Politecnico di Torino

Politecnico di Torino

Ingegneria Informatica
A.a. 2021/2022
Sessione di Laurea Ottobre 2022

Dependable Hardware for Trustworthy Artificial Intellicenge

Relatori:
Prof. Ernesto Sanchez Sanchez
Prof. Alberto Bosio
Dr. Annachiara Ruospo

Candidati:
Salvatore Pappalardo

Abstract

Deep Neural Networks (DNNs) are currently one of the most intensively and widely used predictive models in the field of machine learning. DNNs have proven to give very good results for many complex tasks and applications, such as object recognition in images/videos, natural language processing, satellite image recognition, robotics, aerospace, smart healthcare, and autonomous driving. Nowadays, there is intense activity in designing custom Artificial Intelligence (AI) hardware accelerators to support the energy-hungry data movement, speed of computation, and memory resources that DNNs require to realize their full potential. Hardware for AI (HW-AI), similar to traditional computing hardware, is subject to hardware faults (HW faults) that can have several sources: variations in fabrication process parameters, fabrication process defects, latent defects, i.e., defects undetectable at time-zero post-fabrication testing that manifests themselves later in the field of application, silicon aging, e.g., time-dependent dielectric breakdown, or even environmental stress, such as heat, humidity, vibration, and Single Event Upsets (SEUs) stemming from ionization. All these HW faults can cause operational failures, potentially leading to important consequences, especially for safety-critical systems. Therefore, ensuring the reliability of HW-AI platforms is crucial, especially when HW-AI is deployed in safety-critical and mission-critical applications, such as robotics, aerospace, smart healthcare, and autonomous driving. This thesis has a double purpose: creating a neural accelerator based on the systolic array architecture and studying the reliability of a hardware implementation of the well-known LeNet DNN. Different from the other works, the reliability of the network is assessed based on the specific architecture, underlying its strengths and weaknesses. The results show that the architecture is safe, on average, in 86% of the cases with the considered faults. Furthermore, the thesis provides a full explanation of the environment used for such a study in detail, allowing interested fellows to replicate the study.

Table of Contents

List of Figures IV

1 Introduction 1

2 Background 2
2.1 Systolic array . . . 2
2.2 Neural Network . . . 4
2.3 Hardware mapping . . . 6
2.4 Halved array mapping . . . 9
2.5 Required hardware . . . 10

3 Proposed approach 11
3.1 Hardware model . . . 11
3.1.1 Test bench . . . 11
3.1.2 Convolutions layer . . . 12
3.1.3 Channel . . . 12
3.1.4 j-shift register . . . 12
3.1.5 Systolic array . . . 13
3.1.6 Activation block . . . 13
3.1.7 Optimizations . . . 13
3.2 Neural Network program . . . 13
3.2.1 Argument parsing . . . 14
3.2.2 Input sequence generation . . . 14
3.2.3 Simulation output propagation . . . 14
3.3 Operating System Environment . . . 15
3.3.1 Optimization . . . 15

4 Experimental Results 16
4.1 Network parameters . . . 16
4.2 Faults . . . 17
4.2.1 Fault campaign 1 . . . 18
4.2.2 Fault campaign 2 . . . 20
4.2.3 Fault campaign 3 . . . 22
4.3 Metrics . . . 24
4.3.1 Fault campaign 1 . . . 25
4.3.2 Fault Campaign 2 . . . 32

4.3.3 Fault campaign 3 35

5 Conclusions 40

A Failed attempt 41
A.1 Horizontal overlapping region 41
A.2 Working hardware implementation 41

B Fully connected layer implementation 44

C Fault identification 46

Acronyms 49

Bibliography 50

List of Figures

2.1 MAC (multiply-accumulate) processing element 2
2.2 Processing element detailed implementation 3
2.3 4x4 systolic array 3
2.4 Perceptron model 4
2.5 ReLU function 4
2.6 Simple neural network model with three layers. The input layer has three perceptrons, the hidden layer four and 5
2.7 LeNet architecture 5
2.8 Input sequences example 7
2.9 2.9a architecture comprising weights input logic. 2.9b Vertical overlapping principle. 8
2.10 Horizontal overlapping region. Yellow and green lines represent sequences for two different vertically adjacent PE, and the labels of the same colors represent the weights. 9
2.11 Split convolution example 9
2.12 j shift register in which the value of j is n 10

4.1 Weights distribution 17
4.2 Bits weights distribution per position (value 1) 17
4.3 Weights distribution per channel 18
4.4 Weights distribution per channel - detailed 19
4.5 Stimuli distribution 19
4.6 Number of faults per each row 20
4.7 Number of faults per each column 20
4.8 Number of faults per injected bit 21
4.9 Faults per channel 21
4.10 2D histogram describing the number of injections per single location on the systolic array. 22
4.11 Number of faults per each injected bit 22
4.12 Figures 23
4.13 2D histogram describing the number of injections per single location on the systolic array 24
4.14 [%]Safety performance for the three FC. The graphs show the percentage of injections for the different categories. 25

4.15 Number of faults per each row and column. On the left is the plain number (in red), and on the right, the number is normalized with respect to the number of injections. 26
4.16 Cumulative number of unsafe faults per injected row (on the right) and column (on the left). The orange lines represent a linear approximation of the data 26
4.17 faults distribution per channel 28
4.19 Number of faults per each stimulus 30
4.20 Number of faults per fault value (i.e. whether they are stuck-at 0 or stuck-at 1) 31
4.21 Number of unsafe faults comparison between stuck-at 1 and stuck-at 0. In yellow is the number of unsafe faults normalized per bit injection. In blue is the number of unsafe faults generated by a stuck-at 1. 31
4.22 Number of faults per each row and column. On the left is the plain number (in red), and on the right, the number is normalized with respect to the number of injections. 32
4.23 Faults distribution per channel 33
4.25 Number of faults per stimulus class 34
4.26 Number of faults per fault value 35
4.27 Number of unsafe faults comparison between stuck-at 1 and stuck-at 0. In yellow is the number of unsafe faults normalized per bit injection. In blue is the number of unsafe faults generated by a stuck-at 1 35
4.28 Number of faults per row 36
4.29 Faults distribution per channel 36
4.30 Number of faults per bit 37
4.31 Number of faults per stimulus 38
4.32 Number of faults per fault value 38
4.33 Number of unsafe faults comparison between stuck-at 1 and stuck-at 0. In yellow is the number of unsafe faults normalized per bit injection. In blue is the number of unsafe faults generated by a stuck-at 1 39

A.1 Horizontal overlapping region 42
A.2 Selective shift register (SSR) 43
A.3 Selective shift register simulation 43

B.1 Fully connected layer sketch 45
B.2 Fully connected layer implementation with systolic array 45

Chapter 1

Introduction

As of today, neural networks are ubiquitous in the computer engineering world and there are lots of applications exploiting this technology, even in real-time safety-critical systems. Identifying critical inferences, i.e. those that might lead to a safety hazard, is essential for designing reliable hardware.

The objective of this work is to study and characterize the safety of a neural accelerator to assess its reliability. Specifically, the study shows that on average 86% of the faults are safe on the architecture, and 68% are entirely masked. The study, similarly to [1], shows the reliability of the network but rather than studying the network itself, this study underlines its reliability on a hardware implementation. An extra point of this approach is related to the types of faults applicable to the network. Indeed, there is the possibility of propagating the faults not only through the network but also in the layer itself. The DNN, described later, is the LeNet-5 [2]. This particular implementation is fully explained in chapter 3. It uses a limited precision opening the possibility of deploying the network to limited power hardware, such as embedded devices, and enables the architecture to have a smaller footprint while remaining acceptably stable, as shown by [3], [4]. Indeed, the former shows how a Neural Network works with a binary set of weights $\{-1, 1\}$, while the latter use a ternary set of weights $\{-1, 0, 1\}$. These networks were able to perform almost equally to the full precision network, motivating the use of this data type. Furthermore, the authors of [5] show how the fault tolerance of the network depends on the data type used. Indeed, they show that a balance exists which allows enough precision for performance and good fault reliability since the tolerance is shown to be smaller with the most precise representations.

This thesis aims to assess the reliability of a specific architecture implementing the neural network and seeks for particularly peculiar vulnerabilities.

Chapter 2

Background

In this chapter, the concept of the systolic array will be introduced. In Section 2.2 the structure of the used Neural Network will be explained, introducing also the type of layer considered[1]. Later, the mapping between the hardware operations and the considered layers will be discussed (Section 2.3) to eventually conclude with the required hardware (section 2.5).

2.1 Systolic array

A systolic array is defined as a lattice of synchronous and locally connected processing elements (PE, also called nodes or cells) that can perform iterative algorithms with regular data dependencies [6]. Each PE independently computes a partial result as a function of the data received from its upstream neighbors, stores the result within itself, and passes it downstream. Figure 2.3 shows the architecture.

One type of PE is MAC (multiply-accumulate) which can be used in different applications such as matrix multiplication or convolution (as in this case).

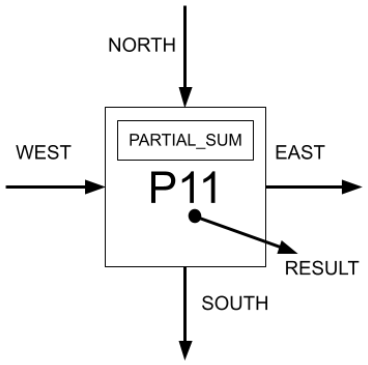


Figure 2.1: MAC (multiply-accumulate) processing element

Figure 2.1 shows a PE. The inputs are `NORTH` and `WEST`, while the outputs are `EAST`, `SOUTH` and `RESULT`. The architecture is synchronous (i.e. the data flow is regulated by a clock signal) with a synchronous reset input, which is essential because the reset signal put 0 in `PARTIAL_SUM` register and allows the element to be used for different unrelated computations.

[1]Two types of layers were considered, but only one has been simulated. A discussion about the other type can be found in the appendix B

A PE does fundamentally two things on each clock cycle:

- it applies a function to the inputs and the state[2],
- and puts `NORTH` data on `SOUTH` and `EAST` data on `WEST`.

There is a `PARTIAL_SUM` register which is directly tied to `RESULT`. This register contains the result of the computation $\texttt{NORTH} * \texttt{WEST}$

A more detailed implementation can be seen in Figure 2.2.

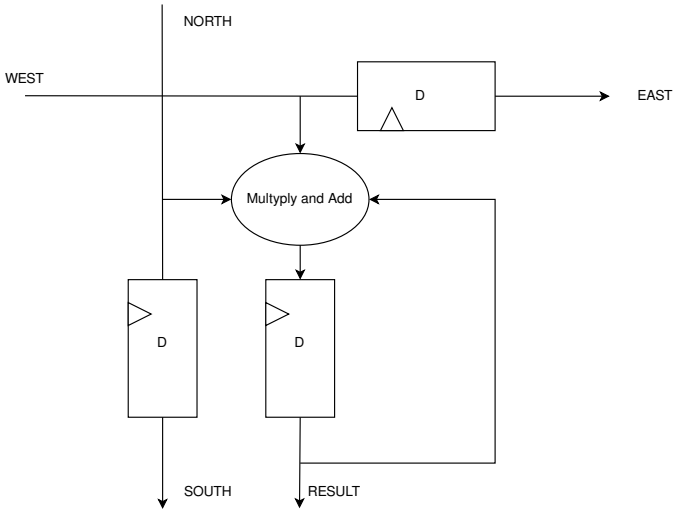


Figure 2.2: Processing element detailed implementation

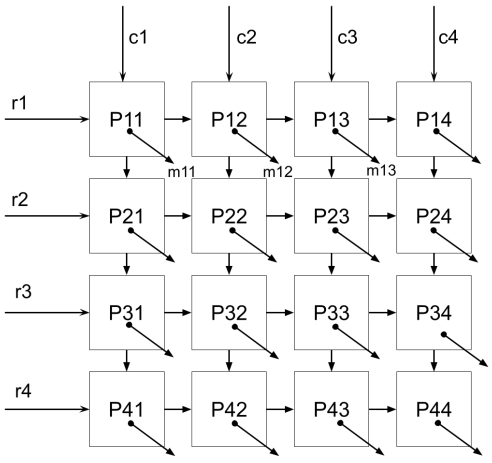


Figure 2.3: 4x4 systolic array

[2]In our case the state is the `PARTIAL_SUM` register and the applied function is $PARTIAL\_SUM + NORTH * WEST$)

2.2 Neural Network

A neural network is a program that tries to imitate the inner working of a brain. The general structure of a NN is composed of layers of perceptrons [7]. A perceptron represents a single *neuron* having an input, an activation function and an output. Figure 2.4 shows an example of perceptron. It has 3 inputs x_1, x_2, x_3 and three weights w_1, w_2, w_3, each of which associated to the corresponding input. The perceptron performs a computation of the inputs which is as follows:

$$y = \sigma(w_1 x_1 + w_2 x_2 + w_3 x_3)$$

The function σ is called **activation function** and introduce non-linear computations which are needed when classify non-linear data. Other than that, it tries to imitate actual neurons which have some sort of threshold for being considered active.

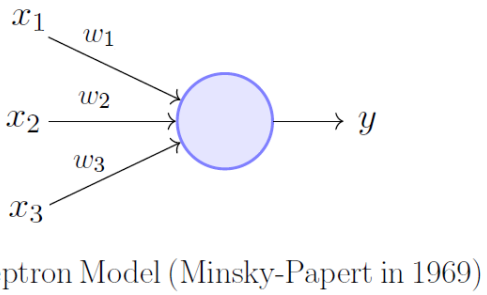


Figure 2.4: Perceptron model

The function can have different shapes. The most basic one is the following:

$$\sigma(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}$$

Nowadays different activation function exists, but the basic idea has been always the same: non-linearity. The implementation we will see later uses a function called ReLU (Rectified Linear Unit) [8]:

$$\text{ReLU} = \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}$$

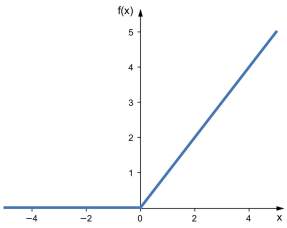


Figure 2.5: ReLU function

A network is then composed of perceptrons organized in layers. In particular, the first layer corresponds to the inputs given to the network from the external world. The last layer corresponds to the output of the network. The layers in the middle are called *hidden* and they are the place where *the magic happens*. Figure 2.6 shows a simple network with three layers. The input layers have three perceptrons, there is only one hidden layer (with four perceptrons) and the output is composed of two perceptrons. Note that each perceptron of a layer is **fully connected** to each perceptron of the following layer. That is why this type of level is usually referred to as **fully connected**.

This type of level is mono-dimensional since there only one single value determines the position of a perceptron (i.e. inside a layer: the network as a whole is

bi-dimensional). More complex networks exist. Indeed, a fully-connected disposition hardly copes with multi-dimensional input such as images. For this type of input, different strategies have been created. In [8] the authors were the first to create a convolutional layer. It performs a 2D convolution operation on an input image. The perceptrons are the same, but the relationship between the levels is different. Furthermore, in this context a convolutional layer is 3-dimensional.

The input is an image, so there is spatial information that without a 2D convolution would be lost. Furthermore, the convolution process is repeated more than once with different weights, generating a third dimension that will be later called **channel**.

