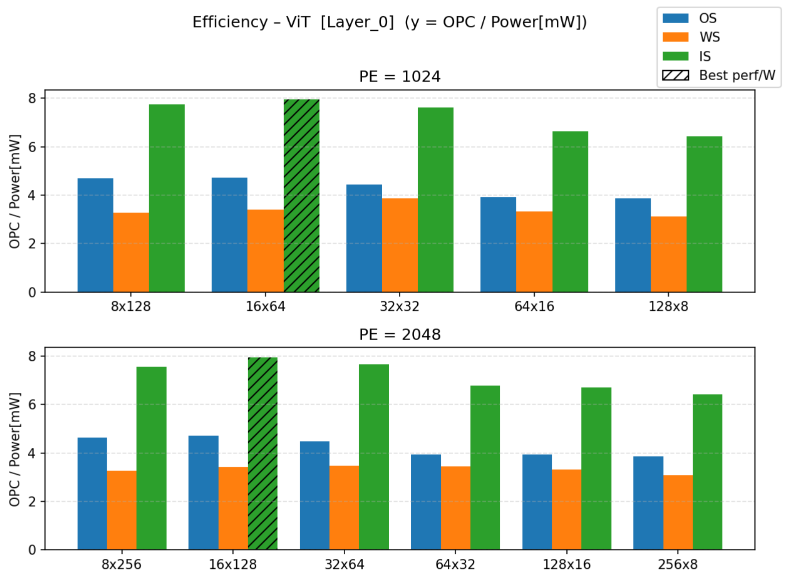

Figure 3.18: ViT[30] - Self-attention - results.

The behavior of Layer 0 (Figure 3.18) of the Vision Transformer is very different from what was observed in MobileNet and ResNet, but it resembles the trends seen in the case of GPT and TinyBERT. In the graph, in fact, a general dominance of the IS configuration emerges, with values almost twice as high as OS and far above WS. This result is consistent with the nature of the first layer in ViTs: the model performs a linear projection on the patches (patch embedding), an operation that works directly on the input without any significant reuse of weights or outputs, with a structure very similar to that of a sentence in LLM models. In this condition, what matters most is the reuse of activations, precisely what the Input Stationary dataflow favors. Consequently, both OS and WS struggle to reach comparable values because a model like the patch embedding is heavily biased toward being "input-driven."

- OS remains at mediocre values since the linear projection of the patches does not require numerous accumulation cycles as in convolutions. Therefore, output reuse is quite limited, although it stays fairly consistent as the array shape varies.
- WS is the most penalized dataflow in this layer. Even though the operation is technically a linear operation, the structure of the weights still cannot produce reuse that is advantageous enough to compensate for the losses caused by moving the inputs.
- IS is the best dataflow for this layer, achieving the highest performance in all configurations regardless of the number of PEs or the shape of the array.

This behavior is due to how ViT is designed: patch embedding consists of a linear transformation that makes intensive use of the inputs and reuses them many times before producing the final token. This mechanism maps optimally onto IS, which keeps the activations "stationary" within the PEs. Differently from all the other layers, the most interesting configurations are the highly asymmetric ones.

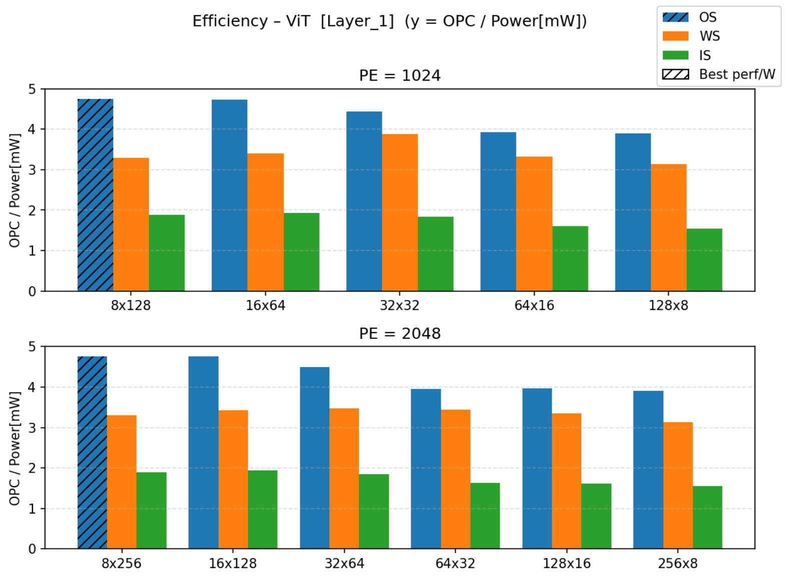

Figure 3.19: ViT[30] - Feed-forward - results.

Concerning Layer 1 of the ViT (Figure 3.19), the data deviate considerably from the patch embedding (Layer 0 - Figure 3.18). The ViT, in fact, shows the same behavior, also in this case, as the FFN model of GPT and TinyBERT, displaying a generalized dominance of the OS dataflow. The FFN performs a linear transformation that is computationally much heavier than the linear projection on the patches, making the reuse of the outputs much more significant. This behavior is clearly reflected in the graphs, in which IS becomes the least efficient dataflow, while WS, although comparable to OS, never manages to reach the same performance. In conclusion, after the first stage, the workflow of the ViT becomes very similar to a dense layer, where accumulations play a critical role. Also in this case, the pseudo-symmetric shapes are the best. The result is therefore perfectly consistent with the structure of the model: the ViT moves from a layer strongly driven by the inputs to one dominated by accumulations. From the experimental data:

- OS is the best dataflow in this layer. Keeping the partial outputs locally reduces the internal traffic and also allows obtaining optimal values in all shapes and for any number of PEs in the array.

- WS shows stable values in all the examined cases. It obtains performance comparable to the OS case and generally higher than IS. The high number of channels generates a consistent reuse of the weights, making this configuration more competitive compared to Layer 0.
- IS shows a significant and widespread reduction in performance, resulting in the least efficient option. As expected, the reuse of the inputs is much more limited compared to the reuse of the outputs.

3.4 Discussion

TinyBERT, GPT, MobileNet, ResNet and ViT models follow two clear and almost opposite behaviors, which reflect their intrinsic nature: on one hand, the Transformer-based models (MobileNet, ResNet) show very similar patterns, while on the other, the convolutional networks display computation dynamics and data-reuse characteristics that require significantly different hardware-mapping strategies.

