POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Image Test Libraries for the on-line self-test of functional units in GPUs running CNNs

Supervisors

Prof. Ernesto SANCHEZ

Dr. Annachiara RUOSPO

Dr. Gabriele GAVARINI

Candidate

Antonio PORSIA

October 2023

Abstract

The widespread use of artificial intelligence (AI)-based systems brings with it several questions about the deployment of such systems in safety-critical contexts. Several industry standards exist, such as ISO26262 for automotive, that require detecting hardware faults during the mission of the device. Similarly, new standards are being released concerning the functional safety of AI systems (e.g., ISO/IEC CD TR 5469). Hardware solutions have been proposed for the in-field testing of the hardware executing AI applications, but when used in conjunction with complex applications such as Convolutional Neural Networks (CNNs) in image processing tasks, they may increase the hardware cost and affect the application performances. In this thesis, a methodology to develop high-quality test images, to be interleaved with the normal inference process of the CNN application is proposed. An ITL that targets GPU single-precision floating-point multipliers is developed with the aim of performing an on-line test of said functional units. The proposed approach does not require changing the actual CNN (thus incurring in very costly memory operations) since it is able to exploit the actual CNN structure. In particular, the ITL is built to exploit the convolution operation between an input image and a series of filters, which consists of multiply-and-add operations, to pass test patterns generated beforehand to multipliers. Since the fundamental objective is to keep the CNN structure, and thus also the weights, unchanged, the only elements that can be manipulated are the input images. For this reason, test patterns for a multiplier must be generated with methods exploiting Automatic Test Pattern Generation (ATPG) techniques, putting the already trained network weights as constraints. The generated test patterns are placed into the right spots of the input image (or images), where it is guaranteed that a certain multiplier will multiply them by the weight used as constraint. The main issue that arises at this point is ensuring the correct placement of test patterns into the ITL, which depends on the scheduling algorithm of the GPU and the convolution algorithm. In particular, the thesis work consisted of analyzing existing implementations of convolution algorithms such as GEMM (General Matrix Multiplication), analyzing the GPU scheduling policy and developing an algorithm that exploits this knowledge to correctly predict, given an input pixel-weight pair, which multiplier will perform that multiplication. The experiments that have been performed on the first layer of a ResNet-20 CNN and a DenseNet-121 CNN, show that a 6/8-image ITL is able to achieve about 95% of stuck-at test coverage on the single-precision floating-point multipliers in a GPU. The obtained ITL requires a very low test application time and has a very low memory footprint, needing space only to store the test images and the golden test responses.

Table of Contents

List of Tables

List of Figures

Acronyms

AI	artificial intelligence
ATPG	Automatic Test Pattern Generation
BIST	Built-In Self-Test
BLAS	Basic Linear Algebra Subprograms
CNN	Convolutional Neural Network
DL	Deep Learning
DNN	Deep Neural Network
DUT	Device Under Test
EDA	Electronic Design Automation
FC	Fault Coverage

FI	Fault Injection
FPGA	Field Programmable Gate Array
GEMM	General Matrix Multiplication
GPC	Graphics Processing Cluster
GPGPU	General-Purpose computing on GPUs
GPU	Graphics Processing Unit
HDL	Hardware Description Language
ITL	Image Test Library
LFSR	Linear Feedback Shift Register
MLP	Multi Layer Perceptron
PI	primary input
PO	primary output
RTL	register-transfer level

SBST	Software-Based Self-Test
SIMT	Single Instruction, Multiple Threads
SM	Streaming Multiprocessor
STL	Software Test Library
TC	Test Coverage

Chapter 1

Introduction

The ever-increasing adoption of artificial intelligence (AI)-based solutions in modern systems labeled as safety-critical is requiring the academic and industrial communities to increase their efforts to guarantee higher reliability for these products. Among AI solutions based on Deep Learning techniques, those based on Convolutional Neural Networks (CNNs) are among the most used for their outstanding capabilities in computer vision tasks like image classification, object detection and object localization. In the last years, many standards have been proposed to guide the adoption of different mechanisms to face reliability issues. For example, the ISO 26262 standard is commonly followed in the automotive industry. Similarly, new standards are being released concerning the functional safety of AI systems (e.g., ISO/IEC CD TR 5469).

Among the possible solutions adopted in the different fields, 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 is performed through the periodic execution of Software Test Libraries (STLs) composed of a set of assembly programs able to thoroughly excite the processor core and detect possible permanent faults. STL solutions allow the system to perform on-line tests and do not require any hardware overhead since they only need memory space to save test libraries.

STLs have been proposed as an effective safety mechanism to test 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 interleaved to CNN inferences may negatively affect performance [4].

Recently, an in-field testing solution for testing Deep Learning (DL) accelerators has been suggested in [5]. As a case study, they exploit NVDLA, an open-source

Nvidia DL accelerator. Their technique resorts to combinational Automatic Test Pattern Generation (ATPG) to generate functional test patterns to detect permanent faults in both computational and logic units. These patterns depend on the specific targeted unit and consist of sets of {input, weight} pairs mapped to one or more Deep Neural Network (DNN) test programs. In particular, each test pattern generated for multiply-and-accumulate (MAC) and accumulation (ACC) compute units is mapped to a separate DNN test program, resulting in thousands of test programs. The flaw in this approach is that the execution of thousands of DNN test programs requires a non-negligible time for context switching, and, above all, can only be performed during dead times, i.e., boot or reset. Additionally, the total test storage can require, in some cases, up to 600 MB for a single unit.

This thesis describes a method to feed test patterns to an already trained CNN during its execution time via carefully generated test images. In particular, this method exploits the convolution operation between an input image and a set of filters, since convolutional layers in CNNs account for more than 90% of the total operations [6]: test patterns are generated and placed in input images such that, when the convolution is performed, permanent faults affecting the target hardware unit can be detected and a high test coverage can be reached. The idea comes mainly from the following observation: when a CNN is deployed in the field, the trained version is loaded, and weights remain always the same. Thus, carefully developed test images can be periodically fed to the *same* CNN to test on-line specific hardware units. Since the network remains untouched and the inference process is fast, this method allows to perform an on-line self-test without interfering with the network's operation. A comparison mechanism is then adopted to possibly alert for the presence of a fault. The purpose of the proposed method is to generate a set of images – an Image Test Library (ITL) – for the on-line test of multipliers in a GPU, which have a relevant role in convolutional operations. Experimental results reveal that with a fairly small set of test images, this technique achieves about 95% single stuck-at test coverage for all GPU multipliers. As a case study, two ITLs have been developed for two CNNs: ResNet-20 and DenseNet-121. In addition, it is experimentally demonstrated that the developed ITLs can propagate the effect of the faults up to the software level.

The thesis is organized as follows: chapter 2 provides some background knowledge about GPUs, convolutional neural networks and digital circuit testing. With regards to CNNs, the main focus is on the convolution operation and convolution algorithms. Chapter 3 describes the proposed approach to generate ITLs and validate them. Then, Chapter 4 reports experimental results obtained on a real GPU. Finally, Chapter 5 draws conclusions and future directions.

Chapter 2

Background

This chapter gives an overview of the concepts used in subsequent chapters. Section 2.1 gives a quick introduction on data parallelism and describes the inner mechanisms of modern Graphics Processing Units (GPUs). Section 2.2 introduces Convolutional Neural Networks (CNNs) and describes in detail the convolution operation, as well as convolutional algorithms. Finally, Section 2.3 gives an overview about digital circuit testing techniques used in this work.

2.1 Graphics Processing Units

In recent years, the scope of application of GPUs has extended beyond graphics processing, even more so after the introduction of platforms, such as Nvidia CUDA, for General-Purpose computing on GPUs (GPGPU). In fact, before these platforms were available, GPUs were found to be useful for general-purpose tasks such as linear algebra operations [7], but problems had to be expressed in terms of graphics primitives using pixel or vertex shader languages. GPGPU platforms instead offer an API to access GPU resources using general-purpose programming languages such as C and C++. The availability of these platforms, in conjunction with the highly parallelizable nature of Deep Neural Network (DNN) computations, has put GPUs among the most popular hardware accelerators used for efficient training and deployment of DNNs.

2.1.1 Data parallelism

Data parallelism is a property that expresses the capability of a task to be decomposed into subtasks that can execute the same operation on different data in parallel, independently from each other. For example, vector addition exhibits a high level of data parallelism, since each element of the output vector can be computed independently from the others. On a multi-core machine, one could write a program that spawns a thread for each output element, with each thread executing the same operation (addition of two values), but with different inputs.

Task parallelism instead, expresses the capability of a task to be decomposed into subtasks that can execute a different sequential operation in parallel with the others, e.g., drawing a GUI in one thread and performing blocking I/O operations in another.

CPUs are designed to optimize sequential execution of a single thread by employing strategies such as large multi-level cache memories, out-of-order execution, higher clock frequencies and so on. In this way, CPUs are able to efficiently execute sequential tasks, eventually in parallel with a few others, at the expense of the number of cores. It follows that CPUs are particularly efficient for computations that present a high level of task parallelism, but struggle with tasks exhibiting a high level of data parallelism.

GPUs on the other hand, are designed to maximize the execution throughput by enabling the parallel execution of a massive number of threads, at the expense of complex control logic, memory access hardware and arithmetic units performance [8, pp. 3–5]. In particular, GPUs adopt an execution model defined as *Single Instruction, Multiple Threads (SIMT)*, in which groups of threads execute the same instruction in lockstep, but with different data. This approach eliminates the need to keep track of the execution state[1], i.e., program counter and call stack, for each thread, as well as the need to keep separate instruction caches. However, this approach also entails that each individual GPU core becomes slower than its CPU counterpart [9]. However, the large number of threads allows the GPU hardware to mask memory and arithmetic latencies by finding other threads that need to perform work. It follows that GPUs are particularly suitable to execute tasks with a high level of data parallelism.

2.1.2 Nvidia GPU organization

Recent Nvidia GPUs feature a number of *Graphics Processing Clusters (GPCs)*, which are the highest-level block in the GPU hardware hierarchy. Each GPC contains multiple physically close *Streaming Multiprocessors (SMs)*, in charge of most of the operations performed by the device. A single SM contains a certain number of processing blocks, each with a dedicated *warp scheduler*, in charge of scheduling groups of 32 threads called *warps*, the fundamental SIMT units of the GPU. Each processing block in turn contains a certain number of CUDA cores, responsible for integer and floating-point operations, Load/Store Units and Special

[1]Note that this is not entirely true for newer architectures, e.g., the Nvidia Volta architecture introduced Independent Thread Scheduling

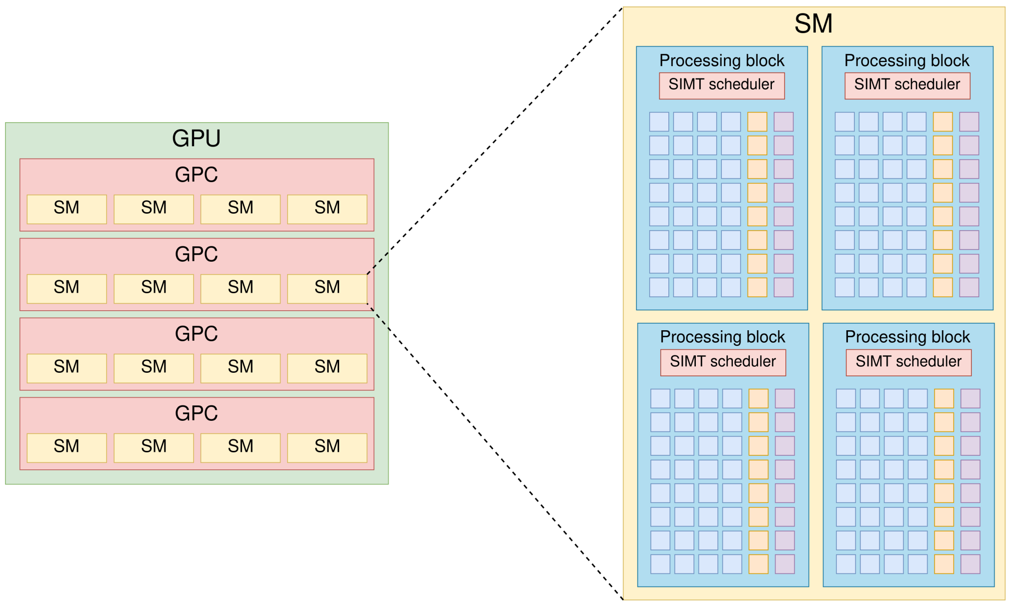
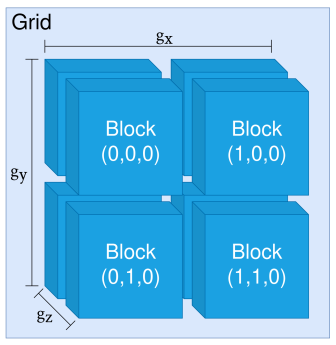
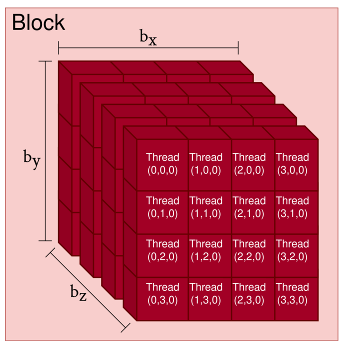


Figure 2.1: Nvidia GPU structure. Within a processing block, CUDA cores are represented in blue, Load/Store Units in Yellow and SFUs in purple.



