POLITECNICO DI TORINO

M.Sc. in Electronic Engineering

Master's Degree Thesis

Design of a Coarse Grain Reconfigurable Array for Neural Networks

Supervisors
Prof. Maurizio Martina
Prof. Ahmed Hemani

Candidate
Guido Baccelli
student number: 236872

Academic Year 2019-2020

Summary

In recent years, the development of Systems on Chip (SoCs) is facing increasing challenges, which are starting to expose the limitations of current design philosophies. In spite of many conservative measures, often at the expense of computational efficiency, realising a functional SoC has come to require tens of Millions USDs. Such a tendency threatens to hinder innovation and keeps away applications that require the efficiency of ASIC. Among them, a prime example are Artificial Neural Networks (ANNs), which have become one of the most popular research topics and already see many commercial applications in countless fields. The increasing complexity of ANNs makes them quite demanding in terms of computational power, so the inefficient approach of modern SoCs is quickly becoming unsuited for them. This thesis proposes a novel VLSI design framework called SiLago, which has the potential to overcome the main architectural limitations of modern SoCs while also cutting their engineering costs. One of the main innovations is the adoption of a complete hardware approach, where all computation relies on a Coarse Grain Reconfigurable Array (CGRA) of custom blocks that are able to accelerate different applications reusing the same hardware. Implementing popular algorithms such as ANNs on a SiLago platform is a good opportunity to prove its advantages and this is precisely the rationale behind this thesis, which is about the design of two CGRAs compatible with SiLago and customised to support three classes of ANNs. One CGRA targets Convolutional Neural Networks (CNN) and Long-Short Term Memory (LSTM), while the other is suited for Self-Organizing Maps (SOM). In particular, DataPath Unit and Compression Engine for both types of SiLago blocks have been designed from scratch. Special care has been given to obtaining versatile and efficient implementation for compression algorithms and for the nonlinear functions sigmoid, hyperbolic tangent, exponential and softmax required by ANNs. The design flow has been carried out end-to-end, from algorithm specifications to post place & route verification. The final results are two fully functional CGRAs, detailed down to the physical level and accompanied by extensive reports on their area occupation.

Keywords: SoC, SiLago, CGRA, ASIC, Neural Networks

Acknowledgements

I would first like to thank professor Ahmed Hemani for giving me the precious opportunity to do my Master's Thesis at the Department of Electronics at KTH, it meant a lot to me.
My sincere thanks also go to professor Maurizio Martina as my supervisor at Politecnico di Torino for revising my work and to the Ph.D. students Dimitrios Stathis and Yu Yang, who were always there to help whenever I needed and gave an important contribution to my thesis.
None of my achievements would have ever been possible without the unconditional love and support of my parents Carlo and Silvia and of my sister Delia. To them goes my most heartfelt gratitude.

Contents

List of Tables

List of Figures

Chapter 1

The SiLago Framework

SiLago is a novel VLSI design framework, conceived to solve the main problems that the state-of-the-art SOC designs are facing. It is based on the new concept of Synchoricity [8], the division of space in a uniform grid. Together with it, the key proposal is raising the physical design abstraction to Register Transfer Level, by using coarse grain reconfigurable building blocks called SiLago blocks. These components are hardened, i.e. their physical design is already done, so they represent the new basic element for VLSI designs. By coupling these components with the idea of Synchoricity, it becomes possible to create arbitrarily complex systems just by abutting SiLago blocks, without any further logic or physical synthesis. Compared to modern SOCs, this new design philosophy shows the potential to increase the system efficiency and to cut down most of the engineering costs by enabling Application and System level synthesis [9]. In the following, Section 1.1 describes the main problems that modern SOCs are facing, while Section 1.2 details how SiLago proposes to solve them. Sections 1.2.3 and 1.2.4 analyze the computational and storage fabrics that are the main interest for this work.

1.1 Modern SOCs and their issues

The increasing design complexity and low power demands of modern applications are overcoming the improvements in performance offered by technology scaling [10]. This ever-widening architecture efficacy gap has led into the Dark Silicon era, where power limitations result in exponentially smaller percentage of the chip that can be active with each processor generation [11]. Since this constraint allows to turn on only a restricted number of transistors at a given time, it becomes necessary to use them with the highest possible silicon and computational efficiencies. It is well established [10] that custom hardware implementations are significantly more efficient than general purpose, software based approaches; however, it is also clear that they come at much higher engineering and manufacturing costs. Motivated

by these arguments, the VLSI design community has settled for a compromise, an architectural style that achieves only partial customization at two levels. The first one is relying on heterogeneity in processors. Powerful general purpose VLIW or superscalar CPUs with deep pipelines and sophisticated control logic are known to waste a significant amount of energy in overheads, so it is not advisable to use them for all kinds of applications. Instead, functionalities that require lower performance can be moved to simpler and smaller processors with a lower power consumption and greater efficiency. Ultimately, heterogeneity is implemented by including several processors with different footprints, and powering only one of them depending on performance requirements. The second level of customization relies on mapping power and performance critical parts of the functionality to custom hardware designs called accelerators. The two approaches are different, but the underlying rationale is the same: customizing the hardware to improve computational and silicon efficiencies. This architectural style based on two optimization levels, called 'accelerator-rich heterogeneous multi-processor', naturally comes at a steeper engineering cost, so in order to tackle it SOCs have been based around popular general purpose processors and their associated interconnect and peripheral systems. In this way, platforms are built around pre-designed and pre-verified IPs that enable rapid integration of processors and custom accelerators.
This design approach is well justified, but it is still affected by some fundamental flaws that threaten to stifle future innovation. The most evident issue is that customization is still very limited, since it is only focused on improving arithmetic-logic operations. Storage, interconnect, control and address generation have a far greater impact on the overall efficiency, but are still handled by bulky centralized processors. Customizing all elements of an SOC would be overly expensive, so the inefficient software-centric implementation style is still preferred in spite of its disadvantages.
The other main issue that afflicts SOC is the ever-increasing engineering cost. In spite of the adoption of platform-based design, the expense for designing and manufacturing a SOC has reached hundreds of millions USD as reported in [12], with around 90% of the total amount being engineering cost. There are several reasons for this phenomenon, all boiling down to the large abstraction gap between the system level and the basic design elements, the standard cells. This distance leads to an overly wide design space to be searched, as can be seen in Figure 1.1: going down from the system perspective towards the physical, the number of possible solutions increases exponentially, making the automation process too long to be profitable. For this reason, automatic tools are effective only up to the RTL level, with the result that synthesis from system down to RTL is still largely a manual task. The severe downside becomes then the introduction of a costly verification step, where fulfilling the performance constraints is by far the most problematic aspect. The cost metrics of a design are known with certainty only when the physical design is finished; all abstraction levels above require the syntheses (manual or automatic) to

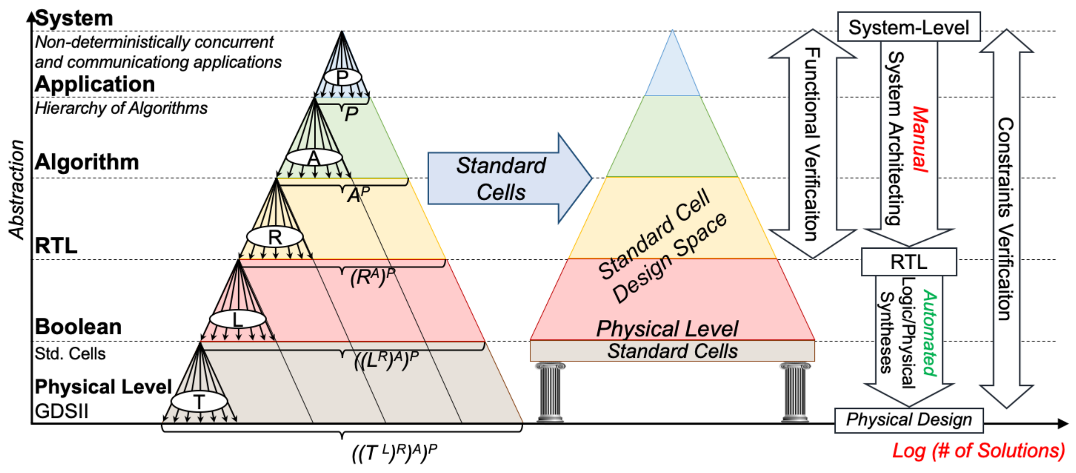


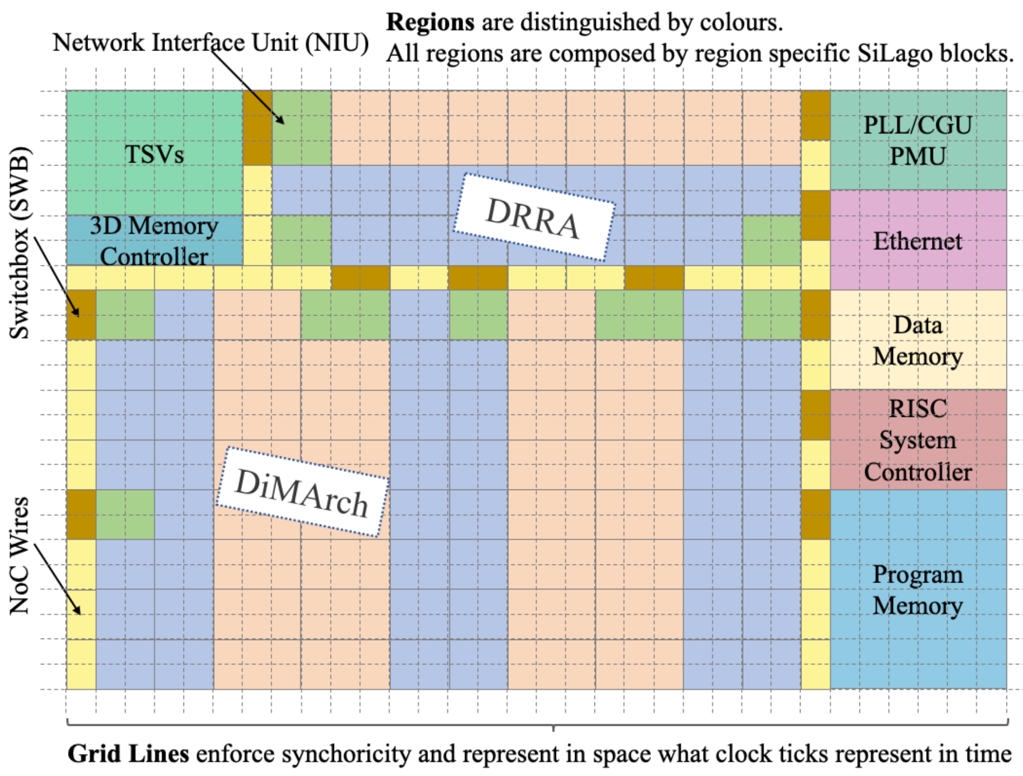
Figure 1.1: Digital systems design space (left). Current standard design flow (right)

make decisions based on estimates. As the abstraction gap increases, the accuracy of these estimates further degrades, so that the design refinement has to go through multiple tedious iterations before meeting its constraints. Ultimately, this part of verification is the main responsible for the huge engineering cost of SOC.

1.2 The SiLago Solution

This Section is meant to provide a general overview of the fundamental concepts behind SiLago, while a more detailed approach is taken in [12]. An in-depth, complete explanation of SiLago is available in [13].
The key consideration to be taken from the problems analyzed in Section 1.1 is that the standard cell-based approach is no longer scalable for modern designs with billions of gates. The fundamental idea behind SiLago is then raising the physical abstraction level to RTL: boolean level standard cells should be replaced by micro-architectures called SiLago blocks as the atomic building components of VLSI systems. In doing this, it is critical to avoid the same problem that was left unsolved during the first change from fully custom layouts to standard-cell based designs. Standard cells in fact only *partially* raised the abstraction from physical level when they fixed the circuit level decision for logic but not for wires. Not only data connections, but also infrastructural ones like clock tree and power lines must be laid out as part of the physical synthesis step. This flaw has led to even larger the design spaces and more inaccurate cost metrics. SiLago wants to take a step forward and completely raise the physical design abstraction to RTL. Once

the design is refined down from system to RTL, the dimension and position of not just every transistor but also every wire segment *in the entire design* is decided. This includes all functional wires but also clock trees and power grids. Being able to compose such regular and predictable designs naturally requires a strict physical design discipline, and SiLago provides it in the form of Synchoricity. The word comes from the Greek ‘σύν’, which indicates union or concurrence, and ‘χορός’, space. It is useful to compare it to the already well known concept of synchronicity: as a synchronous system implements complex functionality by distributing it over uniform time frames, so a synchoros system is composed by a regular division of space. This design philosophy is enforced by using a virtual grid, so that each instance of a SiLago block occupies a contiguous number of cells. Synchoricity then enables to build arbitrarily complex systems by just placing compatible types of SiLago blocks next to each other, i.e. abutting them. Figure 1.2 provides an example with a typical SiLago-based SOC. Now that the general rule of Synchoricity



Regions are distinguished by colours.
All regions are composed by region specific SiLago blocks.
Network Interface Unit (NIU)
TSVs
3D Memory Controller
DRRA
PLL/CGU PMU
Ethernet
Data Memory
RISC System Controller
Program Memory
DiMArch
Switchbox (SWB)
NoC Wires
Grid Lines enforce synchoricity and represent in space what clock ticks represent in time

Figure 1.2: SOC based on SiLago Framework

has been set, it is possible to explain in detail how SiLago blocks must be designed to enable it.

1.2.1 SiLago VLSI design flow

SiLago blocks implement micro-architecture level operations, they are 3 to 4 orders larger than boolean level standard cells and they replace them as atomic building blocks of VLSI designs. They are hardened, which means that their physical design is done, so they can be characterized with post-layout accuracy and their cost metrics can be directly exported to higher abstraction synthesis tools. Just like standard cells, their design is a one-time engineering effort, which in this case can be carried out on standard EDA tools for ASIC. The abutment process enabled by Synchoricity implies that all interconnects of neighbouring SiLago blocks must align to create a valid VLSI design without any further logic or physical synthesis. This property is implemented during the hardening process, and it follows essentially three rules. First, no dedicated point-to-point connections are allowed. All wires whose span goes beyond one block are divided in equal parts and absorbed within each block. This holds for functional wires, but also for the infrastructural ones. Second, all these interconnects are brought to the periphery on the correct position and metal layer, so that they automatically connect when SiLago blocks are abutted. Third, arbitrary blocks cannot be neighbours. A SiLago-based system does not allow every block to abut to all others, because it must be still organized in regular regions that correspond to specific functionalities. This translates into a difference in the type and number of interconnections depending on the block specialization: only SiLago cells that are related in functionality can abut correctly.
All the rules just laid out enable rapid generation of valid VLSI design instances just by aligning blocks on a grid. Moreover, they also ensure that SiLago cells of the same type are all identical, so that the cost metrics are invariant to their position. In reality, cells at the region boundaries are slightly different, but there is only a finite number of possible corner types so they can all be characterized and included in the model.
The final output of the SiLago physical platform design is a set of hardened SiLago blocks, each one with its supported micro-architectural operations. From this abstraction level true High Level Synthesis becomes viable: tools can map algorithms with ease, by exploring all combinations of SiLago blocks in different architecture styles and parallelism. The set of solutions for each algorithm is grouped inside a separate FIMP (Function Implementation) library, which has to be derived as a one-time engineering effort. FIMPs then can act as basic blocks for higher abstraction tools like ALS (Application Level Synthesis) and SLS (System level Synthesis), opening the possibility of an automated design flow starting from system level. Having presented the new concepts of SiLago, it is now possible to summarize how they overcome the limitations of modern SOC design flow. The key idea of raising the physical abstraction level reduces the large gap between the overall system view and the basic building blocks, so that the design space becomes exponentially smaller to search. Synchoricity leads to functional blocks that are pre-designed, verified and

characterized with post-layout accuracy, and that can be directly abutted to create valid VLSI designs without any further logic and physical synthesis. Coupling these two main innovations together takes down completely the need for costly system verification, and opens up to the possibility of a fully automated flow from system level specifications to timing and DRC clean GDSII physical layout. As a recap, Figure 1.3 takes the traditional SOC flow that was previously shown, and compares it to the newly proposed SiLago approach.

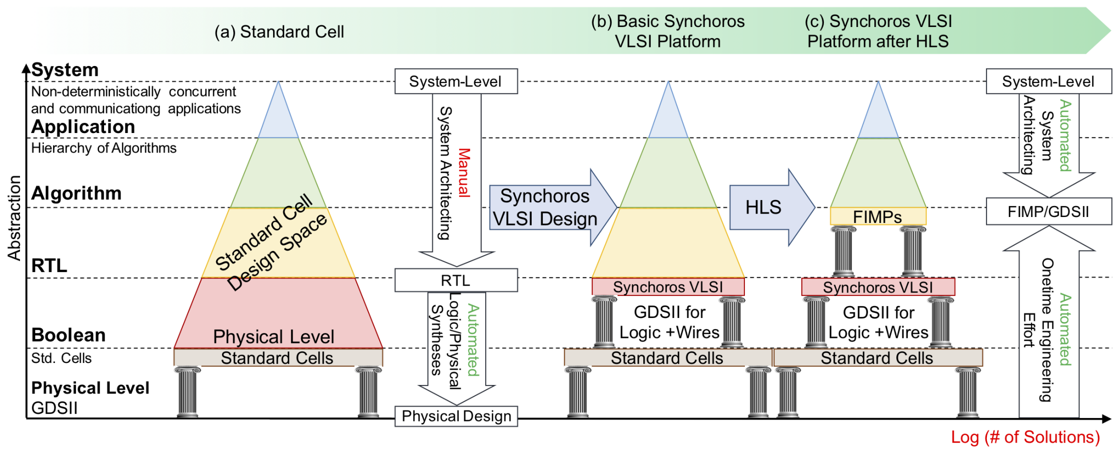


Figure 1.3: Standard Cell-based flow compared to Synchoros SiLago flow

Up to this point, the attention has been focused on solving the large abstraction gap problem, but Section 1.1 has also pointed out the need to overcome a software-centric implementation style and to move towards complete hardware customization. SiLago addresses this problem as well, with a system architecture template based on functional regions that is described in Section 1.2.2.

1.2.2 Silago Regions and Customization

SiLago regions are the enablers of design flexibility and customization. Figure 1.2 provides an overview of a complete SiLago-based SOC, where the organization in regions is clearly visible. Each one of them is tailored for a specific type of functionality in all aspects of its architecture – computation, control, address generation, interconnect, storage and access to it. The generality and completeness of the SiLago framework lies in aggregation of the highly customized region types.
SiLago regions are split in two categories: Infrastructural and functional. For the infrastructural, examples are clock-reset generation, power management, memory conrol, system control and global NOC. Except for the global NOC, the system needs only one region instance for each type. The functional regions are dedicated to the execution of applications, and are implemented as two different Coarse

Grain Reconfigurable Arrays (CGRA), built by abutting customized SiLago blocks. The Dynamic Reconfigurable Resource Array (DRRA) covers computation, control and address generation, while the Distributed Memory Architecture (DiMArch) provides streaming scratchpad storage with local address generation. Figure 1.4 presents the two fabrics and a detailed explanation of their content is given in Sections 1.2.3 and 1.2.4.

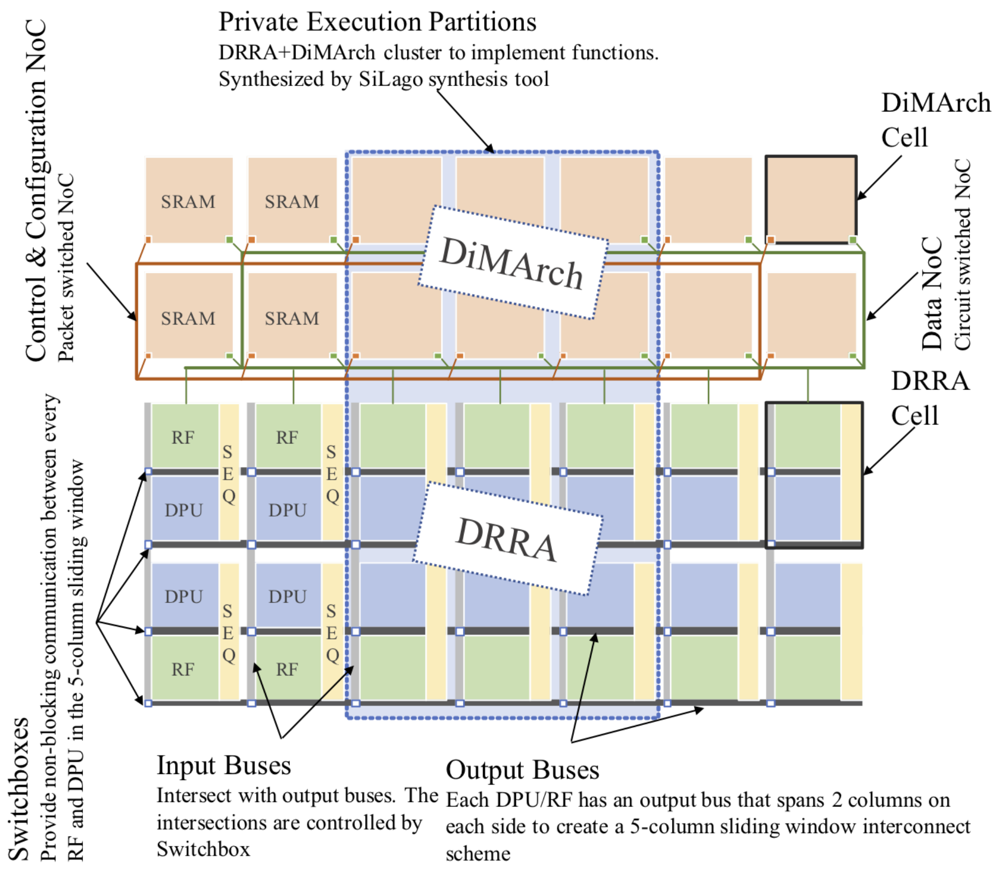

Private Execution Partitions
DRRA+DiMArch cluster to implement functions. Synthesized by SiLago synthesis tool
DiMArch Cell
Control & Configuration NoC
Packet switched NoC
Data NoC
Circuit switched NoC
SRAM
DiMArch
DRRA Cell
RF
DPU
SEQ
DRRA
Switchboxes
Provide non-blocking communication between every RF and DPU in the 5-column sliding window
Input Buses
Intersect with output buses. The intersections are controlled by Switchbox
Output Buses
Each DPU/RF has an output bus that spans 2 columns on each side to create a 5-column sliding window interconnect scheme

Figure 1.4: DRRA and DiMArch structure

Different classes of algorithms require different sets of functionalities, so the SiLago cells that form these fabrics have to be customized depending on the target application, as a one-time engineering effort. This research work serves as a direct example, being in fact the design of a DRRA SiLago basic block to implement three different classes of Neural Networks. It must be pointed out that in general DRRA and DiMArch are meant for data-parallel streaming algorithms, but they are not purely arithmetic and storage fabrics as other CGRA fabrics tend to be. In fact,

they have a rich parallel distributed control to handle the necessary FSM hierarchy for control of streaming functions. Still, in case of control-intensive algorithms where the address generation is compile-time dynamic, the fabrics are not suitable and simple processors named Flexilators have to be coupled with the DRRA to handle the more complex control loops.
Overall, in modern SOCs general purpose processors handle most of the execution, while critical computation are left to the accelerators. Instead, in SiLago SOCs the core functionality is implemented by dedicated hardware (the fabrics), while only the control-intensive parts are delegated to small general purpose processors. To customize the execution even further, ALS tools can detect the optimal number of resources to allocate for an application in order to meet its constraints. This is possible because all SiLago blocks are characterized in throughput, latency and power footprint with measurement-level accuracy. Such level of predictability allows then to reserve just the optimal amount of DRRA, DiMArch cells and flexilators to execute an application. These clusters can be created and changed dynamically during runtime and they do not share their resources: different applications employ separate components, so the energy and performance guarantees of an application are not violated when new ones are instantiated at the same time. In other words the clusters are *private*, and that is why they are named Private Execution Partitions (PREXEs). In essence, software-centric heterogeneous multi-processor platforms are based on time multiplexing of its resources, while PREXes make the SiLago platform a space division multiplexing platform which instead allows for a complete hardware style implementation. This is ultimately how the SiLago framework proposes to address and overcome the efficiency limitations in the state-of-the-art SOCs.

1.2.3 Dynamic Reconfigurable Resource Array - DRRA

DRRA is a coarse grain reconfigurable fabric that targets the computation of data parallel streaming functions, and is heavily oriented towards vector operations. It is formed by specific DRRA blocks (also called cells or tiles), as shown in Figure 1.4. Each block is composed by five main elements: Register File, Address Generation Unit (AGUs), DataPath Unit (DPU), Sequencer and the Switchbox for the Sliding Window Interconnect.
The Register File can have different number of locations depending on the application, but in all cases it is equipped with two read and two write ports. Unlike RFs inside standard ALUs that are driven by general purpose processors, the ones inside the DRRA are provided with dedicated Address Generation Units, or AGUs. AGUs are reconfigurable FSMs that enable streams of data with spatial and temporal programmability: they support all addressing patterns described by two-level affine functions, and allow programmable delays. For the algorithms where address generation is predictable and run-time static, the distributed control of AGUs is

much more efficient than a centralized processor, because it brings a smaller overhead and saves on the cost of address transportation.
The DPU is the core of computation inside the DRRA. It offers significant freedom in design time customization, so its supported operations heavily depend on the target algorithms. The customization of the DPU for Neural Networks is the core topic of this work, so it is described in detail in Section 4.
The DRRA sequencer is mainly a configuration unit but it can also handle control of compile-time static functionalities. Unlike the traditional processor sequencer, the DRRA sequencer has a small local store of 64 words. The fetch-decode-execute path is single cycle, i.e. there are no pipeline stages. The main task of the sequencer is to setup and launch vector operations. This is done by configuring Register Files to source and sink a stream of data in right spatio-temporal pattern, by programming the switchboxes to connect DPU and Register Files, and by programming the correct DPU mode. Which vector operations to perform and with which constraints is decided by loop and/or branch instructions. DRRA is designed to support only minimal branching and loops, so functionalities that are dominated by control sequences are instead mapped to the Flexilators, that are tightly coupled to the Sequencers.
The DRRA intra-regional interconnect scheme consists of a sliding window nearest neighbor connectivity. Each DPU and RF outputs to a bus (bundle of wires) that crosses two columns on each side. In this way, each DRRA block has a connectivity span of five columns, and the overall scheme consists of several overlapping sliding windows. The output busses, horizontal in their orientation, are intersected by input busses, vertical in their orientation. Inside each DRRA cell, at the intersection of input and output bus lie two Switchboxes, one for the RF and one for the DPU. These Switchboxes can be programmed to select the input source from any Register File or DPU in the same column or from the two columns on each side. The Sliding window interconnect and the programmable Switchboxes bring great flexibility in implementing data flows and are a major contribution to the hardware customization provided by the DRRA.

1.2.4 Distributed Memory Architecture - DiMArch

The DiMArch is a fabric dedicated to storage, which provides a large scratchpad memory and a parallel storage access, so as to match the high computation parallelism of the DRRA. DiMArch is also created by array disposition of SiLago cells; each cell takes the space of one or two DRRA rows so that it has enough room for an SRAM memory bank. Just like the RF, DiMArch cells are provided with AGUs that enable flexibility in address generation. The DiMArch banks are glued together by a circuit-switched NOC, whose switches can be programmed. In this way, different SRAM blocks can be clustered to make them look like one larger SRAM. The circuit switched NOC is preferrable for data transfers because it has low overhead when traffic patterns are deterministic. DiMArch blocks do not contain their own sequencer for configuration, but they are handled by the ones inside the DRRA. This connection happens by a packet-switched configuration NOC, which has been chosen because it allows to easily reach any node in the network.

