POLITECNICO DI TORINO

Master of Science in Computer Engineering

Master Degree Thesis

A CoSimulation Framework for Assessment of Power Knobs in Deep-Learning Accelerators

Supervisor:
prof. Andrea Calimera

Co-Supervisor:
prof. Amit Trivedi

Candidate
Antonio Cipolletta

Academic year 2017 – 2018

Summary

The contribution of this work is a CoSimulation Framework for the Assessment of Power Knobs in Deep-Learning Accelerators. The tool is a reply to the demands of collaborative works between machine learning experts and digital designers in order to exploit at their fullest the possibilities of deep-learning-on-chip.

Acknowledgements

I would first like to thank my advisor Professor Andrea Calimera not only for the role of technical guide during the development of this work, but also for his humanity and the passion for research and innovation that he was able to impart to me during our infinite conversations.

I would like to thank Roberto Rizzo for his support during this intense period of work. He was available to investigate any doubts and ideas at any time of the day and always smiling.

Thanks to all people of lab 4, they are friends and not only colleagues.

I would like to thank my family for the invaluable support and for teaching me perseverance and the real meaning of knowledge.

To my friends ... because friendship is not important, it is the only thing that matters.

To my house mates ... thanks for your infinite patience.

To Elisa ... thanks for your harmonic behavior always in phase with mine.

Antonio Cipolletta
Torino, 18/10/2018

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In the last few years there has been a real renaissance of Machine Learning, the field of computer science studying algorithms allowing machines to sense complex information from raw data without being specifically designed for that.
Neural Networks and especially Deep Neural Networks, a particular branch of Machine Learning, have shown broad applicability from object classification and detection, to speech recognition and natural language processing.
Before employing a neural network it is necessary to perform a training phase where all the *learnable parameters* are set to optimize the network for a specific task. The training is usually done once and executed on power hungry systems like High Performance CPU, clusters of CPUs and/or clusters of GPGPUs.
Recently a huge effort has been put on bringing the inference phase, which consists in using a trained neural network to perform the task on the field, to resource-constrained devices with restricted power/energy capabilities. Different motivations could be addressed for this choice, but the general philosophy is the desire to exploit at their fullest the sea of devices that have been used until now just to collect and send raw data to data-centers. Security, reliability, reduced traffic on the network such as the perspective of having *smart devices* with the *sensemaking* ability is for sure highly attractive in many practical applications. For this reason, at algorithmic level it has been tried to optimize the neural network model using energy-aware heuristics, thus the birth of `pruning` to reduce memory footprint and total number of computations, `light networks` optimized for low hardware capabilities and `low resolution computations` using reduced bitwidth arithmetic operators. In parallel, on the computational side, digital designers and low-level software engineers started to rethink computer architectures and computational kernels to improve energy-efficiency and throughput of the hardware platforms in order to enable the deploy of neural networks on chip. Indeed, many hardware accelerators and innovative CPUs have been designed and fabricated and also optimized software libraries have been developed.
In order to facilitate research and use of machine learning algorithms many software frameworks like TensorFlow, PyTorch, Theano have been released. These frameworks provide the user with a modular infrastructure where he can choose how to customize the available basic blocks to adapt existing model to his specific application or to develop new models.

In particular they are all highly optimized for the execution on different hardware platforms like CPUs, GPUs and application specific architectures like the TPU[1] designed by Google.
On the opposite side, even if the hardware domain is full of CAD softwares aimed at providing powerful optimization tools to help engineers in the complex task of designing ASIC, there is a lack of tools enabling the exploration of joint hardware-software optimization in the deep-learning-on-chip domain. In particular focusing on power and energy optimization techniques, data-driven strategies are particularly appealing for embedded systems working in a multi-context scenario, where the non-functional requirements can change over time. The concept is to tune at run-time quantities affecting the power consumption of the systems, such as the power supply voltages of different die areas, clock frequencies, body-biasing or the precision of arithmetic operators. These power knobs affect other factors of merit, like throughput and/or the quality of result, thus designers need to verify that the system is able to work at the best trade-off point, while fulfilling the requirements of each working condition.
This thesis provides a tool able to meet the demands of a collaborative work between machine-learning experts and digital designers, with a particular attention to hardware accelerators implementing a spatial architecture. The latter consists of a large number of processing elements, interconnected with a network-on-chip allowing the sharing of operands and to carry on computations spatially. In particular, the contribution of this work has consists in the development of a co-simulation framework able to:

- Evaluate the effective energy efficiency of realistic workload of spatial accelerators avoiding the simulation of the entire accelerator micro-architecture.

- Explore the design space to evaluate pros and cons of the designated HW architecture and power-management strategy.

- Enable an early efficiency testing of energy-aware neural network model, without waiting for the complete design of the accelerator, thanks to an accurate estimation of the energy profile of the real hardware platform.

The tool has been designed in order to be easily interfaced with common frameworks for machine learning and with the industrial ASIC design flow. The general philosophy behind the co-simulation framework is to have a behavioral neural network inferential engine that communicates with a gate-level simulator: the inferential engine can provide stimuli to the circuit, collect responses, status signals and modify the configuration of power knobs. The aim is to simulate the system also from a non-functional perspective, thus the need for a gate-level simulator, but with only the minimum hardware required to verify the impact of a specific power management strategy on the network accuracy. In particular, the effect of power knobs on the system is emulated through a library of SDF files, one for each working condition, which can be loaded by the gate-level simulator when a power-context switch is performed.

[1] Tensor Processing Unit.

1.1 Outline

The thesis is organized as follows: in **Chapter 2** a general overview on the actual state of IC CMOS technology is reported, with a particular emphasis on power optimization techniques and power knobs, which are relevant to this work. **Chapter 3** is devoted to a description of neural networks with a focus on the computational perspective and hardware accelerations on spatial architecture. **Chapter 4** is fully focused on the functionality and architecture of the framework. It explains in detail how the front-end is designed to be interfaced with common machine learning frameworks and with the standard ASIC industrial design flow targeting std. cells technology. The architectural choices are illustrated and motivated also by means of examples. In **Chapter 5** a design case of an hardware accelerator supporting an output stationary dataflow has been used as case of study. In particular, an Approximate Voltage Overscaling technique called *MAC-Drop* has been used as power management strategy under testing.

Chapter 2

Low Power Design Practice

As already mentioned in the introduction, this chapter presents the fundamentals of Low Power Design practice. Before diving into the details of the techniques used in the design flow, it can be useful to review the current state of Integrated Circuits technology.

2.1 State of CMOS IC Technology

The progress of technology nodes, together with the development of EDA tools and new computing architectures, led to a significant increase of number of transistors on the same die area. This is not only due to the effects of what was already predicted by Moore's law [1], but also to the ability of digital designers to create complex and high-performance circuits. The new architectures they developed, in fact, allowed for a complete exploitation of the improved technology nodes, thus increasing at the same time both the complexity and the efficiency of hardware capabilities. As the graph reported in fig. 2.1 shows, by neglecting the role of power consumption, the employed design strategies would soon have led to ICs with a power density greater than that of a nuclear reactor. The only solution is to include in the standard design flow **power consumption** as a fundamental design metric. As it should be clear by now, this is not only a must in energy-constrained embedded systems, but also in devices targeting the high-performance market.

It is important to notice that even if energy and power are correlated since power, in case of computing systems, is the rate at which energy is consumed, techniques reducing the power do not necessary decrease the energy consumption of the system. The choice between which metric has to be used depends on the specific application, context and device. For example, if the clock frequency is halved but the application execution time is doubled, the energy consumption remains quite the same, whilst dynamic power is reduced by half.

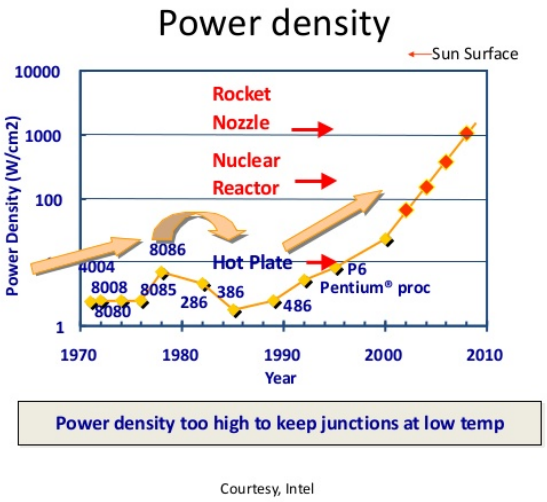


Figure 2.1: Power density vs Year in CMOS IC market.

2.2 Power Consumption in current CMOS technology

In CMOS technology the power consumption can be decomposed as the sum of two components: **dynamic power** and **static power**.

2.2.1 Dynamic Power Consumption

Dynamic power results from the energy consumed by the activity of circuits, i.e. the change of a stimulus in case of combinatorial logic or the sampling process in sequential cells like flip-flop and latches.
It is possible to identify two sources of dynamic power consumption: short-circuit current and charging/discharging of capacitances.

$$P_{dynamic} = P_{sw} + P_{sc} \tag{2.1}$$

The switching power, P_{sw}, is related to the energy needed to charge/discharge the load capacitance, the wire capacitance and the self capacitance of the gate. Figure 2.2 depicts all the capacitances seen by a CMOS inverter.

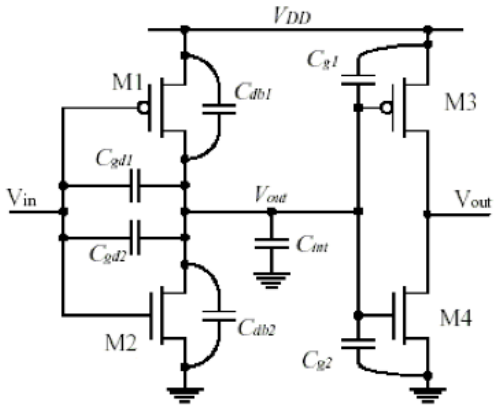


Figure 2.2: All the parasitic capacitances seen by a CMOS inverter driving another inverter.[2]

It is possible to express the switching power as:

$$P_{sw} = \frac{1}{2} \cdot V_{dd}^2 \cdot C_{load} \cdot f_{clk} \cdot e_{sw} \tag{2.2}$$

According to equation 2.2, there are four terms contributing to the power consumption, towards which the optimization effort should be directed:

- The load capacitance C_{load} depends on physical parameters of gates and wires.
- V_{dd} and f_{clk} are two design parameters and they are strictly related to the speed of the system.
- e_{sw} takes into account the activity of the node, i.e. the fact that, except for the clock tree network, not all the cells have a transition at each clock cycle.

The short-circuit power term, P_{sc}, is related to the current flow during the switching of the cell through both pull-up and pull-down stages.
CMOS technology, as the name suggests, is a complementary logic, so the pull-up and the pull-down stages should not be active at the same time. However, since the switching process is not instantaneous, i.e. the transition between ON-state and OFF-state is continuous, there is a current flowing through the cell.
The contribution of P_{sc} to the total power is lower than the one of switching power P_{sw}, but still it has to be optimized. In particular, the sizing of transistors is fundamental, as well as the relation between the transition time of input signal and output signal. As reported in [3], the matching between transition times of input and output signals is a rule of thumb for the overall short circuit current minimization.

2.2.2 Static Power Consumption

Usually the term static power refers to the current flowing in a stationary situation, i.e. when no switching is in progress. The static power consumption is expressed by:

$$P_{static} = I_{leak} \cdot V_{dd} \tag{2.3}$$

Ideally in CMOS technology the static power consumption should be equal to 0, since the complementary nature of the logic should have no static current flowing between the power rails. Actually, in real devices a static current is always present and can be caused by different physical phenomena [4]. Three main current contributions are:

- `Gate-oxide leakage`: The advancement of processing node has caused a reduction of the gate oxide thickness, thus an increased electrical field across the oxide. This causes a leakage current due to tunneling effect.
- `pn Junction Reverse-Bias Current`: The leakage current flowing due to the reverse biasing condition of drain and source to well junctions.
- `Subthreshold leakage`: The simplest model of a MOS is the one representing the connection between drain and source as an ideal switch controlled by the voltage applied to the gate, V_{gs}. Actually in MOS transistors it is possible to have a drain-source current even if the gate voltage is under threshold. This is called subthreshold leakage current.

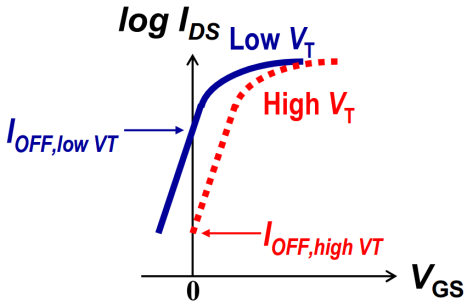


Figure 2.3: $I_{DS}(V_{GS})$ for different values of V_{th}.

As reported in fig. 2.3 [3], the lower the threshold voltage V_{th}, the higher the current I_D value when V_{gs} is equal to 0. This means that also static power consumption will be higher. This relation is of fundamental importance to understand the key role that leakage power consumption has in deep-sub-micron technology. Indeed, scaling the technology node leads, as reported also before, to higher transistor density and an increase in the speed of gates, which in turn leads to lower clock period. With a higher switched capacitance and clock frequency the dynamic power consumption, as well as the power density, increases such that not all the area of the chip can be active at the same time: this is usually called in literature **dark silicon phenomenon**.
The first approach used by IC designers to bring down dynamic power consumption was to reduce supply voltage. A simple model for the propagation delay of a cell is reported in equation 2.4.

$$D_p = C_l \cdot \frac{V_{dd}}{(V_{dd} - V_{th})^{\alpha}} \tag{2.4}$$

It is clear that, in order to compensate the effect of delay degradation due to reduced supply voltage, also the threshold voltage has to be lowered.

It is important also to notice that leakage current highly depends on temperature, therefore the working temperature of the silicon has to be consistently low. This can be achieved by a well designed packaging, a physical-design stage aware of the temperature issue, and of course by reducing the power consumption.

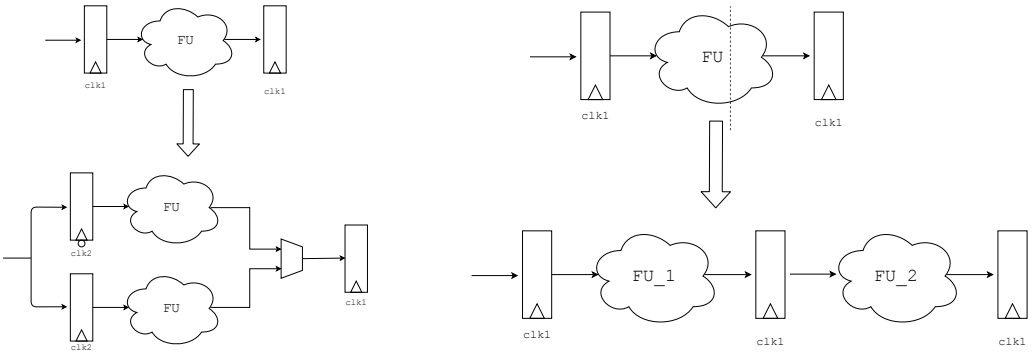
2.3 Power Optimization Techniques

Different techniques from architectural to transistor level have been proposed in order to optimize power consumption. In the next subsections a few examples at each different abstraction level have been reported.

2.3.1 Architectural Level

A first group of optimization techniques is applied to the architectural level. The main point is to change the architecture of the system to make it faster, then reduce the V_{dd} to return at the original speed. In this way the throughput remains the same but the dynamic power consumption has decreased.
Two possible architectural modifications are based on **parallelization** and **pipelining**. Be aware that in order to exploit this techniques some additional circuitry has to be employed, thus in some conditions the power consumption may increase.

The **parallel approach** consists in substituting a functional unit working at frequency f_{clk1} with two functional equivalent units working at frequency $f_{clk2} = \frac{f_{clk1}}{2}$ but on two different edges of the clock. An illustration of the methodology is reported in fig. 2.4a. It is possible to use the slack created by increasing the clock period lowering the supply voltage. This approach presents two main issues: a more than doubled area with increasing on leakage power consumption and a scalability issue. Indeed, it is not possible to reduce too much the V_{dd}, since when it approaches V_{th} the delays increase so rapidly that it is not possible to compensate them while at the same time reducing the power consumption.



(a) Parallelization

(b) Pipelining

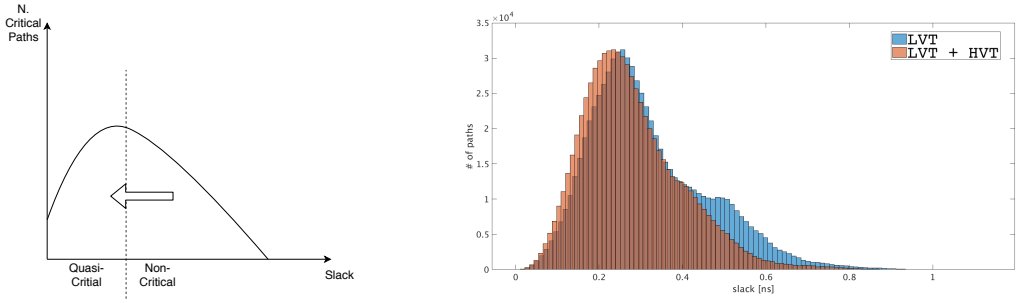
Figure 2.4: Architectural level dynamic power optimization.

Pipeline involves splitting a combinatorial circuit in more parts enclosed in a synchronous

environment. In this way each part has a worst case propagation delay lower than the total circuit, thus V_{dd} can be decreased to consume the available slack. Also this technique presents some drawbacks like the increased latency, more area occupied on the die and also scalability issues similar to the previous case.

2.3.2 Logic Level

Another group of optimization techniques is applied to the logic level. At this stage the view of the IC designer is focused on logic gates, thus the architecture of circuits has been already defined.