(a) Grid



(b) Block

Figure 2.2: CUDA Thread Organization

Function Units (SFUs), responsible for the computation of transcendental functions. Figure 2.1 reports a graphical representation of this organization.

CUDA Programming Model The Nvidia CUDA platform offers a C/C++ extension that allows programmers to write functions, called *kernels*, to be executed on the GPU. The body of a kernel function is executed N times by N threads, where N is derived from the *execution configuration* of the specific kernel launch. The execution configuration can contain various parameters, but the most important

```
__global__ void vecAdd(float *a, float *b, float *c) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    c[i] = a[i] + b[i];
}

...
// Array definition. For simplicity, it is assumed
// they have already been allocated in GPU device memory
float a[512], b[512], c[512];
...
dim3 blocks(4,1,1);    // Grid size
dim3 threads(128,1,1); // Block size

vecAdd<<<grid_size, block_size>>>(a, b, c); // Kernel launch
```

Listing 2.1: Example of kernel definition and kernel launch

are *grid size* and *block size*.

CUDA threads are organized in 3-dimensional *grids* of 3-dimensional *blocks* of threads. Grid size specifies the number of blocks in a grid, while block size specifies the number of threads in a single block. Given grid size $\mathbf{G} = (g_x, g_y, g_z)$ and block size $\mathbf{B} = (b_x, b_y, b_z)$, the total number of threads launched on the device is given by $N = g_x g_y g_z \cdot b_x b_y b_z$.

Each block is assigned a unique 3-dimensional ID within the grid, accessible inside kernel functions via the `blockIdx` symbol, containing the fields `x`, `y` and `z`. Threads have an analogous unique ID within the block they belong to, accessible inside kernel functions via the `threadIdx` symbol. Grid size and block size are accessible via `gridDim` and `blockDim`, respectively.

Kernels specify the actions that will be executed by each thread, as well as the data they will be executed on. For example, a vector addition between two 512-element vectors can be defined as in Listing 2.1, dividing the computation between 4 blocks of 128 threads. As it can be observed in the example, the input data that will be processed by a thread is usually identified using block and thread indices.

SIMT Execution Model As outlined before, GPUs handle data parallelism by executing a large number of threads and using the SIMT execution model and Nvidia GPUs are no exception.

When a kernel is launched, thread blocks are distributed among the available SMs and partitioned into warps, each comprising up to 32 threads. This policy results in threads within the same thread block being always executed on the same SM. Subsequently, as an SM executes a thread block, each warp is dispatched to a warp scheduler, which issues instructions to the CUDA cores of the corresponding processing block.

The warp scheduler issues, in SIMT fashion, the *same*[2] instruction for each of the 32 threads in the warp, meaning that threads in the same warp execute in lockstep. In case of thread divergence, e.g., a branch taken only by some threads, the threads that do not participate in the branch are temporarily masked while the others execute instructions inside the branch. When the execution flow of all threads within the warp reconverges, lockstep execution is resumed. This behavior, in conjunction with the fact that a processing block cannot simultaneously execute instructions from two different warps, implies that in case of thread divergence, some cores do not perform useful work.

Memory hierarchy Since data parallelism implies handling large quantities of data, another aspect of GPU design covers the memory hierarchy. GPU accelerated applications often require high memory bandwidth to keep up with the large number of threads that may want to perform a memory access. GPUs come with several gigabytes of GDDR (Graphic Double Data Rate) DRAM, a type of memory specifically devised to offer higher memory bandwidth than ordinary DDR DRAM. In the context of Nvidia GPUs, this off-chip memory is referred to as *global memory* and is shared between all SMs within the device. When using CUDA, the host application can pass input data to a kernel by transferring it to global memory. Conversely, kernels store eventual output data into global memory, so that the host application may transfer it back. Even if it features a higher bandwidth than standard memory, global memory is still not sufficient to obtain a high execution throughput. The facts that it (i) resides off-chip, (ii) is shared among all SMs within the device, (iii) has to accommodate requests from a large number of threads and (iv) has a long access latency, may cause the bandwidth to rapidly saturate. For this reason, two types of on-chip memory are available.

Shared memory is a medium-sized on-chip memory area that features higher bandwidth and lower access latency than global memory, enabling very fast parallel accesses. Shared memory owes its name to the fact that it is shared among all threads in the same thread block. Its higher performances with regard to global memory and its data sharing capability make it the ideal choice to transfer frequently-accessed data from global memory and/or to store intermediate computation results.

Registers fulfill roughly the same tasks as CPU registers and are the second type of on-chip memory available in a GPU. They have an even higher bandwidth and lower access latency than shared memory, but are private to each thread and

[2]Except obviously for the input data of the instruction

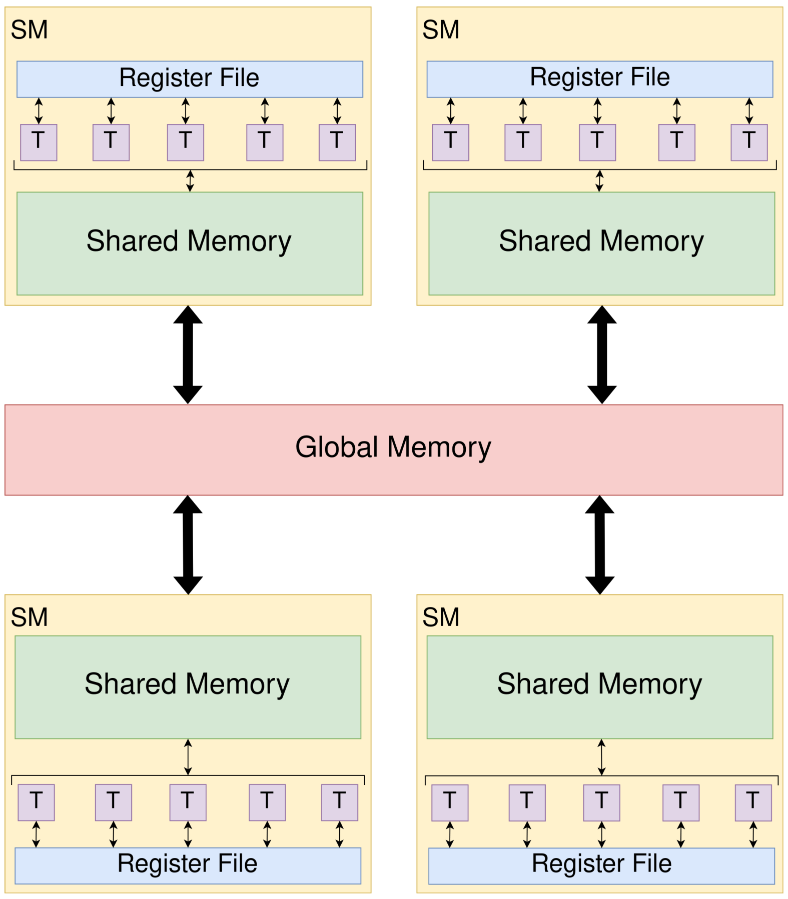


Figure 2.3: Nvidia GPU Memory Hierarchy

there is a limit on how many registers a thread can use. Registers are contained in a *Register File*, one per *SM*.

Threads also have access to another memory area called *local memory*. It is not a separate region, but an area of global memory reserved to each thread to hold arrays declared inside kernels and hold spilled register[3].

2.2 Convolutional Neural Networks

A *Convolutional Neural Network*, or CNN, is an artificially implemented neural network that is specialized in processing data with grid-like structures, such as

[3]Register spilling consists in temporarily moving the content of some registers in memory when there are not enough registers in the Register File to accommodate all the data a thread needs

time-series data, which can be considered as a 1D grid of samples taken at regular intervals, and grayscale images, which can be considered as 2D grids of pixels [10, p. 326].

While in classical feed-forward neural networks, or *Multi Layer Perceptrons (MLPs)* [11], each neuron has its own set of weights, i.e., its own set of trainable parameters, each convolutional layer of a CNN has a single set of filters, or kernels, each containing a set of weights. Each filter scans all input data focusing on a small section at a time, using the same weights for each section. This approach has three important properties [10, pp. 329–335]:

1. *Sparse connectivity* – in a MLP layer, the output of each neuron depends on the output of all the neurons of the previous layer and is fed to every neuron of the next, i.e., each neuron is connected to every neuron of the previous layer and every neuron of the next. In the context of CNNs, such a layer is defined as a *Fully Connected* (FC) layer and will be referred as such from now on.

   Each element of the output of a convolutional layer depends (at most) on a number of input elements equal to the size of the filter.

   If the input has size m, the output has size n and the filter has size k, the computational complexity of calculating the output of a FC layer is $O(mn)$, while for a convolutional layer it is $O(kn)$. While m and n are usually the same order of magnitude, k can be kept several orders of magnitude smaller than both, resulting in better performance.

2. *Parameter sharing* – in a FC layer, each neuron has its own set of weights different from all the other neurons, i.e., each weight gets multiplied by a single input element and never reused.

   In a convolutional layer, each weight of the filter gets multiplied by almost every input element. This allows us to reduce the memory footprint of the layer: while a FC layer requires to store $m \cdot n$ parameters, a convolutional layer requires to store only k parameters.

3. *Equivariance to translation* – this property is a consequence of the form of parameter sharing that takes place in a CNN. A function is said to be equivariant to translation if a translation applied to the input results in the same translation applied to the output.

   For example, if all the pixels of an image are shifted one place to the right and then a convolution is performed, the output will be the same as if the convolution was performed on the original image and all the pixels of the output were subsequently shifted to the right.

2.2.1 Convolution

The fundamental operation of a CNN is the *convolution*, an operation where a filter, or kernel, is used to extract a set of *features* from input data. Input data and extracted features are called *input feature map* and *output feature map*, respectively. Mathematically speaking, what in the context of CNNs is called convolution is actually defined as *cross-correlation*, an operation which measures the similarity between two signals at different time lags.

CNNs designed to operate on RGB images work on multidimensional grid-like objects called *tensors*. An RGB image can be represented as a 3-dimensional tensor $I \in \mathbb{R}^{C \times H \times W}$, where $C = 3$ is the number of channels and H and W are height and width of the image, respectively. Usually, especially during training, the input of a CNN is not a single image, but a *batch* of N images, resulting in a 4-dimensional tensor $I \in \mathbb{R}^{N \times C \times H \times W}$.

A single filter operating on a single 3D input feature map will itself be a 3-dimensional tensor, with a number of channels equal to the number of channels of the input feature map. Since a single filter can only extract a single feature from the input, multiple filters can be used on the same input. This results in a 4-dimensional tensor $F \in \mathbb{R}^{M \times C \times H_F \times W_F}$, where M is the number of different 3-dimensional filters or, in other words, the number of different features that one may want to extract from the input.

Before examining how a convolutional layer handles these objects, the convolution operation as it is defined for 1D and 2D input data will be introduced.