Chapter 2

Artificial Neural Networks

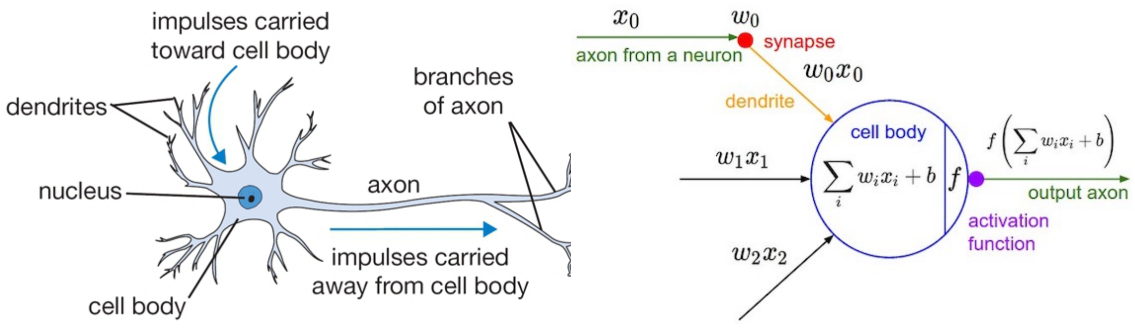
The three algorithms targeted in this work all belong to the field of Artificial Neural Networks (ANNs):

- Convolutional Neural Networks (CNN);
- Long Short-Term Memory (LSTM), a subset of Recurrent Neural Networks (RNN);
- BioSOM, a customized Self-Organizing Map (SOM) for bacterial genome identification.

Aside from their distinctive features and fields of application, all Neural Networks are based on a common founding theory, so it is worth to have a general overview on the concepts that inspired their development. The goal in this introduction is not to provide an extensive, in-depth knowledge on ANNs, but to just present their role in the context of Artificial Intelligence and to outline the basic principles behind their capabilities. The fundamental concepts of ANNs explained in this Chapter are freely adapted from the two surveys [2] and [14]. According to John McCarthy, the computer scientist who coined the term AI, this broad field of study can be defined as "the science and engineering of creating intelligent machines that have the ability to achieve goals like humans do". Most recently, AI has been given a more complete definition as the ability of a system to "correctly interpret external data, learn from such data, and to use those learnings to achieve specific goals and tasks through flexible adaptation" [15]. Within AI lies the large domain of Machine Learning (ML), which is defined as "the field of study that gives computers the ability to learn without being explicitly programmed". Arthur Samuel, the inventor of the term Machine Learning itself, quotes: "A computer can be programmed so that it will learn to play a better game of checkers than can be played by the person who wrote the program" [16]. This key consideration allows to better grasp the difference between ML and standard programming approaches: traditionally, algorithms are written *ad-hoc* to solve specific tasks, and their quality depends directly on programmer's skill and knowledge. On the contrary, in ML humans are only bound to provide a good learning paradigm and data to learn on, then the machine is capable of building its own functionalities that address the problem. To our current knowledge, the human brain is simply the best system for learning and decision-making and that makes it the focus of a relevant area of ML called Brain-Inspired Computing (BIC). It must be stressed out that the aim of BIC is not to simulate the complex biological processes that underline a brain. Instead, BIC starts from existing theories on brain functionality provided by computational neurobiology, and from them derives abstract, simplified models that emulate the human learning process to solve specific tasks. A branch called Spiking Computing is based on the idea that information inside the brain travels in pulses, and is encoded in their amplitude, width and frequency. The interest of this work lies instead on Artificial Neural Networks, the branch that models neural processes only considering the signal amplitudes.

2.1 General Architecture of an Artificial Neural Network

Taking inspiration from the brain structure, all types of ANNs share a common underlying architecture, that consists in a network of neurons and interconnections. To our current knowledge, most of the information processing occurs inside the neuron. Figure 2.2 shows its structure and details some naming conventions. The



(a) Physical structure of a neuron **(b)** Computational model of a neuron

Figure 2.1: Biological Neuron and its ANN model. Both pictures taken from [1]

input and output signals of neurons are called 'activations' and propagate in the network through the 'axons'. An intermediate connection called 'synapse' is found between axons and neurons: it is believed that its task is to describe the relevance of the incoming signal, so it is modelled as a scaling factor called 'weight'. Neurons take the linear combination of all synapses, add a ‘bias’and then process the whole input. The whole structure including a neuron and their synapses is called ‘perceptron’. Research in computational neurobiology suggests that neurons trigger their output axons only if the overall input crosses a certain threshold. Consequently, activation functions are modelled in a similar, highly non-linear way. The more common activation functions are listed in Table 2.1.

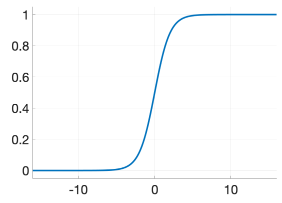
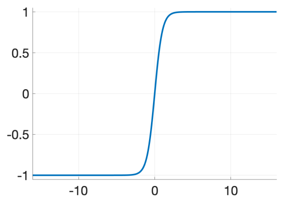
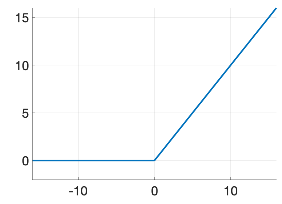
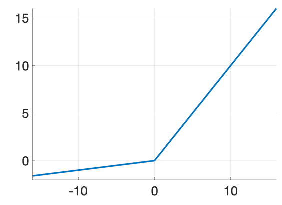
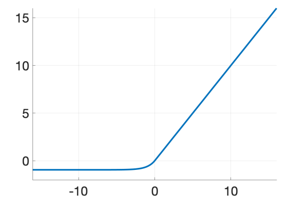
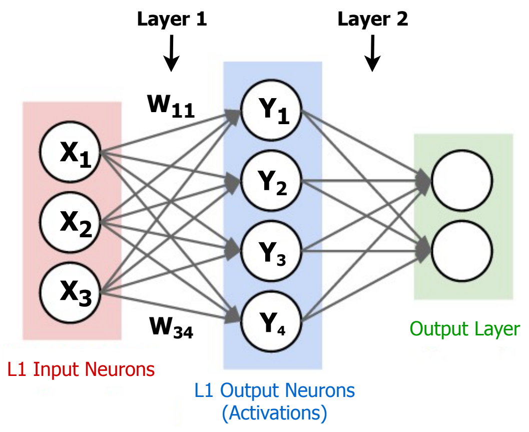
Activator	**Plot**	**Equation**
Sigmoid		$\frac{1}{1+e^{-x}}$
Tanh		$\frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$
ReLU		$\begin{cases} \text{x}, & \text{x}\geq 0 \\ 0, & x<0 \end{cases}$
Parametric ReLU		$\begin{cases} x, & x\geq 0 \\ ax, & x<0 \end{cases}$
ELU		$\begin{cases} x, & x\geq 0 \\ a\left(e^{x}-1\right), & x<0 \end{cases}$

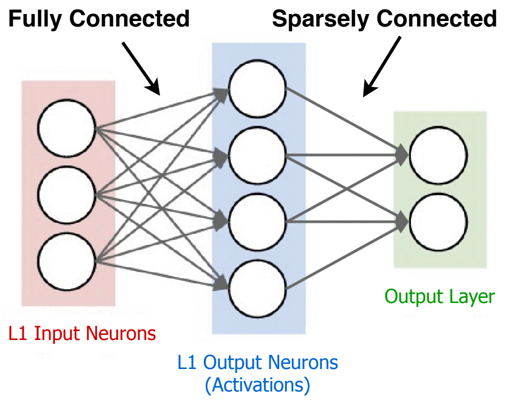
Table 2.1: Activation functions and their equations

The similarity of Sigmoid and Hyperbolic Tangent (Tanh) with real neurons behavior made them a common choice in the past, but then fell out of favor due to the stability problems that they induce during training [1]. In the past few years, the

more modern activators Rectifier Linear Unit (ReLU) and its variants have been found to solve the previous issues and so have quickly taken over in popularity. To date, the ReLU activator is the most common choice for CNNs while LSTM networks still rely on Sigmoid and Tanh.
ANN architectures are organized in neuron layers, as illustrated in Figure 2.2a. This specific pattern has been chosen since it shows similarities with how some



(a) General ANN structure



(b) FC and SC Layers

Figure 2.2: Typical interconnection schemes of ANNs. Both pictures adapted from [1]

parts of the brain are arranged, a well-known example being the visual cortex. From the theoretical point of view, each network has a certain depth associated to it. Considering the ANN as a graph, depth is defined as the longest path (measured in number of arcs) that a signal can cross while going from the input to the output. Given the specific network structure, the depth ends up being equivalent to the number of neuron layers, so in the following the two features will be considered identical.
Input data is collected by the Input layer, processed by neurons and then passed onto the Intermediate Layers. The number and type of axons are fixed when designing the network and do not change dynamically; this is a good example of simplification from the real brain, whose connections evolve over time. Data propagates through layers until it reaches the last one that provides the final outputs (Output Layer). This procedure of giving an input and retrieving the system output response is called ‘Inference’.
A layer is defined as ‘Fully Connected’ (FC) or ‘Multi-Layer Perceptron’ (MLP) when each neuron input is connected to all neuron outputs from the preceding layer. On the other hand, if some of the connections are missing the layer is called Sparsely Connected (SC). Figure 2.2b gives a visual example.

In all cases, each weighted sum x_j entering the neuron is:

$$x_j^{l+1} = \sum_{i=1}^{m} w_{ij}^l y_i^l + b^l \quad \forall\, j = 1, \dots, n \tag{2.1}$$

Where m and n are the number of neurons in layers l and $l+1$ respectively. The relation between neuron outputs and inputs of the next layer is easily expressed as matrix-vector product:

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}^{l+1} = \begin{bmatrix} w_{11} & w_{12} & \dots & w_{1m} \\ w_{21} & w_{22} & \dots & w_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n1} & w_{n2} & \dots & w_{nm} \end{bmatrix}^{l} \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ x_m \end{bmatrix}^{l} \tag{2.2}$$

Or in a more coincise form:

$$\mathbf{x}^{l+1} = \mathbf{W}^l \mathbf{y}^l \tag{2.3}$$

Where l is the layer number, $\mathbf{y}^l$ are the l-th neuron outputs, $\mathbf{x}^l$ the (l+1)-th inputs, $\mathbf{W}^l$ is the l-th weight matrix. The activation function is then applied to each x_j to get the neuron outputs:

$$\mathbf{y}^{l+1} = f(\mathbf{x}^{l+1}) \quad \Longleftrightarrow \quad \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}^{l+1} = \begin{bmatrix} f(x_1) \\ f(x_2) \\ \vdots \\ f(x_n) \end{bmatrix}^{l+1} \tag{2.4}$$

ANNs that include more than three layers (i.e. more than one hidden layer) are defined as Deep Neural Networks (DNNs). To date, DNNs can range between five and more than a thousand layers. Their area of research within ANNs is called Deep Learning.
Once the ANN structure has been defined, it is possible to detail the process that enables decision-making capabilities, which is called 'Training'. It is believed that the natural brain learns by adjusting the influence of synapses over the signals, so that the response to a set of inputs can change. As a consequence, 'Training' translates into adjusting the weights (w_{ij}) to obtain an output response that gets closer to the ideal. The network learning capabilities are thus highly dependent on the quality of the Training input set, which should provide an exhaustive representation of the problem. At first, all weights are initialized to random values. Then, for each input the learning process takes place in three steps:

1. Inference: input is fed to the network and the output scores are retrieved;

2. The distance between the ideal scores and the real ones is called Loss and is evaluated with a Cost (Loss) Function;
3. Weights are updated depending on the Loss to get closer to the ideal behavior.

In many applications, DNNs like CNN and LSTM are faced with classification problems: all possible outcomes are divided into a set of classes, and the network task is to identify the one that matches the input. In practice, the final output Layer needs a different activation function called Classifier, which turns its inputs into a vector of scores (or probabilities); the highest result corresponds to the identified class for the input sample. In such cases, the Loss can be related to the difference between real and expected outcomes, as in this example of Mean Squared Error:

$$L_1(\mathbf{W}, \mathbf{y}_1) = \frac{1}{2}\|\mathbf{y}_1 - \mathbf{y}_{1,ideal}\|^2 \tag{2.5}$$

For simplicity, the output layer result $\mathbf{y}_1^{nl-1}$ is just reported as $\mathbf{y}_1$. The subscript 1 refers to the first input vector. A prime case of classification is image recognition: a number of classes is chosen to cover the whole set of possible inputs, then a picture is given to the network and the output is the estimated class the image belongs to. In these cases the technique used to train DNNs is the iterative algorithm called Gradient Descent. The Cost function can be taken on one single input at a time, averaged over a batch of inputs or even over the whole training set. Each solution has a different trade-off in terms of speed and stability, but in the following the most general case will be used. The average over all $\mathbf{x}$ is done to obtain a figure of merit that accounts for all inputs at the same time. Input data is not under the algorithm's control, so the only free parameters that can be changed are the w_{ij}. As a consequence, L can be considered only a function of the network weights.

$$L(\mathbf{W}) = \frac{1}{N}\left(L(\mathbf{W}, \mathbf{x}_1^0) + L(\mathbf{W}, \mathbf{x}_2^0) + \cdots + L(\mathbf{W}, \mathbf{x}_N^0)\right) \tag{2.6}$$

Where $\mathbf{W} = \{\mathbf{W}^0, \mathbf{W}^1, \ldots, \mathbf{W}^{nl-1}\}$ is the set of all network weights and N is the number of training inputs.
By iteration, the Gradient Descent finds a series of weight sets $\mathbf{W}_0, \mathbf{W}_1, \ldots, \mathbf{W}_A$ such that the corresponding series of Losses $L_0, L_1, \ldots, L_A$ converges to its global minimum. This condition means that the network has reached the lowest error for all inputs and so has completed its learning process. It can be proved that, for each step a, the weight correction that decreases L the fastest is:

$$\mathbf{W}_{a+1} = \mathbf{W}_a - \alpha_a \nabla L\left(\mathbf{W}\right) \quad \Longleftrightarrow \quad w_{ij}^{a+1} = w_{ij}^a - \alpha_a \frac{\partial L}{\partial w_{ij}} \quad \forall\, i, j \tag{2.7}$$

Where α_a is the 'Learning Rate', a coefficient whose value can change depending

on the iteration step.
The most efficient method to compute all partial derivatives is Backpropagation. Starting from the gradient of L with respect to the output values $\partial L/\partial y^{nl-1}$, the derivative chain rule allows to obtain all the $\partial L/\partial w_{ij}$ of the last layer. A similar computation also brings the $\partial L/\partial y^{nl-2}$, which are equivalent to the output derivatives $\partial L/\partial y^{nl-1}$, but related to the previous neuron layer. From this point, the same procedure can be applied to layer $nl-2$ to get the values of layer $nl-3$ and so on, moving backwards along the network until all weights are covered.
Compared to Inference, Training a DNN requires a lot more computation, storage and precision, so in most cases it is carried out on HPC (High performance Computing) systems. Since most embedded devices are driven by low power requirements, they are restricted to the inference only. For the same reasons, the Silago platform is limited to Inference as well.
Several types of learning approaches exist, with the main ones being:
Supervised learning: all training samples are labeled, i.e. their corresponding output class is known and used to drive the update of weights. This is the most common procedure.
Unsupervised learning: no training inputs are labeled, so the network has to infer on its own groups of inputs (clusters) that share similar features.
Semi-supervised learning: It is based on the unsupervised approach, but here a small amount of the input set is labeled so that data clusters can be associated to actual classes.
Reinforcement Learning: a radically different method from supervised and unsupervised learning. An agent (the software) is set to achieve a particular goal by interacting with a given environment. There is no training data available, so the agent must learn through a trial and error approach by taking actions within the environment and evaluating the corresponding rewards. Over time, the agent learns the best policy to maximize the long-term total reward. A common application example are games: the environment is the set of game rules and the agent is a player that tries to win. The result of learning is the optimal strategy, i.e. the best actions to take at every possible game state that ensure victory.
This concludes the overview on the general properties of ANNs, so now it is worth shifting the focus on Deep Neural Networks: this domain has become the most popular branch of AI to date and is also the area including CNN and LSTM networks, so the following chapter 2.2 takes a more detailed look at its features. Self-Organizing Maps are a distinct class of ANNs, so they are separately covered in chapter 2.3.

2.2 Deep Neural Networks

DNNs were first proposed in the 1960's, around twenty years after the theory on ANNs was laid out. However, the sheer lack of computational power prevented their use in practical applications until 1989, when the network LeNet was introduced for hand-written digit recognition [17]. A greater attention towards DNNs has risen up since the 2010's thanks to two groundbreaking innovations: A speech recognition system from Microsoft (2011) and AlexNet (2012) [18], a network that outperformed all other machine learning approaches in image recognition tasks. In the recent years, DNN have kept growing in popularity, to the point of becoming the main research target in the whole AI. Moreover, they deliver state-of-the-art performance in real applications for many different fields, ranging from multimedia to genomics. A major example of the success of DNN in the field of image classification and speech recognition is the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [19]: Algorithms are trained on a common database of 1.2 million images and 1000 classes, each one corresponding to an object. In the inference phase, they are given new unseen images and they must detect their content. As already remarked, in 2012 AlexNet (a CNN) led to a breakthrough reduction of error rate; it can be seen on Figure 2.3 that after only three years NNs managed to outperform human detection capabilities.

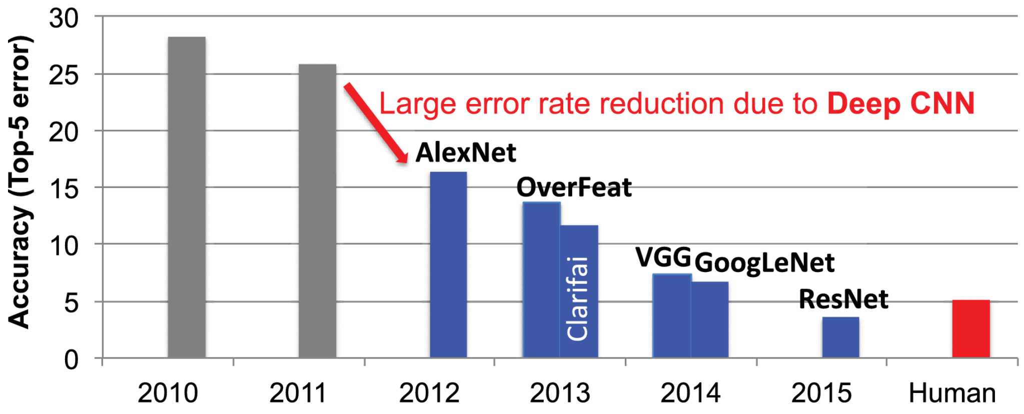


Figure 2.3: ImageNet Competition Winners until 2015. Taken from [2]

The research work in [20] goes into detail on explaining the greater learning capabilities that are inherent to Deep Neural Networks. It is thought that humans build their interpretation of external stimuli (like images and speech) by intuitively decomposing the raw sensory data into multiple sub-elements at different levels of abstractions, and then putting them all back together to gain a very high-level representation that can be associated to a known category. This association is what ultimately provides meaning to the stimuli from the external world. Moving to ANNs, theoretical results point towards the inherent limitations of shallow networks in deriving complex interpretations of an input space: it is demonstrated that the required number of computational elements to describe a given function increases exponentially as the network depth decreases. In other words, solving AI tasks with a comparable degree of reliability as humans is remarkably more difficult for shallow ANNs than for DNNs. These results strongly suggest that deep architectures are needed to build more complex functions that are able to describe high-level abstractions in a similar way as the human brain does. It has been shown that the initial layers of DNNs are only capable of extracting lower-level features. By progressing through the network, these features get combined into more and more complex abstractions, until a very high-level representation is matched to a known category (class). Following the example of image recognition, pixels are fed into the first layer, which detects simple elements such as lines and edges. At subsequent layers, these features are then merged into simple shapes, which are then further combined into sets of shapes. In the end, the network obtains complex representations, tries to match them with particular objects or scenes (the classes) and outputs the probabilities of detection. Ultimately, it has been established that the deep feature hierarchy is what enables DNNs to achieve superior performance in all tasks. This general approach to learning is universal and provides DNNs with the ability to accomplish most kinds of AI tasks; starting from this common ground, several types of DNNs with different kinds of layers and connectivity have been developed to specialize on specific tasks. The most widely used variants of DNNs are precisely the CNN and LSTM networks that are the object of this work, so they are presented more in detail in Sections 2.2.1 and 2.2.3.

2.2.1 Convolutional Neural Networks

Convolutional Neural Networks are DNNs designed with a structure similar to the human visual processing system, which makes them highly optimized for learning abstractions of 2D and 3D images. For these reasons, CNNs are most suited for all kinds of tasks in the field of Computer Vision, especially image processing.
One of the main benefits of CNNs is the reduced need for Fully Connected layers as compared to standard DNNs. In a FC layer, each output neuron is connected to all input neurons, requiring a higher amount of storage and computation. Such degree of complexity is not needed in CNNs, that are instead modeled as in Figure 2.4.

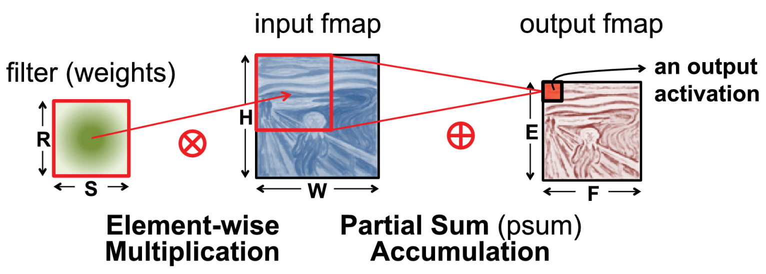


Figure 2.4: CNN Neuron Layer

In standard 2-D image recognition algorithms, the presence of a characteristic feature (for example, a line or an edge) is detected by scanning the picture with a filter, i.e. performing a convolution. Each output pixel is only associated to a limited input region, and its magnitude tells whether the feature has been found or not in that area. The very same concept has been applied to CNN, but in this case pixels are replaced by neurons and filters are made of weights. Inputs and outputs of each layer are called Input Feature Maps (IFMs) and Output Feature Maps (OFMs) respectively. Each OFM neuron only depends on a restricted window of IFM neurons (called the Receptive Field) through a fixed set of weights. The same weight values can be shared to compute all OFM elements, introducing a deep level of structured sparsity that drops down the storage requirements. On the other hand, CNNs generally require convolutions on a higher dimensionality than standard 2-D image processing. Figure 2.5 is included as a visual reference. The raw

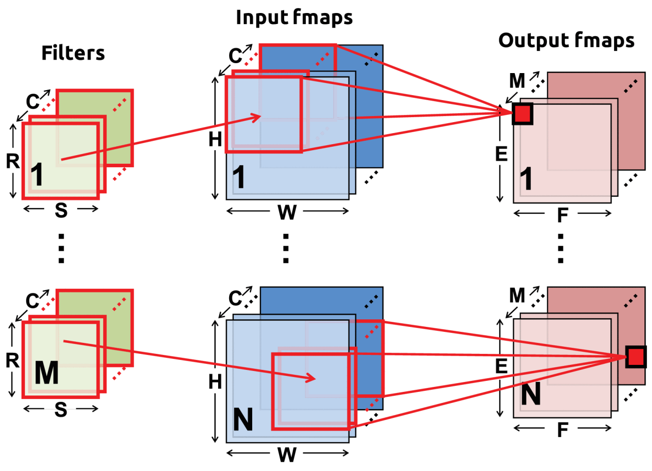


Figure 2.5: Extended dimensionality of CNN convolutions

input sensory data can be split into C different IFMs, each one called a channel. A stack of C filters (called a 3-D filter) is applied to each input channel, and convolution results on the same area across all channels are summed up to become one entry in the Output Feature Map. The use of M different 3-D filters on the same IFM stack gives rise to M different OFM channels. Just as in standard techniques, filters allow to detect specific features, so in CNN the presence of multiple cascaded convolutional layers allows to combine feature maps towards higher and higher levels of abstraction. The other big novelty introduced by CNN is that programmers do not need to design filters and the way of combining them to detect more complex objects, because the network is capable of learning the parameters on its own through training.

2.2.2 CNN structure

A typical CNN requires the presence of several convolutional layers, nonlinear activations and 1-3 FC layers in the end that are linked to the final classifier activation. In addition to those, it may also include some optional layers like Pooling and Normalization. Normalization is introduced to keep a balanced value distribution for the neuron layers inputs, which improves the training process and the accuracy during inference. The current state-of-the-art method is Batch Normalization (BN) [21]. Each neuron input $\mathbf{x}^l$ is rescaled to a distribution of mean $\mu = 0$ and variance $\sigma = 1$ with:

$$\mathbf{x}^l_{BN} = \frac{\mathbf{x}^l - \mu}{\sqrt{\sigma^2 - \epsilon}}\gamma + \beta \tag{2.8}$$

During training the values γ and β are learned, while μ and σ^2 are computed for each batch of inputs $\mathbf{x}^l_1, \ldots, \mathbf{x}^l_m$. For the inference, mean and variance are instead pre-computed for each layer basing on the whole input data set. This way, all coefficients involved in BN for inference are known a priori, so the operation becomes a simple linear transformation. It must be stressed out that BN must be performed just before the nonlinear activators to have effect; if it gets placed after the neurons, it can be folded into the weight matrices of convolutional/FC layers and result in no additional computation.
Pooling, also known as sub-sampling, introduces invariance to shifts and distortion of features and also reduces the OFMs size. The OFM is divided into non-overlapping blocks (receptive fields), and all values within each block are combined together into a single output value. The more common operations are maximum extraction or average.
The classifier function that is used for CNN is the Softmax. Given the input to the final layer $\mathbf{x}^{nl-1} = [x_1, x_2, \ldots, x_n]$, the output scores are computed as in Equation 2.9.

$$y_i = SM(x_i) = \frac{e^{x_i}}{\sum_j^n e^{x_j}} \qquad i = 1, \ldots, n \tag{2.9}$$

Softmax turns all inputs into positive values ranging from 0 to 1. Considering also the property:

$$\sum y_i = \frac{\sum_i^n e^{x_i}}{\sum_j^n e^{x_j}} = 1 \tag{2.10}$$

Softmax can be seen as an operator that normalizes its inputs into a probability distribution; output classes are then associated to probabilities, and the highest score is matched with the winning class. The non-linearity of the exponentials tends to saturate the highest scores towards 1 and to squish the lowest towards 0, so that the function acts like a "softened" version of the *Max* function (hence the name Softmax).
Figure 2.6 resumes all basic components of a CNN along with their main functionalities.

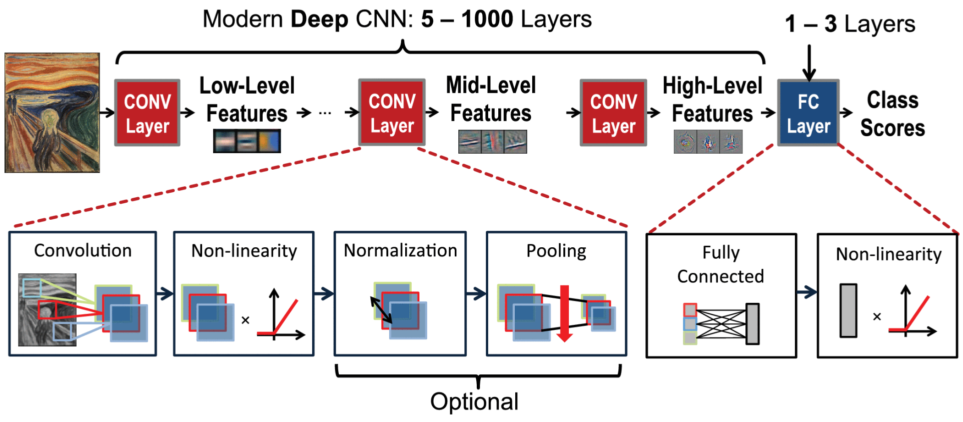


Figure 2.6: Typical CNN Architecture

2.2.3 Recurrent Neural Networks and LSTM

This class of DNN is defined by its peculiar structure that makes it particularly suited for the world of audio and text processing: RNNs are applied with good success in speech and audio recognition, as well as machine translation, natural language processing and audio generation.
The main intuition behind RNNs is that the understanding of written or spoken language is a process that lasts over time. While having a conversation or reading a text, the meaning of each word and sentence depends on the understanding of all previous ones. In other terms, the human language inherently requires memory to combine past information with the present. Due to their limited structure in which one output exclusively depends on one single input, standard DNNs cannot provide this functionality. This led to the creation of Recurrent Neural Networks, in which the results of an activation layer are fed back to its input. The looped structure can also be unrolled in multiple copies of the same activation layer, each taking as input a different x_t and the result from a predecessor. Each output sample is fed to a classifier (e.g. Softmax) just like in all other DNNs. Both the standard and unrolled structures are pictured in Figure 2.7.

