Politecnico di Torino

POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Software-based test images for in-field fault detection of hardware accelerators

Supervisors

Prof. Annachiara RUOSPO

Prof. Ernesto SANCHEZ

Dr. Antonio PORSIA

Candidate

Giacomo PERLO

Academic Year 2023-2024

A mia madre e a mio padre, per il loro amore incondizionato, il sostegno costante e la fiducia che mi hanno sempre dimostrato.

Ai miei professori e collaboratori, la Professoressa Annachiara Ruospo, il Professor Ernesto Sanchez, ed il Dottor Antonio Porsia per la loro guida, pazienza e preziosi consigli che hanno reso possibile il completamento di questo lavoro.

Ai miei amici più cari
Dima, Nico, Edo, Fre, Lori, Umbi, Eric, Ale, Cappi, Andre, Simo, Zuhdy,
Laura, Giulia, Matilde, Valentina, Sofia, Giorgia
che sono stati, ognuno a proprio modo,
faro nel mare di ansie e trionfi, dubbi e certezze.
Questo traguardo è anche vostro.

Infine, al mio collega Fabrizio, per essere stato una fonte di ispirazione, forza e sostegno costante in ogni momento di questo percorso.

A chi ci ha creduto fin dall'inizio.

Al futuro che attende.

Grazie.

Giacomo

Abstract

The increasing adoption of artificial intelligence (AI)-based systems has led to growing concerns about their deployment in safety-critical environments. Industry standards, such as ISO 26262 for the automotive sector, mandate the detection of hardware faults during the device's canonical operations. Similarly, new standards are emerging to address the functional safety of AI systems (e.g., ISO/IEC CD TR 5469). Hardware solutions have been proposed for in-field testing of the hardware executing AI applications, nevertheless these approaches can increase hardware costs and may potentially negatively impact the strive for performance maximization, especially in applications involving Convolutional Neural Networks (CNNs) for image processing tasks. This thesis inquire into a methodology for creating high-quality test images that can be interleaved with the normal inference process of a CNN executing in the realm of edge devices, specifically exploiting the blend of the open-source configurable and extendable X-HEEP platform, designed to support the exploration of ultra-low power edge accelerators, and the Xilinx PYNQ-Z2 board. Image Test Library (ITL) has been developed in order to facilitate the on-line testing of a 32-bit integer multiplier. The proposed methodology does not necessitate modifications to the existing CNN, thereby avoiding costly memory loading operations, as it effectively leverages the CNN's existing structure. Additionally, the proposed ITL requires minimal test application time and memory space for storing the test images and corresponding golden test responses. The ITL is specifically designed to leverage the convolution operation, which involves multiply-and-add computations between an input image and a set of filters. Since the primary goal is to maintain the original CNN structure, weights must remain unchanged, while the input images are the only elements which can be altered. Consequently, test patterns for the multiplier must be created using Automatic Test Pattern Generation (ATPG) techniques, with the pre-trained network weights serving as constraints. These generated test patterns are then strategically placed in specific areas of the input image to ensure that the multiplier will process them with the corresponding weights used as constraints. Finally, the image containing the test patterns, called faulty image, is tested during the fault injection phase to ensure that its content allows to successfully propagate the fault along the network, resulting in a wrong class prediction. Experimental results demonstrate that the proposed methodology achieves 86.16% test coverage on the hardware unit under test managing to detect the 93.22% of faults, utilizing 21 test patterns generated during the ATPG phase for the quantized (int8) LeNet-5 CNN.

Table of Contents

List of Tables

List of Figures

Acronyms

AI
artificial intelligence

ANN
artificial neural network

CNN
convolutional neural network

STL
software test library

EDA
electronic design automation

GPU
graphics processing unit

ITL
image test library

TC
test coverage

X-HEEP
eXtendible Heterogeneous Energy-Efficient Platform

MLP
multi-layer perceptrons

RGB

red green blue

NLP

natural language processing

ML

machine learning

RISC-V

reduced instruction set computer five

ISA

instruction set architecture

CPU

central processing unit

XAIF

eXtendible Accelerator InterFace

IP

intellectual property

RTL

register-transfer level

SOC

system on a chip

CUT

circuit under test

DC

digital circuit

CMOS

complementary metal–oxide–semiconductor

BIST
: built-it self-test

SBST
: software-based self-test

FI
: fault injection

DNN
: deep neural network

FPGA
: field programmable gate array

ATPG
: automatic test pattern generation

FC
: fault coverage

ICV
: image class visualization

LFSR
: linear feedback shift register

TP
: test program

TF
: tensorflow

OFMAP
: output feature map

IFMAP
: input feature map

PI

primary input

PO

primary output

NAN

not a number

FP32

32-bit floating point

RBF

radial basis function

MSB

most significant bit

LSB

least significant bit

IICV

inverted image class visualization

Chapter 1

Introduction

In recent years, we have witnessed a rapid proliferation of products leveraging Artificial Intelligence (AI) to enhance performance and efficiency across various sectors, from healthcare to industrial manufacturing, automotive to cybersecurity. Among the most widely-used AI techniques, Artificial Neural Networks (ANNs) and, in particular, Convolutional Neural Networks (CNNs), have revolutionized the way machines process and interpret complex data obtaining remarkable results, thereby opening up new horizons in the field of hardware accelerators.

Furthermore, due to the ever-increasing spread of AI products in safety-critical environments as well, such as self-driving cars, facial recognition and medical image analysis, both industrial and academic communities are intensifying their efforts in ensuring the reliability and safety of these technologies. As the ISO 26262 standard commonly followed in the automotive field, or the ISO/IEC CD TR 5469 standard concerning the functional safety of AI systems testify, the extremely fast evolution of safety-related systems is requiring new and evermore reliable solutions to be found out.

Among the possible solutions, on-line testing strategies based on functional methods have become a common solution in industry sectors such as the automotive one [1]. This method executes the on-line test through the periodic execution of Software Test Libraries (STLs) composed of a set of assembly instructions able to excite the processor core and detect permanent faults; furthermore no hardware overhead is requested but suitable memory space where to save the test libraries and a large amount of manual and semi-automatic work, since no Electronic Design Automation (EDA) tools are available for their generation. In the literature, STLs have been proposed as an effective safety mechanism to test hardware accelerators, such as Graphics Processing Units (GPUs), widely used to accelerate AI Applications [2] [3]. However, the execution of STLs interleaving CNN inferences may jeopardize the strive for performance maximization [4].

This thesis describes the Image Test Library (ITL) [5] technique which tries to

overcome the aforementioned issues. This technique proves way more efficient than STLs and consists in exploiting a set of carefully-crafted high-quality test patterns in the form of test images fed to an already trained CNN, specifically targeting convolutional layers which account for more than 90% of the total operations [6]. ITL-based testing is periodically performed during the normal CNN inference process thanks to the input test images which contain the test patterns and allow to detect, with a high test coverage (TC), permanent faults affecting the target hardware unit without affecting the neural network performance. Indeed, they require minimal test application time and memory space for storing the input test images.

Furthermore, the widespread adoption of CNNs has led to an increasing demand for hardware solutions capable of efficiently and swiftly handling the computational operations required by these increasingly complex neural networks. In response to this need, the field of hardware accelerators, particularly in the realm of edge computing, has seen the emergence of new technologies and design approaches aimed at maximizing both the performance and the energy efficiency of such devices. Edge devices offer advantages such as low computation latency and high energy efficiency for executing convolutional neural networks (CNNs). However, deploying CNNs on resource-constrained devices turns out to be a tricky task due to the high computational intensity of CNNs and limited hardware on-chip resources. One of the latest and most interesting solutions is represented by X-HEEP (eXtendible Heterogeneous Energy-Efficient Platform). This architecture aims to provide customization options to match specific application requirements by exploring various core types, bus topologies, and memory addressing modes. It also enables fine-grained configuration of memory banks to match the constraints of the integrated accelerators [7]. This thesis inquires into the synergy of the ITL technique and edge devices (i.e. X-HEEP running on the TUL PYNQ-Z2 physical board) focusing on the in-field test of a 32-bit integer multiplier at the core of convolutional operations. As a case study, a 1-image ITL has been developed for a LeNet-5 CNN.

This thesis is structured as follows: Chapter 2 introduces the fundamental concepts related to CNNs, the architecture of the X-HEEP SoC, the digital circuit testing, the STL and the ITL methods. Chapter 3 outlines the proposed method for generating ITLs for edge accelerators and the process used to validate them. Chapter 4 presents the experimental results obtained exploiting X-HEEP, TUL PYNQ-Z2 board and a LeNet-5 CNN. Chapter 5 suggests directions for future research and draws conclusions.

Chapter 2

Background

2.1 Introduction to CNNs

CNNs present many similarities to ordinary Multi-Layer Perceptrons (MLPs), which are models inspired by the structure and function of biological neural networks in animal brains. The first difference between the two resides in the input: CNNs make the explicit assumption that inputs are images while MLPs expect numeric vectors.

In MLPs the output is computed through a series of hidden layers containing each a set of neurons, each fully connected to all neurons in the previous layer. A weight is associated to each connection, representing the importance of the connection in the neuron's output computation. All neurons in a single layer function completely independently and do not share any connections. These features don't allow MLPs to scale well to full images: for instance, a $200 \times 200 \times 3$ image (so, a RGB image of 200×200 pixels) would lead to neurons in the first hidden layer having 120,000 weights each. The huge number of parameters would quickly lead to overfitting, as well as causing high space occupation. [8]

CNNs, given the constraint of working with images, present a different architecture. Indeed, hidden layers may include one or more convolutions: each convolutional layer has a single set of filters, or kernels, each one containing a set of weights.
As the convolution kernel slides along the input matrix for the layer, the convolution operation generates a feature map, which in turn contributes to the input of the next layer. This is followed by other layers such as pooling layers, fully connected layers, and activation layers [9].

This architecture presents three important features: [9]

1. *Sparse interactions* – in traditional MLP hidden layers the output of each neuron depends on the output of all the neurons of the previous layer, and

so on; such a layer is called *Fully Connected* in the CNN context. However, CNNs typically have sparse interactions. This is accomplished by making a kernel, smaller than the input image, slide over the input. Thereby each output element depends on a small number of inputs.

When processing an image with thousands or millions of pixels, small, meaningful features such as edges can be detected with kernels that occupy only tens or hundreds of pixels. This means that fewer parameters need to be stored, which both reduces the memory requirements of the model and improves its statistical efficiency.

Indeed, If the input has size m, the output has size n and the kernel has size k, the computational complexity of calculating the output of a *Fully connected* layer is *O(mn)*, while for a convolutional layer exploiting the sparsely connected approach it is *O(kn)*.

2. *Parameter sharing* – in traditional neural networks each neuron of a *Fully connected* layer has its own set of weights, resulting in each weight being used at most once in the whole model.

   The idea behind parameter sharing is using the same weight for more than just one neuron; in a convolutional layer, in fact, each weight of the kernel matrix gets multiplied by almost every input element, resulting in a smaller layer memory occupancy: a *Fully connected* layer requires to store $m \cdot n$ parameters, while a convolutional layer requires to store just k parameters

3. *Equivariance to translation* – a function f is equivariant to another function g if they both produce the same result. More specifically, if the input changes, the output changes in the same way.

   For instance, shifting every pixel of an image one unit to the right and then applying the convolution produces the same results as first applying the convolution and then shifting every pixel one unit to the right.

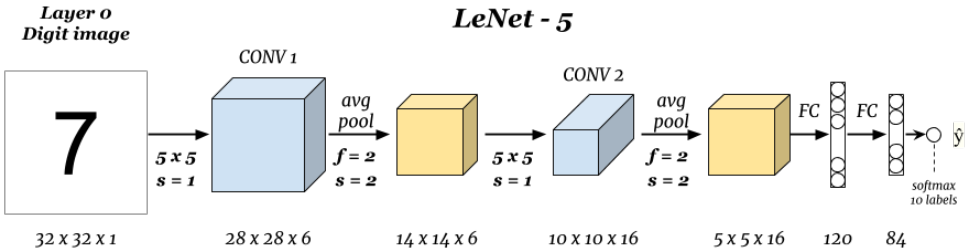

Figure 2.1: Well known CNN: LeNet-5 [10][11]

2.2 Convolutional Layer

At the heart of CNNs there is the convolution operation, a mathematical process that enables the network to learn spatial hierarchies of features from input data, typically images. Convolutional layers in CNNs account for more than 90% of the total operations [6].

In the context of CNNs, the convolution operation involves sliding a filter (or kernel) over the input data to produce a feature map. The kernel is a small matrix of learnable weights that is applied to a local region of the input data. This process allows the network to detect various features like edges, textures, and shapes at different spatial locations. Formally, given an input image I and a kernel K the convolution operation produces an output feature map F. The convolution output at a specific position *(i,j)* is defined as:

$$F(i,j) = \sum_{m=-k}^{k} \sum_{n=-k}^{k} I(i+m, j+n) \cdot K(m,n) \tag{2.1}$$

Consider an input image I of size $H \times W \times C$, where H and W are the height and width of the image, and C is the number of channels (e.g., 3 channels for RGB images). A convolutional layer applies N different kernels K, each of size $k \times k \times C$, to produce an output feature map F of size $H' \times W' \times N$.

The output feature map is computed as:

$$F_n(i,j) = \sum_{c=1}^{C} \sum_{m=1}^{k} \sum_{n=1}^{k} I_c(i+m, j+n) \cdot K_{mn}^{c} \tag{2.2}$$

Here, $F_n(i,j)$ is the output at location (i,j) for the n-th filter, and the sum is performed over all input channels C and the kernel dimensions.

2.2.1 Convolution and Dimensions

- **1D Convolution** – the convolution operation can be applied over 1 dimensional input feature map and kernel.
  Let $I \in \mathbb{R}^n$ be a 1D input feature map of length n and $K \in \mathbb{R}^k$ a 1D filter of length k. The length o of the output feature map $O \in \mathbb{R}^o$ is defined as:

  $$o = n - k + 1$$

  Intuitively, the operation can be imagined as a sliding window (which represents the kernel) moving over the input feature map; each element of the input feature map is multiplied elementwise with the kernel and then summed up to obtain the relative output feature map element. A 1D convolution can extract

features from various kinds of sequential data, and is especially prevalent in audio processing, NLP and financial applications.