Starting from the Transformer-based group (TinyBERT, GPT and ViT), a strong coherence can be noticed. The dominant operations are dense, regular, and with a high number of accumulations. The structure of the layers - projection in TinyBERT and GPT and the patch embedding in ViT - is almost entirely composed of dense matrices in which the dataflow is very uniform. This leads to an almost predictable behavior on Systolic Arrays: dataflows that favor input reuse (IS) tend to dominate in the first layer, while in the deeper projections the dataflows centered on the outputs (OS), thanks to the high number of accumulations, win. In other words, TinyBERT, GPT, and ViT "pull" the hardware in the same direction: dense matrices, regular flows, high parallelism, and a clear preference for rather regular arrays, often close to square shapes.

Instead, the observations that arise with MobileNet and ResNet are completely different. In those models the computation is dominated by various types of convolutions. MobileNet, optimized for efficiency and MAC reduction, introduces operators like depthwise separable convolutions that fractionalize the computation into smaller steps with highly variable data reuse. This makes the behavior significantly less uniform and more sensitive to the shape of the SA: small variations in array geometry or in the choice of dataflow can significantly affect overall efficiency. On the other end, ResNet presents extremely heavy initial layers - like the 7×7 convolution - that saturate the hardware almost regardless of the array shape. However, in deeper layers, the situation gets reversed, with different reuse patterns and the need to balance inputs, outputs, and weights in a more variable way. In a nutshell, while Transformers present a "stable and regular" behavior, CNNs bring a much more heterogeneous nature, producing peaks, transitions, and non-uniform hardware requirements.

This contrast between the two "worlds" leads to the important observation that there does not exist one fixed combination of shape and dataflow that can be optimal across all modern models. An architecture optimized to reach maximum efficiency on Transformers would underutilize the available resources when running depthwise or pointwise layers. Similarly, an array optimized with only CNNs in mind would clearly not manage to exploit the regular characteristics of the dense matrices of LLMs. Even within a single architecture, such as ViT, with Layer 0 following completely different dynamics from the subsequent layers, the optimal mapping is not constant. In summary, from the obtained results, an immediate desire to adopt a reconfigurable Systolic Array will emerge, one that can adapt in shape and dataflow according to the type of operation to be executed. This flexibility lets the hardware follow more closely the characteristics of different workloads: more regular configurations and IS/OS data flows are more suitable for Transformer models, while less symmetric shapes and specific strategies perform better within convolutional layers.

Instead of having a single, rigid design, which would be suboptimal in most cases, a reconfigurable SA allows approaching each time the best efficiency point, improving the overall throughput per Watt. The variety present in the behaviors of the models thus does not constitute a problem but simply underlines how a single configuration cannot be valid for everything. Here lies the interest in a multi-configuration architecture, capable of choosing the most opportune layout for the context and of covering a wider range of computational patterns, both those of classic CNNs and those introduced by Transformer models, widespread today.

Chapter 4

Proposed Reconfigurable Systolic Array Architecture

This chapter presents the reconfigurable architecture developed during this thesis work. The objective is to illustrate the design choices adopted and to show how the results obtained in the previous chapters were leveraged for the integration into a single hardware structure capable of adapting to multiple operational configurations. After defining the design objectives, the internal organization of the Processing Element will be examined, followed by the reconfiguration mechanism, highlighting the logic behind its design and its impact on system complexity. In the final part, the experimental results obtained through Python simulation (with cross-verification via RTL simulation) and physical synthesis are reported, with particular attention to the costs associated with multi-configuration support. The overall analysis will make it possible to assess the effectiveness of the proposed architecture and to discuss its limitations and potential extensions.

4.1 Design Goals and Specifications

The decision to present a reconfigurable array is based on a set of objectives arising from the increasingly demanding workloads of neural networks, as well as insights drawn from the most recent literature. Among the works that most significantly influenced this research is EPFL's “Scale-out Systolic Arrays”[32] which demonstrates that the efficiency of SAs does not depend exclusively on the number of PEs operating but, depending on the model being executed, varies with the shape of the array itself and with the way it can be modified to accommodate different computational patterns. The research group shows that an array that is too large or too symmetrical compared to the real use case leads to underutilization of the PEs, consequently reducing the overall efficiency of the system. This study was crucial in guiding the idea, in this thesis, of introducing explicit support for multiple shapes and multiple operating modes.

In the modern context, it is necessary to ensure a good balance between throughput and energy consumption. Indeed, in architectures dedicated to inference, energy efficiency per Watt now plays a central role in hardware selection. This has pushed research to shift its focus toward energy optimization rather than absolute computational power: if, by reducing the number of PEs (sometimes at the cost of longer computation time), it is possible to achieve significant power savings—potentially by reorganizing the shape—without excessively increasing latency, the resulting hardware is better. The choice of the various test configurations and the proposed number of PEs was also driven by the need to maintain good utilization even when the problem size does not match the ideal shape of the array, drawing inspiration from EPFL's considerations regarding the relationship between SA granularity and the variability of DNN kernels.

A second objective concerns flexibility. As discussed in chapter 3, modern neural networks do not exhibit a single dominant computational pattern: 2D convolutions, linear layers, and attention mechanisms require different structures, both in terms of matrix dimension ratios and data reuse. For this reason, an array was proposed that could adapt to multiple dataflows and modify its shape without compromising internal regularity or excessively increasing the complexity of the control unit.

Another parameter that must be carefully monitored is the overhead introduced by the reconfiguration mechanism. Supporting multiple configurations inevitably means adding selection logic, alternative data paths, and components that are unused in certain modes. For this reason, the number of configurations supported within the same SA was carefully chosen to integrate this flexibility with as small an increase in area and power as possible, so as not to negate the advantages achieved in the most efficient configurations.

Finally, the choice of the reference technology — 28 nm standard-cell — reflects a library frequently used in the SA domain for latency and energy consumption estimation. The specifications of the array, in conclusion, were defined based on the performance required to cover the main kernels while also maintaining scalability toward larger sizes, partially adopting the "scale-out" logic proposed by EPFL[32] but integrating it into a single, reconfigurable design rather than a collection of separate pods.

4.2 Processing Element (PE) Design

The Processing Element (PE) constitutes the basic block of the systolic architecture under analysis. In particular, since the main objective of this proposal is the complete modularity of the system, the PEs have also been equipped with specific mechanisms to support multiple operating modes and multiple array configurations without requiring any modification of the PE itself. All of this must, however, continue to guarantee the regular structure and predictable data flow that characterize systolic architectures.

