POLITECNICO DI TORINO

Master's Degree in Embedded Systems

Politecnico di Torino

Master's Degree Thesis

Resilience analysis of FPGA-based Dataflow accelerators

Supervisors

Prof. Claudio PASSERONE

Prof. Maurizio MARTINA

PhD. Pierpaolo MORÌ

PhD. Emanuele VALPREDA

Candidate

Giovanni POLLO

April 2023

Abstract

Neural networks are becoming increasingly popular in several domains, from image recognition to natural language processing. In addition, neural networks are entering safety-critical fields, such as autonomous driving. In these cases, it is crucial to have a fast and reliable inference. It is then necessary to deeply understand a specific model's robustness and resilience. These days, neural network models are also becoming bigger and more complex due to the higher number of parameters and layers. Therefore, many techniques have been developed to reduce the model's size, such as pruning and quantization. These techniques allow the execution of the model on edge devices, such as FPGA (Field Programmable Gate Arrays). In this thesis, the focus is on quantized neural network models. Another core aspect when dealing with edge device deployment is the design of the accelerator. Currently, there are two main architectures: systolic arrays and dataflow. In general, the characteristics of both accelerators are to parallelize computational tasks, such as convolution, and speed up the inference of the model. This allows for obtaining very high throughput on edge devices. The most significant advantage of the dataflow architecture is that each network layer has a custom architecture with a custom number of Processing Elements, contrary to the systolic array where all the layers share the same computation units. Additionally, on dataflow architecture, the access to the off-chip memory is minimized, significantly reducing latency with respect to systolic arrays . On the contrary, systolic arrays allow for more flexibility and do not require a different binary file to reconfigure the logic when the network topology changes. This thesis focuses on Dataflow-style architectures. The most significant contribution of this thesis is the development of a Fault Injector. This module allows the injection of errors (soft errors, such as bit-flips) inside a specific network layer to test how resilient networks, or parts of them, are. The critical aspects considered when developing the Fault Injector are the smallest possible footprint, low latency overhead, usability, and flexibility. All the modifications were done directly on the FINN compiler (the chosen framework for generating dataflow-style accelerators) to obtain higher usability and flexibility. . This ensured tight integration with the framework and guaranteed the compatibility of the Fault Injector with almost all already existing neural network models supported by FINN. Another feature implemented to achieve maximum flexibility is the addition of configurable parameters for the custom layer. The Fault Injector has some variables that enable deep customizability. The end user can specify if the inputs of the layer should be flipped; if the weights of the layer should be flipped; which bit should be flipped; which operation is faulty and which is not; the frequency of the occurrence of a fault . Moreover, these parameters are configurable at runtime thanks to a

simple JSON file, removing the need to re-synthesising the model. Small footprints and low latency were achieved thanks to an optimized fault injection algorithm and HLS tools, such as pragmas , which are special directives that can be used to optimize the design, reduce latency, improve throughput performance, and reduce area and device resource usage. All these optimizations allowed the building of a faulty model of Mobilenet while still having some free LUTs and FFs. The experiments shows how a faulty version of *Mobilenet-v1* was build with a 24.1% LUT and 18.8% FF overhead with respect to the original model. The observed results show that the higher the flipped bit importance (MSB instead LSB) the lower the final accuracy. Same behaviour is noticed when increasing the percentage of faulty operations as well as their frequency. Another interesting result is the comparison between flipping inputs or weights. In general, the experiments show that the weights are more sensitive compared to the inputs.

Acknowledgements

I would like to express my sincere gratitude to my thesis supervisor *Passerone Claudio* for his invaluable guidance, support, and encouragement throughout my research. His expertise and feedback have been instrumental in shaping and refining my work. I am also grateful to my co-supervisors *Martina Maurizio*, *Morì Pierpaolo* and *Valpreda Emanuele* for their insightful feedback and technical assistance that have contributed to the quality of my research.

Furthermore, I would like to thank my family for their unwavering love and support, which has been a constant source of inspiration and motivation throughout my academic journey. Their encouragement, understanding, and patience have been invaluable to me, and I would not have been able to complete this thesis without their support.

Finally, I would like to acknowledge the support of my friends and colleagues, who have encouraged me throughout the process of completing this thesis. Their understanding, support, and kindness have been a tremendous help to me.

"Vado così forte in salita per abbreviare la mia agonia"
Marco Pantani

Table of Contents

List of Tables

List of Figures

List of Listings

Acronyms

AI	Artificial Intelligence
ALU	Arithmethic Logic Unit
API	Application Programming Interface
BNN	Binary Neural Network
CNN	Convolutional Neural Network
DMA	Direct Memory Access
DNN	Deep Neural Network
DRAM	Dynamic random-access memory
DSP	Digital Signal Processor
FC	Fully Connected
RTL	Register Transfer Level
FIFO	First In First Out
FPGA	Field Programmable Gate Array
GEMM	General Matrix Multiplication
GPU	Graphics Processing Unit

HLS	High-Level Synthesis
IP	Intellectual Property
LUT	Look-Up Table
MAC	Multiply-Accumulate
ML	Machine Learning
ONNX	Open Neural Network Exchange
PE	Processing Element
QAT	Quantization Aware Training
QNN	Quantized Neural Network
QONNX	Quantized Open Neural Network Exchange
SIMD	Single Instruction Multiple Data
VHDL	Very High Speed Integrated Circuit Hardware Description Language
LSB	Least Significant Bit
MSB	Most Significant Bit
NN	Neural Network
MVAU	Matrix Vector Activation Unit
VVAU	Vector Vector Activation Unit
MUL	Multiply
JSON	JavaScript Object Notation

FF	Flip Flop
FM	Feature Map
FAT	Fault-Aware Training
RGB	Red Green Blue
ReLU	Rectified Linear Unit
ASIC	Application Specific Integrated Circuit
CPU	Central Processing Unit
DL	Deep Learning

Chapter 1

Introduction

Neural networks are becoming increasingly popular in several domains, from image recognition to natural language processing. In addition, neural networks are entering safety-critical fields, such as autonomous driving. In these cases, it is crucial to have a fast and reliable inference. It is then necessary to deeply understand a specific model's robustness and resilience.

These days, neural network models are also becoming bigger and more complex due to the higher number of parameters and layers. Therefore, many techniques have been developed to reduce the model's size, such as pruning and quantization. These techniques allow the execution of the model on edge devices, such as FPGA (Field Programmable Gate Arrays). In this thesis, the focus is on quantized neural network models.

Another core aspect when dealing with edge device deployment is the design of the accelerator. Currently, there are two main architectures: systolic arrays and dataflow. In general, the characteristics of both accelerators are to parallelize computational tasks, such as convolution, and speed up the inference of the model. This allows for obtaining very high throughput on edge devices. The most significant advantage of the dataflow architecture is that each network layer has a custom architecture with a custom number of Processing Elements, contrary to the systolic array where all the layers share the same computation units. Additionally, on dataflow architecture, the access to the off-chip memory is minimized, significantly reducing latency. On the contrary, systolic arrays allow for more flexibility and do not require a different bitfile when the network topology changes.

For this thesis, the chosen framework to generate dataflow accelerators is FINN [1], which is an experimental framework from Xilinx Research Labs to explore deep neural network inference on FPGAs. It specifically targets quantized neural networks, with emphasis on generating dataflow-style architectures customized for each network. The first step of the thesis was based on exploring and understanding in detail how FINN works. The framework is composed of three main parts. The

first one, external to the FINN compiler, prepares the quantized neural network by defining and training the model. The chosen framework is Brevitas, a PyTorch library for neural network quantization focusing on quantization-aware training (QAT) [2]. The prepared model is then given to the FINN compiler. Here the network is manipulated thanks to various transformations, which are fundamental in tidying up the model (for example, by collapsing layers or removing useless ones) and preparing each layer for the HLS conversion. Subsequently, the HLS code is generated, and the accelerator is synthesized. When the synthesis is completed, a bitstream is produced and flashed on the board to program the FPGA. This execution of the accelerated model on hardware relies on the PYNQ framework; PYNQ is an open-source project from AMD that makes it easier to use Adaptive Computing platforms by enabling the access and control of FPGA overlays.

The most significant contribution of this thesis is the development of a *Fault Injector*. This module allows the injection of errors (soft errors, such as bit-flips) inside a specific network layer to test how resilient networks, or parts of them, are. The critical aspects considered when developing the *Fault Injector* are the smallest possible footprint, low latency overhead, usability, and flexibility. All the modifications were done directly on the FINN compiler to exploit the last two characteristics. This ensured tight integration with the framework and guaranteed the compatibility of the *Fault Injector* with almost **all** already existing neural network models supported by FINN. The *Fault Injector* has some variables that enable deep customizability. The end user can specify if the inputs of the layer are faulty; if the weights of the layer are faulty; which operation is faulty and which is not; the frequency of the occurrence of a fault. Moreover, these parameters are configurable at runtime thanks to a simple JSON file, removing the need to re-synthesising the model. Small footprints and low latency were achieved thanks to an optimized fault injection algorithm and HLS tools, such as pragmas (special directives that can be used to optimize the design, reduce latency, improve throughput performance, and reduce area and device resource usage). The main contribution of this thesis are:

- Design of a *Fault Injector* module for a Dataflow-style architecture.
- Integration of the *Fault Injector* with a framework that generates neural network accelerators, to extend its functionality and scale it to be used for testing specific models' resilience.
- Comparison with state-of-the-art, highlighting a higher sensitivity to weights-flipping of the dataflow architecture.

The thesis is organized as follows:

- **chapter 1** gives a brief introduction of the topic
- **chapter 2** gives the reader all the basic knowledge needed to analyze this work properly
- **chapter 3** presents state-of-the-art, focusing on why this thesis could be a great advancement in it
- **chapter 4** groups together all the tools used for the development of the *Fault Injector*, as well as explaining in details the actual implementation of the module
- **chapter 5** presents the experiments and the results
- **chapter 6** wraps up the thesis and outlines future works

Chapter 2

Background

This section aims at introducing all topics that are necessary for the full comprehension of this thesis. It is organized as follows: **section 2.1** introduces Artificial Intelligence (AI), Machine Learning (ML) and Neural Networks (NN), focusing on their characteristics and relationships; **section 2.2** presents Convolutional Neural Networks (CNN) and explains the structure of such kind of models; **section 2.3** explains what are Quantized Neural Networks (QNN) and why they can be useful; **section 2.4** presents Neural Network Accelerators explaining different architectures with their pros and cons; **section 2.5** explains in details how the convolution is executed in hardware.

2.1 Artifical Intelligence and Machine Learning Introduction

Artificial intelligence (AI) is a field of computer science that aims to create intelligent machines. The term was coined in 1956 by John McCarthy, who defined it *as the science and engineering of making intelligent machines, especially intelligent computer programs.*

These days, AI is a vast field with many real-world applications. Some examples are the use of AI in self-driving cars [3], medical diagnosis [4], speech recognition [5], computer vision [6], natural language processing [7], and many more [8, 9, 10, 11, 12]. On the other hand, Machine Learning (ML) is a subfield of AI that focuses on developing computer programs that can improve their performance through experience.

During the last decades, thanks to the recent advances in computing power and the availability of large datasets [13], ML has become a trendy field. This allowed the development of many ML algorithms and the advancement of the state-of-the-art [14]. Some examples are the development of Deep Learning (DL)

algorithms, a class of ML algorithms inspired by the brain's structure and function (Figure 2.1). Neural Networks (NNs) are a computation model at the core of Deep learning, inspired by the human brain structure. They are composed of a set of interconnected units called neurons. Each neuron receives input from other neurons and produces an output that is a weighted sum of its inputs. The output of each neuron is then passed to the next layer of neurons. This process is repeated until the output layer is reached. Neural Networks can be divided into two big subsets:

1. **Shallow Neural Networks**: These are neural networks with a small number of layers. They are usually composed of an input layer, an output layer, and few hidden layer. An example of a shallow neural network is shown in Figure 2.2

2. **Deep Neural Networks**: These are neural networks with many layers. They are usually composed of an input and an output layer, and many hidden layers.

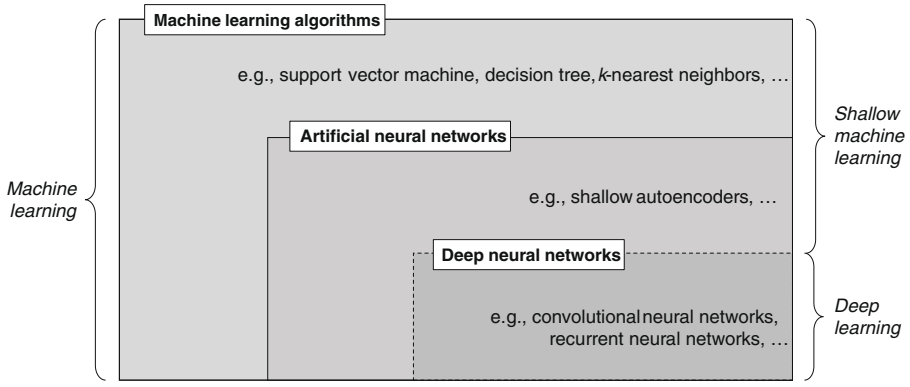


Figure 2.1: Venn diagram of machine Machine learning algorithms [14]

2.2 Convolutional Neural Networks

Convolutional neural networks (CNNs) are a type of deep neural network that is very popular in computer vision [16]. Since computer vision is mainly focused on images and video, the input of these models are usually composed of images (a video can be decomposed into a sequence of images). CNNs are composed of a set of convolutional layers, pooling layers, and fully connected layers [17]. A visual representation of a CNN is shown in Figure 2.3. Since most of the models used in this thesis are CNNs, the following sections will go into more detail about the utility of each layer.

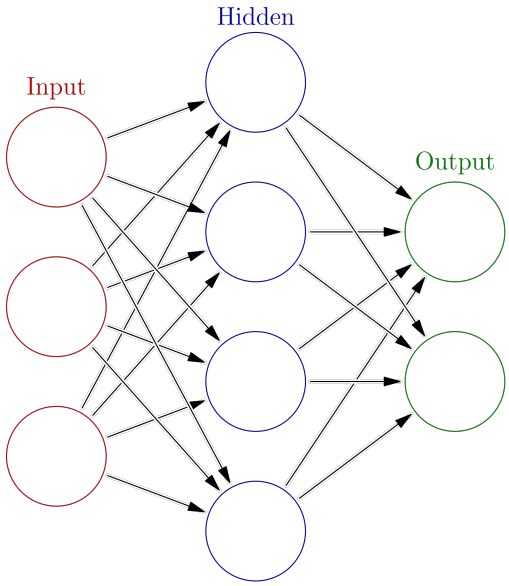


Figure 2.2: Shallow neural network [15]

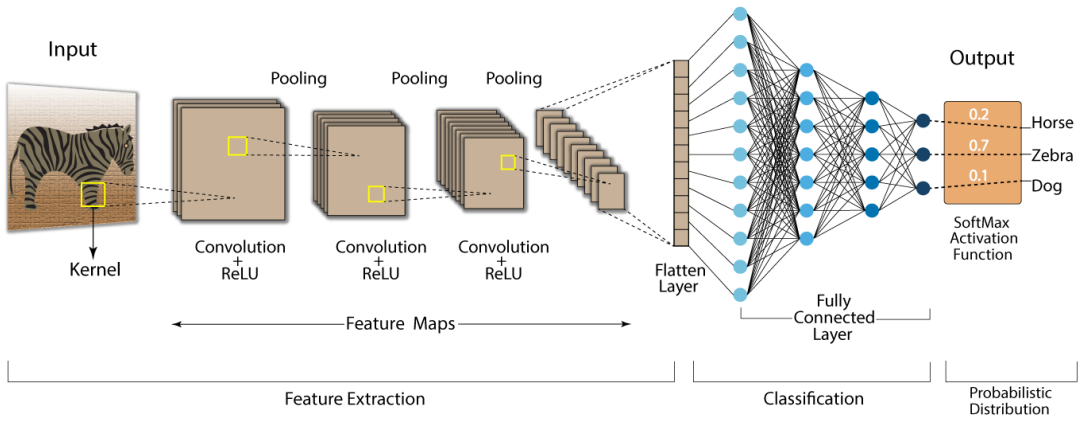


Figure 2.3: Convolutional neural network [16]

2.2.1 Convolutional Layer

As the name suggests, convolutional layers are the primary building block of CNNs. In ML convolution is applied to multi-dimensional tensors, a N-dimension generalization of arrays and matrices, and it is computed as a multiplication between matrices [18]. Its primary usage is to extract features, such as edges and corners, from the input image by sliding different kernels, a matrix used to extract the specific features from the input, over it. A visual representation of this operation

is shown in Figure 2.4

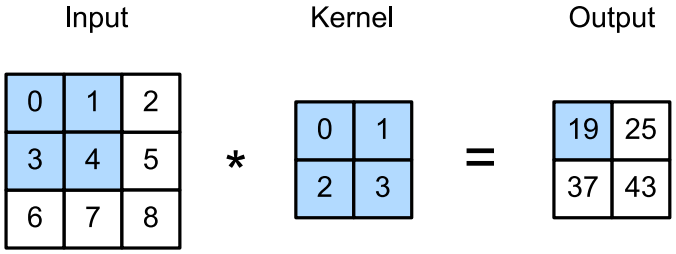


Figure 2.4: Convolution example

It is essential to understand the impact that different dimensions of inputs and kernels have on the output. In general, the convolution's output, also known as output feature map, is spatially smaller (if padding, stride and other techniques are not considered) than its input, also known as input feature map . Considering a 2D Convolutional Layer and defining an input tensor with spatial shape $N_{ix} \times N_{iy}$ and a kernel of shape $N_{kx} \times N_{ky}$ the output tensor shape will be $(N_{ix} - N_{kx} + 1) \times (N_{iy} - N_{ky} + 1)$. Since CNNs usually deal with images in real-world applications, a third dimension N_{if} (also called channel) is introduced. For example, in RGB images, three channels are used to represent the color of each pixel. Therefore, assuming an input tensor with shape $N_{if} \times N_{ix} \times N_{iy}$ and a kernel of shape $N_{if} \times N_{kx} \times N_{ky}$ the output tensor shape will be $1 \times (N_{ix} - N_{kx} + 1) \times (N_{iy} - N_{ky} + 1)$. It's easy to notice that the output tensor has only one channel in the latter example. N_{of} kernels are defined to get an output with multiple channels. Therefore, defining an input tensor with shape $N_{if} \times N_{ix} \times N_{iy}$ and a kernel of shape $N_{of} \times N_{if} \times N_{kx} \times N_{ky}$ the output tensor shape will be $N_{of} \times (N_{ix} - N_{kx} + 1) \times (N_{iy} - N_{ky} + 1)$.