1D convolution

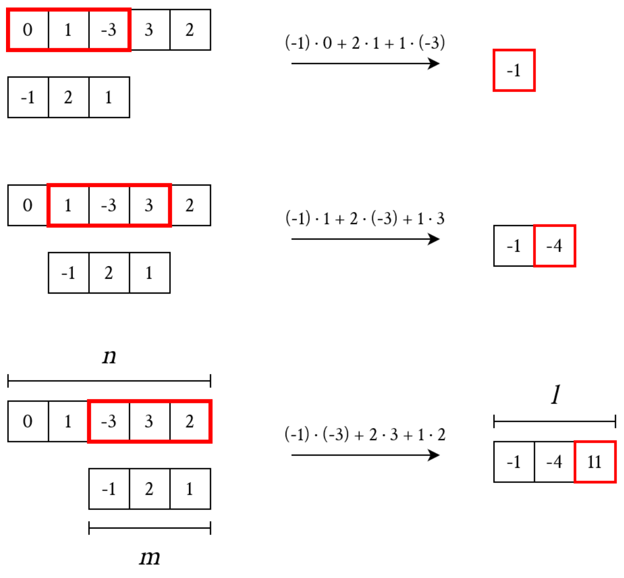
Let $I \in \mathbb{R}^n$ be a 1D input feature map of length n and $F \in \mathbb{R}^f$ a 1D filter of length f. The length l of the output feature map $O \in \mathbb{R}^l$ is defined as:

$$l = n - f + 1$$

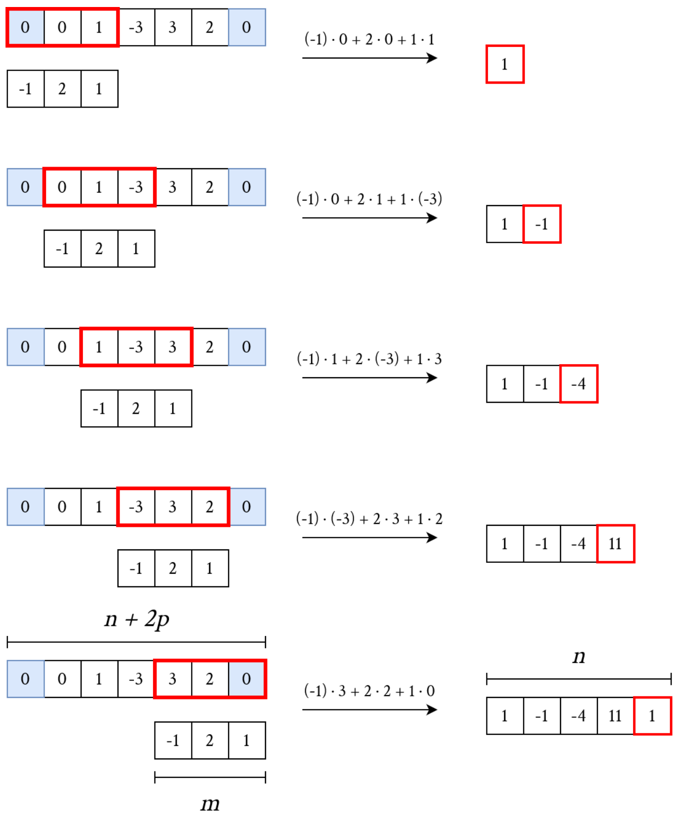
Mathematically, the i-th element of G resulting from the 1D convolution between F and I is defined as:

$$O[i] = (F * I)[i] = \sum_{k=0}^{f-1} F[k]I[i+k] \quad i = 0, 1, \ldots, l-1 \tag{2.1}$$

Intuitively, F can be pictured as a sliding window over I. Each position of the sliding window corresponds to an element of O, and for each position the underlying elements of the input are multiplied elementwise with the filter and summed (Figure 2.4).



(a) 'Valid' padding



(b) 'Same' padding

Figure 2.4: 1D convolution

Padding It can be observed that the output feature map O is smaller than I (Figure 2.4a). Roughly speaking, the information contained at the edges of I is lost. To preserve that information, or equivalently to have O be the same size as I, each end of the input feature map can be *padded* with p zeros, where $p = \frac{f-1}{2}$ (Figure 2.4b).

Let $\tilde{I} \in \mathbb{R}^{n+2p}$ be the padded input feature map. Then:

$$\tilde{I}[k] = \begin{cases} 0 & 0 \leq k < p \\ I[k-p] & p \leq k < n+p \\ 0 & k \geq n+p \end{cases}$$

I can be substituted with $\tilde{I}$ in Equation 2.1, obtaining:

$$O[i] = (F * \tilde{I})[i] = \sum_{k=0}^{f-1} F[k]\tilde{I}[i+k] \quad i = 0, 1, \ldots, n-1$$

When the input is not padded, the convolution is said to have *valid* padding; in the other case, the convolution is said to have *same* padding.

Stride If on the contrary the goal is reducing the size of the output feature map, whether it is to reduce the memory footprint or because the resolution is unnecessarily high, the concept of *stride* can be applied.

Returning to the graphical example, the stride s defines the quantity by which the sliding window moves over the input. If $s > 1$, it has the effect of reducing the length of the output by a factor equal to s. Considering input length n, filter length f, padding p and stride s, the length of the output feature map is now defined as:

$$l = \left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor$$

Equation 2.1 becomes:

$$O[i] = \sum_{k=0}^{f-1} F[k]\tilde{I}[s \cdot i + k] \quad i = 0, 1, \ldots, l-1 \tag{2.2}$$

2D convolution

Extending the concept of convolution to two dimensions is fairly straightforward. Let $I \in \mathbb{R}^{H_I \times W_I}$ be a 2D input feature map and $F \in \mathbb{R}^{f \times f}$ a 2D filter[4], where H_I

[4]For the purposes of this thesis, the filter will be always assumed to be a square filter

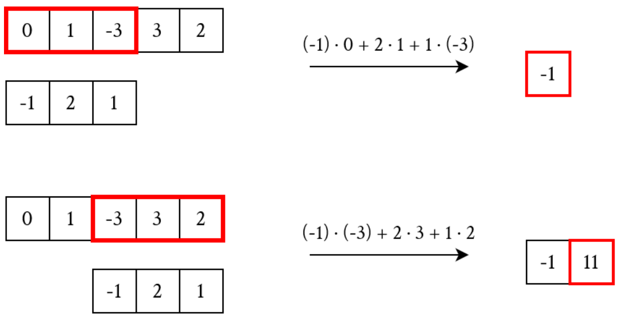


Figure 2.5: 1D convolution with 'valid' padding and stride 2

and W_I are respectively height and width of the input feature map and m is the size of the filter.

The output feature map $O \in \mathbb{R}^{H_O \times W_O}$ has height and width equal to:

$$H_O = H_I - f + 1 \quad W_O = W_I - f + 1$$

The element at index (i, j) of the output feature map is defined as:

$$O[i,j] = \sum_{k=0}^{f-1} \sum_{l=0}^{f-1} F[k,l] I[i+k, j+l] \qquad \begin{aligned} i &= \;\; 0, \dots, H_O - 1 \\ j &= \;\; 0, \dots, W_O - 1 \end{aligned} \tag{2.3}$$

For conciseness, from now on indices of multidimensional objects will be represented with subscript notation, e.g., $O_{i,j}$ instead of $O[i, j]$.

It is also immediate to introduce the concepts of padding and stride for 2D convolutions. Since there are two axes, both padding and stride may have two different values: one for the horizontal axis and one for the vertical one. They will be represented as (p_x, p_y) and (s_x, s_y), where p_x, s_x are relative to the horizontal axis and p_y, s_y to the vertical one. H_O and W_O now become:

$$H_O = \left\lfloor \frac{H_I + 2p_y - f}{s_y} + 1 \right\rfloor \qquad W_O = \left\lfloor \frac{W_I + 2p_x - f}{s_x} + 1 \right\rfloor \tag{2.4}$$

For convenience, from now on it will be assumed that padding and stride have the same value for both axes.

The padded input feature map $\tilde{I} \in \mathbb{R}^{(H_I+2p)\times(W_I+2p)}$ can be defined as:

$$\tilde{I}_{i,j} = \begin{cases} 0 & 0 \le i < p \quad \text{or} \quad 0 \le j < p \\ I_{i-p,j-p} & p \le i < H_I + p \quad \text{and} \quad p \le j < W_I + p \\ 0 & i \ge H_I + p \quad \text{or} \quad j \ge W_I + p \end{cases} \tag{2.5}$$

Equation 2.3 becomes:

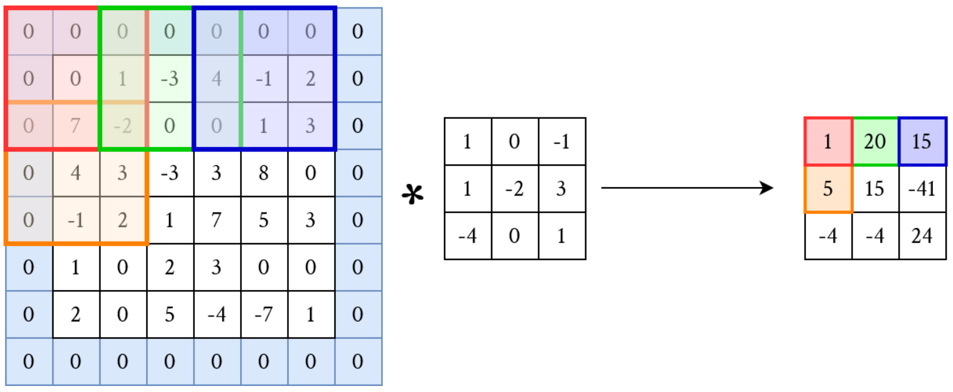
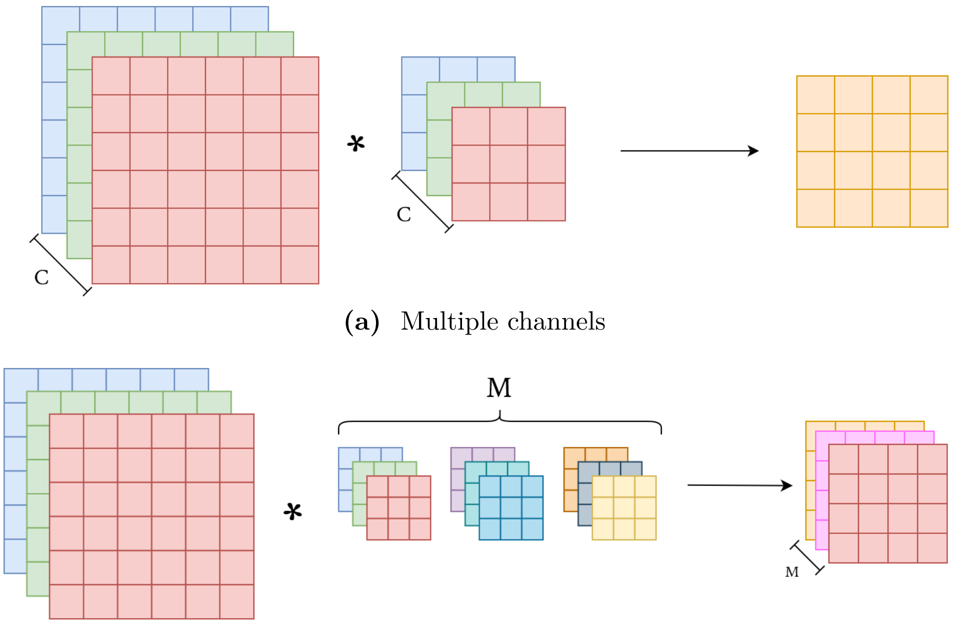


Figure 2.6: 2D convolution with $p = 1$ and $s = 2$



(a) Multiple channels

(b) Multiple filters

Figure 2.7: 2D convolution with multiple channels/filters

$$O_{i,j} = \sum_{k=0}^{f-1} \sum_{l=0}^{f-1} F_{k,l} \tilde{I}_{s \cdot i+k, s \cdot j+l} \qquad \begin{aligned} i = &\ 0, \dots, H_O - 1 \\ j = &\ 0, \dots, W_O - 1 \end{aligned}$$

Multiple channels The great majority of CNNs deals with RGB images in the input layer and feature maps with a high number of channels in the hidden layers, so it becomes necessary to extend the convolution operation to a third dimension to account for multiple channels and even a fourth if more than one image is involved.