For this reason, data enters through a network of multiplexers. At this stage, only the innermost layer of multiplexers is considered, as it is synthesized as an integral part of the PE. These muxes are of the 2-to-1 type and, depending on the control signal, select which input is the stationary data. This level of flexibility is thus what enables the variation of the dataflow used. Although the inclusion of these components introduces a slight area increase, it offers a substantial advantage: using only 3 control bits, it is possible—while keeping the PE unchanged—to support all three classical DNN dataflows.

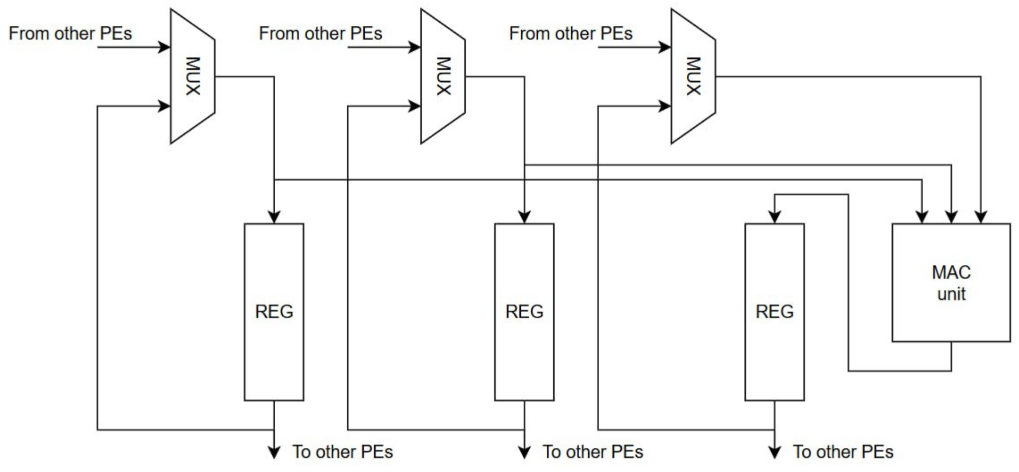

Figure 4.1: PE RTL view.

After the selection stage, the signal reaches the register layer, which in this case serves two roles. On one hand, the registers create an internal pipeline, ensuring that each value follows a regular and timed propagation along rows and columns. On the other hand, they ensure correct synchronization between the value recirculating inside the PE and the new wavefront step propagating through the array. Indeed, the register is isolated when the mux selects the value coming from the outside.

The core of the PE is the MAC unit, which performs the fundamental operation of DNN kernels (multiplication of two operands followed by accumulation). To better support different types of workloads and optimize energy efficiency, the PE—as well as the entire testing flow—has been designed for the use of a variable-precision MAC unit, capable of performing multiple MAC operations in parallel by reducing the number of bits per operand. In the VHDL unit, as well as during synthesis, a 3-bit bus is included for controlling the MAC unit itself.

In the configuration used in this thesis, the inputs A_i/A_o and B_i/B_o (see **??**) are 16-bit vectors, while the accumulator C_i/C_o is 48 bits wide. However, the code structure is designed to generalize these widths and allow future exploration of other precision combinations. This design choice enables the use of reduced precision when the model allows it—thus lowering area and power—while still supporting higher precision for applications requiring a greater numerical margin, without altering the array organization.

From an interface perspective, the PE has three input data ports and three corresponding output ports. The A and B pairs carry the operands of the multiplication, introducing a one-register delay between input and output. In accordance with the chosen dataflow, the output is then directed either to other PEs or to the output buses. In weight-stationary configurations, for example, weights may remain locally in the PE while activations propagate, whereas in other modes the roles can be defined differently. The C signal is the partial-accumulation bus and supports the same reconfigurability described for the other two inputs. The fact that all three data channels have well-defined input and output ports simplifies the regular interconnection between PEs and enables the construction of arrays of different sizes without modifying the interface of individual nodes.

As mentioned earlier, in addition to the data signals, the PE entity also includes two control buses. The first groups the signals governing the internal behavior of the MAC unit. The second controls the configuration of the multiplexers that select the data source feeding the MAC unit, determining which of the three inputs recirculates through the PE and, consequently, the mode in which the PE is configured (OS, WS, IS). In this way, by modifying only the control signals, it is possible to locally reconfigure the dataflow within the PE, switching from a convolution-oriented configuration to one suited for linear layers or attention, without altering the physical connections.

Once the MAC operation is completed, the result is sent to the subsequent PEs according to the connections defined by the array topology. The symmetry between the PE's inputs and outputs was intentionally preserved. This symmetry allows the definition of a modular structure that can be easily replicated on multiple levels, without introducing dedicated paths that would increase routing complexity. Such an approach helps maintain a high degree of layout regularity—an important aspect of systolic architectures to ensure orderly routing and reduce interconnect congestion.

4.3 Reconfiguration Mechanism

The reconfiguration mechanism adopted in this work is based on a static approach, easily integrable into the design flow, and with reduced overhead. The configuration of the Systolic Array does not occur at runtime but is entirely defined at compile time through a constant declared in the VHDL code. This constant controls directly and straightforwardly the generation of the interconnection between the PEs during the simulation and synthesis: depending on the constant set, the compiler automatically introduces the multiplexers on the inputs of each PE, determining in this way the data path inside the array. It means that the set of supported configurations is defined before the synthesis and that, for each desired configuration, VHDL generates only the logic really needed without introducing unused structures or performing complex dynamic control.

From an architectural point of view, this approach allows for an important decoupling between physical layout and logical topology. The PEs are physically disposed on silicon in a regular grid, but the way they are connected does not depend on their geometric position. Actually, the i-th PE is not bound to its spatial coordinate (x, y): depending on the chosen connections, it can behave like it occupied another position in the logical matrix, receiving or forwarding data along rows or columns different from the physically adjacent ones. It is hence the interconnection network which is generated that defines the effective "shape" of the array. This mechanism allows for the obtaining of wide and short arrays, tall and narrow ones, or more regular and square ones, all based on the very same underlying physical structure.