(a) Conceptual Image

(b) Dual-Vth Post-Synthesis Assignation.

Figure 2.5: Slack Redistribution for power optimization.

Dual-Voltage Gates. A first optimization for dynamic power consists in employing dual-voltage assignment to the logic gates. As the name suggests, this technique belongs to a wider set of solutions based on voltage scaling, like those reported in the previous subsection. As explained in subsection 2.2.1, both dynamic power consumption and propagation delay of a logic gate depend on V_{dd}. Generally in a circuit not all gates belong to critical or quasi-critical paths, thus it is possible to consume the slack available on gates belonging to non-critical paths by lowering their V_{dd}. This concept is depicted in the graph reported in fig. 2.5. Due to the high complexity of on-chip voltage regulators and power delivery network, however, the number of power supply voltages must be restrained. Usually two values are the best trade-off. Different algorithms as [5], [6] have been proposed for automatic assignation of the V_{dd} to the cell: in general they are heuristic methods based on some graph visit that iteratively try to find the partitioning of the netlist satisfying timing constraint with minimum power consumption. Figure 2.6 illustrates an example of circuit partitioned by the Clustered Voltage Scaling Algorithm for dual voltage assignment. The main drawback of this technique is the use of level shifters necessary to interconnect the tiles of die at different voltage, which of course generates overhead and complexity in the physical design.

Clock Gating. Another technique for dynamic power reduction is based on the observation that the outputs of a logic block are useful only under certain conditions. Thus, it is possible to disable the switching of the clock through the insertion of an activation

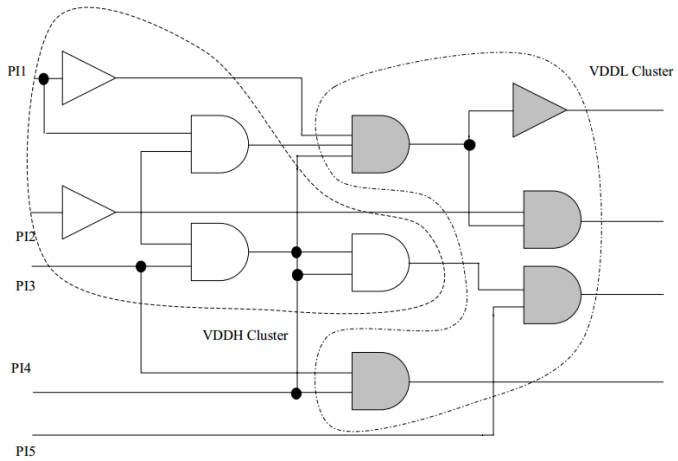


Figure 2.6: An example of circuit partitioned by the Clustered Voltage Scaling Algorithm for dual voltage assignment.

function which enables the clock gating. The effect of clock gating is not only a decrease of power consumption on the clock tree and on the register itself, but mostly a reduction of the switching activity of the combinatorial logic fed by the gated registers. Different ways of realizing the clock gating from a circuit perspective have been proposed, as reviewed in [7]. The most common used is *Latch Based And clock gating* reported in fig. 2.7, where the enable signal is applied through a latch before being conjuncted with the clock signal. The latch is employed to avoid hazard propagation and to reduce the degradation of clock signal rising and falling transitions.

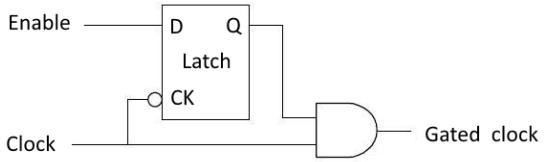


Figure 2.7: Latch And Clock Gating structure.

Power Gating. The idea behind power gating technique [8] is to reduce leakage currents of gates that are in idle state for a long period. This is achieved by inserting high V_{th} transistors in series with the pull-up and/or pull-down networks of a number of gates, in order to reduce the sub-threshold current when these transistors are deactivated. A conceptual representation of power gating is reported in fig. 2.8. Instead, when the sleep transistors are activated, the gates should continue to work normally, thus they have to be properly sized to reduce the delay penalty. The insertion of sleep transistors turns out to be a very complex design task: it requires different optimization algorithms for regulating cell clustering, sleep transistors sizing and distribution network of enable signals.

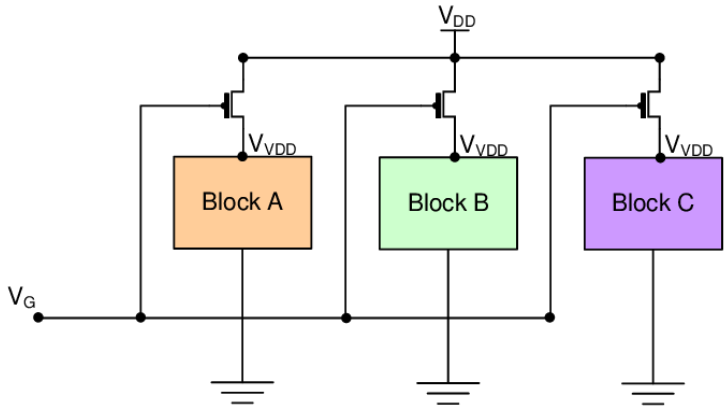


Figure 2.8: A conceptual representation of power gating.

Dual V_{th} Assignment. A technique to reduce leakage power consumption consists in using multi V_{th} cells. Nowadays silicon vendors provide technological libraries composed of high speed cells with high leakage power consumption and low speed cells with low leakage power consumption, which can be integrated in the same technological process. The basic idea is similar to the one used for dual V_{dd} assignment and reported in fig. 2.5, where the design is first synthesized and mapped onto all low-V_{th} cells, then the cells belonging to non-critical paths are replaced with the high-V_{th} version. Also in this case some heuristic algorithms are used, as the one proposed in [9], or in [10], where a simultaneous assignment of V_{th} and gate sizing reduces total power consumption.

2.4 Power Knobs

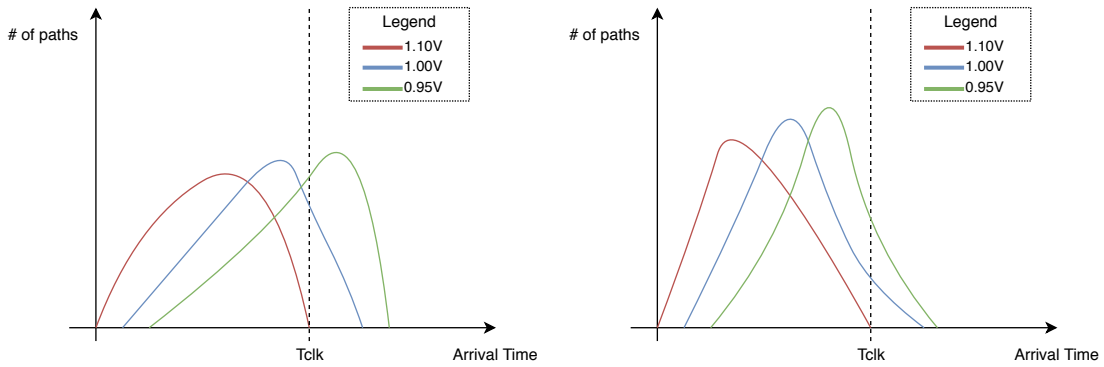
The common characteristic between the power optimization techniques reported in the previous section is the fact that they are applied at design time. Thus they can be defined static because the trade-off with the other design metrics is defined once during the construction of the IC and cannot be changed at run-time. Another approach instead is to change dynamically the power consumption of the circuit during its operation, by acting on the figures of merit which directly affect power and energy. This is an interesting scenario since, as reported in 2.1, the same system is placed in a multi-context environment where the non-functional requirements can vary consistently. For this reason the design of a system in the worst case scenario turns out to be highly inefficient when the circuit is operating in the other contexts. The online modification of system operating conditions is achieved by means of knobs which act on V_{dd} and f_{clk} for regulating the power consumption, but at the same time also throughput and quality of result. This power knobs can be managed with a control system that, based on information sensed from the circuit itself and from the environment, tries at run-time to find the best trade-off. In the following subsections three major dynamic power optimization techniques will be presented.

2.4.1 Dynamic Voltage and Frequency Scaling

The first technique presented is Dynamic Voltage and Frequency Scaling. The system is characterized in order to work at different fixed combinations of supply voltage and frequency value. As explained in [11], DVFS technique has proven to be effective at achieving low power consumption while fulfilling performance requirements. Unfortunately in deep-sub micron technology the efficacy of this technique has been lowered due to the increased importance of leakage power consumption. DVFS consists in dynamically scaling V_{dd} and f_{clk}, in order to reach the condition where average circuit speed is consistent to meet total computation time and/or throughput requirements, but at minimum energy.

2.4.2 Dynamic Voltage Scaling using Razor-FF

The idea of "just-enough" energy is pushed even further through dynamic voltage scaling using timing sensors, i.e. a logic circuit able to sense timing violations. In this case only the V_{dd} is used as power knob, whilst the clock frequency is not modified, thus timing violations due to the dependency of cells delays on power supply voltage can occur. At this point an important consideration must be pointed out. The working frequency of the system is based on the timing of critical paths without any consideration on the dynamic behavior of the system: this means that the actual number of times that the paths are synthesized is not taken into account. From a static point of view this could lead to think that the number of timing violations is huge even with a small decreasing in the V_{dd}, as the number of paths with an arrival time greater than the required one is extremely large. Actually in common circuits and under realistic workload this does not occur. As it is well represented in fig. 2.9, by weighting the number of paths with respect to the sensitization frequency, it is possible to lower the V_{dd} still keeping the majority of paths under the required time. In fig. 2.9b it is reported the dynamic distribution under different V_{dd}s of a 32bits multiply-and-accumulate unit: it is clear that up to the wall-of-slack, i.e. the point where the majority of paths cross the required time barrier, the circuit can continue to work correctly in most cases with a much lower power consumption.



(a) Static Path Arrival Time Distribution (b) Dynamic Path Arrival Time Distribution

Figure 2.9: Static vs Dynamic Path Distribution under different power supply voltages.

One of the most important and diffused in-situ timing sensors is Razor, proposed in [12]. Razor is a method of error detection and correction in the wider range of measure and control strategies applied to IC in the last years, [13], [14], [15]. The idea behind Razor is illustrated in fig. 2.10. A single flip-flop is augmented with a shadowed latch which is

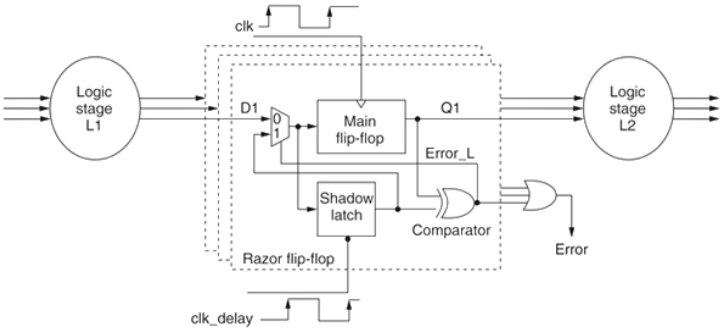


Figure 2.10: Razor-FF logic diagram, [12].

controlled by a delayed clock. The flip-flop is sensitive to the rising edge of the clock, while the shadowed latch is transparent on the low state of the delayed clock. In this way, when the input signal is late with respect to the clock, the shadow latch is still transparent, thus it can latch the correct value. When the latch goes in memory mode, the flip-flop and shadowed latch outputs can be compared to know if a mismatch has occurred. In order to avoid spurious transitions of the error flag, it must be ensured that there is no path with a delay that is smaller than the detection window. This is called short-path padding and it is equivalent to hold-time fixing which can be imposed at synthesis time as a minimum delay constraint. An example of timing diagram when an error is detected and corrected by Razor is reported in fig. 2.11. This error flag can be used as trigger for an error recovery

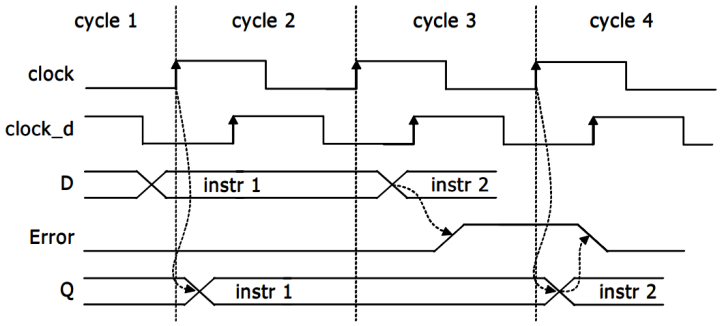


Figure 2.11: An example of timing diagram: an error is detected and corrected by Razor, [12].

mechanism, allowing the entire system to keep producing always-correct results. Possible error recovery mechanisms are:

- clock gating on the pipe, to make stall of previous and next stage.

- Architectural replay. For example, in modern μPs flush mechanisms for branch misprediction, speculative instruction and exception management are already implemented, so that they can be used also to deal with error recovery state.

Clearly the recovery from a timing error implies losing a clock cycle, but also wasting energy due to the invalidation of computations already performed. However, if the recovery state is rare, then lower power consumption is achieved at the cost of a slightly inferior throughput of the system. If the number of errors is high enough to excessively degrade the timing performance, then the error control unit communicates with power unit to increase the V_{dd}. Substantially the aim is to make the system work at the minimum voltage with an acceptable average throughput.

2.4.3 Voltage Scaling for Error-tolerant Applications

The last type of dynamic power optimization, differently from those introduced before, exploits the quality of results as an additional figure of merit. In certain applications, for example those dealing with human sensing like image, video and music processing, it is possible to design non-always correct systems, i.e. employ some approximations in order to improve other properties of the system. The traditional approach is called **approximate computing** and it usually consists in designing mathematical operators that differ from the ideal ones in their functional behavior. For example, it is possible to have an approximate adder with a bounded error while performing addition compared to an exact one, but with smaller area or higher throughput. As reported before, in multi-context embedded systems also the requirements on the quality of results, thus on the error magnitude of the approximation, can be variable. The circuit-level design of an operator with configurable approximation is challenging [16]. For this reason another possible approach is to use V_{dd} as both power and approximation knob. Two very effective techniques are: Adaptive Voltage Over-Scaling and Dynamic Voltage Accuracy Scaling.

Adaptive Voltage Over-Scaling. The architecture of this technique is similar to 2.4.2, where voltage is lowered without changing the frequency. Actually, Razor is here used only as an error detection strategy, since no error recovery mechanism is employed. The error flags generated by the Razor, as done in [17], can be accumulated in a counter and a threshold based mechanism can be used to regulate the scaling of V_{dd}. The mechanism reduces at minimum the overhead introduced, and the trade-off between energy and quality of results can be easily controlled by simply regulating the error threshold.

Dynamic Voltage Accuracy Scaling. Another possible approach for approximate computing consists in introducing intentionally quantization errors at run-time by reducing the precision of arithmetical operators [18]. Thanks to bitwidth reduction two main effects can be exploited from a power consumption point of view:

- `Reduction of circuit activity.` The first effect of gated inputs is a drastic reduction of switching activity. The relationship between the reduction order of switching activity and bitwidth depends on the circuit architecture.

- `Voltage over-scaling.` Reducing the bitwidth of the circuit causes also a shortening of the critical paths. The generated slack can be consumed through a V_{dd} scaling without any frequency reduction. An example of a scalable array multiplier is reported in fig. 2.12. It is possible to see how, by using only the two most significant bits, the critical path becomes shorter than the full bitwidth circuit.

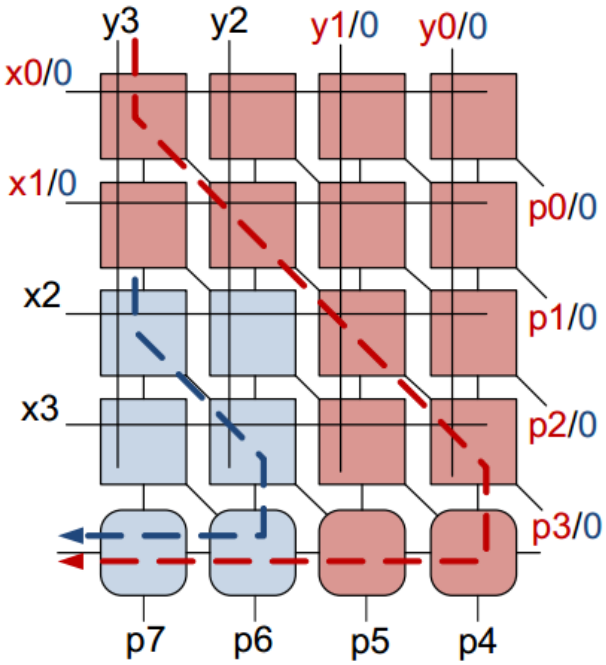


Figure 2.12: Bitwidth reduction in array multiplier, [18].

Chapter 3

Hardware Accelerators for Neural Networks

3.1 Background Neural Network

Machine learning is the field of computer science studying algorithms allowing machines to sense complex information from raw data and learn specific tasks without being specifically designed for that, [19].
Neural Network is a specific area of machine learning. It can be defined as a huge parallel system made up of interconnected elementary processing units with some parameters that are configurable through a process called learning, [20]. From a mathematical perspective it is represented by a graph $G = \{V, E\}$ where the vertexes are called neurons and the edges weights. It is inspired by the human brain, where the neurons are highly specialized biological cells able to transmit and receive information through electrical and chemical processes.
A more quantitative definition of the learning process can be formulated as follows. The neural network has to perform a task T learning by a set of samples D, thus a score function S is introduced to estimate the goodness of the network at T. Learning means that the network is able to improve its score S by processing D.
Only in the last few years neural networks have reached a wide range of use with remarkable results, even if the first proposal dates back to 1943 [21]. This is attributable mainly to the availability of data set in different fields with million of samples, the big-data phenomenon, and also to the diffusion of powerful parallel hardware, in particular GPGPU with mature, stable and easy to use software libraries.
A few application examples of Neural Network are:

- `Image Classification`. To assign a label from a defined set to an input image.
- `Image Detection`. To draw a bounding box on each recognized object of the input image. Example reported in fig. 3.1a.
- `Segmentation`. To highlight the edges of each recognized object of the input image. Example reported in fig. 3.1b.