Each *i-th* element of the output feature map is computed as follows:

$$O[i] = (I * K)[i] = \sum_{j=0}^{k-1} I[i+j] \cdot K[j], \quad i = 0, 1, \ldots, n-1 \tag{2.3}$$

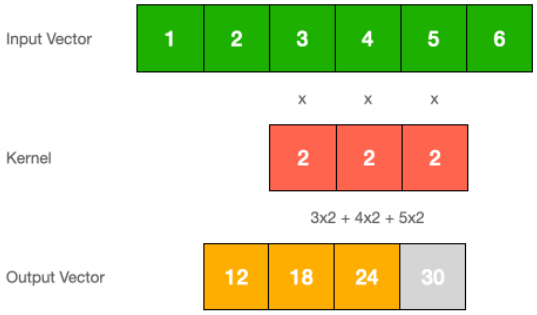

Figure 2.2: 1 Dimension Convolution example [12]

- **2D Convolutions** – the convolution operation can be applied over 2 dimensional input feature map and kernel, as well. Indeed, this is the prevalent case in CNNs dealing with 2D images (the most common scenario). Extending the convolution operation to 2D is pretty straightforward: let $I \in \mathbb{R}^{H_I \times W_I}$ be a $H_I \times W_I$ 2D input feature map and $K \in \mathbb{R}^{H_K \times W_K}$ a $H_K \times W_K$ 2D filter. The output feature map $O \in \mathbb{R}^{H_O \times W_O}$ has height and width defined as:

$$H_O = H_I - k + 1$$

$$W_O = H_O - k + 1$$

Each element at index *(i, j)* of the output feature map is computed as follows:

$$O[i,j] = (I * K)[i,j] = \sum_{w=0}^{W_K-1} \sum_{h=0}^{H_K-1} I[i+w, j+h] \cdot K[w,h], \tag{2.4}$$

$$i = 0, 1, \ldots, W_O - 1$$
$$j = 0, 1, \ldots, H_O - 1$$

- **3D Convolutions** – the concept of 2D convolution can be extended by adding another dimension. This becomes useful for analyzing volumetric data. The most useful applications are medical imagining and scientific computing.

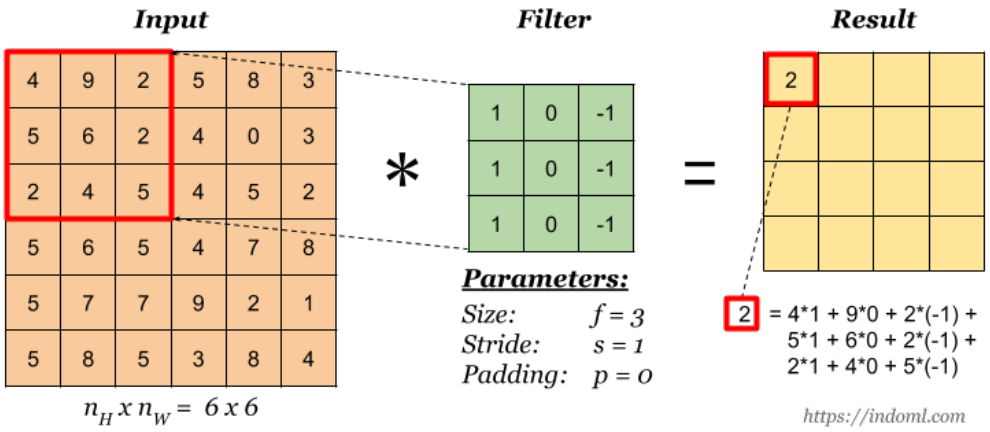

Figure 2.3: 2 Dimension Convolution example [11]

2.2.2 Convolution Hyperparameters

Two important hyperparameters which influence the behavior of the convolution operation are **padding** and **stride**. These parameters control how the convolutional filters interact with the input data.

Padding

Padding refers to the practice of adding extra pixels around the border of the input image. This hyperparameter allows to control the spatial dimensions of the output feature map and allows the network to extract features from the edges of the input image. Without padding, the output feature map would be smaller than the input, and information near the image boundaries could be lost.

The two main padding choices are:

- *Valid Padding* – also known as "no padding," this approach does not add any extra pixels around the input image. As a result, the output feature map is smaller than the input. If the kernel size is $k \times k$, the size of the output feature map O will be reduced by $k - 1$ in both height and width.

- *Same Padding* – this approach involves padding the input image in such a way that the output feature map has the same size as the input. For a kernel of size $k \times k$, the input is padded with $\left\lfloor \frac{k-1}{2} \right\rfloor$ pixels on each side, ensuring that the output feature map O maintains the same height and width as the input.

Mathematically, if the input image has dimensions $H \times W$ and the kernel has dimensions $k \times k$, the stride is s, and the padding is p, the output feature map O will have dimensions:

$$H' = \left\lfloor \frac{H + 2p - k}{s} + 1 \right\rfloor, \quad W' = \left\lfloor \frac{W + 2p - k}{s} + 1 \right\rfloor$$

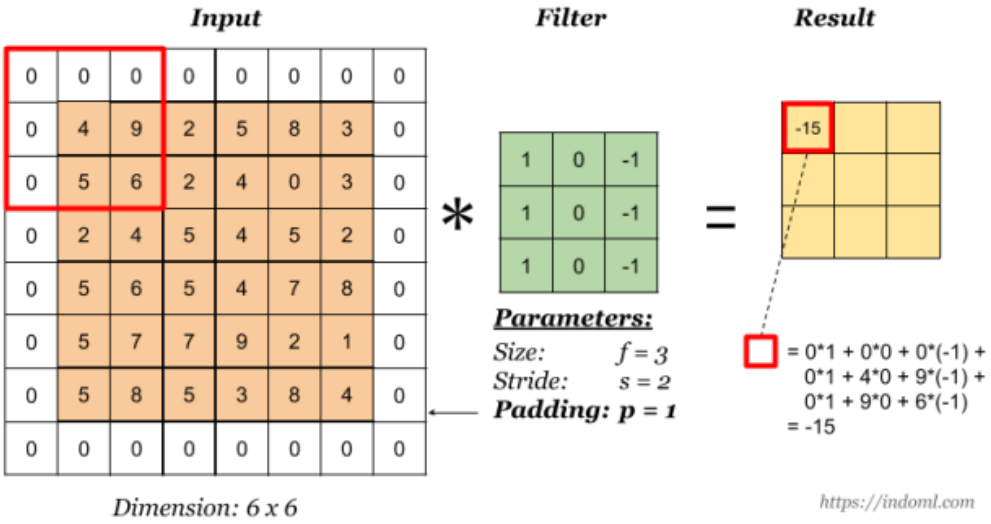

Figure 2.4: 2D Convolution: "Same Padding" = 1, "Stride" = 2 [11]

Stride

Stride refers to the number of pixels by which the filter moves or "strides" across the input image during the convolution operation. The stride modifies the size of the output feature map, reducing it.

A stride value of 1 means that the filter moves one pixel at a time across the input image, while larger stride values (> 1) reduce the overlap between adjacent applications of the filter, effectively downsampling the input image.

Multiple Channels

Most CNNs take RGB images as inputs, which consist of three color channels (red, green and blue), and generate feature maps with many channels in the hidden layers. The 2D convolution operation, indeed, must be extended to a third dimension: given an input feature map $I \in \mathbb{R}^{C \times H_I \times W_I}$ and a kernel $K \in \mathbb{R}^{C \times H_K \times W_K}$, the resulting output feature map O will have a single channel and be represented as $O \in \mathbb{R}^{1 \times H_O \times W_O}$. For the purpose of this work, from now on, the kernels will only be assumed as square (with same height and width).

The output feature map O is computed by performing a 2D convolution for each channel c of the input I with the corresponding channel c of the filter F, and then

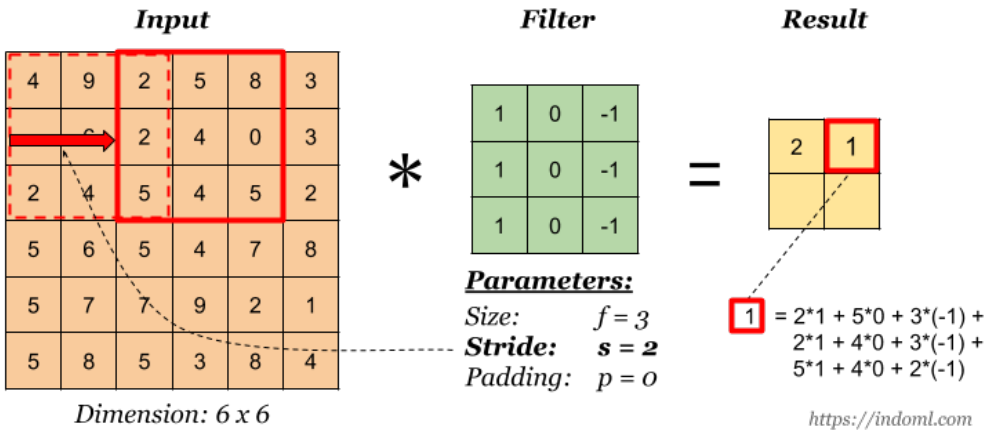

Figure 2.5: 2D Convolution: "Stride" = 2, "Valid Padding" [11]

summing these results elementwise. The same concepts of padding and stride that apply to 2D convolutions also apply in this case, for each 2D slice of the input. Therefore, the element at position (i, j) of the output feature map is defined as:

$$O[i,j] = \sum_{c=0}^{C-1} \sum_{w=0}^{W_K-1} \sum_{h=0}^{H_K-1} I_{c,s\cdot i+w,s\cdot j+h} \cdot F_{c,w,h} \tag{2.5}$$

$$i = 0, 1, \ldots, W_O - 1$$
$$j = 0, 1, \ldots, H_O - 1$$

where s represents the stride value.

If we consider a batch of N images being convolved with the same filter K, an additional dimension is introduced to the input feature map tensors, making $I \in \mathbb{R}^{N\times C\times H_I\times W_I}$. In this case, the convolution operation as defined above is applied independently to each image in the batch, resulting in an output feature map $O \in \mathbb{R}^{N\times 1\times H_O\times W_O}$, meaning one feature map per image in the batch.

Multiple Filters

Convolutional layers are capable of using more than one filter/kernel. Since each kernel detects a specific feature in the input, such as vertical edges, it can be useful to apply multiple filters to capture a wide range of features from the input data. To facilitate this, a fourth dimension is added to the filter tensor, which represents the number of different features we want to learn from the input.

Consider a batch of input feature maps represented by $I \in \mathbb{R}^{N\times C\times H_I\times W_I}$. If we wish to extract M distinct features from each input feature map, the filter

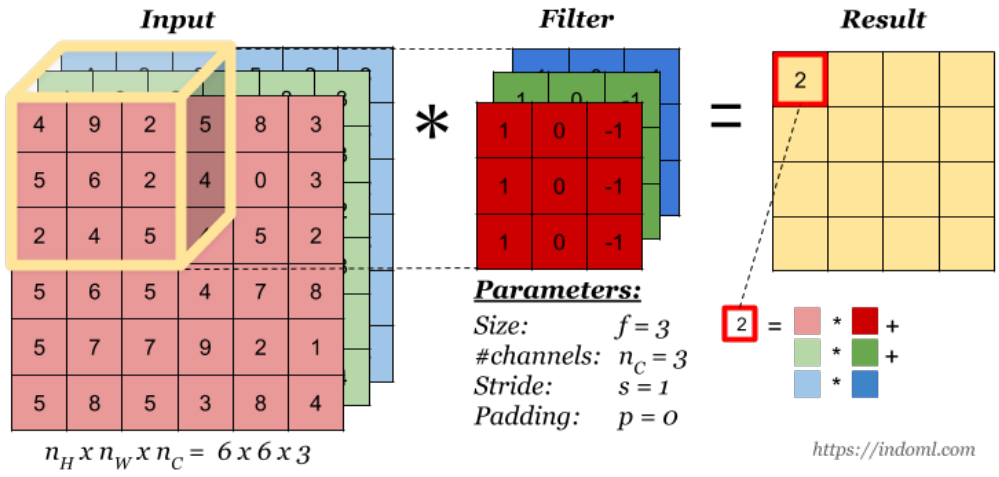

Figure 2.6: 2D Convolution with multiple channels [11]

tensor is defined as $K \in \mathbb{R}^{M \times C \times H_K \times W_K}$, where M represents the number of filters. Consequently, the batch of output feature maps is given by $O \in \mathbb{R}^{N \times M \times H_O \times W_O}$.

The value at position (i, j) in the m-th feature map, derived from the n-th input feature map, is computed as:

$$O_{n,m,i,j} = \sum_{c=0}^{C-1} \sum_{w=0}^{W_K-1} \sum_{h=0}^{H_K-1} I_{n,c,s \cdot i+w,s \cdot j+h} \cdot K_{m,c,k,l} \tag{2.6}$$

$$i = 0, 1, \ldots, W_O - 1$$
$$j = 0, 1, \ldots, H_O - 1$$

2.3 CNN Layers

CNNs are a sequence of layers, each layer transforming one tensor of activations to another through a differentiable function. Four main types of layers are used to build a CNN architecture:

- **Convolutional layers** – the core building block of the CNNs, widely explained in the previous section
- **Pooling layers** – used to reduce the spatial size of the model and control overfitting
- **Fully-connected layers** – the same as hidden layers in MLPs
- **Activation layers** – used when facing non-linear problems

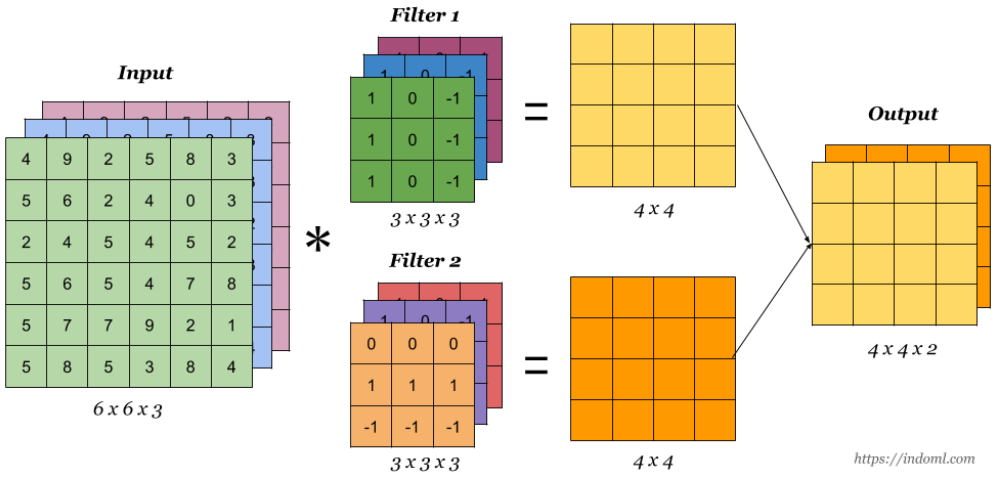

Figure 2.7: 2D Convolution with multiple filters [11]

2.3.1 Pooling Layer

Pooling layers are commonly inserted after convolutional operations. Their function is to progressively reduce the spatial size of the representation to reduce the amount of parameters and computation in the network, hence also controlling overfitting [8]. Different types of pooling exist:

- **Max Pooling** – this pooling operation works independently on each dimension of the input and, usually, it applies a 2×2 filter with a stride of 2, downsampling the input by 2 along both dimensions. This operation consists in selecting the maximum among the values under the sliding filter [8]

- **Average Pooling** – this pooling operation is performed in a similar way to max pooling, but consists in computing the average of all the numbers within each region

2.3.2 Fully-connected Layer

Fully-connected layers, also known as dense layers, capture global patterns and relationships in the input data by connecting every neuron from the previous layer to every neuron in the fully connected layer. It is usually placed at the end of the CNN, following the convolutional and pooling layers. The fully connected layer acts as a feature extractor, transforming the learned features into a format that can be used for classification or regression tasks.