2.2.2 Padding

As explained in the previous section, the output of the convolutional layer is smaller than its input. Sometimes, this behaviour is not good, and it is better to avoid it. To deal with it, *padding* is introduced. One straightforward solution to this problem is to add extra pixels of filler around the boundary of the input tensor, thus increasing the effective size of it. Typically the value of the extra pixels is 0. The padding is represented by two values P_x, P_y used to set how the input should be padded in the spatial dimensions. Padding is important also becasue it improves the information quality at the borders.

After the introduction of padding, the dimensions are the following:

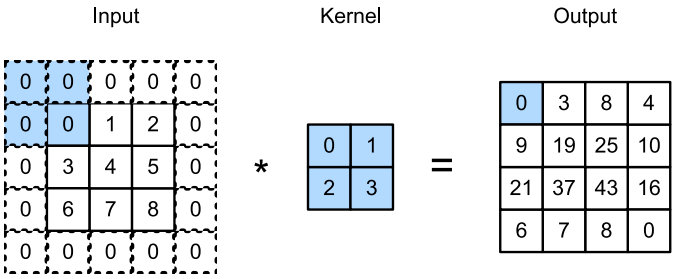


Figure 2.5: Padding example with padding 1 for height and 1 for width [19]

$$
\begin{aligned}
Input &: N_{if} \times N_{ix} \times N_{iy} \\
Weights &: N_{of} \times N_{if} \times N_{kx} \times N_{ky} \\
Output &: N_{of} \times (N_{ix} - N_{kx} + P_x + 1) \times (N_{iy} - N_{ky} + P_y + 1)
\end{aligned}
$$

2.2.3 Stride

In the previous paragraphs, kernel sliding is assumed to be one pixel at a time. However, for various reasons, such as computational efficiency or the need for spatial reduction , it may be useful to slide the kernel more than one pixel at a time. The stride is represented by two values $S_x \times S_y$, indicating the jump between the convolution windows in the spatial dimensions.

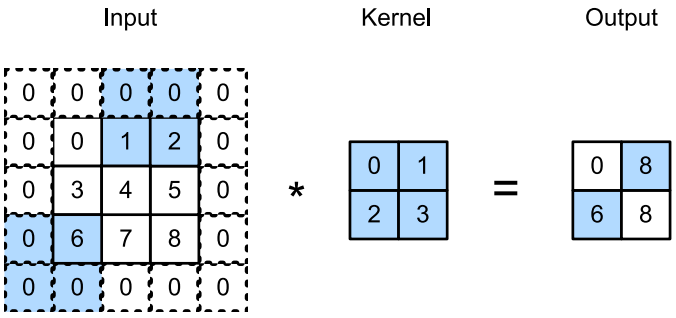


Figure 2.6: Stride example with a value of 3 for height and 2 for width [19]

Considering a convolution operation between a weight tensor $\mathcal{W}^l \in \mathbb{R}^{N_{of} \times N_{if} \times N_{kx} \times N_{ky}}$ and an input feature map $\mathcal{A}^{l-1} \in \mathbb{R}^{N_{if} \times N_{ix} \times N_{iy}}$ with stride $S_x \times S_y$ and padding $P_x \times P_y$, produces an output feature map $\mathcal{A}^l \in \mathbb{R}^{N_{of} \times N_{ox} \times N_{oy}}$. Where N_{ox} and N_{oy} are defined as reported in the following equation:

$$N_{ox} = \frac{N_{ix} - N_{kx} + P_x + S_x}{S_x}$$
$$N_{oy} = \frac{N_{iy} - N_{ky} + P_y + S_y}{S_y}$$

2.2.4 Activation Functions

Following the structural diagram of Figure 2.3, mentioning the activation functions is essential. These functions are differentiable operators to transform input signals to outputs and always introduce non-linearity [20] in order to reduce overfitting. The most common activation functions are the Rectified Linear Unit (ReLU), the Sigmoid, and the Tanh. The ReLU is the most used activation function in image processing because it is computationally efficient and easy to implement. The ReLU function is defined as follows:

Definition 2.2.1 (ReLU)**.** *Given an element, the function is defined as the maximum of that element and* 0

$$\text{ReLU}(x) = \max(0, x) \tag{2.1}$$

The plot of the function is shown in Figure 2.7

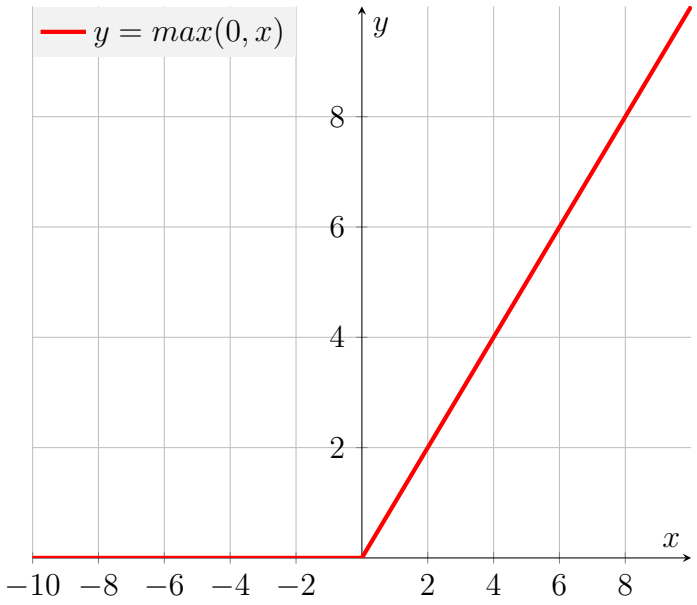


Figure 2.7: ReLU function

2.2.5 Pooling

Pooling layers are usually applied after the convolutional layers and are really important for the following reasons:

- **Dimensionality reduction**: The pooling layer reduces the spatial dimension of the feature map.
- **Overfitting reduction**: The pooling layer reduces the overfitting of the model by making the low-level feature (such as edges and corners) extraction location independent [21].

There are two different types of pooling layers; **Max Pooling** and **Average Pooling**.

Definition 2.2.2 (Max Pooling)**.** *Given an input matrix of size $N_{ix} \times N_{iy}$, a pooling window of size $N_{kx} \times N_{ky}$ and a stride of value S, with $N_{kx} \leq N_{ix}$ and $N_{ky} \leq N_{iy}$, the output matrix is defined as the maximum value of each pooling window.*

$$\text{Max Pooling}(x) = \max(x) \tag{2.2}$$

where x represents the pooling window. Figure 2.8 shows an example of Max Pooling.

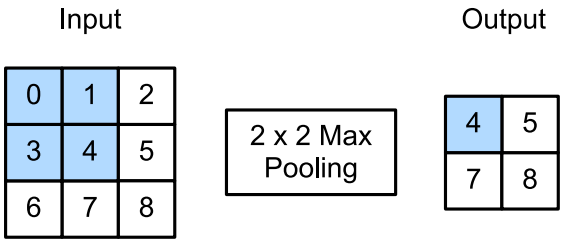


Figure 2.8: Max Pooling operation with stride = 1 [19]

Definition 2.2.3 (Average Pooling)**.** *Given an input matrix of size $N_{ix} \times N_{iy}$, a pooling window of size $N_{kx} \times N_{ky}$ and a stride of value S, with $N_{kx} \leq N_{ix}$ and $N_{ky} \leq N_{iy}$, the output matrix is defined as the average value of each pooling window*

$$\text{Avg Pooling}(x) = avg(x) \tag{2.3}$$

where x represents the pooling window. One of the reason of using pooling is the spatial dimensionality reduction. These are the dimension values:

$$\begin{aligned} Input &: N_{ix} \times N_{iy} \\ Pooling\ Window &: N_{kx} \times N_{ky} \\ Output &: \frac{N_{ix} - N_{kx}}{S} \times \frac{N_{iy} - N_{ky}}{S} \end{aligned} \tag{2.4}$$

2.2.6 Fully Connected Layer

Fully connected (FC) layers are commonly added at the end of a (CNN) to enable the neural network to classify images into various categories. Here are some reasons why FC layers are necessary in CNN architectures:

- **Classification**: Fully connected layers are used for classification since they can identify relations between all pixels in the input feature map.
- **Feature Composition**: Convolutional Layers detect features in an image but they do not capture feature combinations or variations. FC layers use these deep features detected from the convolutional layers and combine them in different ways, enabling them to learn higher-level representations of features, and allowing for more precise classification.

In conclusion, fully connected layers are widely used for accurate image classification in CNN architectures.

2.3 Quantized Neural Networks

These days, neural network models are becoming bigger and more complex due to the higher number of parameters and layers. This led to the need to find new methods to reduce footprint of specific models to fit in the embedded domain for edge deployment . The most important technique for this thesis is quantization [2, 22].

The tensors of modern neural networks are usually represented with a `float32` datatype. This allows a very high precision, but on the other hand, it requires a lot of resources for training and inference . A solution may be reducing the number of bits used to represent the tensors. This guarantees a lower resource impact but on the other side provokes a decrease in accuracy. This tradeoff is not always feasible and for this reason, it is really important to define the quantization level to be able to maintain an acceptable accuracy. It is common to find models whose inputs and weights are quantized to 2, 4 or 8 bits. The quantization level can be pushed up to binarization [23], in which inputs and weights can be represented on 1 bit.

2.4 Neural Networks Accelerator

The need of high throughput and low latency also on edge devices, such as FPGAs, created the need to define some techniques to bring neural network models to hardware [24]. This process has also been simplified by using techniques like quantization that allowed big network models to fit on resource-constrained devices. Moreover, to utilize all the possibilities that various hardware architectures offer, it is crucial to design custom architectures to parallelize as much as possible the computationally intensive operations. This section is intended to briefly overview the possible architectures used to accelerate neural networks with their positive and negative characteristics, focusing on cost, development complexity, flexibility and throughput [25].

- **Graphical Processing Units (GPU)**: GPUs are one of the most common architectures to significantly improve neural networks performance. Thanks to the very high number of ALU they are able to achieve a really high throughput. Due to their large space requirements and high power consumption, they are not a common choice for edge deployment . However, thanks to their flexibility and high performance, they are usually used in datacentres.
- **Application Specific Integrated Circuit (ASIC)**: These architectures, as the name suggests, are specific for the considered application and are not flexible. For this reason, the performance are really high, but the complexity and engineering costs are really high as well, mainly because the design cycle may span a protracted timeframe.
- **Field Programmable Gate Array (FPGA)**: FPGA stands in the middle of the other two architectures. They are flexible because they provide *field programmability*, but at the same time they maintain high throughput performance thanks to the high bandwidth of the on-chip memory.

For this work, the choice has been an FPGA, and all the tests have been held on a Xilinx ZCU104. As mentioned in the Introduction, the two most popular accelerator architectures are the **Systolic Array** [26] and **Dataflow**. The systolic array is the simpler one structure-wise. It is composed of a matrix of computational units shared between all layers. On the other side, the Dataflow architecture is much more complex since the number of computational units is customizable per layer offering higher customizability. These are the advantages and disadvantages of both architectures:

- **Compactness**: Having one computational matrix in common between all layers, guarantees that the systolic is more compact architecture-wise with respect to the datafow one.

- **Scalability**: the simplicity and the generality of the systolic architecture ensure that the same architecture might be used to accelerate different neural network models. On the contrary, since the Dataflow is much more network-specific, it is more difficult to scale it to different models.

- **Latency**: Having one computational unit in each layer drastically reduces the off-chip memory access. In fact the Dataflow structure is a streamlined architecture where data are passed through one layer to another.

- **Performance**: Both architectures are able to achieve really high throughput. It isn't easy to compare them since they are completely different structure-wise.

2.5 Convolution in Hardware: Im2col

To properly understand how a neural network is accelerated on hardware, it is crucial to know how to accelerate its main operation, the convolution. As explained in subsection 2.2.1, the convolution is a series of matrix multiplications. There are various type of hardware implementations, but for this thesis we will focus on the General Matrix Multiplication (GEMM). To simplify the comprehension of the GEMM let's consider the following example.

The input I has a size $N_{ix} \times N_{iy} \times N_{if}$ of $3 \times 3 \times 2$, meaning that there are two channels of shape 3×3. The filters K have a size $N_{of} \times N_{if} \times N_{kx} \times N_{ky}$ of $2 \times 2 \times 2 \times 1$. Supposing a stride of 1 and no padding, referring to Equation 2.4, it is easy to derive that the output FM size will be $2 \times 2 \times 1$.

$$I_0 = \begin{bmatrix} I_{0,0,0} & I_{0,1,0} & I_{0,2,0} \\ I_{1,0,0} & I_{1,1,0} & I_{1,2,0} \\ I_{2,0,0} & I_{2,1,0} & I_{2,2,0} \end{bmatrix} \quad K_0 = \begin{bmatrix} K_{0,0,0} & K_{0,1,0} \\ K_{1,0,0} & K_{1,1,0} \end{bmatrix}$$

$$I_1 = \begin{bmatrix} I_{0,0,1} & I_{0,1,1} & I_{0,2,1} \\ I_{1,0,1} & I_{1,1,1} & I_{1,2,1} \\ I_{2,0,1} & I_{2,1,1} & I_{2,2,1} \end{bmatrix} \quad K_1 = \begin{bmatrix} K_{0,0,1} & K_{0,1,1} \\ K_{1,0,1} & K_{1,1,1} \end{bmatrix}$$

To obtain the output, these are the computations that need to be performed:

$$
\begin{aligned}
FM_{0,0} &= I_{0,0,0}K_{0,0,0} + I_{0,1,0}K_{0,1,0} + I_{1,0,0}K_{1,0,0} + I_{1,1,0}K_{1,1,0} + \\
&+ I_{0,0,1}K_{0,0,1} + I_{0,1,1}K_{0,1,1} + I_{1,0,1}K_{1,0,1} + I_{1,1,1}K_{1,1,1} \\
FM_{0,1} &= I_{0,1,0}K_{0,0,0} + I_{0,2,0}K_{0,1,0} + I_{1,1,0}K_{1,0,0} + I_{1,2,0}K_{1,1,0} + \\
&+ I_{0,1,1}K_{0,0,1} + I_{0,2,1}K_{0,1,1} + I_{1,1,1}K_{1,0,1} + I_{1,2,1}K_{1,1,1} \\
FM_{1,0} &= I_{1,0,0}K_{0,0,0} + I_{1,1,0}K_{0,1,0} + I_{2,0,0}K_{1,0,0} + I_{2,1,0}K_{1,1,0} + \\
&+ I_{1,0,1}K_{0,0,1} + I_{1,1,1}K_{0,1,1} + I_{2,0,1}K_{1,0,1} + I_{2,1,1}K_{1,1,1} \\
FM_{1,1} &= I_{1,1,0}K_{0,0,0} + I_{1,2,0}K_{0,1,0} + I_{2,1,0}K_{1,0,0} + I_{2,2,0}K_{1,1,0} + \\
&+ I_{1,1,1}K_{0,0,1} + I_{1,2,1}K_{0,1,1} + I_{2,1,1}K_{1,0,1} + I_{2,2,1}K_{1,1,1}
\end{aligned}
$$

This algorithm is very efficient on CPU, GPU and ASIC . On the contrary, to achieve high throughput performance on FPGA too, is crucial to minimize the off-chip access [27]. This is achieved by preparing the data exploiting data-redundancy, with the downside of increasing memory occupation. the tradeoff memory overhead and performance is crucial and it changes based on the available resources.

Let's take $FM_{0,0}$ as an example. The first four terms are obtained by multiplying the submatrix of I_0 composed by the first two rows and the first two columns with K_0. The other four terms are obtained by multiplying the submatrix of I_1 composed by the first two rows and the first two columns with K_1. The same logic can be applied to the other terms of FM. The problem is that the submatrices of I_0 and I_1 are not contiguous in memory. Therefore, the data must be loaded from memory and then transferred to the accelerator. This is a costly operation, and it is crucial to minimize it. To solve this issue, the easiest solution is to exploit data redundancy. Therefore, the data are organized as follows:

$$
I = \begin{bmatrix}
I_{0,0,0} & I_{0,1,0} & I_{1,0,0} & I_{1,1,0} & I_{0,0,1} & I_{0,1,1} & I_{1,0,1} & I_{1,1,1} \\
I_{0,1,0} & I_{0,2,0} & I_{1,1,0} & I_{1,2,0} & I_{0,1,1} & I_{0,2,1} & I_{1,1,1} & I_{1,2,1} \\
I_{1,0,0} & I_{1,1,0} & I_{2,0,0} & I_{2,1,0} & I_{1,0,1} & I_{1,1,1} & I_{2,0,1} & I_{2,1,1} \\
I_{1,1,0} & I_{1,2,0} & I_{2,1,0} & I_{2,2,0} & I_{1,1,1} & I_{1,2,1} & I_{2,1,1} & I_{2,2,1}
\end{bmatrix}
\qquad
K = \begin{bmatrix}
K_{0,0,0} \\ K_{0,1,0} \\ K_{1,0,0} \\ K_{1,1,0} \\ K_{0,0,1} \\ K_{0,1,1} \\ K_{1,0,1} \\ K_{1,1,1}
\end{bmatrix}
$$

The computation of $FM_{0,0}$ is now performed dot-multiplying the green values with the purple ones as follows:

$$I = \begin{pmatrix} I_{0,0,0} & I_{0,1,0} & I_{1,0,0} & I_{1,1,0} & I_{0,0,1} & I_{0,1,1} & I_{1,0,1} & I_{1,1,1} \\ I_{0,1,0} & I_{0,2,0} & I_{1,1,0} & I_{1,2,0} & I_{0,1,1} & I_{0,2,1} & I_{1,1,1} & I_{1,2,1} \\ I_{1,0,0} & I_{1,1,0} & I_{2,0,0} & I_{2,1,0} & I_{1,0,1} & I_{1,1,1} & I_{2,0,1} & I_{2,1,1} \\ I_{1,1,0} & I_{1,2,0} & I_{2,1,0} & I_{2,2,0} & I_{1,1,1} & I_{1,2,1} & I_{2,1,1} & I_{2,2,1} \end{pmatrix} \quad K = \begin{pmatrix} K_{0,0,0} \\ K_{0,1,0} \\ K_{1,0,0} \\ K_{1,1,0} \\ K_{0,0,1} \\ K_{0,1,1} \\ K_{1,0,1} \\ K_{1,1,1} \end{pmatrix}$$

The exact process shown in the example is repeated for the other terms of FM. A clear example is visible in Figure 2.9.