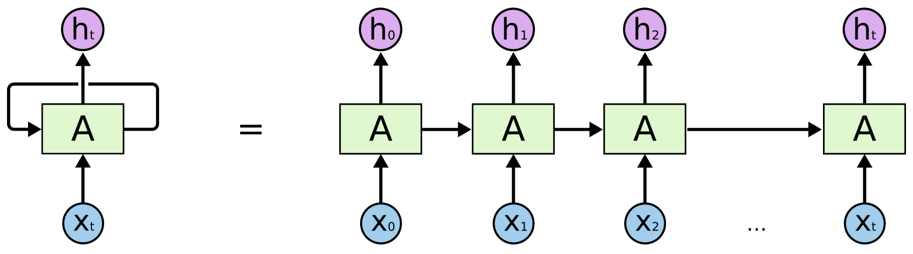


Figure 2.7: Rolled and Unrolled RNN basic blocks

The inside of a layer also becomes more complex. The so-called Jordan version of RNN activators loops the output y_t back to the input, while the Elman version reuses the hidden output h_t. The whole RNN activator structure is expressed in Equation 2.11:

$$\begin{aligned} h_t &= \sigma_h\left(w_h x_t + u_h \mathbf{f_{t-1}} + b_h\right) \\ y_t &= \sigma_y\left(w_y h_t + b_y\right) \end{aligned} \qquad \mathbf{f_{t-1}} = \begin{cases} h_{t-1}, & Elman \\ y_{t-1}, & Jordan \end{cases} \tag{2.11}$$

Where the subscripts t and $t-1$ denote the present and the previous time instant respectively. Independently on the chosen model, the feedback loop allows the network to retain information from the past sequence frames in order to understand the meaning of the current one. Standard RNNs however are affected by a relatively short context window, which prevents them from learning long-term dependencies. Long Short-Term Memory networks have been designed in order to overcome this limitation, at the cost of a more complex internal structure, reported in Figure 2.8.

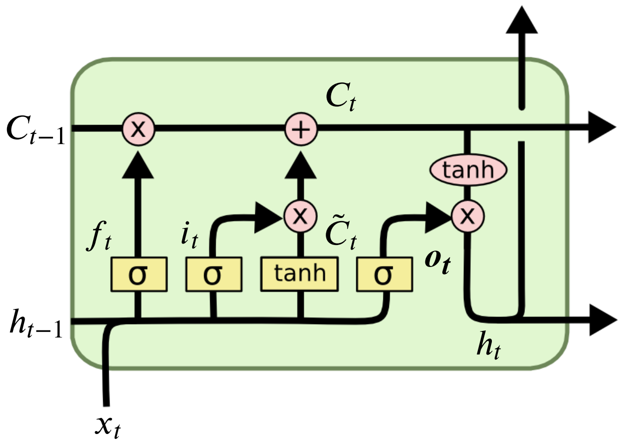


Figure 2.8: Elman LSTM Activation Block

Its constituting equations are:

$$\begin{aligned} f_t &= \sigma\left(W_{fx} x_t + W_{fh} h_{t-1} + b_f\right) \\ i_t &= \sigma\left(W_{ix} x_t + W_{ih} h_{t-1} + b_i\right) \\ o_t &= \sigma\left(W_{ox} x_t + W_{oh} h_{t-1} + b_o\right) \\ \widetilde{C}_t &= tanh\left(W_{hx} x_t + W_{oh} h_{t-1} + b_o\right) \\ C_t &= f_t \circ C_{t-1} + i_t \circ \widetilde{C}_t \\ h_t &= o_t \circ tanh(C_t) \end{aligned} \tag{2.12}$$

The behavior of an LSTM block is ruled by the four activation gates (yellow operators in the picture): the output values of sigmoid and tanh can decide if a certain state information is allowed to propagate or is filtered out. The vector C_t retains information on the Cell t, so it is named Cell state. The Forget Gate f_t is used to control how much of the previous Cell state C_{t-1} must be remembered. Adding new information to C_{t-1} requires two components: the Candidate Gate $\widetilde{C}_t$ proposes new values to add, and the Input Gate i_t decides which of them should be actually used. In the end, the previous state C_{t-1} receives the new information and becomes the Current State C_t that is propagated to the next block. Finally, the Output Gate o_t acts on a filtered version of C_t to decide the block output h_t.
The standard LSTM structure comes with two small variations, the 'Peephole LSTM' [22] and the Gated Recurrent Unit (GRU) in [23]. Since they share the same working principles as basic LSTM, they are not further detailed.

2.3 Self Organizing Maps and BioSOM network

The Self Organizing Map is an approach introduced by Teuvo Kohonen in 1990 [24], and it constitutes a completely different approach from other ANNs. The basic computational nodes are still neurons, but instead of being arranged in layers, they form a 2-D grid of hexagonal or rectangular shape called 'Map'. The working principle takes inspiration from the fact that different functionalities of the brain (such as vision, hearing, speech) are mapped to different spatial locations in the cerebral cortex. Accordingly, SOM learn over the training data by mapping common input features to localized areas within the map. In this way, the spatial location of cells corresponds to a particular domain (or cluster) of input signal patterns.
A visual representation of a SOM array and the data clusters it creates is given in Figure 2.9:

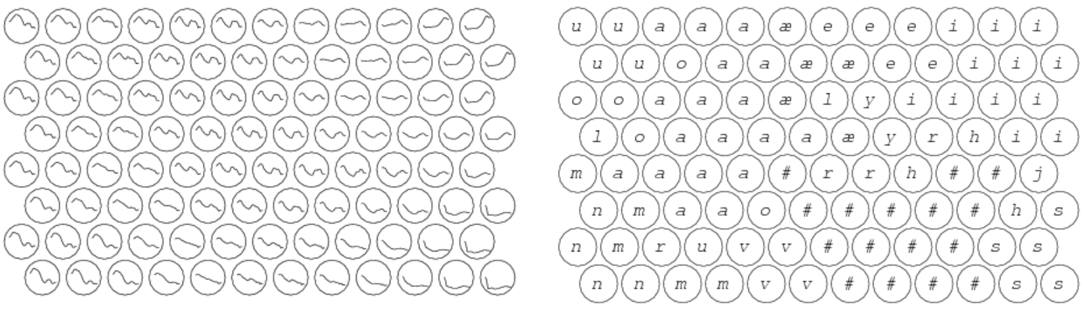


Figure 2.9: Left: Models of acoustic spectra of Finnish phonemes. Right: clustering of models into phonemic classes. Picture taken from [3]

The name Self Organizing Map follows from the type of unsupervised learning that SOM have adopted, that is called Competitive Learning. In the standard training techniques employed for DNNs, all synapses (weights) "cooperate" by adjusting their values to minimize a given Loss function. In SOM instead neurons compete in the right to respond to a subset of the input data, increasing their specialization. This ultimately leads to localized groups of neurons that corresponds to clusters of similar features in the input. Each node (neuron) is associated with a weight vector of the same size as each input sequence. The map topology is fixed, so training consists in adjusting the weight vectors to the input data without spoiling their positioning. Thus, the self-organizing map describes a mapping from a higher-dimensional input space to a lower-dimensional map space. Once trained, the map can classify a vector by finding the node with the closest (smallest distance metric) weight vector to the input space. An example of SOM behavior can be shown directly by explaining the target algorithm BioSOM.

2.3.1 BioSOM

BioSOM is an SOM-based neural network that targets bacterial genome identification. This algorithm retains all main features of standard SOM, but inlcudes some minor customisations. All the content provided in this chapter is based on a previous mapping of BioSOM on Silago [25].
The main element that distinguishes BioSOM is the structure, which is not a 2-D grid but a circular array: this choice is motivated by the nature of genomic data, which are a continuous circular stream with no start nor end. Since the input stream is split in multiple portions, only a closed-loop structure can prevent edge effects. The whole bacterial genome to analyze is divided into an Input Sequence IS, and each sequence element is a small DNA portion modelled as a vector containing M nucleotides. Each nucleotide can take a value among the set $\{A, T, C, G\}$. N neurons are arranged in the circle and each of them is assigned a specific position and a set of M weights matching in size with the inputs. Effectively, BioSOM can be represented as a $N \cdot M$ weight matrix.
Before the training starts, all neuron weights are initialized to random values. An input vector I from the IS is then correlated to all neurons. The set $\{A, T, C, G\}$ is mapped into couples of coordinates (X, Y) with equal distance from the origin, so that a distance function like the sum-of-squared-difference can be used to determine how "close" each neuron is to I. The neuron with the strongest correlation wins: this means that the weights of all neighboring neurons are updated to get closer to the winner and create competition. The update factor β decreases exponentially with the distance from the winner, to restrict the specialization process only to a small region around the winning neuron. All inputs I inside the IS are processed in the same way, and at the end the SOM contains a spatial map of the salient DNA features. The training process is reported in Algorithm 1.
In practical uses, a separate BioSOM is used for each bacterial strain of interest. For inference, an unknown genome is extracted and the corresponding new IS is

correlated to all networks. Tthe BioSOM network that provides the minimum overall distance corresponds to the inferred strain.

Algorithm 1: Pseudo code SOM learning and inference for genome identification

```
1  **Algorithm part 1** *SOM training for one bacterial genome;*
2  **Input** *N: number of neurons initialization;*
3  **Input** $I = [i_1, i_2, \ldots, i_m]$*: Input Vector;*
4  **Input** $IS = \{I_1, I_2, \ldots, I_S\}$*: Sequence of Input-Vectors, Each IS represents
     one bacterial genome;*
5  **Input** $W_{i,j} = M \times N$*:Weight Matrix for one bacterial genome;*
6  $\beta_{min} = 0.01$; $decay\_factor = 0.99$; $\beta = 1.0$
7  **for** $I_k \in IS$ **do**
8      $dist_{min} = \min_{j=1\ldots N} \left(\sum_{i=1}^{M} |I_{k,i} - W_{i,j}|\right)$
9      $j_{min} = j$ where $dist_j = dist_{min}$
10     **for** $j \in \{1 \ldots N\}$ **do**
11         $dist = \frac{N}{2} - ||j - j_{min}| - \frac{N}{2}|$; //toroid distance
12         $W_j = W_j - \frac{\beta}{2^{dist}} (W_j - I_k)$ ;
13     **end**
14     $\beta = min(\beta \cdot decay\_factor, \beta_{min})$;// decay $\beta$
15 **end**
16 **Algorithm part 2** *SOM inference;*
17 **Input** $TIS = \{I_1, I_2, \ldots, I_S\}$*: Test Input Sequence of Input Vectors for
     which the bacterial genome is to be identified;*
18 **Input** $W_{r,j,i} = R \times N \times M$*: Weight Matrices for R bacterial genomes;*
19 $Inferred\_r$ is $r$ with:
20 $score = \min_{r=1\ldots R} \left[\sum_{k=1}^{S} \min_{j=1\ldots N} \left(\sum_{i=1}^{M} |I_{k,j} - W_{r,i,j}|\right)\right]$
```

Chapter 3

Related Work

This Chapter is dedicated to a review of previous researches on the implementation of Sigmoid, Hyperbolic Tangent, Exponential and the Data Compression Engine for CNN. All the proposed methods are shortly resumed and evaluated basing on the needs and constraints of the DRRA cell. The final design decisions for the DRRA implementation are reported in Chapter 4.

3.1 Exponential Implementation

Three implementations for the Exponential have been reviewed. The work in [4] proposes a Taylor expansion of the sixth order, that follows Equation 3.1 and the block scheme in Figure 3.1.

$$e^x \simeq e^a(d_0 + x(d_1 + x(d_2 + x(d_3 + x(d_4 + xd_5))))) \tag{3.1}$$

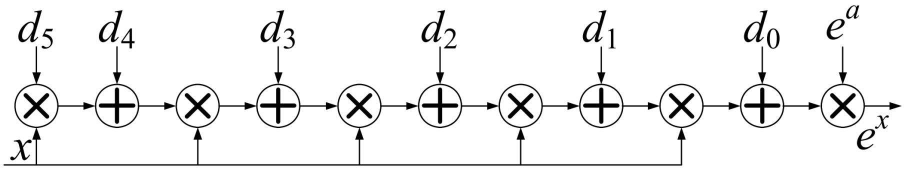


Figure 3.1: Block scheme of Taylor expansion for exponential. Taken from [4]

This approach has not been considered suitable for the DRRA cell, because the dataflow paradigm of the fabric provides the best performance with pipelined operations: a chain of several MAC stages inside each DPU would entail too much overhead in area, power and latency. The other research in [5] compares two different methods: parabolic synthesis and CORDIC-based. Results from this work prove that the parabolic approach is better performing both in area and frequency, so the CORDIC implementation has not been taken in consideration from the start. Parabolic synthesis is a general method that approximates a function by multiplying a series of parabolic functions $s_i(x)$, as in Equation 3.2. The proposed implementation reaches the fourth order, i.e. uses all the subfunctions up to $s_4(x)$.

$$f(x) = s_1(x) \cdot s_2(x) \dots s_n(x) \tag{3.2}$$

As can be seen from the block scheme in Figure 3.2, too many adders and multipliers would be needed for the implementation, which would cause the same issues as in Taylor expansion methods. For this reason, the parabolic synthesis has been excluded as well.

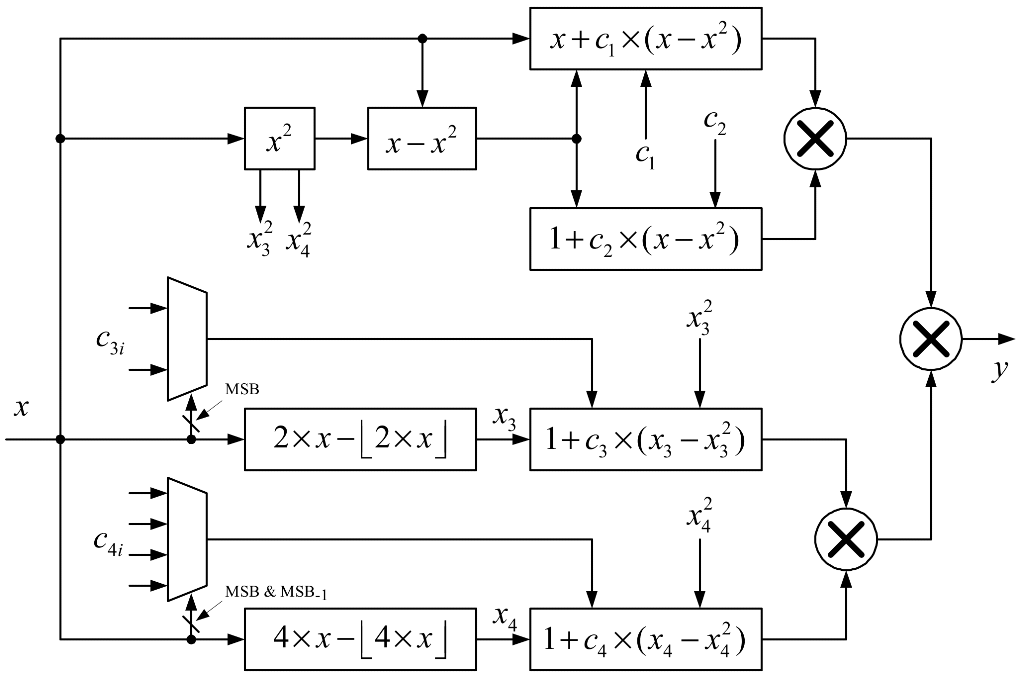


Figure 3.2: Block scheme of parabolic functions for exponential. Taken from [5]

A completely different method is presented in [6]. The exponential is approximated as in Equation 3.3:

$$e^x = 2^{log_2 e^x} = 2^{(log_2 e)\cdot x} \simeq 2^{1.44x} \tag{3.3}$$

A further simplification comes from $2^{1.44x} \simeq 2^{1.5x}$, which allows to avoid a multiplier since $1.5x = x + x/2$. Once $1.5x$ has been derived, it is possible to split the power of two as shown in Equation :

$$2^y = 2^{\sum_{j=-fb}^{ib-1} 2^j \cdot b_j} = 2^{\sum_{j=0}^{ib-1} 2^j \cdot b_j} \cdot 2^{\sum_{j=-fb}^{-1} 2^j \cdot b_j} = 2^{(integer\ part)} \cdot 2^{(fractional\ part)} \tag{3.4}$$

Since the first term is an integer power of two, the Equation can simply be interpreted as a bit shift on the fractional power of two. On the interval [0,1], the power of two can just be approximated as $2^x \simeq 1 + x$, so the exponential is ultimately derived by computing $1 + (fractional\ part)$ and then by shifting it with a barrel shifter driven by the integer part. If x is positive, the input is left-shifted, otherwise it is right-shifted. The overall scheme is the one in Figure 3.3. This implementation is surely promising because it requires the least amount of hardware among the reviewed options and especially because it avoids the multipliers, but it is still less efficient than the final adopted method presented in Section 4.6, which reuses all the hardware already included in the DPU.

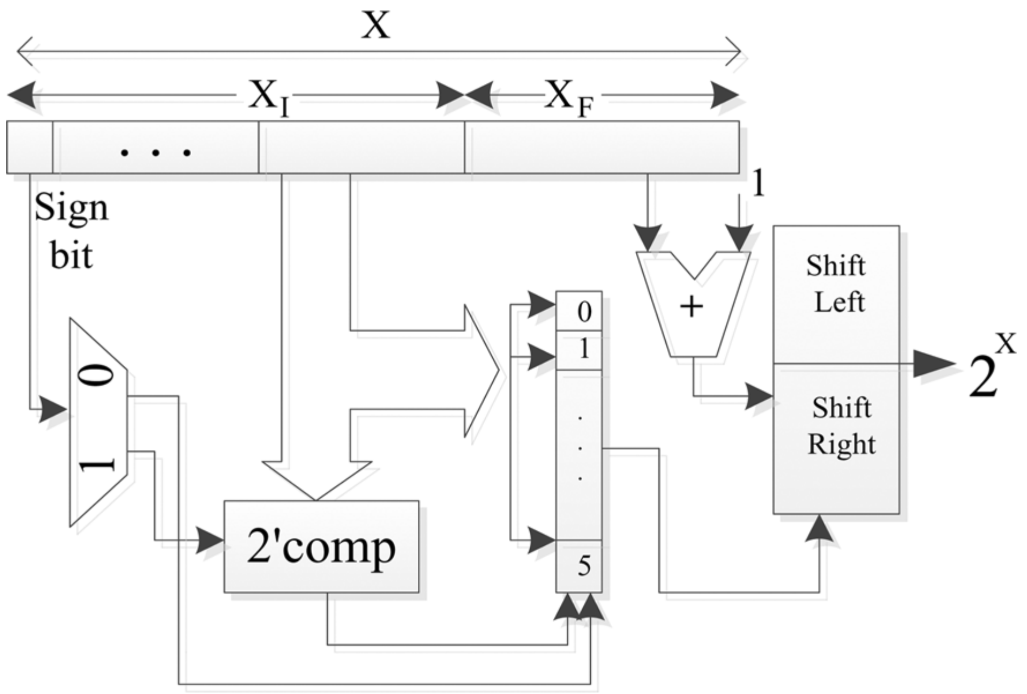


Figure 3.3: Block scheme of 2-power based exponential. Taken from [6]

3.2 Sigmoid and Tanh Implementation

In the recent literature, different approaches to the implementation of sigmoid have been presented: [26] Proposes a Taylor expansion-based approach that, coupled

with the Lagrange form of the error, allows to control the maximum error. The same DSP core that computes the input to neurons (matrix-vector multiplication) is reused for the nonlinear activation. To obtain the first n orders of the Taylor expansion, the same multiplier-adder chain is reused n times in a loop. Since the chain has input and output registers and a 1-stage pipeline, the overall latency is equal to $3n$.
Just like for the Taylor-based exponential, the chain of multiple MAC stages brings too much overhead, so this option must be discarded. On the other hand, the proposed idea of reusing the same components used in MAC also for the activations is particularly suited for SiLago, since the DPU is designed to execute only one type of operation at a time.
The Sigmoid implementation in [7] is derived by directly applying the function definition: first the term $1 + e^{-x}$ is computed, and then it is divided by 1. The exponential is implemented with the same method from [6] that has already been analyzed in Section 3.1. The RTL scheme is given in Figure 3.4.

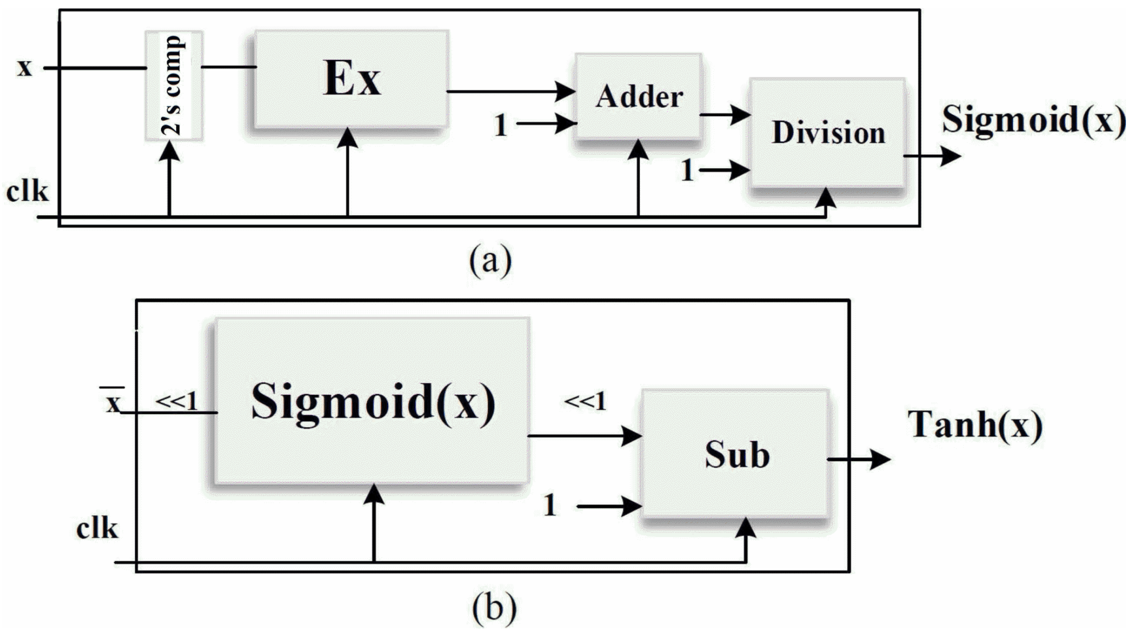


Figure 3.4: Block scheme of Exp-based Sigmoid and Tanh. Taken from [7]

This solution would be viable because the DPU has to contain a pipelined divider for the Softmax layer. However, each neuron activation would require a division, which is quite costly in terms of power. Considering that activations are required within each hidden layer of a network and that there is an ongoing trend of DNNs becoming deeper and deeper, the high power cost of division would simply not be affordable.
Other methods for approximating sigmoid-like functions are reviewed in [27].

Lookup Table (LUT): From the set of all inputs, a subset of uniformly spaced points is selected, and then each one of them is directly associated to a corresponding output value. LUTs can be obtained by storing the outputs inside a memory and addressing it with the inputs, or by combinatorial logic (a decoder). The latter approach is generally better because logic optimizations can be carried out on inputs that share the same output values. Either way, higher accuracy requires a higher number of bits Nb, but the number of LUT entries depends exponentially on Nb. As a result, area increases exponentially with the accuracy, and this makes the approach practical only for low-quality approximations.
Range addressable LUT (RALUT): a RALUT is a LUT in which one output value can be associated to an arbitrary range of inputs. This means that RALUTs allow to define non-uniform partition intervals: output sections with higher slopes can be better approximated by smaller intervals, while flatter parts can be covered by broader ones. For these reasons, RALUTs are able to provide higher precision and to potentially reduce the number of entries, but at the cost of an additional layer of comparators that assign inputs to their predefined range.
Piece-Wise Linear Interpolation (PWL): A function is approximated by a set of straight lines, instead of constant values. Each line is obtained by linear interpolation of the output over predefined intervals. Two tables store the values of slope and offset for each interval. If these intervals are all equal, then PWL is based on standard LUTs and is called Uniform PWL, UPWL. If intervals are non-uniform, PWL are instead based on RALUTs and are defined as Non-Uniform PWL, NUPWL. These solutions yield the best precision compared to LUT/RALUT, since two degrees of freedom (slope and offset) are available instead of just one (offset):

$$PWL : y(x) = m(x) \cdot x + q(x) \qquad LUT\ based : y(x) = q(x) \tag{3.5}$$

As visible from Equation 3.5, PWL approaches have the downside of requiring two LUTs/RALUTs plus a multiplier and an adder to build the straight line, so they have a considerably higher area footprint.
Piece-Wise Nonlinear Interpolation: the core idea is the same as PWL, but instead of straight lines polynomials of degree $n > 1$ are used.

$$y(x) = m_n \cdot x^n + ... + m_2 \cdot x^2 + m_1(x) \cdot x + q \tag{3.6}$$

The key problem is the same as Taylor expansions, several chained multiply-add units are too costly in terms of area, power and delay.

3.3 Compression Engine

Data compression in CNNs is a hardware technique that aims at reducing memory bandwidth and number of computations by exploiting statistics on network parameters and intermediate results (feature maps).

The possibility of using compression for feature maps has opened up with the introduction of the Linear Rectifier *ReLU* (Equation 4.1) as a neuron activator: the function replaces negative values with zeros, and in doing so it introduces sparsity inside Output Feature Maps (OFMs). This advantage has been one of the main reasons for popularity of *ReLU* over the more traditional activators as Sigmoid and Tanh. As OFMs move through the network and cross neuron activation layers, more and more entries are flattened out: [28] reports that the average OFM sparsity in *AlexNet* increases from almost 40% of Layer 2 to around 75% of Layer 5; in the case of *VGG-16*, it ranges from 48% up to 88% . Several encoding techniques can exploit this widespread sparsity to reduce the effective information content in OFMs.
Compression is also viable for network weights. It is well established that DNNs are usually oversized [29], with a prime example being AlexNet. While this allows an easier training, it also causes unnecessary overheads in memory footprint and computation cost for inference, which is a considerable problem for resource-limited embedded platforms. A solution has been found in a technique called *Network Pruning* [30]: identifying and removing the set of redundant parameters that contribute less to the accuracy. Its result is the introduction of sparsity in weight matrices as well.

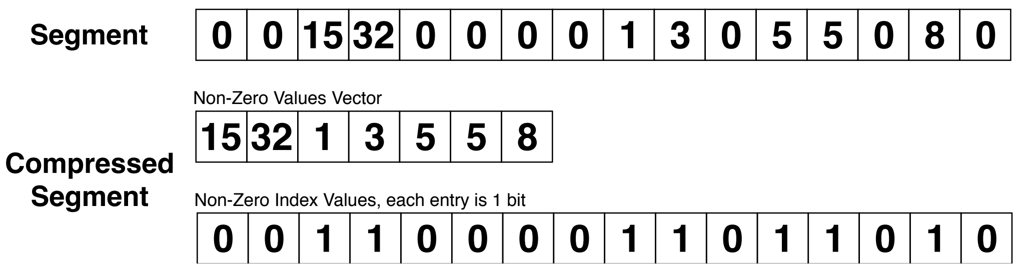


Figure 3.5: NZIV Example

In both cases of OFMs and network weights, compression mechanisms are only aimed at exploiting sparsity in matrices, so in the field of CNNs it is more appropriate to talk about Zero Compression.
As resumed in [14], The most common CNNs have a total number of weights that ranges from hundreds of thousands (LeNet-5, 431k) to around a hundred million (VGG-16, 146M, Overfeat 138M). The number of MAC operations can reach several billions. Such large networks certainly require considerable energy for computation, but the overall consumption is dominated by memory: the large amounts of parameters can never fit on-chip, so they require an external DRAM along with its costly accesses. Interconnects are a concurrent cause for the power overhead. This is especially true for DRAM connections that travel off-chip, but also for the on-chip wires between memory L1 and L0, whose consumption is getting more and more

relevant with the scaling of technological processes.
ASIC architectures for CNNs like MIT Eyeriss [31] have focused on reducing DRAM data transfers as they still constitute the main source of power overhead, and in fact they make use of compression/decompression engines at the interface between L2 and L1 memories. The data stored in DRAM and travelling through the off-chip wires is always compressed, in order to reduce bandwidth and number of memory accesses, while it is decompressed when kept inside the SRAM, so it is ready to be sent to register files and then used for computation.