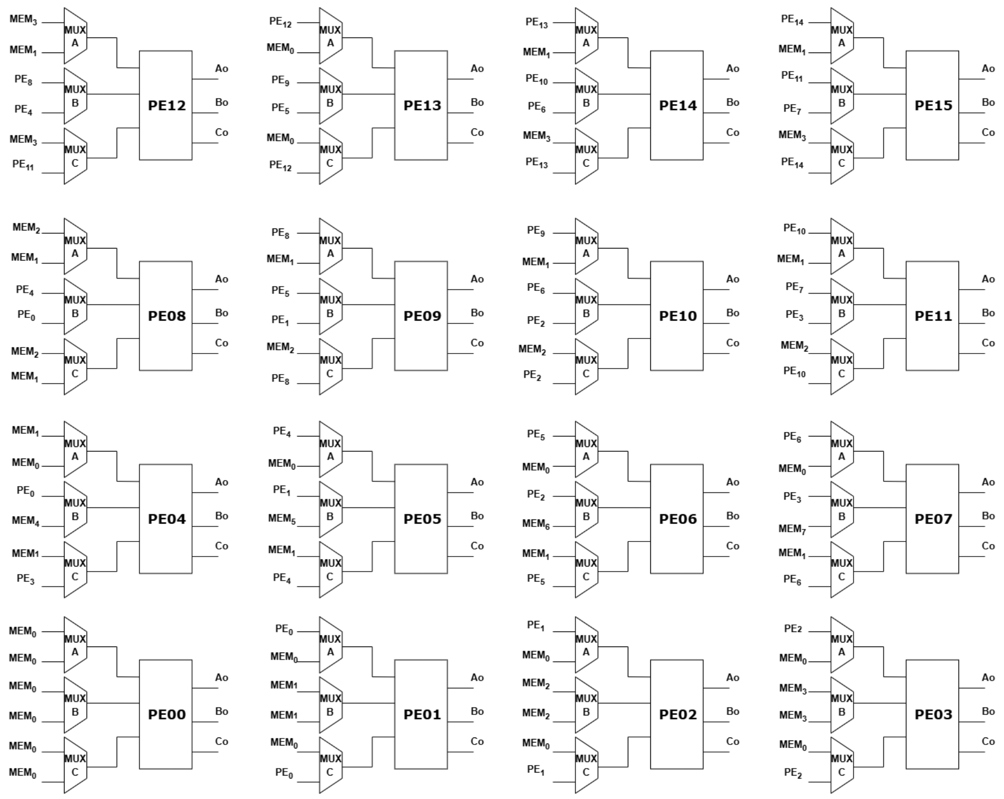

Figure 4.2: Example of a Reconfigurable 4×4 Systolic Array Supporting 4×4 OS and 2×8 IS Shapes

The core of this process is represented by the use of multiplexers on the PE inputs (Figure 4.2). In fact, depending on the configuration, each PE can receive data from the left neighbour, from the upper one, or from an external source, thus modifying the natural data-propagation path typical of traditional Systolic Arrays. By imagining interconnections that potentially are not limited to adjacent PEs in a two-dimensional structure, it is possible to even consider the optimization of algorithms such as convolutions, which exploit multiple spatial dimensions. The set of mux choices builds up a different connection graph for every configuration: some paths are enabled, others completely removed. This permits control of not only the size of the logical matrix but also the type of dataflow that can actually be implemented, allowing the SA to adapt to operations very different from one another. The advantage of this strategy is twofold. On one hand, it keeps the hardware overhead extremely small because adding multiplexers involves in a limited increase in area and does not introduce complex control logic. On the other hand, it makes the array extremely flexible at the functional level. The same physical layout can be used for configurations optimized both for Transformer models-which benefit from more squared shapes and dataflows with strong input/output reuse-but also for convolutional architectures like MobileNet and ResNet, where instead rectangular shapes or specific data paths can be useful in order to favor weight reuse or a particular direction of dataflow. It is nevertheless necessary to keep the number of configurations under control. In this work, the exploration was carried out up to a maximum of 3 configurations supported simultaneously; however, even if theoretically feasible, increasing the number of configurations too much - due to the area overhead caused by the additional mux logic and the increased power linked to a more complex routing and consequently larger drivers - might not justify the performance gains. This solution provides an effective compromise between flexibility and efficiency, with a limited overhead it should be possible to archive better performance.

4.4 Performance Results

This section discusses the outcome obtained for each explored configurations and interprets it. The idea is to relate performance trends with architectural choices - with particular emphasis on shape and dataflow - and the peculiar characteristics of the executed models. Such a comparison could bring out common trends, significant divergences, and structural limitations and give a critical reading of the observed behaviors and of the factors that most influence the overall efficiency of the array.

4.4.1 Methodology for data computation and reason behind the selection of comparison shape.

The procedure used to compute the data and compare the different configurations of the array is based on an energy normalization, which is necessary to evaluate in a

Shape	Area [mm^2]	Incremento [%]
32x32	1,065	+12,84
32x32(OS) + 16x64(OS) + 64x16(OS)	1,202	
32x32	1,066	+7,64
16x64(OS) + 64x16(OS)	1,147	
64x32 (32x64)	2,125	+7,79
64x32(OS) + 32x64(OS)	2,291	

Table 4.1: Confronto fra area della shape fissa e shape riconfigurabile

coherent way layers with different sizes and characteristics. The issue arises when, in the case of reconfigurable arrays, there is a difference in power consumption between one shape and another. For each layer, starting from the power estimated during synthesis and from the number of cycles required to complete the operation, the total energy was calculated according to the relation:

$$\begin{cases} e_{L0} = W_{L0} \cdot nccs_{L0} \\ e_{L1} = W_{L1} \cdot nccs_{L1} \end{cases} \tag{4.1}$$

Once the energy contribution of the two layers is obtained, the total energy is simply:

$$e_{tot} = e_{L0} + e_{L1} \tag{4.2}$$

In parallel, for each configuration, the total number of operations performed was also calculated:

$$n_{ops,tot} = OPs_{L0} + OPs_{L1} \tag{4.3}$$

These two quantities make it possible to derive the same efficiency metric used in the fixed–dimension array cases, expressed as OPC per Watt:

$$\frac{OPC}{mW} = \frac{n_{ops,tot}}{e_{tot}} \tag{4.4}$$

This approach makes it possible to uniformly compare models that are very different from one another, preventing particularly long or heavy layers from distorting the result.

For the choice of the shapes to compare, configurations commonly adopted in the literature on Systolic Arrays were preferred. Square matrices, such as 32×32, are a classical choice due to their symmetry and good balance between vertical and horizontal dataflow. In fact, even reference architectures, such as the one used as a basis in the EPFL[32] study, employ this shape as their design starting point. The semi-square configurations 64×32 and 32×64, instead, reflect scenarios often discussed in research to represent arrays unbalanced toward rows or columns, while still maintaining a regular and easily mappable structure. The adoption of these shapes therefore enables a meaningful comparison aligned with the solutions most widely studied in the state of the art. The proposed configurations are three in total:

- The first two are based on an array composed of 1024 PEs. The first supports two reconfiguration options, while the second supports three. The introduction of a configuration with three selectable modes was intended to provide a broadly adaptable architecture capable of performing effectively across all the models analyzed.
- The third configuration is based on a 2048-PE array and supports two combinations of shape and dataflow. This design is closer to a datacenter-oriented array while still maintaining a more limited PE count, thus offering a balanced trade-off between scalability and hardware cost.