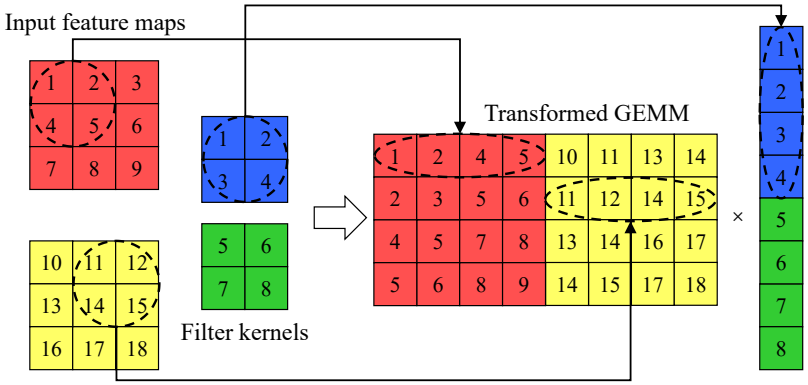


Figure 2.9: General Matrix Multiplication [27]

2.6 High-level Synthesis (HLS)

High-level Synthesis is the process of generating a Register Transfer Level (RTL) description starting from a higher level of abstraction. The abstracted design is usually defined using C or C++ and after the Sythesis process, the obtained RTL design is represented in Verilog or VHDL. During this translation process, the tool executes some optimization in order to obtain the maximum performance while respecting some user-defined parameters (such as area occupation, latency and power consumption). Generally it tries to find the best possibile implementation given a specific set of inputs. The main goal of HLS is to allow the hardware designer to work at a higher level of abstraction, focusing on algorithmic optimization rather that RTL. HLS gained a lot of popularity in the last decades, because they

allowed the design of complex architecture with ease and increased the designer's productivity. The main advantages of HLS over RTL designs are the following:

- **Productivity**: By abstracting design to the functional level and relying on automated high-level synthesis for implementation, designers can more rapidly explore and develop more optimal architectures
- **Reusability**: Being free of low-level details, the code is easier to reuse or to port to a different architecture.
- **Simplicity**: Being described in C or C++, compared to RTL, high-level design descriptions typically have an order of magnitude fewer lines of code and are significantly easier to read and maintain.

Chapter 3

Related Works

This chapter aims to provide an overview of the current state of knowledge, identify gaps in the literature, and emphasize the importance of the thesis's contribution to the field. The presented works are the most relevant for this thesis since they served as a starting point and a methodology inspiration. The chapter is organized as follows: **section 3.1** presents the framwork that has been used for developing the neural network accelerators; **section 3.2** presents a methodology for analyzing the resilience of neural network accelerators; **??** dives into how FINN implements the convolution.

3.1 FINN: A Framework for Fast, Scalable Binarized Neural Network Inference [1]

Since the growth in popularity of neural networks, the research to accelerate them has been really active. In the past years, Dataflow architectures had been discussed a lot and the need of a framework able to accelerate complex models came up. FINN is a framework, developed by Xilinx Research Labs, that focuses on building fast and flexible FPGA accelerators. It has been designed with highly quantized neural networks in mind (up to BNNs). For the development of this thesis FINN has been fundamental, and it has been the starting point for developing the *Fault Injector*. The approach of this thesis can be scaled and ported to other Dataflow-oriented frameworks for accelerating neural networks.

The big selling point of FINN is the customizability and the tight integration with Brevitas, a very popular framework for QNNs built on top of Pytorch. Regarding customizability, being a Dataflow-oriented framework, FINN guarantees a per-layer parallelism decision for the accelerator. The authors introduced two parameters, Processing Elements (PE) and Single Instruction Stream Multiple Data Stream (SIMD) that are really important when applying the parallelism configurations.

PE are a subset of the output feature map, while SIMD are a subset of the input feature map. The results presented by the authors are very convincing, especially the throughput where *"the design outperforms all others"*.

3.2 Mind the Scaling Factors: Resilience Analysis of Quantized Adversarially Robust CNNs [28]

As introduced in previous sections, neural networks are widespread in many fields. It is essential to build fast and reliable architectures in some of them, such as automotive and aerospace. This paper analyzes the resilience of a Systolic Array accelerator and compares the failure rate of vanilla-trained networks with respect to adversarially-trained ones. This paper greatly impacted this thesis since it inspired the approach and the parameters used to design a Fault Injector from scratch. The considered parameters are the following:

- **Frequency of Bit-Flip occurence**
- **Datatype**
- **Bit Position**
- **Percentange of MAC faulty**

As it deals with Systolic Arrays architecture, the mentioned paper does not have a per-layer control of the faults. In fact, as already explained in section 2.4, that type of architecture presents a unique computational module, made of N computational units, shared between all layers. The novelty in the state of the art introduced by this thesis is the development of a similar *Fault Injector* design (keeping the same parameters for consistency) on top of a Dataflow architecture. Thanks to the different architecture, a per-layer control of the fault is exploited, allowing end users more granular control of the behaviour of each layer. The technique used in the paper to extrapolate very interesting results have been a solid starting point for the execution of various experiments reported in chapter 5. The authors also investigated the quantization scaling factor. They identified a large difference in the quantization scaling factors of the CNNs which are resilient to hardware faults and those which are not. However, since adversarial training focuses on input perturbations, the internal weights are more prone to hardware faults. To solve this problem the authors proposed a weight decay and obtained convincing results.

Chapter 4

Methodology

This chapter focuses on the methodology behind the implementation of the *Fault Injector*. It also presents and explains all the tools, libraries and framework that have been used. It is organized as follows: **section 4.1** presents Brevitas and some of its features; **section 4.2** presents FINN and explains in details how the tool is composed and how it works; **section 4.3** introduces the concepts of the *Fault Injector* and lists its parameters; **section 4.4** explains the first desing of the *Fault Injector* highlighting its characteristics as well as its problems; **section 4.5** presents the final *Fault Injector* design, giving also some examples of how it works internally; **section 4.6** shows the core parts of the code of the final version of the *Fault Injector*.

4.1 Brevitas

Brevitas [29] is a python library that sits on top of PyTorch (an optimized tensor library for deep learning using GPUs and CPUs) , for QNNs with a focus on Quantization Aware Training . To undesrstand the power of Brevitas, it is important to mention the various existing techniques to work with QNNs. In general, the need of quantization for neural network has already been explained in section 2.3. There are two types of quantizations:

- **Post-Training quantization**: in this case, the precision reduction happens after the training. For this reason, it is really easy to apply . There are various type of post-training quantization, based on what needs to be quantized and the wanted precision. Tensorflow, for examples, offers *Dynamic Range*, *Full Range* and *Float16* quantizations. Post Training Quantization can be easily used up to 8 bit. However, for low-precision it is not accurate.
- **Quantization-Aware Training**: this technique, as the name suggests, takes

place before and during training. In fact, the network is quantized at training , and therefore the training process is done directly with the quantized values. This allows a lower accuracy degradation compared to Post-Training Quantization and it is mandatory for lower than 8 bits quantization [30, 31].

Brevitas exploits the second approach, since it deals with low-precision network (down to 1 bit) . It provides the quantized implementation of the most common neural network layers, such as the fully connected and convolutional layer. FINN is tightly integrated with Brevitas and expects a quantized neural network exported directly from brevitas as an ONNX model.

4.2 FINN

The FINN project is an experimental framework developed by *Xilinx Research Labs* to explore deep neural network inference on FPGAs. Reading the documentation [32] of the project, we find out that FINN is composed of *"two separate but highly related things:"*:

- **The FINN Project**: FINN research project is an experimental framework to explore deep neural network inference on FPGAs. It specifically targets QNN, with emphasis on generating *dataflow-style architectures* customized for each network. The project is open-source and available on GitHub [33]. For this thesis, the **0.8.1** version of FINN has been used.
- **The FINN Compiler**: the compiler is a central part of the FINN project that maps QNNs dataflow-style FPGA architectures.

4.2.1 End-to-End Flow

The FINN compiler is composed of three main parts: *Brevitas Export*, *Network Preparation* and *Harware Build*. The following sections focus on the steps that are executed and which kind of manipulation is done on the network.

Brevitas Export

As already explained in section 4.1, FINN works with QNNs. Thanks to the QAT and the features of brevitas, it is possible to export the model in ONNX format. From the ONNX documentation, we can read that:

> *ONNX aims at providing a common language any machine learning framework can use to describe its models. With ONNX, it is possible to build a unique process to deploy a model in production and independent from the learning framework used to build the model*

The catch is that the ONNX format is independent from the learning framework used to build the model. In addition to that, its ease of use makes it a very good choice for manipulating and modifying neural network models. Moreover, during the years the ONNX has grown a lot in popularity and nowadays a lot of visualization tools exists, such as Netron [34].

In order for FINN to support quantization, the FINN developement team has created a custom Python class, called `ModelWrapper`, that is used to import ONNX models into the framework with the support of all the quantization attributes that are present in the model.

At this step, the model has been correctly imported into FINN, and the transformation process can start.

Network Preparation

This step, as expected, comes right after the Brevitas export. This section will explain the transformations applied to the model to make it compatible with the FINN compiler. The scope behind the transformation is to optimize the network and convert the nodes to custom nodes that correspond to *finn-hlslib* [35]. The latter contains the C++ description of custom layers for the implementation of quantized neural networks using dataflow architecture.

The first group of transformations is called **Tidy-up** , it is executed after every modification done to the network and it is composed of the following functions:

- **InferShapes**: Ensure every tensor in the model has a specified shape (ValueInfo).
- **FoldConstants**: Replace the output of a node with const-only inputs with a precomputed result.
- **GiveUniqueNodeNames**: Give unique names to each node in the graph using enumeration, starting with given prefix (if specified in the constructor)
- **GiveReadableTensorNames**: Give more human-readable names to all internal tensors. You should apply GiveUniqueNodeNames prior to this transform to avoid empty node names, as the readable names are based on the node names.
- **InferDataTypes**: Infer QONNX [36] DataType info for all intermediate/output tensors based on inputs and node type.

These transformations are significant since they are repeated after every step of the whole process. This ensures that the model is always in a consistent state.

```python
def step_tidy_up(model: ModelWrapper, cfg: DataflowBuildConfig):
    """
    Run the tidy-up step on given model. This includes shape
    and datatype inference, constant folding, and giving nodes
    and tensors better names.
    """
    model = model.transform(InferShapes())
    model = model.transform(FoldConstants())
    model = model.transform(GiveUniqueNodeNames())
    model = model.transform(GiveReadableTensorNames())
    model = model.transform(InferDataTypes())
    model = model.transform(RemoveStaticGraphInputs())
    if VerificationStepType.TIDY_UP_PYTHON in
    cfg._resolve_verification_steps():
        verify_step(model, cfg, "initial_python", need_parent=False)
    return model
```

Listing 1: Python code of the Tidy-up step

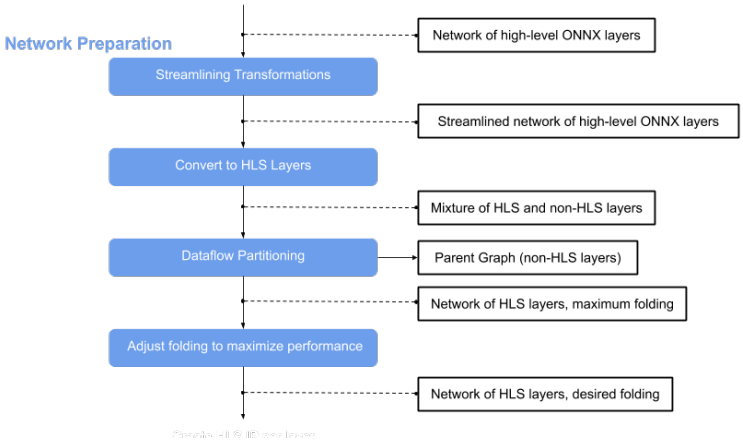


Figure 4.1: FINN transformations

Moving to Figure 4.1, the first step is called **Streamlining Transformations**. Streamlining eliminates floating point operations in a model by moving them around, collapsing them into one operation and transforming them into multi-thresholding nodes. Several transformations are involved in this step. To have a deeper understanding of the streamlining transformations, the FINN documentation [37] provides a detailed explanation of each transformation.

The next step is the **Convert to HLS Layers** . This step is very important

to understand the tight integration that exists between the FINN compiler and finn-hlslib. In fact, as stated on the documentation of this step:

> *HLS layers are layers that directly correspond to a finn-hlslib function call. For example pairs of binary XNORPopcountMatMul and MultiThreshold layers are converted to MatrixVectorActivation layers.*

In this thesis, the focus is on one particular layer: the `MatrixVectorActivation`. At the end of this step the network is divided into a mixture of several HLS and non-HLS layers. The **Dafaflow Partitioning** step is the second-last step in the Figure 4.1. Here the HLS and non-HLS layers are separated and further processed. Moreover, a starting parallelism of 1 PE and 1 SIMD is assigned to each HLS layer.

The last step is the **Folding Adjustment**. This step is responsible for assigning the desired parallelism to each layer of the network. The assignable parameters are the number of PEs and the number of SIMDs. This allows achieving the desired performance, latency and area overhead. The higher the PEs and SIMDs the higher the performance, but the higher the area.

Hardware Build and Deployment

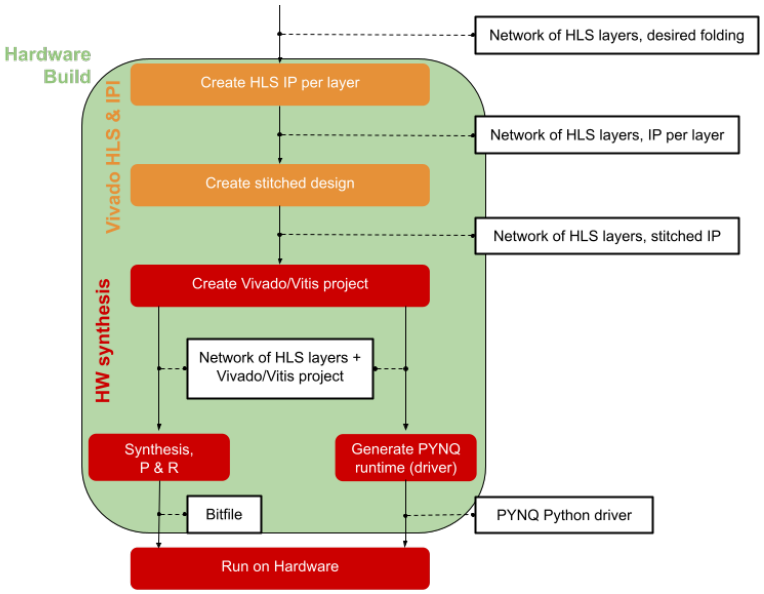


Figure 4.2: FINN transformations

Entering this part of the flow, the model is ready to be synthesized thanks to the Xilinx tools, such as Vitis HLS and Vivado. Firstly, for each layer of the network,

an IP is created. Afterwards, all these blocks are interconnected to replicate the network model's structure. Moreover, some FIFOs are inserted between streaming layers to ensure the correct data flow. The need of FIFOs is motivated by the fact that data are read and written in bursts; therefore, if there are no data ready to be read, the computation slows down. It is then essential to correctly size the depth of the FIFOs, to avoid stalls in the execution pipeline. Choosing the wrong depth can lead to a deadlock in the execution pipeline or high latency.

Now that the accelerator is ready, the process continues by preparing the top-level project and the host code. This step is mainly done by Vivado, responsible for generating the bitstream. FINN will now insert custom hardware-oriented ONNX nodes into the graph. These are DMA engines for moving data into and out of the accelerator (from DRAM) and data width converters between consecutive nodes where required.

FINN generates a Python driver to test the model's performance and accuracy on hardware. This driver is responsible for loading the bitstream into the FPGA, initializing the accelerator, and packing/unpacking the input and output tensors. All these features are exploited thanks to the PYNQ [38] APIs.

The driver has two execution modes:

- **Validate**: this mode tests the model's accuracy on the FPGA.
- **Throughput**: this mode evaluates the model's performance on the FPGA. In particular, it measures the following metrics:
  - **Runtime**: the time required to execute the model on the FPGA.
  - **Throughput[images/s]**: the number of images processed per second.
  - **DRAM_in_bandwidth[MB/s]**: the amount of data read from DRAM per second.
  - **DRAM_out_bandwidth[MB/s]**: the amount of data written to DRAM per second.
  - **Fold_input[ms]**: the time required to fold the input tensor into the accelerator's input format.
  - **Pack_input[ms]**: the time required to pack the input tensor into the accelerator's input format.
  - **Copy_input_data_to_device[ms]**: the time required to copy the input tensor from the host to the FPGA.
  - **Copy_output_data_from_device[ms]**: the time required to copy the output tensor from the FPGA to the host.
  - **Unpack_output[ms]**: the time required to unpack the output tensor from the accelerator's output format.

– **Unfold_output[ms]**: the time required to unfold the output tensor from the accelerator's output format.

FINN Internal Architecture

For this thesis, the core part of the FINN internal architecture is the module that executes the convolution. In the following subsections, all the details of how FINN implements the convolution are explained.

Convolution Input Generator

This module is responsible for preparing the input data that are used to execute the matrix vector multiplication. Functionality wise, the Convolution Input Generator uses the algorithm explained in section 2.5. The FINN variant of the Convolution Input Generator introduces a new parameter, the **SIMD**. This parameter is responsible for the parallelism and it is very important to set it coherently with the SIMD of the Matrix Vector Activation, otherwise the execution will return wrong results.

Matrix Vector Activations

This module executes the actual dot product between the prepared input data and the weights. The two most important parameters of this module are the **SIMD** and the **PE**:

- **Single Instruction Multiple Data (SIMD)**: This value exploits the input lanes of each PE.
- **Processing Element (PE)**: This value exploits the number of computational units.

The overview of the structure is shown in Figure 4.3. P represents the number of PE, while S represents the number of SIMD. Each SIMD of each PE receives a specific value of input and weights and computes the multiplication. The temporary results is generally saved in an accumulator, as shown in Figure 4.6, until the convolutional window has been completed when the final value is then written the output tensor.