2.3.3 Activation Layer

Activation layers allow the model to deal with non-linear problems by applying *activation functions*. An activation layer is typically placed after each convolution and fully-connected layer to introduce non-linearity into the model. In essence, activation functions serve as the "switch" in artificial neurons that decide whether that neuron should be activated or not based on the weighted sum of the input. The most used *activation functions* are:

- **Sigmoid** or **Logistic**
$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.7}$$
- **Rectified Linear Unit (ReLU)**
$$\text{ReLU}(x) = \max(0, x) \tag{2.8}$$
- **Tanh**
$$\text{Tanh}(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{2.9}$$
- **Softmax** – Differently from previous ones, the Softmax function is computed over all neurons in the layer, returning a probability distribution of K possible outcomes.
$$\text{Softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=0}^{K-1} e^{x_j}} \tag{2.10}$$
where $i = 0..K-1$ [9].

2.4 CNN Quantization

The integration of AI solutions with edge computing is transforming real-time data processing. However, executing vision tasks on resource-constrained edge devices remains a challenge. To overcome this, model compression techniques, particularly quantization, play a pivotal role in optimizing computational power and memory usage.

Quantization aims to convert high-precision floating-point numbers into integer representations. Traditional AI models that use 64-bit floating-point numbers face significant computational burdens. By reducing precision, often down to 8-bit integers, quantization significantly decreases the model's size and processing time, making it more suitable for resource-limited edge devices. Although accuracy may be a concern, modern quantization techniques effectively mitigate this impact.

2.4.1 Quantization Scheme

Let r be a real number to be quantized in its N-bit integer representation q, and let S and Z be the scale and zero-point quantization parameters, respectively. The equation 2.11 shows how to quantize a real number into an integer one, while equation 2.12 shows how to do the reverse operation, indeed, dequantize.

$$q = \left\lfloor \frac{r}{S + Z} \right\rfloor \tag{2.11}$$

$$r = S \cdot (q - Z) \tag{2.12}$$

The quantization parameters S and Z are influenced by the range of both floating point values and integer ones and are computed as follows:

$$S = \frac{r_{max} - r_{min}}{q_{max} - q_{min}} \tag{2.13}$$

$$Z = \left\lfloor \frac{q_{max} - r_{max}}{S} \right\rfloor \tag{2.14}$$

When data is quantized, some information is inevitably lost, creating differences between the original data and its quantized version. These differences are referred to as quantization noise, which can potentially affect the accuracy of the model's predictions. The number of bits (N) is crucial during the quantization process, in fact, fewer bits reduce precision but increase quantization error, risking accuracy loss. More bits enhance precision but require higher resources, crucial in edge computing with limited resources. Figure 2.8 shows a float distribution and its quantized results for different N-bit quantizations [13].

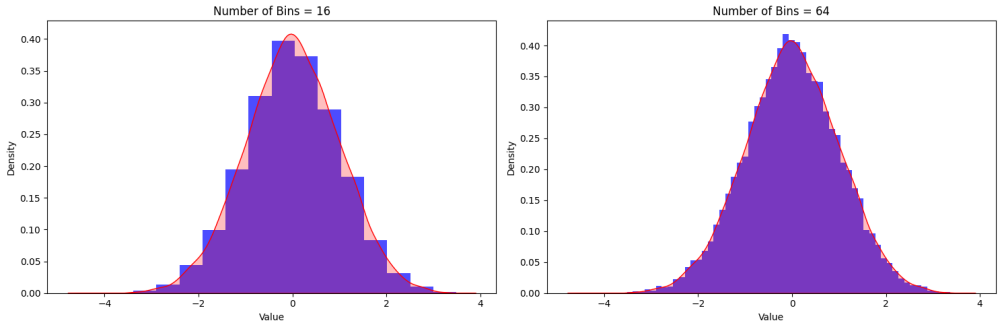

Figure 2.8: Quantization levels (N-bits) influencing quantization noise [13]

2.5 X-HEEP

In recent years, the field of edge computing has witnessed remarkable growth and adoption in commercial products. This process has been driven by the increasing demand for real-time computing solutions, specifically Artificial Intelligence (AI) and Machine Learning (ML) algorithms. As data processing at the edge (for new edge AI computing) has become more prevalent, the performance and power consumption limitations of edge-computing devices have become increasingly apparent.

Heterogeneous architectures have emerged to overcome these challenges relying on a combination of ultra-low-power host processors, and custom accelerators tailored to specific application domains, such as artificial intelligence.

Each accelerator comes with unique requirements, such as memory size, area, performance, and power, to meet the constraints of the target applications. For this reason, proper customization of host platforms is imperative. This may include exploring different CPUs to trade performance and power, bus topologies and memory hierarchy, memory sizes to accommodate the required computational data, peripherals to provide the necessary I/O connectivity, power domains and strategies, etc.

Today, there are an increasing number of open-source projects related to heterogeneous systems, thanks to the open RISC-V instruction set architecture (ISA) revolution. However, many of such platforms focus only on the CPU part, whereas microcontroller-based state-of-the-art projects lack the flexibility and customization options needed to fulfill accelerator requirements natively. These limitations include restricted configurability for the internal platform's components (core, memory, bus, etc.), limited support for external accelerator connectivity, and inadequate built-in power management strategies to optimize energy efficiency. Thus, hardware developers need to extensively modify the platform to properly align with the target applications on their own copy of the platform.

To address the limitations mentioned above X-HEEP [14] was introduced. X-HEEP is an open-source configurable and extendable platform designed to support the exploration of ultra-low power edge accelerators. X-HEEP is a streamlined configurable host architecture based on RISC-V and built reusing existing IPs from well-known open-source projects.

To allow users to explore their custom solutions, X-HEEP can be natively extended via the proposed eXtendible Accelerator InterFace (XAIF), which allows the integration of a wide range of accelerators with different area, power, and performance constraints.

Additionally, to offer high degree of optimizations, X-HEEP offers internal configurability options through the selection of different CPU core types [15], bus topology and addressing mode, memory size and finally peripherals. This

configurability enables designers to tailor the platform to specific application requirements and meet area, power, and performance constraints [7].

2.5.1 X-HEEP Architecture

X-HEEP includes a configurable RISC-V CPU, a configurable bus, a configurable memory, two configurable peripheral domains, and a debug unit.

X-HEEP leverages existing widely adopted open-source IPs to maintain compatibility with existing systems and reuse available software routines and hardware extensions [7].

- *CPU*, the user can choose among the CV32E20, CV32E40X, and CV32E40P as core options [15], to trade off power and performance. In particular, the CV32E20 core is optimized for control-oriented tasks, while the CV32E40P core is optimized for processing-oriented tasks. The CV32E40X core offers power consumption and performance similar to the CV32E40P core, without featuring the floating-point RVF and custom Xpulp ISA extensions. Moreover, it provides an external interface, known as CORE-V-XIF [16], that allows for the plug-in of custom co-processors to extend the RISC-V ISA without the need to modify the RTL code of the core.

- *Memory*, the user can select the memory size and number of memory banks to trade off area, power, and storage capacity.

- *Bus*, the user can choose either a one-at-a-time topology, where only one master at a time can access the bus (one decoder), or a fully connected topology (same number of decoders as simultaneous masters), where multiple masters can access multiple slaves in parallel, to trade off area and bandwidth. In addition, to connect additional components, the bus also exposes a configurable number of slave and master ports to the external XAIF interface to accommodate one or multiple accelerators with different bandwidth constraints.

- *Peripheral domain*, this domain includes peripherals that can be removed from the design or powered off if not needed to trade off area or power and functionality.

- *Always-on peripheral domain*, this domain includes IPs that are always powered on. The power manager is responsible for implementing low-power strategies.

```
package core_v_mini_mcu_pkg;

  // CPU core type selection
  typedef enum logic [1:0] {
    cv32e40p,
    cv32e20,
    cv32e40x,
    cv32e40px
  } cpu_type_e;
  localparam cpu_type_e CpuType = cv32e20;

  // bus topology
  typedef enum logic {
    NtoM,
    onetoM
  } bus_type_e;
  localparam bus_type_e BusType = onetoM;

  // memory configuration
  localparam int unsigned MEM_SIZE = 32'h00080000;
  localparam int unsigned NUM_BANKS = 16;
  localparam logic [31:0] RAM0_START_ADDRESS = 32'h00000000;
  localparam logic [31:0] RAM0_SIZE = 32'h8000;
  localparam logic [31:0] RAM0_END_ADDRESS = RAM0_START_ADDRESS +
   RAM0_SIZE;
  localparam logic [31:0] RAM0_IDX = 32'd1;
  [...]
endpackage
```

Listing 2.1: X-HEEP system definition

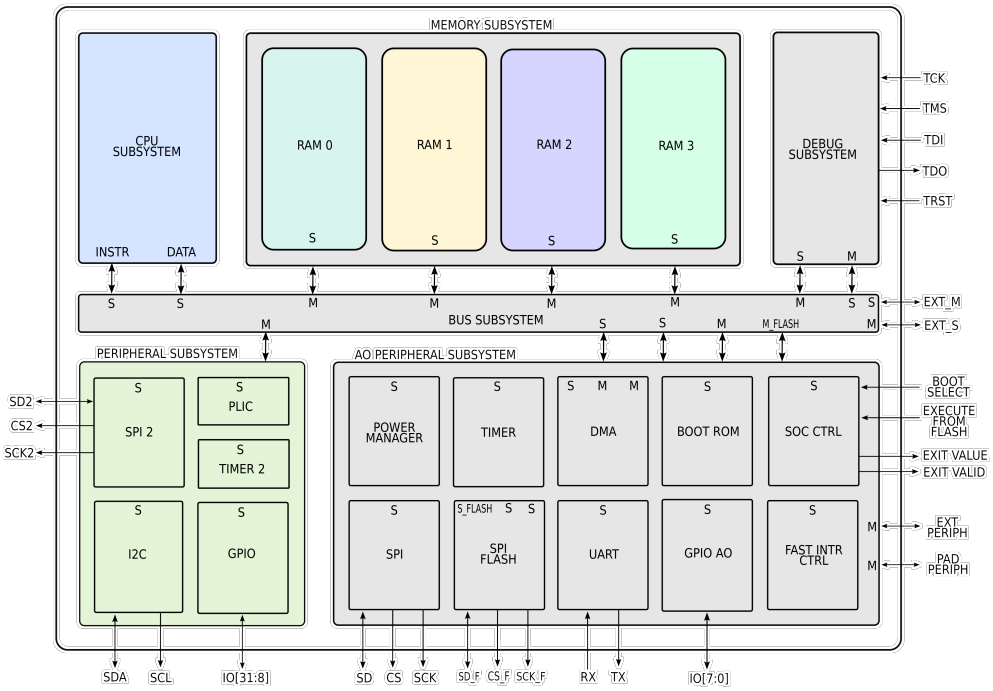

Figure 2.9: X-HEEP architecture diagram [7]

2.6 Digital System Testing

All devices that pass production tests are not identical. When put to actual use, some will fail very quickly while others will function for a long time. Digital system testing consists of periodically applying a set of test patterns, or *stimuli*, to a Circuit Under Test (CUT) observing the response. If the response does not match the expected values, then the CUT is considered to be faulty. In general, each chip could be subjected to different types of tests: [17]

- *Parametric Tests* – DC parametric tests include maximum current test, leakage test, output drive current test, and threshold levels test. These tests are usually technology-dependent.
- *Functional Tests* – these consist in applying stimuli and observing the corresponding responses. Functional testing checks for proper operation of a verified design by testing the internal chip nodes. Functional tests cover a very high percentage of modeled faults (e.g., stuck-at type) in logic circuits. Often, functional vectors are understood as verification vectors, which are used to verify whether the hardware actually matches its specification.

  Functional testing is essential for verifying a design, but it is very challenging

to conduct comprehensively. For a circuit with n input lines, a complete functional test would involve checking all 2^n possible input combinations, which becomes impractical for circuits with a large number of inputs.

- *Structural testing*, on the other hand, is a more practical method because it focuses on the circuit's structure. This approach involves selecting specific test patterns based on the circuit's description and a fault model, which helps in identifying faults caused by manufacturing defects.

In testing literature, it is important to differentiate a defect, an error, and a fault: [17]

- *Defect* – unintended difference between hardware implementation and hardware design
- *Error* – wrong output signal produced by a defective system, i.e. caused by a defect
- *Fault* – representation of a defect at the abstracted function level. Faults, furthermore, can be transient or persistent over time.

2.6.1 Fault Models

To simplify the generation of test patterns by abstracting from hardware defects, **fault models** have been developed. Fault models allow to abstract, level by level, from the physical implementation of the circuit.

The most employed fault models related to Deep Learning can be found at *gate level*, where the circuit consists of a netlist of gates [17].

Fault models in digital circuits can fall under one of the following assumptions:

- **Single fault** assumption – only one fault can occur in a circuit at the same time. Given k possible fault types in the circuit and n signal lines, by single fault assumption, the total number of single faults is $k \times n$
- **Multiple fault** assumption: multiple faults may occur in a circuit

Fault models include:

- **Stuck-at faults** – an input signal, or gate output, is stuck (fixed) at a 0 or 1 value, independently of the inputs to the circuit. Depending on the value, these faults are classified as stuck-at-0 (SA0) or stuck-at-1 (SA1).

  Ideally a gate-level circuit would be completely tested by applying all possible inputs and checking that they compute the right outputs, but this is completely

impractical: a 32-bit adder would require $2^{64} = 1.8 \times 10^{19}$ tests, taking 58 years at 0.1 ns/test. The stuck-at fault model assumes that only one input on one gate will be faulty at a time, assuming that if more are faulty, a test that can detect any single fault, should easily find multiple faults.

Nevertheless, not all faults can be analyzed using the stuck-at fault model, e.g. redundant circuits cannot be tested, since by design there is no change in any output as a result of a single fault.