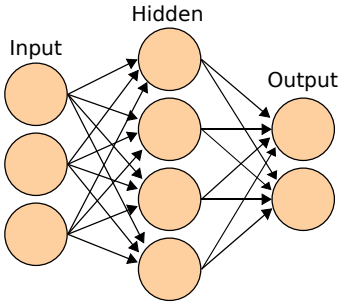


Figure 2.6: Simple neural network model with three layers. The input layer has three perceptrons, the hidden layer four and

The first goal of this thesis was to implement the first layer of a neural network and, among all, a custom implementation of LeNet-5[2] was chosen for its simplicity. The used implementation has a 32x32 gray-scale input (for a total of 3 dimensions) and has a limited number of layers while still remaining a Deep[3] Convolution Neural Network (CNN). Figure 2.7 shows its architecture.

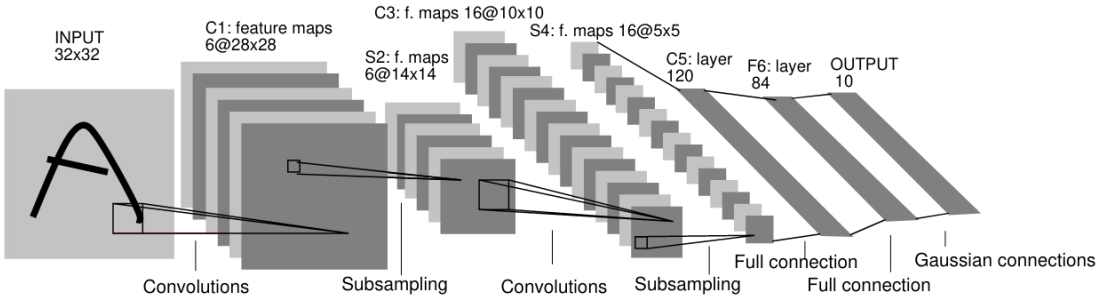


Figure 2.7: LeNet architecture

An important difference is the type of data that this implementation uses. Rather than the common float 32-bit IEEE-754 standard, it was made with signed and unsigned 8-bit values.

This is because this representation is much more efficient than full floating point

[3]a CNN is *deep* when there are at least 3 hidden layers.

32-bit values, so it can be used with embedded devices which usually have limited computational power and memory [3].

The first layer is a convolutional layer and implements a convolution operation[4]. More precisely it is a discrete 2D convolution:

$$(f * g)[x, y] = \sum_{i=-\infty}^{+\infty} \sum_{j=-\infty}^{+\infty} f[x, y]g[x - i, y - j]$$

In the context of this thesis, we will have a **kernel** containing the weights and will be denoted by $\mathcal{K}$ and an image or **stimulus** and will be denoted by $\mathcal{I}$. Furthermore, we will assume for simplicity that both $\mathcal{K}$ and $\mathcal{I}$ are squared. As a final general assumption, we will assume that the kernel $\mathcal{K}$ is always smaller than the image $\mathcal{I}$.

All the experiments were made using the MNIST [2] validation set, composed of 10'000 images of 32x32 gray-scale pixels. The first layer is thus implemented as a 28x28 array. This is because the dimensions of the array depend on the output.

2.3 Hardware mapping

The first task to be performed for the study was understanding how to map a convolution operation to the hardware. In other words: we need to find the correct input sequences such that the systolic array performs the wanted convolution.

First of all, we need to clarify the meaning of *input sequences*. Each input sequence is an ordered set of values that are given as input to one of the marginal PE[5]. In figure 2.8 the input sequences for the following matrix multiplication.

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} \times \begin{pmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{pmatrix}$$

It is noticeable the presence of leading and trailing zeros for each input sequence. Those zeros are vital because provide the synchronization needed for each PE. Indeed, for instance, during the second clock cycle element a_{14} reaches P_{12} and only at that point it is possible to put b_{42} inside P_{12}. Furthermore, especially for the considered application (bi-dimensional convolution) some elements need to be passed forward on sum PE but do not participate in the computation for that specific PE, thus the zero assures no *garbage computation* is put on any PE.

Continuing on the input sequences construction, we can make a very useful observation: since the kernel $\mathcal{K}$ is always the same for each pixel of $\mathcal{I}$, we will have the same input sequence for the weights on each PE, but shifted (i.e. inserting zeros) accordingly with some measure of **depth**. In this context, we define depth as the maximum between its vertical position and its horizontal position. For example, element P_{34} has *depth* of $max(3,4) = 4$ and element P_{14} has *depth* of $max(1,4) = 4$

[4]In the literature, the term "convolution" is widespread, but mathematically speaking the operation is better described as a correlation

[5]A marginal PE is one whose position is along the first row or column.

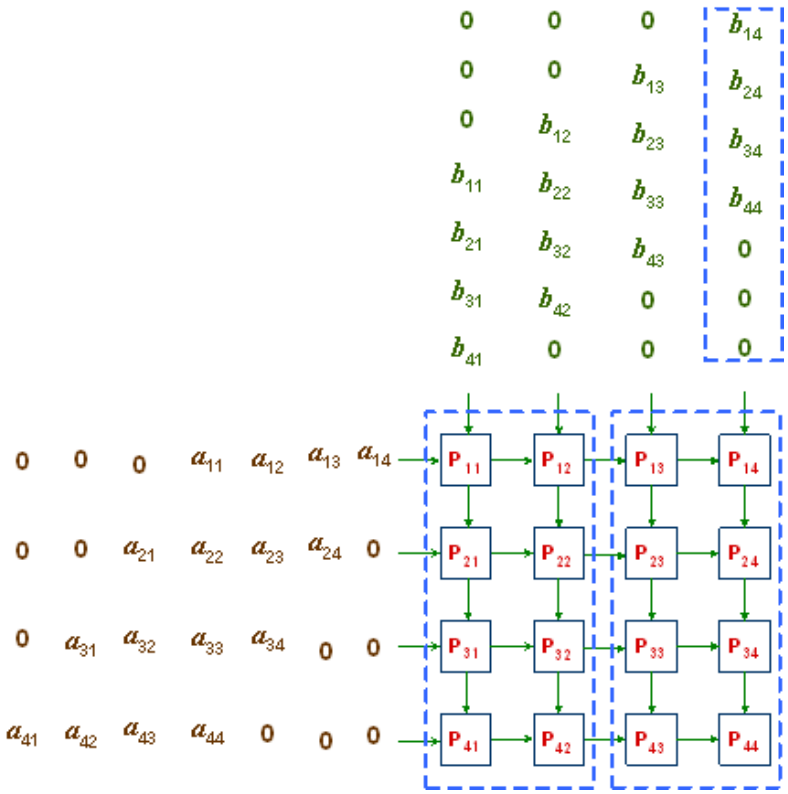
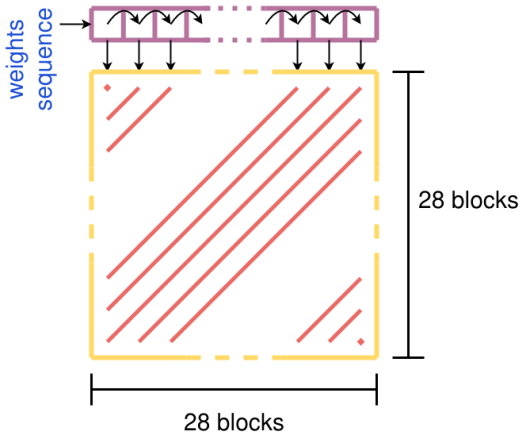


Figure 2.8: Input sequences example

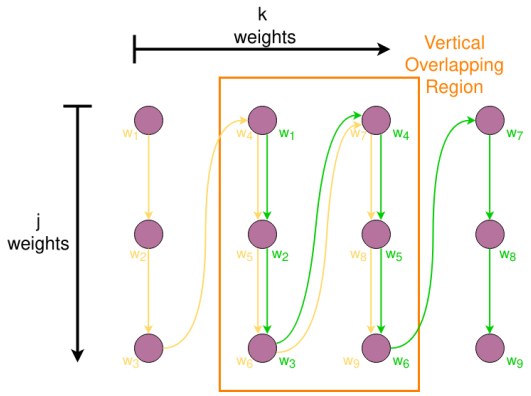
and element P_{11} has *depth* of $max(1,1) = 1$. The depth represents the number of clock cycles an element has to *wait* before doing any useful computation. This measure also corresponds to the number of leading zeros to add to each input sequence. Figure 2.9a shows a simplified architecture demonstrating this principle. Vertically, it is possible to see that, since each PE passes data from `NORTH` to `SOUTH`, each PE is able to synchronize quite automatically. Horizontally, we need to take care of the leading zeros, as explained above. To achieve this goal, we can just use a shift register and it fills automatically the sequences with the leading zeros. This little extra hardware guarantees a single memory to store the weights and automatic construction of the correct input sequences on the fly.

The input sequences for the stimulus are a little more involved. It is necessary to use a so-called **vertical overlapping region**, as shown in figure 2.9b. That is to exploit the systolic array architecture. For each output pixel (directly corresponding to a single PE, it is necessary to do $j \times k$ multiplication. Notice that, to completely exploit the vertical overlapping region, it is needed to process each output pixel in a column-by-column fashion. These observations are enough to build the stimulus input sequences, but for making a working model, we need to modify the weights sequences.

To explain that, let us focus on Figure 2.9b. The yellow and green labels indicate the weight to be used for each pixel, and it is notable that weight w_1 is used again only **after the first j elements**. This implies that before forwarding the weight horizontally, it is necessary to wait for j clock cycles. That is why it was necessary



(a) Weights sequence generation with a shift register. In yellow is a systolic array, in purple is the shift register, and in blue are the weights. The red lines represent PE with the same weight at the same clock cycle and thus have the same depth.



(b) Vertical overlapping region in orange. Each purple dot represents a pixel of $\mathcal{I}$ and the two curves in yellow and green represent two different convolutions. The kernel has j rows and k columns.

Figure 2.9: 2.9a architecture comprising weights input logic. 2.9b Vertical overlapping principle.

to introduce a piece of hardware called *j-shift register*. Such a piece of hardware is described in section 2.5

The vertical forward exploits the corresponding **horizontal overlapping region**, as shown in figure 2.10. It is visible that the weights can be forwarded on the next clock cycle and also a part of the stimulus is shared between the different outputs. Indeed, an attempt to use additional hardware in order to generate stimuli sequences (as done with the weights) was made, but unsuccessfully. Refer to appendix A for more information.

As one can understand by the mapping explained above the size of the array correspond to the size output of the layer. For instance, the first layer has 32x32 input pixels and 28x28 output pixels. The size of the array will be 28x28, while the input size is *hidden* in the computations. Indeed, each output node performs a different convolution which is just a finite number of multiplications and additions.

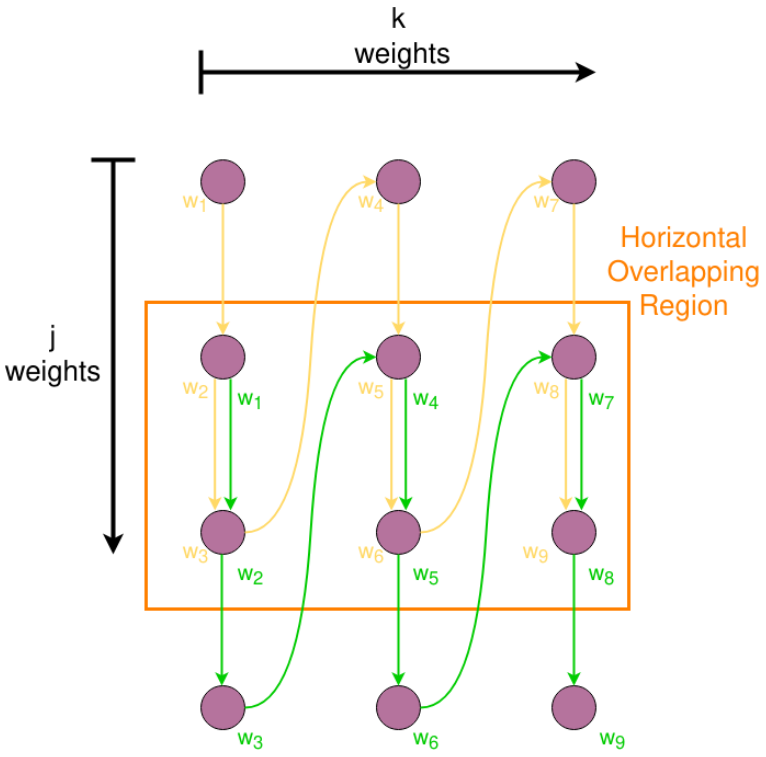


Figure 2.10: Horizontal overlapping region. Yellow and green lines represent sequences for two different vertically adjacent PE, and the labels of the same colors represent the weights.

2.4 Halved array mapping

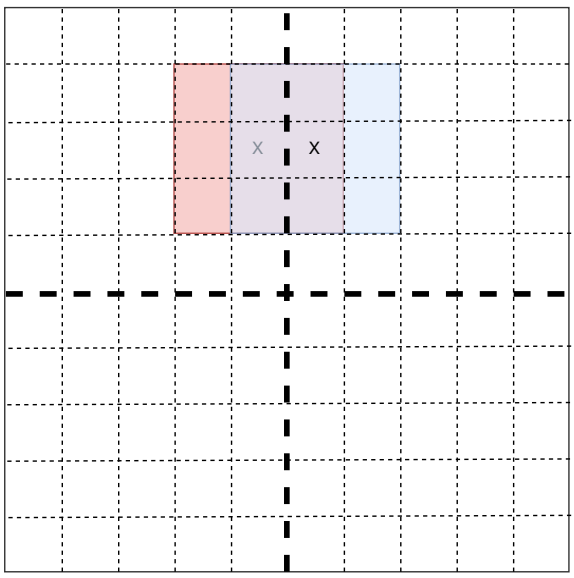


Figure 2.11: Split convolution example

It is in theory possible to resize the array. This approach has the merit of minimizing the production cost and the complexity of the hardware, which is simple to manufacture. In doing so, it is necessary to subdivide the image into sub-images and then convolve each sub-image independently from the other. Furthermore, some sort of additional logic has to be considered to synchronize the different convolutions.

First of all, consider the different convolutions. When splitting the image, it is necessary to consider that at the pixels at the borders of the split are used for more than one passage in the convolution. This

means that some parts of the image are used multiple times. In other words: the parts of the image used more than once have to be used as padding. This fact has to be taken into account when generating the input sequence. Figure 2.11 shows an example of split convolution. The two `xs` identify the location of the convolutions. There is, as said, an overlapping region and it exits outside the sub-image border.

2.5 Required hardware

This section summarizes the full hardware needed for performing a VHDL simulation.

The first important component is called `j-shift register` and it is a custom component. The `j-shift register` is nothing but a shift register that delays j clock cycles before the data is shifted to the next position. Conceptually it is a *super-shift register* in which each super-cell is itself a shift register with j positions. The output of the super-cell is the first cell of the inner register. This concept is shown in figure 2.12. Another way to think about it is a simple shift register

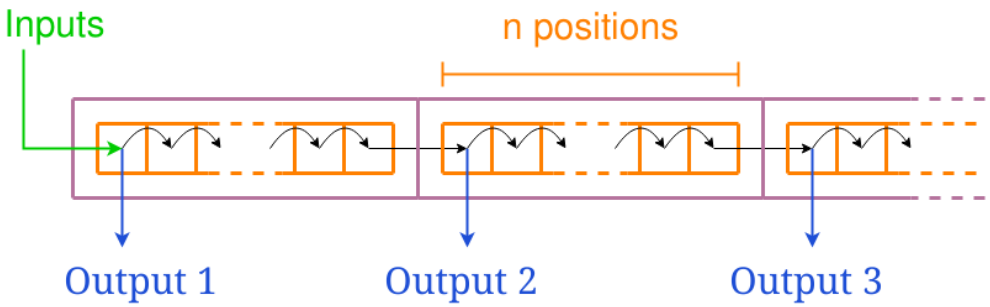


Figure 2.12: j shift register in which the value of j is n

whose output is every j-cells. This description corresponds to the actual VHDL implementation and it is delved into in section 3.1.4.