Convolution in FINN

To clearly explain how the convolution operation is implemented in FINN based on the value of PE and SIMD, the following example can be very useful. First, let's define the dimensions and parameters of the layer [40]:

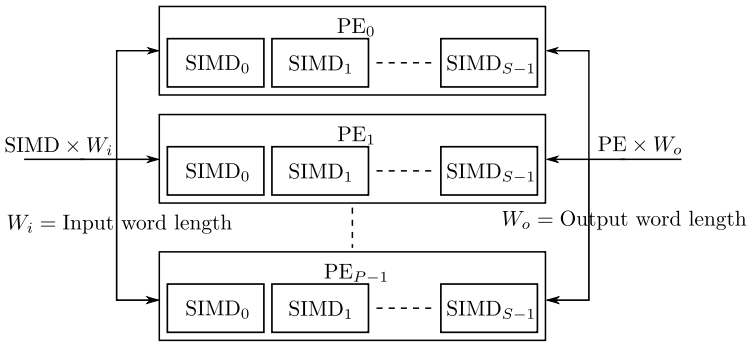


Figure 4.3: Structure of the computational unit[39]

- **Padding**: 0
- **Stride**: 1
- **Input shape**[1]: $N_{if} \times N_{ix}$
- **Weight shape**[2]: $N_{of} \times N_{if} \times N_{kd}^2$
- **Output tensor**[3]: $N_{of} \times N_{od}^2$
- **PE**: PE
- **SIMD**: SIMD

The input tensor is reshaped based on the number of *SIMD* and its shape is:

$$N_{od} \times N_{od} \times \frac{N_{kd}^2 \cdot N_{if}}{SIMD} \times SIMD$$

The reshaped input tensor has more values than the original one. This happens because data redundancy is exploited in the reshaping phase to maximize performance. Figure 4.4 shows an example of reshaping of an input tensor with 2 SIMD.

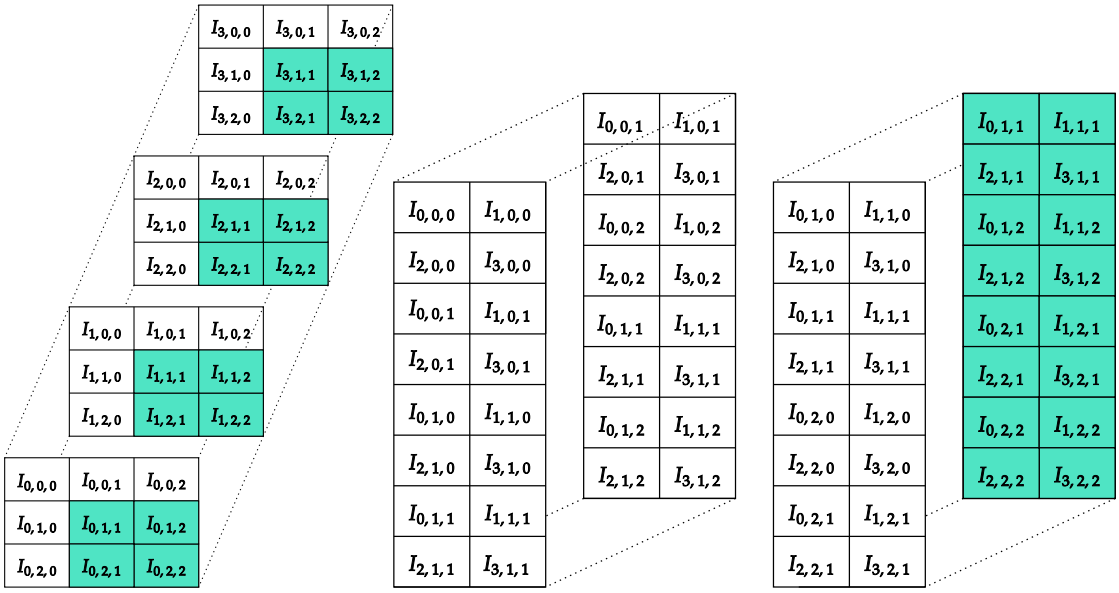
The weight tensor is reshaped (Figure 4.5) based on both the number of PE and SIMD and its shape is:

$$\frac{N_{of} \cdot N_{if} \cdot N_{kd}^2}{PE \cdot SIMD} \times PE \cdot SIMD$$

[1]Supposing $N_{ix} = N_{iy} = N_{id}$

[2]Supposing $N_{kx} = N_{ky} = N_{kd}$

[3]Supposing $N_{ix} - N_{kx} - 1 = N_{iy} - N_{ky} - 1 = N_{od}$



(a) Input Tensor Pytorch representation

(b) Input Tensor after reshaping with 2 SIMD

Figure 4.4: Input tensor before and after reshaping

The reshaped weight tensor is read N_{od}^2 times. So its final shape is:

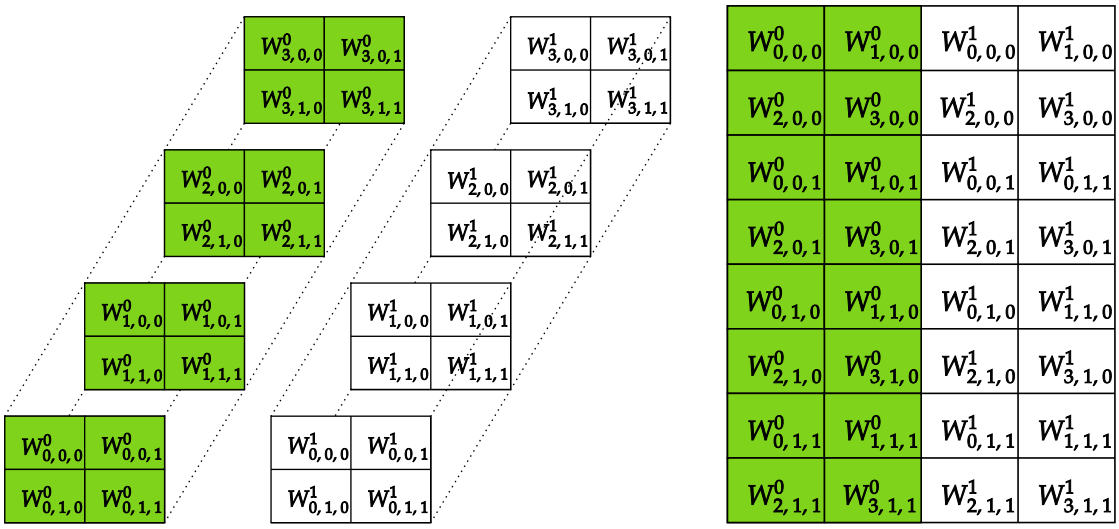
$$N_{od}^2 \times \frac{N_{of} \cdot N_i f \cdot N_{kd}^2}{PE \cdot SIMD} \times PE \cdot SIMD$$

When the tensors are in the correct shape, the computation can start. The structure of the computational unit is shown in Figure 4.6. The accumulator plays a crucial role and since the computation takes more than one cycle, the temporary value is accumulated in that block (usually initialiazed to 0).

Figure 4.7 shows the first iteration of the computational loop. The orange values are passed to both computational units. Conversely, the green values are passed to the first computational unit, while the yellow ones are to the second. So these are the operations executed by the units:

$$\begin{aligned} \textit{Accumulator Computational Unit } 0 = \ & I_{0,0,0} \cdot W_{0,0,0}^0 + I_{1,0,0} \cdot W_{1,0,0}^0 \\ \textit{Accumulator Computational Unit } 1 = \ & I_{0,0,0} \cdot W_{0,0,0}^1 + I_{1,0,0} \cdot W_{1,0,0}^1 \end{aligned}$$

Figure 4.8 shows the second iteration. In this case, the new value of the accumlators is the old value of the accumulators plus the new computed value based on the Inputs and Weights received. After $\frac{N_{kd}^2 \cdot N_{if}}{SIMD}$ iterations (when $I_{2,1,1}$ and $I_{3,1,1}$ are reached), the accumulators contain two of the final values, which are written to the output tensor. Therefore, the accumulators are reset to 0 and the



(a) Weight Tensor Pytorch representation

(b) Weight Tensor after reshaping with 2 SIMD

Figure 4.5: Weight Tensor before and after reshaping

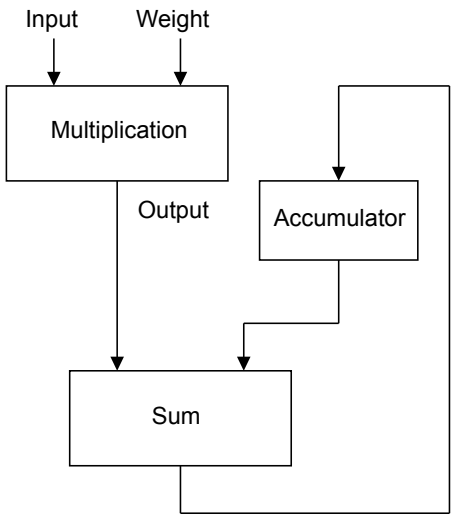
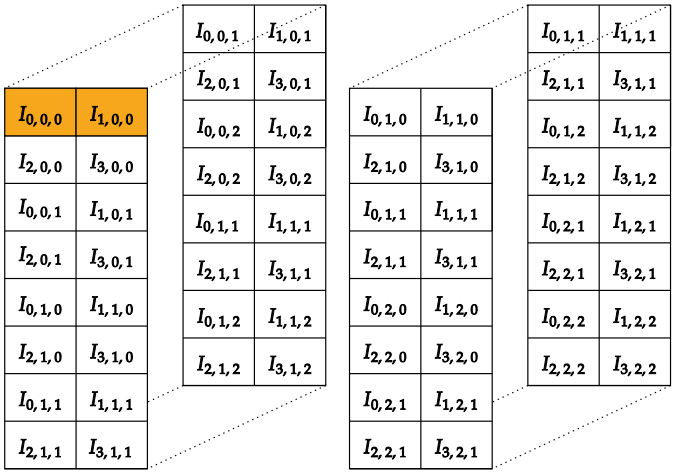


Figure 4.6: Processing Element structure

computation restarts to compute another convolution window (Figure 4.9). This process is repeated until all values have been processed and the output tensor is complete.



(a) Input tensor. The orange values are passed to both computational units

$W^0_{0,0,0}$	$W^0_{1,0,0}$	$W^1_{0,0,0}$	$W^1_{1,0,0}$
$W^0_{2,0,0}$	$W^0_{3,0,0}$	$W^1_{2,0,0}$	$W^1_{3,0,0}$
$W^0_{0,0,1}$	$W^0_{1,0,1}$	$W^1_{0,0,1}$	$W^1_{0,1,1}$
$W^0_{2,0,1}$	$W^0_{3,0,1}$	$W^1_{2,0,1}$	$W^1_{3,0,1}$
$W^0_{0,1,0}$	$W^0_{1,1,0}$	$W^1_{0,1,0}$	$W^1_{1,1,0}$
$W^0_{2,1,0}$	$W^0_{3,1,0}$	$W^1_{2,1,0}$	$W^1_{3,1,0}$
$W^0_{0,1,1}$	$W^0_{1,1,1}$	$W^1_{0,1,1}$	$W^1_{1,1,1}$
$W^0_{2,1,1}$	$W^0_{3,1,1}$	$W^1_{2,1,1}$	$W^1_{3,1,1}$

(b) Weight Tensor. The green values are passed to the first computational unit, the yellow ones to the second one

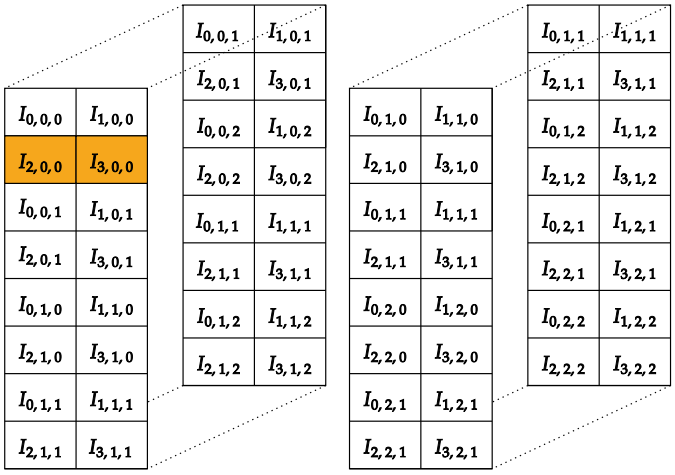
Figure 4.7: First iteration of the computation

4.3 Fault Injector

As introduced in chapter 1 , it is crucial to evaluate the accuracy of a neural network model in adversarial conditions. The design of this Fault Injector is mainly focused on soft-errors, such as bit-flips. The parameters of the Fault Injector, keeping them consistent with [28], are the following:

- **Input Faulty**: whether the input is faulty or not.
- **Weight Faulty**: whether the weights are faulty or not.
- **Bit Faulty**: the position of bit to be flipped.
- **Percentage of MAC faulty**: the percentage of MAC that is faulty.
- **Frequency**: the frequency of the fault occurrence.

To better understand the use of these parameters, it's better to first dive into the design on the Fault Injector.



(a) Input tensor. The orange values are passed to both computational units

$W^0_{0,0,0}$	$W^0_{1,0,0}$	$W^1_{0,0,0}$	$W^1_{1,0,0}$
$W^0_{2,0,0}$	$W^0_{3,0,0}$	$W^1_{2,0,0}$	$W^1_{3,0,0}$
$W^0_{0,0,1}$	$W^0_{1,0,1}$	$W^1_{0,0,1}$	$W^1_{0,1,1}$
$W^0_{2,0,1}$	$W^0_{3,0,1}$	$W^1_{2,0,1}$	$W^1_{3,0,1}$
$W^0_{0,1,0}$	$W^0_{1,1,0}$	$W^1_{0,1,0}$	$W^1_{1,1,0}$
$W^0_{2,1,0}$	$W^0_{3,1,0}$	$W^1_{2,1,0}$	$W^1_{3,1,0}$
$W^0_{0,1,1}$	$W^0_{1,1,1}$	$W^1_{0,1,1}$	$W^1_{1,1,1}$
$W^0_{2,1,1}$	$W^0_{3,1,1}$	$W^1_{2,1,1}$	$W^1_{3,1,1}$

(b) Weight Tensor. The green values are passed to the first computational unit, the yellow ones to the second one

Figure 4.8: Second iteration of the computation

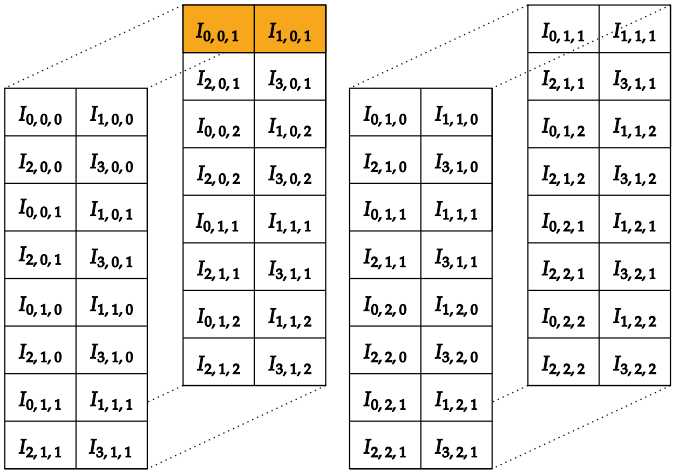
4.4 Intial Fault Injector Design

At the begging of the project, the easiest solution was to create a Fault Injector as an additional layer to the original neural network. The interface of the added layer, shown in Figure 4.10, is the following:

- **input_values**: The values computed by the previous convolutional layer.
- **frequency**: The frequency of the fault occurrence.
- **output_values**: The faulted values computed by the Fault Injector.

To clarify the setup, Figure 4.11 shows a zoom of the original model, where each layer is interconnected to the next one. After inserting the Fault Injector, the network structure is shown in Figure 4.12.

The internal of the added layer is really simple. Based on the frequency, it flips some bit of the input. Let's say the frequency is 50%, then every two bits one will be flipped.



(a) Input tensor. The orange values are passed to both computational units

$W^0_{0,0,0}$	$W^0_{1,0,0}$	$W^1_{0,0,0}$	$W^1_{1,0,0}$
$W^0_{2,0,0}$	$W^0_{3,0,0}$	$W^1_{2,0,0}$	$W^1_{3,0,0}$
$W^0_{0,0,1}$	$W^0_{1,0,1}$	$W^1_{0,0,1}$	$W^1_{0,1,1}$
$W^0_{2,0,1}$	$W^0_{3,0,1}$	$W^1_{2,0,1}$	$W^1_{3,0,1}$
$W^0_{0,1,0}$	$W^0_{1,1,0}$	$W^1_{0,1,0}$	$W^1_{1,1,0}$
$W^0_{2,1,0}$	$W^0_{3,1,0}$	$W^1_{2,1,0}$	$W^1_{3,1,0}$
$W^0_{0,1,1}$	$W^0_{1,1,1}$	$W^1_{0,1,1}$	$W^1_{1,1,1}$
$W^0_{2,1,1}$	$W^0_{3,1,1}$	$W^1_{2,1,1}$	$W^1_{3,1,1}$

(b) Weight Tensor. The green values are passed to the first computational unit, the yellow ones to the second one

Figure 4.9: n^{th} iteration of the computation

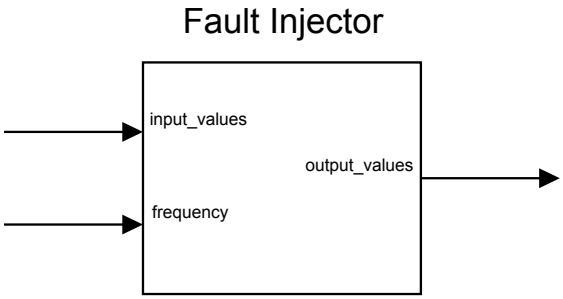


Figure 4.10: Fault Injector interface

4.4.1 Adding a new layer in FINN

To exploit the flexibility and integration with FINN, the insertion of the Fault Injector in the network is done coherently with the FINN flow.

The flow, as already described, is composed of various steps. Inserting a new layer in the network is as easy as adding a new phase in the flow. The added step receives the ONNX model of the network, that is modified as needed. To insert a custom layer define a new class that inherits from `HLSCustomOp`. It is essential to override some original class methods to obtain the expected behaviour when generating the HLS code. For simplicity, only the most important are listed here.