4.4.2 Synthesis and Resource-Overhead Results

2 Shapes - 1024 PEs

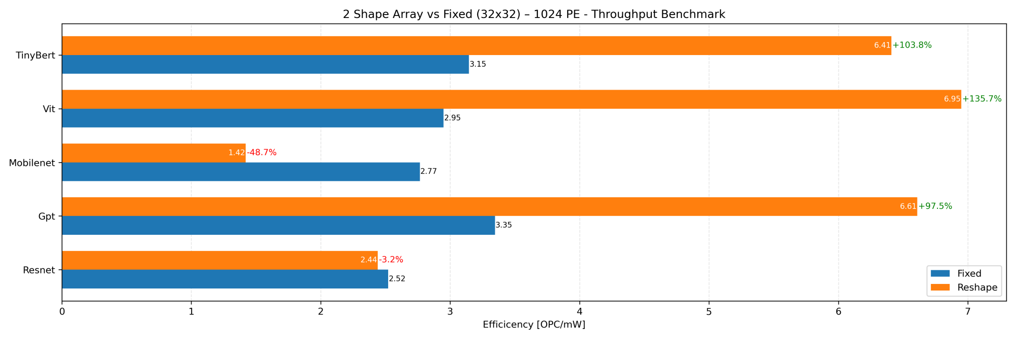

Figure 4.3: 2-Way Reconfigurable vs Fixed Shape Array - 1024 PEs

Figure 4.3 clearly shows the impact of introducing a reconfigurable architecture compared to a traditional fixed 32×32 Systolic Array. Even when limited to two alternative shapes, the reconfigurable array achieves significant improvements across almost all the models analyzed. The most evident gains appear in Transformer-based models — TinyBERT, GPT, and ViT — where the efficiency increase exceeds +90% in all cases, with particularly high peaks for ViT (+135%) and TinyBERT (+103%). This behavior was expected: the Transformers considered exhibit a large efficiency variation for a given dataflow between their first and second layer. There is, in fact, a complete shift between the projection layer and the FFN, where IS and OS respectively emerge as the most efficient dataflows, almost independently of the shape. The flexibility introduced by reconfiguration therefore allows the dataflow of the array to be better aligned with the nature of the matrices used in self-attention and linear projections. GPT also shows a substantial improvement (+97%), further confirming that many attention-based networks exhibit a data-reuse profile much closer to the ideal case for which the second shape included in the reconfigurable design was conceived.

The case of MobileNet, on the other hand, shows the opposite behavior: the reconfigurable version experiences a loss of about 48%, which is consistent with what was observed in the depthwise/pointwise layers of efficient networks. MobileNet presents less regular computation patterns and is more difficult to map onto rigid and symmetric shapes; indeed, in many cases, the fixed 32×32 array already represents a better compromise than the shape selected for the two-configuration design. MobileNet, like the early layers of a CNN, does not exhibit large enough variations in dataflow or shape to justify the need for a reconfigurable array. In such cases, introducing a multi-shape configuration and paying the associated power overhead actually leads to a deterioration in efficiency. This result highlights that certain workloads cannot be effectively supported by adding only a single alternative shape, and that reconfigurability must be carefully calibrated to the characteristics of the model.

ResNet lies in an intermediate position: the gain is minimal and essentially neutral (3.2%). Again, when limiting the study to the first two layers alone, the variation in efficiency between them is not large enough to justify the presence of a second shape. However, this negligible variation in efficiency offers insight into the equally negligible energy overhead of the variable configuration. In this case, for instance, there is a power-dissipation increase of 7.73% in the IS configuration and 4.46% in the OS configuration, which corresponds to an area increase as reported in Table 4.1.

3 Shapes - 1024 PEs

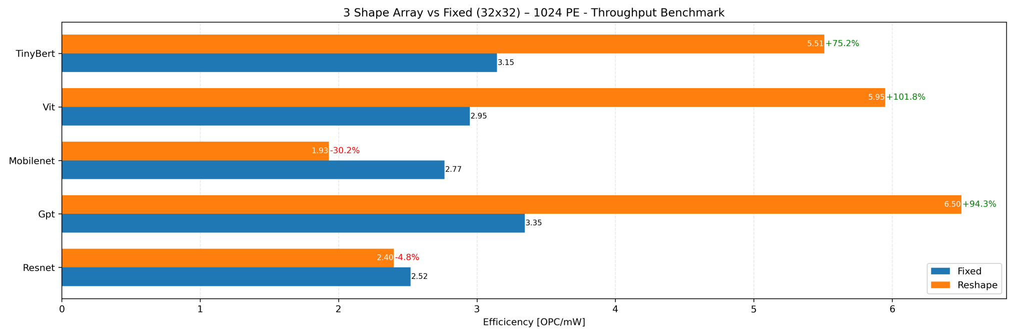

Figure 4.4: 3-Way Reconfigurable vs Fixed Shape Array - 1024 PEs

The third shape (results in Figure 4.4) was introduced with the specific goal of improving performance on convolutional networks, and in particular on MobileNet, which suffered the most under the two-shape configuration. It is inspired by the fact that, for a large portion of convolutional layers, the 32×32 shape is reported in the literature as the optimal solution. The addition of an extra shape was then meant to offer a more suitable compromise for the depthwise and pointwise convolutions typical of MobileNet, with the hope to reduce the gap with respect to the fixed configuration.

Despite this rationale, results demonstrate that the addition of the third shape is not a solution and introduces a very undesirable effect: supporting an extra configuration increases general power consumption and this increment is not compensated by the benefits obtained. The final result is that MobileNet continues to perform worse than the fixed 32×32 array –30.2%, although it improves compared to the two-shape array, which still remains considerably below the baseline. In other words, the third shape reduces the problem but far from removing it, and taking into consideration the higher energy cost, it is not justified.

For ResNet, too, the improvement is marginal or even slightly negative (–4.8%): its heavy convolutional layers do not take advantage of the new configuration, and the increased power related to the third shape cancels out any potential benefit. Transformer architectures enjoy large improvements again, but these were already there with the two-shape array; the third shape does not introduce any additional significant increase beyond what was already achieved by two configurations.