- `Speech Recognition`. To transform an audio sample of human voice in a text.
- `Text Translation`. To translate a text in different languages.
- `Decision-making`: To decide the next action, given the actual state of the environment and of the system.

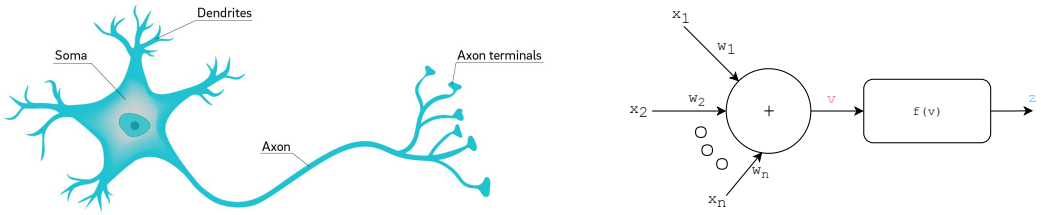
(a) Image detection

(b) Image segmentation

Figure 3.1: Examples of computer vision task performed by a neural network.

Biological Analogy. Roughly speaking, the biological neuron is composed of a nucleus surrounded by dendrites and an axon with its terminals, as depicted in fig. 3.2a. The axon terminal of a neuron is connected to the dendrites of the same or other neurons through the synapses. A synapse is a chemical/electrical connection able to modulate the strength of the communication between neurons. Different mathematical models of a neuron have been proposed in literature, but the most used in the artificial neural network field is the one called Rosenblatt perceptron, [22], depicted in fig. 3.2b. The idea is to have a model able to emulate that:

- the communication between neurons can be of different strength and also it can be both excitatory or inhibitory. Thus the use of weighted inputs.
- the neuron integrates the impulses and applies a firing function to determine its state. Therefore the use of a summing node and a non-linear activation function.



(a) Simplified view of biological neuron.

(b) Rosenblatt perceptron.

Figure 3.2: The perceptron model proposed by Rosenblatt and a simplified view of a biological neuron.

The mathematical representation of the perceptron is reported in eq. 3.1. The most common activation functions are *sigmoid*, *tanh*, and *rectifier*.

$$z = f(\sum_{k=0}^{n} w_k \cdot I_k) \tag{3.1}$$

A single perceptron can be used to linearly separate the multidimensional input space. Connecting multiple neurons to build complex network allows to manipulates the input in different ways, such that the tasks reported before can be accomplished. Figure 3.3 from [23] is an excellent pictorial representation of common neural network architectures built from the perceptron or other neuron model.

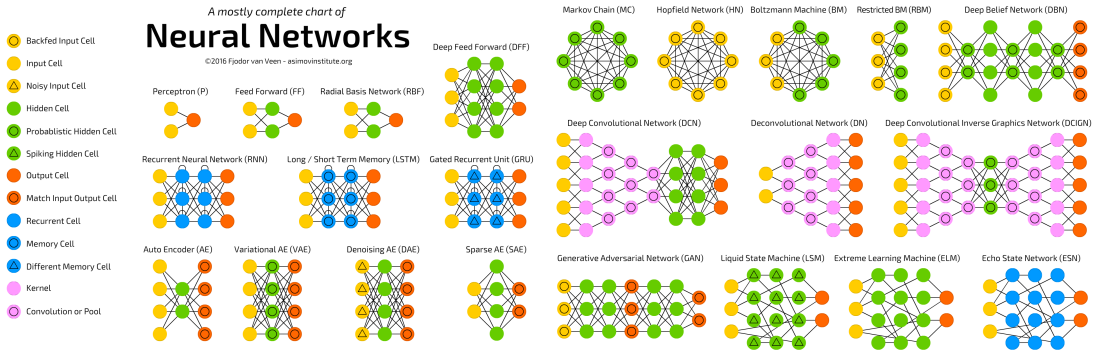


Figure 3.3: Neural Model Zoo [23].

Among all neural network architectures the most widely diffused are:

- `Feed-Forward Neural Network`: it is a multilayer perceptrons able to divide the multidimensional input space in different regions, thus it is used for classification purposes.

- `Deep Convolutional Neural Network`: as the name suggests, it is built by cascading many convolutional layers, even if it also involves the use of other kinds of layer in the final stage. The idea is to increase the number of layers in order to improve both capacity and generalization capability of the network. The convolutional part is usually called *feature extractor*, since it is employed to filter and group representative information from the raw data received as input. For example, when CNN are used for classifying images, the convolutional layers extract the features from input images, then a feed-forward network is used to perform the classification. In particular, some works on visualization and interpretation of DNNs [24], [25] have shown that in the first layers the network tries to learn how to recognize simple patterns like texture and geometric primitives. Instead, going deep in the network, the level of abstraction is higher, since the geometric primitives are combined into shapes and then into ensembles of shapes. At the end complex pattern like human mouth, animal ears, tires are recognized.

In order to correctly employ machine learning to solve a specific task it is necessary to perform a **training phase**. Training a neural network means setting the value of weights and bias for each neuron.
It is possible to identify three major categories for the training process:

- `Supervised Learning`. The training data is labeled and the network has to learn starting from input-output examples. A loss function is defined to compute the closeness of the output of the network to the ground truth, and in a certain way the error is back-propagated to tune the learnable parameters in order to mimic the correct behavior.
- `Unsupervised Learning`. The training is performed with any information about the ground truth. The objective is to find structures and patterns in the input data, or to learn statistical moments from the data samples that are seen as a representation of a stochastic process.
- `Semi-Supervised Learning`. It is a mixture of the previous two. It is usually employed when a huge unlabeled data set and few labeled examples are available.

From a system designer perspective it has to be considered that the training phase is usually done once and it is very resource expensive. The parameters update for large network requires multiple hours on power-hungry and powerful devices like GPGPU.

For the inference phase, instead, a promising option is to bring it on the edge device. In many applications security and reliability are major concerns, thus relying on the connection to perform the computation in the cloud could be a problem. Furthermore, nowadays a huge amount of devices is employed mainly to collect and send raw data to servers located in data-centers. However, in order to exploit cyber-physical systems at their fullest, it is necessary to provide these resource-constrained devices with the ability of *sensemaking*, i.e. the capability of extracting from raw data complex and valuable knowledge.

3.2 Computational view of Deep Neural Network

In order to understand what are the computational requirements of Deep-Neural Network, it is useful to introduce two famous architectures having different application domains.

AlexNet. Proposed by Alex Krizhevsky et others [26], it has been one of the first papers of the renaissance of NNets in the last years. It won the 2012 ImageNet LSVRC-2012 competition [27] with more than 10% improvement on top-5 accuracy over the second classified. The authors proposed a new way to deal with the computational complexity during the training phase exploiting:

- The highly parallel architecture of GPU.
- ReLu activation function instead of sigmoid or tanh..
- Dropout methods, i.e. a stochastic way to avoid over-fitting through the temporary elimination of some weight.

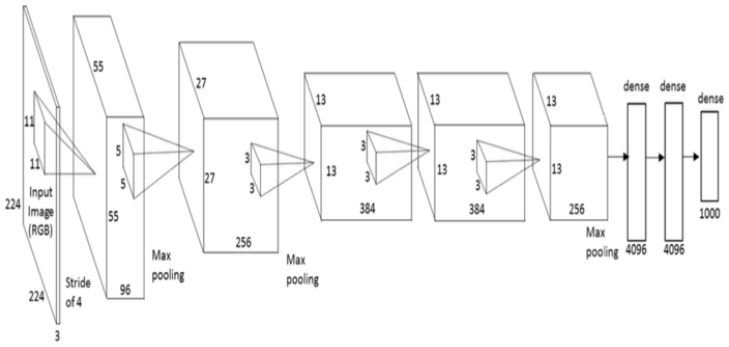


Figure 3.4: AlexNet: Convolutional Neural Network for classification task.

As shown in fig. 3.4, it is composed of five convolutional layers followed by three fully-connected layers. Actually the convolutional layers are composed of 2D convolutional filters, max-pooling and ReLu activation. It is important to notice that the feature extractor part occupies about 95% of the total execution time, while the classifier has a large impact on the memory footprint.

PyDnet. Proposed by Poggi and others [28], it is a neural architecture optimized for CPU inference that extracts depth maps from single-camera images. It has been proposed for embedded applications and it allows to obtain results similar in terms of accuracy to state-of-the-art for monocular depth estimation. The architecture is shown in fig. 3.5. It contains three main blocks:

- `Pyramidal Feature Extraction`: six couples of convolutional layers aimed at downsizing the image from 1/2 to 1/64 of the initial resolution. It acts as a decoder stage.

- The obtained feature map at each resolution is processed by an `estimator` composed of four convolutional layers.

- A `transposed convolution layer` to upsize the feature map from lower resolution in order to be processed by the estimator of an higher resolution layer together with the pyramid output. This allows to speed-up the training and reduce the number of parameters, i.e. number of weights, and also the number of computations required to perform the inference.

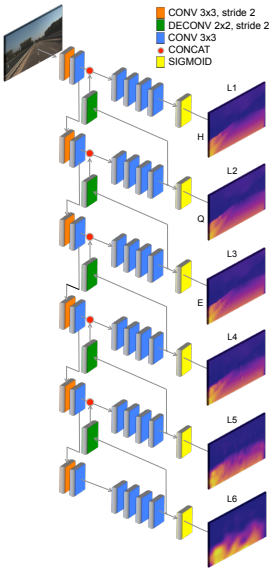
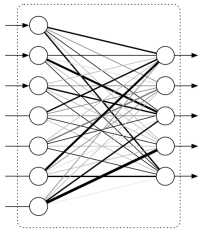


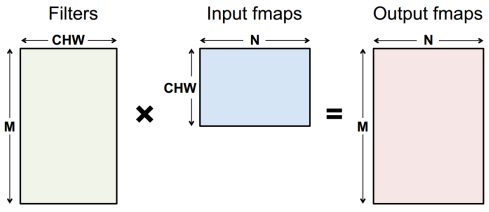
Figure 3.5: PyDnet: a neural architecture for unsupervised monocular depth estimation optimized for CPU platform, [28].

The previous neural network architectures are used as examples to analyze which are the common layers employed in state-of-art neural networks, thus to understand the computational effort of each layer and how they can be supported on different hardware platforms.

Fully-Connected Layer. From a computational point of view it is possible to describe the fully connected layer as a matrix multiplication. For these reason it is possible to use many optimized algorithms developed for high-performance linear algebra packages, [29]. A fully connected layer of m output neurons and n input neurons, represented in fig. 3.6a, can be formalized as follow: $W \in R^{m,n}$ is the weight matrix, $I \in R^{n,1}$ is the input activation tensor and the output tensor is obtained by $O = W \cdot I$, with $O \in R^{m,1}$. Usually the matrices involved are large, thus fully-connected layers have a huge memory occupation and require high bandwidth. Figure. 3.6b depicts the computational perspective of a batched fully-connected layer.



(a) Illustration of a fully connected layer based on neurons and synapses.



(b) Computational point of view of Fully-connected Layer, [30].

Figure 3.6: Fully Connected Layer.

Convolutional Layer. A convolution layer transforms the input feature maps into the output maps through multiple 3D filters called kernels. The layer performs the convolution of the filters across the three dimensions width, height and depth of the input feature map. Many experiments, like [31], state that convolutional layers take up most time of computation volume in both inference and training phases on both CPUs and GPGPUs. The computation of a convolutional layer can be configured specifying the following parameters:

- Shape of input and output tensor.
- Size of the kernel.
- Stride and Dilation of the 2D convolution.
- Padding of the input tensor

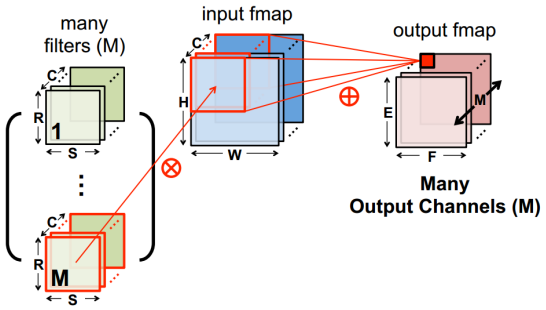


Figure 3.7: Representation of the computations involved in the execution of a convolutional layer, [30].

The pseudo code describing a convolutional layer with zero padding, stride and dilation set to 1 is reported in list. 1. Fig. 3.7 is a pictorial representation of the computations involved.

```
for ( m = 0; m < n_output_channels; m++ ) {        // Output channels
  for ( n = 0; n < n_input_channels; m++ ) {       // Input channels
    for ( r = 0; r < dim_y_output; r++ ) {         // Ouput neuron y coordinate
      for ( c = 0; c < dim_x_output; c++ ) {       // Output neuron x coordinate
        for ( i = 0; i < dim_y_kernel; i++ ) {     // y coordinate of the kernel
          for ( j = 0; j < dim_x_kernel; j++ ) {   // x coordinate of the kernel
            O(m,r,c) += I(n,r+i,c+j)*W(m,n,i,j)    // MAC operation
        }
      }
    }
  }
}
```

Listing 1: Convolution Pseudo Code

The reason why convolutional layers are so expensive, in terms of execution time, is not only related to the large number of computations, but also to the high number of memory accesses, both read and write of operands, partial and final results. In the case of the code reported in 1 having an input tensor with dimensions $c_{in} \cdot y \cdot x$ and a weight tensor with dimensions $c_{out} \cdot c_{in} \cdot k_y \cdot k_x$ the number of operations is: $\Theta(c_{out} \cdot c_{in} \cdot (y - k_y + 1) \cdot (x - k_x + 1) \cdot k_y \cdot k_x)$. For example CONV2 of AlexNet requires $\approx$ 2Gops. Two main elements must be considered in order to optimize the convolutional operation: the high data-reuse and the exploitable parallelism. It is clear that there are multiple sources of data-reuse:

- `Weights Reuse`: The same weights are reused multiple times on each 2D plane with different activation values.
- `4D Weight Reuse`: Even if this is exploitable only in specific cases like training and applications where an high latency is acceptable, in case multiple input tensors, i.e. a batch of inputs, are available then the entire weight tensor is reused for each element of the batch.
- `2D Activation Reuse`: Since the kernel 2D window of dimension $k_x \cdot k_y$ is slided across the input plane, the same activation will be present in multiple point-wise multiplications.
- `3D Activation Reuse`: Each output channel is obtained through the convolution of different weights with the same 3D input tensor, thus the reuse of all input values across multi-output computation.

For what concerns the parallelism, as reported in [32], it is possible to point out a basic taxonomy based on how the six loops, reported in 1, are unrolled. Among all the possibilities, three of them are reported below, since they have been used as mapping strategies in many hardware accelerators developed in the last years.

- Synapses parallelism: Multiple synapses are executed in parallel, i.e. the multiplications between weights and activations, while a single input feature and a single output neuron is considered. The pseudo code is reported in 2.

```
1                  ......................................
2  for ( i = 0; i < dim_y_kernel; i+=Ti ) {              // y coordinate of the
   ↪  kernel
3    for ( j = 0; j < dim_x_kernel; j+=Tj ) {            // x coordinate of the
   ↪  kernel
4      parallel for ( ti = 0; ti < Ti; ti++ ) {          // y coordinate unrolled
5        parallel for ( tj = 0; tj < Tj; tj++ ) {        // x coordinate unrolled
6          O(m,r,c) += I(n,r+i+ti,c+j+tj)*W(m,n,i+ti,j+tj) // MAC operation
7        }
8      }
9    }
10 }
11                 ......................................
```

Listing 2: Single Feature Single Neuron Multiple Synapses

- Feature parallelism: Multiple input and output channels and output are processed in parallel, but with only one output neuron and one synapse at a time.

```
                .....................................
for ( m = 0; m < n_output_channels; m++ ) {           // Output channels
  for ( n = 0; n < n_input_channels; m++ ) {          // Input channels
    parallel for ( tm = 0; tm < Tm; tm++ ) {          // y coordinate unrolled
      parallel for ( tn = 0; tn < Tn; tn++ ) {        // x coordinate unrolled
        O(m + tm,r,c) += I(n,r+i,c+j)*W(m+tm,n+tn,i,j) // MAC operation
      }
    }
  }
}
                .....................................
```

Listing 3: Multiple Feature Single Neuron Single Synapses

- Neuron parallelism: Multiple output neurons are mapped on different processing elements working concurrently.

```
                .....................................
for ( r = 0; r < dim_y_output; r+=Tr ) {          // Ouput neuron y coordinate
  for ( c = 0; c < dim_x_output; c+=Tc ) {        // Output neuron x coordinate
    parallel for ( tr = 0; tr < Tr; tr++ ) {      // y coordinate unrolled
      parallel for ( tc = 0; tc < Tc; tc++ ) {    // x coordinate unrolled
        O(m,r,c) += I(n,r+i+tr,c+j+tc)*W(m,n,i,j) // MAC operation
      }
    }
  }
}
                .....................................
```

Listing 4: Single Feature Multiple Neuron Single Synapses

Once the parallel structure has been decided, it is possible to optimize the computational kernel to exploit data-reuse. In temporal architectures like CPU and GPGPUs this means managing the memory layout and the memory access patterns in order to be as cache-friendly as possible. On the contrary, in hardware accelerators, the data-reuse is exploited by sharing operands and partial results between processing element, thus implementing a spatial architecture with optimized network-on-chip.