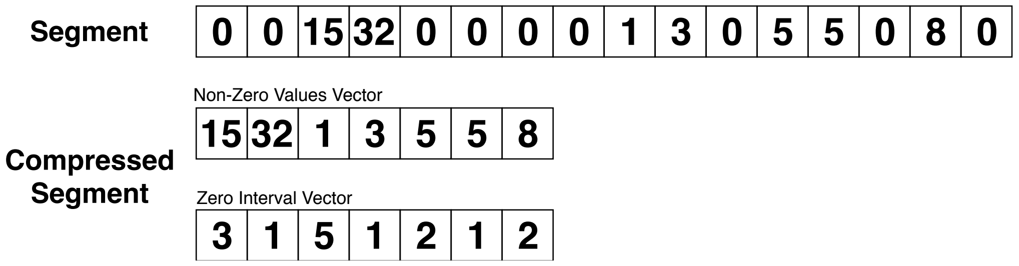


Figure 3.6: ZI Example

Among the many available encoding algorithms for compressing zeros, all recent CNN architectures have adopted the Zero Run Length Encoding (ZRLE), since it provides the best tradeoff between compression efficiency and power consumption. A previous research on a custom fabric for CNNs that is quite close to the DRRA [32] contains a detailed analysis on various algorithms and proves that ZRLE is the most efficient, making it the natural choice for the current iteration of the platform. Three different variants for implementing ZRLE have been found in [28] and [31].
NZIV: The original data stream is split into two vectors: Non-Zero Values (NZV) and Non-Zero Index Value (NZIV). NZV has variable length and holds all non-zero values from the input. NZIV is a Nw-bit vector, where Nw is the number of entries of the original sequence. A 0 is placed in position i if the $i-th$ element of the input data stream is zero, 1 otherwise. Figure 3.5 shows an example.
ZI: this variant uses the NZV as well, but stores positions of non-zero values inside a Zero Interval vector (ZI). Each ZI entry contains the distance between a non-zero value and the following one, so NZV and ZI always have the same size. The ZI encoding is illustrated in Figure 3.6.

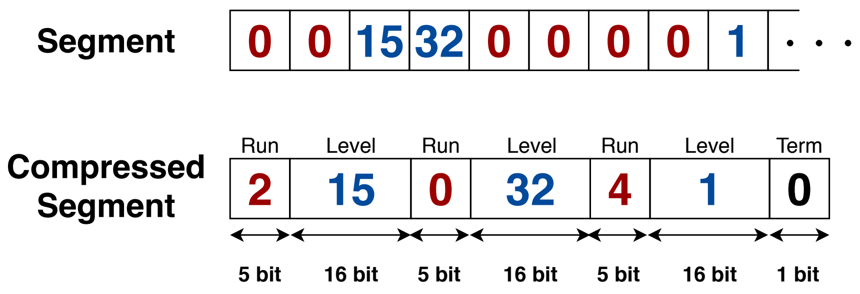

Segment
0 0 15 32 0 0 0 0 1 . . .
Compressed Segment
Run Level Run Level Run Level Term
2 15 0 32 4 1 0
5 bit 16 bit 5 bit 16 bit 5 bit 16 bit 1 bit

Figure 3.7: RLC Example

RLC: this approach differs from the previous ones since it outputs multiple data vectors of fixed size. The input vector is turned into many 64-bit output sequences, each one alternating 5-bit *Runs* to 16-bit *Levels* and ending with a *Term*, as portraited in Figure 3.7. A *Run* contains the number of consecutive zeros, a *Level* stores a non-zero value, while *Term* indicates if the last word is the end of the stream.

Chapter 4

DataPath Unit implementation

The DataPath Unit is the element of the DRRA basic block that is in charge of performing all arithmetic and logic operations. Due to the architectural paradigm of the DRRA, that allows a wide reconfigurability of data streams within the fabric, each DRRA cell (and so each DPU) must be able to implement all operations needed by the target applications. The great variability in size and shape of DNNs requires to change the computation pattern depending on the specific layer, in order to achieve the best efficiency. Finding a general algorithm to map the target DNNs on the DRRA is beyond the scope of this work. Instead, the goal has been providing a set of basic operations that is wide enough to satisfy the needs of whichever mapping is going to be implemented. The three target ANNs of this work (Convolutional Neural Networks, Long Short-Term Memory networks and Self-Organizing Maps) have been broken down into the following basic operations:

- Sum/subtraction
- Multiplication
- Multiply And Accumulate (MAC)
- SOM distance function
- Division
- Maximum/Minimum detection
- Bit shifts
- Non-linear Activation functions: Sigmoid, Tanh, ReLU, Leaky ReLU and ELU
- Exponential

It follows that the DPU must include all basic algebraic operations, which constitute its core functionality, together with several non-linear functions that are specific to ANNs and thus have to be designed *ad-hoc*.
Section 4.1 lists the general features of the DPU. Sections 4.2, 4.3 and 4.4 present the hardware implementation for basic operations such as addition, multiplication, max/min and their combinations. Section 4.5 details the implementation of Sigmoid and Tanh, while Sections 4.6 and 4.7 cover the Exponential and Softmax functions. Once all the individual operations have been described, Section 4.8 explains how they have been merged within the same hardware block.

4.1 General DPU Features

The DPU has been designed at RTL in VHDL language. In previous versions, the optimal number of input and output ports had already been determined as a trade-off between degree of parallelism and the size of DRRA interconnect system. The values have been set to:

- Number of Inputs: 4
- Number of Outputs: 2

While the DPU interface has been fixed at design time, the number of basic computational units has been made compile time dynamic by means of VHDL generics. In principle, their value can be set to any number, but it doesn't make sense to use more components than the maximum number of output ports. In practice, there cannot be more than 2 components of the same type. The following list shows all generics together with their default value:

- Adders - default: 2
- Multipliers - default: 2
- Squash Units - default: 2
- Max/Min Units - default: 2
- Shift Units - default: 2
- Dividers - default: 1

The first component of a kind is bound to work with the first couple of Inputs 0 and 1 and sends results only to Output 0, while the second component of the same type works with Inputs 2 and 3 and outputs to port 1. To clarify, let us take the MAC operation as an example: Multiplier 0 can only take Inputs 0 and 1, while Multiplier 1 can only take Inputs 2 and 3. Adder 0 then gets the result of Multiplier 0 and Accumulator 0 and updates Accumulator 0. The final results is sent to

Output 0. In the same way, Adder 1 uses data from Multiplier 1 and Accumulator 1 to update Accumulator 1, which Outputs to port 1. If there are two adders but just one multiplier, then only one MAC is supported and is always restricted to use Inputs 0 and 1. In general, with this kind of approach the parallelism of composite operations such as MAC or Exponential depends on the number of basic blocks that form them. This setup allows all possible combinations of generics, which result in many variations of the DPU with different parallelism for each type of operation. Such degree of flexibility has been introduced to allow the reuse of the same DPU block for future iterations of the DRRA fabric.
The current version of the DPU only targets fixed point operations. *Nb*, The number of bits for data words, is kept fixed at 16. At first, the possibility for the DPU to change the fixed-point format at runtime was explored. This flexibility was available at the cost of several extra multiplexers to select the right data representation, so in the end the fixed-point format has been turned to a generic in order to save as much area as possible. With $Nb = 16$, the optimal fractional point format has been determined to be $Q4.11$: 1 sign bit, 4 integer bits and 11 fractional bits. This choice is based on the sigmoid activation function, so it is discussed in Section 4.5.
The DPU has input and output registers for data, with the latter ones also acting as accumulators. Every output data that is represented on more than Nb bits is saturated back to Nb bits before being registered. A Saturation Unit is dedicated to each output port, a multiplexer driven by the operation code determines which data has to be saturated and passed at the output.
In the following and in Chapter 8, several variable names and acronyms have been employed, so all of them have been resumed in Table 4.1 for a quick reference.

4.2 Adders and Multipliers

In CNN and LSTM networks computation is dominated by matrix-vector products, which involve MAC operations (Multiply-Add-Accumulate). DPU has to support MAC but also simple additions and multiplications. Since stand-alone adders and multipliers have to be used anyway, it becomes more efficient to implement MAC by reusing them, instead of employing dedicated MAC units. Behavioral descriptions for multipliers and adders have been used, allowing synthesis tools to infer the optimal implementation. With Nb bits data, multipliers are simply $Nb - bits$, while adders have to be $2Nb$ as required by Softmax function (chapter 4.7). This size for the adders also has the upside of allowing more precise MAC: the multiplier outputs can be directly fed to the adder input without saturation. A pipeline stage after multipliers is needed to ensure timing constraints are met.
Figure 4.1 displays the basic adder and multiplier, and how they are connected to form a MAC unit.

ib	Number of integer bits
fb	Number of fractional bits
Nb	Total number of bits: $Nb = ib + fb + 1$
$Q(ib).(fb)$	Q notation for fixed point
a_i	integer bits with $i \in [0,\ ib]$
bj	fractional bits with $j \in [0,\ fb]$
I	Input range
Ni	Number of intervals in Input range
m	Slope of straight line
q	Offset of straight line
LUT	Uniform Look-Up Table
RALUT	Range-Addressable Look-Up Table
PWL	Piece-wise Linear Interpolation
UPWL	Uniform Piece-wise Linear Interpolation
NUPWL	Non-Uniform Piece-wise Linear Interpolation

Table 4.1: Summary of variables and acronyms for DPU implementation

4.3 Max/Min, Shift Units

Bit Shifts are simply done by barrel shifters, which get two inputs: data and number of bits to shift. Since the DPU is always working on signed numbers, the unit performs arithmetic shifts. Left and right shifts are given separate opcodes, making them two distinct operations.
Maximum and minimum detection are performed by Max/Min Units. They can simply compute the result between two inputs or between one input and the accumulator register, allowing to map the max/min vector operations that are needed in SOM (Algorithm 1) and in Softmax (Section 4.7).

4.4 Rectifiers: ReLU, Parametric ReLU, ELU

The definitions of rectifiers from Table 2.1 are recalled in Equation 4.1.

$$\begin{aligned} &ReLU = max(0,x) \\ &Parametric\ ReLU = max(ax,x) \qquad where\ 0 < a < 1 \\ &ELU = \begin{cases} x, & x > 0, \\ a(e^x - 1), & x < 0 \end{cases} \end{aligned} \tag{4.1}$$

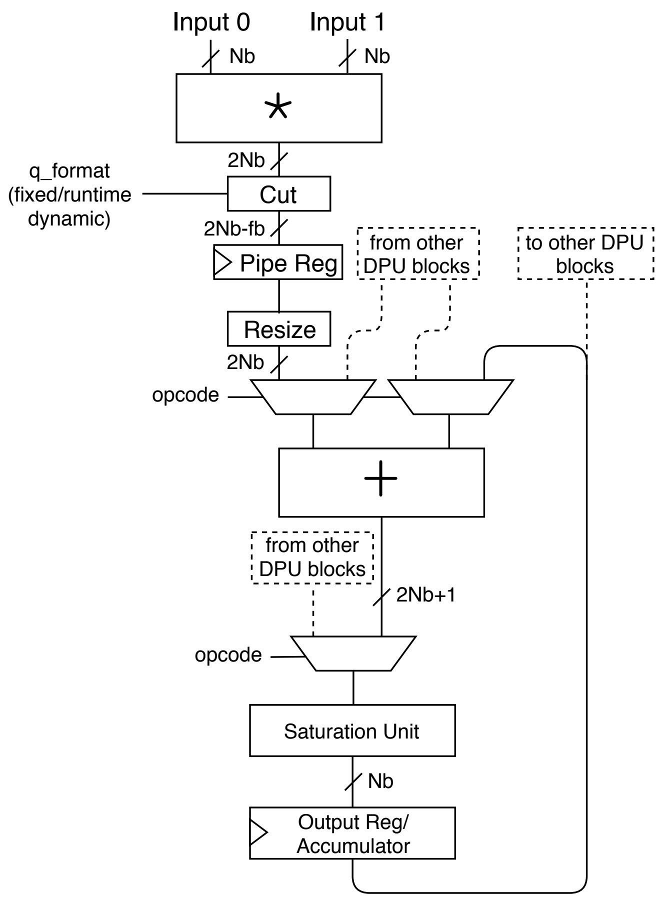


Figure 4.1: MAC Unit connected to the first two inputs

All of them consist of a selection between x, the input data to the neuron, and some transformation $f(x)$ applied to it. In the case of ReLU and Parametric ReLU, $f(x)$ and the max operation can fit inside one DPU. On the other hand, the exponential for the ELU occupies more resources, so it requires to chain two DRRA blocks: the first one computes the exponential, the other applies the function $f(x) = max(a(y-1), x)$ (where $y = e^x$) and uses the ELU unit, i.e. a multiplexer driven by $sign(x)$, to complete the computation.

4.5 Sigmoid and Tanh

Sigmoid and Tanh functions are described by the following equations, recalled from Table 2.1:

$$\sigma(x) = \frac{1}{1+e^{-x}} \qquad Tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{4.2}$$

The research in [7] points out that the two functions are closely linked with Equation 4.3.

$$\begin{aligned} Tanh(x) &= \frac{e^x - e^{-x} + e^{-x} - e^{-x}}{e^x + e^{-x}} = 1 - 2\frac{e^{-x}}{e^x + e^{-x}} = \\ &= 1 - 2\frac{e^{-x}}{e^x + e^{-x}}\frac{e^x}{e^x} = 1 - 2\frac{1}{1+e^{2x}} = \\ &= 1 - 2\sigma(-2x) \end{aligned} \tag{4.3}$$

However a simpler expression for Tanh has been found as in Equation 4.4.

$$Tanh(x) = \frac{e^x}{e^x + e^{-x}} - \frac{e^{-x}}{e^x + e^{-x}} = \frac{1}{1+e^{-2x}} - \frac{1}{1+e^{2x}} = \sigma(2x) - \sigma(-2x) \tag{4.4}$$

By exploiting the symmetry of sigmoid, the equality can be further simplified as done in Equation 4.5.

$$\sigma(-x) = 1 - \sigma(x) \Longrightarrow Tanh(x) = 2\sigma(2x) - 1 \tag{4.5}$$

Equation 4.5 is equivalent to 4.3, but slightly more efficient to map in hardware since it contains one less minus sign, which in fixed-point format means one less 2's complement. Obviously, the equation could be turned the other way around, expressing $\sigma(x) = f(Tanh(x))$, but in the end the sigmoid has been used as a base because it has a smaller image of [0,1] (compared to the [−1,1] of Tanh), and also smaller slopes ($\frac{d\sigma}{dx} = \frac{1}{2}\frac{d(Tanh)}{dx}$ for all x). It follows that if two LUTs would implement Sigmoid and Tanh with the same accuracy, the Sigmoid one would require fewer entries. Further advantages of using Sigmoid as a base are some optimizations to its hardware mapping (Equations 4.11, 4.13, 4.15 in Section 4.5.1).
Equation 4.5 shows that the Hyperbolic Tangent is nothing but a stretched and translated version of the sigmoid. Conveniently, both scaling factors are equal to 2, which in fixed-point corresponds to a simple 1-bit left shift. In practice, the operation $2x$ also needs saturation to keep the input from overflowing, but overall Equation 4.5 allows to derive the *Tanh* directly from the sigmoid with minimal overhead.

4.5.1 Adopted Sigmoid Implementation

Among all options presented in Section 3.2, the suitable ones have been narrowed down to the LUT-based approaches: LUT, RALUT, UPWL and NUPWL. The main drawback of PWL approaches is that they require a multiplier-adder chain, but these components are already included in the DPU for the MAC operation. Adopting PWL would require two LUTs for slope and offset instead of just one, an extra cost in area that would be far outweighed by the increase in accuracy. By comparing the four different LUT-based approaches, Section 8.1 provides evidence for this statement.

It must be pointed out that not all fixed point formats are suitable for representing the sigmoid correctly: the reason is that with too few integer bits, the number representation range would not be broad enough to cover the tails of the function. The goal is to find the most precise Q format that still covers the Sigmoid tails. The focus must be only on the portion reached by positive inputs, because the Sigmoid for negative inputs is derived from it. The condition of Sigmoid saturation translates into Equation 4.6.

$$\sigma(A) = \frac{1}{1+e^{-A}} = 1 \qquad A = 2^{ib} - 2^{-fb} \tag{4.6}$$

Where A is the biggest positive number on a fixed-point format.

The equation solves for $e^{-A} = 0$, which for fixed-point numbers corresponds to the condition $e^{-A} < 2^{-fb}$: if the exponential value is lower than the smallest number available at the output, then it is approximated to 0. Recalling that $ib+fb+1 = Nb$ it is possible to find the minimum value of ib that allows to represent the Sigmoid correctly. The proof is given in Equation 4.7.

$$\begin{aligned} e^{-(2^{ib}-2^{-fb})} < 2^{-fb} \qquad &\Longrightarrow \qquad -(2^{ib} - 2^{-fb}) < (ln2)\cdot(-fb) \qquad \Longrightarrow \\ (2^{ib} - 2^{-fb}) > (ln2)\cdot fb \qquad &\Longrightarrow \qquad 2^{ib} > (ln2)\cdot\frac{(Nb-ib-1)}{(1-2^{(1-Nb)})} \end{aligned} \tag{4.7}$$

Equation 4.7 cannot be expressed in closed form, so it has to be solved case by case with Nb and ib as parameters. For $Nb = 16$, the smallest valid ib is 4 so the data format must have at least 4 integer bits. The final choice on the Q format is decided basing on the general features of the target ANN, such as the maximum value that a number can reach within the network or the desired precision. Since this work targets general DNN architectures, neuron activators have to provide flexibility for the Q format and separated LUTs have been dedicated for each suitable fixed-point format.

Once the Q format has been discussed, it is possible to describe the hardware implementation. As already explained in Section 3.2 PWL requires two LUTs, one for the slope $m(x)$ and one for the offset $q(x)$. A remarkable optimization comes from the Sigmoid symmetry (Equation 4.5), which allows to halve the LUT at the cost of some extra logic. Thanks to this, the LUT contains the slope $m(x)$ and the offset $q(x)$ only for positive x, so it must be combined with other logic to obtain the full Sigmoid and Tanh functions. Equations 4.5 and 3.5 are resumed in Equation 4.8. Their combination yields the set of Equations 4.9.

$$\sigma(-x) = 1 - \sigma(x) \quad Tanh(x) = 2\sigma(2x) - 1 \quad \sigma(x) = m(x) \cdot x + q(x) \tag{4.8}$$

$$\begin{aligned}
&\sigma(x) = \mathbf{m} \cdot x + \mathbf{q} \\
&\sigma(-x) = 1 - m \cdot x - q = \mathbf{m} \cdot (-x) + \textbf{(1-q)} \\
&Tanh(x) = 2(m \cdot 2x + q) - 1 = \mathbf{4m} \cdot x + \textbf{(2q-1)} \\
&Tanh(-x) = 2(1 - \sigma(2x)) - 1 = -2(m \cdot 2x + q) + 1 = \mathbf{4m} \cdot (-x) + \textbf{(1-2q)}
\end{aligned} \tag{4.9}$$

Formulas in 4.9 mean that slope and offset for $\sigma(-x)$ and for Tanh can be directly derived from the original LUT. Table 4.2 resumes all coefficients.

	Slope	**offset**
$\sigma(x)$	m	q
$\sigma(-x)$	m	$1-q$
$Tanh(x)$	$4m$	$2q-1$
$Tanh(-x)$	$4m$	$1-2q$

Table 4.2: Summary of slopes and offset for Squash Unit

A straightforward mapping of these coefficients would require an adder and some multiplexers to select its inputs, a potentially relevant cost. Instead, the use of sigmoid as a base for deriving the Tanh allows for significant optimizations. If the linear interpolation is properly performed, the slope m will never exceed the maximum derivative which is 0.25, and similarly the offset will never be greater than 1. Under these assumptions, the hardware implementations for slopes and offsets can be greatly simplified. The final results are the optimised mappings summarised in Equations 4.11, 4.13, 4.15. It is worth noting that these Equations are valid for all data widths and fixed-point formats that allow a correct representation of Sigmoid, so they can be used for any PWL implementation.
In case of $\sigma(-x)$, m does not have to change, while for q there are two cases.
In the first one, $\frac{1}{2} \le q < 1$ so it holds:

$$\begin{gathered}
1 - q = 2^0 - \sum_{j=-fb}^{-1} c_j \cdot 2^j = 2^{-fb} \cdot \left(2^{fb} - \sum_{j=0}^{fb-1} c_j \cdot 2^j \right) \\
= (scale\ factor) \cdot (2's\ complement\ of\ fractional\ bits)
\end{gathered} \tag{4.10}$$

In the other case $q = 1$, the result of subtraction is 0. All c_j are 0, so their 2's complement yields 0 and Equation 4.10 is still valid. The integer part of the result is 0 in all cases. Overall, the optimized implementation for $(1-q)$ is performed by setting the whole integer part to 0 and by performing the 2's complement of the fractional part only. The resulting Equation is 4.11.

$$(1-q) = 00\cdots 0.\,(2's(c_{-1}\;c_{-2}\cdots\;c_{-fb})) \tag{4.11}$$

In case of $Tanh(x)$, $(2q) \in$ [1,2]. Equation 4.12 shows the subtraction computations for the two cases $2q < 2$ and $2q = 2$.

$$\begin{array}{rl} If & (2q) < 2: \\ 2q & 000\cdots 01.**\cdots * \;- \\ 1 & 000\cdots 01.000\cdots 0 \;= \\ & 000\cdots 00.**\cdots * \end{array} \qquad \begin{array}{rl} If & (2q) = 2: \\ 2q & 000\cdots 10.000\cdots 0 \;- \\ 1 & 000\cdots 01.000\cdots 0 \;= \\ & 000\cdots 01.000\cdots 0 \end{array} \tag{4.12}$$

So the result of $2q-1$ follows these rules:

- The fractional bits of the result are same as the ones from $-2q$
- The result bits a_i with $i > 0$ are always going to be 0
- The result bit $a_0 = \overline{c_0}$ where c_i are the bits of $2q$

Overall the operation only requires one inverter gate as shown in Equation 4.13.

$$(2q-1) = 000\cdots 0\,\overline{c_0}.**\cdots * \tag{4.13}$$

In case of Tanh(-x), $(-2q) \in [-2,-1]$. Also here two cases $2q < -1$ and $2q = -1$ can be isolated and Equation 4.14 shows the subtraction computations.

$$\begin{array}{rl} If & -2 \le 2q < -1: \\ 1 & 000\cdots 01.000\cdots 0 \;+ \\ -2q & 111\cdots 10.**\cdots * = \\ & 111\cdots 11.**\cdots * \end{array} \qquad \begin{array}{rl} If & 2q = -1: \\ 1 & 000\cdots 01.000\cdots 0 \;+ \\ -2q & 111\cdots 11.000\cdots 0 = \\ & 000\cdots 00.000\cdots 0 \end{array} \tag{4.14}$$

These rules apply to compute the result of $1-2q$:

- The fractional bits of the result are same as the ones from $2q$
- All integer bits of result are equal to $\overline{c_0}$, the least significant integer bit of $-2q$

Overall the computation of $1-2q$ is reduced to Equation 4.15.

$$(1-2q) = \overline{c_0}\;\overline{c_0}\cdots\overline{c_0}\;\overline{c_0}.**\cdots * \tag{4.15}$$

So the same inverter from the $Tanh(x)$ case can be reused.
Figure 4.2 illustrates the Squash Unit design and its use with the MAC chain to implement Piece-wise Linear Interpolation.

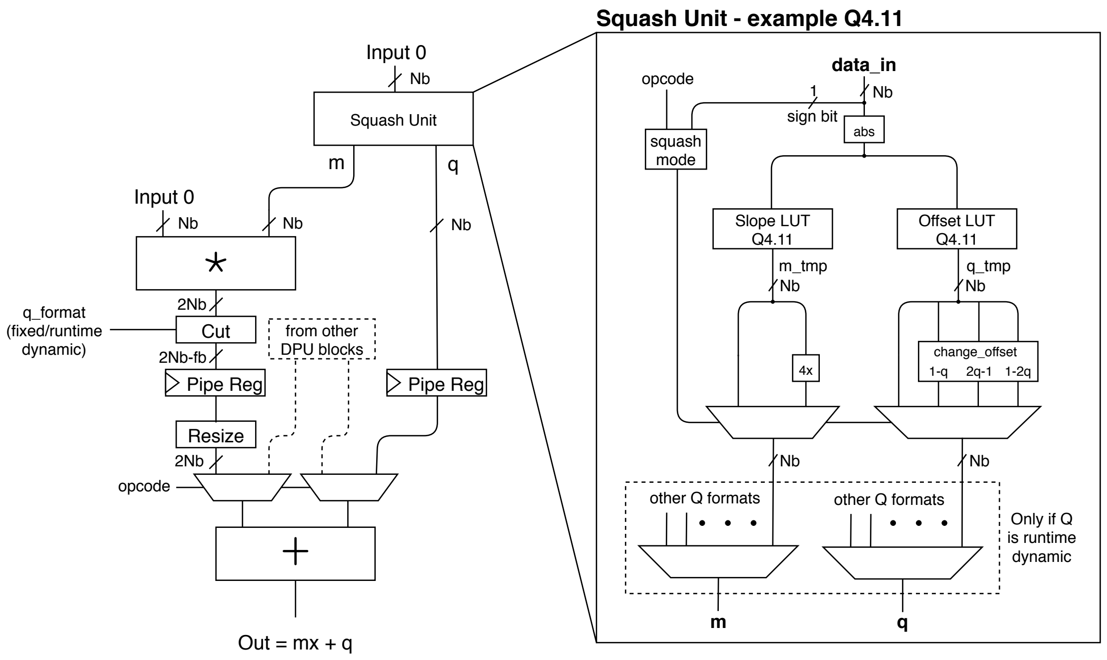


Figure 4.2: Block scheme of Squash Unit. Example for format Q4.11.