The most relevant outcome that can be highlighted from this comparison is that increasing the number of shapes does not automatically imply better performance and can even worsen the most critical cases when the power overhead is not compensated by a real gain in efficiency. Even though the introduction of a third shape was well-motivated, it does not overcome the challenges of lightweight CNNs, and in the light of these results, it does not represent a beneficial choice with respect to the two-configuration solution.

2 Shapes - 2048 PEs

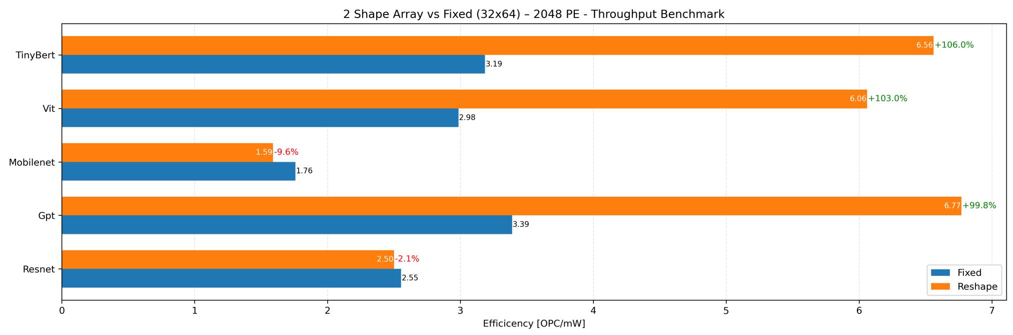

Figure 4.5: 2-Way Reconfigurable vs Fixed Shape Array - 2048 PEs (32x64 baseline)

Increasing the number of PEs from 1024 to 2048 significantly alters the context of the analysis: this configuration is closer to the high-end architectures typical of datacenter environments where the workload is dominated by Transformer models and dense operators. In this scenario, the introduction of reconfigurability thanks to two alternative shapes comes along with effects that are even more pronounced when compared to the results obtained with the smaller array.

In Transformer-based models—TinyBERT, ViT, and GPT—the effect of reshaping stays extremely high; there, improvements oscillate around +100%, depending on the chosen shape (as in Figure 4.5). The larger array size allows the regularity of the matrix patterns typical of attention and linear projections to be fully exploited, and the more balanced shapes (32×64 and 64×32) increase array saturation and reduce parallelism waste. This behavior is precisely what one would expect in a datacenter-like scenario: more PEs translate into greater "sensitivity" for data layout, and Transformer models are the ones that profit the most from this combination.

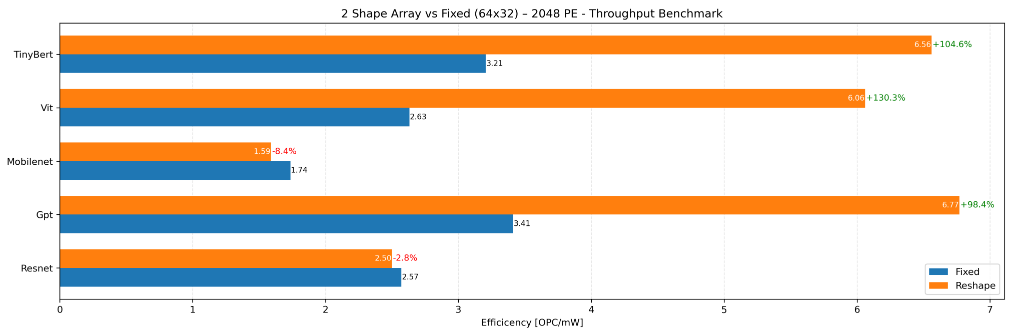

Figure 4.6: 2-Way Reconfigurable vs Fixed Shape Array - 2048 PEs (64x32 baseline)

As far as CNNs are concerned, the picture is more nuanced. MobileNet still appears to be suffering from a degradation with respect to the fixed 32×32 configuration, around values of –9% to –8% — smaller than in the 1024-PE case, but negative nonetheless. The efficiency is, however, extremely low even in fixed configuration, due to the small size of the model compared to the considered compute matrices, which results in a limited array filling and correspondingly low PE utilization. Introducing reshape causes a negligible efficiency drop, primarily due to the increased dissipated power.

Instead, ResNet exhibits a mild degradation (–2% / –3%), similar to the 1024-PE case (see Figure 4.6): even a datacenter-scale array is not enough to substantially accelerate the initial convolutional layers of ResNet, due to the two additional shapes that fail to model the weight-reuse pattern required by either the 7×7 convolution or by the subsequent bottleneck convolutions. On the whole, these results show that even with a significantly larger number of PEs — and therefore in a scenario closer to datacenter architectures — reconfigurability amplifies the advantages for regular models such as Transformers, while remaining limited in the presence of highly heterogeneous workloads, such as lightweight CNNs or CNNs with very complex kernels. This confirms that reconfigurability is an effective approach mainly when the array can be uniformly saturated, whereas in less regular models more targeted and more numerous shapes would be needed in order to have tangible performance gains.

Chapter 5

Conclusions and Future Work

The following chapter summarizes the main results obtained during this work, putting forward some reflections on the implications that emerged from the analysis of the different configurations of the Systolic Array. After evaluating how the shape and dataflow of standard arrays behave when executing the models of interest — from classical CNNs to Transformers — and, subsequently, how the variable systolic architecture influences their efficiency, it is possible to draw some general considerations on the effectiveness of the adopted approach and on its limitations. Starting from these observations, several possible directions for extending the project are then outlined, both from an architectural and a methodological point of view. The goal is to provide an overall view that does not limit itself to the results obtained, but that also highlights the opportunities and open challenges for the design of reconfigurable accelerators in future generations.

5.1 Conclusion

The data presented in chapter 4 highlighted how shape and dataflow critically influence the efficiency of the Systolic Array across the various models considered. The experiments confirm that there is no single universal configuration capable of adapting effectively to the heterogeneity of modern workloads: Transformer-based models, characterized by dense and regular operations, benefit significantly from more symmetric shapes, while convolutional networks — such as MobileNet — require different topologies, often more unbalanced and difficult to capture with a small set of generic configurations.

The comparison between fixed and reconfigurable arrays clearly shows that reconfiguration, even when limited to two shapes, can substantially improve efficiency in the presence of regular models, while it remains more challenging to apply effectively to traditional CNNs. Extending the system to three shapes, although motivated by a slight improvement in convolutional networks, does not provide benefits sufficient to justify the higher power cost and the increased architectural overhead, indicating that the choice of configurations must be carefully reasoned and not merely incremental.