Deconvolutional or Transposed Convolutional Layer. When the neural network is used to generate images like the depth map of PyDnet, or the mask applied to the inputs to accomplish segmentation tasks, there is the need for a computational kernel able to up-sample from low-resolution to a higher resolution. The deconvolutional layer performs an up-sampling process, but with learnable parameters. The idea is to go in the other direction compared to a convolution, in particular to reverse the many-to-one mapping resulting from the convolution kernel. The main computational kernel implementing deconvolution is the

one based on the transposed convolutional matrix, resorting in a matrix-multiplication, and the one based on up-sampling the input tensor through zero insertions and then performing a convolution.

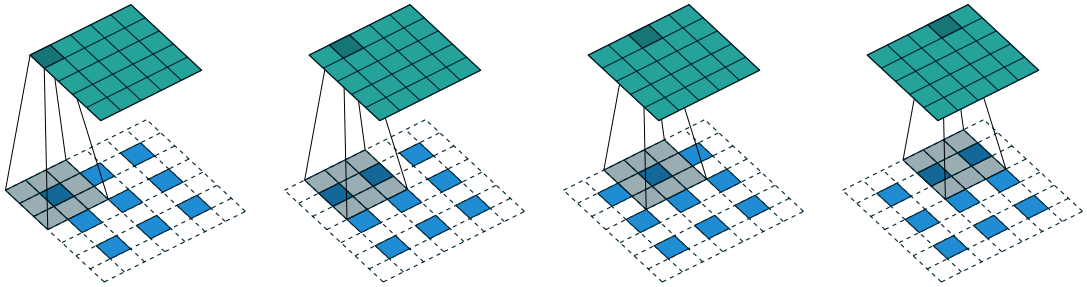


Figure 3.8: Representation of a deconvolutional layer with a 5×5 kernel. Credits to [33].

Pooling Layer. The pooling layer is used to reduce the spatial dimension of the feature maps in order to lower the number of computations, but also to avoid overfitting on training data. It is applied to each channel of an activation tensor as a sliding window and a reduction function like max or average. It has almost the same configurable parameters of a convolutional layer.

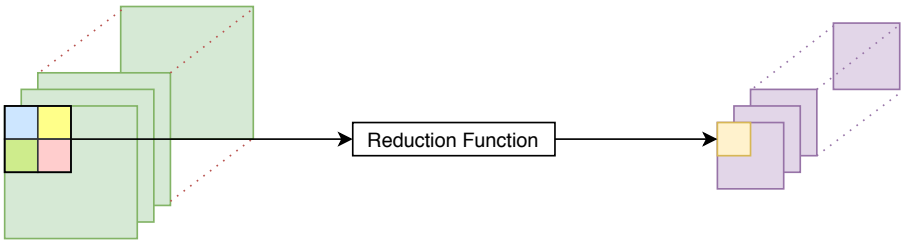


Figure 3.9: Representation of a pooling layer.

Activation. The activation layer is simply computed through a point-wise function. For performance reason it can be fused with the computation of the previous layer in order to avoid useless data movement in memory hierarchy. The most common used activation functions are:

- `Relu`: $y = max(0, x)$
- `Leaky Relu`: $y = step(x) * x + step(-x) * \alpha$
- `Sigmoid`: $y = \frac{1}{1+e^{-x}}$
- `Tanh`: $y = \frac{1-e^{-2 \cdot x}}{1+e^{-2 \cdot x}}$

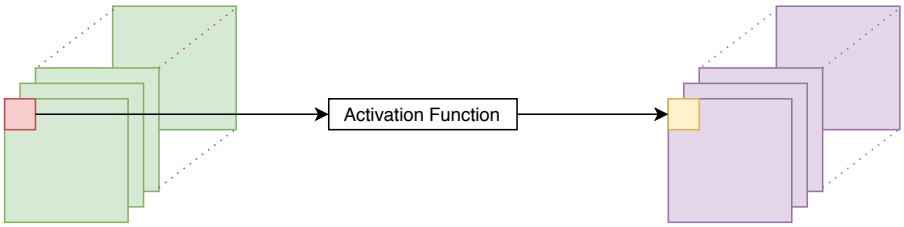


Figure 3.10: Representation of an activation layer.

In the general case when possible they are computed with a polynomial representation, thus employing only multiplication and addition, or through a look-up table.

3.3 Hardware Accelerators

This section gives an overview of three energy-efficient deep-learning accelerators.

3.3.1 Eyeriss

Eyeriss has been designed and implemented on TSMC 65nm LP technology in 2016 by MIT researchers of the Energy-Efficient Multimedia Systems Group, [34].
They introduced a taxonomy of previous hardware accelerators based on the implemented data-reuse methods. The three basic strategies are:

- `Weight Stationary`: the weights are stored locally at each processing element and they stay stationary as much as possible. The input features are broadcasted to the processing elements, while partial results are accumulated spatially.

- `Output Stationary`: each processing element is in charge of at least one output neuron. This means that the partial results stay stationary at each processing element. Weights are broadcasted or multicasted from the on-chip memory, then reused together with the activations between neighboring processing elements.

- `No-Local Reuse`: particularly attractive for hardware with less flexibility on placement of memory near computational logic or with a very large number of computational nodes.The access to local buffer is maximized, thus both activations and weights are multicasted to the processing elements and the partial results are accumulated spatially.

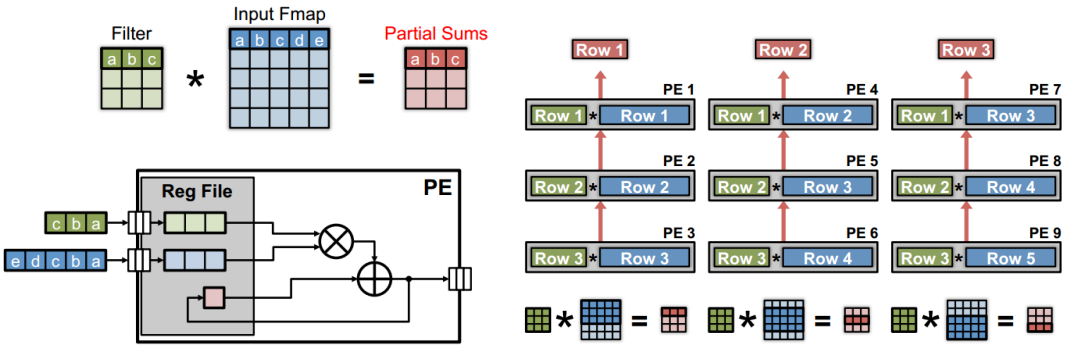


Figure 3.11: Row Stationary proposed in [30].

They proposed a mixed dataflow which tries to maximize the reuse of both weights and activations, called **row stationary**. Fig. 3.11 depicts how the operands are assigned to each PE and also the data reuse in the computational grid. In particular, at each processing element a row of weights and of input feature map is assigned. The partial results are then collected vertically over the grid of processing elements. In this way it is possible to have a simple yet efficient Network-on-Chip, since rows of the weight tensor are broadcasted over the rows of the processing element grid, the partial sums accumulated vertically and rows of the activation tensor are broadcasted on the diagonals of the processing element grid.

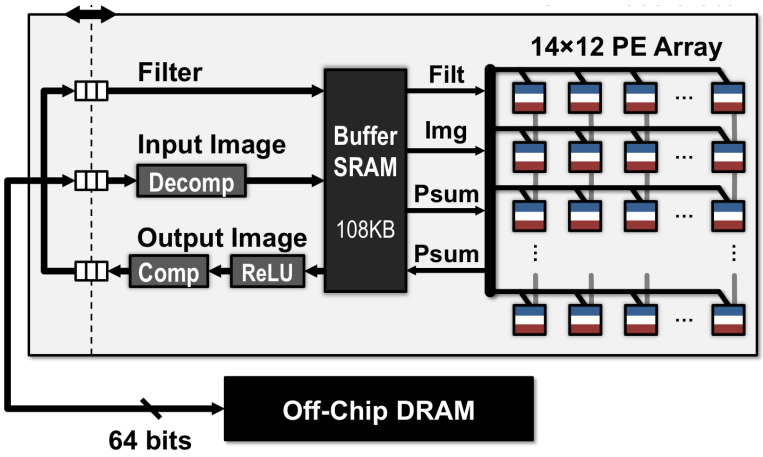


Figure 3.12: Architectural representation of Eyeriss, a CNN accelerator proposed by V.Sze and others at MIT, [34].

Fig. 3.12 depicts a macro view of Eyeriss architecture. Each Processing Element has an energy-efficient register file with a dimension of 0.5KB, a MAC unit and a local control unit. The NoC provides connection between different processing elements, in particular there are multiple common data buses with control signals to specify which are the processing elements involved in the communication. The implemented connections allow unicast and multicast communications from the global buffer and the processing elements. The global buffer is implemented as an SRAM in order to reduce the energy cost of the accesses, but

this implies some constraints on the total size, for example in the chip fabricated it was set to 108KB. For this reason, there is a DRAM memory as large storage space. Two clock domains are present, one for the DRAM and another for the on-chip SRAM and PEs grid. The synchronization between different clock domains is performed through FIFOs. The choice of having multiple clocking domains allows to better explore the energy-throughput trade-off, enabling at the same time the integration of the accelerator in a System-on-Chip.

3.3.2 Efficient Inference Engine

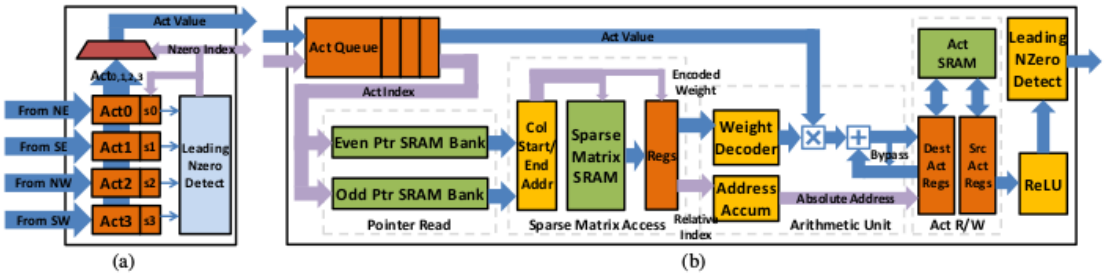


Figure 3.13: Architectural representation of EIE, a fully-connected accelerator optimized for sparse NNets, [35].

Efficient Inference Engine, EIE, is an an accelerator for compressed neural network proposed by Song Han and others at Stanford University [35]. In particular, it accelerates sparse matrix-vector multiplication in an energy efficient way. The main motivation behind EIE is to design a hardware platform able to exploit pruning and weight sharing, two energy-aware optimizations performed on the neural network model. The weight matrix is represented in a compressed sparse column format, i.e. each column is stored in memory through two vectors, one for the non-zero values and another to encode the number of zeros between the elements. For example the vector [0,7,8,0,0,0,9] is encoded through $v = [7,8,9]$ and $z = [1,0,3]$. Since all the sparse columns are stored in a memory, a vector of pointers to the first element of each column is needed. The block diagram of the processing element architecture is reported in fig. 3.13.
The matrix W and the activation vector a are distributed across all the processing elements such that PE_i stores all the rows W_j with $j \ mod \ N = i$[1]. The aim is to maximize the efficiency given by both activation and weight sparsity. In particular, the non-zero activations with the relative index are broadcasted to all PEs and stored in the *input activation queue*. The index is used to retrieve the pointers to the start and to the end of the sparse column assigned to each PE and stored in the *sparse matrix SRAM*. The sparse matrix SRAM contains the tuple <v,x>, where v is an index to the 16bits fixed-point weight, while x is used as index in the accumulation array stored in the *activation r/w unit*. The

[1]N is the total number of processing elements.

activation r/w unit is composed of two register files, such that one contains the source activations and the other the destination activations, but, after an entire matrix multiplication is performed, the roles of the two can be swapped, in order to support multi-layer neural networks without any additional data movement. After the ReLu operation is executed, the activation vector is compressed and distributed again to the PEs through a *distributed leading non-zero detection* unit. The accelerator is controlled by a *central control unit* which generates control signals and manages the communication with the host CPU.

3.3.3 Tensor Processing Unit

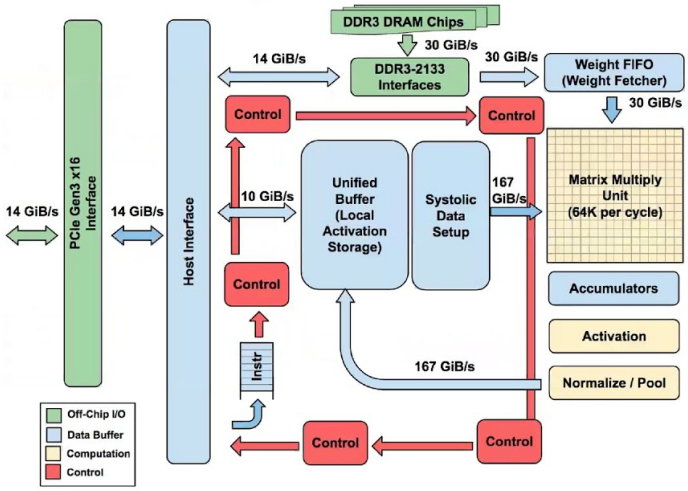


Figure 3.14: Architectural representation of the tensor processing unit, accelerator proposed by Google to enhance data-analytics in data center, [36].

The Tensor Processing Unit is designed to be a powerful co-processor improving performances of machine-learning workload in Google servers, [36]. The device has been silicon fabricated and mounted on a PCB with a PCIe I/O bus to be easily connected to already existing servers. A block diagram of the TPU architecture is reported in fig. 3.14.
The TPU computational heart is the matrix multiply unit, a huge 256x256 systolic array. The instructions are streamed from the host CPU and stored on a on-chip *instruction buffer*. The weights are stored in the *DDR3 DRAM* since they are loaded once for each input batch and they can be prefetched into the *Weight FIFO* while a computation is in progress on another set of weights. The *unified buffer* is allocated for the activation matrices. Since it is accessed from both the host CPU and the matrix-multiply unit, the unified buffer is a fast and large memory with an high speed $167GiB/s$ bus for the matrix multiply unit and a $10GiB/s$ for the host CPU. The results of the matrix-multiply unit, stored in the accumulator queues, are firstly processed by the *activation pipeline*, an arithmetical and logical unit executing the activation function, then stored back in the unified buffer.
A CISC ISA has been designed from scratch specifically for the TPU.
The main instructions are:

- `Read_host_memory`: it performs the data transfer from the host shared memory to the internal on-chip buffers called Unified Buffer.
- `Read_weights`: it is used to read weights from the read-only off-chip DDR3 DRAM to the Weight FIFO.
- `Matrix_multiply`: it starts the execution of the matrix multiplication.
- `Activate`: it triggers the execution of the activation/pooling function on the accumulator queues.
- `Write_host_memory`: it starts a memory transaction from the unified buffer to the host shared memory.

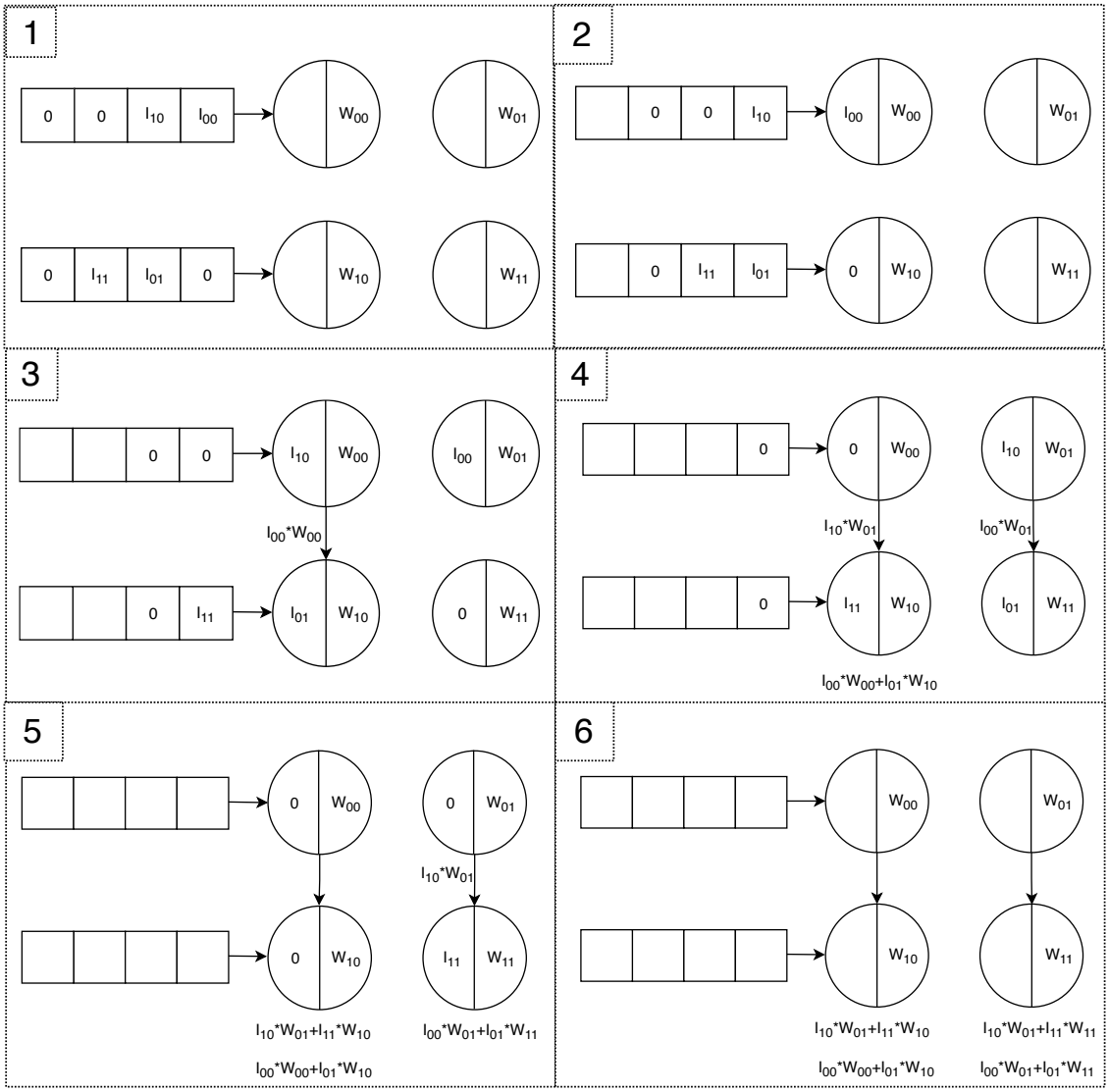


Figure 3.15: Matrix Multiplication 2x2 performed on a systolic array.