This extension is fairly straightforward. Let $I \in \mathbb{R}^{C \times H_I \times W_I}$ be the input feature map and $F \in \mathbb{R}^{C \times f \times f}$ be the filter. Then the output feature map is $O \in \mathbb{R}^{1 \times H_O \times W_O}$. Note that O contains a single channel. In fact, each channel c of I is 2D-convolved with channel c of the filter, then the results are summed elementwise. The concepts of padding and stride apply also here for each 2D channel. Thus, the element at index (i, j) of the output feature map is defined as:

$$O_{i,j} = \sum_{c=0}^{C-1} \left(\sum_{k=0}^{f-1} \sum_{l=0}^{f-1} F_{c,k,l} \tilde{I}_{c,s \cdot i+k, s \cdot j+l} \right) \qquad \begin{aligned} i &= \ 0, \dots, H_O - 1 \\ j &= \ 0, \dots, W_O - 1 \end{aligned} \tag{2.6}$$

where H_O and W_O are calculated as in Equation 2.4.

Let us suppose that we want to convolve a batch of N images with the same filter F. In that case, another dimension can just be added to the input feature map, which becomes $I \in \mathbb{R}^{N \times C \times H_I \times W_I}$. Equation 2.6 is applied separately to each image, producing the output feature map $O \in \mathbb{R}^{N \times 1 \times H_O \times W_O}$, i.e., one output feature map for each image. It can be observed that the additional fourth dimension has no particular meaning for input and output: it is just a convenience to represent a batch of 3D tensors. This is not the case when a fourth dimension is added to the filter.

Multiple filters Convolutional layers are not restricted to use a single filter. Since a single filter corresponds to a single feature, e.g., vertical edges, it is desirable to use multiple filters to extract as many features as possible from the input. A fourth dimension can be added to the filter tensor to account for the multiple number of features that one might want to extract from the input.

Let $I \in \mathbb{R}^{N \times C \times H_I \times W_I}$ be a batch of N input feature maps and M be the number of features we want to extract from each feature map. Then the filter becomes $F \in \mathbb{R}^{M \times C \times f \times f}$ and the batch of output feature maps becomes $O \in \mathbb{R}^{N \times M \times H_O \times W_O}$. The element at position (i, j) of the m-th feature map extracted from input feature map n is calculated as:

$$O^{(n)}_{m,i,j} = \sum_{c=0}^{C-1} \left(\sum_{k=0}^{f-1} \sum_{l=0}^{f-1} F^{(m)}_{c,k,l} \tilde{I}^{(n)}_{c,s \cdot i+k, s \cdot j+l} \right) \tag{2.7}$$

It can be observed that adding a fourth dimension to the filter has a different meaning than adding a fourth dimension to the input. Here the additional dimension has the effect of producing multiple feature maps per input image, one for each feature/filter.

Bias Neural networks employ a *bias* to represent the value to which the output of a neuron tends in absence of inputs. In the case of feed-forward neural networks, the bias is a single scalar value, while in a CNN it is a vector whose size is equal to the number of output channels of the convolution.

Let $I \in \mathbb{R}^{N \times C \times H \times W}$ be an input feature map and $F \in \mathbb{R}^{M \times C \times f \times f}$ be a set of M filters. The bias vector is defined as a vector $B \in \mathbb{R}^{M}$. Given an element in channel m of the output, the bias vector element at position m is summed to the result of the convolution, i.e., Equation 2.7 becomes:

$$O^{(n)}_{m,i,j} = B_m + \sum_{c=0}^{C-1} \left(\sum_{k=0}^{f-1} \sum_{l=0}^{f-1} F^{(m)}_{c,k,l} \tilde{I}^{(n)}_{c,s \cdot i+k, s \cdot j+l} \right) \tag{2.8}$$

Activation Convolution as we have seen it in Equation 2.8 represents an affine transformation of the input pixels. This kind of transformation is appropriate to solve linear problems, but it stops being effective when facing nonlinear problems, such as learning the XOR function [11][10, pp. 167–172]. For example, the output of a multi-layer feed-forward neural network is still a linear function of its inputs.

This problem can be solved by introducing a nonlinear function $g(x)$, called *activation function*, that is applied to the output of the affine transformation and yields a nonlinear transformation. With an activation function, Equation 2.8 becomes:

$$O^{(n)}_{m,i,j} = g \left(B_m + \sum_{c=0}^{C-1} \sum_{k=0}^{f-1} \sum_{l=0}^{f-1} F^{(m)}_{c,k,l} \tilde{I}^{(n)}_{c,s \cdot i+k, s \cdot j+l} \right) \tag{2.9}$$

Some examples of activation functions are:

- *Rectified Linear Unit (ReLU)* (Figure 2.8a) – ReLU : $\mathbb{R} \to [0, +\infty)$

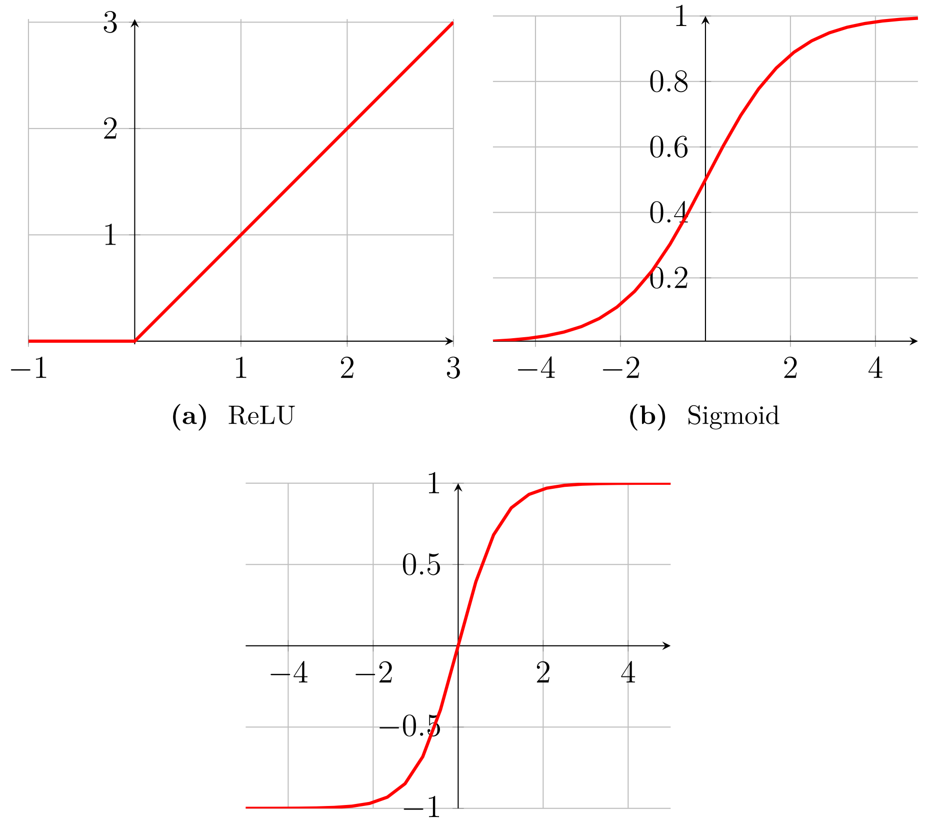
$$\mathrm{ReLU}(x) = \max(0, x)$$

- *Sigmoid function*, or *logistic function* (Figure 2.8b) – $\sigma : \mathbb{R} \to (0,1)$

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

- *Hyperbolic tangent* (Figure 2.8c) – tanh : $\mathbb{R} \to (-1, 1)$

$$\tanh x = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$



(a) ReLU **(b)** Sigmoid **(c)** Hyperbolic tangent

Figure 2.8: Activation functions

- *Softmax* – for an output layer consisting of K neurons with output vector $\mathbf{x}$, $\text{Softmax} : \mathbb{R}^K \to (0{,}1)$:

  $$\text{Softmax}_i(\mathbf{x}) = \frac{e^{x_i}}{\sum_{k=0}^{K-1} e^{x_k}} \qquad i = 0, \ldots, K-1$$

  Compared with the others, this activation function has the distinctive feature of considering the output of all neurons in the layer. Its output is a probability distribution: in fact, the sum of the outputs of Softmax for each i amounts to 1.

The choice of an activation function instead of another depends on the nature of the problem. For example, if the task to be performed by the neural network consists of calculating a probability, the sigmoid function is more appropriate than ReLU or the hyperbolic tangent, since its output ranges between 0 and 1.

2.2.2 Layers

Neural networks are usually represented as a sequence of *layers*. In the case of feed-forward neural networks, layers are composed of several neurons which take as inputs the outputs of the previous layer and produce a single output. The input layer and the output layer are particular: the first does not have inputs; the second does not feed its output to another layer. The number of neurons in a layer and the number of hidden layers are called *hyperparameters*, i.e., constant parameters chosen arbitrarily, whereas the weights of each neuron are called *parameters* and are not set by the programmer, but learned by the network during the training step.

CNNs are similar in this regard: they have an input layer, an output layer and several hidden layers with adjustable hyperparameters and learnable parameters that receive inputs from the previous layer and feed their outputs to the next. The main difference lies in the performed transformation: feed-forward neural networks are based on the application of a nonlinear function to an affine transformation of all the inputs at once, while in CNNs the transformation involves only one part of the inputs at a time. Besides, convolutional layers are not the only type of layer that can be employed in a CNN, but there are at least two other types of layers that can be used, namely *pooling* and *fully connected* layers. The first is in charge of compressing the input by pooling together parts of it, hence the name, while the latter is equivalent to a standard hidden layer of a feed-forward neural network.

Convolutional (CONV) Equation 2.9 fully represents what happens inside a basic convolutional layer in a CNN. It can be observed that a convolutional layer is described by the following hyperparameters:

- Height and width of the filter. Channels are not specified, since the number of filter channels must be equal to the number of input channels
- Number of filters (equal to the number of output channels)
- Padding
- Stride
- Activation function

Given a filter $F \in \mathbb{R}^{M\times C\times f\times f}$ and a bias vector $B \in \mathbb{R}^{M}$, the number n_T of trainable parameters (filters and bias) is $n_T = M \cdot (C \cdot f^2 + 1)$. Usually convolutional layers are specified with the size of their output, filter size, padding and stride.

Pooling (POOL) Pooling layers perform a similar task to strided convolutions, since they reduce the size of the input feature map. However, their purpose is to make the output approximately invariant to small translations of the input [10, 335–336] by splitting the input into small regions and outputting a summary for each of these regions. There are several choices for pooling functions, the most common being:

- *Max pooling* – the summary is the maximum value within the region
- *Average pooling* – the summary is the average of all values within the region

Similarly to convolution, pooling can be visualized as a sliding window scanning the input and producing an output for each position.

Pooling layers have no trainable parameters and two hyperparameters, namely pooling region size and stride.

Fully Connected (FC) Fully Connected layers work precisely as the hidden layers in a feed-forward neural network: they contain a set of neurons, each with its own set of weights, that compute an affine transformation of the input and apply a nonlinear function to the output of said transformation. When the input comes from a convolutional or pooling layer, it is first *flattened* into a 1D vector and then fed to the FC layer.

Usually this kind of layer is used at the end of a CNN. For example, when performing an image classification task it is convenient to output for each class the probability that the image belongs to that class: this can be achieved by placing a FC layer at the end of the CNN with one neuron per class and a Softmax activation function.