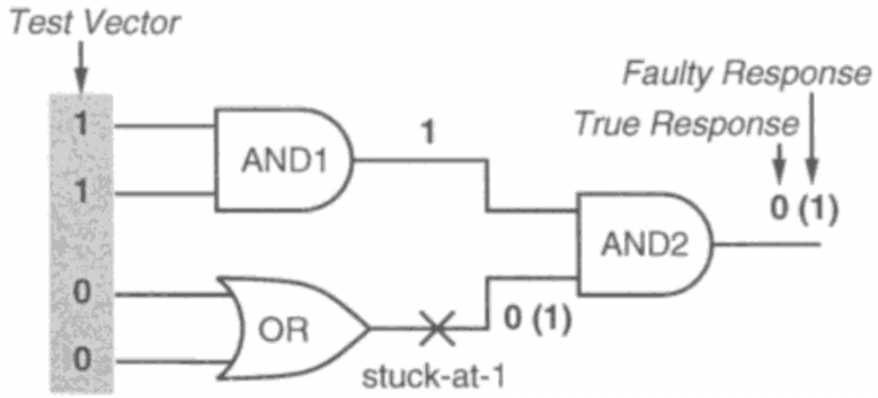

Figure 2.10: Single Stuck-at fault example [17]

- **Bridging faults** – two signals are improperly connected when they should remain separate. This can lead to a wired-OR or wired-AND logic function, depending on the type of logic circuitry used.

- **Transistor faults** – these faults can be found in Complementary Metal Oxide Semiconductor (CMOS) logic gates. At the transistor level, a fault can be categorized as either stuck-short or stuck-open. In a stuck-short condition, the transistor behaves as if it is always conducting (stuck-on), while in a stuck-open condition, the transistor never conducts current (stuck-off).

For the purposes of this thesis, the focus will be placed on stuck-at faults only. Indeed, a worth mentioning property regarding stuck-at faults is the possibility of collapsing different faults. Two main ways of collapsing faults exist:

- **Equivalence collapsing** – all single faults of a digital circuit can be divided into disjoint equivalence subsets, where all faults in a subset are mutually equivalent. A collapsed fault set contains one fault from each equivalence subset, allowing the circuit to reduce the number of single stuck-at faults. Two faults F and F' are equivalent if all tests that detect F also detect F'.

- **Dominance collapsing** – a fault F is called dominant to F' if all tests of F' detects F. In this case, F can be removed from the fault list. Furthermore, if F dominates F' and F' dominates F, then these two faults are equivalent.

2.6.2 Testing Phases

Digital system testing consists of multiple stages, each corresponding to a specific phase in the circuit's lifecycle.

Post-Production Testing

Post-production testing is performed on every manufactured chip to ensure that each device meets its design specifications. This stage focuses on detecting manufacturing defects that could cause the device to fail in the field. Since performing exhaustive testing is both time-consuming and costly, post-production testing typically aims to achieve high fault coverage while minimizing the number of test patterns used [18].

Burn-In Testing

The purpose of burn-in is to accelerate the occurrence of latent defects, thereby causing any potentially defective devices to fail early in their lifecycle. Research has shown that burn-in testing effectively reduces early-life failures in digital circuits, helping manufacturers to improve product reliability.

On-Line Testing

On-line testing is performed while the CUT is in use in its intended application to detect faults that occur during normal operation. This stage often involves *Built-In Self-Test* (BIST) techniques, where additional circuitry is integrated into the device to enable self-testing without external equipment [19]. Alternatively, *Software-Based Self-Test*(SBST) techniques can be used, which rely on executing specific test programs capable of detecting faults without requiring extra hardware. On-line testing is particularly important for safety-critical applications where continuous operation must be assured.

2.6.3 Fault Injection

The effectiveness of the testing stages is often evaluated using Fault Injection (FI) techniques. FI involves deliberately introducing faults into a circuit to study its behavior under failure conditions. Specifically, the circuit is fed of input stimuli and the response is compared with a *golden* reference to assess possible mismatches.

FI Methodologies

The FI methodologies can be classified into simulation-based, platform-based, and radiation-based approaches [20]:

Simulation-based – the injection process is conducted without relying on the physical device executing the Deep Neural Network (DNN). Depending on the level of abstraction, they can be further classified as:

- **Software-level** – Software-level injections are performed on a high-level model of the DNN, without considering any details of the actual hardware architecture.
- **Hardware-level** – Hardware-level injections are performed on a more accurate model of the DNN that simulates the target hardware architecture. For example, the target can be represented at the register transfer level (RTL) or at the gate level

Platform-based – measurements and analyses are performed directly on a physical device that emulates the final implementation of a design, e.g. using FPGAs or on physical platforms that run DNNs, such as CPUs and GPUs.

Radiation-based – reliability assessment is carried out through accelerated radiation test campaigns mimicking external electromagnetic interference, such as the occurrence of ionizing particles, on the actual platform running the DNN.

2.6.4 Test Generation

As discussed earlier in this section, digital circuits are typically tested by applying a set of test patterns. Performing an exhaustive functional test of a device with n inputs would require testing all 2^n possible input combinations, which quickly becomes impractical as n increases. Therefore, structural testing is employed to limit the test patterns to those required to detect specific types of faults, as defined by a fault model.

To evaluate the effectiveness of a structural test, a commonly used metric is Fault Coverage (FC), which is defined as:

$$FC = \frac{\text{number of detected faults}}{\text{total number of faults}} \tag{2.15}$$

There are several test pattern generation techniques, each based on different methodologies, including Automatic Test Pattern Generation (ATPG), random test pattern generation, and evolutionary-based test pattern generation.

Automatic Test Pattern Generation (ATPG) is an Electronic Design Automation (EDA) technique used to find input test vectors that, when applied

to a digital circuit, allow automatic test equipment to differentiate between the correct behavior of the circuit and the faulty behavior. The patterns generated by ATPG are utilized to test semiconductor devices after manufacturing, particularly exploiting the On-Line testing strategy.

A fault is considered detected by a test pattern if the output resulting from that test pattern, when applied to a device with only that specific fault, differs from the expected output. The ATPG process for a targeted fault involves two main phases: *fault activation* and *fault propagation*. Fault activation sets a signal value at the fault model site that is the opposite of the value produced by the fault model. Fault propagation then advances the resulting signal value, or fault effect, by sensitizing a path from the fault site to a primary output. Since ATPG algorithms rely on a low level description of the circuit they are able to generate very accurate test patterns but may require long time span to complete [17].

Random Test Pattern Generation is a EDA technique which aim to generate pseudo random test patterns. This method is particularly useful for scenarios where exhaustive testing is impractical due to the large number of possible input combinations. Random patterns can be quickly generated using simple hardware structures, such as Linear Feedback Shift Registers (LFSRs), which makes them a popular choice in Built-In Self-Test (BIST) environments. The main drawback which can be observed is the large number of patterns which this technique may require to achieve high fault coverage, especially for circuits with random-pattern-resistant faults. [17, 21].

Evolutionary-Based Test Pattern Generation leverages evolutionary algorithms to create effective test patterns for digital circuits. These algorithms mimic the process of natural evolution, utilizing operations such as selection, crossover, and mutation to evolve a population of test patterns over several generations. The primary goal is to maximize a fitness function that represents the fault coverage capability of each test pattern. This approach is particularly useful for complex circuits where traditional test pattern generation methods may fall short [17].

2.7 Software Test Library (STL)

On-line testing strategies based on functional methods have been incorporated as a common solution by industry sectors such as the automotive one [1]. In these cases, the on-line test of the processor core and the related peripherals, including Graphics Processing Units (GPUs), is performed through the periodic execution of **Software Test Libraries (STLs)** composed of a set of assembly programs, i.e. Test Programs (TPs) able to thoroughly excite the processor core and detect possible permanent faults, such as SA-1 and SA-0. Adopting STL solutions allows the system to perform the test on-line, and does not require any hardware overhead

since it only needs a suitable memory space for saving the test libraries. In the literature, STLs have been proposed as an effective safety mechanism to test in the field systems such as Graphics Processing Units (GPUs), widely used to accelerate AI applications [2, 3]. However, devising an STL requires a large amount of manual and semi-automatic work, since no EDA tools are available for their generation. In particular, the execution of specific STLs interleaving CNN inferences may jeopardize the strive for performance maximization [4] [5].

A high-level architecture of an STL is shown in the figure below. The architecture is divided into four components [22]:

- Simple API
- Scheduler
- Blocks: logical group of parts representing the functional blocks of the processor, i.e. core, to ensure configurability of STLs in accordance with the CPU configuration.
- Parts: generated either by constrained random test generator or directed tests written to target a specific logic. These tests are written in assembly code to enable efficient execution and to avoid compiler optimizations that may occur for code written in C.

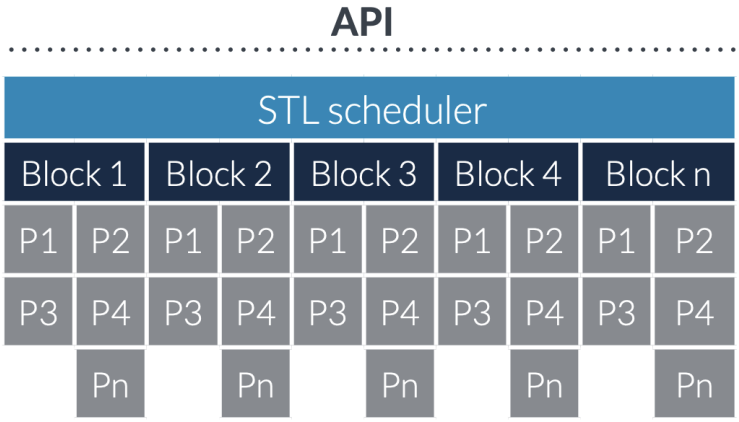

Figure 2.11: ARM STL example architecture [22]

2.8 Image Test Library (ITL)

This section aims to present a method to construct software-based test images that can effectively detect in-field permanent faults occurring during the operational

phase of a multiplier in GPUs, e.g., due to aging effects. The overall idea relies first on an ATPG-based approach (highly effective for regular structures like accelerator functional units) to find out a set of suitable input values at the functional unit level and then transform them into a test image. By utilizing the CNN architecture and its weights, particularly those of the convolutional layers, these test images can be processed to achieve high fault coverage on the accelerator under test. As a result, the generated test images can be executed in-field using the same CNN, alternating between "normal" inferences and self-test images from the *Image Test Library* (ITL), without needing to load new weights into memory. The concept is primarily based on the following observation: once a CNN is deployed in the field, the trained model is loaded, and its weights remain static. Unlike its software counterpart, Software Test Libraries (STL), the proposed method enables a self-test routine on an accelerator by directly utilizing the CNN without altering or interrupting its execution. This approach avoids the costs linked to memory load operations and context switching [5].

2.9 ITL Generation

The idea behind the generation of test images is based on three steps [5]:

1. **Dataflow algorithm extraction** – the objective of this phase is to determine how various operations are scheduled on the GPU's architecture and how convolution processes are executed. It is essential to identify the correlation between input pixels, CNN weights and the GPU's multipliers which compute convolutions. In accelerators, the dataflow algorithm is generally predefined in the architectural specification.

2. **ATPG-based pattern generation** – once the dataflow algorithm is determined, an ATPG process is initiated to identify the set P_c of input-weight test patterns $\langle i, w \rangle$ that maximize the test coverage of the GPU's multipliers. The parameter w corresponds to the actual trained weights W of the CNN, which are imposed as constraints during the ATPG generation. As a result, the generated test patterns depend on both the real CNN weights and the input values produced by the ATPG. For online testing, these carefully-designed test patterns $\langle i, w \rangle$ are applied to the multiplier unit through appropriate images that make up the ITL.

3. **Self-test images generation** – given the list of suitable input positions multiplied by each weight, it is possible to reconstruct the images placing each pattern into a free input position.

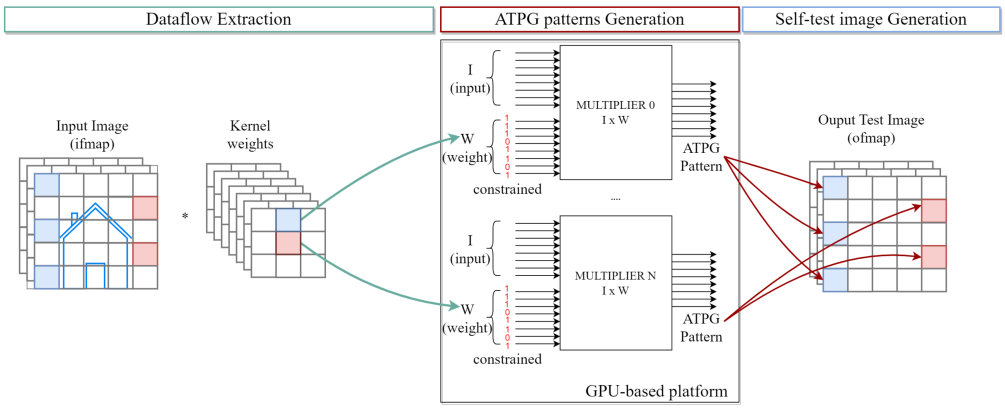

Figure 2.12: Graphic representation of the proposed method to generate ITLs [5]

2.9.1 Dataflow Algorithm Extraction

Let $I \in \mathbb{R}^{N \times H_I \times W_I \times C}$ represent the input feature map, $W \in \mathbb{R}^{M \times f \times f \times C}$ the filter tensor, and $O \in \mathbb{R}^{N \times H_O \times W_O \times M}$ the output feature map resulting from the convolution of I and W performed by the CNN.

The Dataflow Algorithm Extraction stage involves mapping both the elements of O to the specific multipliers that compute their values, and the elements of O to the elements of I to the elements of W. This mapping is referred to as the dataflow algorithm.

Three main phases compose this process:

1. *Thread-core mapping* – the mapping between a software thread and a hardware core. Its purpose is to uniquely associate a thread T to a GPU core C:

$$\begin{aligned} T_0 &\iff C_0 \leftarrow [\langle I_{0,0,0,0}, w_{0,0,0,0} \rangle, \ldots] \\ T_1 &\iff C_1 \leftarrow [\langle I_{1,0,0,1}, w_{1,0,0,1} \rangle, \ldots] \\ &\vdots \end{aligned}$$

2. *Workload-thread mapping* – the mapping between an element of the OFMAP and a software thread. Its purpose is to associate a thread T to a specific output element $O_{n,h,w,m}$ and, consequently, associating T to a list of multiplications represented as $\langle i, w \rangle$ pairs:

$$\begin{aligned} T_0 &\iff O_{0,0,0,0} \leftarrow \{[\langle i_{0,0,0,0}, w_{0,0,0,0} \rangle, \ldots], [\langle i_{0,0,1,0}, w_{0,0,0,0} \rangle, \ldots], \ldots\} \\ T_1 &\iff O_{1,0,0,1} \leftarrow \{[\langle i_{1,0,0,1}, w_{1,0,0,1} \rangle, \ldots], [\langle i_{1,0,1,1}, w_{1,0,0,1} \rangle, \ldots], \ldots\} \\ &\vdots \end{aligned}$$

Understanding the convolution algorithm is crucial for linking each output element to the specific multiplications that produced it. These multiplications are represented as pairs $\langle i, w \rangle$, where $w \in W$ represents a weight, and $i \in I$ is an element in the input feature map (IFMAP).