The heart of the TPU is the matrix multiply unit implementing a huge **systolic array**. Figure 3.15 depicts how a multiplication of 2x2 matrices is carried on a systolic array. Basically the systolic array is a grid of fused multiply-and-accumulate units interconnected to accumulate partial sums on the vertical dimension and forward input activation to the right neighbor.
At each clock cycle, a single MAC unit performs the following actions:

- Multiply the weight stored locally with the input coming from the left or from the input queue, in case of units located on the left border.
- Add the product to the partial sum coming from the above MAC unit, only in case of units not located on the top border of the grid.
- Send input activation to the processing element located on the right, if present.
- Forward the partial sum to the MAC unit located below, while, in case of units located on the bottom edge of the grid, the partial sum is stored in the activation queue.

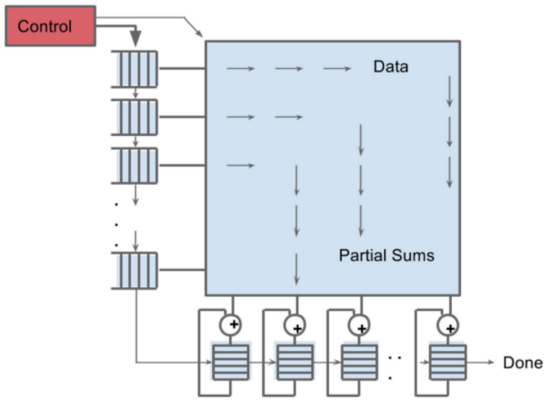


Figure 3.16: TPU matrix-multiply unit, [36].

Fig. 3.16 reports the direction of data movement together with the input queues on the left border and the accumulation queues located at the bottom of the systolic array. The weights, which stay stationary at each node of the grid, are sent during an initial configuration phase.

Chapter 4

The CoSimulation Framework

The chapter presents the contribution of this work: a co-simulation framework for the evaluation of power management in early phase of design of spatial architecture for deep-leaning on chip.

4.1 CoSimulation key aspects

As explained in chapter 3, hardware acceleration of neural network workload is fundamental to achieve energy efficiency and acceptable throughput in embedded systems near the sensors.
The wide range of neural network use, together with the deep theoretical effort that this subject has experienced in the last years, makes urgent a joint work between machine learning experts and hardware designers to develop new computing architectures able to satisfy the requirements of realistic applications, and at the same time to unlock the potential of deep-learning.
Even if this Deep Learning computing paradigm is highly application specific, flexibility and programmability are still needed to support multiple networks, but also to efficiently meet the peculiarities of each layer within the same network. Furthermore the requirements on the quality of results could be dependent on the environmental conditions in which embedded systems operate. All these sources of variabilities can be exploited by an adaptive power management control system that, using power knobs like supply voltage, clock frequency and body biasing, can change at run-time the working point in terms of energy, throughput and quality of result.
Performing a gate-level-simulation of the entire accelerator to understand the available degrees of freedom and to explore the energy/throughput/accuracy trade-off is unfeasible both in terms of simulation time and time-to-market. Moreover, waiting for the complete micro-architecture before developing and testing power management strategies may be truly inefficient, since early analysis could lead to a more performing accelerator.
The objective of this work is to meet these demands with a co-simulation framework able to:

- Evaluate the effective energy efficiency of realistic workload of spatial accelerators avoiding the simulation of the entire HW.

- Explore the design space to evaluate pros and cons of the designated HW architecture and power-management strategy.
- Enable an early efficiency testing of the neural netowork architecture thanks to an accurate estimation of the energy profile of the real hardware platform.

4.2 Tool Overview

The tool has been designed in order to be easily interfaced with common frameworks for machine learning and with the industrial ASIC design flow. The general philosophy behind the co-simulation framework is to have a behavioral neural network inferential engine that communicates with a gate-level simulator: this framework can provide stimuli to the circuit, collect responses and status signals and modify the configuration of power knobs. The aim is to simulate the system also from a non-functional perspective, thus the need for a gate-level simulator, but with only the minimum hardware required to verify the impact of a specific power management strategy on the network accuracy. In particular, the effect of power knobs on the system is emulated through a library of SDF files, one for each working condition, which can be loaded by the gate-level simulator when a power-context switch is performed.
QuestaSim, a commercial tool by MentorGraphics, is used as gate-level simulator. It allows the simulation of VHDL, Verilog, SystemVerilog code, but it provides proprietary APIs called FLI to interface the simulation kernel with imperative C code. The foreign language code is compiled and built as a shared library which is loaded at the startup by QuestaSim. The back-end initializes all internal data structures based on the information parsed from the configurations file, it emulates the modules that are not described in hardware and implements the power management control system. Since the software emulation is completely hardware independent, it has to mimic only the functionality and it can be written efficiently in C. Concurrently, all the other gate-level components are instantiated, managed and simulated by QuestaSim.
Synopsys Design Compiler is used for synthesis and optimization of HDL code with std. cell as target technology, while **Synopsys PrimeTime** has been employed for static timing analysis, power estimation and SDF library creation.
The front-end of the framework, written in Python, receives the RTL description of the hardware and the neural network trained model as an ONNX file, so that it can configure the whole system. The front-end receives also a dataset which is used as a series of testing inputs: these can be split based on how many concurrent simulations the user wants to run. At this point the back-end of the framework can be started. Multiple instances of QuestaSim load both the shared library containing all the software needed to emulate parts of the hardware accelerators and the post-synthesis netlist with the proper SDF.
At the end of the co-simulation, the framework generates a human-readable report about the energy consumption, average power, throughput and achieved accuracy on the dataset. Moreover, in order to set up a co-simulation for its system, a future user can focus only on the behavioral representation in C, while all the infrastructure is designed to be easily used.
The next sections of this chapter explain in details the tool interface to the external world and the internal architecture.

4.3 External Interface

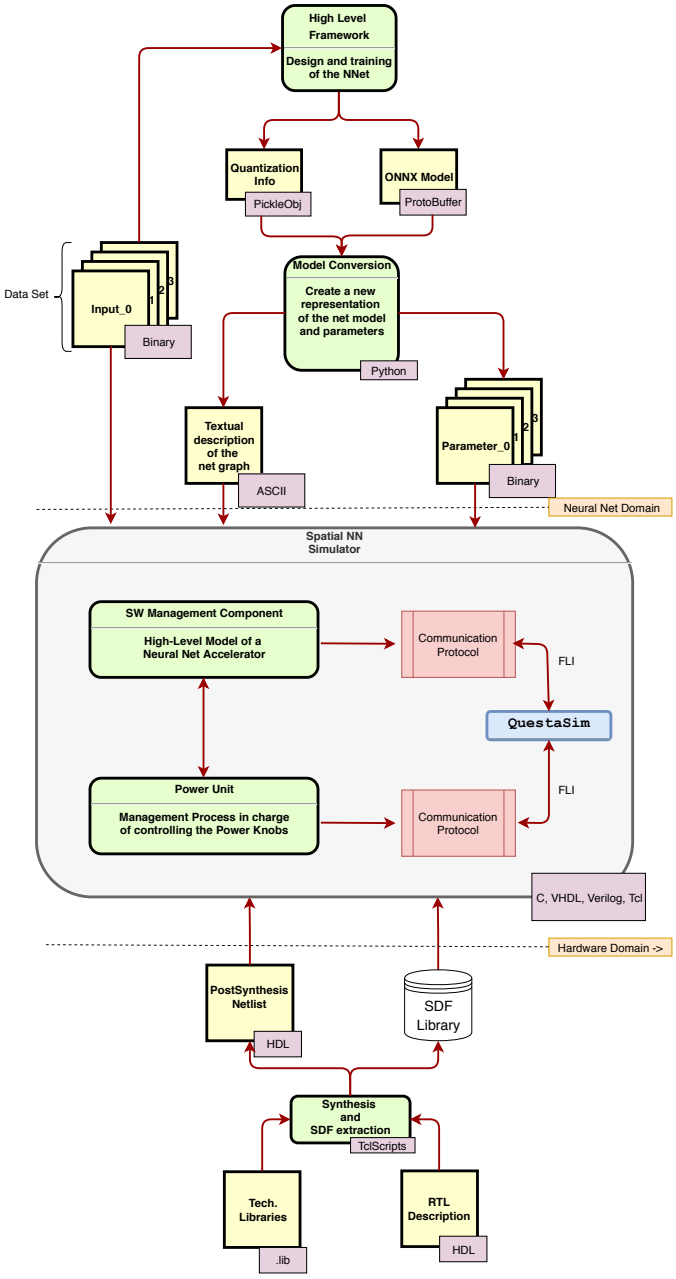


Figure 4.1: Architecture diagram depicting the tool external interface. It is shown that it is fully integrated in standard design flow both in terms of input and of language used.

The interface of tool to the external world is depicted in fig. 4.1. It is conceptually divided in two different domains: the neural net domain and the hardware domain.

4.3.1 Neural Net Domain

During the last years, Different machine learning libraries have been designed with the aim of helping the development of machine learning algorithms. The most diffused for neural networks are **PyTorch**[1] and **TensorFlow**[2]. These frameworks provide a set of ways to describe, train and test neural networks. The main common characteristics of the frameworks are:

- Possibility to define a neural network as interconnection of available layers, each one with several customizable parameters.
- Compute gradients to train a neural network using gradient descent or other optimizers that are already built inside the framework.
- Provide access to a library of defined and pre-trained models that can be easily loaded in custom user program.
- Native support to parallel execution on cluster of processors and/or on GPGPUs.

After training, testing and optimizing the neural architecture, a final model is obtained with a set of trained parameters. In order to easily integrate the co-simulation tool with one of these common machine learning frameworks, it has been decided to use ONNX, an open standard developed by multiple industrial partners to easily transfer models between tools.

ONNX has been designed to facilitate the transition from the framework used to train and test the network to the inferential engine necessary to deploy the model on final hardware platform. ONNX uses as serializing mechanism protocol buffers. Protocol buffers[3] are developed and maintained by Google and represent an efficient way to exchange complex data structures. This mechanism is based on the definition of data structures, called messages, in a human readable *.proto* file. The *.proto* file is processed by the *protobuf* compiler generating methods for encoding/decoding the specified data structures. It is a language-neutral format, so it is possible to use it in C++, Ruby, GO, Python and other programming languages.
The onnx.proto file provides mainly the definition of the following messages:

- `GraphProto`. It contains a list of nodes topologically sorted and without cycle. Each node represents a layer and it has at least one output. It also contains a list of tensor messages called initializers, which specify data for the constant tensor, i.e. weights and biases of all layers. In GraphProto it is possible to find also the information about primary input and primary output of the entire graph.
- `NodeProto`. Each node is described in terms of an optional string identifier, the keyword specifying the layer type and the names of input and output tensors. Inside the network each tensor is identified uniquely by a string identifier. Multiple AttributeProto messages are used to specify properties of the specific layer.

[1] https://pytorch.org/

[2] https://www.tensorflow.org/

[3] https://developers.google.com/protocol-buffers/

- `TensorProto`. It is defined in order to fully describe a tensor, thus it provides information about dimensions and data type. In case of a constant tensor, it contains a vector of raw data in binary format.

Since ONNX does not support fixed-point model, the information about radix point is provided through a *pickle dump object*. In particular, three Python dictionaries are used to specify radix-point positions of weights, biases and output features of convolutional and fully connected layers. More specifically, each dictionary maps a layer identifier with a radix-point value, thus it maps a string to an integer number. The framework front-end has a Python script that receives as inputs the *pickle dump object* and the ONNX file to produce a human-readable description of the network and a binary file for each constant tensor. The format specification of this ASCII description is reported in listing. 5 and an example of input file for LeNet5 Architecture quantized on 8 bits is showed in listing. 6.

```
{Id of the net : string}
{quantized_net : boolean} [{bitwidth used : int} if quantized_net is TRUE]
{n_tensors : int}
{{id : int } {tensor_type: string} {dimensions : int }
  [{path_file : string} if constant]
  [{radix_point : int} if quantized_net is TRUE]} repeated x n_tensors lines
{n_layers : int}
{{id : string} {op_type : string} {id_input_activation : int}
  {id_output_activation : int} [Additional_parameters]} repeated x n_layers lines

################################################################################
.  Additional_parameters, they depend on the type of layer

"Relu"      ->  None

"Fc"        ->  {id_weight_tensor: int  }   {id_bias_tensor: int  }
                {alpha           : float}   {beta          : float}
                {broadcast       : int  }   {transA        : int  }
                {transB          : int  }

"Conv"      ->  {id_weight_tensor: int  }   {id_bias_tensor: int  }
                {dims_y_kernel   : int  }   {dims_x_kernel : int  }
                {padding_y_dim   : int  }   {padding_x_dim : int  }
                {stride_y_dim    : int  }   {stride_x_dim  : int  }

"MaxPool"  ->   {dims_y_kernel   : int  }   {dims_x_kernel : int  }
                {padding_y_dim   : int  }   {padding_x_dim : int  }
                {stride_y_dim    : int  }   {stride_x_dim  : int  }

"Reshape"  ->   {num_next_attr   : int  }
                {how_to_reshape  : int  } repeated x num_next_attr

"Flatten"  ->   {axis            : int  }
################################################################################
```

Listing 5: Structure of the ASCII file describing a neural network

```
LeNet5_8bit
29
-2 output 1 1 1 10 2 8
-1 input 1 3 32 32 6 8
0 constant 6 3 5 5 lenet_data/1.data 8 8
1 constant 1 1 1 6 lenet_data/2.data 9 8
2 constant 1 1 1 1 lenet_data/3.data 0 8
3 constant 16 6 5 5 lenet_data/4.data 7 8
4 constant 1 1 1 16 lenet_data/5.data 8 8
5 constant 1 1 1 1 lenet_data/6.data 0 8
6 constant 1 1 120 400 lenet_data/7.data 7 8
7 constant 1 1 1 120 lenet_data/8.data 8 8
8 constant 1 1 1 1 lenet_data/9.data 0 8
9 constant 1 1 84 120 lenet_data/10.data 7 8
10 constant 1 1 1 84 lenet_data/11.data 8 8
11 constant 1 1 1 1 lenet_data/12.data 0 8
12 constant 1 1 10 84 lenet_data/13.data 7 8
13 constant 1 1 1 10 lenet_data/14.data 8 8
14 constant 1 1 1 1 lenet_data/15.data 0 8
15 constant 1 1 1 1 lenet_data/16.data 0 8
16 feature 1 6 28 28 5 8
17 feature 1 6 28 28 5 8
18 feature 1 6 14 14 5 8
19 feature 1 16 10 10 4 8
20 feature 1 16 10 10 4 8
21 feature 1 16 5 5 4 8
22 feature 1 1 1 400 4 8
23 feature 1 1 1 120 3 8
24 feature 1 1 1 120 3 8
25 feature 1 1 1 84 3 8
26 feature 1 1 1 84 3 8
12
Conv4 Conv -1 16 0 1 5 5 0 0 1 1
Relu5 Relu 16 17
MaxPool6 MaxPool 17 18 2 2 0 0 2 2
Conv11 Conv 18 19 3 4 5 5 0 0 1 1
Relu12 Relu 19 20
MaxPool13 MaxPool 20 21 2 2 0 0 2 2
Flatten14 Flatten 21 22 1
Gemm19 Fc 22 23 6 7 1.0 1.0 1 0 1
Relu20 Relu 23 24
Gemm25 Fc 24 25 9 10 1.0 1.0 1 0 1
Relu26 Relu 25 26
Gemm31 Fc 26 -2 12 13 1.0 1.0 1 0 1
```

Listing 6: Description of LeNet5 quantized on 8 bits

4.3.2 Hardware Domain

For what concerns the hardware domain, it has been decided to focus on standard industrial flow for ASIC design. As reported in fig. 4.2, due to the high complexity of ASIC,

the standard pipeline is composed of different stages, each with its own optimized EDA tool. The design flow starts with the functional specification of the system, followed by an architectural design where the system is decomposed in macro blocks taking into consideration performance specifications. At this point designers start to write HDL code, usually in VHDL or Verilog, specifying behavior and/or structure of the architectural blocks. The RTL description of the system is used as entry point for the logic stage, which produces an optimized gate-level netlist. This consists of a structural description of the circuits in terms of std. cells available in the used technological library. The netlist has to be compliant with user defined timing, area and power constraints. From this stage the so called Back-end design takes care of the physical design of the IC, from gate placement and routing up to packaging design. Then the chip is taped out, fabricated and tested.