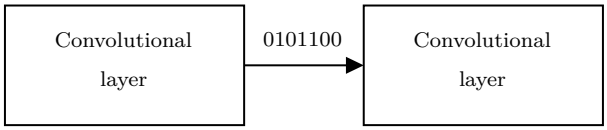


Figure 4.11: Layers without Fault Injector

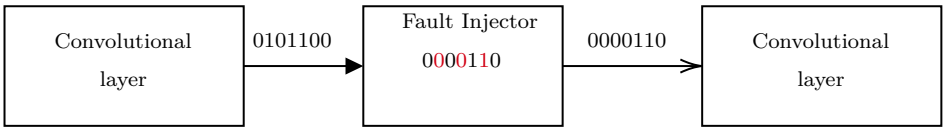


Figure 4.12: Initial Fault Injector design

The complete list of methods is visible by reverse engineering the original class.

- **get_nodeattr_types**: this method returns a dictionary containing the node's attributes. Some of them are very important, such as PE and SIMD, that define the parallelism of the layer.
- **defines**: this method writes in the output dictionary the `#define` directives that will be used in the HLS code.
- **pragmas**: this method writes in the output dictionary the `#pragma` directives that will be used in the HLS code.
- **blackbox_function**: this method writes the top-level function of the HLS code in the output dictionary.
- **docompute**: this method writes the call to the *finn-hlslib* function that implements the layer in the output dictionary.

4.4.2 Problems of the Initial Design

It's clear from the previous section that implementing this design is pretty simple. However, some major drawbacks make this design not suitable for real-world applications.

- **Network alteration**: the network is altered by adding one or more layers and this may cause a very high overhead in terms of latency and resources.
- **Control of the Faults**: the control of the faults is not as granular as it should be. The user can only control the frequency of the faults, but this is only one of the many parameters that are suitable for a complete fault injection [28] (refer to the bullet points in section 4.3)

4.5 Final Fault Injector design

The final design is not only an improvement of the initial design but a complete redesign. The main difference is that the fault injector is not a standalone layer anymore but an extension of the original convolutional layer. This means the network is no longer altered structure-wise, and the user can control the faults more granularly.

4.5.1 MatrixVectorActivation

For this purpose, the `MatrixVectorActivation` has been modified and adapted. Listing 8 shows the interface of the original `MatrixVectorActivation` module, while Listing 9 shows the interface of the custom `MatrixVectorActivation` that supports faults. In the next sections, all the custom `MatrixVectorActivation` parameters are explained.

4.5.2 Parameters

In this section, the additional parameters of Listing 9 with respect to Listing 8 are explained.

- **is_input_faulty**: this parameter enables the fault on the layer's input. If it is set to 1, the input is corrupted. Figure 4.13 shows a possible hardware implementation of this module.

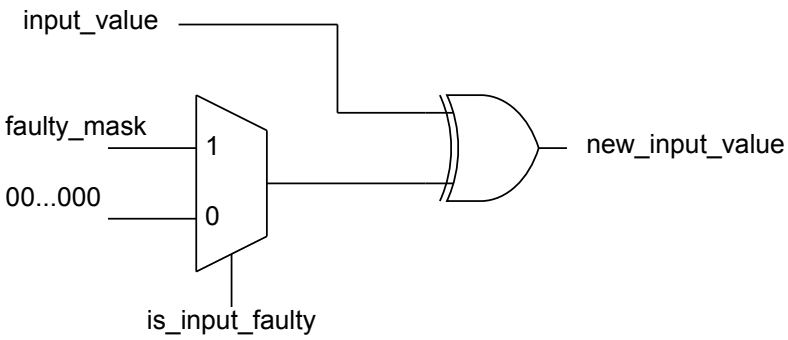


Figure 4.13: Hardware structure of `is_input_faulty parameter`

- **is_weight_faulty**: this parameter enables the fault on the layer's weight. If it is set to 1, the weights are corrupted. Figure 4.14 shows a possible hardware implementation of this module.

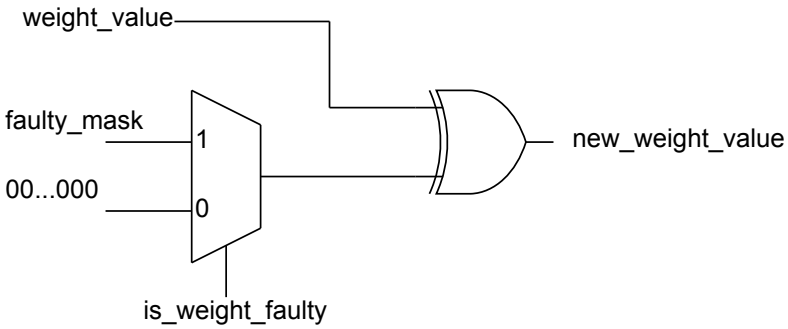


Figure 4.14: Hardware structure of `is_weight_faulty parameter`

- **mac_frequency_mask**: this parameter is used to control the frequency of the faults. It is a `NUM_BITS_FREQUENCY`-bit integer that is rotated to the right every time a MAC operation is performed. If the bit at the rightmost position (LSB) is set to 1, the MAC operation is faulty, otherwise, it is not. In chapter 5, `NUM_BITS_FREQUENCY` is set to 128. In general, the `mac_frequency_mask` can assume any value between 0 and `NUM_BITS_FREQUENCY`. The lowest possibile non-zero frequency is $\frac{1}{NUM\_BITS\_FREQUENCY}$, while the highest is $\frac{NUM\_BITS\_FREQUENCY}{NUM\_BITS\_FREQUENCY}$. Figure 4.15 shows a possibile hardware implementation of this module.

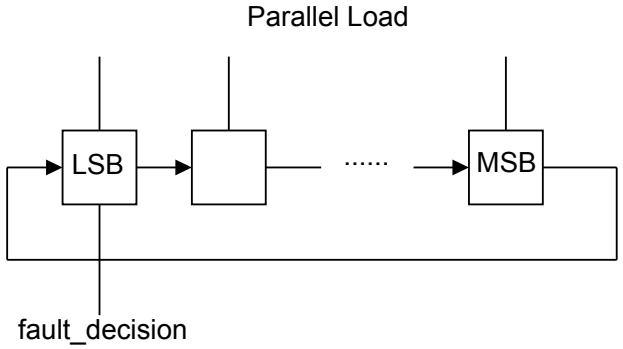


Figure 4.15: Hardware structure of `is_input_faulty parameters`

- **bit_faulty**: this parameter controls the bit to flip. It is the same for both inputs and weights, and its size is

$$max(\log_2(num\_bits\_input), \log_2(num\_bits\_weights))$$

This value is `NUM_BITS_BIT_FAULTY`-wide. Figure 4.16 shows a possible hardware implementation of this module.

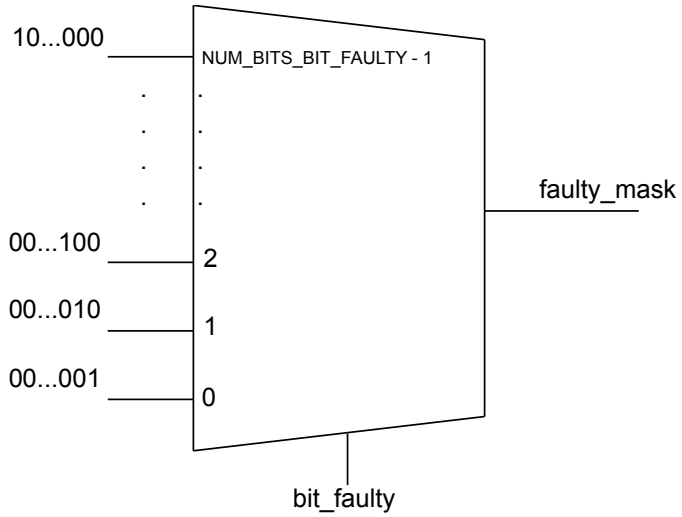


Figure 4.16: Hardware structure of `is_input_faulty parameters`

- **mac_faulty**: this parameter controls whether a specific MAC is faulty. It is a 2-dimensional array (that collapses to a 1-dimension array when one parameter between `PE` and `SIMD` is 1) of 1-bit values. If the bit is set to 1, the corresponding MAC is faulty.

Figure 4.17 shows a top-level view of the full module.

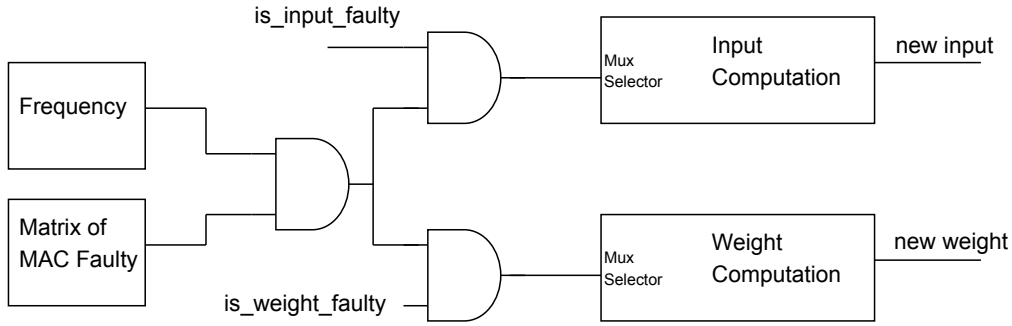


Figure 4.17: Top-level block diagram of the final fault injector implementation

To better clarify the meaning of these parameters, let's see two examples considering the following configuration for a convolutional layer.

- **SIMD**: 2
- **PE**: 2

- **num_bits_input**: the number of bits needed to represent the input values. In this case, it is 8.
- **num_bits_weights**: the number of bits needed to represent the values of the weights. In this case, it is 8.
- **is_input_faulty**: 1
- **is_weight_faulty**: 1
- **mac_frequency_mask**: defined, for simplicity, on 8 bits, with a value of 11111111
- **bit_faulty**: 1
- **mac_faulty**: $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{matrix} SIMD0 \\ SIMD1 \end{matrix}$ $\begin{matrix} PE0 & PE1 \end{matrix}$

In this case, the input and weights are both faulty. So let's imagine that the computational unit represented by PE0 and SIMD0 (position (0,0) of the `mac_faulty` matrix) receives 24 as input and 2 as weights. Let's convert these values to binary on 8 bits:

$$24 = 00011000$$
$$2 = 00000010$$

The value of `mac_faulty` at the position (0,0) is 0. This means that the operation will be executed normally. So the result of the MAC operation is:

$$24 \cdot 2 = 48$$

Now, keeping the same value of input and weights for simplicity, let's consider the computational unit represented by PE0 and SIMD1 (position (0,1)). In this case, the value of `mac_faulty` at the position (0,1) is 1. This means that the operation may be faulty and depends on the value of `mac_frequency_mask`. In this case, the value of `mac_frequency_mask` is 11111111, which means that *every* MAC operation is faulty. Since `is_input_faulty` and `is_weight_faulty` are both 1, the value of input and weights are faulted accordingly to the value of `bit_faulty`. In this case, `bit_faulty` is 1, so the LSB - 1 flipped. Therefore the value of input and weights are:

$$26 = 00011010$$
$$0 = 00000000$$

The result of the MAC operation is:

$$26 \cdot 0 = 0$$

To understand the real usage of `mac_frequency_mask`, let's continue the example keeping the same configuration but changing the value of `mac_frequency_mask` to 00000001. In this case, only *one operation out of eight* executed by each MAC is faulty. For this variation of the example, it is crucial to consider two iterations, where in every iteration each MAC executes one operation.

- **Iteration 1**: the value of `mac_frequency_mask` is 00000001. This means that the first operation executed by each MAC is faulty because the LSB is 1. Considering the computational unit represented by PE0 and SIMD1, the result will be 0, as already shown above.
- **Between Iteration 1 and Iteration 2**: the value of `mac_frequency_mask` is rotated (it is not important the direction of the rotation). For this example, the rotation is executed to the right, so the value of `mac_frequency_mask` becomes 10000000 , as shown in Equation 4.1:

$$00000001 \rightarrow 10000000 \tag{4.1}$$

- **Iteration 2**: the value of `mac_frequency_mask` is 10000000 . This means that the second operation executed by each MAC is not faulty because the LSB is 0.

4.6 Fault Injection code

Now that all the parameters are defined and clearly explained, let's see the actual code implementation. There have been many iterations of the code, but for simplicity, only the best one is reported. At the end of the section, where the results in terms of area overhead are shown, there will be a comparison between the first version of the Fault Injector (the worst one) and the one explained (the best one). Since the code is long and complex, only the core parts are reported.

4.6.1 Matrix Vector Activation

The Matrix Vector Activation Unit, as already explained, is the module that effectively executes the computations behind the convolution. Before going into the explanation of the modified code, let's analyze the logic behind the original code. The code is shown in Listing 2.

```
// compute matrix-vector product for each processing element
for(unsigned  pe = 0; pe < PE; pe++) {
#pragma HLS UNROLL
    auto const  act = TSrcI()(inElem, 0);
    auto const  wgt = TWeightI()(w[pe]);
    //auto const  wgt = w[pe];
    accu[0][pe] = mac<SIMD>(accu[0][pe], wgt, act, r, 0);
}
```

Listing 2: Part of the code of the original Matrix Vector Activation Unit

The loop iterates over all the PEs and for each one of them, it executes the MAC operation. The most important parameters of the `mac` are:

- **accu[0][pe]**: the accumulator of the MAC operation.
- **wgt**: the weight of the MAC operation.
- **act**: the input of the MAC operation.
- **r**: the type of resource used for the MAC operation. There are three types of resources:
  1. **ap_resource_lut**: force HLS to implement the multiplier in LUTs
  2. **ap_resource_dsp**: force HLS tool to implement the multiplier in DSP48
  3. **ap_resource_dflt**: let the HLS tool choose the best one

It's important to notice the use of `#pragma`. Citing the Xilinx Documentation [41]:

> *The HLS tool provides **pragmas** that can be used to optimize the design, reduce latency, improve throughput performance, and reduce area and device resource usage of the resulting RTL code*

In most of the pieces of code shown in this section, there is the use of the `#pragma HLS UNROLL` [42]. This directive is used to exploit the unrolling technique, which is a technique that allows the compiler to expand the loop into a sequence of instructions. This may seem counterintuitive since it increases the code size. However, in the hardware implementation, these operations can be parallelized and drastically improve the throughput. The loop bounds must be known at compile time to unroll it completely.

4.6.2 Multiply Accumulate

The (MAC) function is really simple. It is composed of a for-loop that iterates over the number of SIMDs and for each one of them, it calls the `mul` function, which executes the multiplication of the weight (represented by `c`) and the input (represented by `d`) and then it adds the result to the accumulator. The code is shown in Listing 3.

```
template<
    unsigned N,
    typename T,
    typename TC,
    typename TD,
    typename R
>
T mac(
    T const &a,
    TC const &c,
    TD const &d,
    R const &r,
    unsigned mmv
) {
#pragma HLS inline
  T  res = a;
  for(unsigned  i = 0; i < N; i++) {
#pragma HLS unroll
    res += mul(c[i], d(i,mmv), r);
  }
  return  res;
}
```

Listing 3: Original Multiply Accumulate (MAC) function

The `#pragma HLS inline` [43] directive is used to force the compiler to inline the function. This means that the compiler will replace the function call with the function body. This is done to reduce the number of function calls and to increase the performance of the code, but can also increase the area overhead of the RTL implementation.

4.6.3 Multiply

The Multiply (MUL) function is the one that executes the multiplication of the weight and the input. The code is shown in Listing 4. The `#pragma HLS BIND_OP`

[44] specifies that for a specific **variable**, a specific **operation (op)** should be mapped to a specific device resource for **implementation (impl)** in the RTL.

```
template<
    typename TC,
    typename TD
>
auto mul(
    TC const &c,
    TD const &d,
    ap_resource_lut const&
) -> decltype(c*d) {
#pragma HLS inline
    decltype(c*d) const  res = c*d;
#pragma HLS BIND_OP variable=res op=mul impl=fabric
    return  res;
}
```

Listing 4: Original Multiply (MUL) function

4.6.4 Modified code

The interface of the custom Matrix Vector Activation Unit (MVAU) is shown in Listing 9. What may look strange is the definition of the `mac_faulty`, that is a 2D array of size (PE, SIMD). After some test, it was found that the best implementation in terms of LUTs usage is the 2D array, instead of the more straightforward 1D array. The overhead of selecting the correct element from the 1D array is higher than the reduction in LUTs obtained using an `ap_unit<PE*SIMD>`.

Therefore, by using an `ap_uint<1> mac_faulty [PE][SIMD]` the indexing is done really easily, by using `mac_faulty[pe]`. The other parameters that characterize a fault are propagated to the `mac` function, as shown in Listing 5.

```
for(unsigned  pe = 0; pe < PE; pe++) {
#pragma HLS UNROLL
    auto const  act = TSrcI()(inElem, 0);
    auto const  wgt = TWeightI()(w[pe]);
    accu[0][pe] = mac<SIMD, NUM_BITS_BIT_FAULTY>(
        accu[0][pe],
        wgt,
        act,
        r,
        0,
        is_input_faulty,
        is_weight_faulty,
        mac_frequency_mask[0],
        bit_faulty,
        mac_faulty[pe]
    );
}
```

Listing 5: Part of the code of the custom Matrix Vector Activation Unit

The `mac` function is shown in Listing 11. Also in this case, the implementation is straightforward, since all the necessary parameters are propagated to the `mul` function. The main reason behind the decision of moving all the computational complexity to the deepest function is to execute the most expensive operations with the smallest possible parallelism. This is done to reduce the number of hardware resources (LUTs and DSPs) used in the implementation.

The function where the faults are injected and the computations are executed is the `mul`, whose code is shown in Listing 6. To better explain the flow, let's adapt (keeping the same configuration) the example made in subsection 4.5.2 to be coherent with the code nomenclature.

Let's consider the computation unit represented by PE0 and SIMD0. The value received by the function are the following:

- **c (weight)** = 2
- **d (input)** = 24
- **is_input_faulty** = 1
- **is_weight_faulty** = 1

- **bit_faulty** = 1
- **mac_faulty** = 0
- **mac_frequency_mask** = 1

The first thing to do is to check, based on the `mac_frequency_mask` value, if the MAC is faulty. This value is saved in the `mac_faulty_frequency_mask` variable, which must be anded with `is_input_faulty` and `is_weight_faulty` to check if the input and the weight are faulty. If, after all this computation, the obtained value is 1, then the MAC is operation must be faulted. To manage the fault, which is effectively a bit, the decision is to use a XOR operation (truth table in Table 4.1). In fact, xoring a value with 1 flips the value, while xoring a value with 0 does not change it. After all these computations, we obtained the new value for both the weights and the activations. The final result is the multiplication of the two values, which the function returns.