2.2.3 Convolution algorithms

Since their introduction by LeCun *et al.* [12] for a handwritten digit recognition task, CNNs have progressively shown to be extremely effective tools for computer vision tasks, both discriminative, such as image classification [13] and real-time object detection [14], and generative [15]. The availability of very large datasets, such as ImageNet [16] and Pascal VOC [17], and challenges such as the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [18] have constituted a significant incentive to develop complex CNN architectures, e.g., GoogLeNet [19], but Simonyan and Zisserman [20] have shown that besides network complexity, depth also plays a significant role in improving classification accuracy. In particular, they have shown that architectures based on conventional models such as LeNet-5 [21] and AlexNet [13] can achieve state-of-the-art performance simply by increasing the number of layers. The downside of this approach is that the increase in network depth results

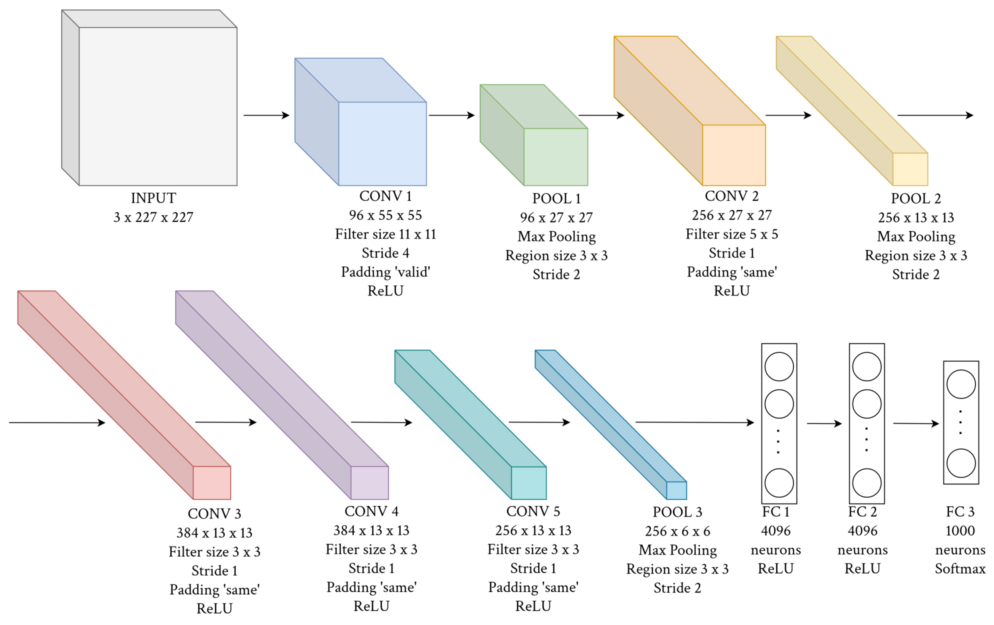


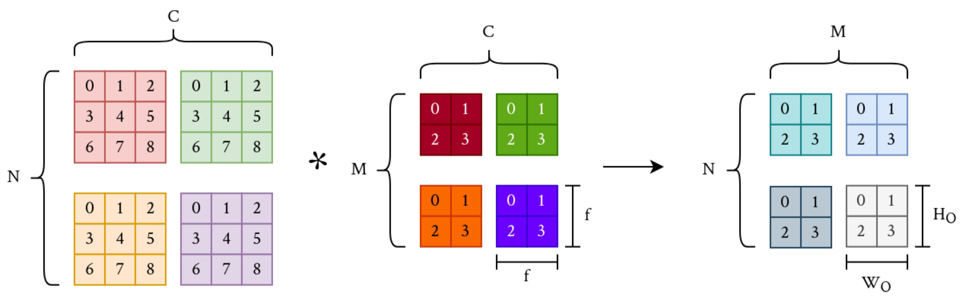
Figure 2.9: AlexNet Architecture. Output size is reported under the name of the layer.

in an increase in number of operations, during both training and inference. Since convolution accounts for most of the operations in a CNN, it is sensible to develop and use efficient convolution algorithms.

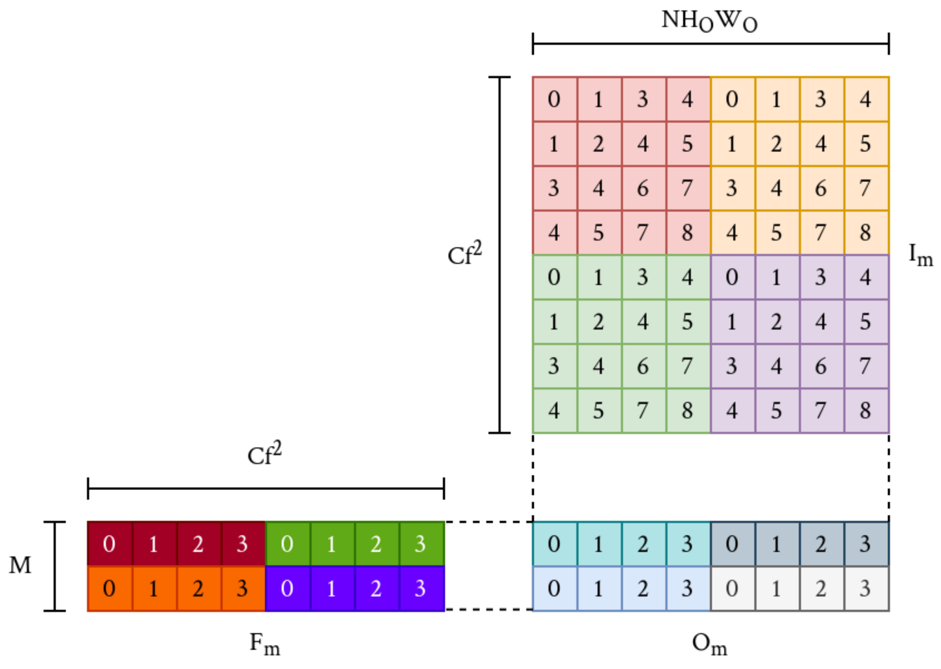
Fortunately, convolution presents a high level of data parallelism, since each element of the output can be computed independently from the others. Modern DNN frameworks such as TensorFlow [22] and PyTorch [23] leverage GPU acceleration to make both training and inference more tractable in terms of time. However, processing power alone offered by modern GPUs is not sufficient to achieve optimal performance. Computing the convolution directly as described by Equation 2.9 is not optimal, even with parallelization [24], so various convolution algorithms have been developed. Nvidia's cuDNN library for DNN primitives [25][26] offers various implementations of three popular convolution algorithms: General Matrix Multiplication (GEMM) [27][26], FFT (Fast Fourier Transform) [28] and Winograd's algorithm [29]. Since FFT and Winograd's algorithm are outside the scope of this thesis, only the GEMM algorithm will be described in detail.

GEMM

GEMM-based algorithms turn convolution into a matrix multiplication that can be very efficiently computed using Basic Linear Algebra Subprograms (BLAS) libraries [30] [31].



(a) Direct convolution



(b) GEMM convolution. Each column of I_m corresponds to a position of the filter.

Figure 2.10: Convolution expressed both as a direct convolution and as a matrix multiplication. Elements are uniquely identified by color and index

Let $A \in \mathbb{R}^{M \times K}, B \in \mathbb{R}^{K \times N}$ be matrix inputs, $C \in \mathbb{R}^{M \times N}$ a pre-existing output matrix and $\alpha, \beta \in \mathbb{R}$ scalar inputs. The GEMM operation is defined as:

$$C = \alpha AB + \beta C \tag{2.10}$$

From now on, it will be assumed that $\alpha = 1, \beta = 0$, reducing GEMM to a matrix multiplication.

Given a set of M filters $F \in \mathbb{R}^{M \times C \times f \times f}$ and a batch of N input feature maps

Algorithm 1 Im2col with no padding and stride 1

Inputs: Input tensor $I \in \mathbb{R}^{N \times C \times H \times W}$; filter size $f \times f$; output height and width H_O, W_O

Outputs: GEMM Input matrix $I_m \in \mathbb{R}^{Cf^2 \times NH_OW_O}$

```
 1: $I_m \leftarrow []$
 2: for $n \leftarrow 0$ to $N-1$ do
 3:     for $h \leftarrow 0$ to $H_O-1$ do
 4:         for $w \leftarrow 0$ to $W_O-1$ do
 5:             for $c \leftarrow 0$ to $C-1$ do
 6:                 for $i \leftarrow 0$ to $f-1$ do
 7:                     for $j \leftarrow 0$ to $f-1$ do
 8:                         row $\leftarrow c \cdot f^2 + i \cdot f + j$
 9:                         col $\leftarrow n \cdot H_OW_O + h \cdot W_O + w$
10:                         $I_m[\text{row}][\text{col}] \leftarrow I[n][c][h+i][w+j]$
11:                     end for
12:                 end for
13:             end for
14:         end for
15:     end for
16: end for
17: return $I_m$
```

$I \in \mathbb{R}^{N \times C \times H_I \times W_I}$, F is reshaped into a matrix $F_m \in \mathbb{R}^{M \times Cf^2}$ and I is expanded into a matrix $I_m \in \mathbb{R}^{Cf^2 \times NH_OW_O}$. Each row of F_m contains a whole unrolled filter and each column of I_m contains the input elements that concur in the computation of one output element. The output matrix $O_m \in \mathbb{R}^{M \times NH_OW_O}$ can be computed using Equation 2.10 with $A = F_m, B = I_m$. Each row of O_m, one per filter, contains the N feature maps produced by convolving the input with the corresponding filter. O_m is then reshaped into the output feature map $O \in \mathbb{R}^{N \times M \times H_O \times W_O}$. Figure 2.10 shows a graphical representation of this transformation.

While FFT and Winograd's algorithm focus on reducing the number of operations required to compute a convolution, it is interesting to observe that the computational complexity of this operation is $O(M \cdot C \cdot f^2 \cdot N \cdot H \cdot W)$, which is the same as direct convolution. The reason why GEMM is preferred over direct convolution is that matrix multiplication is a highly optimized operation for which extremely efficient GPU implementations exist. In particular, its efficiency derives from the fact that it "has a high ratio of floating-point operations per byte of data transferred" [26].

Im2col While reshaping filters is a fairly straightforward operation, computing I_m is less obvious. Input elements have to be rearranged and duplicated so that each dot product between a row of F_m and a column of I_m performed during the matrix multiplication corresponds to an output element of the convolution.

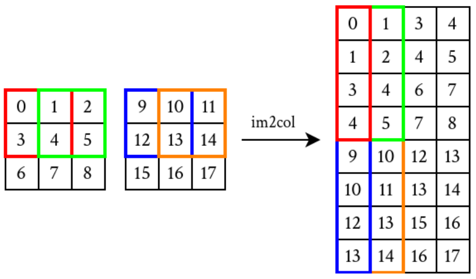


Figure 2.11: Im2col with input size $1 \times 2 \times 3 \times 3$, filter size 2×2, 'valid' padding and stride 1

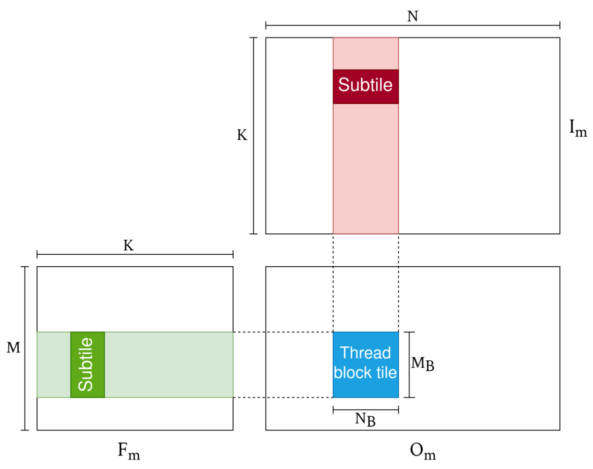
The algorithm that performs this transformation is called *im2col* (algorithm 1). Given an input batch $I \in \mathbb{R}^{N \times C \times H \times W}$ and a filter $F \in \mathbb{R}^{M \times C \times f \times f}$, its purpose is to take blocks of size $C \times f \times f$ from the input and rearrange them into columns. We can visualize it as a sliding window the size of the filter: for each position of the window, im2col takes the elements contained inside and unrolls them into a column. It can be observed that this operation closely resembles a convolution, the only difference being that no further operations, other than unrolling, are performed on the input elements.