Another important component is the ROM. In reality, it is most likely to talk about RAM, since its role consists in holding the input sequences, but for the sake of the simulation, a rom model was used. Exactly as it sounds, it binds together an address to a value. It is important to remark the *sequentiality* of the inputs which is strictly related to the rom component. Indeed the ROM has to be filled in order so that a counter can be used for the inputs.

A counter model was implemented as well and it was used in a *program counter* fashion. Each counter instance is coupled with a counter so that the ROMs are independent of one another. And with this also a simple clock model was needed.

To implement the reduced version of the convolutional layer (Section 2.4) it was necessary to use a multiplexer so that for each section of the image processed the output could be saved and the systolic array reset.

Chapter 3

Proposed approach

In this chapter we will see the detailed circuit implementation, after that, we will see the Neural Network software and how it was modified for our purposes, and finally, we will see the remote environment used for running all the experiments.

3.1 Hardware model

In order to produce the convolutional layer, as explained in the previous chapter, VHDL was used extensively. The design (as common with hardware design) is hierarchical. Each level will be briefly explained and then each one of the main components will be detailed.

The uppermost level of the design is a test bench. The test bench instantiates a convolutional layer and contains all sorts of constants for the layer. All the logic for the management of the simulation are inside the `convolutional_layer` component, which in turn instantiates a number of `channel` components and also a few `rom` components and the relative address `counters` components. Each `channel` instantiates a so called `j_shift_register` for the input of the weights (as explained in the previous chapter in section 2.3) and a `systolic_generic` component, which is an implementation of a systolic array. In the first version, `activation_block` components were also used but were later removed for performance issues that will be explained later.

3.1.1 Test bench

The test bench, as previously explained, has the only purpose of instantiating a `convolutional_layer`. It is used for declaring an entity called tb (which stands for test-bench) and instantiating a layer. This component also contains some layer's hyper-parameters, such as the output width and height (respectively `rows` and `cols`), the kernel width and height (resp. `kerR` and `kerC`) and the bit depth (`depth`). The entity also needs to describe the `conv_layer` component for instantiating it.

3.1.2 Convolutions layer

The convolutional layer component is the heart of the implementation. This component instantiates all of the main signals (such as clock and reset) and the channels which contain the systolic arrays. This component also contains the most important logic:

- a `reset process` which assures each component and signal is in the correct initial state,
- a `clock process` which drives the clock signal high and low with the specified `CLOCK_PERIOD`,
- and a `write in memory process`, which waits for a specific amount of time and then drives high a signal indicating it is time to stop the simulation. This process is essential for data gathering. Indeed this signal is given in input to the channels which have the important task of writing the output files.

As previously anticipated, `conv_layer` also instantiates some ROMs and the relative address counters. These are simple components not worth deeply explaining.

The only important thing about the `rom` component is a function that reads a file in order to make the model flexible from the input point of view. This way, the files could just be replaced, thus skipping a new compilation[1] of the design.

3.1.3 Channel

The `channel` component is, in a way, the most important, since it contains the `systolic array`. Apart from that, this component contains a `rom` and a `j_shift_regiter` for the weights[2].

The last thing worth mentioning is the process `WriteInMemory`. This process has the crucial role (as anticipated in subsection 3.1.2) of writing the output files. The listing shows a `wait` for the `done` signal to become high and afterwards the function `writeMatrixContent` is called to store the computed values to a file. All the other function parameters are obtained directly from the test bench.

3.1.4 j-shift register

As explained in section 2.3, a j-shift register was implemented as a long shift register in which not every cell is connected to the output, but on one each j.

It is possible to see the *long shift register* as identified in the signal `register`. Indeed, as shown in the code, each clock cycle it is shifted by one position.

The outputs, however, are tied to each `i*j-1` register, as seen in line 15. That means that only the indices which obey the rule $i \mod j = 0$ are tied to the output.

[1] Each time the VHDL code changes, the model has to be compiled again.

[2] Each channel has its own set of weights, that is why such components are needed.

3.1.5 Systolic array

The component `systolic_generic` is a generic description of a systolic array in which each processor is a MAC (multiply-accumulate).

This implementation has the possibility to choose the type of operands (i.e. whether they are signed or unsigned) and behaves accordingly.

Also, the output is four times as wide as the input, which is due to some overflow issues.

3.1.6 Activation block

The activation block is a rather simple component. It implements a ReLU (Rectified Linear Unit) using the middle input bits. Indeed, the value is right-shifted nine times and then saturated (to 0 if the value is negative or to the maximum re-presentable value).

3.1.7 Optimizations

The first time the data-gathering script was run, it was estimated that it would complete in 231 days. That is a prohibitive time for an experiment in the context of a Master's thesis. So a few optimizations were made. In this section, hardware model optimization will be discussed.

The first thing was to remove the activation blocks. The `activation block` is modeled as an asynchronous circuit. This means that each change on its input schedules a new time delta in order to compute the new value [9] and since the input of each activation block is a MAC, a lot of time was required to perform the activation on useless data. Thus, the activation was moved to a simple `c` program which reads the output of the simulation and performs the activation one single time.

The most important optimization was based on a simple, but powerful, observation. Each injected fault only affects one of the six channels of the network. Since each channel is independent of the other, the unaffected ones performed just as well as the gold simulation. The idea then was to remove the other five channels and simulate only the faulty one. This trick saves more than 5s per simulation!

3.2 Neural Network program

The neural network program is a c-implementation of a convolutional neural network written using the n2d2 platform [3].

In this study's context, the program had two main tasks:

1. generates the input sequences for the simulation,
2. propagates the simulation-generated output to make a prediction.

[3]For more information: https://github.com/CEA-LIST/N2D2

For achieving these objectives the program was modified in order to accept arguments. I will briefly introduce the argument parsing logic and then the main modifications exploiting the parsed arguments.

3.2.1 Argument parsing

The argument parsing is really simple. It is done using some binary variables which are "switched" if the argument was given on the program call.

It iterates over `argv` and whenever it matches with the expected string, the corresponding option variable.

The `OPT_` variables are global and declared in the same file containing `main`, other source files import `opts.h` which contain an `extern` declaration of the same `OPT_` variables.

3.2.2 Input sequence generation

As explained in section 2.3, the input sequences are the actual input of the network. To simplify their generation, the program was exploited since it already provided a reliable file reading mechanism. Indeed, the input is supposed to be a `pgm` image which a separate program was otherwise supposed to read.

This section of code (which will not be entirely listed due to its length) behaves as follows:

- if `OPT_SAVE_FIRST_LAYER_INPUT` is true, then two sets of files are generated: one for the complete sequences and another for the reduced sequences (used for experiment 3),
- if `OPT_SAVE_WEIGHTS` is true, then the weights sequence is generated,
- if `OPT_SAVE_FIRST_LAYER_OUTPUT` is true, two different things happen. The *simple thing* is that the already activated output is saved on a set of files. The *difficult thing* is saving the non-activated outputs. To do that, it was necessary to dig inside the code of the network and find the right section. There I added the required instructions,
- finally if `OPT_ONLY_FIRST_LAYER` is true, the program terminates prematurely after the first layer pass.

These three options offer a great deal of flexibility. It is indeed possible to generate weights independently of input sequences and also entire program execution.

We will later see how these options became so useful.

3.2.3 Simulation output propagation

There is not much to say about this feature, it is simple as it sounds. If the option `OPT_FIRST_LAYER_FROM_FILE` is true, then the set of files corresponding to the output is read and a matrix inside the program is filled. After that, the program continues executions. Finally, if the option `OPT_SAVE_PROB_VEC` is true, the output of the network is saved on a file with space-separated values.

3.3 Operating System Environment

The highest level of the project was using the Operating System for gathering all the data and also analyzing it. Indeed, several bash scripts and simple programs were created for that purpose.

The most important of all are the `start_fault_campaign` scripts. These scripts are the heart of the project. Those are responsible for organizing the files, running the simulation and a preliminary analysis of the data, to see if there is any difference between the gold data and the faulty one or if the fault is completely masked[4]. Generally speaking, these scripts have the following behavior:

1. Verify the weights sequences files exist, if not generate them.
2. Iterate for each stimulus for each fault.
3. If the fault-stimulus pair has been processed, skip it, otherwise run the corresponding simulation.
4. Check the differences between the faulty simulation over the golden one.
5. Activate the output, meaning apply the ReLU function,
6. check the differences between the faulty activated output over the golden one.

3.3.1 Optimization

As said in subsection 3.1.7, some optimizations were needed to speed up the data gathering. Also at this level, some were made.

- Having switched from simulating six channels to just one, the higher level script needed to:
    1. change the name for the relevant files. That is because the simulation will always enumerate the channels starting from 1, so also the input files needed to be correctly enumerated.
    2. change the fault location. Again, only channel 1 exists, so it is impossible to inject a fault into channel 2.
- To maintain a *fresh* process state, instead of simply using two for loops (as it was initially done), the external loop was replaced by an exec of the script itself paired with a numeric increasing argument.

[4]We will add details in chapter 4, but for *completely masked* fault I fault which doesn't affect the output of the layer

Chapter 4

Experimental Results

Using the environment explained in the previous chapter, three main experiments were carried out:

- full 28x28 architecture injecting in the weights lines
- full 28x28 architecture injecting in the results bits
- halved 14x14 architecture injecting in the weight lines

Each experiment was performed by simulating 1'000 stimuli. For each stimulus, a total of 200 different faults were injected for a total of 200'000 simulations. The software used for the simulation is modelsim.

After running all the simulations, for ease of analysis, the data were gathered in a `sqlite .db` file which is relatively small than the plain text representation used in the previous step. The analysis was then performed using python interactively with the software JupyterLab.

4.1 Network parameters

For understanding the behavior of the network, a number of statistics over the network's parameters (including inputs and outputs) had to be gathered. Firstly, the distribution of the weights. It is noticeable in figure 4.1 that almost one-third of the interval of possible values is empty. Indeed, all the weights lie in the interval $[-67, 105]$. Furthermore, most of the weights are concentrated around $[-60, 59]$, potentially reducing the number of bits needed for representing network weights. Another interesting analysis is the bit's weights distribution. This is computed considering the binary value of each weight, then the distribution is computed as the number of 1s that appear in each position. The result is shown in figure 4.2. The distribution resembles a uniform distribution. It will be important to note that bit 1 is the MSB (and as so, also the sign bit), while bit 8 is the LSB important. It is possible to see that positions 8 and 2 are the most common with a total count of just less than 80, while positions 3 and 6 follow. The least common position is 4 with 70 apparitions. The bits distribution guarantees there is no bias in the results shown below.

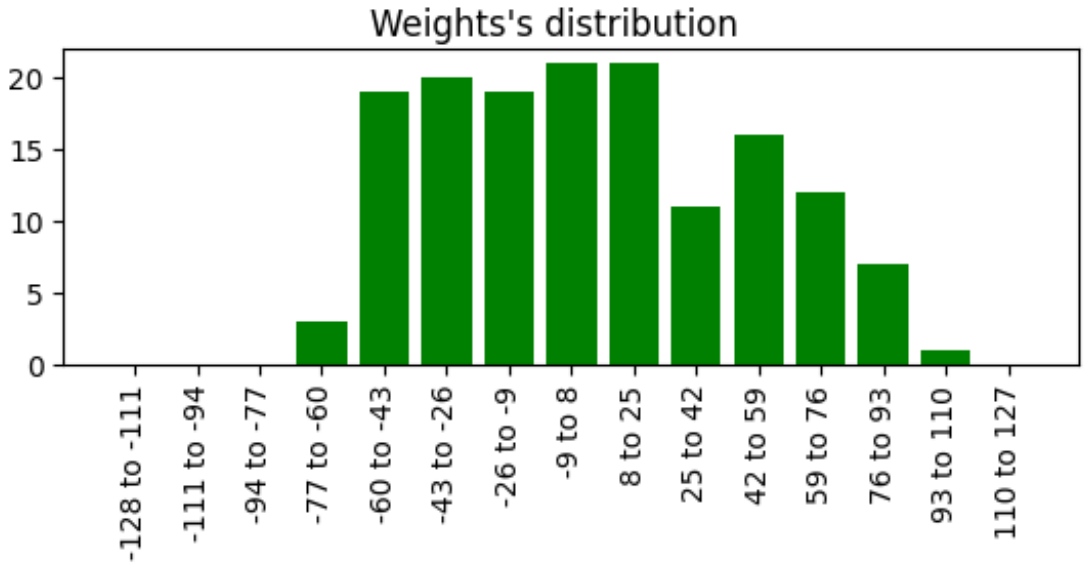


Figure 4.1: Weights distribution

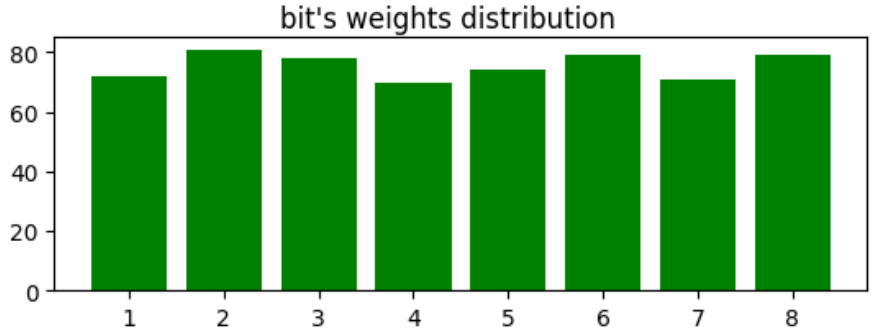


Figure 4.2: Bits weights distribution per position (value 1)

Figures 4.3 and 4.4 show the weights distribution per channel. We can see a general trend which is having a number slightly higher of weights towards the left of the graph. Channel 5 has the narrowest distribution among the six channels. That might be a reason why, as we shall see later, channel 5 has the smallest number of critical faults with respect to the others.

Finally, figure 4.5 shows the distribution of the input stimuli. There are a total of 1'000 stimuli representing digits between 0 and 9. Once again, the distribution is similar to a uniform distribution, which makes the further analysis less prone to errors due to bias in the inputs.