Input 1	**Input 2**	**Output**
0	0	0
0	1	1
1	0	1
1	1	0

Table 4.1: XOR table

Let's try to follow the execution by replacing the variable names with their values:

$$
\begin{aligned}
mac\_faulty\_frequency\_mask &= 0 \ AND \ 1 = 0 \\
new\_c[bit\_faulty] &= 1 \ XOR \ (1 \ AND \ 0) = 1 \\
new\_d[bit\_faulty] &= 0 \ XOR \ (1 \ AND \ 0) = 0 \\
new\_c &= 2 \\
new\_d &= 24 \\
res &= 2 \cdot 24 = 48
\end{aligned}
$$

```
template <
    unsigned NUM_BITS_BIT_FAULTY,
    typename TC,
    typename TD
>
auto mul(
    TC const &c,
    TD const &d,
    ap_resource_dsp const &,
    const ap_uint<1> &is_input_faulty,
    const ap_uint<1> &is_weight_faulty,
    const ap_uint<NUM_BITS_BIT_FAULTY> &bit_faulty,
    const ap_uint<1> &mac_faulty,
    const ap_uint<1> mac_frequency_mask
) -> decltype(c * d) {
#pragma HLS inline
    ap_uint<1> mac_faulty_frequency_mask = mac_faulty &
    mac_frequency_mask;
    TC new_c = c;
    new_c.set_bit(bit_faulty, c[bit_faulty] ^ (is_weight_faulty &
    mac_faulty_frequency_mask));
    TD new_d = d;
    new_d.set_bit(bit_faulty, d[bit_faulty] ^ (is_input_faulty &
    mac_faulty_frequency_mask));
    decltype(new_c * new_d) res = new_c * new_d;
#pragma HLS BIND_OP variable = res op = mul impl = dsp
    return res;
}
```

Listing 6: Custom Multiply (MUL) function

The last modification done to the code is the management of the frequency. The implementation is based on inferring a shift register, as explained in the Vitis HLS Documentation [45]. The code is shown in Listing 7.

```
// Rotate the mac_frequency_mask
ap_uint<1> temp = mac_frequency_mask[0];
for (int i = 0; i < NUM_BITS_FREQUENCY - 1; ++i)
#pragma HLS unroll
    mac_frequency_mask[i] = mac_frequency_mask[i+1];
mac_frequency_mask[NUM_BITS_FREQUENCY - 1] = temp;
```

Listing 7: Code to rotate the mac_frequency_mask

Chapter 5

Experiments

This section is dedicated to explaining the experiments and presenting the interesting results obtained. To keep this thesis related to real-world application, all the experiments have been done on a neural network called *Mobilenet v1* [46] . More in general, Mobilenets are a class of neural networks mainly focused on mobile and embedded computer vision [46]. The structure of Mobilenet v1 is shown in Table B.1.

5.1 Per-layer Overhead

Before moving on to a full network, it is important to analyze the resource overhead of a single layer. Table 5.1 shows the resource usage of *conv7* (refer to Table B.2) of Mobilenet-v1. faulted and not faulted. Table 5.2 shows the comparison of the faulted and not faulted *conv6* (refer to Table B.2). This layer implements the depthwise convolution .

Layer Type	PE	SIMD	Latency (ns)	FF	LUT
Original Layer	16	8	$2.168e+06$	1408	5269
Faulted Layer	16	8	$2.168e+06$	2329	8339

Table 5.1: Matrix Vector Activation Overhead

Layer Type	PE	SIMD	Latency (ns)	FF	LUT
Original Layer	16	1	$1.219e+06$	579	1430
Faulted Layer	16	1	$1.219e+06$	1103	1944

Table 5.2: Vector Vector Activation Overhead

For MVAU the overhead in terms of FF is around 65.4%. In terms of LUT the increase is around 58.3%. On the other hand, VVAU shows an increase of 90.5% in terms of FF and 35.9% in terms of LUT. These results may not look very convincing, since the overheais pretty high. The reason of this increase is mainly due to the addition of 5 parameters, and the logic to control them. . However, as explained in the following section, they allowed building *Mobilenet-v1* with little parallelism reduction (see Figure B.3 and Figure B.4).

5.2 Full Network Overhead

Firstly a reference mobilenet-v1 build has been done. The starting pretrained model is available in the brevitas repository. Therefore, following the instruction of the finn-examples repository allowed the build of the reference model with the folding config shown in Figure B.3. Table 5.3 shows the resource usage.

Resource	Utilization	Available	Utlization %
LUT	189474	230400	82.236984
LUTRAM	9781	101760	9.611832
FF	167276	460800	36.301216
BRAM	281.5	312	90.224365
URAM	69	96	71.875
DSP	50	1728	2.8935184
BUFG	26	544	4.779412

Table 5.3: Resource usage of the reference model with no fault and the original folding config

Moving on to the faulty model, keeping the same folding config , the build did not end up properly because of the need for more resources (LUT) compared to the available ones. The required resources were slightly higher, around 3% more, but no optimization has been found to fit the board. That 3% overhead requires, for the synthesis to complete, a LUT availability of 237312. Comparing the latter value with the one shown in Table 5.3, the percentage increase is around 25.2%, which is much smaller than per-layer overhead already described in section 5.1. The reason of this divergence is due to the presence of addition module used to create all the necessary interconnections to make the FPGA work properly. Figure 5.1 shows the top-level project generated by FINN. The hardware implementation of the considered network is contained inside the `StreamingDataflowPartition_1`. All the other blocks are necessary to create the correct interconnections between the SoC and the Logic, in order to move data to the accelerator and to read data after they have been processed. In addition to that, inside the `StreamingDataflowPartition_1`

between each layer there is a FF, that is added by FINN as already explained in section 4.2.1

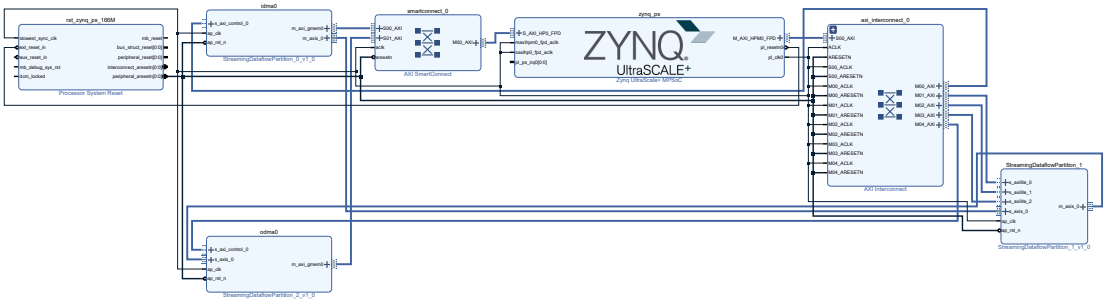


Figure 5.1: Top Level FINN Project (Zoom in **??**)

However, it is a pretty good improvement compared to the initial version of the *Fault Injector*, where the same faulty build required double the available resources (around 100% more). With more time available and further investigation, an optimization to reduce that 3% is perfectly doable. To bypass this problem, a new folding config, showed in Figure B.4) has been generated to lower the resources to fit the faulty model. The resource usage of the reference model (no fault) with the new folding config are reported in Table 5.4. On the other hand, Table 5.5 shows the resource usage of the model with all layer faulty with the custom folding config.

Resource	Utilization	Available	Utlization %
LUT	167549	230400	72.720924
LUTRAM	9781	101760	9.611832
FF	147533	460800	32.01671
BRAM	267.5	312	85.73718
URAM	69	96	71.875
DSP	50	1728	2.8935184
BUFG	11	544	2.022059

Table 5.4: Resource usage of the reference model with no fault and the new folding config

The biggest resource overhead is in the LUT and it is about 18%. However, the faulty model is set up to have all layers faulty so that the end user can decide at runtime if a specific layer should be correct or faulty. If it is decided to be faulty, the parameters explained in subsection 4.5.2 can be set. This setup, also if the resource overhead in terms of LUT and FF is not negligible, allows synthesizing the model only once and then be able to test different configurations at runtime. To make everything as flexible as possible in the case of a smaller FPGA with

Resource	Utilization	Available	Utlization %
LUT	207926	230400	90.24566
LUTRAM	10465	101760	10.284001
FF	175272	460800	38.03646
BRAM	282.5	312	90.544876
URAM	69	96	71.875
DSP	50	1728	2.8935184
BUFG	10	544	1.838235

Table 5.5: Resource usage of the faulted model with all layer faulty with the new folding config

fewer resources, it is possible to decide **before** the synthesis, which layer should be faulty. By doing this, the FINN compiler inserts the normal layer instead of the one with the fault support, consistently reducing the resource overhead as already explained in section 5.1. This guarantees a modular resource usage based on hardware availability. The resource usage is only one of the metrics to take into consideration when comparing the modified design with the original one. Another key metric is the computational performance, measured in images processed in one second. The final version of the *Fault Injector* has been designed in such a way to minimize as much as possible any performance degradation. As it is visible in Table 5.6 and Table 5.7 the performance in terms of throughtput are almost identical, as well as other metrics, which are all within a narrow margin of error (the slight variation of some parameters may be due to the non-deterministic performance of any computational device). What is interesting is the fact that the performance of the models with the Figure B.4 are really similar compared to Figure B.3, which means that the little parallelism reduction is not significant performance-wise.

5.3 Bit Impact

The possibility of the designed *Fault Injector* to decide which bit should be flipped allowed the verification of the **Bit Sensitivity**. As already observed by the authors in [28], increasing the importance of the flipped bit (such as flipping the MSB instead of the LSB) decreases the network accuracy. This is a reasonable behaviour, since the higher the importance of the bit (closer to MSB), the higher the error introduced. Figure 5.2 shows an example of a the accuracy degradation with the bit importance increase. The configuration, reported in the image's caption, is applied identically to all layers.

runtime[ms]	495.8786964416504
throughput[images/s]	201.66222247010143
DRAM_in_bandwidth[MB/s]	30.355811023979427
DRAM_out_bandwidth[MB/s]	0.0020166222247010143
fclk[mhz]	99.999
batch_size	100
fold_input[ms]	0.1327991485595703
pack_input[ms]	0.08940696716308594
copy_input_data_to_device[ms]	16.594886779785156
copy_output_data_from_device[ms]	0.2257823944091797
unpack_output[ms]	109.7860336303711
unfold_output[ms]	0.04792213439941406

Table 5.6: Throughput performance of faulted *Mobilenet-v1* with Figure B.4

runtime[ms]	495.87321281433105
throughput[images/s]	201.6644525572363
DRAM_in_bandwidth[MB/s]	30.356146714535665
DRAM_out_bandwidth[MB/s]	0.002016644525572363
fclk[mhz]	99.999
batch_size	100
fold_input[ms]	0.11563301086425781
pack_input[ms]	0.08177757263183594
copy_input_data_to_device[ms]	14.848709106445312
copy_output_data_from_device[ms]	0.16808509826660156
unpack_output[ms]	107.24759101867676
unfold_output[ms]	0.037670135498046875

Table 5.7: Throughput performance of not faulted *Mobilenet-v1* with Figure B.4

5.4 Frequency Impact

As in the previous section, also for the frequency all other parameters have been kept constant. The considered frequency value are 1, 0.5, 0.25, 0.12, 0.062, 0.0313, 0.0156, 0.0078 . Figure 5.3 shows how the frequency impacts accuracy. It's clear that increasing the frequency, which is equal to increasing the total number of faults, decreases the accuracy.

runtime[ms]	493.9761161804199
throughput[images/s]	202.43893727743708
DRAM_in_bandwidth[MB/s]	30.47272835049805
DRAM_out_bandwidth[MB/s]	0.002024389372774371
fclk[mhz]	99.999
batch_size	100
fold_input[ms]	0.12612342834472656
pack_input[ms]	0.07796287536621094
copy_input_data_to_device[ms]	14.952421188354492
copy_output_data_from_device[ms]	0.14066696166992188
unpack_output[ms]	106.81343078613281
unfold_output[ms]	0.03504753112792969

Table 5.8: Throughput performance of not faulted *Mobilenet-v1* with Figure B.3

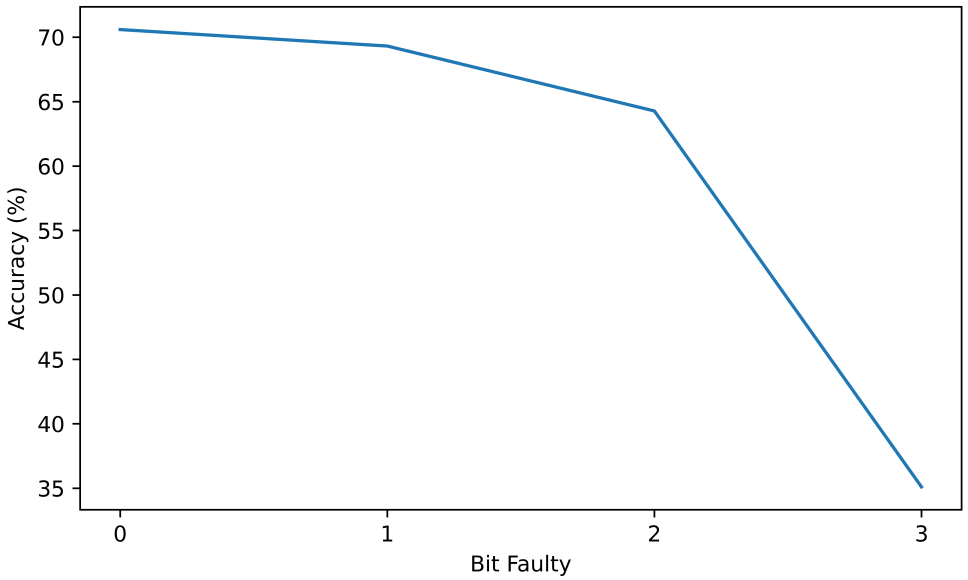


Figure 5.2: Weights faulty, frequency 0.78, percentage of MAC faulty 25%

5.5 MAC Faulty Percentage Impact

To test the impact of the percentage of the MAC faulty on the accuracy, the same methodology explained in section 5.4 has been used. All parameters, except the MAC faulty percentage, have been kept constant. The considered value are 100%, 50% and 25%. To decide which MAC are faulty, a simple randomic algorithm based on the percentage has been used. Figure 5.4 shows an example of the impact

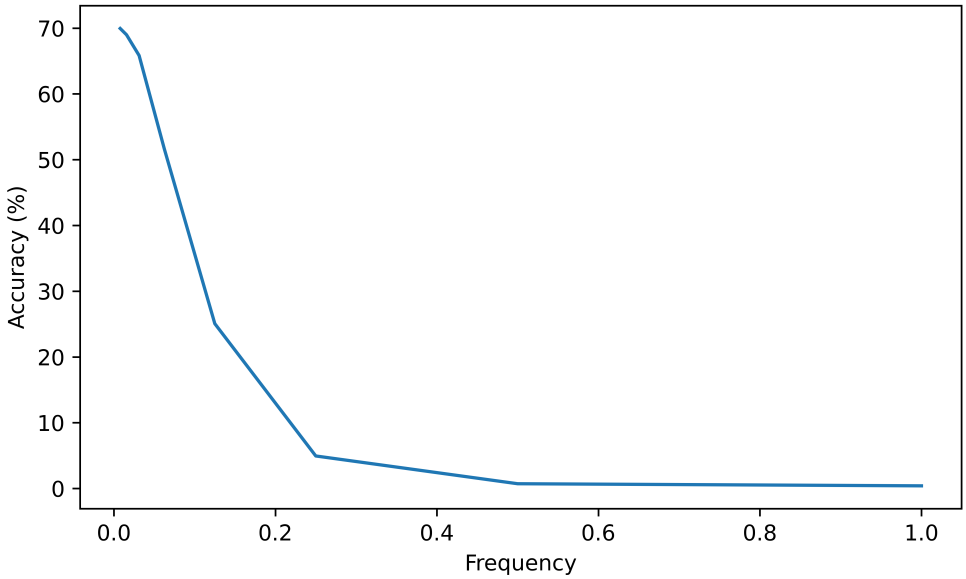


Figure 5.3: Inputs faulty, percentage of MAC faulty 25%, bit faulty 1

of the MAC faulty value. Increasing its value decreases the accuracy. In addition to that, to show the impact of the bit faulty again, each line keeps the bit constant. It's clear that the overall accuracy keeps degrading if the bit faulty is increased.

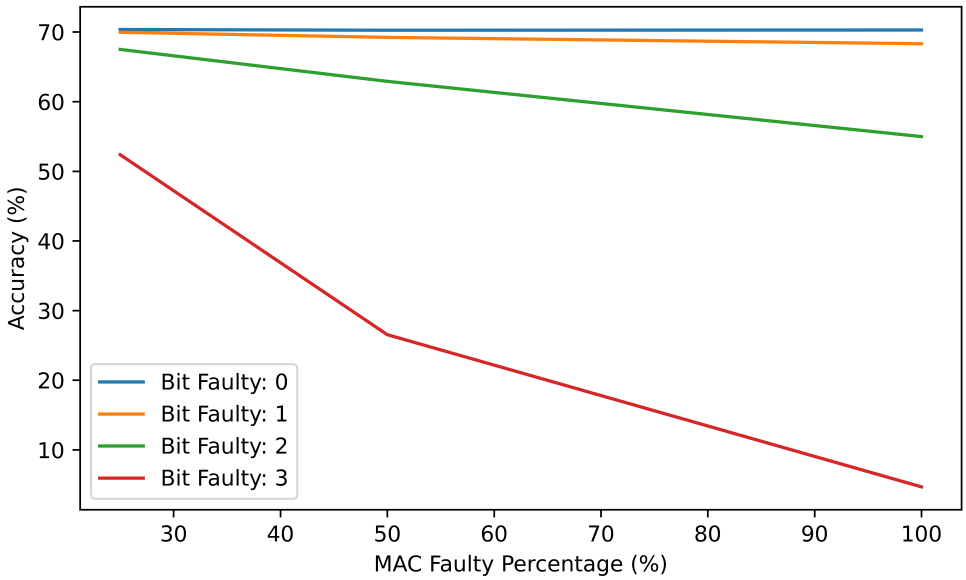


Figure 5.4: Inputs faulty, frequency 0.78

5.6 Weight Sensitivity

In [28] the authors observed that the activations had a higher impact on the accuracy degradation. They stated that

> *Flipping bits of input activations A is more likely to cause failures compared to flipping weight bits in W at any bit-flip position, on any number of multipliers and any frequency of bit-flip injection*

The failure rate is defined as follows:

Definition 5.6.1 (Failure Rate)**.** *Given the predecitions P of the* ***non-faulty*** *model and the predictions $\overset{\triangle}{P}$ of the* ***faulty*** *model, the failure rate is defined as the number of different values between P and $\overset{\triangle}{P}$ divided by the total number of predictions (size of the validation dataset)*

$$failure\_rate = \frac{CountDifference(P, \overset{\triangle}{P})}{len(validation\_dataset)}$$

Results obtained with the experiment of this thesis are partially consistent. They are coherent with any value of MAC faulty percentage if the bit faulty is 0 (see Figure 5.5 and Figure 5.6). When increasing the flipped bit, the results become inconsistent (see Figure 5.7 and Figure 5.8). A possible reason is the different architecture used in this thesis compared to the one used by the authors in [28]. However, drawing any conclusion is unsafe; further investigations must be done.