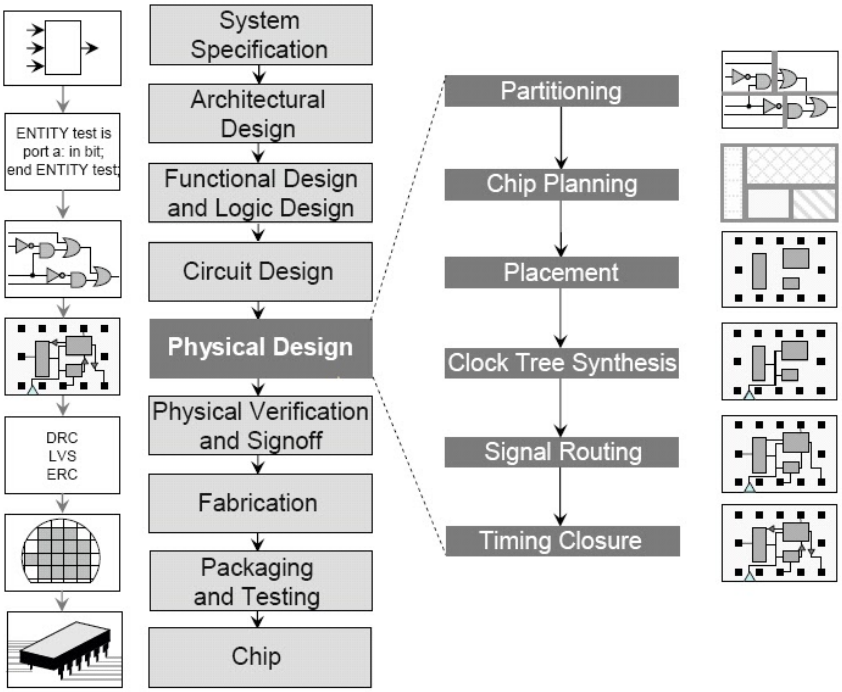


Figure 4.2: Standard design flow of ASIC.

This work deals with architectural, logic and circuit design. The tools used in these parts are Synopsys Design Compiler for synthesis, optimization and tech mapping and Synopsys Prime Time for static timing analysis and power estimation. The silicon vendor provides the designers with a set of cells collected in a library. Timing and power factors of merit of any cell are reported in a standardized human readable file called liberty, *.lib* extension. These parameters are obtained through a number of SPICE simulations of the cell under different conditions, for example rising/falling time of input signals, load capacitance, power supply voltage, process corner and temperature.

The framework developed takes as input an RTL description of the processing element datapath. The information about the synthesis condition in terms of PVT corner is specified in a tcl file, while area, timing and power constraints in a sdc file. The synthesis and optimization flow with Design Compiler has been automated with a series of tcl scripts. The post-synthesis netlist is analyzed with PrimeTime for generating detailed reports about power consumption and static timing of the circuit. In particular, in order to deal with voltage and temperature values different from the one used for synthesis, new operating conditions are created by interpolating the available libraries. PrimeTime is also used to generate the back annotation necessary to perform post-synthesis simulation. The silicon vendor, apart from providing the liberty file of the library, gives also an HDL model of each cell. An example of a NAND2X2 is reported in 7.

```
1  `endcelldefine
2  `celldefine
3  `ifdef functional
4     `timescale 1ns / 1ns
5     `delay_mode_zero
6  `else
7     `timescale 1ns / 1ps
8     `delay_mode_path
9  `endif
10
11 `define NAND2X2_A_R_Z_F_1 0.1
12 `define NAND2X2_A_F_Z_R_1 0.1
13 `define NAND2X2_B_R_Z_F_1 0.1
14 `define NAND2X2_B_F_Z_R_1 0.1
15
16 module NAND2X2 (Z, A, B);
17
18   output Z;
19   input A;
20   input B;
21
22   and    U1 (INTERNAL1, A, B) ;
23   not   #1 U2 (Z, INTERNAL1) ;
24
25   specify
26
27     (A -=> Z) = (`NAND2X2_A_F_Z_R_1,`NAND2X2_A_R_Z_F_1);
28     (B -=> Z) = (`NAND2X2_B_F_Z_R_1,`NAND2X2_B_R_Z_F_1);
29
30   endspecify
31
32 endmodule
```

Listing 7: Verilog description of a library cell

The delays reported in the verilog model of the cells are different from the actual delay of a cell instantiated in the circuit, since the latter depends on the loads, fan-in, fan-out and position in the circuit. Therefore, in order to perform a post-synthesis simulation with an accurate estimation of the circuit timing, PrimeTime offers the possibility to produce

a standard delay format file called *SDF*. The SDF is standardized by IEEE and it is in ASCII format. In this framework SDF is used mainly to annotate delays, timing checks and timing constraints.

```
(CELL
  (CELLTYPE "NAND2X2")
  (INSTANCE U670)
  (DELAY
    (ABSOLUTE
    (COND B (IOPATH A Z (0.033::0.037) (0.047::0.052)))
    (COND A (IOPATH B Z (0.034::0.039) (0.045::0.052)))
    )
  )
)
```

Listing 8: SDF content for IOPATH of a NAND Cell

In listing 8 a part of an SDF file for a NAND2X2 cell is reported. The COND statement is used to specify that the propagation delay between the two pins is valid only when the specified condition is true. The values in the first couple of parentheses correspond to the delay values when Z makes a rising transition, whilst the values in the second couple of parentheses correspond to a falling transition. In general inside each parenthesis three values separated by colon should be reported: minimum, typical and maximum corner values. In this example typical values are missing, thus only the best and worst case gate-level simulation can be performed. The SDF file can be loaded in QuestaSim together with the post-synthesis netlist, and a timing accurate simulation can be performed. For each operating condition of the circuit a different SDF file is generated, in this way it is possible to simulate power management policies where the supply voltage is used as power knob.

4.4 Framework Architecture

The front-end is realized with a Python script that accepts the input files as described in previous section and configures the system to perform the co-simulation.
The back-end is made up of two main parts: one is the infrastructure needed to represent a neural network for the inferential part, while the other is a communication protocol between the software components and the hardware simulators. The latter has been designed in order to simplify the set up of a co-simulation for different dataflow mapping strategies, since they are one of the key aspect of spatial architecture, as explained in subsection 3.2.

4.4.1 Neural Network Computational Model

The basic elements needed to represent a neural network are **tensors** and **layers**.

Tensor. A Tensor is an extension of a matrix on more than two dimensions. In particular, for neural network purposes, they have been considered of maximum four dimensions. For performance reasons they have been stored in contiguous memory locations. In case of a floating-point tensor only an array of raw data and its dimensions are stored as member

values, instead for fixed-point tensors also bitwidth and radix-point position are included as members.
The type of raw data is single precision IEEE-754 float for floating-point tensors, while for fixed-point tensors it has been decided to use integer on 32 bits, but including the information on real bitwidth as a member of the class. In this way it is possible to represent also non power of 2 bitwidth up to 32 bits. The interfaces of a floating-point tensor and of a fixed-point tensor are reported in list. 9 and 10.

```
struct tensor_float {
  unsigned int  dims[4]       ;
  float*        raw_data      ;
};

typedef struct tensor_float* Tensor_float;
```

Listing 9: Row Stationary Dataflow Emulation

```
struct tensor_fixed_point {
  unsigned int  dims[4]      ;
  int32_t*      raw_data     ;
  int8_t        int_bits     ;
  int8_t        frac_bits    ;
  int8_t        tot_bits     ;
};

typedef struct tensor_fixed_point* Tensor_fixed_point;
```

Listing 10: Fixed-point tensor interface

Layer. Even if different layers are employed in a neural net, all of them show common characteristics:

- They can be configured. For example, in case of a convolutional layer, the main options are stride, padding, number of output channels, kernel dimensions and dilation.
- They can be scheduled, i.e. they consume input tensor(s) and perform some kind of processing to produce the output tensor.

In order to deal with both floating point and fixed point, neural network data and computation have been considered totally decoupled. Indeed the layer class is represented by its computational kernel and pointers to I/O and constant tensors. The abstract interface of a layer is listed in 11.

```
typedef struct layer* Layer;

struct layer {
```

```
4     struct vf_table*  vft    ; /**< Pointer to the Virtual Fuction Table     */
5     void*             _layer; /**< Pointer to the struct of specific layer  */
6   };
7
8   /** Pointers to the concrete version of init, print and schedule functions **/
9   struct vf_table {
10    error_code  (*init_layer)       (Layer, void*);
11    void        (*print_layer)      (void*);
12    error_code  (*schedule_layer)   (void*, void*);
13  };
```

Listing 11: Abstract Layer interface

Each layer implements the abstract layer interface through three specific functions:

- `init_layer`. Allocates memory and configures the parameters.
- `schedule_layer`. Performs the computations, i.e. a transformations of the input tensors into the output tensor.
- `print_layer`. Used for debugging purposes, it is employed to print all information about an already configured layer.

The following list describes the implemented layers with the available options:

- `Convolutional Layer`.
  Computational Kernel: Direct Convolution.
  Parameters:
  - Id : string
  - number of input channels: unsigned integer
  - number of output channels: unsigned integer
  - kernel x dimension : unsigned integer
  - kernel y dimension : unsigned integer
  - stride on x dimension : unsigned integer
  - stride on y dimension : unsigned integer
  - padding on x dimension: unsigned integer
  - padding on y dimension: unsigned integer
- `Concatenation Layer`.
  Parameters:
  - Id : string
  - number of input tensors: unsigned integer
  - axis where it has to be performed the concatenation: unsigned integer
- `Fully Connected Layer`.
  Computational Kernel: Matrix Multiplication.
  Parameters:

  - Id : string
  - number of output neurons: unsigned integer
  - number of input neurons: unsigned integer

- `Max Pool Layer`.
  Parameters:
  - Id : string
  - kernel x dimension : unsigned integer
  - kernel y dimension : unsigned integer
  - stride on x dimension : unsigned integer
  - stride on y dimension : unsigned integer
  - padding on x dimension: unsigned integer
  - padding on y dimension: unsigned integer

- `Activation Layer`.
  Parameters:
  - Id : string
  - pointer to the activation function : void*

- `Reshape Layer`.
  Parameters:
  - Id : string
  - new x dimension : unsigned integer
  - new x dimension : unsigned integer
  - new x dimension : unsigned integer
  - new x dimension : unsigned integer

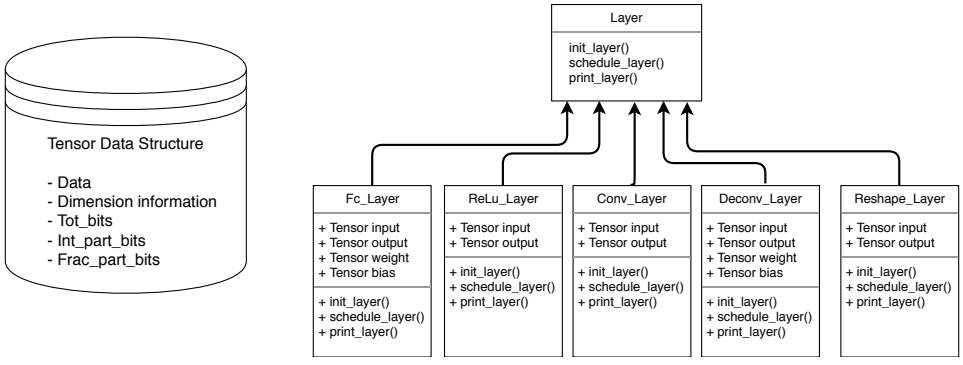


Figure 4.3: Architecture diagram of the main blocks of the simulator.

Network. Layers and tensors can be composed to build a neural network. The class net has an interface reported in listing 12.

```
struct net{
  char                id[50]           ;
  unsigned int        n_layers         ;
  Layer*              layers           ;
  int                 n_tensors        ;
#if   TENSOR_TYPE == FLOATING_POINT
  Tensor_float*       tensors_pool     ;
#elif TENSOR_TYPE == FIXED_POINT
  Tensor_fixed_point* tensors_pool     ;
#endif
#if   TENSOR_TYPE == FLOATING_POINT
  Tensor_float        primary_output   ;
  Tensor_float        primary_input    ;
#elif TENSOR_TYPE == FIXED_POINT
  Tensor_fixed_point  primary_output   ;
  Tensor_fixed_point  primary_input    ;
#endif
};

typedef struct net* Net;

int init_net(
  Net* net_to_init, char* path_init_file
);

void print_net(
  Net net
);

int schedule_net(
  Net net
);

void free_net(
  Net net
);
```

Listing 12: Net interface

All tensors in the network, both constant tensors and activation tensors, are allocated and stored in a tensor pool. To each tensor a numerical id is assigned, so that it is possible to implement the pool as a vector and retrieve the tensor with a direct access. All the layers as well are stored in a vector in topological order. In this way, by simply visiting the vector in order and calling the schedule function of each layer, it is ensured that each input tensor is valid when it has to be processed. The primary input and the primary output of the network are not included in the pool, as the primary input is filled externally and the primary output is processed at the end for obtaining the outcome. Thus, it has been decided to manage them in a different way, in particular as direct pointers.

4.4.2 How to talk with Modelsim: Foreign Language Interface

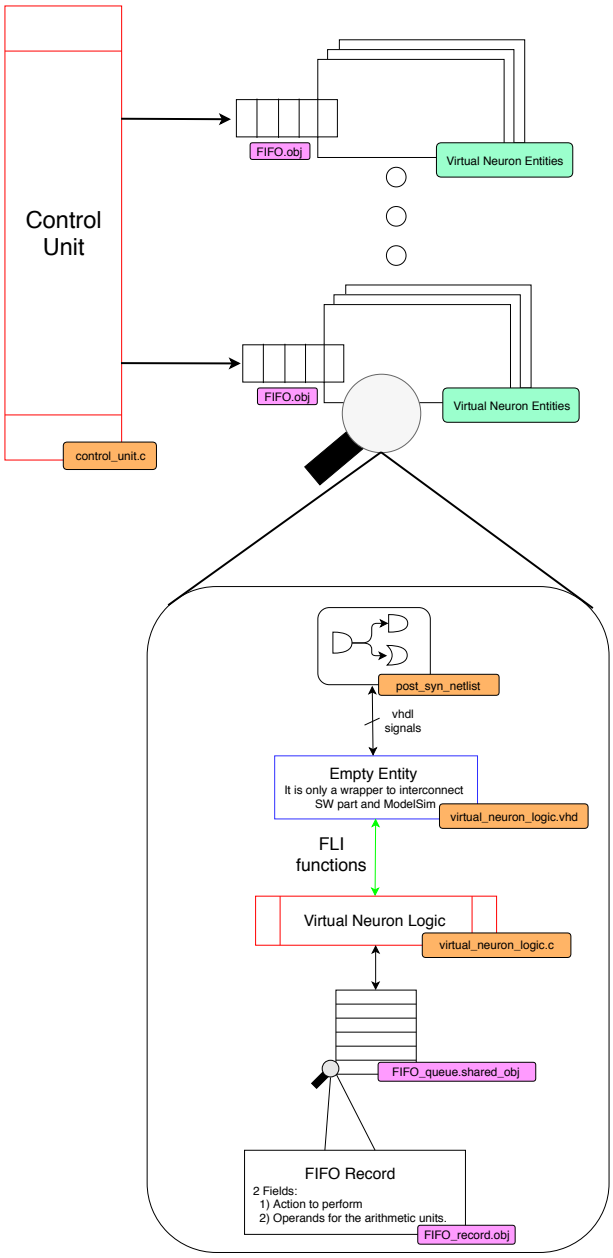


Figure 4.4: Architecture diagram of interface between gate-level circuit and imperative code.

MentorGraphics provides many FLI APIs, i.e. C functions, which can be used to access and modify information inside the HDL simulator.

As reported in FLI manual provided by MentorGraphics, a C program can be used to:

- Explore the hierarchy of an HDL design.
- Read and write values of HDL objects, in particular signals and variables.
- Access information about the simulation status.
- Modify flow and configuration of the simulation.

Usually a VHDL module is composed of two parts: entity declaration and architecture description. The architecture description contains the RTL specification of functionality or structure of the module. In order to use FLI APIs, the architecture is just a link to a foreign C function. In this plain C function the initialization of the module is done, and then multiple threads can be started. It is possible to use threads controlled by the operating system or processes managed by QuestaSim. In the former case every synchronization primitive offered by the operating systems can be used, while in the latter the process is threated as a VHDL process, thus it can have a sensitivity list.
The init function has the following synopsis:

```
1 void function_name(
2   mitRegionIdT       region   ,
3   char              *param    ,
4   mtiInterfaceListT *generics ,
5   mitInterfaceListT *ports
6 )
```

Listing 13: Init function synopsis

The first argument is an ID used to locate the module in the design hierarchy, instead the second allows to specify options when the foreign architecture is instantiated in the *.vhd* file. Generics and ports are linked lists containing handles to generic parameters and ports of the design.
The main FLI functions used in the framework are:

- *mit_FindPort(mtiInterfaceListT *ports, char* port_name).*
  It is used to extract a handle to a port signal in the interface list.
- *mit_CreateDriver(mtiInterfaceListT *ports, char* port_name).*
  It is used to create a driver for a specific signal. The driver is necessary for setting values onto that signal.
- *mit_CreateProcess(char *name_proc, mtiVoidFuncPtrT func_ptr, void *param).*
  It is used to create a process named name_proc, executing the function pointed by func_ptr. It is possible to pass a parameter to the function through the param argument, usually a structure containing all the signals and the drivers created in the init function.

- *mit_Sensitize(mtiProcessIdT proc, mtiSignalIdT sig, mtiProcessTriggerT trigger).*
  It is used to sensitize the process with id proc to signal sig. The trigger parameter is used to specify if the process is sensitive to value changes, MTI_EVENT, or when the signal is active, MTI_ACTIVE.

- *mit_ScheduleDriver(mtiDriverIdT driver_id, long/void* value, mtiDelayT delay, mtiDriverModeT mode).*
  It allows to change the value of the signal controlled by the driver with id driver_id. It is possible to delay the signal with respect to the moment when the function is called. Since in VHDL there are multiple kinds of delay, the argument mode allows to choose between inertial and transport delay. If the signal is an array, then the value is interpreted as a pointer to a vector, otherwise it is a long integer variable.

- *mit_GetSignalValue(mtiSignalIdT signal_id)* and
  mit_GetArraySignalValue(mtiSignalIdT signal_id, void buffer).*
  These two functions are used to read the value of a scalar signal or an array starting from the signal handler.