The ‘change_offset’ block inside the Squash Unit contains no adders, but only requires one 2’s complement unit, one inverter gate and one multiplexer to derive $1-q$, $1-2q$ and $2q-1$. Moreover, many inputs of the multiplexer are constant and equal, so logic optimizations can be carried out by synthesizers to further reduce the area.
The multiplier-adder chain has one pipeline stage in between, so one extra register is needed for the offset to maintain data synchronization. The operation has one clock cycle latency, but inputs and outputs of the DPU are registered so the overall latency is 3 cycles.
The last element to be analyzed is the LUT. In general, a LUT is created by partitioning the whole input range into intervals, and then by associating one output value, i.e. one LUT entry, to each interval. In standard LUTs interval widths are uniform, while in RALUTs they are non-uniform.
All LUTs used for this design have been obtained by means of Matlab scripts. The employed algorithms are generalized for Nb bits and for any Q format that is suitable for representing the Sigmoid on that bitwidth.
It is worth mentioning that only powers of two have been used for Ni, the number of input range intervals. A choice of this kind gives up flexibility for the number of entries but allows to reduce the area footprint of the LUT. In order to prove this, let us consider a complete input range on Nb unsigned bits. Naming x_0 as the first element and x_{Ni} as the last, the interval is $[x_0,\ x_{Ni}] = [0,\ 2^{Nb} - 1]$. Its width is $2^{Nb} - 1$ which is not a power of two. This makes more convenient to set $x_{Ni} = 2^{Nb-1}$

in order to obtain a width of 2^{Nb}. The consequence is that all partition intervals will be equal except the last one, which will be short of one value. This asymmetry causes no problems for the hardware implementation but as already mentioned it enables area optimizations that will be described in the following. By hypothesis, Ni can only be a power of two, so the interval width is finally $2^{Nb}/Ni$ which is still a power of two. The key point is that all interval thresholds differ by $2^{Nb}/Ni$, so to detect them is sufficient to look at a reduced number of bits of the input instead of the full Nb bits. As a result, the logic inside the decoder is simplified and its area reduced.
Algorithm 2 shows how to derive the Uniform LUT for Sigmoid with lowest error. In the case of RALUTs, the two steps of the previous algorithm cannot be separated:

Algorithm 2: LUT for Sigmoid

```
1  Input $Ni$ = power of 2: # of intervals; Nb: # of bits; fb: # of fractional bits;
2  Input $\sigma(x)$ : targetfunction; mf: range multiplication factor
3  Input $I = [I_0, I_1, \ldots, I_{Ni}]$; // Partition
4  $max\_num = 2^{Nb-fb-1}$; // Instead of $2^{Nb-fb-1} - 2^{-fb}$ to get uniform intervals
5  $step = max\_num/Ni$;
6  $min\_step = 2^{-fb}$;
7  $X = [I_0 : min\_step : I_{Ni}]$; // Input Interval derived from I
8  for $k = 0 : 1 : Ni - 1$ do
9      $inrp$ = X values between $I_k$ and $I_{k+1}$; // Positive Input Range
10     $inrn$ = X values between $-I_{k+1}$ and $-I_k$; // Negative Input Range
11     $range\_ctr = (I_k + I_{k+1})/2$; //Positive Range center
12     $qb = quantize(\sigma(range\_ctr), fb)$ //Output at range center
13     $qr = [qb - mf \cdot min\_step : min\_step : qb + mf \cdot min\_step]$ //Output Range
14     for all q in qr do
15         $err\_plus = \sum_{inrp} |qr\_plus(j) - \sigma(inrp)|$;
16         $err\_minus = \sum_{inrm} |(1 - qr\_plus(j)) - \sigma(inrm)|$; // Use of $\sigma$(x) symmetry
17         $error(q) = err\_plus + err\_minus$;
18     end
19     $best\_q = q$ such that $error(q) = min(error)$;
20     $best\_q(k) = qr\_plus(best\_j)$;
21 end
22 Output $best\_q$; // Sigmoid quantization with lowest error over partition I
```

the interval widths adapt to the function derivative, and the factor that determines the right interval size is the resulting error on the function. The specific properties of the Sigmoid allow to simplify the adaptive partition algorithm. The subportion of the Sigmoid with positive inputs is a monotonically increasing function, with a derivative that fades to zero as the function plateaus at one. These characteristics determine a partition with intervals that increase in size, moving from the origin towards the end. The whole algorithm is based on this expected behavior, so it is easier to implement but also restricted to Sigmoid or similar functions. The RALUT generation for Sigmoid is reported in Algorithm 3.
Given a partition of the input interval, slope and offset for the sigmoid PWL interpolation have been derived by means of a Matlab script. Its pseudocode is reported in Algorithm 4. The algorithms for input partitions and for PWL have been run for all suitable $fb \in [0,11]$ and for a number of partition intervals following the powers of two from 2^5 to 2^{14}. The optimal partitions for each fb, along with all other relevant figures of merit, are reported in chapter 8.1.

4.6 Exponential

Recalling from Section 3.2, it is possible to obtain Sigmoid from the Exponential, but the use of division for neuron activations would be too inefficient. Instead, that idea can be turned around by deriving the Exponential from the Sigmoid:

$$\sigma(x) = \frac{1}{1+e^{-x}} \quad \Longrightarrow \quad e^x = \frac{1}{\sigma(-x)} - 1 \tag{4.16}$$

This newly proposed solution brings several advantages. The linear interpolation LUTs for the Sigmoid are smaller than the Exponential Units in [6] and [7]. Moreover, no new components have to be added to the DPU for the Exponential, since multipliers, adders and the divider are already included. Lastly, implementing the sigmoid in a simpler way is much more scalable with the increasing depth of ANNs. In CNN and LSTM networks, neuron activations are needed for each intermediate layer, while the exponential is only used in classification, the very last processing step. Moving greater latency and power consumption to the exponential is the most sustainable solution, because it is only needed for one layer, independently on the network depth. These features comply with the policy of hardware reuse adopted for the DPU, so implementing Exponential through the Sigmoid has been deemed as the best approach. Multipliers and adders are the same ones being reused for other DPU modes, so they are already dimensioned. Choosing the correct size for the divider is mainly dependent on the Softmax Layer implementation, so it is discussed in its related Section 4.7.
Differently from Sigmoid and Tanh, Exponential always needs a Saturation Unit to limit its output values. The current DPU implementation retains the same Q format for both inputs and outputs of Exponential, and under this condition it can

Algorithm 3: Range Addressable LUT for Sigmoid

```
Input Ni =power of 2: # of intervals; Nb: # of bits; fb: # of fractional bits;
Input $\sigma(x) : targetfunction$;
Input $I = [I_0, I_1, \dots, I_{Ni}]$; //Uniform Partition
$max\_num = 2^{Nb-fb-1}$; // Instead of $2^{Nb-fb-1} - 2^{-fb}$ to get uniform intervals
$min\_step = 2^{-fb}$;
$X = [I_0 : min\_step : I_{Ni}]$; // Input Interval derived from I
$lut\_sigm = lut\_sigmoid(Ni, Nb, fb, \sigma(x), I)$; //lut_sigmoid=Algorithm 2
$prev\_err = \sum_X |lut\_sigm - \sigma(X)|$ //Initial total error
$prev\_width = max\_num$;
// For all partition points except the extremes
for $k = (Ni - 2) : -1 : 1$ do
    $lower\_err = 1$;
    while $lower\_err = 1$ do
        $I_k = I_k - min\_step$; // Decrease partition point by 1 LSB
        $width = I_k - I_{k-1}$;
        if $(width >= min\_step)$ AND $(width <= prev\_width)$ then
            //LUT algorithm on a non-uniform interval yields RALUT
            $ralut\_sigm = lut\_sigmoid(Ni, Nb, fb, \sigma(x), I)$;
            $err\_ralut\_sigm = \sum_X |ralut\_sigm - \sigma(X)|$ //New total error
            if $err\_ralut\_sigm <= prev\_err$ then
                $prev\_err = err\_ralut\_sigm$;
                *Save current partition*
            else
                *Restore previous partition*
                $lower\_err = 0$;
            end
        else
            *Restore previous partition*
        end
    end
end
$best\_ralut\_sigm = lut\_sigmoid(Ni, Nb, fb, \sigma(x), I)$; //Final Sigmoid RALUT
Output $I$; // Non-uniform partition
Output $best\_ralut\_sigm$; // Lowest error Sigmoid RALUT
```

Algorithm 4: Piece-Wise Linear Interpolation for Sigmoid

```
1  Input $Ni$ = power of 2: # of intervals; $Nb$: # of bits; $fb$: # of fractional bits;
2  Input $\sigma(x)$ : targetfunction; $\sigma'(x)$: function derivative;
3  Input $mf$: range multiplication factor;
4  Input $I = [I_0, I_1, \dots, I_{Ni}]$; // Partition
5  $max\_num = 2^{Nb-fb-1}$; // Instead of $2^{Nb-fb-1} - 2^{-fb}$ to get uniform intervals
6  $step = max\_num/Ni$;
7  $min\_step = 2^{-fb}$;
8  $X = [I_0 : min\_step : I_{Ni}]$; // Input Interval derived from I
9  for $k = 0 : 1 : Ni - 1$ do
10     $inrp$ = X values between $I_k$ and $I_{k+1}$; // Positive Input Range
11     $inrn$ = X values between $-I_{k+1}$ and $-I_k$; // Negative Input Range
12     $range\_ctr = (I_k + I_{k+1})/2$; // Range center
13     $qb = quantize((\sigma(range\_ctr), fb)$ //Offset at range center
14     $mb = quantize(\sigma'(range\_ctr), fb)$; //Slope at range center
15     $qr = [qb - mf \cdot min\_step : min\_step : qb + mf \cdot min\_step]$; //Offset range
16     $mr = [mb - mf \cdot min\_step : min\_step : mb + mf \cdot min\_step$; //Slope range
17     for all $q$ in $qr$ do
18         for all $m$ in $mr$ do
19             $sigm\_p = m \cdot (inrp - range\_ctr) + q$; //Piece-wise line at + range
20             $sigm\_n = m \cdot (inrn - range\_ctr) + (1 - q)$; //Piece-wise line at - range
21             $err\_plus = \sum_{inrp} |sigm\_p - \sigma(inrp)|$;
22             $err\_minus = \sum_{inrm} |sigm\_n - \sigma(inrm)|$;
23             $error(q, m) = err\_plus + err\_minus$;
24         end
25     end
26     $best\_q(k),\ best\_m(k) = q, m$ such that error $(q, m) = min(error)$;
27 end
28 Output $best\_q$; // Offset LUT
29 Output $best\_m$; // Slope LUT
```

be proven that saturation is necessary for every possible bit width and Q format. Assuming that x^* is the input value for which the Exponential reaches saturation and that $A = 2^{ib} - 2^{-fb}$ is the value cap, Equation 4.17 holds.

$$e^{x^*} = A \quad \Longrightarrow \quad x^* = ln(A) < A \ \ , \forall\ A \tag{4.17}$$

It is interesting to note that the data format for A has not been specified, so the result is not only valid for fixed-point formats but for floating-point as well. The consequence is that there will always be a set of input values for which the exponential has to be saturated. In the DPU Saturation Units are already allocated for each output, so they can be used by the Exponential as well.

4.7 Softmax

The Softmax is a vector-valued function: given a vector of N inputs x_i with $i = 1 \dots N$, the N outputs SM_i are computed according to Equation 4.18.

$$SM_i = \frac{e^{x_i}}{\sum_{j=1}^{N} e^{x_j}} \qquad i = 1 \dots N \tag{4.18}$$

Softmax can be implemented in a straightforward way with the same components that are already employed for the Exponential function, thus requiring no additional hardware. DRRA blocks can first be programmed to compute the Exponentials and then to accumulate the Softmax denominator. At last, they can perform one division for each input to obtain all Softmax outputs. The procedure just described is theoretically correct, but faces numerical stability problems when ported in hardware due to the Exponential saturation. A simple example is best suited to clarify: Let X be a vector of inputs, Y the 64-bit floating point output while Y_f is the 16-bit fixed-point output on $Q4.11$. All of them are reported in Table 4.3.

X	0.5	1	4	8	0.5	2
Y	5.407E-4	8.9154E-4	0.0179	0.9777	5.4075E-4	2.4234E-3
Y_f	0.0363	0.0599	<u>0.3524</u>	<u>0.3524</u>	0.0363	0.1627

Table 4.3: Example of Softmax saturation

For the chosen format, the Exponential saturation threshold is $x^* = ln(A) = 2.773$, so X contains two different inputs that cause the output to max out (underlined values in Y_f). Two major numerical issues due to saturation are evident from the example in Table 4.3. The two outputs that in theory are very different (0.0179 and 0.9777) both become equal to 0.3524. In order to score a correct classification,

Softmax is expected to yield just one output with the highest probability while all others fade to zero. This means that saturation invalidates the very classification purpose of Softmax.
The second issue is evident from Equation 4.18 and from the results in the example: each output value depends on the value of all other outputs, so saturation ends up altering the whole result vector in a significant way.
In light of these considerations, Softmax needs to be numerically stabilized. [33] suggests this can be done thanks to the property in Equation 4.19.

$$SM_i(x+c) = \frac{e^{(x_i+c)}}{\sum_{j=1}^{N} e^{(x_j+c)}} = \frac{e^c}{e^c}\frac{e^{x_i}}{\sum_{j=1}^{N} e^{x_j}} = SM_i(x) \qquad \forall c \in \mathbb{R} \tag{4.19}$$

Equation **??** states that the Softmax is invariant to translation of inputs. It is always possible to choose $c = -x_{max}$, so that the input vector becomes the one in Equation 4.20.

$$X = [x_0,\ x_1,\ \dots,\ x_N] \quad \Longrightarrow \quad [x_0 - x_{max},\ x_1 - x_{max},\ \dots,\ x_N - x_{max}] \tag{4.20}$$

All inputs are shifted to values less than or equal to 0. Exponentials will then be shifted to $0 \leq e^{x-x_{max}} \leq 1$, which corresponds to the tail of the function. In this way saturation is always prevented and the Softmax is stabilized. Another fundamental advantage of restricting the Exponentials to the tail is related to the accuracy, and is described in Section 8.2. The input translation brings the downside that Softmax outputs that were already small before translation are now underflowing to 0. This flaw is due to the unavoidable limits of fixed point format when representing small numbers. It is possible to find the input for which e^x is approximated to 0 with Equation 4.21.

$$e^x = 0 \quad \Longleftrightarrow \quad e^x < 2^{-fb} \quad \Longrightarrow \quad x < -ln2 \cdot fb \tag{4.21}$$

For the fb so that $-ln2 \cdot fb > -2^{ib}$, there exists an interval where the exponential tail is approximated to 0. After the translation $x_i - x_{max}$, a bigger portion of the inputs falls into the interval where $e^X = 0$, causing the Softmax to underflow as well. The accuracy loss is anyway expected to be negligible because it is related to the least relevant values and most importantly because it doesn't affect the very functionality of Softmax.
Once the final model for the Softmax is determined, it is possible to find the correct data width for its operands and thus dimension the required hardware components. The numerator of Softmax is a normal Exponential, so it retains the same format as the inputs. The denominator needs a more detailed analysis: given $X = [x_0,\ x_1,\ \dots,\ x_N]$ as the input vector, with N its number of elements, a worst-case bound for the denominator value is given by Equation 4.22.

$$\sum_{j=1}^{N} e^{(x_j - x_{max})} \leq \sum_{j=1}^{N} 1 = N \tag{4.22}$$

Depending on the employed data format, N could be greater than the upper bound $2^{ib} - 2^{-fb}$ and this could lead the denominator to saturate, compromising the functionality of Softmax. A solution to ensure a correct behavior could be posing a limitation on the number of Softmax inputs and thus on the number of output classes. However, especially in higher precision formats, $2^{ib} - 2^{-fb}$ would be too small of a number when compared to the thousands of classes that are supported by the state of the art ANNs. The only viable option becomes then providing all Q formats with a minimum number of integer bits by increasing the data width for the denominator. Since one 16-bit word is not enough and the DPU has two 16-bit output ports, the most efficient option is to exploit both of them. All fixed point formats are then provided with 16 extra integer bits, so that at least 2^{16} classes are always supported. It must be stressed out that the additional 16 bits only expand the integer part, while the range of possible fb remains the same as in the regular data width. The adders are already set to 32 bits inputs so they naturally support the denominator computation. As for the divider, this size can be used as a design constraint. The divider width is determined by the operand that requires the biggest number of bits, so upper bounds must be found for dividend, divisor and quotient. The divisor worst-case is already established to 32 bits from the Softmax denominator. In order to study the dividend, it is useful to think of division as the inverse of multiplication: if multiplication doubles the fractional bits of the inputs, then division halves them. In other words, if the divisor has fb bits and the quotient is required to be on fb bits as well, then the dividend fractional part must be extended on $2fb$ bits. Since the integer part does not have to change in size, the dividend must have format $Q(ib).(2fb)$. The new bit width is $Nb_{new} = ib + 2fb + 1$ and it gets the highest value when $fb = Nb - 1$. Equation 4.23 calculates the worst-case Nb_{new}.

$$\begin{aligned} Nb_{new} &= ib + 2fb + 1 = (Nb - fb - 1) + 2fb + 1 = \\ &= (Nb - (Nb - 1) - 1) + 2 * Nb - 1 = 2Nb - 1 \end{aligned} \tag{4.23}$$

Nb_{new} corresponds to 31 bits. It is interesting to note that the final result $2Nb - 1$ is completely independent on the number of fractional bits, so this dividend size is valid for all fixed-point data formats.
As for the quotient, the required number of bits can be derived by considering the biggest possible outcome of division. This is done in Equation 4.24.

$$\frac{-2^{Nb-fb-1}}{-2^{-fb}} = 2^{Nb-1} \tag{4.24}$$

The number of integer bits N_I (sign excluded) that is able to represent this value is given by Equation 4.25.

$$2^{N_I} - 1 \geq 2^{Nb-1} \quad \Longrightarrow \quad N_I = \lceil log_2(1 + 2^{Nb-1}) \rceil = Nb \tag{4.25}$$

By adding the worst-case number of fractional bits $Nb - 1$ and the sign, the final number of bits becomes $2Nb$. Ultimately, the divider can be set to 32 bits for all operands. The quotient is always contained in the least significant 16 bits in the case of Softmax, but for a normal division the 32 bit result must be sent to the Saturation Unit to saturate it back to the standard size of 16 bit.
The most straightforward Softmax implementation involves one division for each input. A more efficient approach would be obtaining the reciprocal of the denominator with just one division, and then multiplying it to all inputs. With this method, N divisions would be replaced by 1 division and N products, that are generally faster and less power-consuming. However, the great variability in number of classes and in Q formats makes this option numerically unstable. The main reason is the underflow of the division result, which is certainly prevented only if the condition in Equation 4.26 is met.

$$\frac{1}{\sum_{j=1}^{N} e^{(x_j - x_{max})}} \geq \frac{1}{N} \geq 2^{-fb} \quad \Longrightarrow \quad N \leq 2^{fb} \tag{4.26}$$

This condition cannot be ensured in general, so dividing all terms by the sum of Exponentials remains the safest approach.

4.8 DPU Data Flow Scheme

The DPU combines all previously described blocks into one single design. The general scheme is presented in Figure 4.3. Due to the number of components and the complexity of connection, a data flow scheme has been adopted in favor of a detailed description to allow for better readability. Even though multiple units can be instantiated, the Figure reports only one of each type, again for a simpler view.

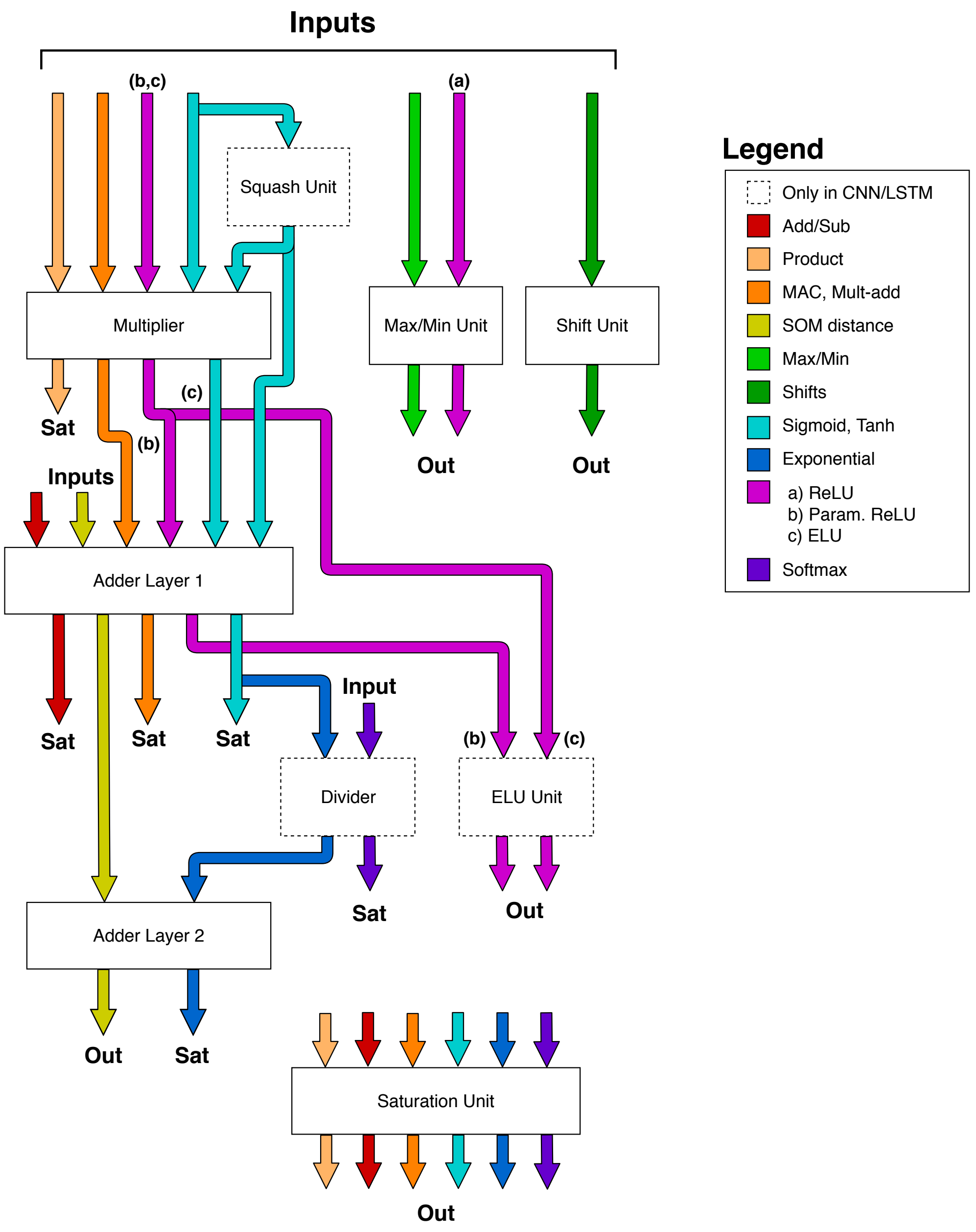


Figure 4.3: Data Flow scheme of DPU.

Chapter 5

Other Silago Cell Components

Similarly to the DPU, the Slice & Pad Unit, a special decompression AGU needed by SOM has been designed from scratch. No other blocks inside the DRRA cell required specific customizations for ANNs, but still optimisations and enhancements have been introduced to the existing Register File and AGU modules.

5.1 Register File and AGUs

The standard version of the Register File has two read, two write ports that provide word-level access and one read, one write DiMArch port that enables the transfer of an entire data block to and from the SRAM. Control and address signals for each port are handled by dedicated AGUs. Previously, the same AGUs for two data ports were shared with the two DiMArch ports, limiting data accesses through the RF during data block exchanges with SRAM. Furthermore RF total size, data width, and block size were fixed to specific values. As a first enhancement, a modified AGU has been introduced to handle the DiMArch ports control signals, so that all RF ports could be used with the maximum flexibility. The only difference of the new DiMArch AGU with respect to the original one is the reduced address space, which points to data blocks instead of just single entries. In order to prevent simultaneous writing of the same word by different inputs, a priority mechanism has been added, which privileges ports in the following order: DiMArch, port 0, port 1. Lastly, the RF has been parameterized in terms of RF total size, data width, and block size, to favor its reuse in different versions of the DRRA fabric.

5.2 Slice & Pad Unit

The SOM algorithm operates on the four nucleotides Cytosine (C), Guanine (G), Adenine (A) and Thymine (T), which on a hardware level can be encoded on two bits. The SOM computations require instead to represent each nucleotide as a couple of 16-bit coordinates (X,Y) with equal distance from the origin, as shown in Table 5.1.

	A	**T**	**C**	**G**
X	1	0	-1	0
Y	0	1	0	-1

Table 5.1

In order to achieve the best storage efficiency, the nucleotides have to be stored in the compressed form so that many of them can fit in one data word. With this approach, a decompressor is needed at each output of the RF to extract the coordinates couple from each encoded two-bit slice. This is precisely the task of the Slice & Pad Unit. The RTL scheme is depicted in Figure 5.1.

The component is enabled by the read signal from the main AGU just like the RF read port. Upon activation, the unit receives the data word from the RF and processes each two-bit slice, one after the other. For each slice (one nucleotide), the coordinates are sent in separate clock cycles with a programmable delay. In order to save one clock cycle for each nucleotide, the Unit has been implemented as Mealy machine, so it sends out the first value in the same cycle it gets activated.

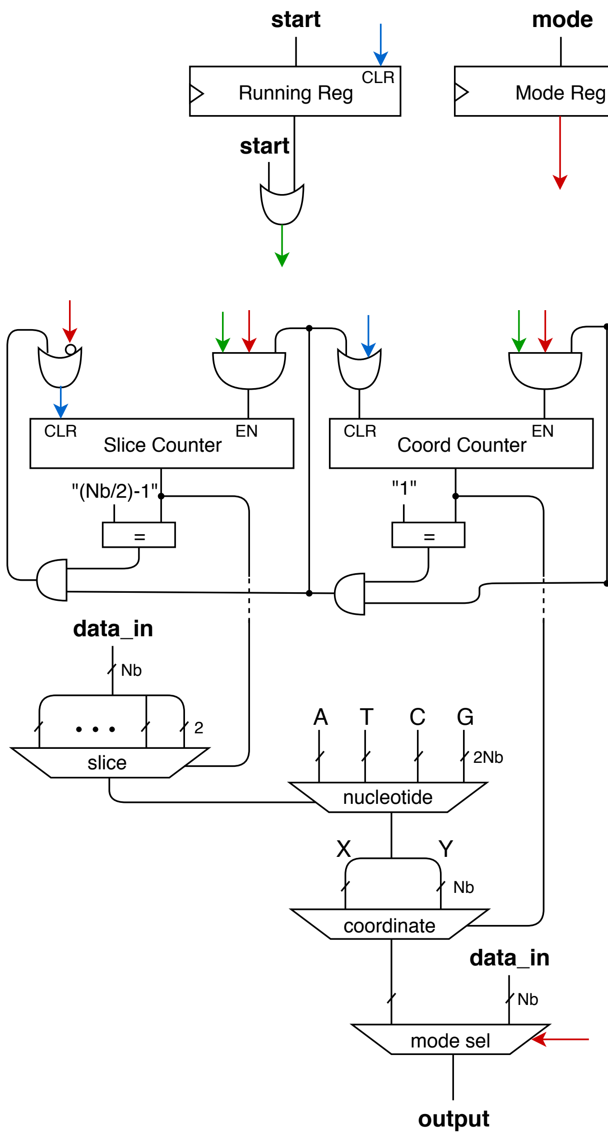


Figure 5.1: Slice & Pad Unit for SOM

Chapter 6

Compression Engine

Recalling from Section 3.3, In state of the art CNN accelerators ZRLE processing is usually placed between L2 and L1 of memory. The goal that has been set for this thesis was to carry compressed data further down the memory hierarchy, namely between L1 and L0. In this way, data travelling between SRAM and RF would always be compressed, and all ZRLE processing would take place just outside of the Register File.
For the DiMArch cells, the extra memory required by NZIV and ZI has been considered too large of an overhead. The SRAM has size $(N_{rows} \cdot N_{words} \cdot N_{bytes})\ Bytes$, so increasing by one the N_{words} would produce a relative size increase in Equation 6.1.

$$\frac{N_{rows} \cdot (N_{words} + 1) \cdot N_{bytes}}{N_{rows} \cdot N_{words} \cdot N_{bytes}} - 1 = \frac{1}{N_{words}} \tag{6.1}$$

This would correspond to the absolute size increase in Equation 6.2.

$$\frac{1}{N_{words}} N_{rows} \cdot N_{words} \cdot N_{bytes} = N_{rows} \cdot N_{bytes}\ \ [Bytes] = 2N_{rows}\ \ [Bytes] \tag{6.2}$$

The resulting overhead scaling factor has been deemed too high for the DiMArch SRAMs.
Another relevant problem for data compression is that the CGRA in its current form is designed for computation-heavy dataflows that mostly use run-time static addressing patterns. The variable size of data vectors in NZIV and ZI would then incompatible with this policy, since the SRAM content would not be fixed and predictable anymore. The RLC method embeds information inside the SRAM word itself so it would require no memory overhead, but the key problem is again that the information quantity inside each SRAM word would be run-time dynamic and thus unpredictable. The only viable solution is then splitting the full input sequence into multiple smaller ones, each one matching the SRAM word length. Compression/decompression are then carried out on each sub-sequence independently. In

order to keep the same information amount in every SRAM word, original and compressed sequences must have the same length: if the latter are shorter, then their remaining space is padded with zeros. This approach to ZRLE turns out to be quite different from the standard ones. The visual example in Figure 6.1 illustrates an SRAM word before and after compression with DRRA ZRLE.