One of the main advantages of using im2col and GEMM is that both padding and stride are easily accounted for. When using padding, it is sufficient to substitute I in algorithm 1 with the padded input $\tilde{I} \in \mathbb{R}^{N \times C \times (H+2p) \times (W+2p)}$. To account for an eventual stride s, line 10 of algorithm 1 becomes

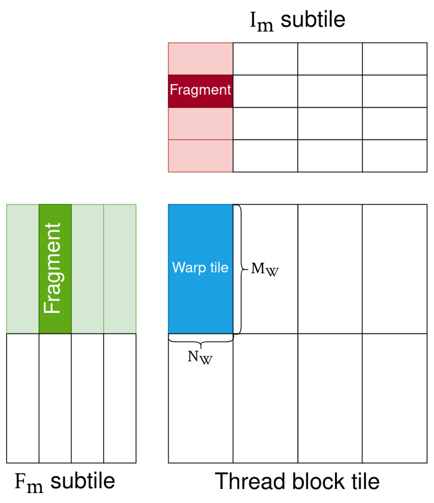
$$I_m[\text{row}][\text{col}] \leftarrow I[n][c][s \cdot h + i][s \cdot w + j]$$

The fact that im2col builds the input matrix explicitly results in additional memory consumption, both in terms of storage and bandwidth. For this reason, GPU accelerated implementations of *implicit* GEMM algorithms [26][32] have been developed that build tiles of the input matrix on the fly in shared memory, instead of using im2col to materialize the input matrix in global memory and then calling a GEMM routine. Other than the fact that the input matrices are not explicitly constructed, implicit GEMM algorithms perform the exact same computations as explicit algorithms.

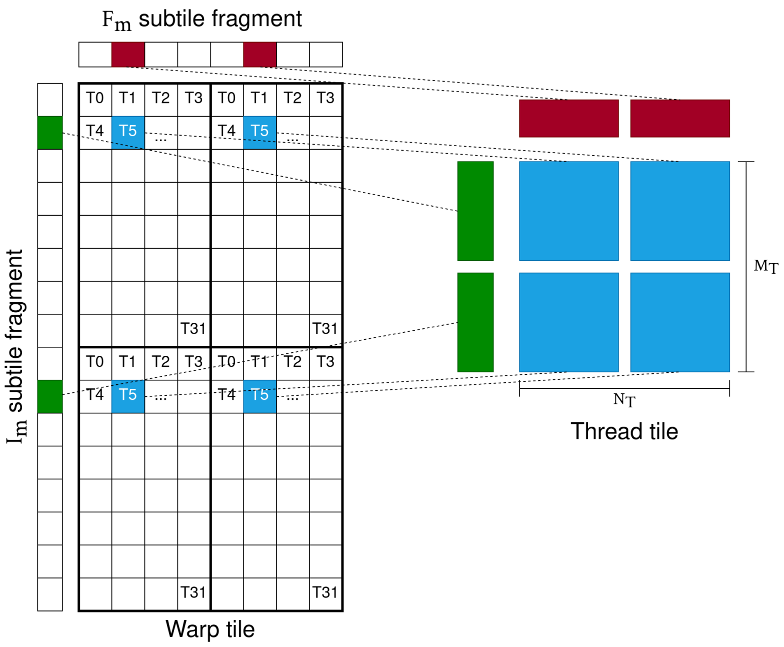
CUTLASS: implicit GEMM convolution for GPUs An example of a state-of-the-art GPU accelerated GEMM convolution is given by Nvidia's CUTLASS library [34]. CUTLASS offers CUDA C++ template abstractions to implement on GPUs highly optimized computations based on GEMM routines, as well as a convolution implementation based on the implicit GEMM algorithm. CUTLASS employs a tiling strategy to decompose the output computation into a hierarchy that mirrors the CUDA programming model. Let $F_m \in \mathbb{R}^{M \times K}, I_m \in \mathbb{R}^{K \times N}$ be the input matrices and $O_m = F_m I_m \in \mathbb{R}^{M \times N}$ be the output matrix. The decomposition is as follows:



(a) Thread block-level tiling



(b) Warp-level tiling



(c) Thread-level tiling

Figure 2.12: CUTLASS GEMM workload distribution [33]

1. O_m is partitioned into thread block tiles of size $M_B \times N_B$, each computed by a different thread block (Figure 2.12a). It follows that each thread block has to read a submatrix of F_m of size $M_B \times K$ and a submatrix of I_m of size $K \times N_B$ to compute its output. These submatrices are not loaded in shared memory in their entirety: each thread block iterates over the K dimension, loading subtiles of size $M_B \times K_B$ and $K_B \times N_B$ at each iteration, computing their matrix product and accumulating the result into the output.

2. Thread block tiles are further partitioned into warp tiles (Figure 2.12b). Each warp further iterates over the K_B dimension of the subtiles, loading fragments of the subtiles from shared memory into the register file, computing their product and accumulating the result into the output.

3. Warp tiles are further partitioned into thread tiles (Figure 2.12c). Each thread participates in the computation of a warp tile by computing the product of a certain number of elements of the fragments of F_m and I_m. The subdivision into thread tiles is engineered to minimize the number of shared memory loads from the same locations. In fact, multiple threads from the same row or the same column load the same elements of F_m and I_m, and since they cannot access each other's registers they have to read from the same locations multiple times.

2.3 Digital circuit testing

Digital circuit testing typically consists of applying a set of *test patterns*, or *stimuli*, to a Device Under Test (DUT) and then observing the response. If the latter does not match the expected response, then the DUT is assumed to be faulty. We can make a first distinction between two types of tests [35]:

- *Parametric tests.* Usually they involve measuring electrical quantities such as voltage and current
- *Functional tests.* They consist of applying a set of binary test patterns to the inputs of the circuit and then checking if the outputs match the expected response. Basically, functional tests verify that a digital circuit behaves as expected and performs its function correctly, i.e., it matches its specification

Functional testing is necessary for design verification, but is extremely difficult to perform thoroughly. For a circuit with n input lines, a complete functional test requires to check all the 2^n possible input patterns, which becomes unfeasible for circuits with many input lines. *Structural testing* represents a more feasible approach to hardware testing, owing the name to its dependency upon the structure of the circuit. In fact, it consists of selecting specific test patterns based upon the circuit description and a *fault model*, allowing to detect faults caused by manufacturing defects.

It is particularly important to specify the difference between defect, error and fault [35]:

- A *defect* is a mismatch between the hardware and its design specification
- An *error* is a wrong output signal caused by some defect
- A *fault* is a functional-level abstract representation of a defect. Depending on their time dependency, they can be classified as *permanent*, i.e., they do not change over time, or *transient*

2.3.1 Fault Models

Fault models represent a way to simplify test pattern generation by abstracting from real hardware defects. In fact, generating test patterns for many specific defects may be particularly difficult. Fault models allow to efficiently and automatically generate test patterns targeting a specific type of fault.

Modeling faults is closely linked to circuit modeling. At the Register-transfer level (RTL) the circuit model consists of a *netlist* of logic gates. At this level, commonly employed fault models related to Deep Learning reliability studies are [36]:

- *Stuck-at faults.* This model consists in assigning a fixed value (0 or 1) to an input or output line of the circuit, usually representing permanent physical damage. Depending on the value, these faults are classified as *stuck-at-0* (sa0) or *stuck-at-1* (sa1).
- *Bit-flip.* This model describes the change of logical state of a single bit from its original value to the other. It models transient randomly occurring changes in the state of a memory element due to an external disturbance.

For the purposes of this thesis, the focus will be placed on stuck-at faults.

Stuck-at faults According to [35, p. 71], a single stuck-at fault is defined by three properties:

1. Only one line is faulty
2. The line is set permanently to 0 or 1
3. The fault can be at the input or output of a gate

The stuck-at fault model, as outlined before, assumes that the circuit is modeled as a netlist, i.e., an interconnection of gates. A stuck-at fault affects one of these interconnections, causing a line to always have a state equal to 0 (or 1) irrespective of the effective logic output of the gate that drives it.

While a circuit with n lines can present multiple stuck-at faults simultaneously (up to $3^n - 1$ combinations), usually only single stuck-at faults are modeled, reducing the enormous number of combinations to a maximum of $2n$.

Fault equivalence Two faults are considered *equivalent* if they affect the circuit in a way such that the two correspondent faulty circuits have the same output.

Let $f(x)$ denote the output function of a circuit with n inputs and m outputs and let a and b be faults affecting the circuit. The outputs of the faulty circuits are denoted with $f_a(x)$ and $f_b(x)$, respectively. Let T be a n-bit test vector. Obviously, this test vector should produce different results when fed to the fault-free circuit and the two faulty circuits. To represent this mismatch, the XOR function can be used:

$$f(T) \oplus f_a(T) = V \quad \text{and} \quad f(T) \oplus f_b(T) = W$$

V and W represent m-bit output vectors.

When a and b have the same set of tests, then the two XOR operations are equal. It follows that:

$$f(T) \oplus f_a(T) \oplus f(T) \oplus f_b(T) = 0$$

Since the $f(T)$ terms cancel out, this leads to

$$f_a(T) \oplus f_b(T) = 0 \tag{2.11}$$

2.3.2 Testing stages

Digital circuit testing consists of several stages, each referring to a specific phase of the circuit's lifetime.

Post production Post production tests are performed on every single manufactured chip. Their purpose is to check that each manufactured device satisfies its specifications. Since performing comprehensive and complete tests is costly and time-consuming, post production tests focus on reaching a high coverage of modeled faults minimizing costs and time, since every device must be tested.

Burn-in Burn-in testing is performed on devices passing production tests to ensure their reliability. Post production tests verify that the devices do not present faults *at the time of testing*, but this does not guarantee that the devices will be fault-free for their whole intended lifetime. The purpose of burn-in tests is to make defective devices fail quickly by testing them over a long period of time, either continually or periodically.

Burn-in tests mainly detect *infant mortality failures.* Since the failure rate is higher at the beginning of the device's lifecycle, burn-in tests can weed out devices bound to fail early.

On-line On-line tests are performed while the device is being used in its intended application to detect faults occurring during device operation. For example, *Built-In Self-Test (BIST)* techniques consist of adding circuitry to allow the device to test itself, e.g., by integrating on the same circuit a test pattern generator and a response evaluator [37]. An alternative to BIST for on-line testing is *Software-Based Self-Test (SBST)* [37][38], which does not rely on additional hardware while retaining the possibility for the device to test itself. SBST techniques mainly consist of executing specific test programs able to detect possible permanent faults [37].

In both cases, test patterns may be pseudorandomly generated on-the-fly or generated beforehand, e.g., by an *Automatic Test Pattern Generation (ATPG)* algorithm.

2.3.3 Test Generation

As outlined at the beginning of this section, digital circuits are usually tested by feeding them test patterns. An exhaustive functional test of a device with n inputs would require testing all the 2^n possible inputs, a problem which rapidly becomes intractable as n grows. Hence, structural testing is used to restrict test patterns to the ones needed to cover a specific type of fault, described by a fault model.

A metric often used to describe the efficacy of a structural test is *Fault Coverage (FC)*, defined as:

$$\text{FC} = \frac{\text{detected faults}}{\text{total faults}} \tag{2.12}$$

Various test pattern generation techniques based on different approaches exist, such as ATPG, random and evolutionary-based test pattern generation.

Automatic Test Pattern Generation *Automatic Test Pattern Generation* (ATPG) is the process tasked with finding suitable patterns to test a circuit for specific faults. Given a netlist, a fault model and a corresponding fault list, ATPG algorithms inject a fault into the circuit described by the netlist and try to activate it and make sure that its effect is propagated to the circuit's output. The fault is detected when the output changes from the value expected for the fault-free circuit. In this case, the input that activated the fault and let it propagate to the output is the desired test pattern.

The test pattern search space in ATPG algorithms is described with a Binary Search Tree (BST), where each intermediate node represents a decision on the value of a primary input signal and each leaf represents one of the possible outputs. In a circuit with n primary inputs, the number of leaves is equal to 2^n, and all ATPG programs implicitly search this tree. In the worst case, the whole tree must be examined to check if a fault is *untestable*, i.e., it does not affect the output. It follows that to better reach the required FC, an ATPG algorithm has to be *complete*, i.e., it must be able to search the entire binary tree to find untestable faults and test patterns for hard-to-test faults [35, p. 159].