Given an input tensor (which may be padded) $I \in \mathbb{R}^{N \times H_I \times W_I \times C}$, a filter tensor $W \in \mathbb{R}^{M \times f \times f \times C}$, and an output tensor $O \in \mathbb{R}^{N \times H_O \times W_O \times M}$, each element $O_{n,h,w,m}$ in O corresponds to the set of multiplications that compute it, e.g.:

$$O_{0,0,0,0} \rightarrow [\langle i_{0,0,0,0}, w_{0,0,0,0} \rangle, \langle i_{0,0,1,0}, w_{0,0,1,0} \rangle, \ldots]$$

$$O_{0,0,1,0} \rightarrow [\langle i_{0,0,1,0}, w_{0,0,0,0} \rangle, \langle i_{0,0,2,0}, w_{0,0,1,0} \rangle, \ldots]$$

$$\vdots$$

The pairs $\langle i, w \rangle$ will be instrumental during the self-test image generation process, identifying the available positions within the input image where the input patterns could be placed.

2.9.2 ATPG-based Pattern Generation

Once the dataflow algorithm is established, each multiplier is assigned to a specific set of multiplications that compute an OFMAP location. An ATPG process is then initiated to identify the set P_c of input-weight pairs $\langle i, w \rangle$ that maximize test coverage for the multipliers in core c. To achieve this, the weights $w \in W \in \mathbb{R}^{M \times f \times f \times C}$, representing the actual trained weights from a CNN layer, are applied as constraints during ATPG generation. Additionally, an extra condition is imposed to force the 30-th bit of i to be 0, preventing the generation of infinities and NaN values. It is important to emphasize that the ATPG process is executed solely on the targeted modules, and the resulting test patterns are specific to their inputs.

2.9.3 Self-test Image Generation

For the online testing phase, the crafted $\langle i, w \rangle$ pairs generated in the previous step are provided to the multiplier units through appropriately constructed images from the ITL. In the initial step, the Dataflow Algorithm Extraction identified the precise location $i_{n,h,w,c}$ of the input feature map (IFMAP) being multiplied by the filter weight $w_{m,f,f,c}$, these information ensure that the patterns are accurately positioned within the input feature map, allowing the multiplier to correctly perform the multiplication with the corresponding weight w.

Algorithm 1 summarizes the entire self-test image generation process. Given a certain core and the weights of the considered convolutional layer, for each of

Algorithm 1 Self-test Image Generation [5]

Inputs: ATPG-patterns - IFMAP inputs
Outputs: ITL - List of test images $[I_{fmap} \in \mathbb{R}^{H_I \times W_I \times C}]$

```
1: ITL ← []
2: for core ← 0 to n_cores do do
3:     inputs ← MAP-CORE-TO-INPUTS(core)
4:     GROUP-BY-WEIGHT(inputs)
5:     patterns ← ATPG-patterns[core]
6:     for (pattern, weight) in patterns do
7:         w_idx ← GET-WEIGHT-INDEX(weight)
8:         positions ← inputs[w_idx]
9:         I_fmap, i_free ← FIND-EMPTY-POS(ITL, positions)
10:        if I_fmap = nil then
11:            I_fmap ← APPEND-NEW-IMAGE(ITL)
12:            i_free ← positions[0]
13:        end if
14:        I_fmap[i_free] ← pattern
15:    end for
16: end for
17: return ITL
```

them, the list of I_{idx} indexes from the IFMAP involved in a multiplication with the considered weight is built (line 4). Then, the ATPG process is executed (line 5) in order to gather the $(pattern, weight)$ test pattern pairs; every pattern must be placed in a different and suitable location of the IFMAP. To reach this goal, all the patterns are considered one by one (line 6) looking for the IFMAP positions in which the considered weight is involved in the multiplication with the pattern index (line 8). If any free position is available (line 9) (i.e. it is not already occupied by another pattern), then the pattern is placed in the chosen position (line 14), otherwise a new ITL image is generated (line 11).

2.10 ITL Validation

To validate the use of ITLs for on-line testing, it is crucial to emphasize their capability to trigger hardware faults in the targeted functional module and allow those faults to propagate to the software level, where their presence is detected. In order to analyze the cross-level propagation of hardware faults within a target unit, architectural-level fault simulations must be conducted. For each injected fault, it is necessary to verify whether the fault is propagated to the software layer.

2.10.1 Fault Injection

The idea proposed in [5] stems from a mathematical observation. Let's consider the inputs (I and W) and the output (O) of a multiplier. In the presence of a fault affecting it, the product $I \times W$ may yield a faulty output $\hat{O}$, that is: $I \times W = \hat{O}$ However, this fault can also be thought of as a faulty input ($\hat{I}$ or $\hat{W}$) entering a golden multiplier and producing the same faulty output $\hat{O}$. Knowing the value of $\hat{O}$ that derives from a fault affecting the multiplier, it is possible to obtain the respective faulty input ($\hat{I}$ or $\hat{W}$), that corresponds to the same fault without injecting it. Assuming a golden multiplier, the same fault can be seen as:

$$\hat{I} = \frac{\hat{O}}{W} \tag{2.16}$$

$$\hat{W} = \frac{\hat{O}}{I} \tag{2.17}$$

Furthermore, if a faulty multiplier performs J multiplications, there will be J corrupted outputs $\hat{O}_j$ for $j \in [1, .., J]$. This is equivalent to having J multiplications executed by a golden multiplier with a set of corrupted inputs $\hat{I}_j$ (or weights $\hat{W}_j$), for $j \in [1, .., J]$ [5].

Fault Equivalence

According to [23], this property describes how two faulty functions are identical [17]. To better explain what fault equivalence is, the XOR function is exploited; let V be a test array for a fault:

$$f_0(V) \oplus f_1(V) = 1 \tag{2.18}$$

, where f_0 is a fault-free function and f_1 is the faulty one. Consider a second fault which produces a faulty function f_2. According to the definition of *fault equivalence*, two equivalence faults have the exactly the same tests. Therefore, two faults to be equivalent, we have

$$[f_0(V) \oplus f_1(V)] \oplus [f_0(V) \oplus f_2(V)] = 0 \tag{2.19}$$

Manipulation of the above equation leads to the following result:

$$f_1(V) \oplus f_2(V) = 0 \tag{2.20}$$

which means that the two faulty functions are identical.

Now, let's apply the *fault equivalence* property to the ITL's context: let f represent a fault, (I, W) be a test pattern for f, $M(I, W)$ be the multiplication

between I (representing the input) and W (representing the weight) performed by a fault-free multiplier, and $\hat{M}_f(I, W)$ be the multiplication of I and W performed by the same multiplier in the presence of fault f. Thus, we have $M(I, W) = O$ and $\hat{M}_f(I, W) = \hat{O}_f$.

Now, as per the equation 2.20, let's assume there is another fault g affecting the input lines of the multiplier such that $\hat{M}_g(I, W) = \hat{M}_f(I, W)$. Since faults f and g yield the same output, they are identical.

For the purpose of fault injection, this fault equivalence implies that the output produced by injecting g and testing it with (I, W) is the same as the output produced by injecting f. However, let's consider the case in which injecting f might be complex, while injecting g might be significantly simpler. It can be observed that g causes the multiplier to receive a faulty input $\hat{I}_g$, but it does not alter the internal multiplier, i.e. the effective computation performed by the multiplier remains correct.

This observation allows to assume that the multiplier is fault-free while shifting the fault g to the input, effectively computing $M(\hat{I}_g, W) = \hat{O}_f$. Since the fault g is considered controllable, instead of injecting f, its equivalent fault g is injected on the input of a fault-free multiplier.

ITL Fault Injection

In the [5] paper, a methodology to perform very accurate software FIs by applying faulty inputs $\hat{I}$ to the CNN which exactly correspond to specific hardware faults internal to the targeted functional unit, is proposed. This approach has two main advantages: it combines the accuracy of the gate-level micro-architectural simulation with the speed of software FIs, and it allows to experimentally demonstrate that the proposed self-test images (ITLs) can excite permanent faults inside functional units while propagating the effects up to the OFMAP. The impact of hardware faults is not simulated by performing complex and costly multi-level simulation environments, but only launching the inference of faulty images that exactly reflect a precise hardware fault within the network. The generation of faulty images that corresponds to injecting a specific fault within a multiplier is described in Algorithm 2.

The Algorithm 2 allows to generate a list of faulty images which corresponds to injecting a specific fault within a multiplier. Once the fault is injected at low level (level 4), for each multiplication which is performed during the convolution, its weight $W[op]$ and faulty output $\hat{O}$ are collected in order to compute the relative faulty input $I[\hat{op}]$ (line 6,7,8). Eventually, the list of all the faulty inputs is pitched into faulty images.

To inject faults at application-level using the images generated with Algorithm 2, it is necessary to combine the information of the list of images in a single faulty

Algorithm 2 Faulty Images for a Fault in an HW Multiplier [5]

Inputs:

- MULx - Selected multiplier;
- fault - A stuck-at fault of MULx;
- ITL - Image test library for a specific CNN;
- Operations - Pairs of $\langle \text{input}, \text{weight} \rangle$ multiplications performed by MULx during the convolution;
- n_op - Number of Operations.

Output: FImg - List of faulty images for a single HW fault.

```
1: FImg ← []
2: Î ← []
3: W ← []
4: MULX-INJECT(fault)
5: for op ← 0 to n_op do
6:     Ô ← MULX-MULTIPLY(Operations[op])
7:     W[op] ← GET-WEIGHT(Operations[op])
8:     Î[op] ← Ô/W
9: end for
10: FImg[fault] ← PATCH-ITL(Î, W, ITL)
11: MULX-CLEAN(fault)
12: return FImg
```

OFMAP.

Software-level Observability

With the developed ITL, during the on-line self test, we want to fix the observability point at the software level. As a consequence, for each self-test image, the respective golden output of the softmax layer is stored, and it is compared to the resulting one on-line: if they differ, a warning is raised.

Chapter 3

ITL for Edge Accelerators

This chapter aims to explain how the ITL generation process, outlined in Section 2.8, has been adapted to function effectively on edge devices running CNNs. While the core steps remain unchanged, certain modifications were necessary to accommodate the specific architecture of the edge accelerator being used.

3.1 ITL Generation for Edge Accelerators

3.1.1 Dataflow Algorithm Extraction

In the context of this thesis, using the X-HEEP architecture to run CNNs means working with a RISC-V microcontroller based on the RISC-V lowRISC Ibex [24], a 32-bit open-source low-power single-core single-thread system. As a result, two of three main phases of which the dataflow algorithm is composed by, can be omitted, specifically:

- *Thread-core mapping* – since the X-HEEP microcontroller is a single-thread single-core architecture, this phase decades.

- *Workload-thread mapping* – since the X-HEEP microcontroller is a single-thread single-core architecture, this phase decades as well. All the elements of the OFMAP are computed by the same thread on the same core.

The last phase of the dataflow algorithm, the convolution algorithm, is crucial for linking each output element to the specific multiplications that produced it. Indeed, the pairs $\langle i, w \rangle$ will be instrumental during both the self-test image generation process, and during the ITL validation process in which they will allow to build the multiplications performed during the convolution bypass described in Section 3.2.

3.1.2 ATPG-based Pattern Generation

During inference in quantized neural networks, the result of an operation may fall outside the range of the chosen number representation. In particular, when using 8-bit signed integers it may happen to incur in an overflow during a multiplication or addition. For this reason, the activations of a quantized layer are computed using 32-bit signed integer arithmetic and the final results are then rescaled to 8 bits. The ATPG process should be designed keeping this in mind, by considering weights and inputs as 8-bit signed integers on which sign extension to 32-bit is performed. Practically, this translates to constraining the most significant 24 bits of the operands to be equal to the sign bit of the operands.

It is important to note that the ATPG process is only executed on the targeted module (i.e. in the context of this thesis, the integer multiplier parts of the X-HEEP accelerator), and the resulting test patterns pertain solely to its inputs.

3.1.3 Self-test Image Generation

Given the context of this thesis, exploiting the single-core single-thread X-HEEP architecture allows to overcome both the thread-core mapping and the workload-thread mapping phases since the whole weights are computed by the same 32-bit core and the same thread. The just mentioned configuration allows to exploit the Algorithm 1 but with $n_{cores} = 1$.

3.1.4 Inverted Image Class Visualization

After placing all the patterns in the IFMAP positions, the self-test image generation process proceeds to the next phase: the *Inverted Image Class Visualization*. This step only considers the IFMAP positions where no patterns are present, enabling the generation of an image as balanced as possible, across all class predictions. This balance increases the likelihood of leading to an incorrect class prediction in the occurrence of a fault [25].

Image Class Visualization

The idea exploited in this thesis stems from the concept of *Image Class Visualization*. According to [26], the Image Class Visualization (ICV) is a technique for visualising a chosen class model. Given a trained classification network and a class of interest, the visualisation method consists in numerically generating an image, which is representative of the class in terms of the network class scoring model.

More formally, let $S_c(I)$ be the score of the class c, computed by the classification layer of the network for an image I. The objective is to find an L2-regularised

image, such that the score S_c is high:

$$\arg\max_I \; S_c(I) - \lambda \|I\|_2^2,$$

where λ represents the regularization parameter. A locally-optimal I can be obtained using the back-propagation method. This process is similar to the network training procedure, where back-propagation is employed to optimize the layer weights. The key difference, however, is that in the case of image class visualization, the optimization is performed with respect to the input image, while the weights remain fixed to those determined during the training phase.

It should be noted that the (unnormalised) class scores S_c are used, rather than the output confidence scores returned by the Softmax layer: $P_c = \frac{\exp S_c}{\sum_c \exp S_c}$. The reason is that the maximisation of a certain output score can be achieved by minimising the scores of other classes. Therefore, S_c has been optimised to ensure that the optimisation concentrates only on the class in question c [26].

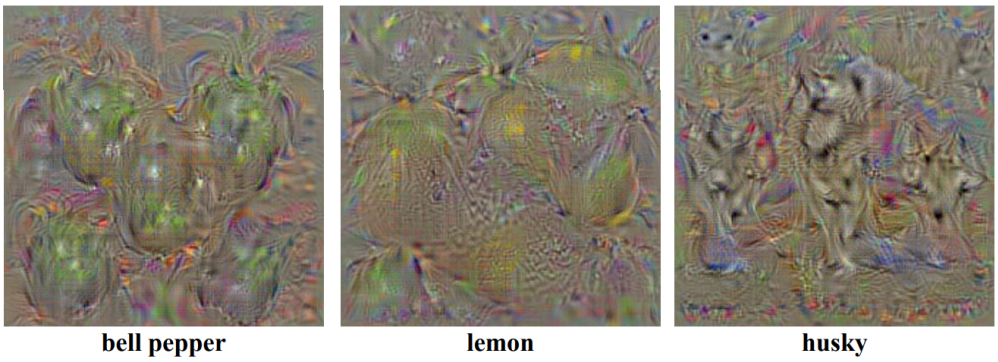

Figure 3.1: ICV example: numerically computed input images [26]

Inverted Image Class Visualization

Built on the original concept of ICV, the opposite reasoning (i.e. Inverted Image Class Visualization (IICV)) aims to maximize the class prediction balance by generating an image that is equidistant from all class predictions (i.e. maximizing the input image's entropy), effectively maximizing the similarity between the predicted probabilities and an array of equally distributed probabilities across the available classes, starting from a completely random generated image.