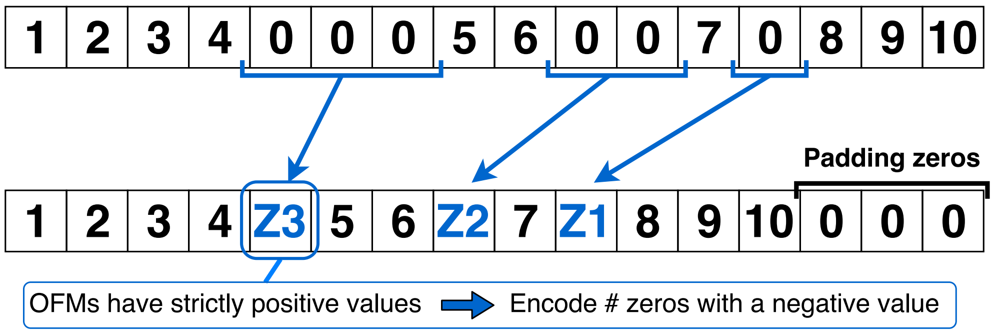

1 2 3 4 0 0 0 5 6 0 0 7 0 8 9 10
Padding zeros
1 2 3 4 Z3 5 6 Z2 7 Z1 8 9 10 0 0 0
OFMs have strictly positive values
Encode # zeros with a negative value

Figure 6.1: Working principle of ZRLE in DRRA

Any word inside the data string could be an encoded value, so *Zero Runs* cannot be identified by position, but they need a specific feature to be distinguished from normal non-zero values. OFMs only contain values greater than or equal to zero, so negative numbers can be employed to encode compression information. On the other hand, filter weights can take any value so they leave no room for encoded data. For these reasons, ZRLE has been limited to OFMs. The natural drawback of all these functionality restrictions is that the number of memory accesses is not reduced, since DRAM and SRAM store the same quantity of data as without compression. Power savings however still occur in data transfers: the last part of each SRAM word (each sub-sequence) will frequently contain many zeros, so a certain portion of the memory output will always stay constant and prevent switching activity in the interconnects.
Overall, compression efficiency of DRRA ZRLE is undoubtedly lower than the traditional approaches, but lack of memory overhead and widespread interconnect power savings can likely compensate and make it a worthy inclusion in the fabric. The validity of this statement must be verified during the characterization step. if results are not promising enough, it is best to move back to traditional implementations of ZRL between L2 and L1 memories.
As already pointed out, ZRLE processing would take place in DRRA cells, before RF block writes and after RF block reads. It follows that the compression engine must be able to process a batch of multiple words at once. In the following, Nb refers to the number of bits for a single data and Nw to the number of data containted in a SRAM word, so that the input to the engine is $Nb \cdot Nw$ bits.
Three different approaches can be taken to implement RLE.

In the Purely Sequential, input data is registered, then words are processed one at each clock cycle. Elaboration is controlled by a simple FSM so few computational elements are required, while most of the space is taken by the memory. The memory size is equal to an entire RF block. The latency is the highest among the three methods, being $1 + Nw$: one cycle to register the inputs, Nw cycles to process each word.
In the Purely Combinatorial, input data coming from the DiMArch is directly processed without being sampled. The area occupation and combinatorial delay can be potentially high due to the amount of logic required for elaborating all words at once, but the component has zero latency.
Lastly, there is the Mixed approach, which is a combination of the previous two methods. Inputs are registered, but multiple words are processed within the same clock cycle. Defining Wp as the number of words computed in parallel, the latency equals $1 + \lceil \frac{Nw}{Wp} \rceil$. The area is likely dominated by the registers as in the Sequential approach, but also includes a relevant amount of logic. The purely sequential approach has been discarded, since its latency grows too fast with Nw. As an example, the standard SRAM for DiMArch contains 16 data, which would require 17 clock cycles to be elaborated: the RLE engine would become the main system bottleneck. Moreover, extended execution times might nullify the energy savings that compression is meant to introduce.
For these reasons, the hardware implementation has focused only on the Combinatorial and Mixed approaches. All the HDL descriptions have been generalized, so as to be compatible with any combination of Nw and Nb.

6.1 ZRL Basic block: the ZRL Layer

It is worth recalling the two peculiar characteristics of DRRA ZRL: there are no fixed positions for the *Zero Runs*, but any of the Nw data inside a compressed sequence can be either a non-zero or an encoded value. Also, data sequences always match with one SRAM word. In case a compressed sequence becomes shorter, padding zeros are appended to restore its length. These two properties imply that the position of each single data word depends on the content of all previous entries inside the sequence. In case of compression, runs of multiple zeros are converted in one encoded word and all the subsequent words must be shifted backwards. For decompression, encoded words are expanded back to many consecutive zeros, so all the next words are pushed forward. From these characteristics it follows that is impossible to process data independently, but instead the input sequence must be elaborated one word after the other, starting from the first. This is the reason that brought the ZRL engine to be organized in cascaded Layers: given Nw words, the Layer L_i reads the word in position i and, depending on its content, updates the remaining part of the sequence and feeds it to L_{i+1}. The Decompression Layer block scheme is shown in Figure 6.2.

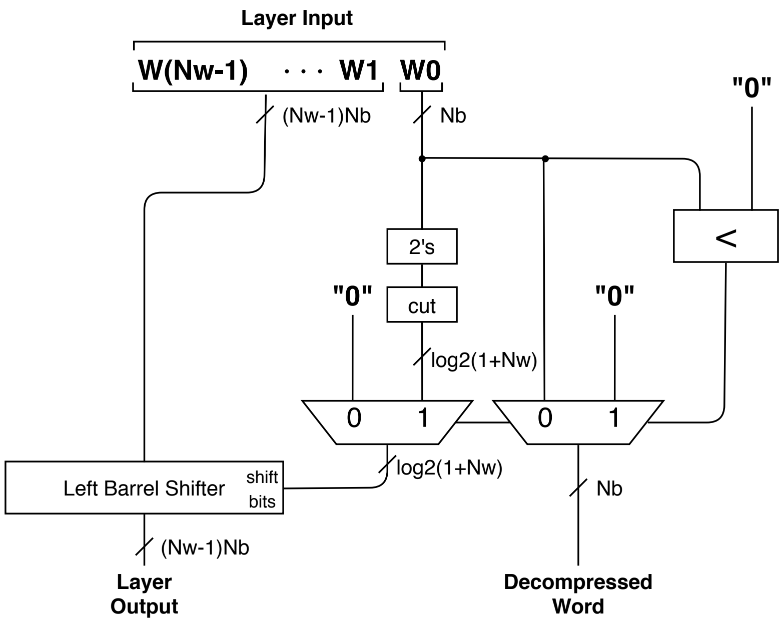


Figure 6.2: ZRLE Decompression Layer

The first word from the input is read and interpreted: if it is a negative number, it is complemented and sent to a left barrel shifter, that introduces the *Zero run* inside the vector. If the active word is instead a positive number then nothing must be changed, so the barrel shifter receives zero. In all cases, the processed word is sent to the output, while the remaining part of the sequence is transferred to the next Layer. The Compression Layer is quite similar, as can be seen in Figure 6.3. A layer of comparators identifies the zeros inside the input word, then the *Zero Run* length is determined by a Trailing Ones Unit. According to its result, a barrel shifter removes the *Zero Run* by pushing the sequence to the right and introducing padding zeros. The two Decompression and Compression Layers share most of their components, so they can actually be joined to save on area an obtain a single hardware component for the entire ZRL processing. The new general ZRL Layer is depicted in Figure 6.4, and can be used for all the three different approaches to SiLago ZRL. The control bit *mode* sets the operation: 1 for decompression, 0 for compression.

6.2 Purely Combinatorial ZRL

This implementation processes the whole input vector at once, so it chains all layers in a fully combinatorial way. Figure 6.5 shows its RTL scheme.

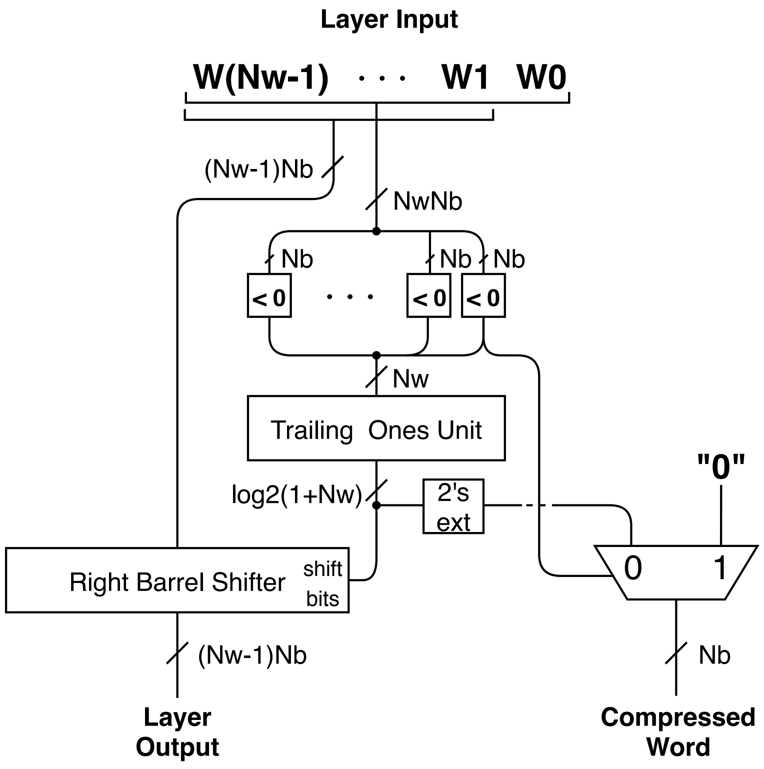


Figure 6.3: ZRLE Compression Layer

6.3 Mixed ZRL

The Mixed approach samples the input data and processes it over $1 + \lceil \frac{Nw}{Np} \rceil$ cycles, so it needs a counter to keep track of the elaboration state. The same register file is used to store the initial vector and the intermediate results, so two left barrel shifters are driven by the counter to select the locations to update. Similarly, a right barrel shifter brings the data back in position for the layer input. After $1 + \lceil \frac{Nw}{Np} \rceil$ cycles, the processed SRAM word is available at the register file output. The RTL scheme can be found in Figure 6.6.

6.4 Optimized Layer

In its straightforward implementation, a ZRL Layer processes only one word, but a closer analysis to the algorithm behavior makes it possible to increase its parallelism to two words at the same time. In the following, first and second input words are noted as L_0 and L_1 respectively. The corresponding processed words in output

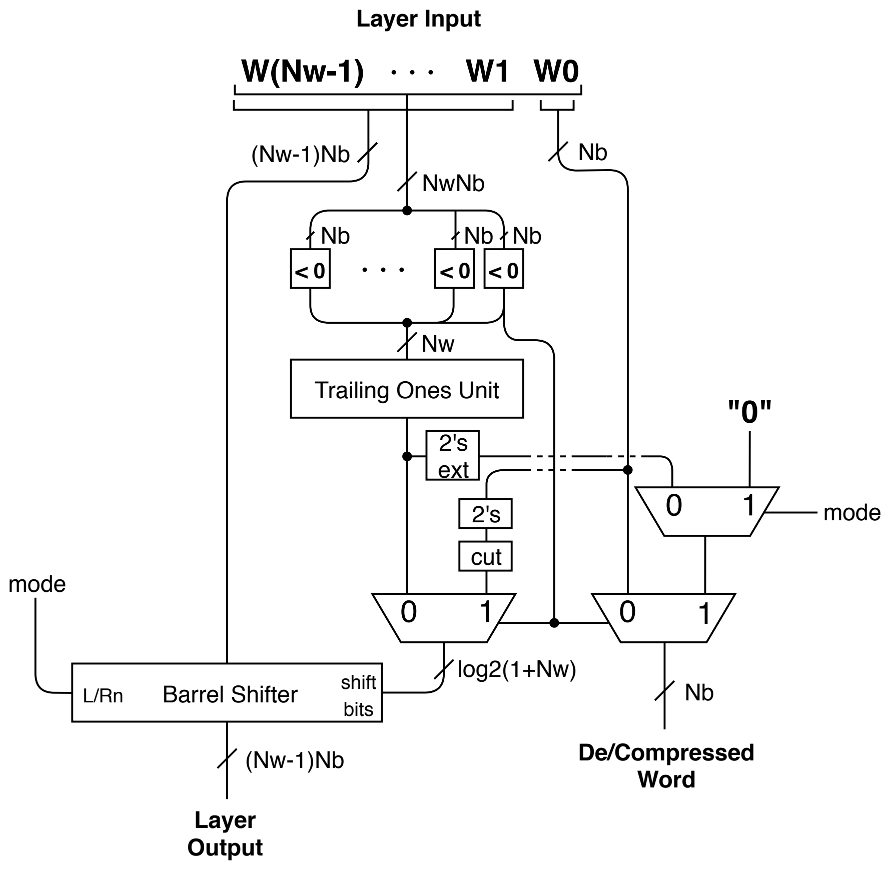


Figure 6.4: ZRLE General Layer

are out_word_0 and out_word_1. For simplicity, let us consider at first compression and decompression separately.
Decompression: If the input word L_0 is encoded, and so it is containing the length of a *Zero run*, then the following word L_1 is the first non-encoded value after the sequence so it is always non-zero. In this case, $out\_word_0 = 0$, while out_word_1 depends on the content of L_0.
If L_1 is encoded, then L_0 can only be non-zero, so $out\_word_0 = L_0$, $out\_word_1 = 0$.
Compression: If L_0 is encoded, no information on L_1 can be derived so the input sequence is processed normally. In this case $out\_word_0 = Zero\ run\ length$ while out_word_1 is the next non-zero value after the *Zero Run*. If L_0 is not encoded, it can be skipped by starting the processing on L_1 directly, so $out\_word_0 = L_0$ and out_word_1 depends on the content of the remaining inputs besides L_0.
Overall, it turns out that if one word is processed, then the result of the following one is already known. This allows to arrange the input stream in couples and so

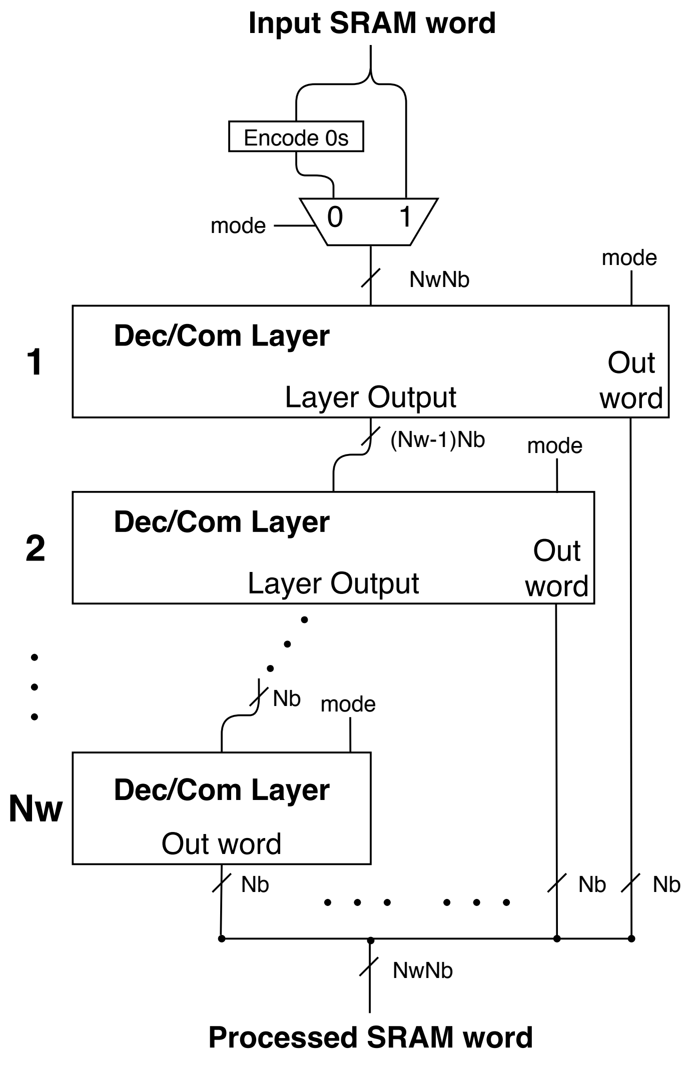


Figure 6.5: Purely Combinatorial ZRLE

to elaborate two words at the same time. Figure 6.7 shows that the new optimized Layer needs more control logic than in the previous version, but doubles the throughput. The new right barrel shifter keeps the input sequence unchanged or shifts out L_0 and moves L_1 to the rightmost position. When this happens, the left barrel shifter just before the output is used to get the words back to their original positioning. The optimized layer just presented can be plugged into the ZRLE architecture just like the old one, but in this case the overall number of Layers would be halved. The overhead to this enhanced layer may not seem negligible, but the extra logic functionally replaces an entire old layer while taking up less area. In particular, the two new barrel shifters can only move 0 or Nb bits, so in general

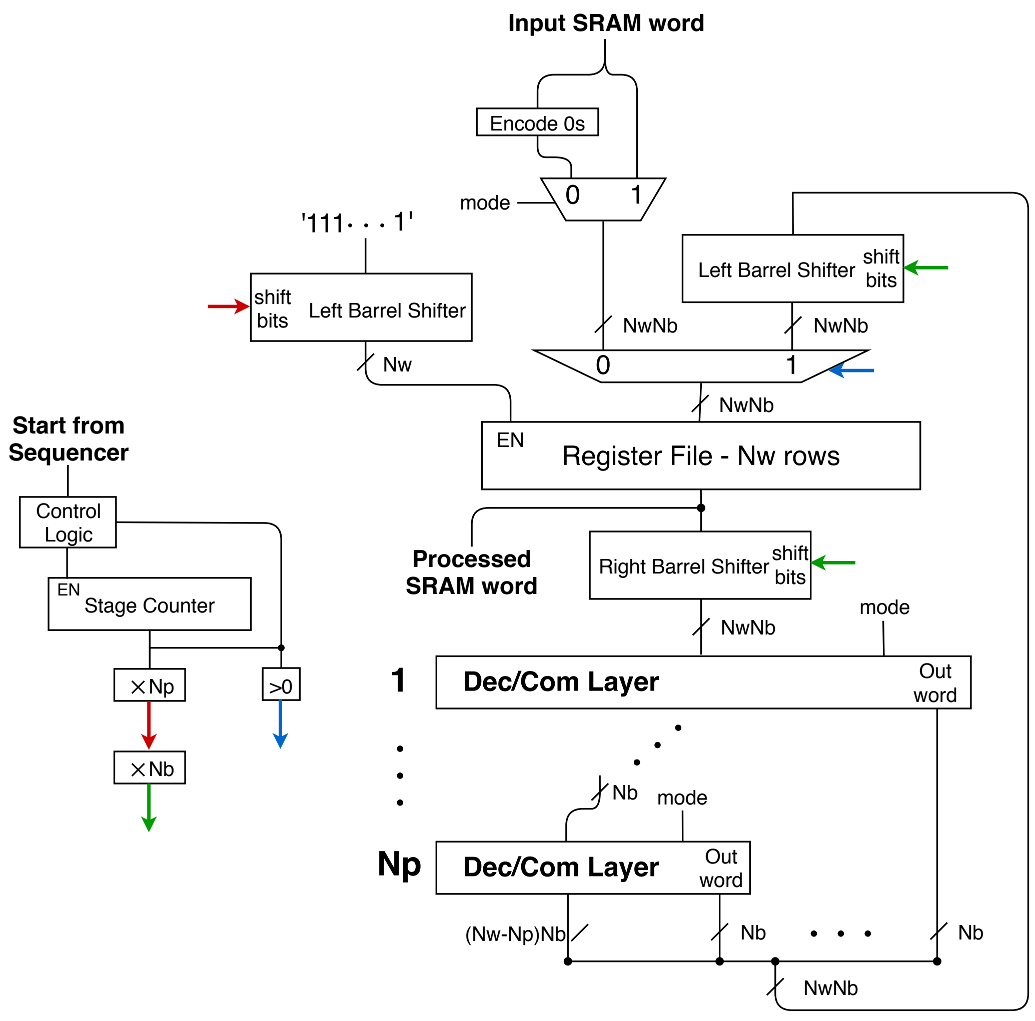


Figure 6.6: Mixed ZRLE. Long connections are color-coded.

they are much smaller than a standard barrel shifter. Overall, area and delay for the new solution are expected to improve by a sensible margin. Results presented in Section 8.4 confirm these expectations.

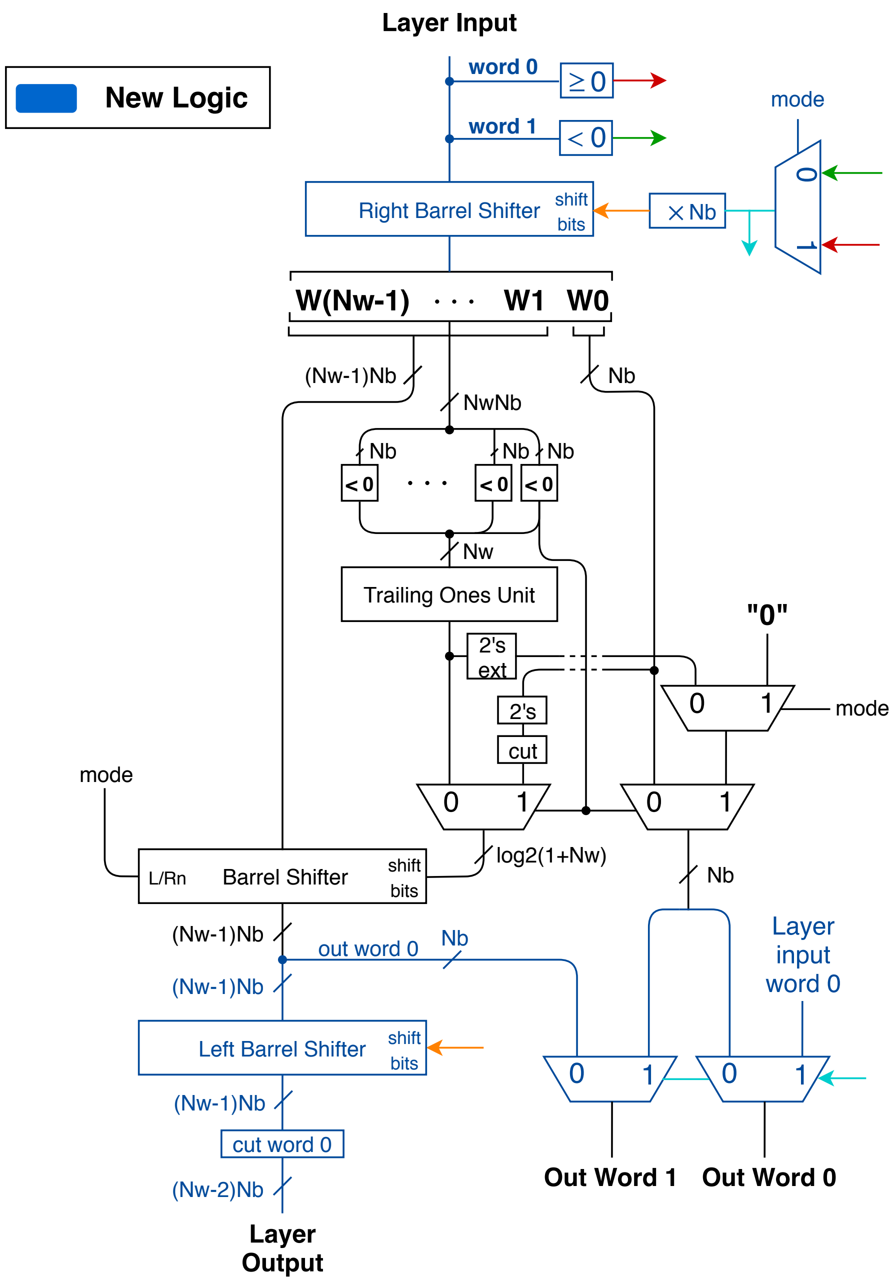


Figure 6.7: ZRLE optimized Layer. Long connections are color-coded.

Chapter 7

CGRA Synthesis

The DRRA and DiMArch arrays are tightly coupled and complement each other's functionality, so in this work they have been synthesized together as a whole bigger fabric. This makes the area analyses more complete and allows to verify the whole system functionality. It must be pointed out however that the Compression Engine has been excluded from the DRRA synthesis due to its area occupation. The area values from logic synthesis are available in Section 8.4. Even the optimized ZRLE engine would have brought an eccessive overhead to the DRRA block area, so it has been temporarily removed. Adjusting the CGRA to include compression engines has been left as future work in Chapter 9. Figure 7.1 presents the general floorplan of both SOM and CNN/LSTM fabrics. In both cases, the array is made up of 8 columns and 3 rows, one for DiMArch and two for DRRA. The top DRRA row is numbered with 0 and all other rows are given a number according to their distance from the origin. Columns instead follow a standard increasing order from left to right. In accordance with the rules of Synchoricity, the design area has been sectioned into equal space intervals. Horizontal and vertical strides have the same value and they generate a grid formed by squares. Any synchoros component must then be contained in an integer number of squares. The stride value has been adjusted to half the side of a DRRA cell, so that each one of them occupies exactly four area units. DiMArch tiles need a bigger area in order to contain the SRAM, so they have been accommodated on a greater number of squares.

7.1 Logic Synthesis

The logic synthesis of the whole fabric has been carried out on Cadence Genus. The employed technology library has been a Low Power (LP) 28nm from TSMC. Timing analyses have been carried out on the three corner cases listed in Table 7.1. The target frequency to be achieved for both SOM and CNN/LSTM has been set to 200 MHz. For a large design like the CGRA which is widely based on repetition

	Worst	**Typical**	**Best**
Voltage [V]	0.855	0.95	1.045
Temperature [°C]	125	25	-40

Table 7.1

of the same components, the bottom-up approach has been deemed as the most suitable. The logic synthesis process requires a few naming conventions, which are introduced in Figure 7.1. DPU, Register File, Sequencer and SwitchBox are in common to all DRRA tiles so they have been grouped under a dedicated block, the Tile Core. In the first synthesis step this component is synthesized only once, and for the rest of the process it is reused without further modifications. DRRA cells at the corners require different interface and control logic than the ones in the center, and the same goes for the top and bottom rows. Corner cells have different IO pins because they are linked to the sliding window interconnect only on one side, whereas top DRRA cells contain all the feedthrough wires from the DiMArch to the bottom row and also include the logic to handle communication between DRRA and DiMArch. As a result, six variations of DRRA cells are needed: top left, top, top right, bottom left, bottom, bottom right. In the second step of the synthesis, the same Tile Core was enclosed by six Wrappers which account for all DRRA variations. As for the DiMArch, all tiles are composed of SRAM blocks and NoC switches so their content doesn't change with their position, however the interface differences still require to distinguish between left, center and right wrappers. For both fabrics, SRAM blocks have been shaped as 128 rows of 16 data words of 16 bit each, for a total size of 4KB. It must be pointed out that SRAMs have not been actually synthesized because their physical macros were not available. Instead, they have been reserved a region inside the cells (the dark rectangles at the center, Figure 7.2) but they have been left as a black box and only their behavioral model has been used during verification. Column 4 in Figure 7.1 represents the generic structure of DRRA and DiMArch cells just described.
The top-level fabric has been set up so that it contains no logic of its own, but only wrappers and wires. As a consequence, the last synthesis step simply consists in instantiating the cells and connecting them.

7.2 Physical synthesis

The physical synthesis followed a standard flow, but required special measures for floorplan and pin assignment. Dimensions and position of all cells had to be exact so that they perfectly aligned to the grid. Their size was chosen to have an initial area utilization around 70%, needed to ensure enough space for buffers and internal routing. Some void space has been left between cells to accommodate the interconnections, so wire channels of equal width have been introduced across the whole fabric. The synchoros grid stride has been adjusted so that all grid lines laid exactly in the middle of the channels. The resulting floorplan is visible in Figure 7.1. Positioning of IO pins was the other critical part that enabled abutment. Pins among all cells were carefully aligned, so that the interconnections were simply formed by straight lines. At the end of physical synthesis, both SOM and CNN/LSTM arrays met the frequency of 200 MHz and were DRC-clean.