4.2 Faults

All the faults are Stuck-at of the type stuck-at. This type of fault is one of the simplest and it is modeled as a line that always has a fixed value. As written

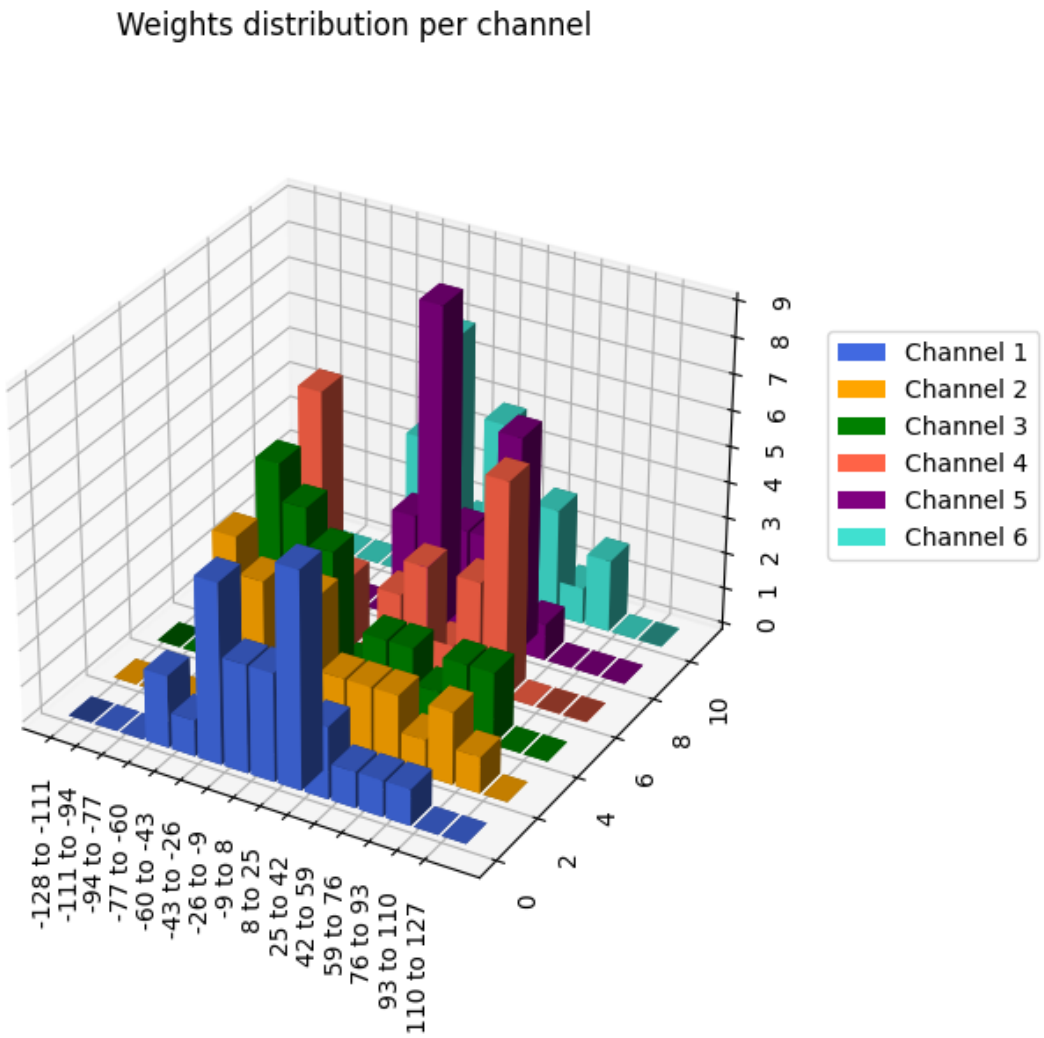


Figure 4.3: Weights distribution per channel

above, three experiments were performed and all of them have similar faults (i.e. the location of the injection and its value).

4.2.1 Fault campaign 1

FC (Fault Campaign) 1 had the goal of investigating the outcome of faults injected in the weight's input of the PEs. A total of 200 faults were randomly generated, choosing the channel, the faulty element (row and column), the bit, and its value (whether to be high or low)

Figure 4.6 shows the number of faults per each row while figure 4.7 shows the distribution of faults per each column of the architecture. This data is used for normalizing the fault distribution of faults per row and column, since the number of injections is relatively small, compared to the number of rows and columns.

Figure 4.8 shows the number of faults for each bit. Each bar represents the number of faults that are injected in that bit position, regardless of the faulty PE

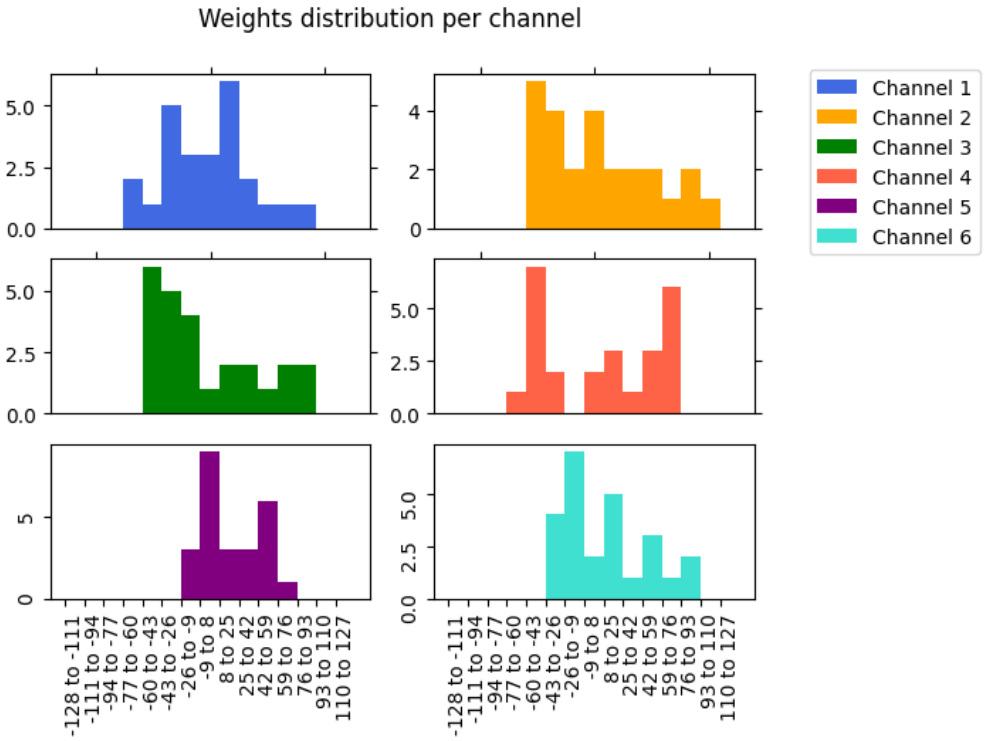


Figure 4.4: Weights distribution per channel - detailed

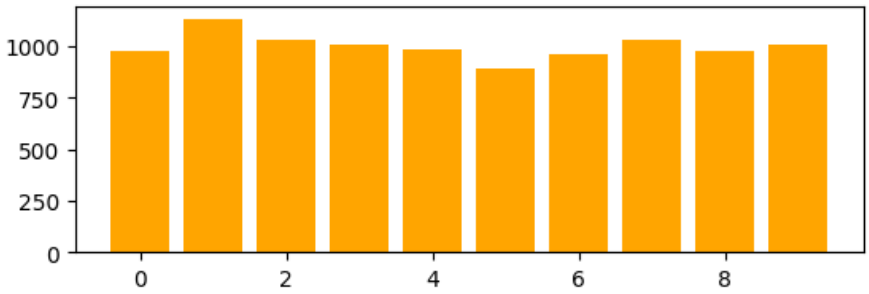


Figure 4.5: Stimuli distribution

or channel.

Finally, figure 4.9 shows the number of fault per channel. A uniform distribution emerges.

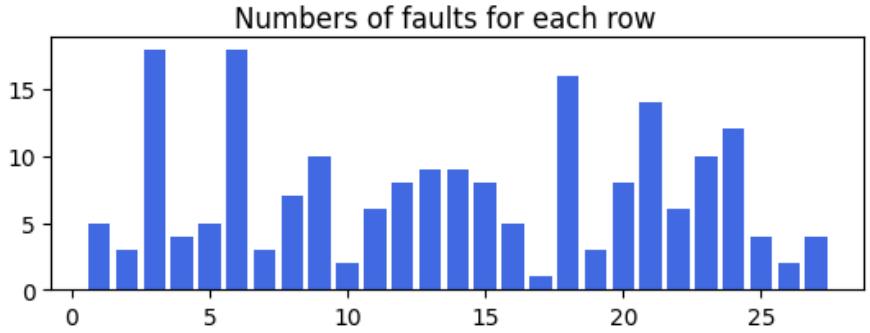


Figure 4.6: Number of faults per each row

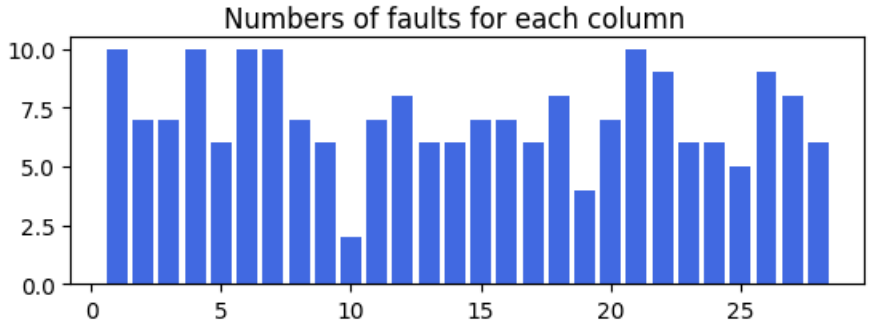


Figure 4.7: Number of faults per each column

4.2.2 Fault campaign 2

In FC 2 the injected lines were the 32-bits output of the PEs. Differently from the first campaign, here the faults were not randomly generated. Instead, for the sake of comparison, the faults were constructed starting from those of FC 1 with the obvious exception of randomly choosing a different bit to inject. The distribution of faults with respect to rows and columns is not much different from that of FC 1, in fact, they should be the same. That is not possible because of the injection point. In FC 1 the *transmission lines* between PEs were injected, since those lines coincide with the output of one PE and the input of the following. Differently, in this context, the injection is directly on the output of the PE. For that reason there is a slight difference between the two, but figures 4.6 and 4.7 completely describe the distribution of faults for FC 2 as well.

Figure 4.10 show a 2D histogram of the injection location combining rows and columns. As said in the previous paragraph, this representation is also valid for FC 1. It is shown that the locations of injection don't have a uniform distribution as it would be ideal. That is because of the limited power the whole experiment was subjected to; indeed the time required to make enough injections with a (much bigger) set of faults satisfying all the uniformity would be long that such an experiment would be out of the scope of this thesis.

The faults distribution over the channels is the same as one of FC 1 and so refer to the previous subsection and specifically to figure 4.9.

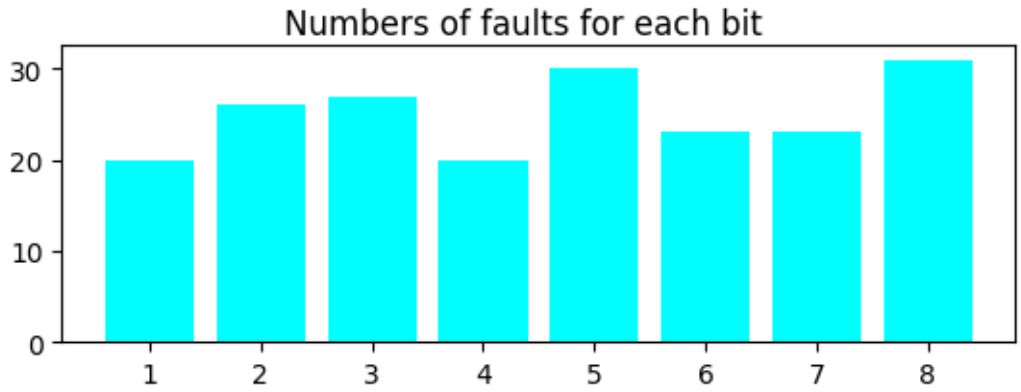


Figure 4.8: Number of faults per injected bit

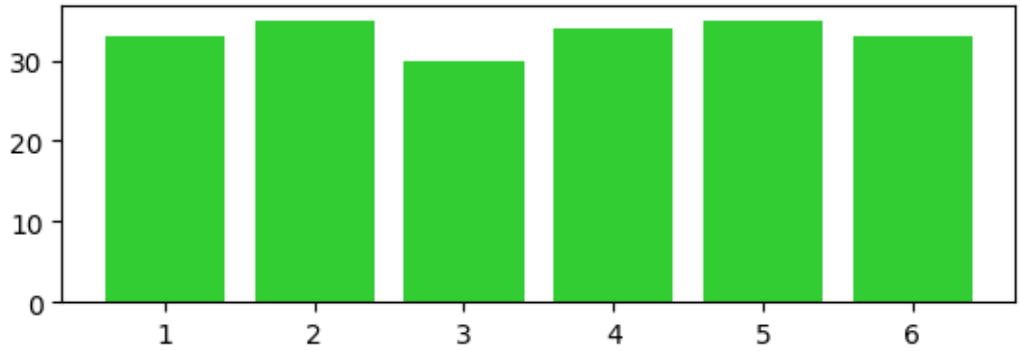


Figure 4.9: Faults per channel

It is worth looking at the fault distribution per bit since in this context there are 32-bits rather than 8. As expected, the distribution is not homogeneous, indeed the number of faults is small compared to the number of possible bits, resulting in a higher number of injections for some bits, like bit 1 or bit 13.

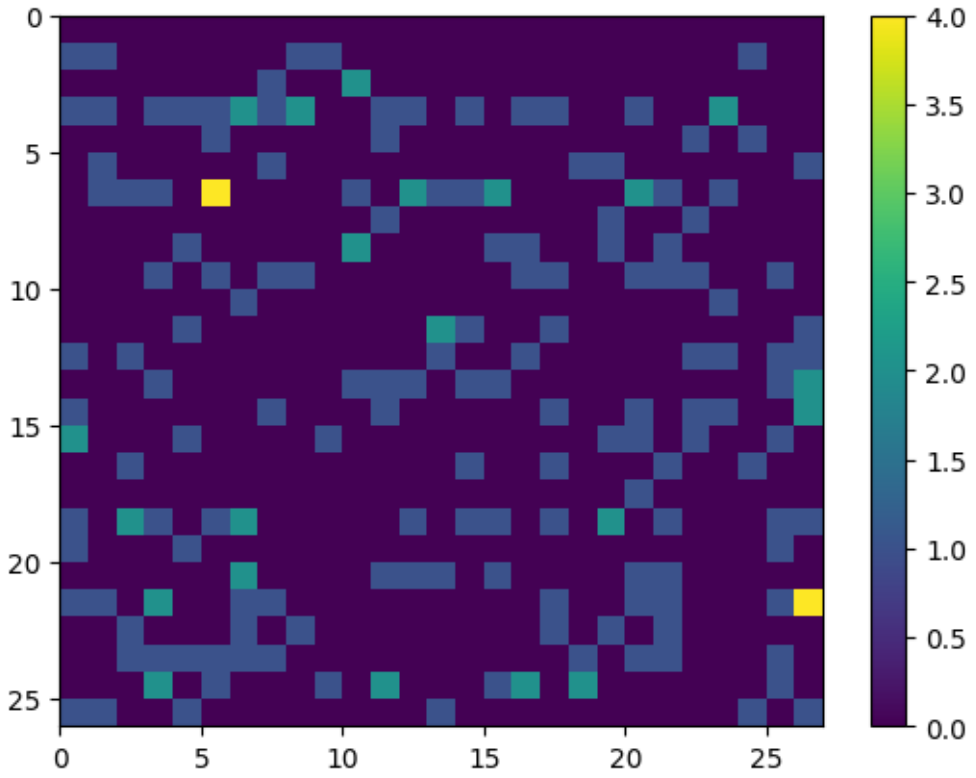


Figure 4.10: 2D histogram describing the number of injections per single location on the systolic array.