Overall, the results demonstrate that reconfigurability is a viable approach to broaden the spectrum of workloads that a Systolic Array can handle, but they also reveal the need for a deeper exploration of the design space and more advanced tools to guide the selection of shapes. These observations serve as the foundation for future developments aimed at a more informed, automated, and comprehensive design of the array's computational and energy behavior.

5.2 Future Work

The field of reconfigurable Systolic Arrays remains broad and rich in possibilities for improvement. A first area of work concerns the definition of more structured strategies for mapping neural network models onto the array. The exploration carried out in this work was conducted by combining theoretical analyses and empirical evaluations, but a promising direction is the use of more formal techniques such as the polyhedral model, or multi-dimensional scheduling models. These tools would make it possible to systematically analyze tensor dependencies, automatically generate computation patterns, and suggest optimal shapes and dataflows for each layer. In perspective, this would enable shifting from a manual management of the exploration to a semi-automatic or fully automatic system for generating configurations.

Another important point raised during the study is the lack of a detailed simulation model of the communication towards memory. Even though the Systolic Array is very effective at reducing the external traffic due to data reuse, memory bandwidth and latency remain important factors, especially considering highly unbalanced layers or models with big activations. A more realistic simulator, which also considers local caches, intermediate buffers, bus conflicts, and possible on-chip network congestions, can discover configurations not yet explored in the design space and lead to the discovery of optimal shapes that would be discarded normally. A better understanding of the memory behavior can also allow the integration of techniques such as dataflow-aware prefetching or dynamic buffer allocation

Another important point is the study of bus management and internal interconnection of the array. Shapes that today are extremely unbalanced toward rows or columns and represent edge cases might become more practical by introducing systems such as bus sharing, multi-hop routing, or more flexible internal communication networks, possibly hybridized with small Network on Chip (NoC) structures.

This will reduce the time required to reorganize the dataflow when switching, for example, from a row-oriented configuration-suited for wide convolutions-to a column-oriented one-better suited for the densely connected layers of Transformers-and also reduce the number of minimal interconnections that today must be placed.

A further consideration can be made regarding the area and power estimates used in this work. RTL synthesis provides useful indications, but it cannot fully capture the real costs of an architecture. In particular, the physical convergence of wires—necessary to support an increasing number of shapes—may have a much more significant impact once the actual layout is generated: denser mux networks, irregular routing paths, and local congestion could substantially increase area and parasitic capacitances, also penalizing the maximum achievable frequency. For this reason, an important direction for future work is the implementation of a complete physical layout to more accurately extimate the cost of reconfigurability in terms of routing and chip-level physical effects.

The same reasoning applies to the power estimation, which in this study is based on the standard pre-layout power analysis provided by synthesis tools. Although this estimation is useful for relative comparisons, it does not fully reflect the dynamic behavior of the neural network during real execution. A more accurate evaluation would require a back-annotated simulation, including post-layout RC values and switching patterns derived from the actual input data of the models. This would allow a more reliable measurement of dynamic power and a better understanding of how reconfigurability affects the real energy consumption of the array. Integrating such a flow therefore represents a natural next step to strengthen the results obtained and to guide future architectural optimizations.

Another line of research relates to the possibility of enriching the PEs with forms of local micro-control. The current design of the PEs, although simple and efficient, is completely static: each PE always performs the same behavior, regardless of the layer or of the characteristics of the computation. The introduction of extremely lightweight micro-controllers or small dedicated FSMs could allow the PE to dynamically change the direction of the dataflow, manage alternative paths, or even collaborate with other PEs to build functional groups useful, for example, for layers with very different dimensions. In the long term, this might lead to SAs able to reconfigure not only at compile-time but also at runtime, adapting to the next layer without the need for any re-synthesis.

From this concept naturally follows the possibility of sectorizing the array, that is, dividing the SA in independent regions that can be turned on, turned off, or resized according to workload: such a configuration would allow a reduction in consumption when computing small or narrow layers, or an increase in parallelism when dealing with large matrices. The introduction of selective power-gating, already widely studied in other hardware domains, may represent a further improvement in energy efficiency.

Another interesting perspective concerns integrating heterogeneous compute units within the same SA: part of the array could keep the traditional PEs; other sections could be equipped with PEs optimized for specific operations, such as activation functions, normalization operations, or multihead-attention in Transformer models. This would reduce the need to move data outside the array for simple but frequent operations, further increasing efficiency.

As a final step, the design flow can be further evolved by including automatic design-space exploration tools, capable of proposing candidate configurations based on composite metrics - OPC/W, area, latency, and memory pressure - and of guiding the designer in the selection of the most promising configurations to synthesize. Such a tool, combined with formal mapping techniques and with an advanced memory model, can eventually allow for more rapid, robust, and scalable design of Reconfigurable SAs.

Overall, the future developments are toward ever more intelligent Systolic Arrays, aware of the workload and able to adapt dynamically both to the structure of computation and to energy constraints. Quite an ambitious goal, but well aligned with the needs of current deep learning models, which ask for accelerators that are not only powerful but also versatile and adaptive.

List of Figures

Bibliography

[1] M. Lemmon, *Deep Learning Book 2025*. University of Notre Dame, 2025, Lecture Notes. [Online]. Available: `https://academicweb.nd.edu/~lemmon/courses/deep-learning/lecture-book/deep-learning-book-2025.pdf`.

[2] F. Rosenblatt, «The perceptron: A probabilistic model for information storage and organization in the brain», *Psychological Review*, vol. 65, no. 6, pp. 386–408, 1958. DOI: `10.1037/h0042519`.

[3] B. Widrow and M. E. Hoff, «Adaptive switching circuits», Stanford University, Stanford Electronics Laboratories, Stanford, CA, Technical Report, 1960.

[4] A. Krizhevsky, I. Sutskever, and G. E. Hinton, «Imagenet classification with deep convolutional neural networks», *Communications of the ACM*, vol. 60, no. 6, pp. 84–90, 2017. DOI: `10.1145/3065386`.

[5] J. Kaplan et al., «Scaling laws for neural language models», *arXiv preprint arXiv:2001.08361*, 2020. [Online]. Available: `https://arxiv.org/abs/2001.08361`.