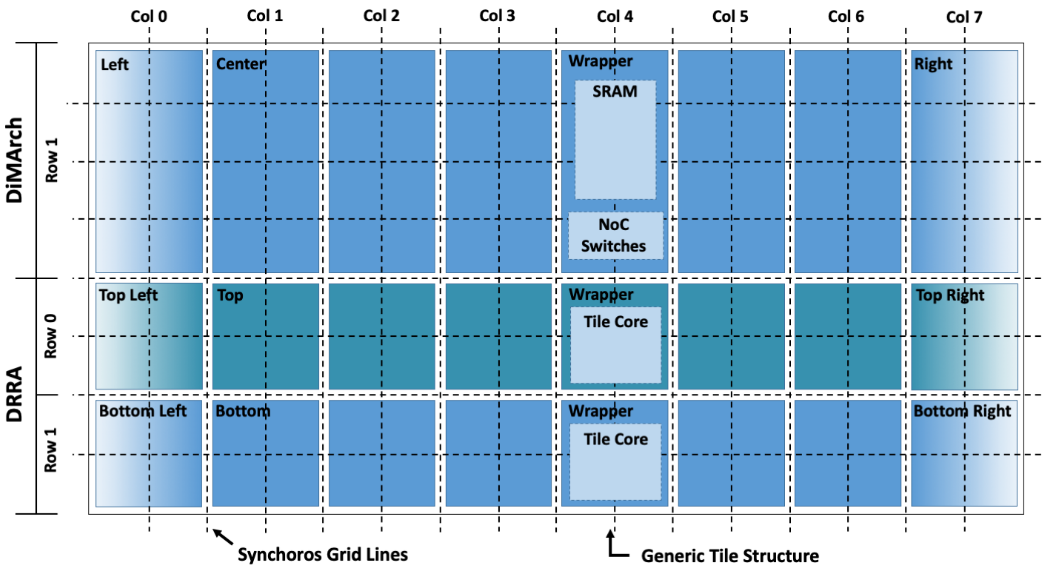


Figure 7.1: CGRA General Floorplan Scheme.

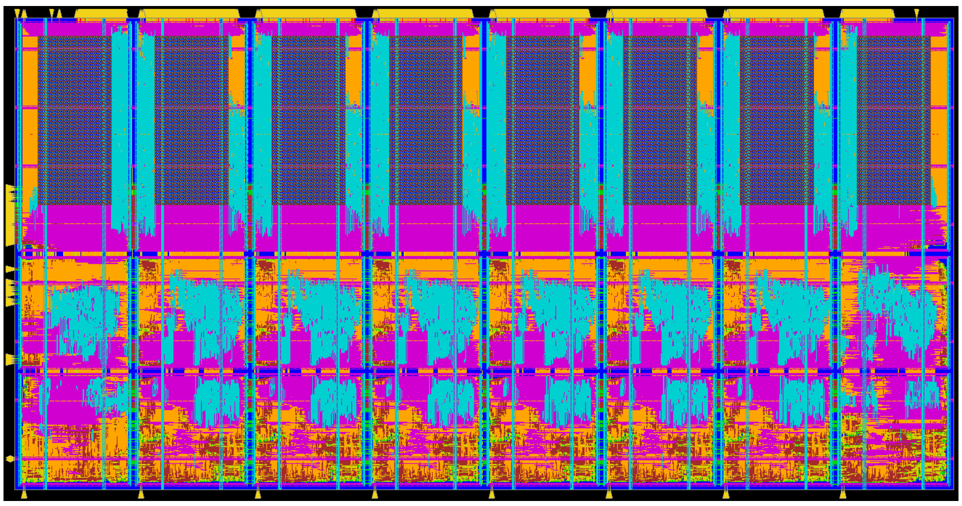


Figure 7.2: Synthesized SOM CGRA, Top-level view.

Chapter 8

Results

This Chapter is dedicated to the results analysis for all components that are part of the DRRA cell and for the CGRA synthesis. The main goal for the ANN activator functions (Sigmoid, Tanh, Exponential, Softmax) is to reach an acceptable accuracy while containing area requirements. As for the Zero Run Length Encoding, the different available implementations are compared basing on latency and area. Power estimations on the synthesized CGRAs were not performed due to lack of time, so only detailed area reports have been included.

8.1 Sigmoid and Tanh

The fundamental design parameter for a LUT is the number of entries, which is always determined in a tradeoff with the final accuracy. The optimal input partition has been found by comparing the maximum absolute error ϵ_{max} against the number of entries N_e through the following figure of merit:

$$FoM = \frac{error_{max}}{max(error_{max})} \cdot \frac{N_e}{max(N_e)} \tag{8.1}$$

Both terms are normalized so they have the same weight. The golden reference models that have been used for measuring the error of Sigmoid and Tanh use 64-bit floating point format.
An alternative to the maximum error could have been the average error, but in the case of sigmoid it turns out to be completely unreliable. This is caused by the following two factors:

1. The sigmoid tails always reach their ideal value so their error is always 0;

2. They take up a greater and greater part of the total input range as the number of fractional bits decreases.

The latter property can be formally proved by finding the value x' at which the quantized sigmoid jumps from $1-2^{-fb}$ to 1. This change happens when the sigmoid is exactly halfway between 1 and the previous quantized value $1-2^{-fb}$. The formula is given in 8.2.

$$1 - \frac{1}{1+e^{-x'}} = \frac{2^{-fb}}{2} = 2^{-(1+fb)} \tag{8.2}$$

Solving for x' one finds the result in Equation 8.3.

$$\frac{1}{1+e^{x'}} = 2^{-(1+fb)} \quad \Longrightarrow \quad x' = ln\left(2^{(1+fb)} - 1\right) \tag{8.3}$$

This means that x', the sigmoid saturation point, gets closer and closer to the origin as fb decreases. The combination of these two factors averages out the more relevant errors of the central part, especially for the low fractional bit formats that are expected to be less precise. The consequence is evident in Figure 8.1b, where the formats $Q15.0$ and $Q14.1$ are given as the most accurate, even though the only allow the values $\{0, 0.5, 1\}$ and $\{0,1\}$ respectively.
In the case of sigmoid, relative errors should also be avoided simply because their formula:

$$\epsilon_\sigma(x) = \frac{error_\sigma(x)}{\sigma(x)} \tag{8.4}$$

Would be undefined for the portion $\sigma(x) = 0$.
The Maximum absolute error should instead be taken as a safer measure of the accuracy: Figure 8.1a shows that the maximum absolute error always decreases as fb increases, in accordance with the expectations.

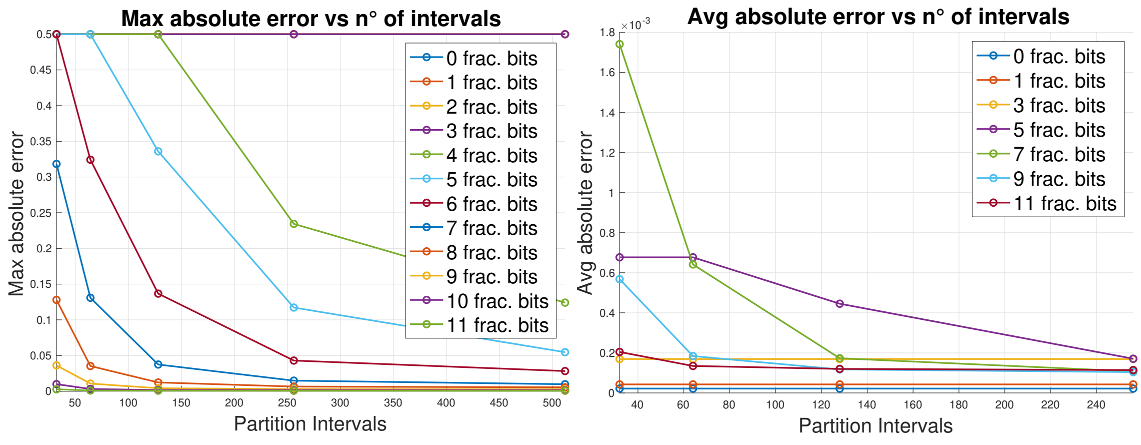
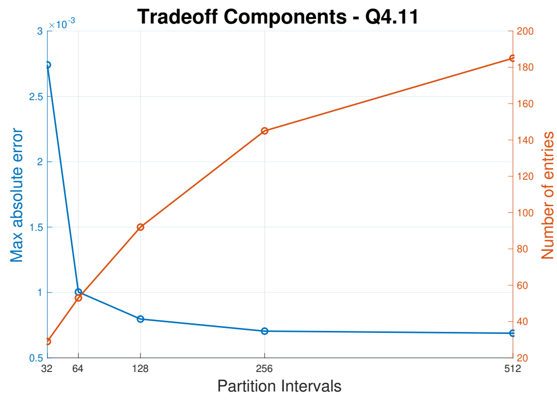


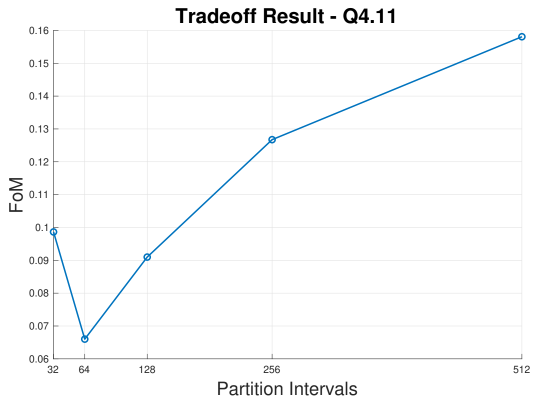
Figure 8.1: Sigmoid Maximum error (a), Sigmoid Average error (b)

An efficient LUT strikes a balance between error and size, so the optimal solution is to be found in the global minimum of the FoM function. As an example, Figure

8.2b shows the tradeoff that led to the chosen size for the uniform PWL look-up table for $Q4.11$. In Figure 8.2a it can be noticed that the number of entries is always different from the number of intervals: all LUTs have been implemented as combinatorial logic, so distinct outputs with the same value can be merged.



(a)



(b)

Figure 8.2: Tradeoff for LUT size: Optimal number of partition intervals corresponds to global minimum of FoM function

This optimum search method has been conducted on LUT, RALUT, UPWL (uniform PWL) and NUPWL (non-uniform PWL) versions of sigmoid, for all $fb \in [0,11]$ and Ni, $Ni_b \in [5,14]$. Table 8.1 reports all optimal solutions through their two main figures of merit (number of entries and maximum error) for all valid fb.

	Number of Entries				**Maximum Error**			
fr. bits	LUT	RALUT	UPWL	NUPWL	LUT	RALUT	UPWL	NUPWL
0	2	3	2	33	0.5	0.5	0.5	0.5
1	3	3	3	31	0.2689	0.2310	0.2689	0.2310
2	4	4	4	20	0.1275	0.1225	0.1275	0.1225
3	6	6	4	18	0.1645	0.0621	0.1645	0.0622
4	2	10	4	26	0.0361	0.0312	0.1240	0.0312
5	4	18	6	41	0.2188	0.0156	0.0547	0.0156
6	7	34	6	23	0.1250	0.0078	0.0429	0.0181
7	12	25	11	28	0.0625	0.0234	0.0148	0.0100
8	22	44	24	35	0.0316	0.0130	0.0065	0.0055
9	43	82	25	45	0.0159	0.0071	0.0042	0.0025
10	84	91	46	45	0.0079	0.0064	0.0017	0.0017
11	166	162	53	46	0.0040	0.0042	0.0010	0.0012

Table 8.1: Figures of merit for Sigmoid against number of fractional bits

RALUTs achieve similar performances to standard LUT. The maximum error provided by RALUTs is only slightly lower, but it is counterbalanced by a greater number of entries. Considering that RALUTs require an additional decoder layer besides the actual LUT, this overhead ends up not being justified by the greater accuracy.
Algorithms 2, 3 and 4 have been used for deriving partitions for all LUT, RALUT or PWL implmentations of sigmoid. The differences between LUT and RALUT can be seen in Figure 8.3, which contains the Sigmoid on 64-bit floating point precision along with LUT and RALUT Sigmoids, for format $Q4.11$ and a partition of 64 intervals. It is possible to notice the RALUT intervals changing width according to the Sigmoid slope and allowing greater accuracy with the same number of partition intervals. The PWL approaches easily prevail on LUT-based ones, as expected.

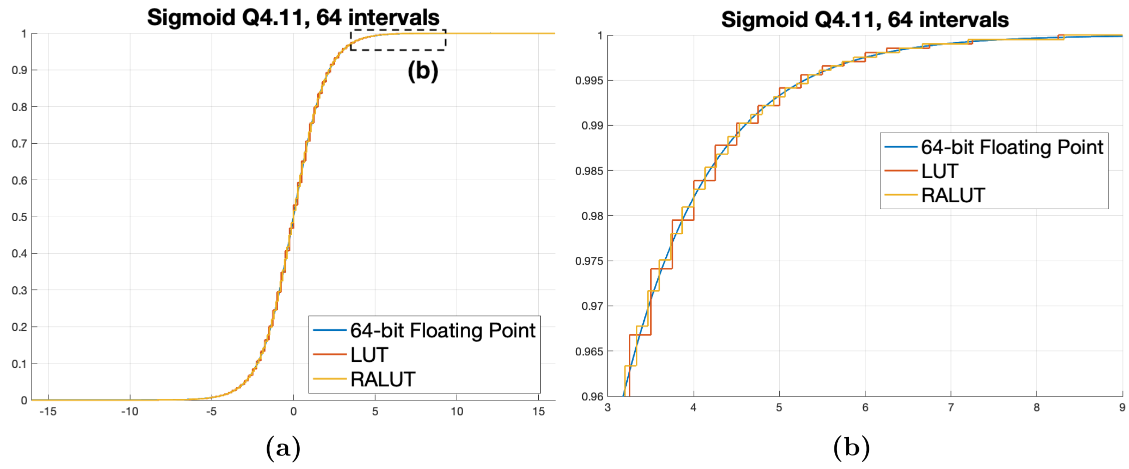


Figure 8.3: LUT and RALUT. Sigmoid full input range (a), Sigmoid Close-up on tail (b).

For low fb the two approaches yield equal results, but it is for higher fb that PWL gains a considerable edge both in acccuracy and in number of entries. For what concerns the Silago platform, multipliers and adders are already included in the DPU, so choosing PWL comes at the very low cost of adding one entry to input multiplexers of the MAC unit. A good example can be format $Q4.11$, for which both PWL implementations have a maximum error around 4 times lower than LUTs. In addition to this, standard LUT and RALUT approaches require more entries that the PWL look-up tables. These considerations lead to the adoption of PWL-based approaches for the DPU, so the final choice narrows down to UPWL or NUPWL. For low fb, accuracies are mostly the same, but NUPWL requires more entries; for greater fb, both accuracy and number of entries are quite similar. NUPWL is based on RALUTs which have an increased area cost, so the more simple UPWL has been chosen instead. In light of these results, PWL approaches in general are confirmed to be the best solution for Silago in terms of area and accuracy.

Figure 8.4 provide visual proof by comparing Sigmoid and Tanh as obtained by a normal LUT and UPWL. It is interesting to note that the tails take the exact same values: this is because the UPWL slope fades to 0, so only one degree of freedom is left just like in a LUT. For the rest of the waveform however the UPWL yields much more accurate results. The final performances reached by Sigmoid and Tanh

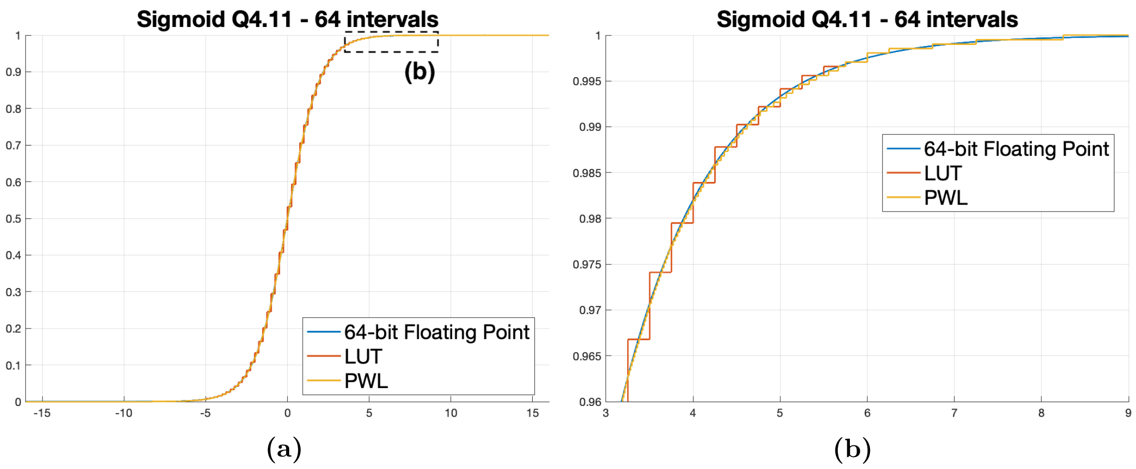


Figure 8.4: LUT and UPWL. Sigmoid full input range (a), Sigmoid Close-up on tail (b).

are resumed in Table 8.2. The maximum absolute error has been taken as the main figure of merit. The average has still been included for completeness, but with the caveat that has already been discussed previously.

The most precise sigmoid ($fb = 11$) has a maximum error of 0.001, so it can be considered a very good approximation. For lower fb (5-10), the accuracy obviously reduces but still attains reasonable values; if fb decreases too much results get quite poor, but that is due to the inbuilt lack of precision of the format, rather than to the interpolation method itself.
The Tanh is derived from the Sigmoid, and the tradeoff from the area savings is an increase of the maximum error, which is around 1.5 times bigger for greater fb, while goes up to 2 times for lower ones. Thanks to the small errors achieved by the Sigmoid, this detrimental effect is kept under control so Tanh maintains good accuracy values. Tanh is a stretched, translated and sped up version of the Sigmoid as in Equation 4.5, so the tail saturation occurs even earlier than the Sigmoid and this leads to the Tanh average error being smaller.

8.2 Exponential

Following the same approach for neuron activators, the accuracy is mainly quantified in terms of maximum absolute error. The golden reference is still the target

	Absolute errors UPWL			
	Sigmoid		Tanh	
fr. bits	**Max**	**Average**	**Max**	**Average**
0	0.5	2.179E-5	1	2.380E-5
1	0.2689	4.262E-5	0.5379	4.359E-5
2	0.1275	8.477E-5	0.2550	8.524E-5
3	0.1645	1.693E-4	0.2384	5.101E-5
4	0.1240	1.027E-4	0.1716	9.141E-5
5	0.0547	8.579E-5	0.0643	6.538E-5
6	0.0429	1.266E-4	0.0625	1.121E-5
7	0.0148	1.074E-4	0.0229	8.404E-5
8	0.0065	1.020E-4	0.0097	6.718E-5
9	0.0041	1.174E-4	0.0066	8.535E-5
10	0.0016	1.164E-4	0.0023	8.056E-5
11	0.0010	1.346E-4	0.0015	9.711E-5

Table 8.2: Figures of merit for Sigmoid and Tanh against number of fractional bits

function on 64-bit floating point, but modified to include saturation on a 16-bit fixed point format. The reason is the following: the exponential reaches very high values for relatively small positive inputs, so for all targeted Q formats the output saturates long before the input range ends. Within the saturation region, the difference between fixed point and floating point becomes so dramatic that it heavily skews the error computation, thus making it useless. The solution can only be saturating the reference exponential in the same way as the fixed-point version, so that the error can be evaluated only on the meaningful part. Although the issue with the maximum error is now solved, for the whole saturation interval the error is 0, so the average error gets skewed and becomes affected by the same reliability issues as in the Sigmoid. Table 4.2 lists exponential errors as function of fb.

Exponential Absolute errors						
fb	**0**	**1**	**2**	**3**	**4**	**5**
Maximum	32764	16381	8184	4090	2026	990
Average	4.468	3.982	3.067	2.878	1.595	1.227
fb	**6**	**7**	**8**	**9**	**10**	**11**
Maximum	473	129	45.35	8.29	0.859	0.198
Average	0.840	0.274	0.0709	0.0169	0.0044	0.0014

Table 8.3: Figures of merit for Exponential against number of fractional bits

The problem that Table 4.2 brings out is the striking precision loss compared to the Sigmoid from which the Exponential has been derived. As an example, it is enough to look at the maximum error for format $Q4.11$, which is the most precise: for Sigmoid it is 0.0010 while Exponential reaches 0.198, 198 times bigger. This dramatic loss in performance has been investigated with the first order Taylor expansion model for error propagation in Equation 8.5.

$$\begin{gathered} exp(\sigma) = \frac{1}{\sigma(-x)} - 1 = \frac{1}{1-\sigma(x)} - 1 \\ \implies \delta exp(\sigma) = \left|\frac{\partial exp}{\partial \sigma}\right| \delta\sigma \quad \implies \quad \delta exp(\sigma) = \frac{1}{(1-\sigma)^2}\,\delta\sigma \end{gathered} \tag{8.5}$$

According to Equation 8.5, the error tends to infinity as the Sigmoid saturates towards 1, so the model might be the correct explanation for the accuracy degradation. In such case, the limit would be inbuilt in the use of division and its way of propagating the error, so it could only be eliminated by changing the exponential implementation altogether. The propagation model can be further studied by introducing a dependency on the fixed-point format. Given Nb bits and a Q format $Q(ib).(fb)$, the biggest number that can be represented is the one in Equation 8.6.

$$2^{Nb-fb-1} - 2^{-fp} = A \tag{8.6}$$

When the exponential saturates, both the reference model and real implementation reach the same value, so the error drops to zero. The worst case for error propagation occurs just before saturation and it is given in Equation 8.7.

$$\frac{1}{1-\sigma^*} - 1 = A \quad \implies \quad \sigma^* = \frac{A}{A+1} \tag{8.7}$$

The value σ^* can be plugged into the error model, which is done in Equation **??**.

$$\begin{aligned} &\implies \delta exp_{max} = \frac{1}{1-\left(\frac{A}{A+1}\right)^2}\,\delta\sigma \quad \implies \quad \delta exp_{max} = (A+1)^2\,\delta\sigma \\ &\implies \delta exp_{max} = \left(2^{Nb-fb-1} + 1 - 2^{-fp}\right)^2 \delta\sigma \\ &\implies \delta exp_{max} \sim 2^{2(Nb-fb-1)}\delta\sigma = 2^{2ib}\delta\sigma \end{aligned} \tag{8.8}$$

For $Nb = 16$, the term $1-2^{-fp}$ can be neglected to highlight the main contribution: approaching saturation, the error propagation coefficient depends exponentially on the number of integer bits ib (excluding the sign bit).

The model resumed in Equation 8.8 can be tested to confirm its validity. For format $Q4.11$:

$$\sigma^* = \frac{A}{A+1} \simeq \frac{15.995}{16.995} \simeq 0.941 \tag{8.9}$$

Knowing the value of σ^* and so of $1-\sigma^*$, it is possible to obtain the corresponding $\delta\sigma^*$ and multiply it by the propagation coefficient to derive a theoretical estimate of the worst-case error for the Exponential, reported in Equation 8.10.

$$\delta exp_{max} = \left(2^{Nb-fb-1} + 1 - 2^{-fp}\right)^2 \delta\sigma^* = 288.98 \cdot 6.75 \cdot 10^{-4} = 0.195 \tag{8.10}$$

The result is reasonably close to the maximum error of 0.198 reported in Table 8.3.
The analysis ultimately shows that the worse performances of the Exponential are due to the use of division itself, thus improvements are only possible by changing implementation method: the decrease in accuracy is the necessary tradeoff to pay by reusing the divider to save area and power. The error values can be somewhat acceptable for format $Q4.11$, but they would get unreasonably large with more integer bits.
Such kind of inbuilt limitation would be an issue for a general implementation of the exponential, but the peculiar use of the function inside ANNs allows to bypass the problem altogether. Exponential is in fact only used in two operations:

- ELU (Exponential Linear Unit). Its definition is recalled in Equation 8.11.

$$ELU = \begin{cases} x, & x > 0, \\ a(e^x - 1), & x < 0 \end{cases} \tag{8.11}$$

  It can be seen that only the Exponential tail is used, which behaves well in terms of error. In fact, the largest propagation coefficient is 4 as in Equation 8.12.

$$\left|\frac{\partial exp}{\partial \sigma}\right|_{max} = \frac{1}{(1-\sigma(0))^2} = \frac{1}{\left(1-\frac{1}{2}\right)^2} = 4 \tag{8.12}$$

  That is an acceptable value and most importantly it is independent on the data format.

- Softmax. As explained in Section 4.7, the input vector is first subtracted by its maximum value, so all exponentials have an argument that is lower than or equal to zero. Also in this case, the effectively used portion is just the tail.

Ultimately, it turns out that ANNs make use of the exponential section that poses no problem in terms of error, so the inbuilt accuracy losses do not need to be solved as they are always avoided. It is worth reporting the error figures of merit for the exponential tail only, which are given in Table 8.4. All new errors are now in line with the performance provided by the Sigmoid, so the chosen Exponential implementation has proven to be valid.

Exponential Tail Absolute errors						
fb	**0**	**1**	**2**	**3**	**4**	**5**
Maximum	1	1	1	1	0.25	0.125
Average	4.828E-5	7.756E-5	1.379E-4	2.597E-4	1.664E-4	1.127E-4
fb	**6**	**7**	**8**	**9**	**10**	**11**
Maximum	0.125	0.03125	0.01165	0.0079	0.0033	0.0015
Average	1.717E-4	1.325E-4	1.090E-4	1.364E-4	1.231E-4	1.423E-4

Table 8.4: Figures of merit for Exponential tail against number of fractional bits

8.3 Softmax

Recalling Equation 4.18, it can be seen that each output of Softmax depends on all inputs but also on their number N. Under these conditions, performing a complete error evaluation with all possible combinations of input values and with varying N becomes very unpractical. A more feasible alternative is finding a proper distribution of inputs that is close to realistic use cases. The Softmax in ANNs serves the purpose of taking the network outputs and normalizing them into a probability distribution. The non-linear and strictly increasing curve of Exponential emphasizes the difference between high and low values, but the overall behavior of the input distribution is left unchanged. Figure 8.5 displays an example.

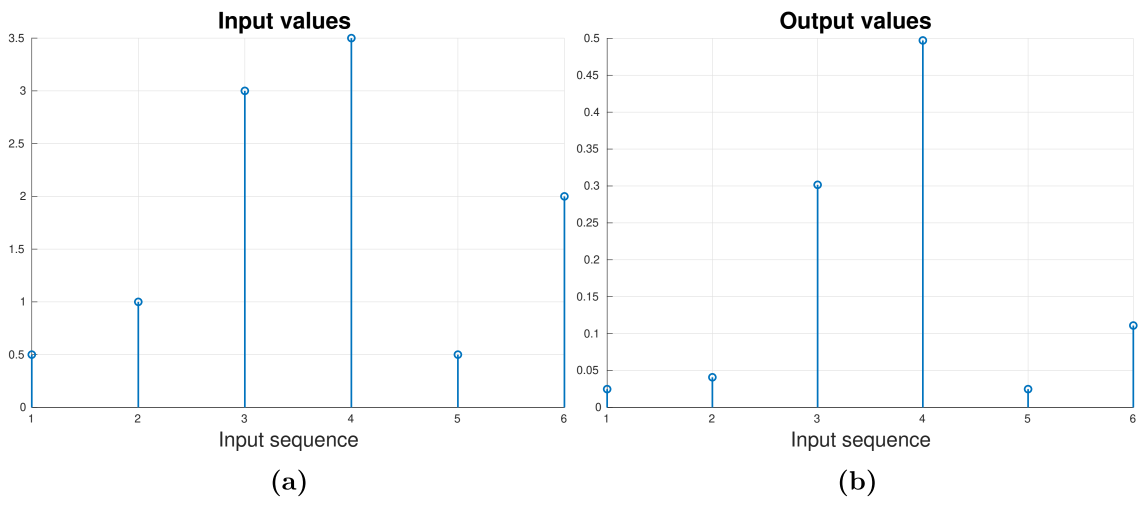


Figure 8.5: Example of Softmax Normalization

The relative weights among inputs have changed, with the higher values being accentuated and the lower ones flattened out, but the overall distribution 'shape' remains the same. The expected behavior for CNN and LSTM is having only one output close to 1 and all others to 0 (a correct data classification), so it is reasonable to assume that this particular behavior can be found at the input of the Softmax Layer as well. The final choice for representing the input vectors fell on the altered Normal distribution in 8.13.

$$X_i(\mu, \sigma^2) = -2^{ib}\left(e^{-\frac{(i-\mu)^2}{\sigma^2}} - 1\right) \tag{8.13}$$

Inside Equation 8.13, i is the position inside the input sequence $1,2,\ldots,N$, X_i is the $i-th$ input, μ is the mean and σ is the standard deviation. This Normal distribution has been modified so that its values now span the interval $[-2^{ib},0]$, modeling the $(x_i - x_{max})$ inputs to Softmax. For each suitable Q format, input vectors have been created with N varying in the interval $[2^4,\ 2^{16}]$. The mean μ for each vector has been chosen at random, while the standard deviation has always been kept fixed to 5, so as to include some intermediate values between the maximum and the function tails. The chosen figures of merit are the maximum and average error over the output vector. An additional check can be done on the sum of all outputs, which should be equal to 1 since the Softmax creates a probability distribution. The golden reference model employs 64-bit floating point data for both Exponentials and division, while the hardware implementation model is limited to fixed-point with fb fractional bits both for exponential and for division. In order to ensure a fair comparison, the same input vectors with discretized values on fb bits have been used for both versions.
All figures of merit have proven to be almost independent on N, so only the average values with respect to N have been reported. Tables 8.5 and 8.6 lists the results as a function of Q.