Cosine similarity is used as a loss function, since it is particularly suited for tasks that involve minimizing the distance between two vectors. The cosine similarity is computed on the output of the Softmax activation function, i.e., on the output confidence scores, differently from ICV, and is defined as:

$$\text{CosineSimilarity}(P_c, \hat{P}_c) = -\frac{P_c \cdot \hat{P}_c}{\|P_c\|^2 \|\hat{P}_c\|^2}$$

where P_c is the vector containing the predicted confidence scores and $\hat{P}_c$ is the vector containing the target confidence scores.

3.2 ITL Validation for Edge Accelerators

The ITL Validation step is necessary to validate the use of ITLs for on-line testing of edge accelerators.

In this thesis, using a 32-bit integer multiplier, and exploiting a quantized network composed by 8-bit signed integer tensors, posed challenges for Algorithm 2, which is responsible for generating faulty images for fault injection and validating the ITL method for traditional GPUs.

$$\hat{I} = \frac{\hat{O}}{W} \tag{3.1}$$

The equation 3.1, deriving from the equivalence $M(\hat{I}, W) = \hat{O} \equiv \hat{M}(I, W) = \hat{O}$, where M is the multiplication between I and W, cannot be exploited in order to retrieve the faulty input $\hat{I}$ (and, consequently, the faulty images which correspond to inject a specific fault within the multiplier, according to line 10 of Algorithm 2), since the multiplicative inverse of an integer in a finite field – required to compute the division – is not always guaranteed to exist.

The solution to this issue is to directly compute the faulty convolution starting from the faulty outputs obtained by fault injection. Each element of the output feature map is manually computed by summing the outputs of multiplications involved in the computation of that element. The resulting feature map is then passed to activation functions to obtain the final output.

As a result, the ITL validation process could carry out faulty multiplications and bypass the traditional convolution operation in favor of a "custom" convolution that utilizes a faulty multiplier. The general process is reported in algorithm 3.

Algorithm 3 Faulty Convolution

Inputs:

- MUL - Multiplier;
- fault - A stuck-at fault of MUL;
- Ifmap - Input feature map of a specific convolutional layer;
- Operations - Pairs of $\langle \text{input}, \text{weight} \rangle$ multiplications performed by MUL during the convolution;
- n_op - Number of Operations.

Output: FOfmap - Faulty output feature map for a single HW fault.

```
1: FOfmap ← []
2: Ô ← []
3: MULInject(fault)
4: for op ← 0 to n_op do
5:     Ô[op] ← MULMultiply(Operations[op])
6: end for
7: for i ← 0 to Size(FOfmap) do
8:     FaultyMuls ← GetMultiplications(i)
9:     FOfmap[i] ← Sum(FaultyMuls)
10: end for
11: return FOfmap
```

Chapter 4

Results

This chapter presents the experimental results that demonstrate the effectiveness of the proposed approach, evaluated using the X-HEEP microcontroller platform along with the TUL PYNQ-Z2 board. As a case study, ITL was generated for one CNN architecture, namely LeNet-5 [10]. The ITL was specifically designed to detect permanent faults affecting the accelerator's 32-bit integer multiplier.

Section 4.1.1 provides an overview of the X-HEEP platform, CNN architecture, and multiplier used in the experiments. Section 4.3 delves into the ITL generation process, highlighting the steps outlined in Section 3.1, as well as detailing the ATPG process and key ITL parameters such as test coverage and storage needs. Additionally, a visual representation of the generated ITL is provided. Section 4.4 outlines the results obtained through the ITL validation method described in Section 3.2.

4.1 Experimental Setup

4.1.1 X-HEEP

The chosen X-HEEP platform involves:

- a CPU subsystem based on the RISC-V lowRISC Ibex, a 32-bit open-source low-power single-thread single-core written in SystemVerilog originally designed by ETH Zurich. Ibex offers several configuration parameters to meet the needs of various application scenarios. Specifically the CV32E20 core was chosen.
- a one-to-many bus topology.
- 16 banks of memory of 32 KB per each.

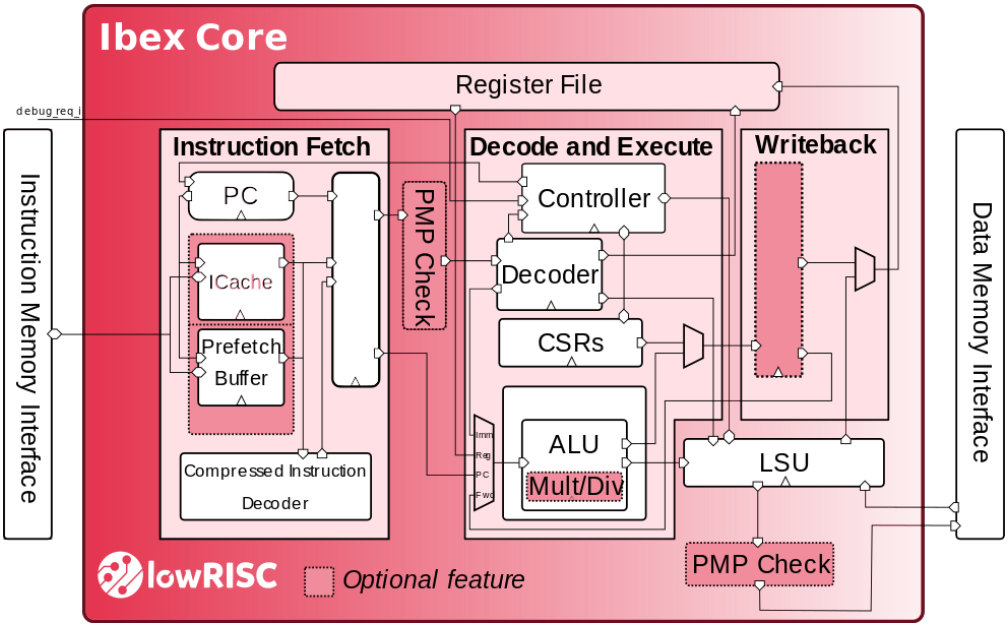

Figure 4.1: lowRISC Ibex core architecture [24]

4.1.2 Multiplier

The multiplier used in the Ibex core and involved in the convolution operations is denoted as RV32MFast. It is a 32-bit integer multi-cycle multiplier/divider which computes integer multiplications in 3-4 clock cycles [27]. Each 32-bit output is computed between *operand_a* and *operand_b* either in signed or unsigned mode. The ATPG process was carried out using the Synopsys TetraMAX tool. The synthetized gate-level unit features a total of 9928 stuck-at faults.

4.1.3 TUL PYNQ-Z2 Board

The TUL PYNQ-Z2 board, based on Xilinx Zynq XC7Z020-1CLG400C SoC, has been designed for the Xilinx University Program to support PYNQ (Python Productivity for Zynq) framework and embedded systems development [28].

The research done in this thesis relies on the X-HEEP architecture implemented over the PYNQ-Z2 board running CNNs.

Indeed, the chosen configuration includes the CV32E20 CPU, 16 memory banks of 32 KBytes each and the one-at-a-time bus topology. This configuration, indeed, allows to achieve a edge computing system with a small memory and a 32-bit RISC-V CPU core.

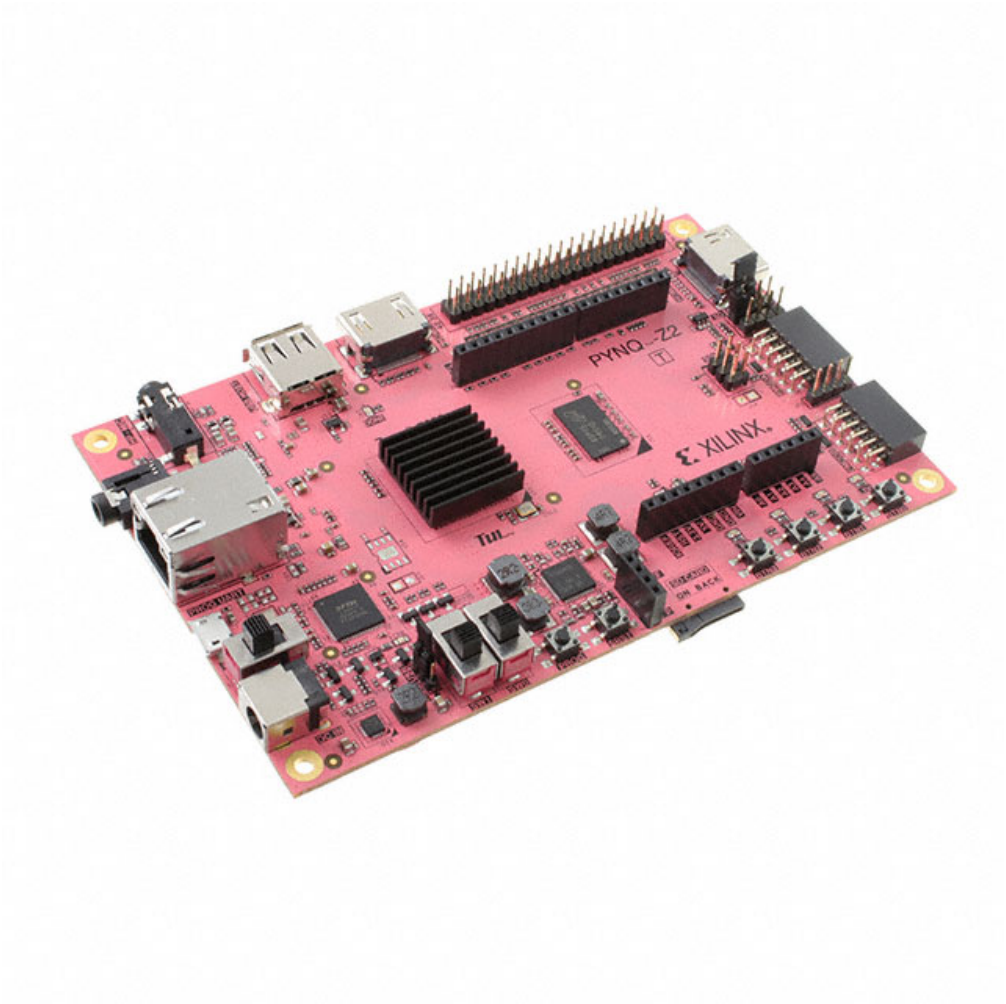

Figure 4.2: PYNQ Z2 board

4.1.4 CNN

An ITL has been generated for the LeNet-5 CNN, which was trained and tested on the CIFAR-10 dataset using Keras. In the first layer of LeNet-5, a convolution with 'valid' padding and a stride of 1 is applied to a $1 \times 32 \times 32$ input image. This operation uses a filter consisting of 6 filters, each with dimensions $1 \times 5 \times 5$, resulting in a total of 150 32-bit Floating Point (FP32) weights. It must be noticed that the model has been quantized to 8-bit Integer (INT8) to be compatible and optimized for the 32-bit integer RV32:MFast Multiplier.

Level	Feature Maps	Dimensions	Connections
Input Layer	-	32 × 32 pixels	-
C1 (Convolution + Tanh)	6	28 × 28	117,600
S2 (Subsampling + Sig)	6	14 × 14	5,880
C3 (Convolution + Tanh)	16	10 × 10	151,600
S4 (Subsampling + Sig)	16	5 × 5	2,000
C5 (Convolution + Tanh)	120	1 × 1	48,000
F6 (Fully Connected + Tanh)	84	-	Fully Connected
F7 (Fully Connected)	10	-	Fully Connected
Softmax	RBF unit	-	-

Table 4.1: LeNet-5 Architecture Breakdown

LeNet-5 architecture

1. **C1 (Convolutional layer + Tanh activation function)**
   - **Feature Maps:** 6 feature maps.
   - **Connections:** each output feature map cell is connected to a 5×5 neighborhood in the input, producing 28×28 output feature maps to prevent boundary effects.
   - **Operation:** convolution operation followed by a Tanh activation function.
2. **S2 (Subsampling layer + Sigmoid activation function)**
   - **Feature Maps:** 6 feature maps.
   - **Connections:** each output feature map cell is connected to several 2×2 neighborhood in the input, producing 14×14 output feature maps.
   - **Operation:** average pooling operation followed by a Sigmoid activation function.
3. **C3 (Convolutional layer + Tanh activation function)**
   - **Feature Maps:** 16 feature maps.
   - **Connections:** each output feature map cell is connected to several 5×5 neighborhoods at identical locations in a subset of S2's feature maps.
   - **Operation:** convolution operation followed by a Tanh activation function.
4. **S4 (Subsampling Layer + Sigmoid activation function)**
   - **Feature Maps:** 16 feature maps.

- **Connections:** each output feature map cell is connected to several 2×2 neighborhood in the input, producing 5×5 output feature maps.
- **Operation:** average pooling operation followed by a Sigmoid activation function.

5. **C5 (Convolutional layer + Tanh activation function)**
   - **Feature Maps:** 120 feature maps.
   - **Connections:** each output feature map cell is connected to a 5×5 neighborhood on all 16 of S4's feature maps.
   - **Operation:** convolution operation followed by a Tanh activation function.
6. **F6 (Fully Connected Layer + Tanh activation function)**
   - **Units:** 84 units.
   - **Connections:** Each output feature map cell is fully connected to C5
   - **Operation:** fully connected operation followed by a Tanh activation function.
7. **F7 (Fully Connected Layer + Softmax activation function)**
   - **Units:** 10 units.
   - **Connections:** Each output feature map cell is fully connected to F6
   - **Operation:** fully connected operation followed by a Softmax activation function which estimates the class predictions.

LeNet-5 training

The LeNet-5 convolutional neural network has been trained using Tensorflow.keras in Python within the Google Colab environment. Specifically, Keras framework and the Mnist dataset have both been involved in order to properly train the network allowing it to achieve an accuracy value around the 96%.

Listing 4.1: LeNet Model Training

```
(x_train, y_train) = datasets.mnist.load_data()
x_train = tf.pad(x_train, [[0, 0], [2, 2], [2, 2]]) / 255
model = Lenet()
model.build(x_train.shape)
EPOCHS = 5
BATCH_SIZE = 128

model.compile(optimizer='adam', loss=losses.
    sparse_categorical_crossentropy, metrics=['accuracy'])
history = model.fit(x_train, y_train, batch_size=BATCH_SIZE, epochs=
    EPOCHS, validation_data=(x_val, y_val))
```

4.2 TensorFlow Lite

TensorFlow Lite is a streamlined version of TensorFlow, an open-source machine learning framework developed by Google, designed to build, train, and deploy machine learning models across a variety of platforms.