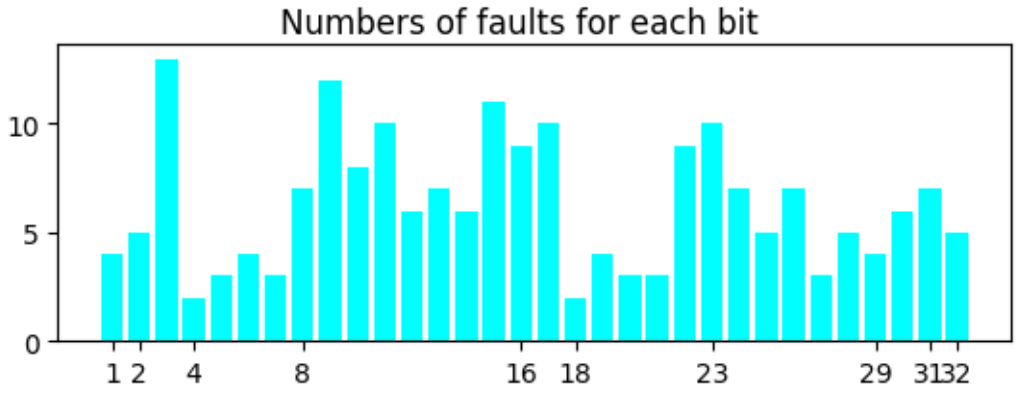


Figure 4.11: Number of faults per each injected bit

4.2.3 Fault campaign 3

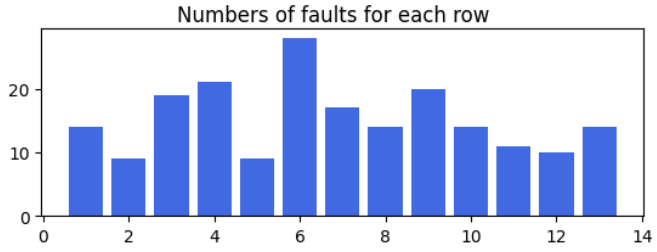
FC 3, as anticipated, is different from the other two. Indeed only *a quarter* of the hardware is present since instead of having a 28×28 array per channel, it is a 14×14 one. This means that also the position of the faults has to be different. The faults were generated starting from those of FC 1, retaining the locations for the channel, bit and value. The injected PE changed according to the following

two rules, given x is the location considered[1]:

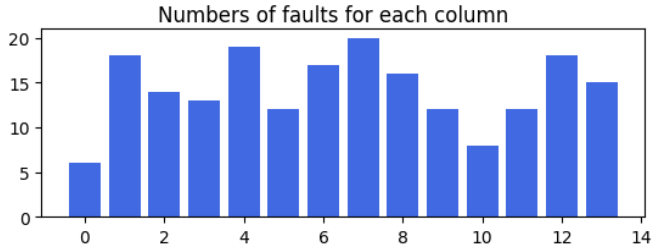
- if $x \mod 14 \neq 0$ then the new location x' would be assigned the value $x' = x \mod 14$
- else $x' =$ random value

The reason to generate a random value lies behind the structure of the VHDL architecture. Since the fault is injected in the transmission lines between two PE, the line 0 would correspond to the external input applied to the systolic array, which is an undesirable situation for the purposes of this thesis.

Figures 4.13, 4.12a and 4.12b show the distribution on rows and columns. Unluckily the random generation was a bit *repetitive* to the point of having 7 injections on the same PE; that is unavoidable with the number of this thesis, the injections were too few compared to the other parameters of the experiments. Nevertheless, they will serve as a good approximation.



(a) Faults per row



(b) Faults per column

Figure 4.12: Figures

[1]In this context x can be both the value of the row and the value of the column.

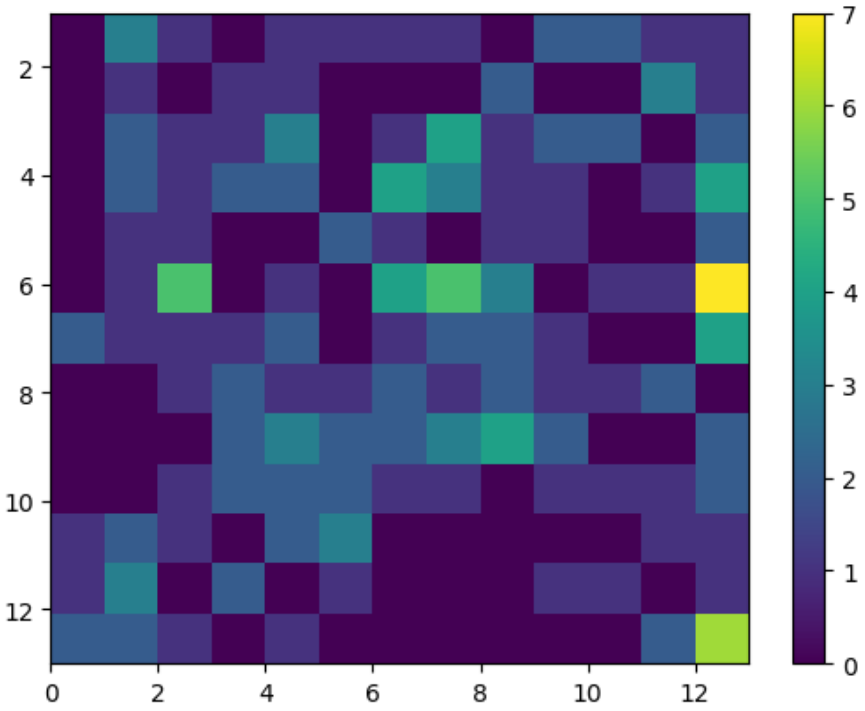


Figure 4.13: 2D histogram describing the number of injections per single location on the systolic array

4.3 Metrics

In this section, the actual results of the experiment will be highlighted.

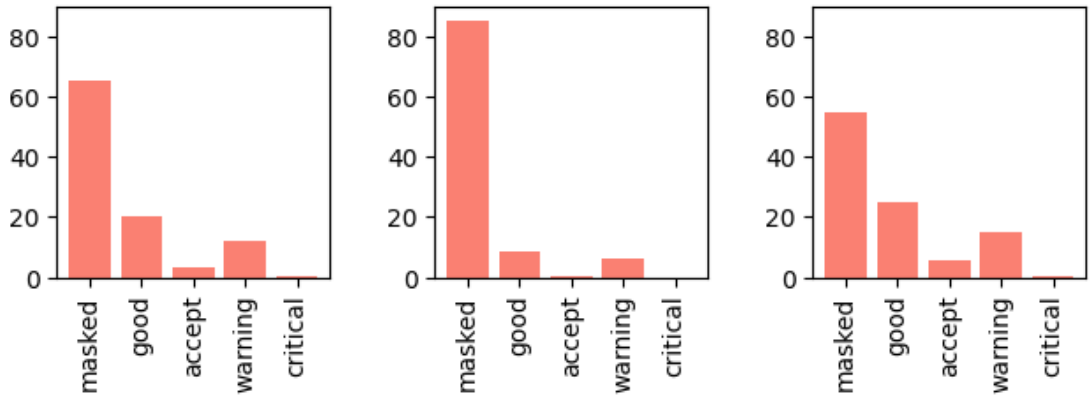
The faults are categorized following the same criteria of [5]. First of all, if the output of the network of the faulty execution is the same as the output of the golden execution of the network, the fault is classified as **masked**. Otherwise, the fault is classified as **observed** and there are four sub-categories:

- **Good**: the top-ranked element is the same as the gold execution one and the confidence of the network is higher than the golden execution;
- **Accept**: the top-ranked element is the same as the gold execution one and the confidence is smaller within 5
- **Warning**: the top-ranked element is the same as the gold execution one and the confidence is smaller and greater than 5
- **Critical**: the top-ranked element is different.

The architecture performed quite well. Figure 4.14 shows the performance of the three FC. As expected, the most critical case is FC 3. That happens because the same faulty PE process the image four times in four different places. On the other hand, FC 2 performed the best, because the faults were injected in the output of the PEs before the activation of the node. This means that about half of the bits

change the resulting value in tiny amounts. The last 9 bits (from bits 24 to 32) are completely ignored, while bits 16 to 23 constitute the output after the activation. Bits 2 to 22 saturate the value. Indeed, the activation node checks whether the value made by bits 2 to 23 "fits" inside an 8-bit register. If it is bigger, the value is saturated to `0xFFFF`. Finally, if bit 1 (the MSB and sign bit) is set, the output gets a value of 0.

Another way to see the data is to understand that a fault in the output of a PE is **propageted** when the fault bit is multiplied with the other input, while a fault in the output stays as it is. In other words: **the faults of FC 2 are a subset of those seen in FC 1**.



(a) FC 1 **(b)** FC 2 **(c)** FC 3

Figure 4.14: [%]Safety performance for the three FC. The graphs show the percentage of injections for the different categories.

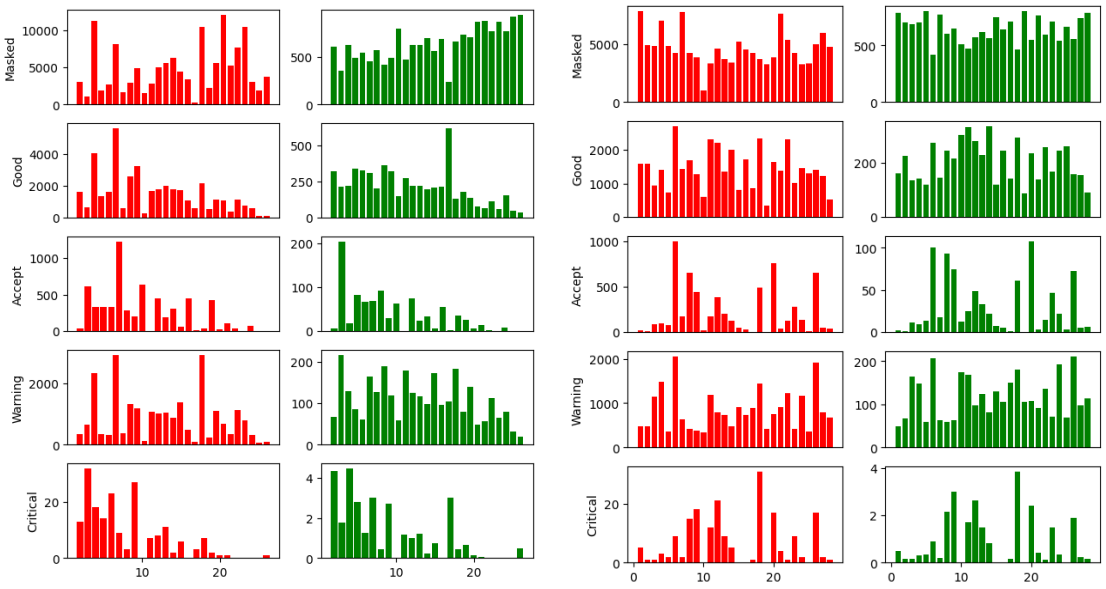
The results for each FC are now presented.

4.3.1 Fault campaign 1

Processing element position

The position of the injected PE is very important. Something that was not specified before is that the fault is obviously propagated throughout the line of the injected PE. In facts, the input on `west` (or `north`) is copied (whether faulty or not) to the `east` (or `south`). It is clear from figure 4.15a that the deeper the row, the smaller the probability of having an unsafe fault. That happens because the fault is injected in the vertical lines of the PE (i.e. on `north` and `south`, allowing the fault to be propagated in that direction.

Figure 4.7 shows a completely different trend. The fault number is about constant and does not show any particular correlation with the faults. Obviously, that is due to the path taken by the weights on the architecture. Looking at figure 2.3, we can see the weights enter in c_i and then for each cycle proceed vertically. The stimulus, instead, proceeds horizontally and that is why we see no correlation in figure 4.7

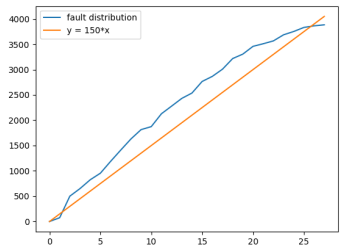


(a) Safety per row

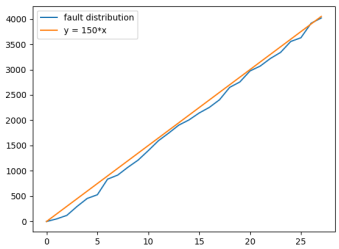
(b) Safety per column

Figure 4.15: Number of faults per each row and column. On the left is the plain number (in red), and on the right, the number is normalized with respect to the number of injections.

Back at figure 4.6, we can see that most of the critical faults are located before row 10. And in general, the number of unsafe faults (accept, warning and critical) decreases with the depth of the row. Figure 4.16a show the cumulative distribution. We can see the distribution is a little bit more than linear. Figure 4.16 shows the cumulative distribution for both the types. Note how much more linear the distribution is for the columns.



(a) Cumulative number of unsafe faults per row normalized with respect to the number of injections.



(b) Cumulative number of unsafe faults per row normalized with respect to the number of injections.

Figure 4.16: Cumulative number of unsafe faults per injected row (on the right) and column (on the left). The orange lines represent a linear approximation of the data

Channel

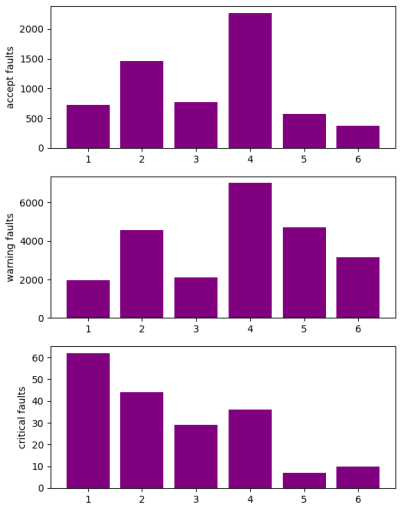
The term channel, as explained before, correspond to the third dimension of the layer. Data show a clear relationship between the safety of the channel and the injected channel. The following table shows the number of faults per each type per channel. Part of the data is reported in the next figures. Figure 4.17b shows clearly that the number of injections on the fourth channel produced a higher number of unsafe faults and it is prevalent also between accept and warning fault types.

Masked faults	Channel 1	17888
	Channel 2	22113
	Channel 3	21277
	Channel 4	19163
	Channel 5	25106
	Channel 6	24234
Good faults	Channel 1	12337
	Channel 2	6809
	Channel 3	5801
	Channel 4	5516
	Channel 5	4584
	Channel 6	5230
Accept faults	Channel 1	729
	Channel 2	1462
	Channel 3	773
	Channel 4	2271
	Channel 5	575
	Channel 6	378

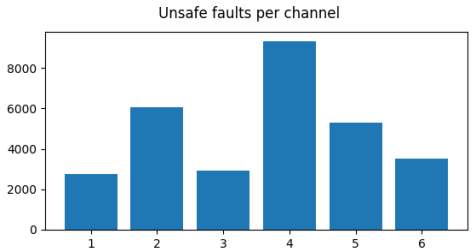
Warning faults	Channel 1	1984
	Channel 2	4572
	Channel 3	2120
	Channel 4	7015
	Channel 5	4728
	Channel 6	3150
Critical faults	Channel 1	62
	Channel 2	44
	Channel 3	29
	Channel 4	36
	Channel 5	7
	Channel 6	10

Furthermore, as figure 4.17a that most of those faults are of accept and warning type. On the other hand, the first channel is responsible for most of the critical faults. This aspect should be investigated since this last result could be biased by the total number of critical faults which is quite small. Another observation that might lead to the biased hypothesis comes from the fact that accumulating the unsafe faults (figure 4.17b) the faults on channel 1 are the least.

Finally, another little observation arises: channel 1 is responsible for 30% of the total number of good faults (which increase the top-rank probability). It might be interesting to how this channel is responsible for the better performance of the network.



(a) Number of different faults per injected channel.