Since ATPG algorithms rely on a low level description of the circuit and a rigorous description of the search space, they are able to generate very accurate test patterns, but may have elevated time requirements.

Random *Random Test Pattern Generation* consists of generating pseudorandom test patterns. For example, some BIST and SBST implementations make use of Linear Feedback Shift Registers (LFSRs) to generate test patterns on-the-fly and feed them to the DUT. This approach may prove not suitable for random-resistant components and result in an elevated number of test patterns required to reach an acceptable FC [38].

Evolutionary-based *Evolutionary-based Test Pattern Generation* consists of generating a set of test patterns by iteratively accepting and rejecting test patterns according to their fault detection capabilities [35, p. 246], i.e., by making the test patterns *evolve* through some learning process. In particular, *genetic algorithms* may be used.

The purpose of a genetic algorithm is to maximize a *fitness function* that describes the required characteristics of the test patterns, e.g., fault detection capabilities. The set of test patterns is called *population* and is improved iteratively, with each iteration called a new *generation*. The population of a new generation is produced by executing some operations on the previous generation, namely:

- *Crossover*. Consists of combining bits from two patterns from the old generation to construct two patterns for the new generation

- *Mutation*. Consists of manipulating bits of a pattern from the old generation to construct a pattern for the new generation

- *Selection.* Consists of selecting highly fit patterns from the old generation to keep them in the new generation

2.3.4 Assessment techniques and fault injection methodologies

The impact of hardware faults on the operation of a digital circuit may be assessed by means of *Fault Injection (FI)* techniques, which allow to evaluate the behavior of a DUT when a fault occurs. More specifically, FI techniques consist of artificially injecting a fault into a circuit, feeding input stimuli to the circuit and comparing the circuit's response with a *golden* reference, looking for an eventual mismatch.

In the context of DNNs, FI techniques can be classified as [36]:

- *Simulation-based.* Simulation-based FI does not rely on the physical device, but on an abstract description of the system either at the software level or at the hardware level. In the first case, faults are injected into a high-level model of the system *without considering the underlying hardware*, hence its purpose is to assess the dependability of the application itself in presence of hardware faults, regardless of the specific underlying hardware. It follows that the injection accuracy may be low, but this disadvantage is offset by low costs and high speed.

  In the second case, the injection target may be modeled at the Register-transfer level (RTL) or at the gate level, thus reaching a higher injection accuracy, but tying the assessment to specific hardware (e.g., [39]). As a matter of fact, simulation-based hardware FIs require the HDL description of the target to be available. Furthermore, the low level of abstraction increases complexity and time requirements of the FI. These drawbacks are complemented by low costs and high accuracy.

- *Platform-based.* Platform-based FI consists of measuring and analyzing the behavior of a physical device that *emulates* the final target implementation, e.g., a FPGA or a GPU. For example, in the context of DNNs executed on GPUs or FPGA, faults are injected into the network's parameters, e.g., weights, activations and hidden states.

- *Radiation-based.* Radiation-based FI presents the highest level of accuracy, since it is performed on the final device implementation in the same environmental conditions in which the device will be deployed. For example, radiation-based FI may consist in irradiating the target device with a neutron beam, which can induce transient faults in the device (e.g., [40]). The main drawback of this technique is the high cost of the irradiation process and the fact that often the target device becomes unusable after irradiation. Furthermore, the experiment controllability is low, since it is not possible to control which fault to inject. Nonetheless, radiation-based FI can reach a very high level of accuracy with a low injection time.

Chapter 3

Proposed ITL generation method

In this chapter, a method to develop test images for the on-line self test of multipliers in GPUs is presented. First, an ATPG-based approach, particularly suitable for regular structures like GPU functional units, is employed to find a set of suitable input values at the functional unit level, with constraints given by the CNN's weights. These values are then transformed into a test image. Feeding these images to the unchanged CNN (in particular, to the first convolutional layer), results in a high fault coverage for the targeted unit. This methodology is described in details in Section 3.1. The proposed method, compared to its software counterpart (Software Test Libraries (STLs)), allows to perform a self-test routine on a GPU by taking advantage of the CNN itself, without altering it or interrupting its execution, hence avoiding costs connected to memory operations and context switches. Interleaving the inference of a few self-test images to the CNN's normal operation has low computational requirements. Therefore, the produced test images can be used in the field on the same CNN, by alternating "normal" inferences with the ITL self-test images (without moving or loading new weights in memory). In Section 3.2, a methodology to validate the effectiveness of the test images is presented.

3.1 ITL Generation

The generation of test images consists of three stages:

1. *Dataflow Algorithm Extraction.* The goal of this stage is to find the relation between output elements and multipliers, i.e., to find out what multipliers perform which multiplications. This stage yields a set of weight-input index pairs for each multiplier

2. *ATPG-based Pattern Generation.* Given a multiplier and the associated set of weights, this stage yields a set of weight-input test patterns

3. *Self-test images generation.* Given a set of multipliers, their set of weight-input index pairs and their set of test patterns, the ITL is generated

3.1.1 Dataflow Algorithm Extraction

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

The Dataflow Algorithm Extraction stage consists in mapping elements of O to the specific multipliers that compute their value. This mapping is referred to as *dataflow algorithm* and depends on three elements:

- *Convolution algorithm.* The convolution algorithm used to compute O determines *how* the convolution will be performed and thus what multiplications will be performed. For example, GEMM-based algorithms only change the structure of I and W, leaving their values intact and performing the same set of multiplications as a direct convolution. On the other hand, FFT-based algorithms perform a completely different set of multiplications, since they operate on the Discrete Fourier Transform of I and W.

  Knowing the convolution algorithm is fundamental in order to associate each output element with the multiplications performed to compute it. The multiplications are represented with pairs $\langle i_{\text{idx}}, w \rangle$, where $w \in W$ is a weight and i_{idx} is an index pointing to an element of the input feature map.

  Given an input tensor (eventually padded) $I \in \mathbb{R}^{N \times C \times H_I \times W_I}$, a filter tensor $W \in \mathbb{R}^{M \times C \times f \times f}$ and an output tensor $O \in \mathbb{R}^{N \times M \times H_O \times W_O}$, each element $O_{i,j,k,l} \in O$ is associated to the multiplications performed to compute it, for example:

$$\begin{aligned} O_{0,0,0,0} &\rightarrow [\langle i_{0,0,0,0}, W_{0,0,0,0} \rangle, \langle i_{0,0,0,1}, W_{0,0,0,1} \rangle, \dots] \\ O_{0,0,0,1} &\rightarrow [\langle i_{0,0,0,1}, W_{0,0,0,0} \rangle, \langle i_{0,0,0,2}, W_{0,0,0,1} \rangle, \dots] \\ &\vdots \end{aligned}$$

- *Workload-thread mapping.* Once the convolution algorithm is known, it is necessary to understand how the computation is split among participating threads. Given an output element $O_{i,j,k,l}$, this mapping associates it to a specific thread T_x.

  The workload-thread mapping depends exclusively on the specific implementation of the convolution algorithm. Since the algorithm specification, and thus the operations concurring in the computation of $O_{i,j,k,l}$, is known, extracting this mapping allows to associate T_x to a list of multiplications, represented as $\langle i_{\text{idx}}, w \rangle$ pairs:

$$\begin{aligned}
\mathrm{T}_0 &\leftarrow \left\{ [\langle i_{0,0,0,0}, W_{0,0,0,0}, \rangle, \ldots], [\langle i_{0,0,0,1}, W_{0,0,0,0}, \rangle, \ldots], \ldots \right\} \\
\mathrm{T}_1 &\leftarrow \left\{ [\langle i_{0,0,1,0}, W_{0,0,0,0}, \rangle, \ldots], [\langle i_{0,0,1,1}, W_{0,0,0,0}, \rangle, \ldots], \ldots \right\} \\
&\vdots
\end{aligned}$$

- *Thread-core mapping.* The final step to associate $O_{i,j,k,l}$ to a multiplier consists in mapping threads to hardware cores.

  The thread-core mapping depends exclusively on the device architecture, in particular on scheduling policies, so it can be extracted regardless of the specific application running on the GPU. Ultimately, its purpose is to associate a thread T to a unique identifier C of a GPU core:

$$\begin{aligned}
\mathrm{T}_0 \rightarrow C_0 &\implies C_0 \leftarrow [\langle i_{0,0,0,0}, W_{0,0,0,0}, \rangle, \ldots] \\
\mathrm{T}_1 \rightarrow C_1 &\implies C_1 \leftarrow [\langle i_{0,0,1,0}, W_{0,0,0,0}, \rangle, \ldots] \\
&\vdots
\end{aligned}$$

  Note that it is not necessary to have complete knowledge of the GPU scheduling policies. As a matter of fact, this mapping requires only to have information about thread dispatching, i.e., *where* a thread is being executed, not temporal information, i.e., *when* a thread is executed.

Finding the relation between multipliers and the set of multiplications they perform is fundamental to build test images. Once this stage terminates, each core/multiplier is associated with a list of pairs $\langle i_{\text{idx}}, w \rangle$, where w is a weight and i_{idx} is an index pointing to an element of the input feature map. Knowing what elements of I are multiplied by which weight and which core performs the multiplication allows to control the multiplications performed by each core. Note that some multiplications, such as multiplications by padding values, are in any case outside the control of the programmer.

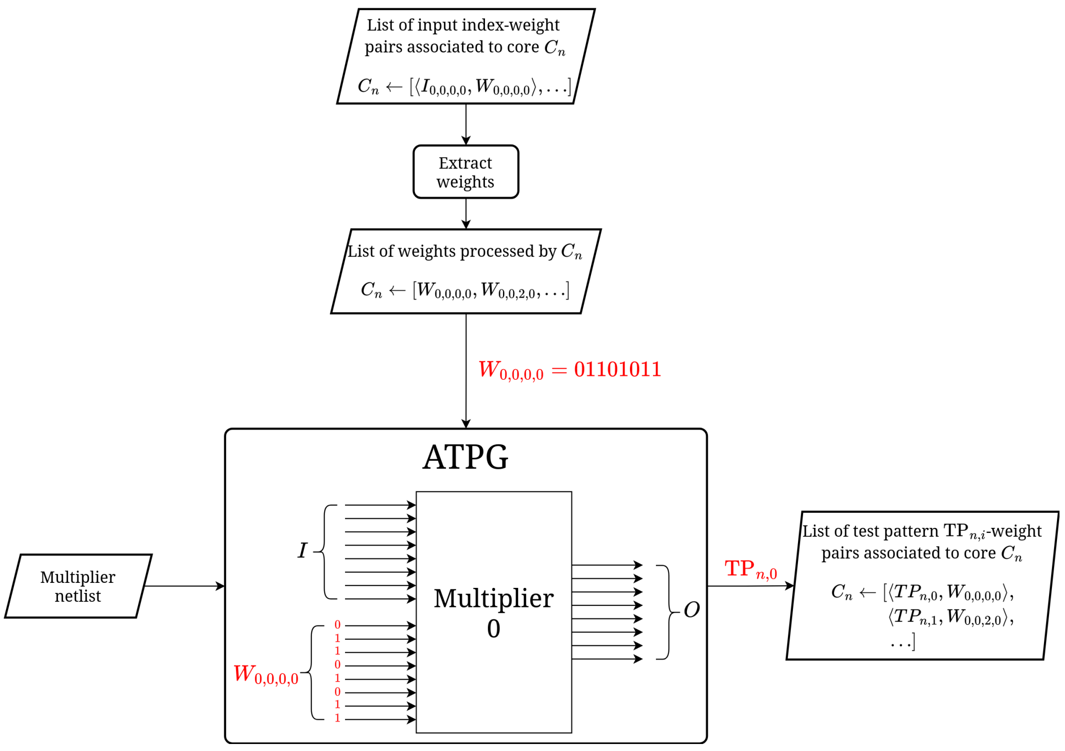

List of input index-weight pairs associated to core C_n
$C_n \leftarrow [\langle I_{0,0,0,0}, W_{0,0,0,0}\rangle, \ldots]$
Extract weights
List of weights processed by C_n
$C_n \leftarrow [W_{0,0,0,0}, W_{0,0,2,0}, \ldots]$
$W_{0,0,0,0} = 01101011$
ATPG
I
Multiplier 0
O
$W_{0,0,0,0}$
0 1 1 0 1 0 1 1
Multiplier netlist
$\mathrm{TP}_{n,0}$
List of test pattern $\mathrm{TP}_{n,i}$-weight pairs associated to core C_n
$C_n \leftarrow [\langle TP_{n,0}, W_{0,0,0,0}\rangle, \langle TP_{n,1}, W_{0,0,2,0}\rangle, \ldots]$

Figure 3.1: ATPG-based Pattern Generation. In red an example of generation of one pattern for a certain weight

3.1.2 ATPG-based Pattern Generation

Once the dataflow algorithm is known, each multiplier is associated to a list of multiplications. An ATPG process is set up to find out the collection P_c of input-weight pairs $\langle i, w\rangle$ that maximize the test coverage of the multipliers of a core c. The parameter $w \in W \in \mathbb{R}^{M\times C\times f\times f}$ corresponds to a real trained weight of a layer of the CNN. To this end, weights are put as constraints for the ATPG generation, together with a constraint forcing the 30th bit of i to be 0, to avoid the process generating infinities and NaN. So, the resulting test patterns depend on both the actual CNN's weights and input values generated by the ATPG.