- *mit_GetArraySignalValue(mtiVoidFuncPtrT func_ptr, void* param).*
  A callback function called when QuestaSim is closed. It is necessary to free all the allocated memory.

In this framework FLI APIs are used mainly to create a foreign architecture emulating the global control unit of the accelerator, the local control unit of each processing element and also the memory system.

4.4.3 Spatial Computing Infrastructure

As shown by the examples of hardware accelerator for deep-learning in section 3.3, the main components of a spatial architecture are:

- `Processing Element`. It is a simple computational node and it can be composed of local memory, control logic and a datapath able to execute arithmetic and logical instructions.

- `Memory system`. Usually it is hierarchically organized to optimize performance, area occupation and power consumption. It could store weights, biases, activations and temporary results.

- `Network-On-Chip`. It is in charge of two main tasks: dispatch data from/to the memory system to/from the processing element and move data between processing elements.

- `Global Control Unit`. It coordinates the functionality of the entire accelerator. It generates high-level control signals for the other components and supervises the interface with the external world.

Since the target for this co-simulation framework is a spatial architecture for deep-learning on chip, each of the previously reported components has to be represented by a data structure, and its behavior must be emulated with a set of methods. As it should be clear by

now, a spatial accelerator is composed of a series of concurrently working elements, thus also the software emulation should have a parallel structure. This is achieved by means of processes managed by the operating systems, POSIX threads since Linux is used as host OS, and processes managed by QuestaSim, in particular as components of a foreign architecture.
The main target is the co-simulation of power management strategies acting on the processing elements. The user can decide how to divide the processing element in two parts: one emulated by a foreign architecture and the other simulated at gate-level. Clearly the user has to provide the RTL description for the gate-level part to the framework. A VHDL entity is declared with a structural architecture such that the two parts are connected by signals, processed as usual in VHDL and with the FLI APIs in the software emulation. The global control unit contains the inferential engine detailed in subsection 4.4.1 and emulates all the high level functionality of the accelerator. The communication between the global control unit and the processing elements is achieved by means of a shared memory. In particular, the global control unit provides instructions and data to the processing elements through thread-safe FIFO queue. A FIFO has been implemented mainly because the PE logic in real hardware does not have a complex mechanism to fetch data, but it just responds to the instruction stream sent by the global control unit. Moreover, from a simulation point of view it solves the problem of different execution speeds by QuestaSim processes and OS processes, meaning that FIFO are also employed due to their asynchronous behaviour as communication primitives.
The shared memory contains also the representation of the tensor. This aspect is extremely important, since in this way it is possible to reduce additional data movement in the simulator: when a processing element has to store a data in the memory, in fact, the address is available, so the foreign architecture is able to write it in memory. The shared memory is also employed to allow the communication between processing elements: by simply pushing instructions and data on the queue of another processing element, it is in fact possible to emulate the movement of data inside the spatial architecture.
Synchronization between the global control unit and the processing elements is done by means of a synchronization barrier. In particular, this barrier is used to synchronize layer scheduling in the net. Only when all the processing elements have finished their work, then the computation for the next layer can be started.
The implemented infrastructure allows to easily emulate different dataflow mapping, since, as reported in subsection 3.2, a dataflow can be represented by means of tiling and loop unrolling of the computational kernel for both convolutional layer and fully connected layer. Thus, it is possible to use the same structure by simply pushing values on the FIFO of the processing element receiving data/action. This allows the user to employ the same index used to write the loop of the computational kernel to address the FIFO of a processing element in the grid.
The following part shows two examples of dataflow mapping: in particular, it explains how to write the computational kernel in order to emulate an output stationary, lst. 15, and a row stationary dataflow, lst. 14.

In case of a **row stationary dataflow**, it is possible to reorganize the six loop code in order to: push activations and weights of the same row on the same FIFO, then proceeds in the vertical direction of the grid to complete one row of the output feature map. The address is sent only to the FIFO corresponding to the top edge of the grid.

```
for ( m = 0; m < n_output_channels; m++ ) {                // Output channels
  for ( r = 0; r < dim_y_output; r++ ) {                   // Ouput neuron y coordinate
    for ( i = 0; i < dim_y_kernel; i++ ) {                 // y coordinate of the kernel
      fifo_ptr = grid[i][r];
      for ( c = 0; c < dim_x_output; c++ ) {               // Output neuron x coordinate
        for ( j = 0; j < dim_x_kernel; j++ ) {             // x coordinate of the kernel
          send_to_fifo(fifo_ptr,I(0,r+i,c+j),W(m,0,i,j));
        }
      }
    }
    for ( c = 0; c < dim_x_output; c++ ) {                 // Output neuron x coordinate
      fifo_ptr = grid[0][r];
      store_to_memory(fifo_ptr, &O(m,r,c));
    }
  }
}
```

Listing 14: Row Stationary Dataflow Emulation

In case of an **output stationary dataflow**, it is possible to reorganize the six loop code in order to: push activations and weights needed for the computation of one output neuron and also the address where to store the result on the same FIFO, then proceeds with the other output neurons advancing the pointer to the FIFO vector in a round-robin fashion.

```
for ( m = 0; m < n_output_channels; m++ ) {        // Output channels
  for ( r = 0; r < dim_y_output; r++ ) {           // Ouput neuron y coordinate
    for ( c = 0; c < dim_x_output; c++ ) {         // Output neuron x coordinate

      fifo_ptr = next_fifo_ptr();

      for ( n = 0; n < n_input_channels; m++ ) {   // Input channels
        for ( i = 0; i < dim_y_kernel; i++ ) {     // y coordinate of the kernel
          for ( j = 0; j < dim_y_kernel; j++ ) {   // x coordinate of the kernel
            send_to_fifo(fifo_ptr,I(n,r+i,c+j),W(m,n,i,j))
          }
        }
      }

      store_to_memory(fifo_ptr, &O(m,r,c));

    }
  }
}
```

Listing 15: Output Stationary Dataflow Emulation

Chapter 5

Use Case: Voltage OverScaling of an Output Stationary Accelerator

This chapter presents a use case of the tool: design space exploration in terms of `dynamic power vs network accuracy` of a spatial architecture implementing an output stationary dataflow. The processing element datapath is augmented with *Razor-FFs* and `MAC-Drop`[37], a **Voltage Over-Scaling** technique, is used as power management strategy.

5.1 Processing Element

The target technology is a 40nm std.cell library by STMicroelectronics.
The accelerator provides fixed-point capability with weights, input and output feature maps on 8 bits, while biases are on 32 bits. A pictorial representation of the bitwidth involved in the I/O interface of the processing element datapath is reported in fig. 5.1. Each layer has its own radix-point value in order to minimize the accuracy drop from the floating-point version of the network, [38].

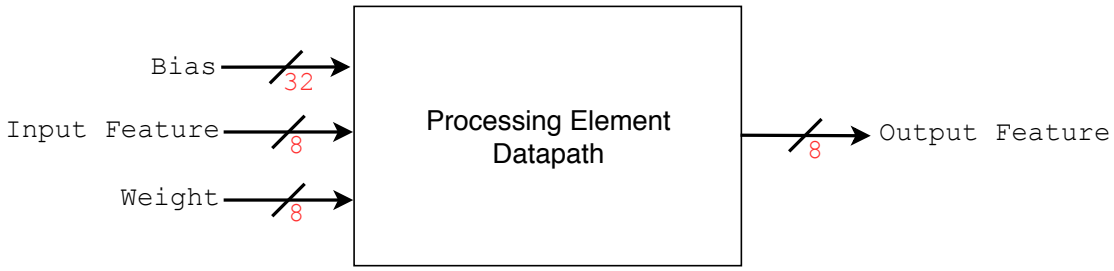


Figure 5.1: Bitwidth involved in the datapath.

For each output neuron, the multiplication between weight and input feature is carried on

a $8 * 8$ integer multiplier, then the output, which is on 16 bits, is added to the 32 bits accumulator register. Even if 16 bits are used as guard-bits to take into account the huge number of accumulations, a saturation logic has been used at the output of the adder in order to make the design overflow-free. At the end of the computations related to one output neuron, the result is shifted, narrowed with saturation on 8 bits and stored in memory. The pseudo-code of the fixed-point computations is reported in lst. 16.

```
int32_t accumulator = bias

for weight in W:
  for input_feature in I:
    int16_t mul_result  = weight*input_feature
    int32_t accumulator = saturated_sum(mul_result,accumulator)

accumulator = accumulator >> ( I.radix_point + W.radix_point - O.radix_point )
int8_t output_feature_map = narrowing_with_saturation(accumulator)
```

Listing 16: Fixed-point computations

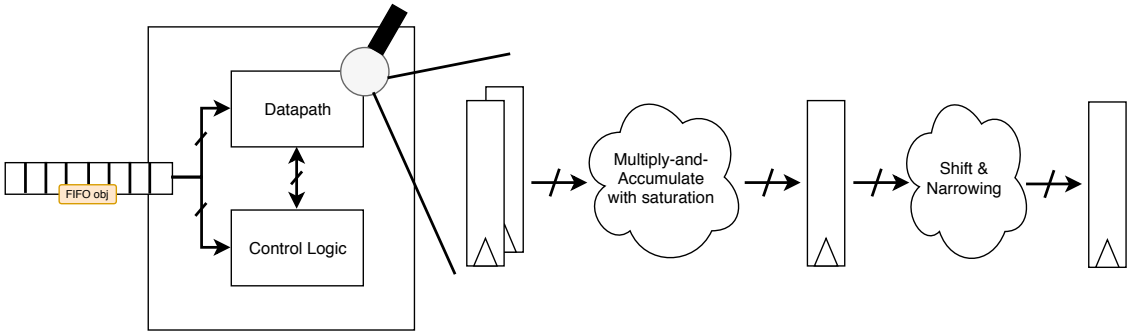


Figure 5.2: High-level view of the PE.

A block diagram of the processing element is depicted in fig. 5.2. It is composed of two main parts:

- `Datapath`: it is a 2 stage pipelined design. The first stage is made of a functional unit able to perform fused multiply-and-accumulate with guard bits on the accumulator and a saturation logic to make all operations overflow-free. While in the second stage it is possible to find the shifter needed to adjust the integer representation according to the radix point value of the weights, input and output feature maps and the narrowing unit with saturation on 8 bits, which is used to restore the 8 bits bitwidth on the generated output neuron.

- `Control Logic`: it reads the stream of instructions and operands from the input FIFO, generates the proper control signals for the datapath and communicates with the memory system in order to write-back the results.

As reported in section 4.4, the user can decide how to partition the spatial accelerator in two parts, one simulated at gate-level, whilst the other emulated with behavioral C code. In this case, it has been decided to bind only the arithmetic core of the processing element datapath to the gate-level simulation. A block diagram of the processing element part emulated at gate-level is shown in fig. 5.3.

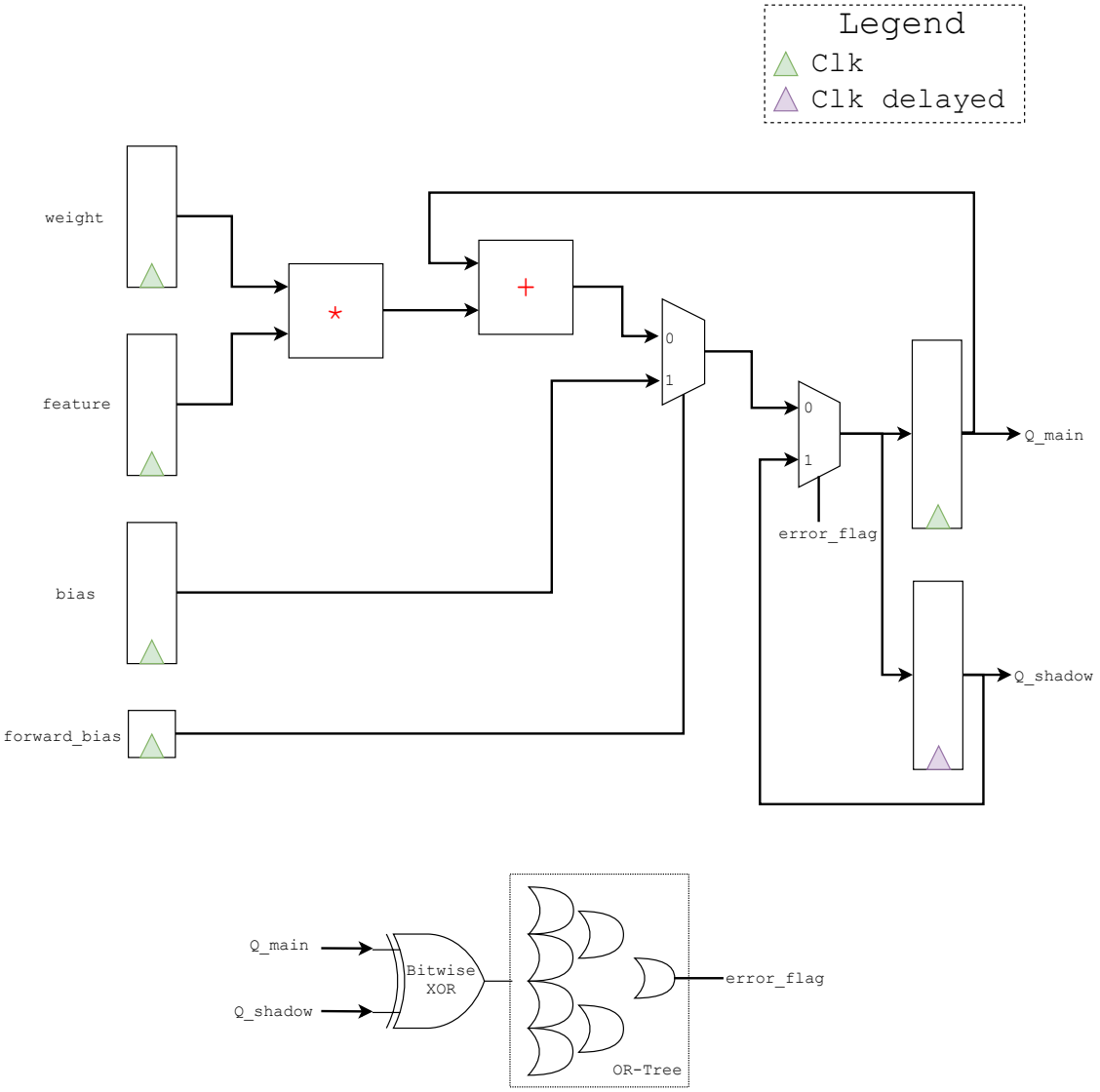


Figure 5.3: RTL view of the processing element stage simulated at gate-level.

As can be seen in fig. 5.3 the datapath stage has been enriched with **Razor-FFs**, [12]. As reported in section 2.4.2, Razor is based on a double-sampling mechanism which is used to detect and, in case, correct timing errors. In this design it has been used as error detection and correction mechanism from timing errors due to **Voltage-OverScaling**. In

particular, the arithmetic cores of each processing element are placed on their own specific V_{dd} domain. An on-chip regulator can change the supply voltage based on the control signals coming from a power management unit. The V_{dd} can be pushed below the point-of-first-failure where the arrival time of all timing paths of the circuit fulfills the required time constraint. In this situation, the main register is not able to sample correctly the signal, but it is ensured that the shadow register could store the right value. This is possible by clocking the shadow register with a delayed clock, which borrows a time-window called `detection window` from the next cycle to wait for late transitions. Classical techniques used to recover from timing errors are not easily applied to spatial architectures since the large number of processing elements makes the adoption of cycle-penalties recovery strategies highly inefficient. For this reason, J.Zhang et al. [37], proposed a timing error recovery technique called `MAC-Drop` which can be easily implemented also in spatial architecture. Instead of correcting a timing error, the cycle following the faulty-one is used to restore the last correct result in the accumulation register. In particular the output of the shadow register is bitwise XORed with the output of the main register. If at least one bit differs between the two registers[1], then the output of the shadow register is provided as input to the main register. In this way one multiplication, i.e. a synaptic connection, is dropped but no time penalty is introduced. If the number of operations dropped due to timing errors is kept under a certain threshold it is still possible to keep the network accuracy sustainable in favor of a lower dynamic power consumption. Fig. 5.4 reports a timing diagram where some cycles are error-free, while in a cycle a timing error occurs, thus it is showed the adopted recovery mechanism.

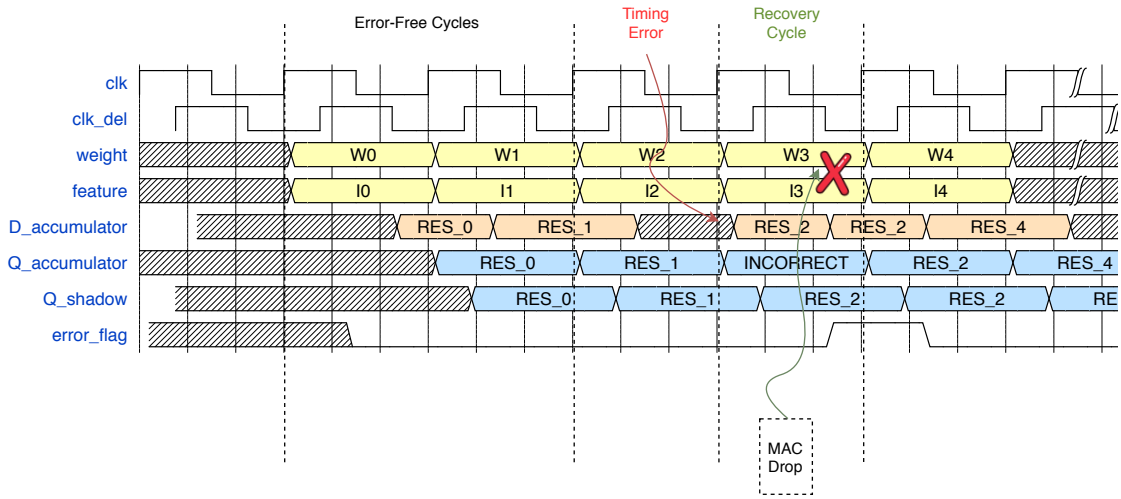


Figure 5.4: Timing diagram of the processing element enriched with Razor.