TF Lite is tailored for mobile and embedded devices enabling machine learning at the edge, allowing pre-trained models to run efficiently on devices with limited computational resources. It employs optimization techniques such as quantization and pruning, reducing memory usage and processing demands while maintaining a reasonable level of accuracy. TensorFlow Lite supports hardware acceleration and is widely used in real-time applications like object detection, speech recognition, and other AI-driven tasks on low-power devices.

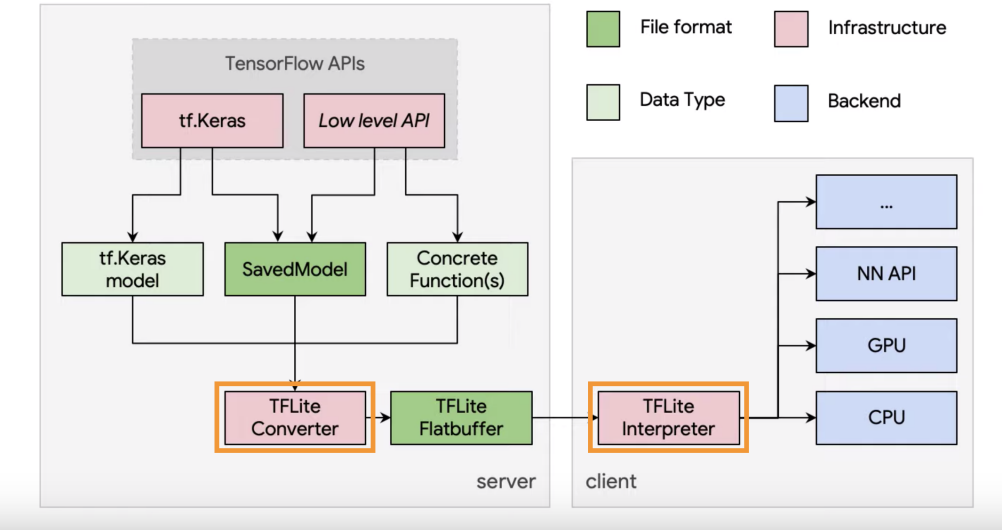

Figure 4.3: TensorFlow Lite toolkit [29]

As shown in Figure 4.3, after training the LeNet-5 model using the TensorFlow API, Keras, TensorFlow Lite successfully converted the model into its 8-bit integer representation using the TFLite Converter, which applied post-training quantization as described in Section 2.4. The TFLite Interpreter, in turn, was used to carry out inference on the quantized LeNet-5 model.

4.3 ITL Generation

As highlighted in Section 3.1, the ITL generation involves three main steps:

```
[
  {"<batch_size>_<out_h_idx>_<out_w_idx>_<n_ch_out>": [
      {
        "<batch_size>,<Input_h_idx>,<Input_w_idx>,<in_ch_in>"
        :
        "<n_ch_out>,<Kernel_h_idx>,<Kernel_h_idx>,<filter_n>"
      },
      ...
    ]
  }
]
```

Listing 4.2: Convolutional Algorithm $\langle i, w \rangle$ pairs

4.3.1 Dataflow Algorithm Extraction

The extraction of the dataflow algorithm focuses on mapping the elements of the output feature map to the corresponding elements of the input feature map and to the specific multipliers responsible for performing the convolutional operations. The process, as explained in Section 2.9.1, consists of three main stages: *thread-core mapping*, *workload-thread mapping* and the *convolutional algorithm*. The first two stages are automatically handled due to the characteristics of the single-core, single-thread X-HEEP architecture. Indeed, the architecture involved in this thesis' case study employs a single 32-bit integer multiplier as the sole computational unit involved in the convolutional operations.

The remaining stage is the one referring to the *convolutional algorithm*. The objective of this stage is, exploiting the Python language, to gather the link between each element of the output feature map and, both, the input feature map element and weight element involved in the multiplication computing the output element. Note that changing this mapping could change the fault propagation and affect the TC of the targeted unit. The result is saved on a .json file and then used during the Self-test Images Generation step. The Listing 4.2 shows an example of the .json file content organization [30].

4.3.2 ATPG-based Pattern Generation

Details of the ATPG process of the multiplier under test are provided in table 4.2. The second column lists the total number of weights used for the convolution in the first convolutional layer, the third column lists the total number of gathered ATPG patterns, while the last one shows the test coverage reached by the ATPG. The ATPG process was configured by imposing constraints on all the weights (for all the 32 bits of the weight), and for all the 24 most significant bits (MSBs) of all the inputs (since the quantized model expects 8-bit signed integer inputs) meaning that only the 8 least significant bits (LSBs) involved in each multiplication are suitable to generate a test pattern. The ultimate aim was to gather a single ATPG pattern for each weight but some weights failed to produce patterns capable of increasing the TC. Indeed, the TetraMAX process achieved a final TC of 86.16% for LeNet-5. Due to the weight and input constraints, 13.82% of the faults

was categorized as ATPG untestable, while 0.001% was categorized as undetectable.

CNN	Number of Weights	Number of ATPG Patterns	Test Coverage (%)
LeNet-5	150	21	86.16

Table 4.2: Summary of ATPG results for LeNet-5

At the end of this procedure, we obtained a test vector of 21 patterns to be placed in an input image [5]. The ATPG patterns are then used to reconstruct the self-test images, as described in Section 3.1.3.

4.3.3 Self-test Images Generation

The ATPG patterns were then used to generate the self-test images, leveraging the information obtained during the dataflow algorithm extraction, as outlined in Section 3.1.3. The LeNet-5 CNN requires a 32 × 32 input image for the first convolutional layer, which provides 1024 potential positions where each pattern could be placed. Since the suitable positions for placing each pattern are related to the weights multiplying the pattern under exam, and consequently the input position the pattern is going to occupy, the Dataflow Algorithm Extraction results were used for determining the suitable locations for each pattern placement.

For the LeNet-5 CNN, the highlighted region in Figure 4.4 indicates the input image locations where all 21 patterns retrieved are multiplied by their corresponding weight. This means that within that area, the patterns could be placed randomly, as they will inevitably be multiplied by their weight (and all the other 20 weights, as well) during the convolutional operation. The 21 patterns have been positioned in the highlighted cells to create the shape of the digit "1", allowing to ensure that the LeNet-5 network, executed on the faulty-free TUL PYNQ-Z2 board, correctly predicts the class "1", as shown by the results in Table 4.3, "Pattern Placement" column.

After placing all the patterns in the IFMAP positions the self-test image generation process proceeds exploiting the inverted image class visualization as described in section 3.1.4. This step considers the IFMAP positions where no patterns are present, enabling the generation of an image balanced across all class predictions. This balance increases the likelihood of triggering a potential fault, leading to an incorrect class prediction. The resulting input image and its class prediction, generated exploiting the IICV method and containing the patterns, is shown in figure 4.5, while the results of the inference are presented in Table 4.3. The image occupancy on disk is 1024 Bytes.

Table 4.3 presents the class prediction percentages for each class when the input image is submitted to the trained LeNet-5 network. The first column from the left displays the predictions for an input image consisting solely of "1"-shaped patterns. The following columns show the results for the non-quantized network performing inference on an IICV input image, and for the quantized network performing inference on an IICV input image.

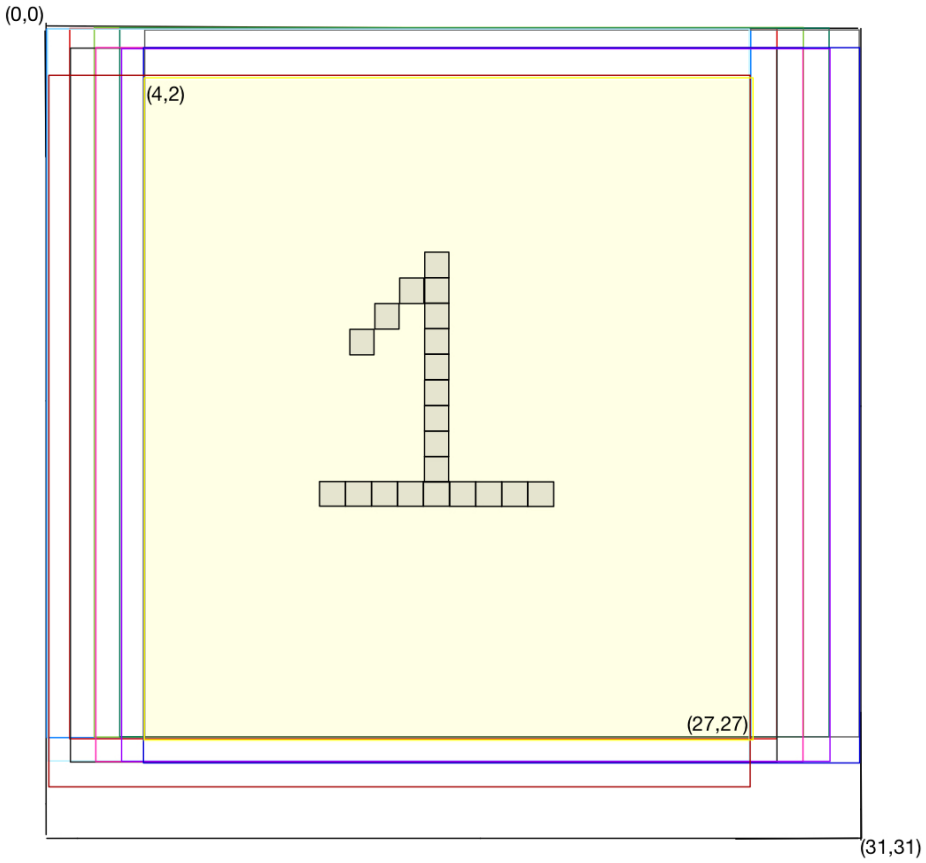

Figure 4.4: LeNet-5 input image patterns positions

4.4 ITL Validation

Capitalizing what discussed in Section 3.2, in order to compute $\hat{O}$ of the first convolutional layer, a simulation needed to be performed by injecting all the stuck-at faults and computing each faulty output $\hat{O}$. The simulation required approximately 11 hours.

Then, to ensure cross-level propagation of hardware faults affecting the RV32MFast multiplier in the LeNet-5 CNN, all the faulty images (computed exploiting the $\hat{M}(I, W) = \hat{O}$ equation) resulting from the 1608 stuck-at-0/1 faults, obtained from gate-level simulations on the multiplier under test, were used as inputs to the CNN model since the $M(\hat{I}, W) = \hat{O}$ equation cannot be used to retrieve the faulty input $\hat{I}$, as outlined in Section 3.2.

Specifically, for the first convolutional layer (*conv1*), faults were injected one by one into the RV32MFast multiplier along with the list of multiplications related to the inference of the ITL image to obtain the resulting faulty outputs. For the first layer, as shown in Figure 4.6, a total of $(5 \times 5 \times 6 \times 28 \times 28) = 117600$ faulty multiplications were performed for each fault.

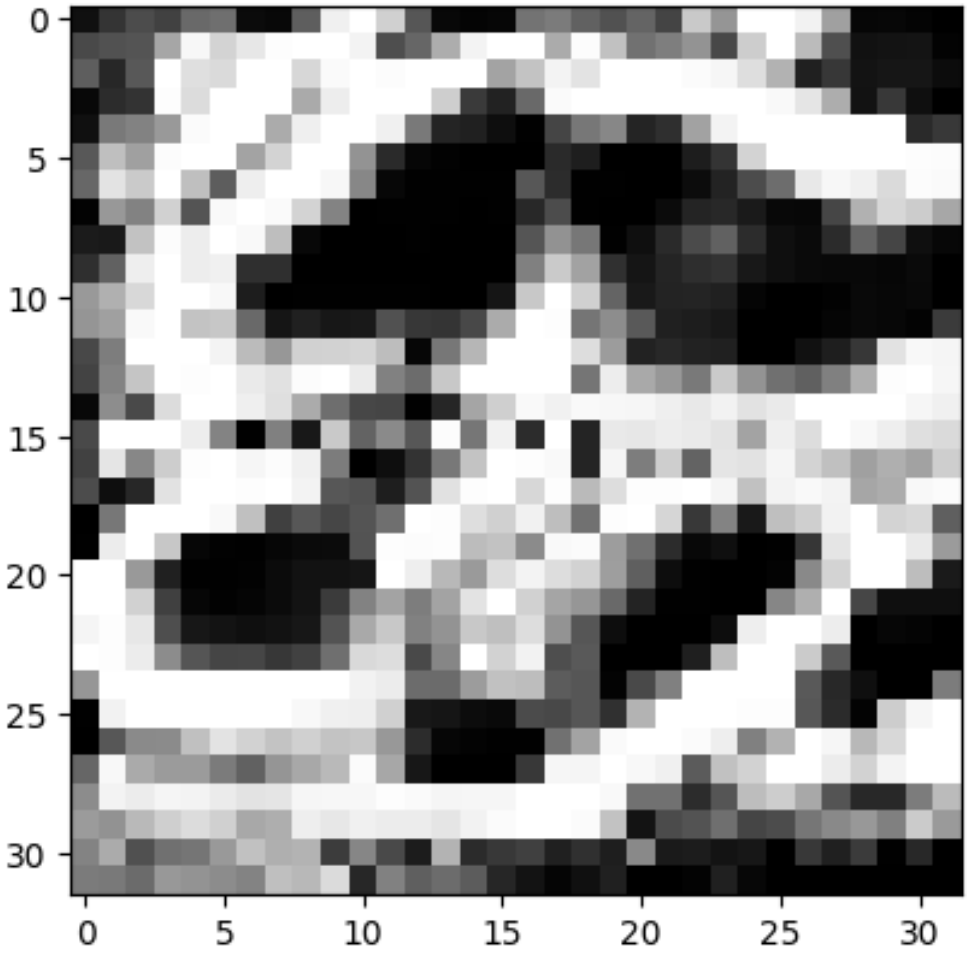

Figure 4.5: LeNet-5 ITL

All the faulty multiplications were then used to compute the faulty output tensor by manually computing the quantized convolution. The output tensor was then fed to the *tanh* and the *avg* layers. The process of extracting multiplications, injecting faults and computing the related output tensor was repeated for each convolutional layer in order to propagate the fault up to the output layer. In total, the process took around 37 hours.

4.5 Results

To validate the ITL generated for the LeNet-5, showed in Figure 4.5, the 3216 faulty predictions, derived from the inference conducted over the fault list of the RV32MFast Multiplier, were compared to the golden output to gather results and metrics.