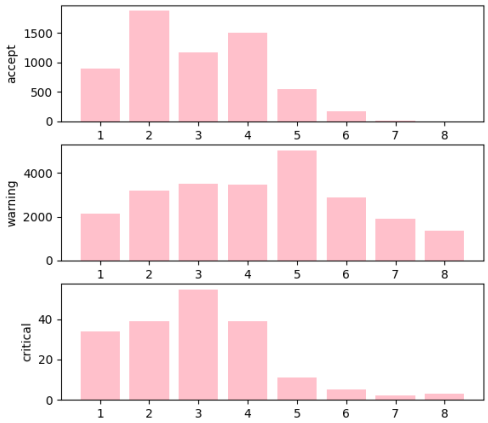
(b) Number of unsafe faults (i.e. aggregating accept, warning and critical faults) per injected channel.

Figure 4.17: faults distribution per channel

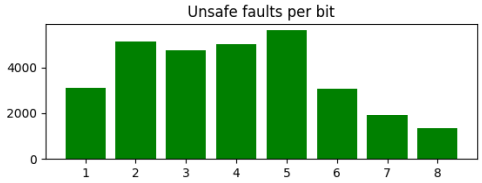
Bit position

The bit position is obviously much more important. Indeed, the position of the bit might make a huge change in the value of the weight. For example, if the original value is -23, in binary `0xE9`, when injecting bit 1 with a stuck-at 0, the new value would be `0x69` which correspond to 105. Not only does the sign change but also the absolute value is much different.

Figure 4.18a show the distribution of the types of fault per each bit position. As expected, the more important the bit position, the worse the fault is. Indeed, most of the critical faults come from bits 1 to 4 which are the most important. Also, the accept faults come from the higher bits. In general, the higher the bit position, the more significant the difference in value.



(a) Number of faults per each injected bit position



(b) Number of faults per each injected bit position

Figure 4.18b shows the number of unsafe faults, aggregating the three types. Overall the last three bits are the less critical. It is important to underline that for this FC, the injected values are the input to the PE, so the faulty value is propagated through the multiplication inside the PE.

Stimulus

The analysis of a possible relationship between the stimulus and the fault types is much unsatisfying because there is not a clear pattern. What can be said is that there seem to be stimuli which are more susceptible to injections than others. A deeper analysis should be made, but in general (figure 4.19) stimuli representing 9 seem to be much more susceptible to critical faults. Indeed there is a clear spike on that label. Otherwise, stimuli representing 1, 7 and 9 are the more prominent for accepting faults. On the other hand, there is no statistical difference in warning faults.

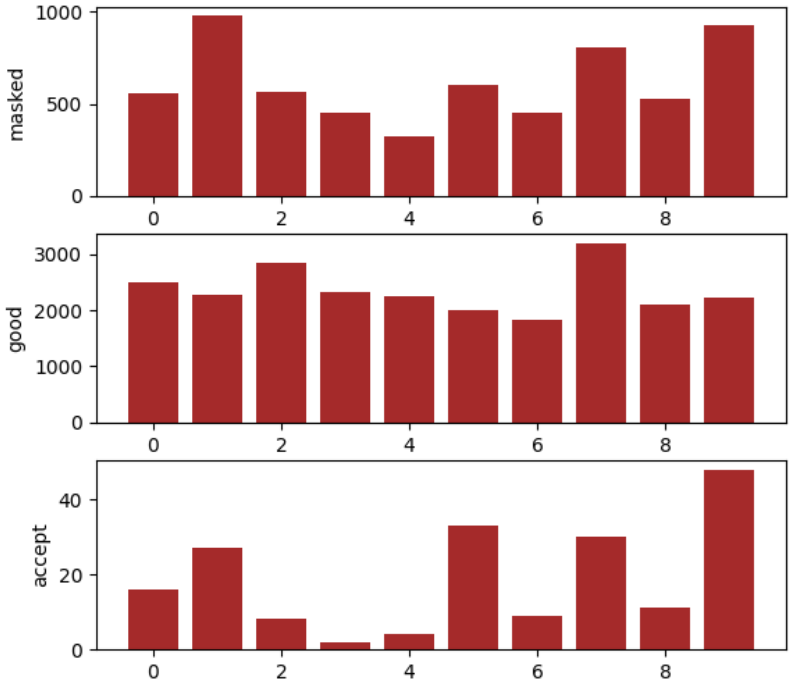


Figure 4.19: Number of faults per each stimulus

Fault value

Figure 4.20 shows the number of faults per fault value. We see that the stuck-at 1 faults are more severe than the stuck-at 0. Indeed more than 75% of the faults are stuck-at 1. This fact might depend on the location of the faults. Indeed it is possible that most of these faults are injected in the most significant bits of the input, thus completely changing the weights' values.

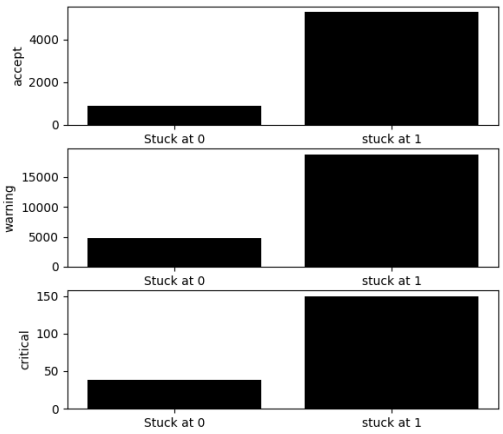


Figure 4.20: Number of faults per fault value (i.e. whether they are stuck-at 0 or stuck-at 1)

Finally, figure 4.21 shows the ratio of stuck-at 1 faults to both types. Bits 2 to 8 show the aforementioned trend. On the other hand, it is interesting to note that, although more than 75% of the faults are stuck-at 1, most of the faults in bit 1 (the most important) are stuck-at 0. Indeed, less than 1/3 of the faults are stuck-at 1.

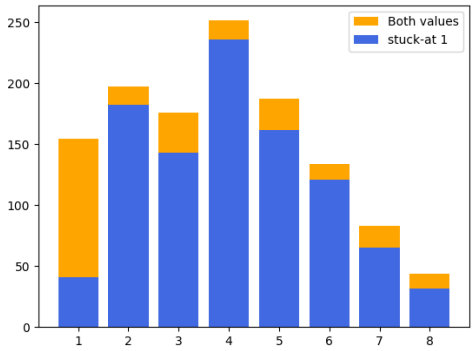
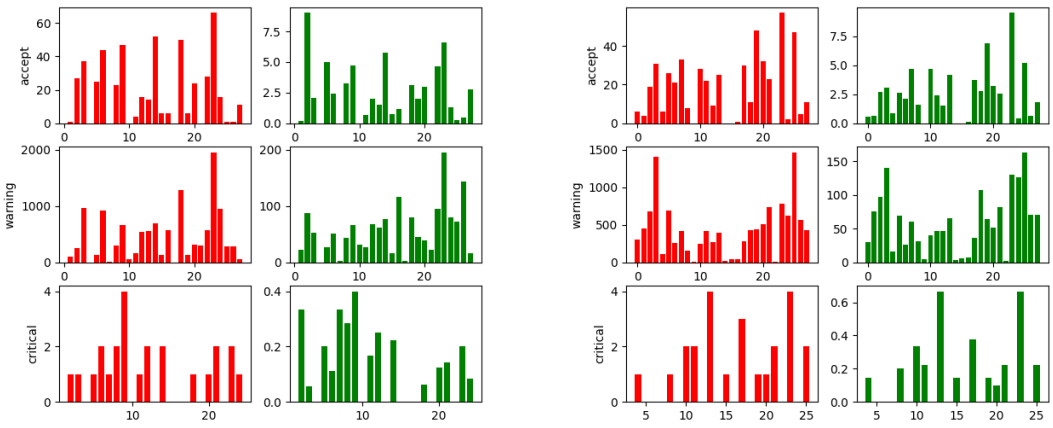


Figure 4.21: Number of unsafe faults comparison between stuck-at 1 and stuck-at 0. In yellow is the number of unsafe faults normalized per bit injection. In blue is the number of unsafe faults generated by a stuck-at 1.

4.3.2 Fault Campaign 2

Processing element position

Differently than FC 1, in this case, the injections are on the result lines of the PEs. This means that the faults are not propagated, indeed looking at figures 4.22 we can see there is not a clear trend or any sort of pattern. The number of faults, in this context, is a random variable.



(a) Number of faults per row

(b) Number of faults per column

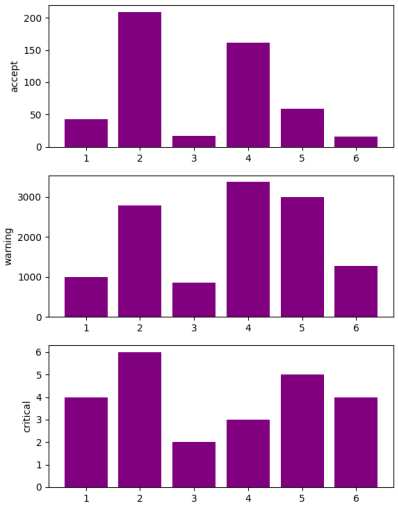
Figure 4.22: Number of faults per each row and column. On the left is the plain number (in red), and on the right, the number is normalized with respect to the number of injections.

Channel

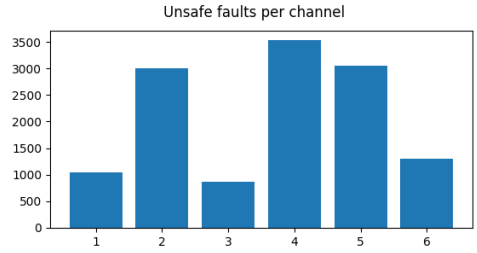
The injected channel, as seen in the previous section, is strongly related to the fault type. Indeed, figure 4.23b shows the same pattern as before: the fourth channel is the most critical. Different story when looking at figure 4.23a. Differently from FC 1, channel 2 has an important role this time.

Bit position

In this FC there are 32 bits to inject, corresponding to the output of a PE. The injection is performed before the activation function (in this case a ReLU). Figure 4.24a this fact. Indeed for any type of fault, bits 24 to 30 never participate in a non-masked fault. That is true because this specific implementation of the ReLU just ignores the last 9 bits. On the other hand, the first half of the bits are the most critical: when any bit between 1 and 15 is set, the output of the ReLU gives `0x7F` (or 127), saturating the output. This means that any fault in one of those positions will insert into the network a weight outside the range (refer to figure 4.1).



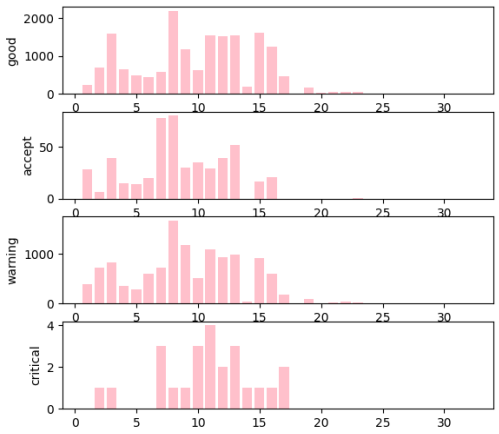
(a) Number of different faults per injected channel.



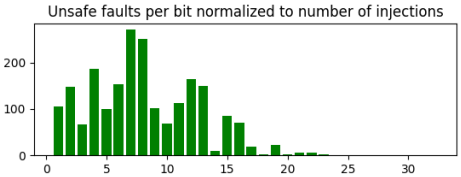
(b) Number of unsafe faults (i.e. aggregating accept, warning and critical faults) per injected channel.

Figure 4.23: Faults distribution per channel

Bit 16 correspond to the sign of the result and the remaining bits (17 to 23) compose the result otherwise. Figure 4.24b shows the number of unsafe faults per bit position normalized with respect to the number of injections. It is clear that the critical bits are those which saturate the output.



(a) Number of faults per bit position



(b) Number of unsafe faults per bit position normalized to the number of injections

Stimulus

Similarly to what we saw in FC 1, some stimuli are more prone to criticality than others. Figure 4.25 shows, analogously as FC 1, that stimuli representing 9 are more frequent than others when talking about critical faults. With respect to the other types, there is not much to say, stimuli 1 and 7 are a little bit more frequent, but there are no spikes as seen before.

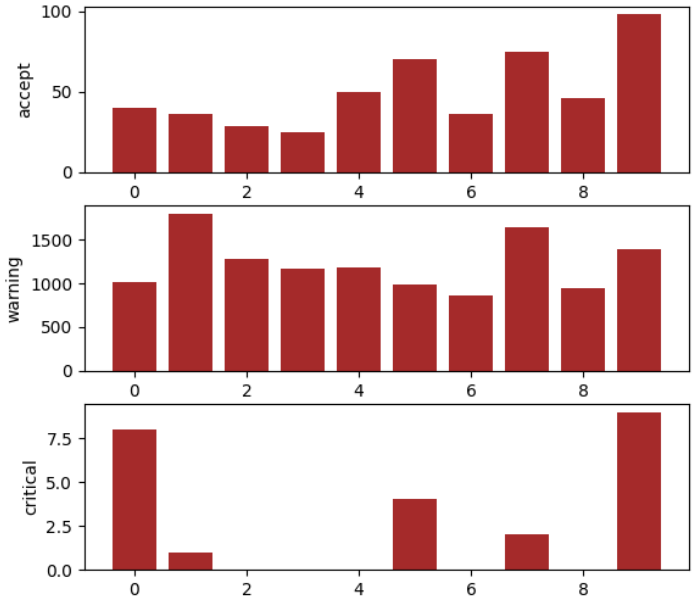


Figure 4.25: Number of faults per stimulus class

Fault value

For this fault campaign, the stuck-at 1 injections are responsible for almost 100% of the unmasked faults. This datum was more than expected, after understanding that half of the bits can saturate the output. Figure 4.26 shows the number of faults with respect to the fault value. Figure 4.27 shows the number of faults comparing the stuck-at 1 with the total. The only bit which makes an exception (as seen in FC 1) is bit 1. That might be due to the fact that the activation function outputs 0 if the 32-bit value is negative. A fault stuck-at 0 fault in that bit might change the value from negative to positive.

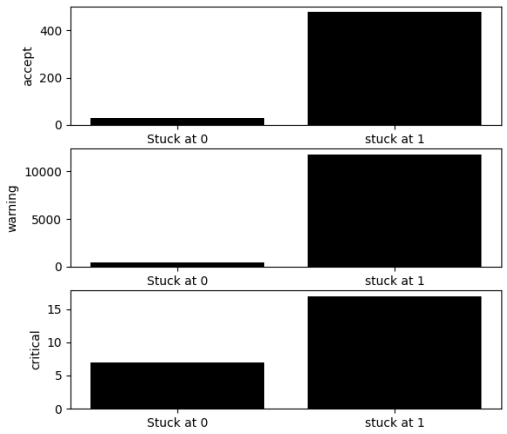


Figure 4.26: Number of faults per fault value

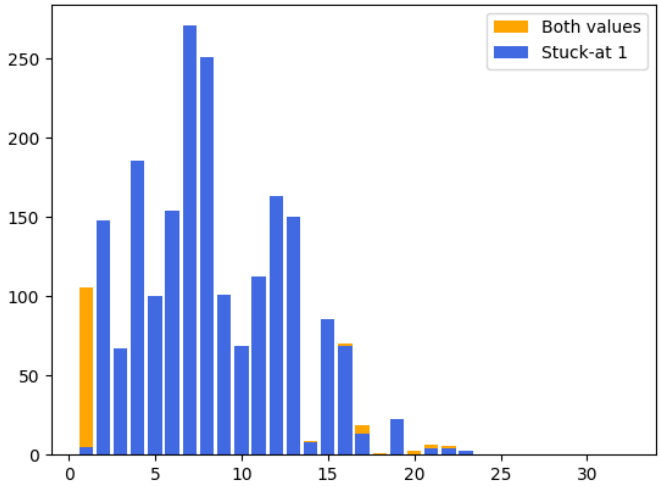


Figure 4.27: Number of unsafe faults comparison between stuck-at 1 and stuck-at 0. In yellow is the number of unsafe faults normalized per bit injection. In blue is the number of unsafe faults generated by a stuck-at 1

4.3.3 Fault campaign 3

Processing element position

The faults are in the same positions of FC 1, modified with the rules explained above. For this reason, the general trend of the number of faults per row and per column is the same as FC 1. This means that the faults per column have no clear trend. Figure 4.28 shows the pattern. Exactly as seen for FC 1, the number of faults decreases when the depth increases. The effect is surely less dramatic because there are half the rows than before.