A particular synthesis flow with specific constraints has to be set-up in order to introduce Razor-FFs in the design. The first design choice is the width of the `detection window`:

[1]This is done by means of an OR-tree.

a greater detection window allows to have lower V_{dd} compared to the nominal one since there is more time for late transitions. Unfortunately, the width of detection window has an impact on the circuit, in particular due to the so called `short-path padding`. In order to be sure that any change in the value of a signal during the detection window is due to a late transition, it must be ensured that no path has an arrival time smaller than the detection window. If this constraint is not satisfied, then a transition during the detection window could be addressed to a computation already finished of the actual cycle and it may not belong to a late computation from the previous cycle. During the synthesis of the circuit the short-path padding can be imposed in the hold-fixing phase, where, instead of using the hold time of the flip-flops, the width of the detection window is imposed as *min-delay constraint*. Fig. 5.5 is a pictorial representation of the phenomenon.

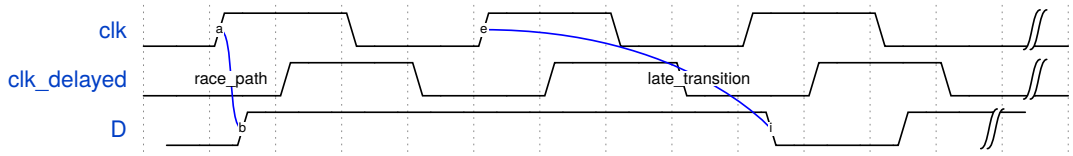


Figure 5.5: Timing diagram of the processing element enriched with Razor.

As reported in 4.4 synthesis, static timing analysis and SDF library extraction are integrated in the framework flow. The synthesis has been done using *Synopsis Design Compiler*, whilst power and timing analysis through *Synopsis Prime Time*. It has been set the voltage step of the on-chip regulator to $20mV$ with a nominal $V_{dd} = 1.10V$. The std.cell library used are characterized at three supply voltages, 1.10V, 1.05V, 0.95V, thus the characterization at the other voltages has been derived by means of an interpolation. The detection window has been set to $0.25 \cdot T_{clk}$ due to the arrival time of the critical paths with the lowest supply voltage, $V_{dd} = 0.96V$. In table 5.1 are reported some factors of merit of the designed processing element with Razor and the one of the baseline without Razor-FFs.

Post Synthesis Results			
	Razor	Baseline	Razor/Baseline
Area	2274.15 μm^2	1320.88 μm^2	1.7
Clock Period	4 ns	4 ns	1
Leakage Power	$2.660 \cdot 10^{-6}$ mW	$1.506 \cdot 10^{-6}$ mW	1.7
Dynamic Power	$3.553 \cdot 10^{-4}$ mW	$2.137 \cdot 10^{-4}$ mW	1.6
Total Power	$3.580 \cdot 10^{-4}$ mW	$2.152 \cdot 10^{-4}$ mW	1.7

Table 5.1: Factors of merit of the used processing element.

5.2 Results

The framework has been used to test the trade-off `accuracy vs` V_{dd} of a five layers CNN, *LeNet5*, quantized on 8 bits and trained on Cifar-10 [39].
The baseline accuracy on the TestSet is: 72%.
In particular the following experiment has been conducted on 1000 images from the TestSet of Cifar-10: the V_{dd} of 8 processing elements has been kept static for the whole inference at different level starting from the nominal one, $1.10V$, down to $0.96V$, with a step of $20mV$. For each layer has been also measured the average timing error rate as:

$$average\_timing\_error\_rate = \frac{\#timing\_errors}{\#images \cdot \#computations} \tag{5.1}$$

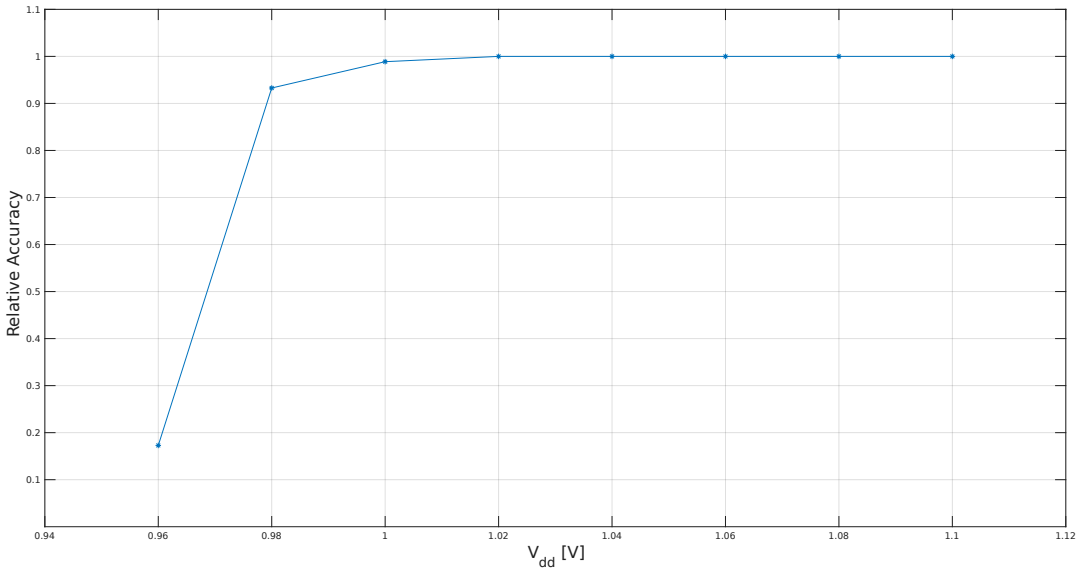


Figure 5.6: Accuracy vs V_{dd}.

Fig. 5.6 shows the accuracy, compared to the baseline, reached at each V_{dd} value.
Fig. 5.7 shows the timing error rate for the convolutional layers, while fig. 5.8 depicts the rate for the fully connected layers. It is possible to notice that at 1.02V, i.e. $80mV$ under the nominal V_{dd}, no timing error occurs, thus there is no accuracy drop compared to the baseline. At $1.00V$ the timing error rate is very low, less than 1% for all layers, indeed the accuracy is still comparable with the baseline. This experiment confirms that even if the static timing analysis states that a certain number of paths will be in violation, only the real workload, i.e. which are the paths actually sensitized, can determine the number of timing violations. At $0.98V$ the timing error rate of each layer increases but the accuracy drop is about 3%, while at $0.96V$ the timing error rate is so high that the accuracy of the network goes down to 17% compared to the baseline.

An appealing outcome of this experiment is also the differences between the layers in terms of timing error rate. For the classifier block, deeper is the layer and lower is the error rate, whilst for the feature extractor it is the opposite. This result can be used to elaborate a more complex strategy where the V_{dd} is different depending on the layer executed. The aim of the power management strategy should be to keep the timing error rate high enough to lower the V_{dd}, therefore minimizing dynamic power consumption, but lower enough to avoid an excessive accuracy degradation.

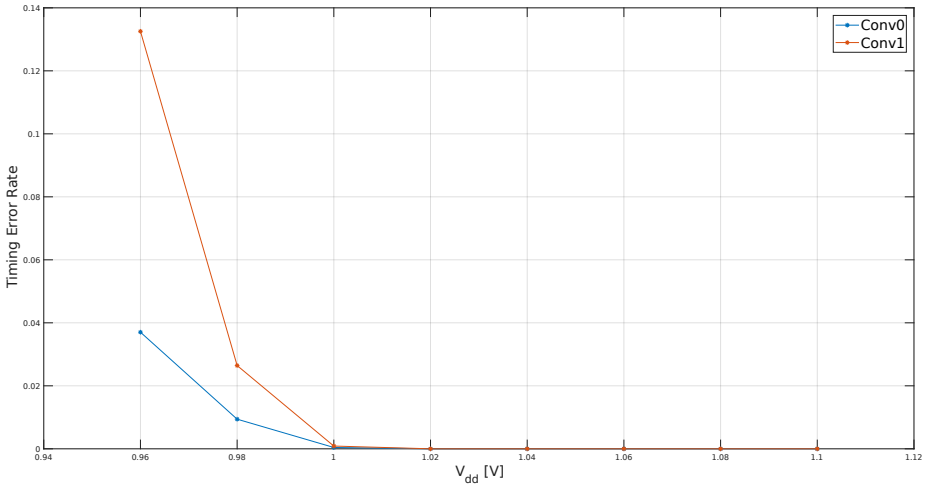


Figure 5.7: Average Timing Error Rate of Convolutional Layers.

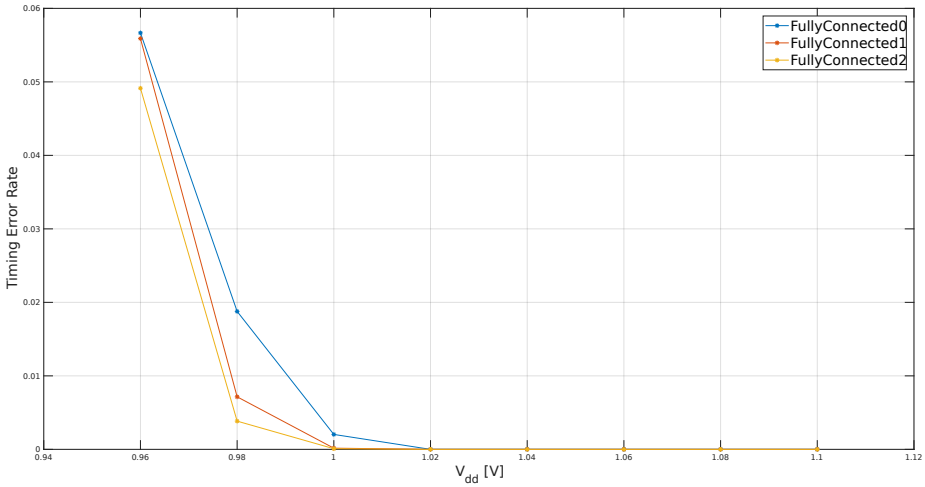


Figure 5.8: Average Timing Error Rate of Fully Connected Layers.

Chapter 6

Conclusions and Future Work

This thesis provides a CoSimulation framework for assessment of power knobs in spatial architectures for deep-learning hardware accelerations. The presented tool provides a full integration both with common machine learning frameworks and with the industrial ASIC design flow.
The effectiveness of the framework as useful tool for early-phase verification of power management strategies has been demonstrated in a real design case of an output stationary accelerator enriched with Razor-FFs.
Possible future works include:

- Integration with parallel gate-level simulator or with an FPGA to speed-up the entire simulation time.
- Investigation of more complex data-driven power management strategies.
- Development of optimization algorithms on neural network model to improve the efficiency of data-driven power management.

Bibliography

[1] G. E. Moore, *Cramming more components onto integrated circuits. Electronics 38 (8): 114–117*, 1965.

[2] *Power Dissipation in portables-Design considerations using low power CMOS*. [Online]. Available: https://www.edn.com/design/power-management-design/4016292/Power-dissipation-in-portables-Design-considerations-using-low-power-CMOS-ICs.

[3] J. M. Rabey, A. Chandrakasan, and B. Nikolic, *Digital Integrated Circuits, a design perspective, 2003.*

[4] K. Roy, S. Mukhopadhyay, and H. Mahmoodi-Meimand, "Leakage current mechanisms and leakage reduction techniques in deep-submicrometer CMOS circuits", *Proceedings of the IEEE*, vol. 91, no. 2, pp. 305–327, 2003.

[5] K. Usami and M. Horowitz, "Clustered voltage scaling technique for low-power design", in *Proceedings of the 1995 international symposium on Low power design*, ACM, 1995, pp. 3–8.

[6] M. Donno, L. Macchiarulo, A. Macii, E. Macii, and M. Poncino, "Enhanced clustered voltage scaling for low power", in *Proceedings of the 12th ACM Great Lakes symposium on VLSI*, ACM, 2002, pp. 18–23.

[7] J. Kathuria, M Ayoubkhan, and A. Noor, "A review of clock gating techniques", *MIT International Journal of Electronics and Communication Engineering*, vol. 1, no. 2, pp. 106–114, 2011.

[8] H. Jiang, M. Marek-Sadowska, and S. R. Nassif, "Benefits and costs of power-gating technique", in *Computer Design: VLSI in Computers and Processors, 2005. ICCD 2005. Proceedings. 2005 IEEE International Conference on*, IEEE, 2005, pp. 559–566.

[9] Y.-T. Ho and T.-T. Hwang, "Low power design using dual threshold voltage", in *Proceedings of the 2004 Asia and South Pacific Design Automation Conference*, IEEE Press, 2004, pp. 205–208.

[10] D. Nguyen, A. Davare, M. Orshansky, D. Chinnery, B. Thompson, and K. Keutzer, "Minimization of dynamic and static power through joint assignment of threshold voltages and sizing optimization", in *Proceedings of the 2003 international symposium on Low power electronics and design*, ACM, 2003, pp. 158–163.

[11] K. Choi, R. Soma, and M. Pedram, "Dynamic voltage and frequency scaling based on workload decomposition", in *Proceedings of the 2004 international symposium on Low power electronics and design*, ACM, 2004, pp. 174–179.

[12] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, *et al.*, "Razor: A low-power pipeline based on circuit-level timing speculation", in *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, IEEE Computer Society, 2003, p. 7.

[13] H. Fuketa, M. Hashimoto, Y. Mitsuyama, and T. Onoye, "Adaptive performance compensation with in-situ timing error predictive sensors for subthreshold circuits", *IEEE Transactions on very large scale integration (VLSI) systems*, vol. 20, no. 2, pp. 333–343, 2012.

[14] J. Zhou, X. Liu, Y.-H. Lam, C. Wang, K.-H. Chang, J. Lan, and M. Je, "HEPP: A new in-situ timing-error prediction and prevention technique for variation-tolerant ultra-low-voltage designs", in *Solid-State Circuits Conference (A-SSCC), 2013 IEEE Asian*, IEEE, 2013, pp. 129–132.

[15] K. Kang, S. P. Park, K. Kim, and K. Roy, "On-chip variability sensor using phase-locked loop for detecting and correcting parametric timing failures", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 18, no. 2, pp. 270–280, 2010.

[16] C. Liu, J. Han, and F. Lombardi, "A low-power, high-performance approximate multiplier with configurable partial error recovery", in *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, IEEE, 2014, pp. 1–4.

[17] P. K. Krause and I. Polian, "Adaptive voltage over-scaling for resilient applications", in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*, IEEE, 2011, pp. 1–6.

[18] B. Moons and M. Verhelst, "Dvas: Dynamic voltage accuracy scaling for increased energy-efficiency in approximate computing", in *Low Power Electronics and Design (ISLPED), 2015 IEEE/ACM International Symposium on*, IEEE, 2015, pp. 237–242.

[19] *What Is Machine Learning?* [Online]. Available: https://www.ibm.com/developerworks/community/blogs/jfp/entry/What_Is_Machine_Learning?lang=en.

[20] S. S. Haykin, *Neural networks and learning machines*. Pearson Upper Saddle River, 2009.

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

[22] F. Rosenblatt, "The perceptron: a probabilistic model for information storage and organization in the brain.", *Psychological review*, vol. 65, no. 6, p. 386, 1958.

[23] *Neural Network Zoo*. [Online]. Available: http://www.asimovinstitute.org/neural-network-zoo/.

[24] J. Yosinski, J. Clune, A. Nguyen, T. Fuchs, and H. Lipson, "Understanding neural networks through deep visualization", *arXiv preprint arXiv:1506.06579*, 2015.

[25] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks", in *European conference on computer vision*, Springer, 2014, pp. 818–833.

[26] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks", in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[27] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge", *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015. DOI: 10.1007/s11263-015-0816-y.

[28] M. Poggi, F. Aleotti, F. Tosi, and S. Mattoccia, "Towards real-time unsupervised monocular depth estimation on CPU", *arXiv preprint arXiv:1806.11430*, 2018.

[29] K. Goto and R. A. Geijn, "Anatomy of high-performance matrix multiplication", *ACM Transactions on Mathematical Software (TOMS)*, vol. 34, no. 3, p. 12, 2008.

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

[31] Z. Lu, S. Rallapalli, K. Chan, and T. La Porta, "Modeling the resource requirements of convolutional neural networks on mobile devices", in *Proceedings of the 2017 ACM on Multimedia Conference*, ACM, 2017, pp. 1663–1671.

[32] W. Lu, G. Yan, J. Li, S. Gong, Y. Han, and X. Li, "Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks", in *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*, IEEE, 2017, pp. 553–564.

[33] V. Dumoulin and F. Visin, "A guide to convolution arithmetic for deep learning", *ArXiv e-prints*, 2016. eprint: 1603.07285.

[34] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks", *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, 2017.

[35] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: efficient inference engine on compressed deep neural network", in *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, IEEE, 2016, pp. 243–254.

[36] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, *et al.*, "In-datacenter performance analysis of a tensor processing unit", in *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*, IEEE, 2017, pp. 1–12.

[37] J. Zhang, K. Rangineni, Z. Ghodsi, and S. Garg, "ThUnderVolt: Enabling Aggressive Voltage Underscaling and Timing Error Resilience for Energy Efficient Deep Learning Accelerators", in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, IEEE, 2018, pp. 1–6.

[38] D. Lin, S. Talathi, and S. Annapureddy, "Fixed point quantization of deep convolutional networks", in *International Conference on Machine Learning*, 2016, pp. 2849–2858.

[39] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images", Citeseer, Tech. Rep., 2009.