Softmax Absolute errors						
fb	**0**	**1**	**2**	**3**	**4**	**5**
Maximum	3.917E-64	1.979E-32	4.4403E-16	2.372E-8	2.177E-4	0.0204
Average	0.0292	0.0074	0.0028	0.0011	9.032E-4	3.989E-4
fb	**6**	**7**	**8**	**9**	**10**	**11**
Maximum	0.125	0.03125	0.01165	0.0079	0.0033	0.0015
Average	1.717E-4	1.325E-4	1.090E-4	1.364E-4	1.231E-4	1.423E-4

Table 8.5

Unintuitively, the lower fb formats appear to have the highest precision. This result is easily explained by the shape of the exponential, that gets closer and closer to the discontinuous function:

$$f_{exp}(x) = \begin{cases} 1, & x = 0 \\ 0, & otherwise \end{cases} \tag{8.14}$$

fb	**0**	**1**	**2**	**3**	**4**	**5**
Softmax Sum	1	1	1	1	1	1
fb	**6**	**7**	**8**	**9**	**10**	**11**
Softmax Sum	0.9844	0.9844	0.9883	0.9941	0.9912	0.9951

Table 8.6

it follows that Softmax for both floating-point and fixed-point tends to assign a very high score (ideally 1) to one single output and zero to all others. Instead, on higher precision formats many more values are available both for inputs and outputs, so more differences between the two Softmax models arise.
Overall, the results point out that the chosen Softmax is very precise and stable with respect to both Q format and vector size N.

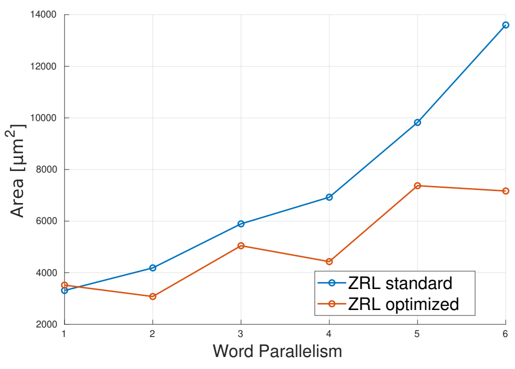
8.4 ZRLE implementation

In order to determine the most efficient version of the ZRLE engine, critical path and area/number of cells from Logic Synthesis have been used together with the latency values.
As also remarked in Section 7.1, The employed technology library has been a 28 nm Low Power from TSMC, with the corner cases in Table 7.1.
Naturally, all results from timing analysis are referred to the Worst Voltage Temperature corner, and only implementations that met the timing constraints have been taken into account. The target clock frequency of the platform is 200 MHz that correspond to a 5 ns period, however an additional slack of $2ns$ has been reserved to account for the wire delays connecting to the DiMArch, and for the some extra control logic around the ZRLE engine. Overall, only implementations with a critical path $\leq 3ns$ have been considered suitable.
Neither the standard combinatorial approach nor its optimized version met the target timing constraints, so they had to be put aside, leaving the mixed approach as the only viable alternative. The optimal solution must strike a balance between clock cycles of latency and area occupation, so it has been searched with the following Figure of Merit:

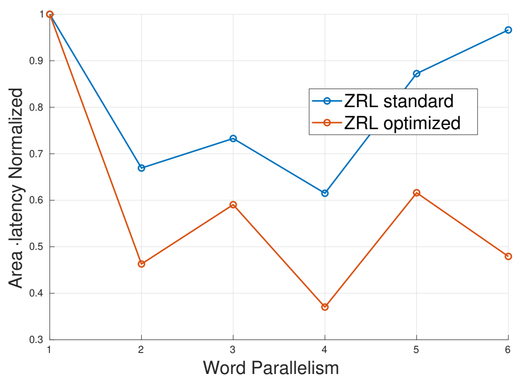
$$FoM = \frac{area}{max(area)} \frac{latency}{max(latency)} \tag{8.15}$$

Since it is desirable to minimize both area and latency, the optimum corresponds to the global minimum of the FoM. Area values at parity of word parallelism Wp are reported in Figure 8.6a, while the FoM can be found in Figure 8.6b.

Except for the case $Wp = 1$ the optimized ZRLE always attains a smaller



(a)



(b)

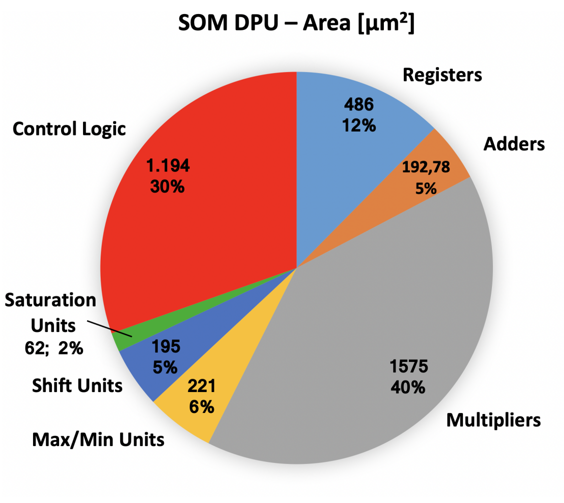
Figure 8.6: Logic Synthesis results for Mixed ZRL

area, with a performance gap that widens for high Wp. As expected, increasing the Layers size but halving their overall number has a major positive impact. It is interesting to see how area constantly increases for the standard ZRLE, while for the optimized ZRLE is bigger for odd Wp values than for even ones. That is a consequence of processing data in couples: one layer is added at every $Wp = 1+2k$, but its outputs are fully exploited only at $Wp = 2k+2$. It follows that odd Wp have same area devoted to Layers, but a bigger control logic to handle the greater number of intermediate results, hence their overhead in size. The best tradeoff has been found for $Wp = 4$ for both versions of ZRLE, so that is the value ultimately recommended for the ZRLE Engine.

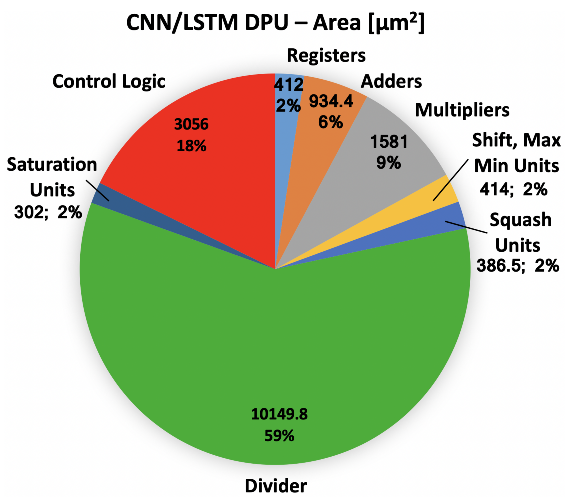
8.4.1 SOM Fabric Area Results

The main parameters related to the SOM fabric floorplan are listed in Table 8.7. DiMArch height has been set to $2h_{drra}+W_{channel}$, so that it could be accommodated on eight grid squares. This leads to the same total area for DRRA and DiMArch arrays. A view of the post place & route SOM fabric is available in Figure 7.2 (Chapter 7). The same view for CNN/LSTM looks very similar, so it has not been reported.
Since the main target of customization for the DRRA has been the DPU, Figure 8.7a presents the area occupation of its inner components. As expected, multipliers take up a fair amount of space, with a close second being the control logic, i.e. the multiplexers that allow to reuse the same blocks for different operations.

A visual representation of synthesis results for the entire SOM DRRA cell can be seen in Figure 8.8a, which uses the Top DRRA Tile as an example. Standard cells have been color coded to provide a glance at area occupation and spatial distribution of each basic DRRA component. It must be specified that Standard Cells falling under the 'Others'category (in grey) are part of the registers and control logic that

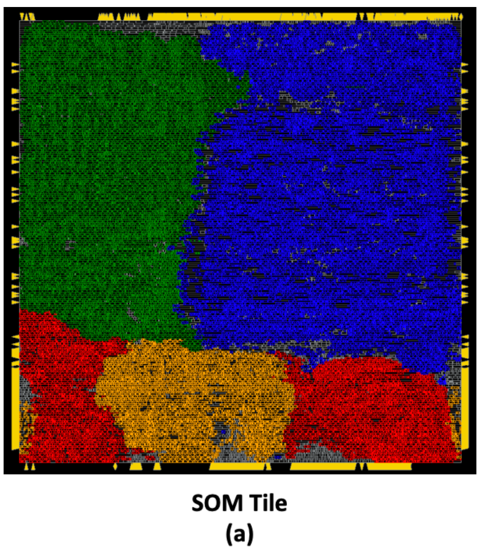


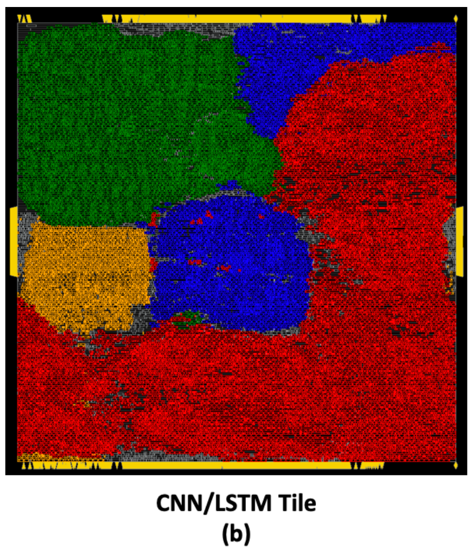
(a)



(b)

Figure 8.7: Comparison between a SOM and CNN/LSTM DPU.

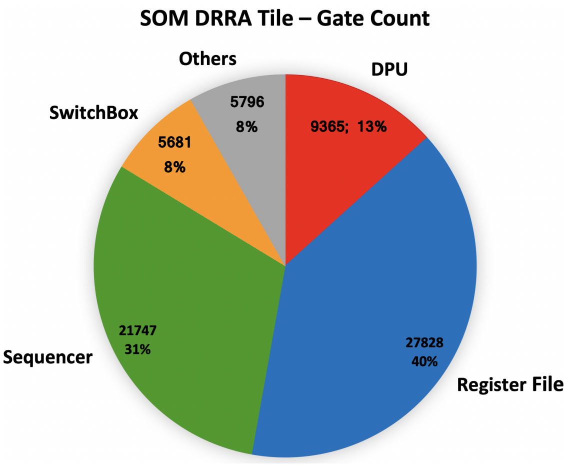




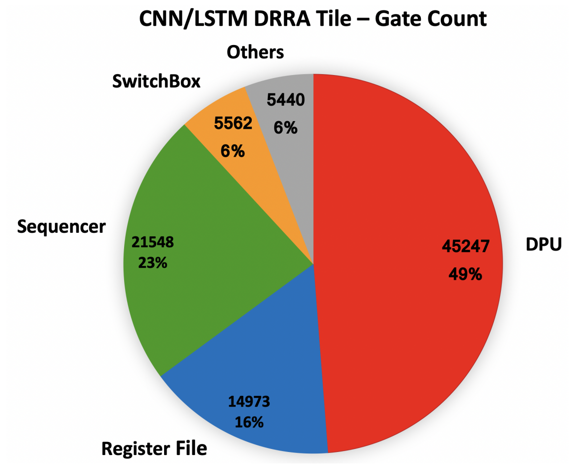
🟥 DPU 🟦 Register File 🟩 Sequencer 🟧 SwitchBox ⬜ Others

Figure 8.8: Area distribution of Top DRRA Tile - Components are color coded.

do not belong to the Tile Core, i.e. are contained in the wrapper. Thanks to its low complexity, the DPU takes up a fairly small area and this allowed to expand the Register File up to 64 locations. As a consequence, the RF became the major contribution to the DRRA area. The chart in Figure 8.9a quantifies the size of DRRA components in terms of gate count and reports their relative contributions. Table 8.8 contains the complete synthesis results for all DRRA tiles. Although not the main focus of this work, DiMArch data have been reported as well for completeness. The values are slightly different from one cell to the other due to the different optimizations performed by Innovus. It is interesting to note that the corner Switchboxes are missing half of the connections and so end up being smaller than the central ones. Moreover, the 'Others' field which represents the wrapper logic has bigger values in all top cells, since they have to handle communication with DiMArch cells.



(a)



(b)

Figure 8.9: Comparison between a SOM and CNN/LSTM Tile. Example based on Top Tile data from Tables 8.8 and 8.10.

8.4.2 CNN/LSTM Fabric Area Results

Floorplan dimensions for this fabric version are available in Table 8.9. In this case, DiMArch cells have a broader base as compared to SOM but contain same logic and same SRAM block, so their height has been set to a lower value of $1.67h_{drra} + W_{channel}$. In this way DiMArch cells don't match perfectly with the grid, but are still contained in eight blocks and have a very similar area occupation to the SOM case.
Figure 8.7b shows the area distribution for the DPU. With an area that is more than two times the SOM DPU and an occupation of 59%, the divider clearly represents the most bulky component, but at least it is reused for both exponential and softmax which reduces its cost. On the other hand, LUTs for Sigmoid and Tanh (Squash Units) appear to have a very low impact with only a 2% weight. Even If they were added to the SOM DPU they would take up around 9% of the area, which is still tolerable.
The considerable area increase induced by the divider is evident in Figure 8.8b. As a consequence, the RF had to be scaled down to 32 words in order to keep the

total area under control, and this is why it occupies less area than in the SOM tile. Sequencer and SwitchBox did not undergo modifications and in fact they take up a similar tile portion. All these considerations are proved correct by the gate count data in Figure 8.9b. It is worth recalling from Chapter 7 that ZRLE is not present in Figure 8.8b because it has been excluded from CNN/LSTM fabric synthesis due to its excessive area requirements. For completeness, Table 8.10 lists all the available data on the fabric synthesis results.

SOM Fabric Floorplan				
-	**Width**	**Height**	**Area**	**Channels Width**
DRRA Tiles	189	189	35721	18.9
DiMArch Tiles	189	396.9	75014.1	18.9
DRRA Area = 699437.025				
DiMArch Area = **DRRA Area** = 699437.025				
Fabric Area = 1398874.05				

Table 8.7: SOM Floorplan sizes. All dimensions are in [μm^2].

SOM Fabric Synthesis Results									
DiMArch Tiles	**Left**			**Center**			**Right**		
	Gates	Cells	Area	Gates	Cells	Area	Gates	Cells	Area
Logic	28849	8699	10095	35325	11989	13352.8	26877	8267	10159.5
SRAM Area	39000			39000			39000		
Utilization	66.5%			69.8%			65.5%		
DRRA Tiles	**Top Left**			**Top**			**Top Right**		
	Gates	Cells	Area	Gates	Cells	Area	Gates	Cells	Area
DPU	9365	4201	3540	10388	4546	3926.9	9868	4280	3730.1
Register File	27828	12042	10519.1	27716	12014	10476.9	28816	12081	10892.4
Sequencer	21747	7757	8220.4	22063	7795	8340.1	21928	7758	8288.8
SwitchBox	5681	3015	2147.4	7362	3333	2783.1	5625	3059	126.2
Others	5796	3176	2191	5622	2664	2124.1	5532	2890	2091.4
Total	70417	30191	26617.9	73151	30352	27651.1	71769	30068	27128.9
Utilization	74.5%			77.4%			75.9%		
DRRA Tiles	**Bottom Left**			**Bottom**			**Bottom Right**		
	Gates	Cells	Area	Gates	Cells	Area	Gates	Cells	Area
DPU	9487	4149	3586.2	9212	4166	3482.4	9315	4188	3521.3
Register File	28120	12252	10629.5	28098	12227	10621	28509	12283	10776.5
Sequencer	21854	7768	8261.1	21780	7783	8233.1	21832	7823	8252.7
SwitchBox	6191	3131	2340.4	5963	3144	2254	5791	3107	2189.2
Others	2350	1120	887.8	2462	956	930.2	1954	963	738.1
Total	68002	28420	25705	67515	28276	25520.7	67401	28364	25477.8
Utilization	72.0%			71.4%			71.3%		

Table 8.8: SOM Fabric Synthesis Results. Area values are in [μm^2].

CNN/LSTM Fabric Floorplan				
-	**Width**	**Height**	**Area**	**Channels Width**
DRRA Tiles	207.9	207.9	43222.41	18.9
DiMArch Tiles	207.9	365.4	75966.66	18.9
DRRA Area = 831604.725				
DiMArch Area = 705430.215				
Fabric Area = 1537034.94				

Table 8.9: CNN/LSTM Floorplan sizes. All dimensions are in [μm^2].

CNN/LSTM Fabric Synthesis Results									
DiMArch Tiles	**Left**			**Center**			**Right**		
	Gates	Cells	Area	Gates	Cells	Area	Gates	Cells	Area
Logic	28263	8871	10683.7	34096	11824	12888.4	26627	8231	10065.1
SRAM Area	40000			40000			40000		
Utilization	66.7%			69.6%			65.9%		
DRRA Tiles	**Top Left**			**Top**			**Top Right**		
	Gates	Cells	Area	Gates	Cells	Area	Gates	Cells	Area
DPU	42953	17404	16236.5	45247	17476	17103.6	42823	17440	16187.2
Register File	14653	5884	5538.8	14973	5845	5659.9	14931	5878	5644.2
Sequencer	21525	7696	8136.7	21548	7670	8145.4	21677	7726	8194.2
SwitchBox	5423	3035	2050.1	5562	3078	2102.4	5359	3046	2025.8
Others	5958	3251	2251.7	5440	2714	2055.9	5317	3076	2009
Total	90512	37270	34213.8	92770	36783	35067.2	90107	37166	34060.4
Utilization	79.2%			81.1%			78.8%		
DRRA Tiles	**Bottom Left**			**Bottom**			**Bottom Right**		
	Gates	Cells	Area	Gates	Cells	Area	Gates	Cells	Area
DPU	42754	17434	16161	42965	17376	16240.8	42935	17406	16229.7
Register File	15052	6090	5689.9	14850	5843	5613.4	15242	6071	5761.5
Sequencer	21647	7739	8182.8	21648	7716	8183.1	21642	7806	8180.6
SwitchBox	5222	3005	1973.9	5773	3111	2182.4	5427	3068	2051.7
Others	2516	1226	950.7	2642	1127	998.2	2141	1085	808.8
Total	87191	35494	32958.3	87878	35173	33217.9	87387	35436	33032.5
Utilization	76.3%			76.9%			76.4%		

Table 8.10: CNN/LSTM Fabric Synthesis Results. Area values are in [μm^2].

Chapter 9

Conclusions and Future Work

The purpose of this thesis was to demonstrate the feasibility of a SiLago-based CGRA for Artificial Neural Network applications. This task has been successfully carried out by showing a end-to-end design flow for the basic CGRA block and for the whole fabric. Starting from the target algorithms, the operations to be included in the SiLago block have been identified. In accordance with the SiLago framework, these operations have been manually implemented at RTL as a one-time engineering effort and grouped into two designs, the DataPath Unit and the Compression Engine, to be included in the CGRA blocks. These two components are the main contribution of this thesis, so the design trade-offs and decisions have been reported in detail. The high degree of parameterization provides both DPU and Compression Engine with good versatility and reusability. They can be used to explore different variants of the CGRA block and choose the best trade-off between area, power and throughput. The other important contribution to the SiLago framework was establishing and testing an EDA synthesis flow for the entire CGRA: this work has shown how to assemble the CGRA blocks into a fabric in compliance with the rules of Synchoricity and in a fully automated way. The end result of this work are two functioning SiLago-compatible CGRAs, one for CNN and LSTM networks and the other for SOM.
Detailed area reports have been included in this work, but due to lack of time power estimations have not been carried out, so completing the fabric characterisation is definitely the primary task to carry out in the future.
This thesis has explored a compression approach that takes data elaboration close to the bottom of the memory hierarchy and has exposed its inherent limitations, so power estimations will be useful to find out if this policy brings higher efficiency than the traditional methods or not. Once this has become clear, the fabric will have to be modified to integrate the Compression Engine.

As a last remark, this thesis has focused on the *hardware* implementation of CGRAs, but to actually execute ANNs they also require to be loaded with the correct configuration software. In other words, ANNs need to be mapped to the CGRAs. This is a task for the SiLago platform compiler so it is outside the scope of this thesis, but nevertheless it is needed to compare the platform perfomance with the other state-of-the-art designs for ANNs.

Bibliography

[1] Stanford, “Cs231n: Convolutional neural networks spring 2017,” 2017.

[2] V. Sze, Y. Chen, T. Yang, and J. S. Emer, “Efficient processing of deep neural networks: A tutorial and survey,” *Proceedings of the IEEE*, vol. 105, pp. 2295–2329, Dec 2017.

[3] T. Kohonen and T. Honkela, “Kohonen network,” *Scholarpedia*, vol. 2, no. 1, p. 1568, 2007. revision #127841.

[4] P. Nilsson, A. U. R. Shaik, R. Gangarajaiah, and E. Hertz, “Hardware implementation of the exponential function using taylor series,” in *2014 NORCHIP*, pp. 1–4, Oct 2014.

[5] P. Pouyan, E. Hertz, and P. Nilsson, “A vlsi implementation of logarithmic and exponential functions using a novel parabolic synthesis methodology compared to the cordic algorithm,” in *2011 20th European Conference on Circuit Theory and Design (ECCTD)*, pp. 709–712, Aug 2011.

[6] S. Gomar and A. Ahmadi, “Digital multiplierless implementation of biological adaptive-exponential neuron model,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 61, pp. 1206–1219, April 2014.

[7] S. Gomar, M. Mirhassani, and M. Ahmadi, “Precise digital implementations of hyperbolic tanh and sigmoid function,” in *2016 50th Asilomar Conference on Signals, Systems and Computers*, pp. 1586–1589, Nov 2016.

[8] N. Farahini, A. Hemani, H. Sohofi, and S. Li, “Physical design aware system level synthesis of hardware,” in *2015 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pp. 141–148, July 2015.

[9] Shuo Li, N. Farahini, A. Hemani, K. Rosvall, and I. Sander, “System level synthesis of hardware for dsp applications using pre-characterized function implementations,” in *2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pp. 1–10, Sep. 2013.

[10] J. M. Rabaey, “Silicon architectures for wireless systems,” *Berkeley Wireless Research Center white paper*.

[11] M. B. Taylor, “Is dark silicon useful? harnessing the four horsemen of the coming dark silicon apocalypse,” in *DAC Design Automation Conference 2012*, pp. 1131–1136, June 2012.

[12] A. Hemani, S. M. A. H. Jafri, and S. Masoumian, "Synchoricity and nocs could make billion gate custom hardware centric socs affordable," in *Proceedings of the Eleventh IEEE/ACM International Symposium on Networks-on-Chip*, NOCS '17, (New York, NY, USA), pp. 8:1–8:10, ACM, 2017.

[13] A. Hemani, N. Farahini, S. Jafri, H. Sohofi, S. Li, and K. Paul, "The silago solution : Architecture and design methods for a heterogeneous dark silicon aware coarse grain reconfigurable fabric," in *The Dark Side of Silicon : Energy Efficient Computing in the Dark Silicon Era*, pp. 47–94, 2017. QC 20171023.

[14] M. Z. Alom, T. M. Taha, C. Yakopcic, S. Westberg, P. Sidike, M. S. Nasrin, M. Hasan, B. C. Van Essen, A. A. S. Awwal, and V. K. Asari, "A state-of-the-art survey on deep learning theory and architectures," *Electronics*, vol. 8, no. 3, 2019.

[15] A. Kaplan and M. Haenlein, "Siri, siri, in my hand: Who's the fairest in the land? on the interpretations, illustrations, and implications of artificial intelligence," *Business Horizons*, vol. 62, no. 1, pp. 15 – 25, 2019.

[16] A. L. Samuel, "Some studies in machine learning using the game of checkers," *IBM Journal of Research and Development*, vol. 3, pp. 210–229, 1959.

[17] Y. Le Cun, L. D. Jackel, B. Boser, J. S. Denker, H. P. Graf, I. Guyon, D. Henderson, R. E. Howard, and W. Hubbard, "Handwritten digit recognition: applications of neural network chips and automatic learning," *IEEE Communications Magazine*, vol. 27, pp. 41–46, Nov 1989.

[18] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25* (F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, eds.), pp. 1097–1105, Curran Associates, Inc., 2012.

[19] 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.

[20] Y. Bengio, "Learning deep architectures for ai," *Found. Trends Mach. Learn.*, vol. 2, pp. 1–127, Jan. 2009.

[21] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *CoRR*, vol. abs/1502.03167, 2015.

[22] F. Gers and J. Schmidhuber, "Recurrent nets that time and count," vol. 3, pp. 189 – 194 vol.3, 02 2000.

[23] J. Chung, C. Gulcehre, and K. Cho, "Gated feedback recurrent neural networks,"

[24] T. Kohonen, "The self-organizing map," *Proceedings of the IEEE*, vol. 78, pp. 1464–1480, Sep. 1990.

[25] Y. Yang, D. Stathis, P. Sharma, K. Paul, A. Hemani, M. Grabherr, and R. Ahmad, "Ribosom: Rapid bacterial genome identification using self-organizing

map implemented on the synchoros silago platform," 07 2018.

[26] I. D. Campo, R. Finker, J. Echanobe, and K. Basterretxea, "Controlled accuracy approximation of sigmoid function for efficient fpga-based implementation of artificial neurons," *Electronics Letters*, vol. 49, pp. 1598 –1600, December 2013.

[27] A. H. Namin, K. Leboeuf, R. Muscedere, H. Wu, and M. Ahmadi, "Efficient hardware implementation of the hyperbolic tangent sigmoid function," in *2009 IEEE International Symposium on Circuits and Systems*, pp. 2117–2120, May 2009.

[28] A. Erdeljan, B. Vukobratović, and R. Struharik, "Ip core for efficient zero-run length compression of cnn feature maps," in *2017 25th Telecommunication Forum (TELFOR)*, pp. 1–4, Nov 2017.

[29] M. Denil, B. Shakibi, L. Dinh, M. Ranzato, and N. de Freitas, "Predicting parameters in deep learning," *CoRR*, vol. abs/1306.0543, 2013.

[30] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural networks," *CoRR*, vol. abs/1506.02626, 2015.

[31] Y. 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, pp. 127–138, Jan 2017.

[32] S. M. A. H. Jafri, A. Hemani, K. Paul, and N. Abbas, "Mocha: Morphable locality and compression aware architecture for convolutional neural networks," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 276–286, May 2017.

[33] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning.* MIT Press, 2016. `http://www.deeplearningbook.org`.