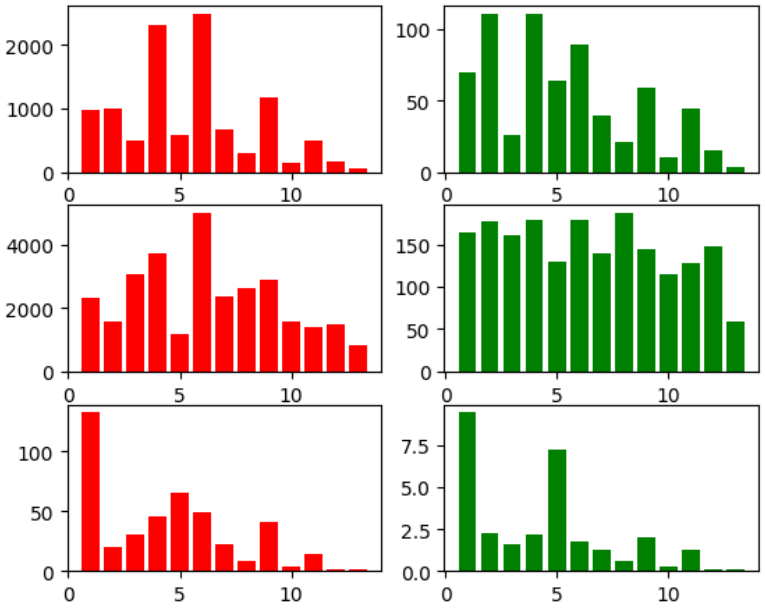
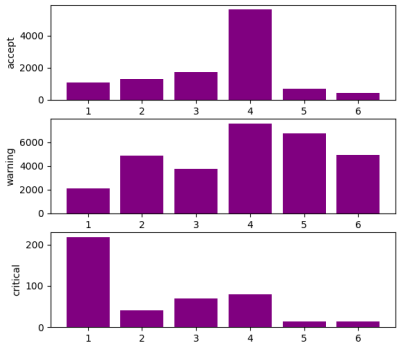


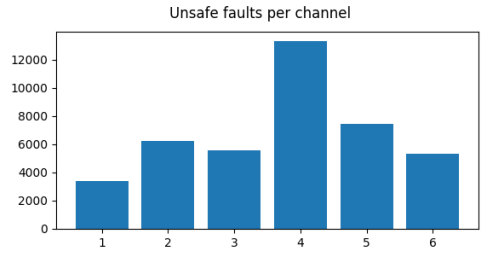
Figure 4.28: Number of faults per row

Channel

This experiment confirms what was already established in the previous sections. Channel 4 is indeed the most susceptible of the 6. Furthermore, most of the critical faults come from channel 1. Once again this result might be misleading, in this case, the reason lies behind the injection positions. Indeed, they are (almost) exactly the same as for FC 1, and for that reason might lead to the same result.



(a) Number of different faults per injected channel.



(b) Number of unsafe faults (i.e. aggregating accept, warning and critical faults) per injected channel.

Figure 4.29: Faults distribution per channel

Bit position

The results of this analysis are predictable too. The higher the bits the more critical the fault. Also, the critical faults number decreases exponentially with the bit position, indicating that a change in those bits is most likely to produce a critical error. Bits 5 and 2 are prominent for accept and warning faults.

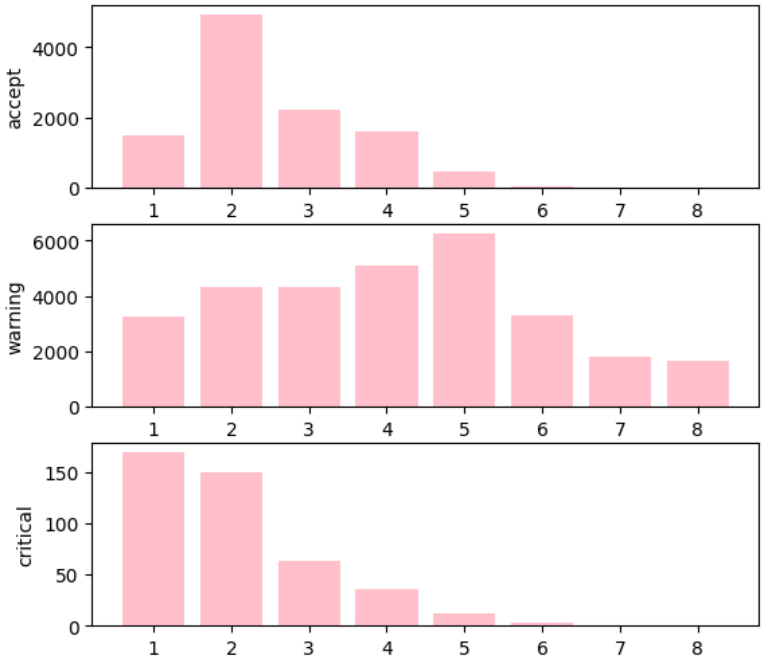


Figure 4.30: Number of faults per bit

Stimulus

The number of faults per stimulus is a little different than the other campaigns. Indeed, we can see that in this case, the critical input is the stimulus representing 2. In all three cases, these stimuli have a significant number of faults, while the stimuli that in the other fault campaigns were most susceptible are not that important.

Fault value

In this context, the stuck-at 1 faults have a role less important than previously. Indeed, even though most of the faults are still stuck-at 1, there is an important increase in the other type. Figures 4.33 and 4.32 demonstrate the concept.

In general, stuck-at 1 faults are more critical than stuck-at 0. As seen with the channel, it is clear that the different locations are more susceptible to faults than others. To better investigate this aspect (and generalize it) it would be interesting to change the implemented network, either retraining the network, thus changing the distribution of the weights, or using another network altogether.

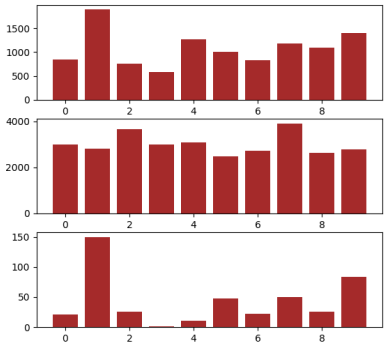


Figure 4.31: Number of faults per stimulus

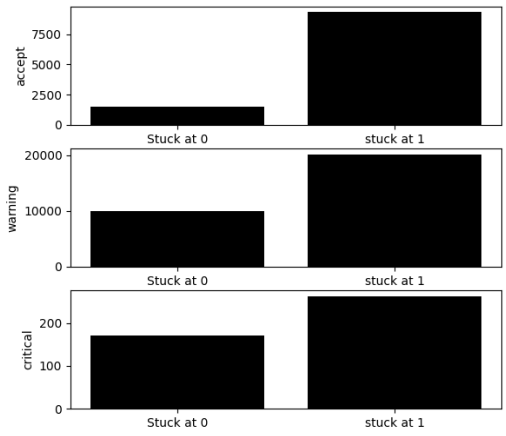


Figure 4.32: Number of faults per fault value

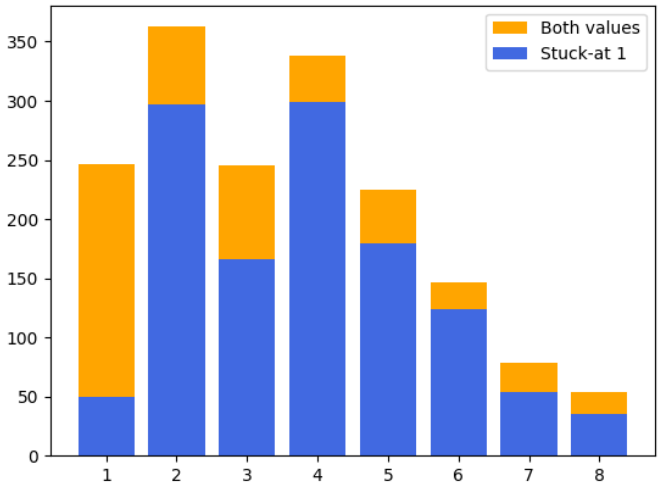


Figure 4.33: Number of unsafe faults comparison between stuck-at 1 and stuck-at 0. In yellow is the number of unsafe faults normalized per bit injection. In blue is the number of unsafe faults generated by a stuck-at 1

Chapter 5

Conclusions

In this thesis we discussed the mapping of a convolutional layer to a systolic array, creating a neural accelerator, and later analyzed the reliability of the architecture injecting permanent faults.

The result obtained shows that a reliable neural accelerator is in fact possible without *reinventing the wheel*, that is using a well-known architecture such as a systolic array. The gathered experiments showed that the architecture seems to be safe, on average, in 85% of the total injections, masking the 68%. Furthermore, the critical faults were only a minimum percentage of 0.22% in the worst case.

Another interesting result shows how the channels do not have the same criticality, indeed channel 4 is the most problematic, while on the other hand, the safest is channel 1. This result should be more investigated since it might depend on the training of the network. Interestingly, we saw that almost 75% of the unsafe faults were caused by stuck-at 1 faults.

The reliability of the network, as seen in other studies, heavily depends on the injected bit. The results show that the more important the injected bit, the higher the probability of unsafe fault. This is true with some exceptions. Indeed it was not always the MSB the most critical, but the most important first half in general.

Note that the injected layer was the first which is the most sensible (as shown by [5]) layer among all.

In conclusion, the result obtained is in line with the expectation and the results seen in the literature. Nevertheless, more research has to be done to better assess the weaknesses of neural networks in general. In future works, it would be interesting to inject faults in the stimuli lines of the systolic array and to study a good trade-off between chip area and reliability.

Appendix A

Failed attempt

Given the structure of the systolic array, as explained in section 2.3, it would have been efficient to exploit the horizontal overlapping region using additional hardware and indeed such an attempt was made, trying to dynamically generate the input stimuli sequences. A piece of hardware, called **selective shift register**, was created and indeed it can generate the correct input sequences using less memory for storing the input sequences.

This chapter will first be explained the general idea and then a working implementation will be shown.

A.1 Horizontal overlapping region

As shown in figure 2.9b (shown again below) each sequence share $j - stride$ values with the horizontal-next sequence. This means it in theory is possible to forward those common values from one PE to the next (vertically-next). The new non-shared values must come from the outside though, for generating the complete sequence. In each "input cycle" (i.e. at the beginning of each new column, given the input is ordered as shown in the figure with yellow and green lines) a new element is injected in the sequence. To be clear, the number of elements injected for creating the new sequence is dependent on the *stride* parameter. Indeed, since $j - stride$ values are shared, the other *stride* values need to be injected as explained above.

A.2 Working hardware implementation

For achieving the behavior explained in the previous section, a quite complex system was created. Figure A.2 shows such a system. It is a shift register, but each cell can get its data directly from the outside, through the multiplexer if the `fire` signal (depicted in blue) goes high. The fire signal is forwarded each clock cycle, enabling the subsequent cells to get data from the outside. Finally, the fire signal has to be synchronized with the rest of the architecture for it to be functional. Indeed, the implementation is thought using a counter and a digital comparator such that the counter resets each `j clock cycles` and the comparator fire when

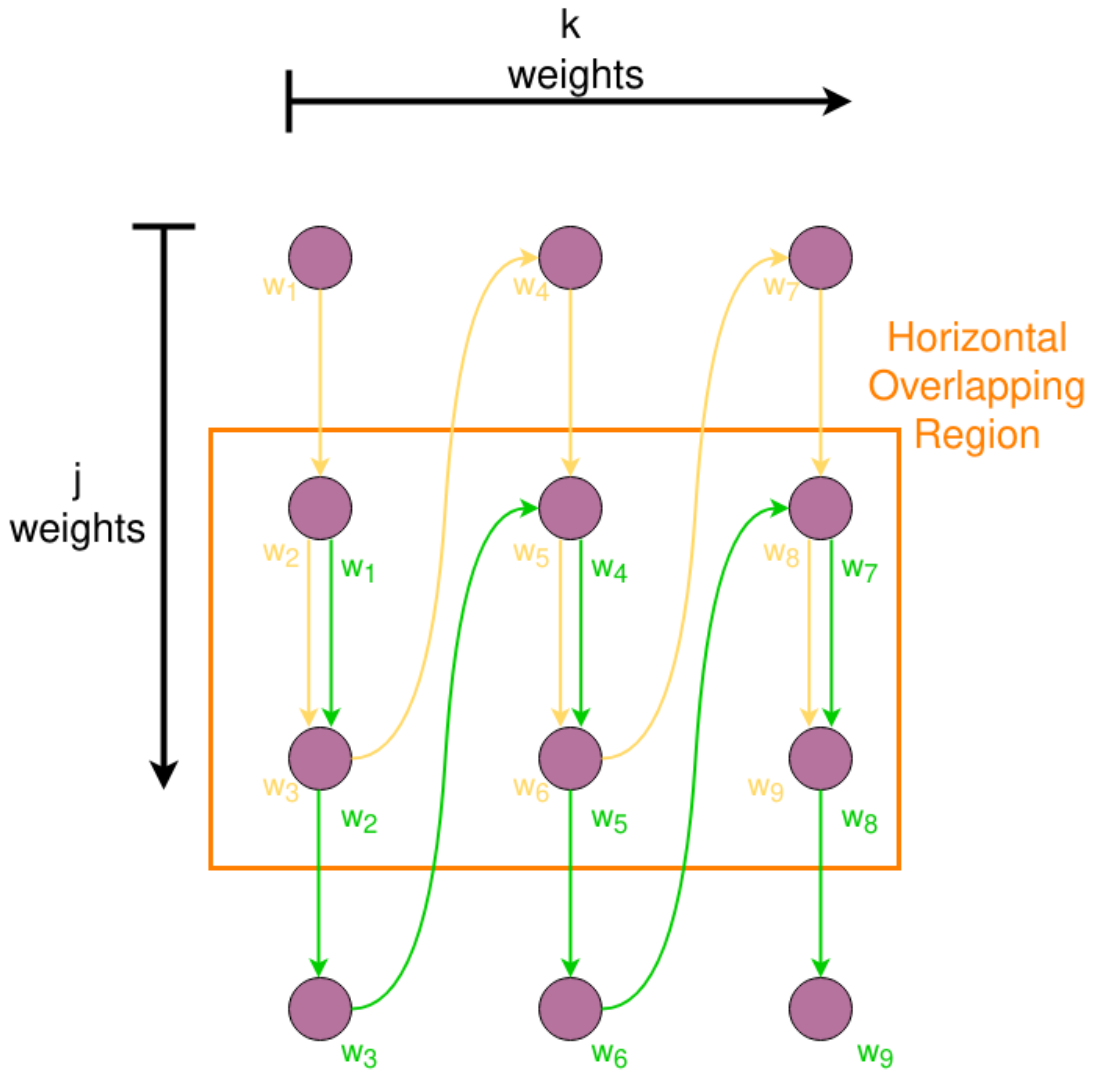


Figure A.1: Horizontal overlapping region

the value is equal to `j`.