[6] J. Sevilla et al., «Compute trends across three eras of machine learning», in *Proceedings of the 2022 International Joint Conference on Neural Networks (IJCNN)*, IEEE, 2022. DOI: `10.1109/IJCNN55064.2022.9891914`. [Online]. Available: `https://doi.org/10.1109/IJCNN55064.2022.9891914`.

[7] D. Patterson et al., «The carbon footprint of machine learning training will plateau, then shrink», *IEEE Computer*, vol. 54, no. 10, pp. 18–28, 2021. DOI: `10.1109/MC.2021.3099094`. [Online]. Available: `https://doi.org/10.1109/MC.2021.3099094`.

[8] H. T. Kung and C. E. Leiserson, «Systolic arrays (for VLSI)», in *Sparse Matrix Proceedings 1978*, Reprinted in "Introduction to VLSI Systems", Addison-Wesley, 1980, Philadelphia, PA: Society for Industrial and Applied Mathematics, 1979, pp. 256–282.

[9] U. S. Solangi, M. Ibtesam, M. A. Ansari, J. Kim, and S. Park, «Test architecture for systolic array of edge-based ai accelerator», *IEEE Access*, vol. 9, pp. 96 700–96 710, 2021. DOI: `10.1109/ACCESS.2021.3094741`.

[10] J. Zheng, Y. Liu, X. Liu, L. Liang, D. Chen, and K.-T. T. Cheng, «Reaap: A reconfigurable and algorithm-oriented array processor with compiler-architecture co-design», *IEEE Transactions on Computers*, vol. PP, pp. 1–14, Dec. 2022. DOI: 10.1109/TC.2022.3213177.

[11] R. Xu, S. Ma, Y. Guo, and D. Li, «A survey of design and optimization for systolic array-based dnn accelerators», *ACM Comput. Surv.*, vol. 56, no. 1, Aug. 2023, ISSN: 0360-0300. DOI: 10.1145/3604802. [Online]. Available: https://doi.org/10.1145/3604802.

[12] NVIDIA Corporation, *Nvidia corporation: Technology and products overview*, General corporate reference; NVIDIA Corporation, 2025. [Online]. Available: https://www.nvidia.com.

[13] Google Cloud. «An in-depth look at google's first tensor processing unit (tpu)». Google Cloud Blog; Accessed: 2025-11-14. [Online]. Available: https://cloud.google.com/blog/products/ai-machine-learning/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu.

[14] E. Gomede. «Under the hood: Why compute primitives and memory layouts matter for cpu, gpu, and tpu». Blog post; [Online]. Available: https://blog.stackademic.com/under-the-hood-why-compute-primitives-and-memory-layouts-matter-for-cpu-gpu-and-tpu-4338c190efbc.

[15] WikiChip Fuse. «Inside tesla's neural processor in the fsd chip». Accessed: 2025-11-14. [Online]. Available: https://fuse.wikichip.org/news/2707/inside-teslas-neural-processor-in-the-fsd-chip/.

[16] A. Karpathy, «Pytorch at tesla», in *Proceedings of the PyTorch Developer Conference 2019*, October 10, 2019; talk given at PyTorch Developer Conference, 2019. [Online]. Available: https://www.youtube.com/playlist?list=PL_lsbAsL_o2BY-RrqVDKDcywKnuUTp-f3.

[17] NVIDIA Corporation, *Nvidia deep learning performance*, Last updated July 27, 2023., NVIDIA Corporation, 2023. [Online]. Available: https://docs.nvidia.com/deeplearning/performance/index.html.

[18] MathWorks Inc., *2-d convolution*, MathWorks Inc., 2025. [Online]. Available: https://it.mathworks.com/help/vision/ref/2dconvolution.html.

[19] GeeksforGeeks. «Cnn | introduction to padding». Last updated: 13 Dec, 2023; [Online]. Available: https://www.geeksforgeeks.org/machine-learning/cnn-introduction-to-padding/.

[20] Google LLC. «Llms: What's a large language model?» [Online]. Available: https://developers.google.com/machine-learning/crash%20course/llm/transformers.

[21] R. Venkatesan et al., «Magnet: A modular accelerator generator for neural networks», in *Proceedings of the 2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2019, pp. 1–8. DOI: `10.1109/ICCAD45719.2019.00008`.

[22] A. Aimar et al., «Nullhop: A flexible convolutional neural network accelerator based on sparse representations of feature maps», *arXiv preprint arXiv:1706.01406*, 2017, Submitted June 2017.

[23] R. Y. Aminabadi, O. Ruwase, M. Zhang, Y. He, J.-M. Arnau, and A. González, «Sharp: An adaptable, energy-efficient accelerator for recurrent neural networks», *ACM Transactions on Embedded Computing Systems*, vol. 22, no. 2, p. 30, 2023. DOI: `10.1145/3552513`.

[24] X. Jiao et al., «Tinybert: Distilling bert for natural language understanding», in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2020, pp. 4163–4174. DOI: `10.18653/v1/2020.emnlp-main.346`.

[25] A. Vaswani et al., «Attention is all you need», in *Advances in Neural Information Processing Systems*, vol. 30, 2017.

[26] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, «Language models are unsupervised multitask learners», OpenAI, Tech. Rep., 2019, OpenAI Technical Report. [Online]. Available: `https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf`.

[27] A. G. Howard et al., «Mobilenets: Efficient convolutional neural networks for mobile vision applications», in *arXiv preprint arXiv:1704.04861*, 2017.

[28] K. He, X. Zhang, S. Ren, and J. Sun, «Deep residual learning for image recognition», in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.

[29] Ultralytics. «Che cos'è resnet-50 e qual è la sua importanza nella computer vision?» Blog post, 27 maggio 2025. [Online]. Available: `https://www.ultralytics.com/it/blog/what-is-resnet-50-and-what-is-its-relevance-in-computer-vision`.

[30] A. Dosovitskiy et al., «An image is worth 16x16 words: Transformers for image recognition at scale», in *International Conference on Learning Representations (ICLR)*, 2021.

[31] G. Scarpi. «Vision transformer, cosa sono e perché rivoluzioneranno l'industria». [Accessed: 19 Nov 2025]. [Online]. Available: `https://www.ai4business.it/intelligenza-artificiale/vision-transformer-cosa-sono-e-perche-rivoluzioneranno-lindustria/`.

[32] A. C. Yüzügüler, C. Sönmez, M. Drumond, Y. Oh, B. Falsafi, and P. Frossard, «Scale-out systolic arrays», *ACM Transactions on Architecture and Code Optimization*, vol. 19, no. 4, pp. 1–26, 2022. DOI: `10.1145/3569014`.