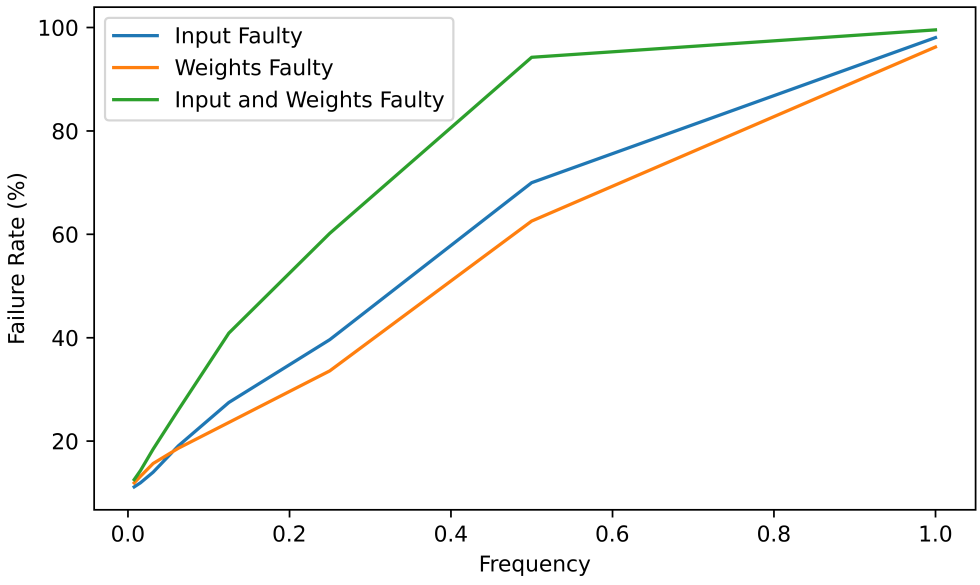


Figure 5.5: MAC faulty 25%, bit faulty 0

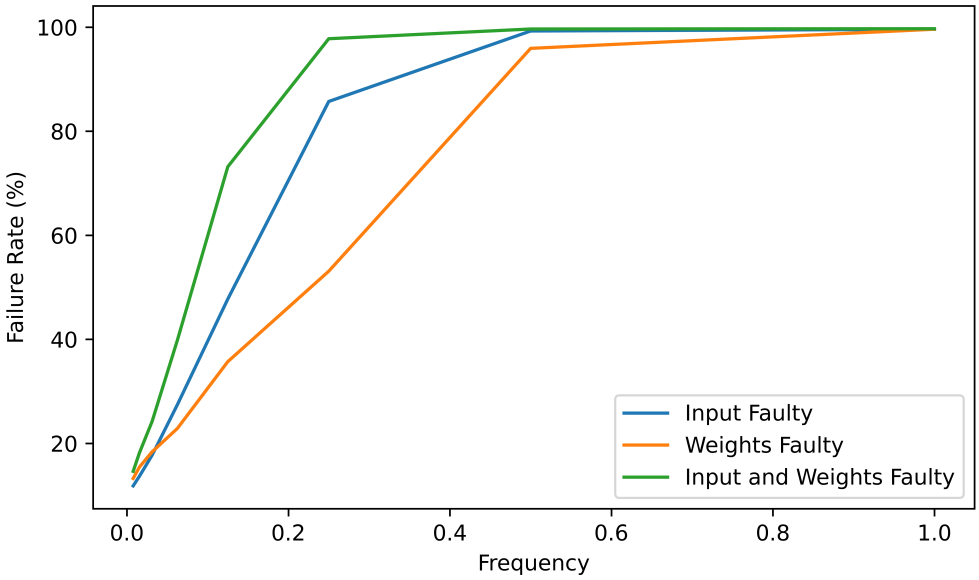


Figure 5.6: MAC faulty 50%, bit faulty 0

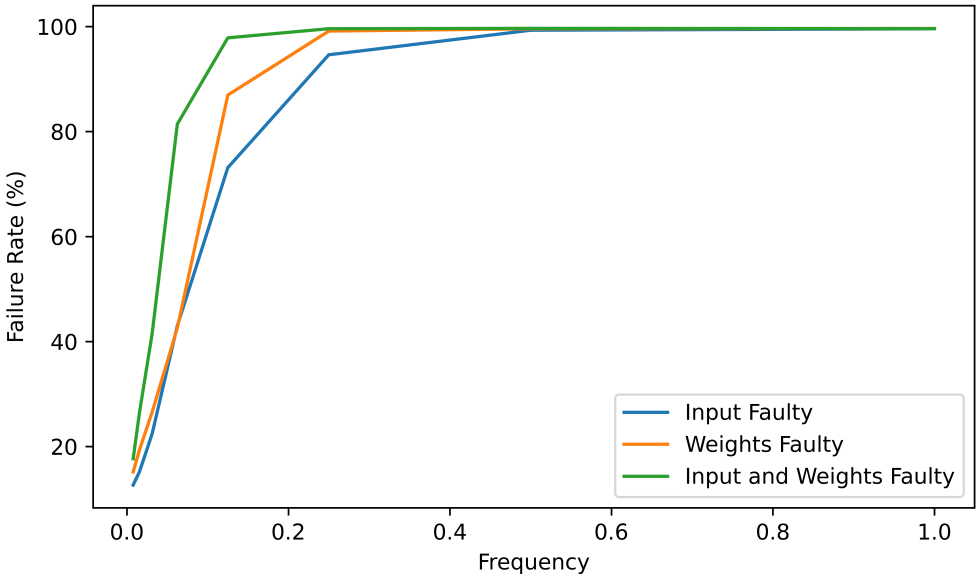


Figure 5.7: MAC faulty 25%, bit faulty 1

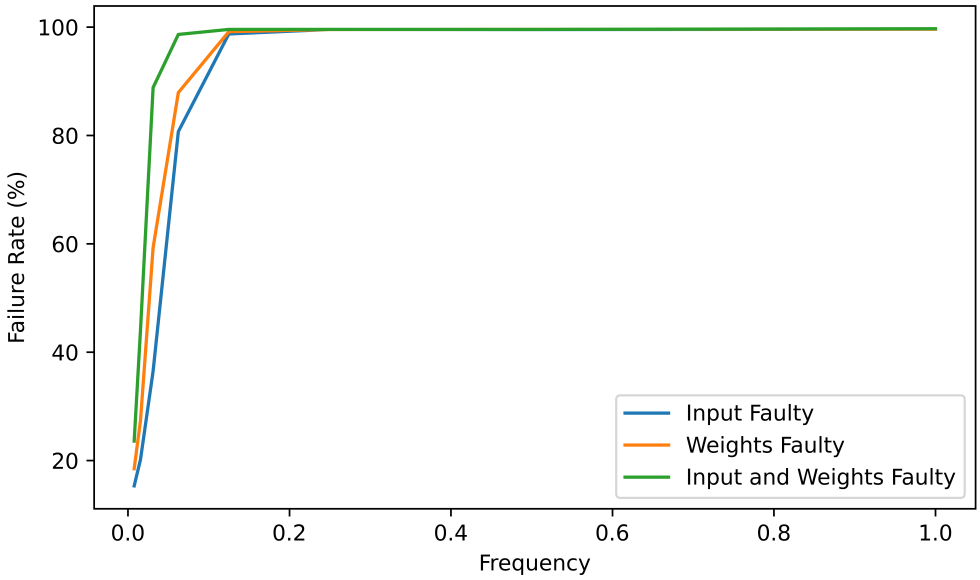


Figure 5.8: MAC faulty 50%, bit faulty 1

5.7 Loop Testing

Following what the authors did in [28], the last tests are done with a similar loop. The pseudocode is shown in Algorithm 1

Algorithm 1 Pseudocode of the loop testing

```
frequency = [0.0625, 0.03125, 0.0156, 0.0078]
type = [weights, activations]
bit = [0, 1, 2]
mac_faulty = [25%, 50%, 100%]
for f in frequency do
    for t in type do
        for b in bit do
            for m in mac_faulty do
                ApplyFaultConfiguration()
                accuracy = EvaluateAccuracy()
                failure_rate = EvaluateFailureRate()
            end for
        end for
    end for
end for
```

The obtained plot is shown in Figure 5.9

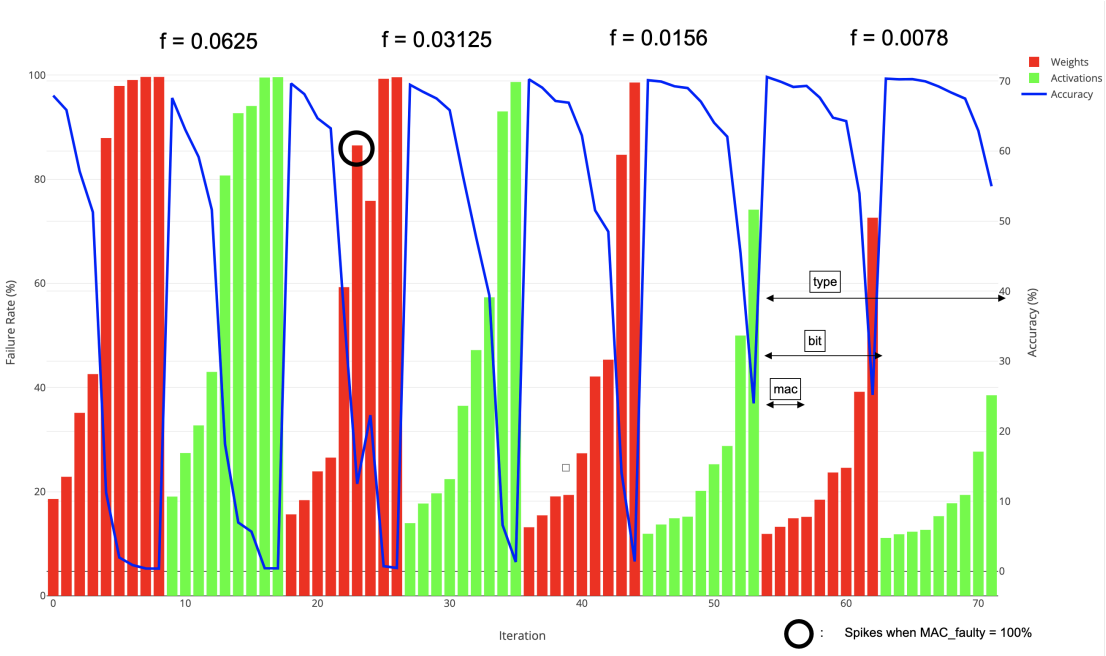


Figure 5.9: Loop testing with Algorithm 1

The graph confirms that weights are generally more sensitive compared to input. It is interesting to explain the reason behind the spike marked by the black circle. The bar in correspondence of the spike is evaluated with the configuration of Table 5.9. Conversely, the bar before the one with the spike has the configuration shown in Table 5.10, while the bar after has the configuration shown in Table 5.11. This highlights that sometimes a higher value of `MAC_faulty` with a lower bit (closer to LSB) can impact on the failure rate more compared to flipping a bit closer to MSB but with a lower value of `MAC_faulty`, considering a constant frequency.

frequency	type	bit	mac_faulty
0.03125	weights	1	100%

Table 5.9: Configuration of the network for the spike bar

frequency	type	bit	mac_faulty
0.03125	weights	1	50%

Table 5.10: Configuration of the network for the bar before the spike

frequency	type	bit	mac_faulty
0.03125	weights	2	25%

Table 5.11: Configuration of the network for the bar after the spike

Chapter 6

Conclusions and Outlook

This thesis focuses on the implementation of a module, called *Fault Injector*, to test the resilience of Neural Networks accelerators against soft errors, such as bit-flips. This thesis's novelty is using a Dataflow architecture for the NN accelerator, compared to [28] where the authors used a Systolic Array architecture. The proposed implementation in chapter 4 has been optimized to work with FINN, a framework that generates dataflow accelerators. However, the methodology can be exported and reused with any accelerator. The parameters of the *Fault Injector* allowed a deep testing, simulating situations as close as possible to real-world usage. Furthermore, the Neural Network under test is *Mobilenet-v1*, which is a really popular streamline convolutional neural network designed for mobile and embedded vision applications. Results clearly showed the importance of the position of the flipped bit, as well as the importance of the frequency of the fault and of the percentage of the faulty computational modules. Future works can span in various directions; analyzing the impact of some faults on *Mobilenet-v1* with different quantization level, such as 2 bits or 8 bits; doing a Fault Aware Training (FAT) [47] on the network and verifying if the accuracy degradation due to the same faults is lower; investigating the reason behind the higher sensitivity of the weights, differently from what observed by the authors in [28]; further optimize the *Fault Injector* to be able to build a full *Mobilenet-v1* model with the folding config of Figure B.3.

Appendix A

Listings

```
template<
    unsigned MatrixW,
    unsigned MatrixH,
    unsigned SIMD,
    unsigned PE,
    typename TSrcI = Identity,
    typename TDstI = Identity,
    typename TWeightI = Identity,
    typename TW,
    typename TI,
    typename TO,
    typename TA,
    typename R
>
void Matrix_Vector_Activate_Stream_Batch(
    hls::stream<TI> &in,
    hls::stream<TO> &out,
    hls::stream<ap_uint<PE*SIMD*TW::width>> &weight,
    TA  const &activation,
    int const  reps,
    R const &r
)
```

Listing 8: Original MatrixVectorActivation

```
template<
    unsigned MatrixW,
    unsigned MatrixH,
    unsigned SIMD,
    unsigned PE,
    unsigned NUM_BITS_BIT_FAULTY,
    unsigned NUM_BITS_FREQUENCY,
    typename TSrcI = Identity,
    typename TDstI = Identity,
    typename TWeightI = Identity,
    typename TW,
    typename TI,
    typename TO,
    typename TA,
    typename R
>
void Matrix_Vector_Activate_Stream_Batch(
    hls::stream<TI> &in,
    hls::stream<TO> &out,
    hls::stream<ap_uint<PE*SIMD*TW::width>> &weight,
    TA  const &activation,
    int const  reps,
    R const &r,
    const ap_uint<1> &is_input_faulty,
    const ap_uint<1> &is_weight_faulty,
    ap_uint<NUM_BITS_FREQUENCY> &mac_frequency_mask,
    const ap_uint<NUM_BITS_BIT_FAULTY> &bit_faulty,
    const ap_uint<1> mac_faulty[PE][SIMD]
)
```

Listing 9: Modified MatrixVectorActivation interface

```
streamline_transformations = [
    ConvertSubToAdd(),
    ConvertDivToMul(),
    BatchNormToAffine(),
    ConvertSignToThres(),
    MoveMulPastMaxPool(),
    MoveScalarLinearPastInvariants(),
    AbsorbSignBiasIntoMultiThreshold(),
    MoveAddPastMul(),
    MoveScalarAddPastMatMul(),
    MoveAddPastConv(),
    MoveScalarMulPastMatMul(),
    MoveScalarMulPastConv(),
    MoveAddPastMul(),
    CollapseRepeatedAdd(),
    CollapseRepeatedMul(),
    MoveMulPastMaxPool(),
    AbsorbAddIntoMultiThreshold(),
    FactorOutMulSignMagnitude(),
    AbsorbMulIntoMultiThreshold(),
    Absorb1BitMulIntoMatMul(),
    Absorb1BitMulIntoConv(),
    RoundAndClipThresholds(),
]
for trn in streamline_transformations:
    model = model.transform(trn)
    model = model.transform(RemoveIdentityOps())
    model = model.transform(GiveUniqueNodeNames())
    model = model.transform(GiveReadableTensorNames())
    model = model.transform(InferDataTypes())
```

Listing 10: Python code of the Streamline step

```
template <
    unsigned N,
    unsigned NUM_BITS_BIT_FAULTY,
    typename T,
    typename TC,
    typename TD,
    typename R
>
T mac(
    T const &a,
    TC const &c,
    TD const &d,
    R const &r,
    unsigned mmv,
    const ap_uint<1> &is_input_faulty,
    const ap_uint<1> &is_weight_faulty,
    const ap_uint<1> &mac_frequency_mask,
    const ap_uint<NUM_BITS_BIT_FAULTY> &bit_faulty,
    const ap_uint<1> mac_faulty[]
) {
#pragma HLS inline
    T res = a;
    for (unsigned i = 0; i < N; i++) {
#pragma HLS unroll
        res += mul<NUM_BITS_BIT_FAULTY>(
            c[i],
            d(i, mmv),
            r,
            is_input_faulty,
            is_weight_faulty,
            bit_faulty,
            mac_faulty[i],
            mac_frequency_mask
        );
    }
    return res;
}
```

Listing 11: Custom Multiply Accumulate (MAC) function

Appendix B

Tables

Type / Stride	Filter / Shape	Input Size
Conv / s2	$3 \times 3 \times 3 \times 32$	$224 \times 224 \times 3$
Conv dw / s1	$3 \times 3 \times 32$ dw	$112 \times 112 \times 32$
Conv / s1	$1 \times 1 \times 32 \times 64$	$112 \times 112 \times 32$
Conv dw / s2	$3 \times 3 \times 64$ dw	$112 \times 112 \times 64$
Conv / s1	$1 \times 1 \times 64 \times 128$	$56 \times 56 \times 64$
Conv dw / s1	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 128$	$56 \times 56 \times 128$
Conv dw / s2	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 256$	$28 \times 28 \times 128$
Conv dw / s1	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 256$	$28 \times 28 \times 256$
Conv dw / s2	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 512$	$14 \times 14 \times 256$
5× Conv dw / s1	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
5x Conv / s1	$1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$
Conv dw / s2	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
Conv / s1	$1 \times 1 \times 512 \times 1024$	$7 \times 7 \times 512$
Conv dw / s2	$3 \times 3 \times 1024$ dw	$7 \times 7 \times 1024$
Conv / s1	$1 \times 1 \times 1024 \times 1024$	$7 \times 7 \times 1024$
Avg Pool / s1	Pool 7×7	$7 \times 7 \times 1024$
FC / s1	1024×1000	$1 \times 1 \times 1024$
Softmax / s1	Classifier	$1 \times 1 \times 1000$