Figure A.3 shows a questasim simulation of the hardware. The fire signal goes high each 3 clock cycles. The signal `input`, corresponds to the outside input which is not shared between sequences while the signal `output` shows the output of each cell of the register. The signal `fire_line` shows when the fire goes high for each multiplexer allowing the corresponding `input` to get inside the register.

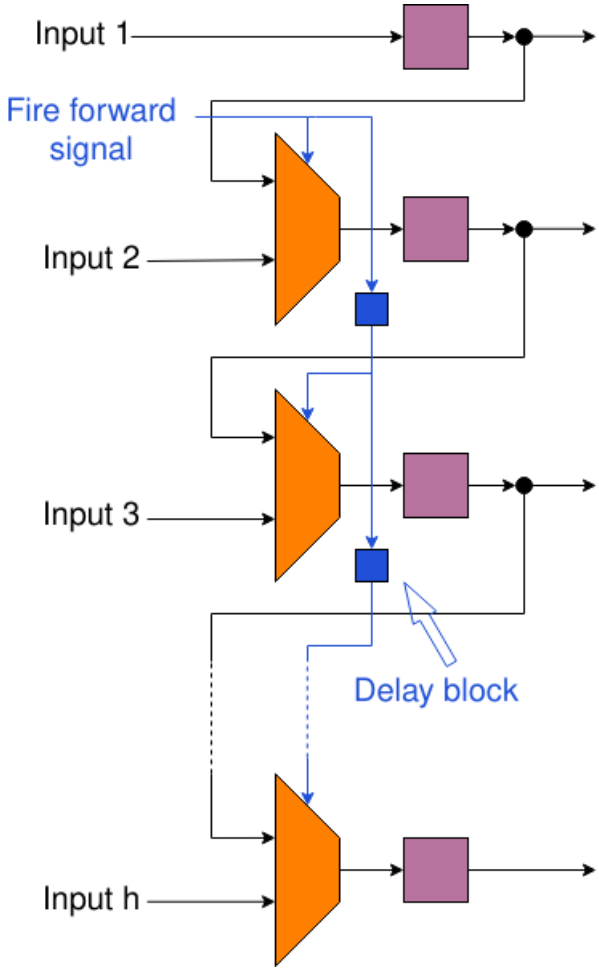


Figure A.2: Selective shift register (SSR)

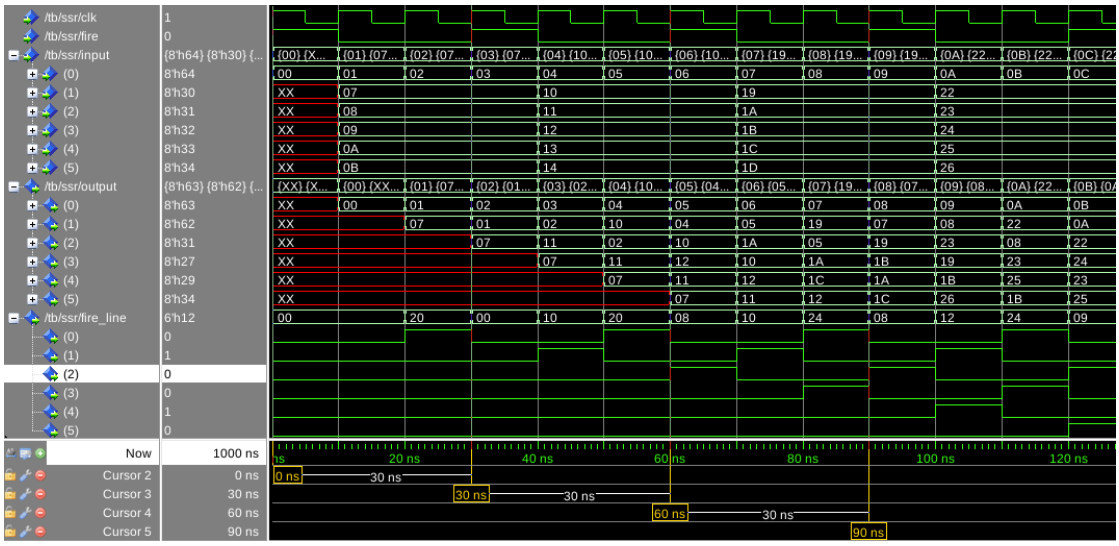


Figure A.3: Selective shift register simulation

Appendix B

Fully connected layer implementation

The systolic array architecture has the potential to be exploited also for fully connected layers, other than for convolutional layers. This means that a network such as LeNet could be (almost[1]) fully implemented in hardware with systolic arrays.

The mapping consists of using only one portion of the systolic array. More precisely, it is enough to use a *line* of subsequent PEs, each one for one output of the layer. Indeed, the main observation to be done is that the inputs are all the same among all inputs and so they can be using a similar methodology as the one explained in section 2.3. Figure B.1 shows the representation of a fully connected layer. It is possible to see that each input is used n_o times (where n_o represents the number of outputs of the layer), each time with different weights. Note that a bias has to be present and it can be conceptualized as an input neuron that has constant input 1. This idea is highlighted in figure B.2, which shows how the architecture can be adapted to this type of layer.

As said, the weights are different for each input, so it is not possible to use the `j-shift register` we explained above. That is a problem that might be solved using a binary multiplexer. The stimulus inputs are instead equal for every neuron. This means that it is possible to exploit the forwarding nature of the PEs, enabling thus the use of the first row only.

[1]it should be studied how to map different layers such as max-pooling layers with systolic arrays.

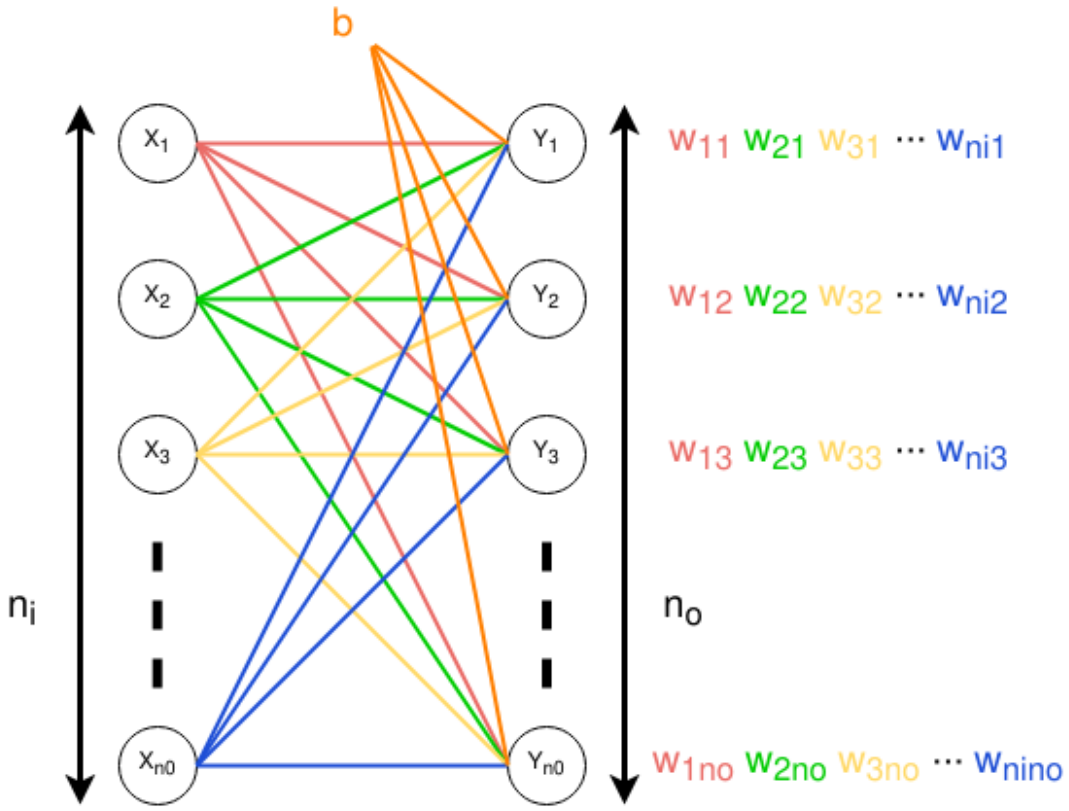


Figure B.1: Fully connected layer sketch

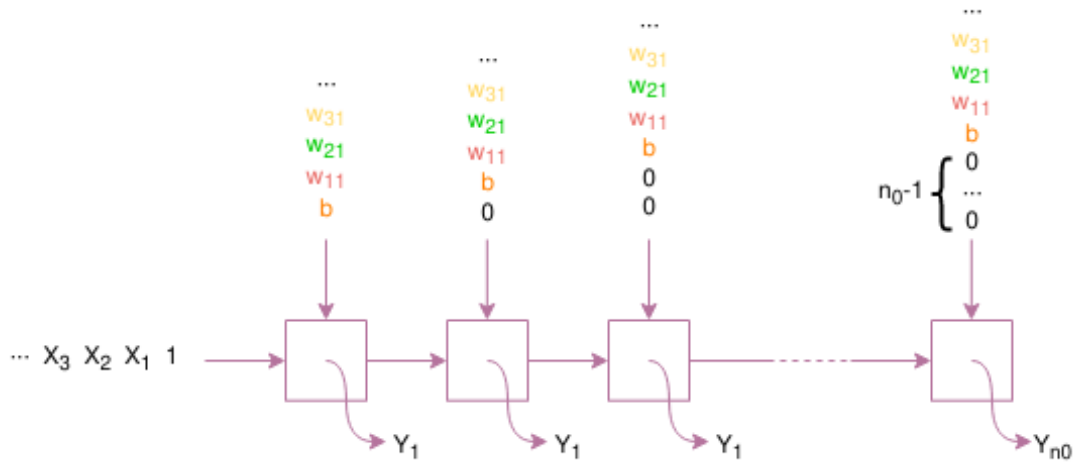


Figure B.2: Fully connected layer implementation with systolic array

Appendix C

Fault identification

The systolic array architecture has an interesting and very useful property. It is a *diagnostic* property, meaning that it can be used to diagnose a malfunction and in this case, with the systolic array it is possible to diagnose any stuck-at fault between the transmission lines of the PEs, provided those are the only fault that can happen.

Indeed, the reason behind this is extremely simple: when multiplying by 0 the result has to be 0. Exploiting this postulate it is possible to not only find a fault but also locate it! This is extremely useful for, as an example, considering a backup system for the architecture.

Assuming that $\mathbb{1}$ identifies the matrix 1×1^T (where 1 is a vector whose components are all 1), $\mathbf{0}$ is a null matrix and the first matrix is input from the west, then the locating procedure is as follows:

1. Input $\mathbb{1} \times \mathbf{0}$ in the array,
2. Compare the result with the expected,
3. if they do not coincide, a fault in the vertical lines can be located,
4. Input $\mathbf{0} \times \mathbb{1}$ in the array,
5. Compare the result with the expected,
6. if they do not coincide, a fault in the horizontal lines can be located.

In the context of the convolution, the input sequences are `all-zeros` or `all-ones`. More in detail, when identifying faults in the stimulus lines, the stimulus sequences are `all-ones` and the weights sequences are `all-zeros`. When identifying the faults in the weights lines are the opposite.

When identifying a fault, it is enough to compare the obtained result with 0. Indeed, if any of the bits of the result is non-zero, then a fault had happened for sure.

For locating the fault, we have to make a couple of observations more. Firstly, we need to underline that the fault is propagated. This means that once the fault has been injected, it will be also present on the other side of the PE. If, for example,

bit 0 of `WEST` of P_{34} is stuck at 0, then also bit 0 of `EAST` will be 0 and thus also `EAST` of P_{35}, being the same line, and so on. This means that identifying the *first* element to have non-zero bits, will correspond to the fault location. We need to clarify that *first* is related to the depth as explained in 2.3, indeed in this case we need to consider also the orientation (i.e. whether we are searching a fault horizontally or vertically). Lastly, the faulty bit directly impacts the result, since its value depends on the inputs which are well known. Specifically, the output is supposed to have all the bits at 0 and the position of any non-zero bit clearly indicates the position of the fault bit.

Acronyms

AI
artificial intelligence

MAC
multiply-accumulate

ReLU
Rectified Linear Unit

PE
Processing Element

MSB
Most Significant Bit

LSB
Least Significant Bit

FC
Fault Campaign

CNN
Convolutional Neural Network

DNN
Deep Neural Network

NN
Neural Network

Bibliography

[1] Annachiara Ruospo, Alberto Bosio, Alessandro Ianne, and Ernesto Sanchez. «Evaluating Convolutional Neural Networks Reliability depending on their Data Representation». In: *2020 23rd Euromicro Conference on Digital System Design (DSD)*. 2020, pp. 672–679. DOI: 10.1109/DSD51259.2020.00109 (cit. on p. 1).

[2] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. «Gradient-based learning applied to document recognition». In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324. DOI: 10.1109/5.726791 (cit. on pp. 1, 5, 6).

[3] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. «BinaryConnect: Training Deep Neural Networks with binary weights during propagations». In: *Advances in Neural Information Processing Systems*. Ed. by C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett. Vol. 28. Curran Associates, Inc., 2015. URL: https://proceedings.neurips.cc/paper/2015/file/3e15cc11f979ed25912dff5b0669f2cd-Paper.pdf (cit. on pp. 1, 6).

[4] FengFu Li, Bo Zhang, and Bin Liu. «Ternary Weight Networks». In: 2016. DOI: https://doi.org/10.48550/arXiv.1605.04711 (cit. on p. 1).

[5] Annachiara Ruospo, Ernesto Sanchez, Marcello Traiola, Ian O'Connor, and Alberto Bosio. «Investigating data representation for efficient and reliable Convolutional Neural Networks». In: 2021. DOI: https://dx.doi.org/10.1016/j.micpro.2021.104318 (cit. on pp. 1, 24, 40).

[6] Lars Wanhammar. «8 - DSP Architectures». In: *DSP Integrated Circuits*. Ed. by Lars Wanhammar. Academic Press Series in Engineering. Section 8.7. Burlington: Academic Press, 1999, pp. 357–385. ISBN: 978-0-12-734530-7. DOI: https://doi.org/10.1016/B978-012734530-7/50008-8. URL: https://www.sciencedirect.com/science/article/pii/B9780127345307500088 (cit. on p. 2).

[7] Marvin Minsky and Seymour A. Papert. *Perceptrons: An Introduction to Computational Geometry*. The MIT Press, Sept. 2017. ISBN: 9780262343930. DOI: 10.7551/mitpress/11301.001.0001. URL: https://doi.org/10.7551/mitpress/11301.001.0001 (cit. on p. 4).

[8] Kunihiko Fukushima. «Cognitron: A Self-Organizing Multilayered Neural Network». In: *Biol. Cybern.* 20.3–4 (Sept. 1975), pp. 121–136. ISSN: 0340-1200. DOI: 10.1007/BF00342633. URL: https://doi.org/10.1007/BF00342633 (cit. on pp. 4, 5).

[9] Douglas L. Perry. *VHDL: Programming by Example*. Chapter 2 - «Simulation Deltas» section. McGraw Hill Professional, 2002 (cit. on p. 13).