The metrics chosen to summarize the results of the experiment are:

Class	Patterns Placement	Not-Quant. IICV	Quant. IICV
0	0.00%	10.00	10.16%
1	56.64%	10.00	10.16%
2	14.45%	10.00	10.16%
3	1.17%	10.00	10.16%
4	0.39%	10.00	10.16%
5	7.81%	10.00	9.385%
6	0.00%	10.00	10.16%
7	27.73%	10.00	10.16%
8	0.00%	10.00	9.385%
9	5.07%	10.00	10.16%

Table 4.3: Comparison between plain patterns placement and inverted class visualization inference results for LeNet-5 (both not quantized and quantized)

1. **SDC-1**, it represents the percentage of faults which cause a top class prediction change with respect to the fault-free inference (also known as, golden output).
2. **SDC-3**, it represents the percentage of faults which cause a top 3 classes prediction change with respect to the fault-free inference (also known as, golden output).
3. **SDC-10%**, it represents the percentage of faults which cause the confidence score to vary of ± 10% with respect to the fault-free inference (also known as, golden output).

   The *confidence score* is a number between 0 and 1 which represents the likelihood that the output of a CNN is correct. It is represented by the highest class prediction score.

	SDC-1	SDC-3	SDC-10%
LeNet-5	93.22%	96.83%	95.55%

Table 4.4: LeNet-5 coverage metrics

As reported in the Table 4.4, the generated ITL allows to successfully identify the 93.22% of the faults detected for the architecture under test without altering the existing CNN or performing expensive memory operations.

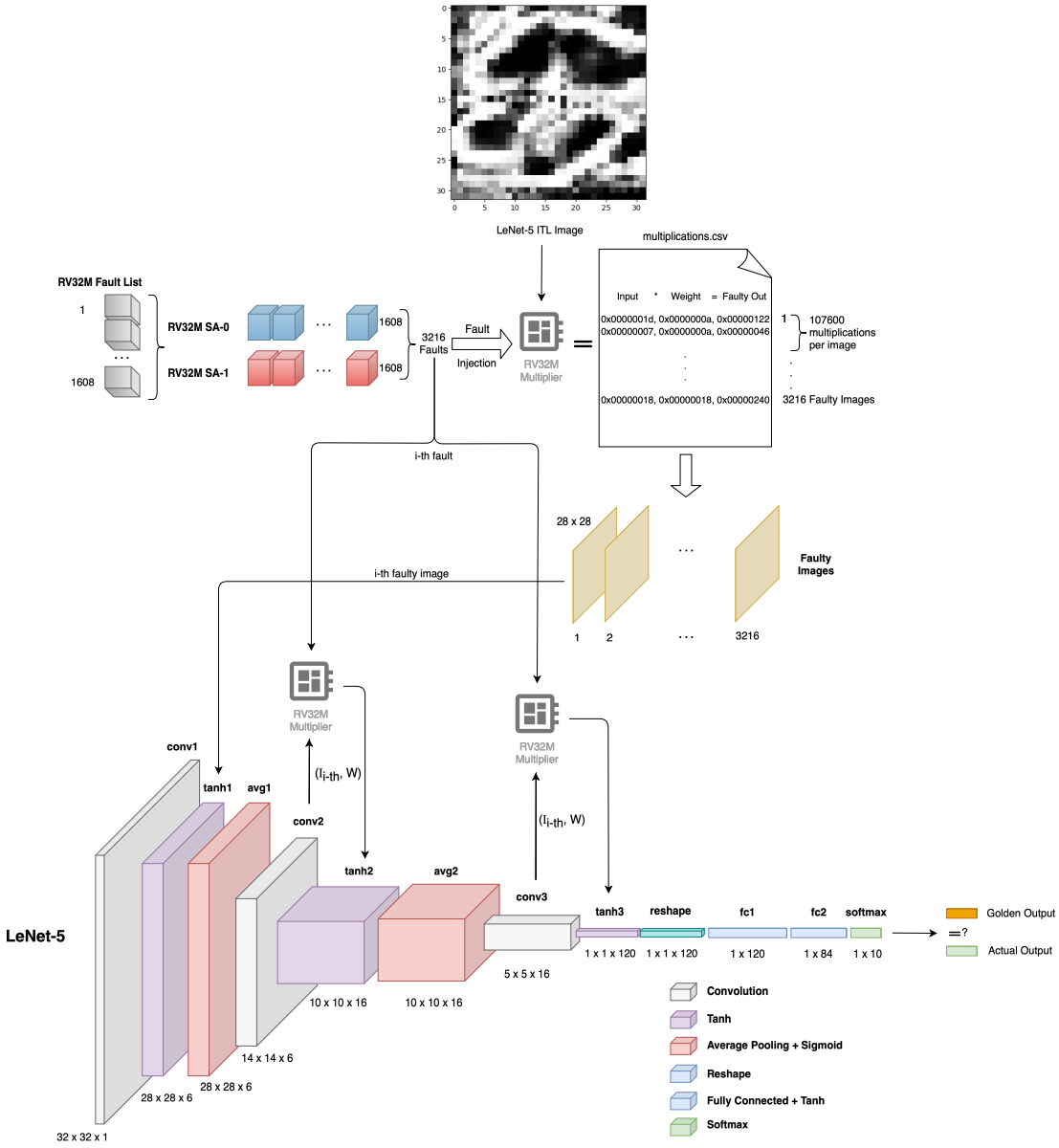

Figure 4.6: LeNet-5 ITL Validation process

Chapter 5

Conclusions

This thesis introduces a novel method for generating test images that can effectively detect stuck-at faults in the multipliers of edge devices in real-time and in-field scenarios. We have demonstrated that a very limited set of images, specifically one image for the LeNet-5 CNN, is capable of covering approximately 93.22% of permanent stuck-at faults. This approach not only minimizes self-test time but also requires low memory resources for storing the Input Test Lists (ITLs), making it particularly advantageous for resource-constrained environments.

Looking ahead, future research will focus on extending this method to more complex convolutional neural networks (CNNs), such as ResNet and AlexNet. These advanced networks present unique challenges and opportunities that could further enhance fault detection capabilities. Additionally, this work could be improved by exploring different types of faults beyond stuck-at faults, including timing faults, which can have significant implications for the performance and reliability of edge devices.

A key observation in this research is that edge devices, due to their limited memory and CPU capacities, could benefit from the implementation of quantized neural networks even if it results in a slight reduction in precision. This strategy would enable the hardware to operate more efficiently while still leveraging the power of neural networks. Moreover, the ITLs developed in this study are tailored specifically for integer multiplier units. We plan to adapt and refine this technique for application in other computational and logic units, thereby broadening the scope and impact of our findings. This ongoing research will contribute to the advancement of fault detection methods in edge devices, enhancing their reliability and performance in real-world applications.

Bibliography

[1] Paolo Bernardi, Riccardo Cantoro, Sergio De Luca, Ernesto Sánchez, and Alessandro Sansonetti. «Development Flow for On-Line Core Self-Test of Automotive Microcontrollers». In: *IEEE Transactions on Computers 65.3* (2016), pp. 744–754. DOI: 10.1109/TC.2015.2498546 (cit. on pp. 1, 22).

[2] Stefano Di Carlo, Giulio Gambardella, Marco Indaco, Ippazio Martella, Paolo Prinetto, Daniele Rolfo, and Pascal Trotta. «A software-based self test of CUDA Fermi GPU». In: *2013 18th IEEE European Test Symposium (ETS)* (2013), pp. 1–6. DOI: 10.1109/ETS.2013.6569353 (cit. on pp. 1, 23).

[3] Josie E. Rodriguez Condia et al. «Using STLs for Effective In-field Test of GPUs». In: *IEEE Design and Test* (2022), pp. 1–1. DOI: 10.1109/MDAT.2022.3188573 (cit. on pp. 1, 23).

[4] A. Ruospo, D. Piumatti, A. Floridia, and E. Sanchez. «A Suitability Analysis of Software Based Testing Strategies for the On-line Testing of Artificial Neural Networks Applications in Embedded Devices». In: *2021 IEEE 27th International Symposium on On-Line Testing and Robust System Design (IOLTS)* (2021), pp. 1–6. DOI: 10.1109/IOLTS52814.2021.9486704 (cit. on pp. 1, 23).

[5] A. Ruospo, G. Gavarini, A. Porsia, M. Sonza Reorda, E. Sanchez, R. Mariani, J. Aribido, and J. Athavale. «Image Test Libraries for the on-line self-test of functional units in GPUs running CNNs». In: *2023 IEEE 28th European Test Symposium (ETS)* (2023), pp. 1–6. DOI: 10.1109/ETS56758.2023.10174176 (cit. on pp. 1, 23–25, 27–30, 43).

[6] Giuseppe Desoli et al. «14.1 A 2.9TOPS/W deep convolutional neural network SoC in FD-SOI 28nm for intelligent embedded systems». In: *2017 IEEE International Solid-State Circuits Conference (ISSCC)*. 2017, pp. 238–239. DOI: 10.1109/ISSCC.2017.7870349 (cit. on pp. 2, 5).

[7] Simone Machetti, Pasquale Davide Schiavone, Thomas Christoph Müller, Miguel Peón-Quirós, and David Atienza. *X-HEEP: An Open-Source, Configurable and Extendible RISC-V Microcontroller for the Exploration of Ultra-Low-Power Edge Accelerators*. 2024. arXiv: 2401.05548 [cs.AR] (cit. on pp. 2, 15, 17).

[8] Andrej Karpathy, Fei-Fei Li, and Ehsan Adeli. *Stanford CS231n: Deep Learning for Computer Vision*. Accessed: 2024-10-01. 2024. URL: https://cs231n.github.io/convolutional-networks/#case (cit. on pp. 3, 11).

[9] Ian J. Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. Cambridge, MA, USA: MIT Press, 2016 (cit. on pp. 3, 12).

[10] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. «Gradient-based learning applied to document recognition». In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324. DOI: 10.1109/5.726791 (cit. on pp. 4, 36).

[11] Belajar Pembelajaran. *Student Notes: Convolutional Neural Networks (CNN) Introduction*. Accessed: 2024-10-01. 2018. URL: https://indoml.com/2018/03/07/student-notes-convolutional-neural-networks-cnn-introduction/ (cit. on pp. 4, 7–11).

[12] Hong Jing. *How Convolutional Layers Work in Deep Learning Neural Networks?* Accessed: 2024-10-01. URL: https://jinglescode.github.io/2020/11/01/how-convolutional-layers-work-deep-learning-neural-networks/ (cit. on p. 6).

[13] Dwith Chenna. *Quantizing Convolutional Neural Networks for Practical Deployment*. Accessed: 2024-10-01. URL: https://www.edge-ai-vision.com/2023/12/from-theory-to-practice-quantizing-convolutional-neural-networks-for-practical-deployment/ (cit. on p. 13).

[14] Pasquale Davide Schiavone et al. «X-HEEP: An Open-Source, Configurable and Extendible RISC-V Microcontroller». In: *Proc. of Int. Conf. on Computing Frontiers. CF '23. New York* (2023), pp. 379–380. DOI: 10.1145/3587135.3591431 (cit. on p. 14).

[15] Pasquale Davide Schiavone et al. «Slow and steady wins the race? A comparison of ultra-low-power RISC-V cores for Internet-of-Things applications». In: *Int. Symp. on Power and Timing Modeling, Optimization and Simulation (PATMOS). IEEE.* (2017), pp. 1–8 (cit. on pp. 14, 15).

[16] *CORE-V X-Interface*. Accessed: 2024-10-01. URL: https://github.com/openhwgroup/core-v-xif. (cit. on p. 15).

[17] M. Bushnell and Vishwani Agrawal. «Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits». In: *Springer Publishing Company, Incorporated* (2013) (cit. on pp. 17–19, 22, 28).

[18] T. W. Williams and K. P. Parker. «Design for Testability - A Survey». In: *IEEE Transactions on Computers* C-31.1 (1972), pp. 2–15. DOI: 10.1109/TC.1982.1675879 (cit. on p. 20).

[19] Nicolaidis M. and Zorian Y. «On-Line Testing for VLSI—A Compendium of Approaches». In: *Journal of Electronic Testing* 12 (1998), pp. 7–20. DOI: 10.1023/A:1008244815697 (cit. on p. 20).

[20] Annachiara Ruospo, Ernesto Sanchez, Lucas Matana Luza, Luigi Dilillo, Marcello Traiola, and Alberto Bosio. «A Survey on Deep Learning Resilience Assessment Methodologies». In: *IEEE* (2023). DOI: 10.1109/MC.2022.3217841 (cit. on p. 21).

[21] Paul H. Bardell, W. H. McAnney, and J. Savir. «Built-In Test for VLSI: Pseudorandom Techniques». In: *John Wiley & Sons* (1987) (cit. on p. 22).

[22] Priyanka Viswanathan, Antonio Priore, and Alexander Griessing. *State of the Art Software Test Libraries (STL) and ASIL B: Truths, Myths, and Guidance*. Accessed: 2024-10-01. 2024. URL: https://armkeil.blob.core.windows.net/developer/Files/pdf/white-paper/state-of-the-art-stl-and-asil-b.pdf (cit. on p. 23).

[23] V.D. Agrawal, A.V.S.S. Prasad, and M.V. Atre. «Fault collapsing via functional dominance». In: *International Test Conference, 2003. Proceedings. ITC 2003.* Vol. 1. 2003, pp. 274–280. DOI: 10.1109/TEST.2003.1270849 (cit. on p. 28).

[24] *lowRISC Ibex*. Accessed: 2024-10-01. 2024. URL: https://github.com/lowRISC/ibex?tab=readme-ov-file (cit. on pp. 31, 37).

[25] Sarah A. El-Sayed, Theofilos Spyrou, Luis A. Camuñas-Mesa, and Haralampos-G. Stratigopoulos. «Compact Functional Testing for Neuromorphic Computing Circuits». In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 42.7 (July 2023). ISSN: 1937-4151. DOI: 10.1109/TCAD.2022.3223843 (cit. on p. 32).

[26] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. *Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps*. 2014. arXiv: 1312.6034 [cs.CV]. URL: https://arxiv.org/abs/1312.6034 (cit. on pp. 32, 33).

[27] *Ibex Reference Guide*. Accessed: 2024-10-11. URL: https://docs.openhwgroup.org/projects/cve2-user-manual/en/latest/03_reference/index.html (cit. on p. 37).

[28] *TUL FPGA - PYNQ Z2*. Accessed: 2024-10-01. URL: https://www.tulembedded.com/FPGA/ProductsPYNQ-Z2.html (cit. on p. 37).

[29] Vittorio Mazzia. *Intro to TensorFlow Lite*. Accessed: 2024-10-01. URL: https://vittoriomazzia.com/tensorflow-lite/ (cit. on p. 41).

[30] G. Perlo. *XHEEP ITL*. Accessed: 2024-10-01. 2024. URL: https://github.com/jackperlo/XHEEP_ITL (cit. on p. 42).