Table B.1: Mobilenet v1 Structure

Type / Stride	Filter / Shape	Input Size	Mapping
Conv / s2	$3 \times 3 \times 3 \times 32$	$224 \times 224 \times 3$	conv1
Conv dw / s1	$3 \times 3 \times 32$ dw	$112 \times 112 \times 32$	conv2
Conv / s1	$1 \times 1 \times 32 \times 64$	$112 \times 112 \times 32$	conv3
Conv dw / s2	$3 \times 3 \times 64$ dw	$112 \times 112 \times 64$	conv4
Conv / s1	$1 \times 1 \times 64 \times 128$	$56 \times 56 \times 64$	conv5
Conv dw / s1	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$	conv6
Conv / s1	$1 \times 1 \times 128 \times 128$	$56 \times 56 \times 128$	conv7
Conv dw / s2	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$	conv8
Conv / s1	$1 \times 1 \times 128 \times 256$	$28 \times 28 \times 128$	conv9
Conv dw / s1	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$	conv10
Conv / s1	$1 \times 1 \times 256 \times 256$	$28 \times 28 \times 256$	conv11
Conv dw / s2	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$	conv12
Conv / s1	$1 \times 1 \times 256 \times 512$	$14 \times 14 \times 256$	conv13
Conv dw / s1	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$	conv14
Conv / s1	$1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$	conv15
Conv dw / s1	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$	conv16
Conv / s1	$1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$	conv17
Conv dw / s1	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$	conv18
Conv / s1	$1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$	conv19
Conv dw / s1	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$	conv20
Conv / s1	$1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$	conv21
Conv dw / s1	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$	conv22
Conv / s1	$1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$	conv23
Conv dw / s2	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$	conv24
Conv / s1	$1 \times 1 \times 512 \times 1024$	$7 \times 7 \times 512$	conv25
Conv dw / s2	$3 \times 3 \times 1024$ dw	$7 \times 7 \times 1024$	conv 26
Conv / s1	$1 \times 1 \times 1024 \times 1024$	$7 \times 7 \times 1024$	conv27
Avg Pool / s1	Pool 7×7	$7 \times 7 \times 1024$	
FC / s1	1024×1000	$1 \times 1 \times 1024$	conv28
Softmax / s1	Classifier	$1 \times 1 \times 1000$	

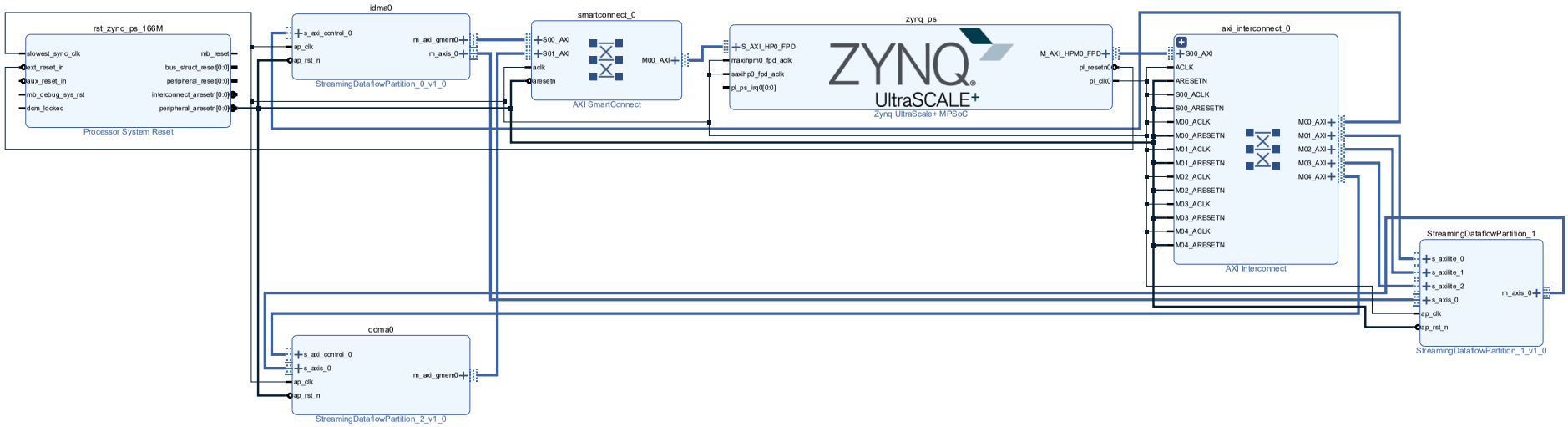
Table B.2: Mobilenet v1 Structure with FINN mapping for the experiments

Layer	PE	SIMD
conv1	16	3
conv2	16	1
conv3	8	8
conv4	8	1
conv5	16	8
conv6	16	1
conv7	32	8
conv8	4	1
conv9	16	8
conv10	8	1
conv11	32	8
conv12	2	1
conv13	16	8
conv14	4	1
conv15	32	8
conv16	4	1
conv17	32	8
conv18	4	1
conv19	32	8
conv20	4	1
conv21	32	8
conv22	4	1
conv23	32	8
conv24	1	1
conv25	16	8
conv26	2	1
conv27	32	8
conv28	1	16

Table B.3: Original Folding Config

Layer	PE	SIMD
conv1	16	3
conv2	16	1
conv3	8	8
conv4	8	1
conv5	16	8
conv6	16	1
conv7	16	8
conv8	4	1
conv9	16	8
conv10	8	1
conv11	16	8
conv12	2	1
conv13	16	8
conv14	4	1
conv15	16	8
conv16	4	1
conv17	16	8
conv18	4	1
conv19	32	8
conv20	4	1
conv21	16	8
conv22	4	1
conv23	32	8
conv24	1	1
conv25	16	8
conv26	2	1
conv27	32	8
conv28	1	16

Table B.4: Modified Folding Config


rst_zynq_ps_166M
slowest_sync_clk
ext_reset_in
aux_reset_in
mb_debug_sys_rst
dcm_locked
mb_reset
bus_struct_reset[0:0]
peripheral_reset[0:0]
interconnect_aresetn[0:0]
peripheral_aresetn[0:0]
Processor System Reset
idma0
s_axi_control_0
ap_clk
ap_rst_n
m_axi_gmem0
m_axis_0
StreamingDataflowPartition_0_v1_0
smartconnect_0
S00_AXI
S01_AXI
aclk
aresetn
M00_AXI
AXI SmartConnect
zynq_ps
S_AXI_HP0_FPD
maxihpm0_fpd_aclk
saxihp0_fpd_aclk
pl_ps_irq0[0:0]
M_AXI_HPM0_FPD
pl_resetn0
pl_clk0
ZYNQ
UltraSCALE+
Zynq UltraScale+ MPSoC
axi_interconnect_0
S00_AXI
ACLK
ARESETN
S00_ACLK
S00_ARESETN
M00_ACLK
M00_ARESETN
M01_ACLK
M01_ARESETN
M02_ACLK
M02_ARESETN
M03_ACLK
M03_ARESETN
M04_ACLK
M04_ARESETN
M00_AXI
M01_AXI
M02_AXI
M03_AXI
M04_AXI
AXI Interconnect
StreamingDataflowPartition_1
s_axilite_0
s_axilite_1
s_axilite_2
s_axis_0
ap_clk
ap_rst_n
m_axis_0
StreamingDataflowPartition_1_v1_0
odma0
s_axi_control_0
s_axis_0
ap_clk
ap_rst_n
m_axi_gmem0
StreamingDataflowPartition_2_v1_0

Bibliography

1. Umuroglu, Y., Fraser, N. J., Gambardella, G., Blott, M., Leong, P. H. W., Jahre, M. & Vissers, K. A. FINN: A Framework for Fast, Scalable Binarized Neural Network Inference. *CoRR* **abs/1612.07119.** arXiv: 1612.07119. http://arxiv.org/abs/1612.07119 (2016) (cit. on pp. 1, 17).
2. Weng, O. Neural Network Quantization for Efficient Inference: A Survey. *CoRR* **abs/2112.06126.** arXiv: 2112.06126. https://arxiv.org/abs/2112.06126 (2021) (cit. on pp. 2, 11).
3. Elallid, B. B., Benamar, N., Hafid, A. S., Rachidi, T. & Mrani, N. A Comprehensive Survey on the Application of Deep and Reinforcement Learning Approaches in Autonomous Driving. *Journal of King Saud University - Computer and Information Sciences* **34,** 7366–7390. ISSN: 1319-1578. https://www.sciencedirect.com/science/article/pii/S1319157822000970 (2022) (cit. on p. 4).
4. Jiang, J., Trundle, P. & Ren, J. Medical image analysis with artificial neural networks. *Computerized Medical Imaging and Graphics* **34,** 617–631. ISSN: 0895-6111. https://www.sciencedirect.com/science/article/pii/S0895611110000741 (2010) (cit. on p. 4).
5. Li, J. *Recent Advances in End-to-End Automatic Speech Recognition* 2022. arXiv: 2111.01690 [eess.AS] (cit. on p. 4).
6. Sultana, F., Sufian, A. & Dutta, P. Advancements in Image Classification using Convolutional Neural Network. *CoRR* **abs/1905.03288.** arXiv: 1905.03288. http://arxiv.org/abs/1905.03288 (2019) (cit. on p. 4).
7. Otter, D. W., Medina, J. R. & Kalita, J. K. A Survey of the Usages of Deep Learning in Natural Language Processing. *CoRR* **abs/1807.10854.** arXiv: 1807.10854. http://arxiv.org/abs/1807.10854 (2018) (cit. on p. 4).
8. Zou, J., Zhao, Q., Jiao, Y., Cao, H., Liu, Y., Yan, Q., Abbasnejad, E., Liu, L. & Shi, J. Q. *Stock Market Prediction via Deep Learning Techniques: A Survey* 2023. arXiv: 2212.12717 [q-fin.GN] (cit. on p. 4).

9. Singh, S. S. & Karayev, S. Full Page Handwriting Recognition via Image to Sequence Extraction. *CoRR* **abs/2103.06450.** arXiv: 2103.06450. https://arxiv.org/abs/2103.06450 (2021) (cit. on p. 4).
10. Keisler, R. *Forecasting Global Weather with Graph Neural Networks* 2022. arXiv: 2202.07575 [physics.ao-ph] (cit. on p. 4).
11. Podder, P., Bharati, S., Mondal, M. R. H., Paul, P. K. & Kose, U. Artificial Neural Network for Cybersecurity: A Comprehensive Review. *CoRR* **abs/2107.01185.** arXiv: 2107.01185. https://arxiv.org/abs/2107.01185 (2021) (cit. on p. 4).
12. Hernandez-Olivan, C., Hernandez-Olivan, J. & Beltran, J. R. *A Survey on Artificial Intelligence for Music Generation: Agents, Domains and Perspectives* 2022. arXiv: 2210.13944 [cs.AI] (cit. on p. 4).
13. Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K. & Fei-Fei, L. *ImageNet: A large-scale hierarchical image database* in *2009 IEEE Conference on Computer Vision and Pattern Recognition* (2009), 248–255 (cit. on p. 4).
14. Janiesch, C., Zschech, P. & Heinrich, K. Machine learning and deep learning. *CoRR* **abs/2104.05314.** arXiv: 2104.05314. https://arxiv.org/abs/2104.05314 (2021) (cit. on pp. 4, 5).
15. *Artificial neural network — Wikipedia, The Free Encyclopedia* https://en.wikipedia.org/w/index.php?title=Artificial_neural_network&oldid=1142333251 (cit. on p. 6).
16. Valueva, M., Nagornov, N., Lyakhov, P., Valuev, G. & Chervyakov, N. Application of the residue number system to reduce hardware costs of the convolutional neural network implementation. *Mathematics and Computers in Simulation* **177,** 232–243. ISSN: 0378-4754. https://www.sciencedirect.com/science/article/pii/S0378475420301580 (2020) (cit. on pp. 5, 6).
17. O'Shea, K. & Nash, R. An Introduction to Convolutional Neural Networks. *CoRR* **abs/1511.08458.** arXiv: 1511.08458. http://arxiv.org/abs/1511.08458 (2015) (cit. on p. 5).
18. Dumoulin, V. & Visin, F. *A guide to convolution arithmetic for deep learning* 2018. arXiv: 1603.07285 [stat.ML] (cit. on p. 6).
19. Goodfellow, I., Bengio, Y. & Courville, A. *Deep Learning* http://www.deeplearningbook.org (MIT Press, 2016) (cit. on pp. 8, 10).
20. Zhang, A., Lipton, Z. C., Li, M. & Smola, A. J. Dive into Deep Learning. *arXiv preprint arXiv:2106.11342* (2021) (cit. on p. 9).
21. *Dive into Deep Learning: Pooling* https://d2l.ai/chapter_convolutional-neural-networks/pooling.html (cit. on p. 10).

22. Nagel, M., Fournarakis, M., Amjad, R. A., Bondarenko, Y., van Baalen, M. & Blankevoort, T. A White Paper on Neural Network Quantization. *CoRR* **abs/2106.08295.** arXiv: 2106.08295. https://arxiv.org/abs/2106.08295 (2021) (cit. on p. 11).

23. Courbariaux, M. & Bengio, Y. BinaryNet: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1. *CoRR* **abs/1602.02830.** arXiv: 1602.02830. http://arxiv.org/abs/1602.02830 (2016) (cit. on p. 11).

24. Yi, Q., Sun, H. & Fujita, M. FPGA Based Accelerator for Neural Networks Computation with Flexible Pipelining. *CoRR* **abs/2112.15443.** arXiv: 2112.15443. https://arxiv.org/abs/2112.15443 (2021) (cit. on p. 12).

25. Wu, R., Guo, X., Du, J. & Li, J. Accelerating Neural Network Inference on FPGA-Based Platforms—A Survey. *Electronics* **10.** ISSN: 2079-9292. https://www.mdpi.com/2079-9292/10/9/1025 (2021) (cit. on p. 12).

26. Liu, Z. G., Whatmough, P. N. & Mattina, M. Systolic Tensor Array: An Efficient Structured-Sparse GEMM Accelerator for Mobile CNN Inference. *CoRR* **abs/2005.08098.** arXiv: 2005.08098. https://arxiv.org/abs/2005.08098 (2020) (cit. on p. 12).

27. Zhang, W., Jiang, M. & Luo, G. *Evaluating Low-Memory GEMMs for Convolutional Neural Network Inference on FPGAs* in (May 2020), 28–32 (cit. on pp. 14, 15).

28. Fasfous, N., Frickenstein, L., Neumeier, M., Vemparala, M. R., Frickenstein, A., Valpreda, E., Martina, M. & Stechele, W. *Mind the Scaling Factors: Resilience Analysis of Quantized Adversarially Robust CNNs* in *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)* (2022), 706–711 (cit. on pp. 18, 29, 32, 47, 51, 53, 56).

29. Pappalardo, A. *Xilinx/brevitas* https://doi.org/10.5281/zenodo.3333552 (cit. on p. 19).

30. Choi, J., Wang, Z., Venkataramani, S., Chuang, P. I., Srinivasan, V. & Gopalakrishnan, K. PACT: Parameterized Clipping Activation for Quantized Neural Networks. *CoRR* **abs/1805.06085.** arXiv: 1805.06085. http://arxiv.org/abs/1805.06085 (2018) (cit. on p. 20).

31. Zhou, S., Ni, Z., Zhou, X., Wen, H., Wu, Y. & Zou, Y. DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients. *CoRR* **abs/1606.06160.** arXiv: 1606.06160. http://arxiv.org/abs/1606.06160 (2016) (cit. on p. 20).

32. *FINN documentation* https://finn.readthedocs.io/en/latest/ (cit. on p. 20).

33. *FINN documentation* https://github.com/Xilinx/finn (cit. on p. 20).

34. *Netron* https://github.com/lutzroeder/netron (cit. on p. 21).

35. *FINN-HLSLIB* https://finn-hlslib.readthedocs.io/en/latest/ (cit. on p. 21).

36. Pappalardo, A. *et al. QONNX: Representing Arbitrary-Precision Quantized Neural Networks* 2022. arXiv: 2206.07527 [cs.LG] (cit. on p. 21).

37. *FINN Streamline Transformations* https://finn.readthedocs.io/en/latest/source_code/finn.transformation.streamline.html (cit. on p. 22).

38. *PYNQ* http://www.pynq.io/ (cit. on p. 24).

39. Alam, S. A., Gregg, D., Gambardella, G., Preusser, T. & Blott, M. *On the RTL Implementation of FINN Matrix Vector Compute Unit* 2022. https://arxiv.org/abs/2201.11409 (cit. on p. 26).

40. Alam, S. A., Gregg, D., Gambardella, G., Preußer, M. & Blott, M. On the RTL Implementation of FINN Matrix Vector Compute Unit. *CoRR* **abs/2201.11409.** arXiv: 2201.11409. https://arxiv.org/abs/2201.11409 (2022) (cit. on p. 25).

41. *HLS Pragmas* hhttps://docs.xilinx.com/r/en-US/ug1399-vitis-hls/HLS-Pragmas (cit. on p. 38).

42. *Pragma HLS Unroll* https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/pragma-HLS-unroll (cit. on p. 38).

43. *Pragma HLS Inline* https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/pragma-HLS-inline (cit. on p. 39).

44. *Pragma HLS Bind-Op* https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/pragma-HLS-bind%5C_op (cit. on p. 40).

45. *Inferring Shift Register* https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Inferring-Shift-Registers (cit. on p. 43).

46. Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M. & Adam, H. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *CoRR* **abs/1704.04861.** arXiv: 1704.04861. http://arxiv.org/abs/1704.04861 (2017) (cit. on p. 44).

47. Zahid, U., Gambardella, G., Fraser, N. J., Blott, M. & Vissers, K. A. FAT: Training Neural Networks for Reliable Inference Under Hardware Faults. *CoRR* **abs/2011.05873.** arXiv: 2011.05873. https://arxiv.org/abs/2011.05873 (2020) (cit. on p. 56).