It is worth underlining that the ATPG process is executed only on the targeted module, and the obtained test patterns only relate to its inputs.

3.1.3 Self-test Images Generation

For the on-line testing, the carefully-crafted $\langle i, w\rangle$ values produced in the previous stage are fed to the multiplier unit by means of suitable images composing the ITL. The dataflow algorithm extracted in the first stage allows to know exactly

Algorithm 2 Self-test Image Generation

```
Inputs: ATPG-patterns - Patterns for each core
Outputs: ITL - List of test images. Each image has the form $I \in \mathbb{R}^{C \times H_I \times W_I}$
1: ITL ← []
2: for core ← 0 to $n_{\text{cores}}$ 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:         $I, i_{\text{free}}$ ← FIND-EMPTY-POS(ITL, positions)
10:        if $I$ = nil then
11:            $I$ ← APPEND-NEW-IMAGE(ITL)
12:            $i_{\text{free}}$ ← positions[0]
13:        end if
14:        $I[i_{\text{free}}]$ ← pattern
15:    end for
16: end for
17: return ITL
```

the position i_{idx} of the element(s) of the input feature map that the multiplier will multiply by w. Such knowledge allows to place i in the correct spot of the input feature map, so that the multiplier will multiply it by w.

ITL Generation Algorithm The ITL is built using algorithm 2. The first step (line 3) is to reverse the mappings described in stage (i), associating each core with pairs of indices $\langle i_{idx}, w_{idx} \rangle$ of elements processed by that core. Here, i_{idx} is the index of an element of the input feature map I (i.e., $I[i_{idx}] = i$), while w_{idx} is the index of an element of the weight tensor W (i.e., $W[w_{idx}] = w$). When dealing with a convolution, a core reuses the same weight for different inputs. For this reason, given a weight index w_{idx} and a core c, a list of the associated input feature map indices $I_{idx}(w_{idx}|c)$ is built (line 4). Given the list of suitable input positions for each core and weight, it is possible to reconstruct the images. The general algorithm consists of two nested loops: the outermost one cycles over the available cores, while the innermost one operates on the ATPG-generated pairs $\langle i, w \rangle$ associated to a specific core c. For each $\langle i, w \rangle$ pair, a list of input feature map indices $I_{idx}(w_{idx}|c)$ associated to the weight index w_{idx} of the element w of the weight tensor W is selected (lines 7-8). The result of this process is a collection of suitable positions where to put the inputs i, associated with a weight w, returned by the ATPG processes. In line 9, an index is selected among *free* positions (i.e., not occupied by another pattern) across all the already-generated images. If a free space is not found, a new image is generated and a new position is chosen (lines 10-13). Finally, the input value i is assigned to the selected position (line 14).

3.2 ITL Validation

To validate their adoption for on-line testing, ITL must be able to excite hardware faults of the targeted functional module in a way that lets the faults propagate at the software level, so that their occurrence may be verified. In the context of CNNs, this translates in the possibility of observing faulty output feature maps after feeding them the ITL.

To study the propagation of hardware faults of a target unit to the software level, it is necessary to perform architectural-level fault simulations and, for each injected fault, check if the fault is propagated to the software level. In literature, many fault injection tools perform architectural-level injection through various methods. For example, SASSIFI [41] can inject errors in GPU registers and memory through source code instrumentation and NVBitFI [42], SASSIFI's successor, can perform dynamic and selective GPU code instrumentation during execution and without access to source code. Hybrid SASSIFI/TensorFlow solutions also exist (i.e., CLASSES [43]). All these solutions focus primarily on the architectural level, considering only registers, primary input (PI) and primary output (PO) of functional units. Presently, only [44] propagates at the software level the impact of permanent faults in functional units. The paper presents a method to combine software profiling with gate-level microarchitectural fault simulation to build syndrome tables, a collection of fault syndromes. Syndrome tables are used during the execution of the CNN to support code instrumentation and propagate the error effects. However, one single hardware fault might yield multiple error syndromes throughout the CNN's execution, potentially resulting in non-negligible syndrome table sizes.

3.2.1 Fault injection

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

$$\widehat{I} = \frac{\widehat{O}}{W}, \quad \text{or} \quad \widehat{W} = \frac{\widehat{O}}{I} \tag{3.1}$$

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

Fault equivalence This approach is further validated considering the concept of *fault equivalence*, described in section 2.3.1.

Let f be a fault, $\langle I, W \rangle$ be a test pattern for f, $M(x, y)$ be the multiplication of x and y performed by a fault-free multiplier and $\widehat{M}_f(x, y)$ be the same multiplication performed by the same multiplier in the presence of fault f. It follows that $M(I, W) = O$ and $\widehat{M}_f(I, W) = \widehat{O}_f$.

Now suppose that a fault g affecting the input lines of the multiplier exists such that $\widehat{M}_g(x, y) = \widehat{M}_f(x, y)$. Since f and g produce the same effect on the output, i.e., they satisfy Equation 2.11, they are equivalent and have the same set of tests.

For the purposes of fault injection, the equivalence implies that the output produced by injecting g and testing it with $\langle I, W \rangle$ is equal to that produced by injecting f. The difference lies in the fact that while injecting f may be complicated, injecting g is extremely simple. It can be observed that g causes the multiplier to receive a faulty input $\widehat{I}_g$, but does not affect the multiplier itself. As a matter of fact, excluding the input lines, the effective computation performed by the multiplier is correct. This fact makes it possible to assume the multiplier to be fault-free while moving g to the input, thus effectively computing $M(\widehat{I}_g, W) = \widehat{O}$. Since the input is assumed to be controllable, it is possible to inject f by injecting the equivalent fault g on the input and passing it to a golden multiplier.

3.2.2 Fault Injection with ITLs

A methodology is proposed to perform very accurate software fault injections by feeding the CNN faulty inputs $\widehat{I}$ equivalent to specific hardware faults internal to the targeted functional unit. This approach has two main advantages: it combines the accuracy of the gate-level microarchitectural simulation with the speed of software fault injections, and allows to experimentally demonstrate that the proposed ITL can excite permanent faults inside functional units while propagating the effects up to the output feature map of the first layer of the CNN, before nonlinearities are applied. The impact of hardware faults is not simulated by performing complex

Algorithm 3 Faulty Images generation for a fault in a multiplier.

Inputs:

- MULx - Selected multiplier;
- fault - A stuck-at fault of MULx;
- ITL - Image test library for a specific CNN;
- Operations - Pairs of ⟨input,weight⟩ multiplications performed by MULx during the convolution.
- n_{op} - Number of Operations

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

```
1:  FImg ← []
2:  $\widehat{I}$ ← []
3:  $W$ ← [];
4:  MULx-INJECT(fault)
5:  for op ← 0 to $n_{\text{op}}$ do
6:      $\widehat{O}$ ← MULx-MULTIPLY(Operations[op])
7:      W[op] ← GET-WEIGHT(Operations[op])
8:      $\widehat{I}$[op] ← $\frac{\widehat{O}}{W[\text{op}]}$
9:  end for
10: FImg[fault] ← PATCH-ITL($\widehat{I}$, W, ITL)
11: MULx-CLEAN(fault)
12: return FImg
```

and costly multi-level simulation environments, but only launching the inference of *faulty images that exactly reflect a precise hardware fault within the first layer.*

Algorithm The generation of faulty images that corresponds to injecting a specific fault within a multiplier is described in Algorithm 3. First, the fault is injected at low level in the multiplier (line 4). Then, for each operation performed by the multiplier during the convolution, its input weight W[op] and the low-level faulty output $\widehat{O}$ are collected (line 6-7). These values are used to compute the faulty input $\widehat{I}$[op] (line 8). Finally, the list of all the input elements (one for each operation) is converted to images following the same logic of Algorithm 2 (line 10).

To inject faults at application-level using the images generated with Algorithm 3, it is necessary to combine the information of the list of images in a single faulty output feature map. Therefore, given a fault f_c affecting core c, the first step to fault simulate a layer l, is to generate the set of faulty images I_{f_c} as described in Algorithm 3. I_{f_c} has the same size as the input feature map, i.e., $I_{f_c} \in \mathbb{R}^{N \times C \times H_I \times W_I}$. Note that the output feature map of the first layer contains values that are not only computed by core c. For this reason, the application of a mask $M(c) \in \{0,1\}^{N \times M \times H_O \times W_O}$ (that depends only on the core c) to the output of the layer is required. An element of this mask is set to 1 if the corresponding element in the output feature map is computed by core c, 0 otherwise. The mask can be constructed using the dataflow algorithm extracted in stage one of the ITL generation. As such, the faulty output l_{f_c} of the layer l, for a clean input tensor $I \in \mathbb{R}^{N \times C \times H_I \times W_I}$, can be computed as the sum of element-wise multiplications:

$$l_{f_c}(I) = l(I) \odot (1 - M(c)) + l(I_{f_c}) \odot M(c) \tag{3.2}$$

Software-level Observability With the developed ITL, the test coverage achieved by executing the test images is observed at the output of the single multipliers. However, during the on-line self test, the observability point has to be fixed at the software level. In particular, it has to be fixed at the output of the first convolutional layer, before any non-linearities are applied. As a consequence, for each self-test image, the respective golden output feature map, referred to as signature output feature map, of the first layer is stored and is compared to the actual one on-line: if they differ, a warning is raised.

Chapter 4

Experimental Results

This chapter presents experimental results that verify the proposed approach's effectiveness, verified using a Nvidia Jetson Nano board and examining its architectural details. The board houses a single-SM, 128-core Nvidia GPU based on 2$^{\text{nd}}$ generation Maxwell architecture [45]. Two ITLs have been generated for the ResNet-20 and DenseNet-121 CNN architectures, respectively. The proposed ITLs are designed to identify permanent faults that affect the targeted GPU's multipliers. Neither hardware-level[46] nor OS-level mechanisms[47] employed to ensure the safety of the GPUs have been taken into account. Section 4.1 reports details about the employed GPU, CNNs and multiplier. Section 4.2 describes the process of ITL generation, focusing on each of the three steps described in chapter 3.1. Furthermore, details about the ATPG process and ITL parameters such as test coverage, self-test time and storage requirements are reported, as well as a graphical representation of the generated ITLs. Section 4.3 reports results obtained following the ITL validation method described in Section 3.2.

4.1 Experimental setup

Nvidia Maxwell The complete implementation of the Nvidia GM204 GPU features 4 GPCs, each comprising 4 SMs for a total of 16 SMs. SMs consist of 128 CUDA cores that are partitioned between 4 processing blocks with 32 CUDA cores each, resulting in 32 FP32 multipliers per SM. Additionally, a warp scheduler is responsible for managing the scheduling of warps to its 32 cores (see Figure 4.1).

The Nvidia Jetson Nano board features only one SM, simplifying the thread-core mapping issue to determining how warps are assigned to warp schedulers. In fact, the task of assigning blocks to SMs is irrelevant when only one SM is present.

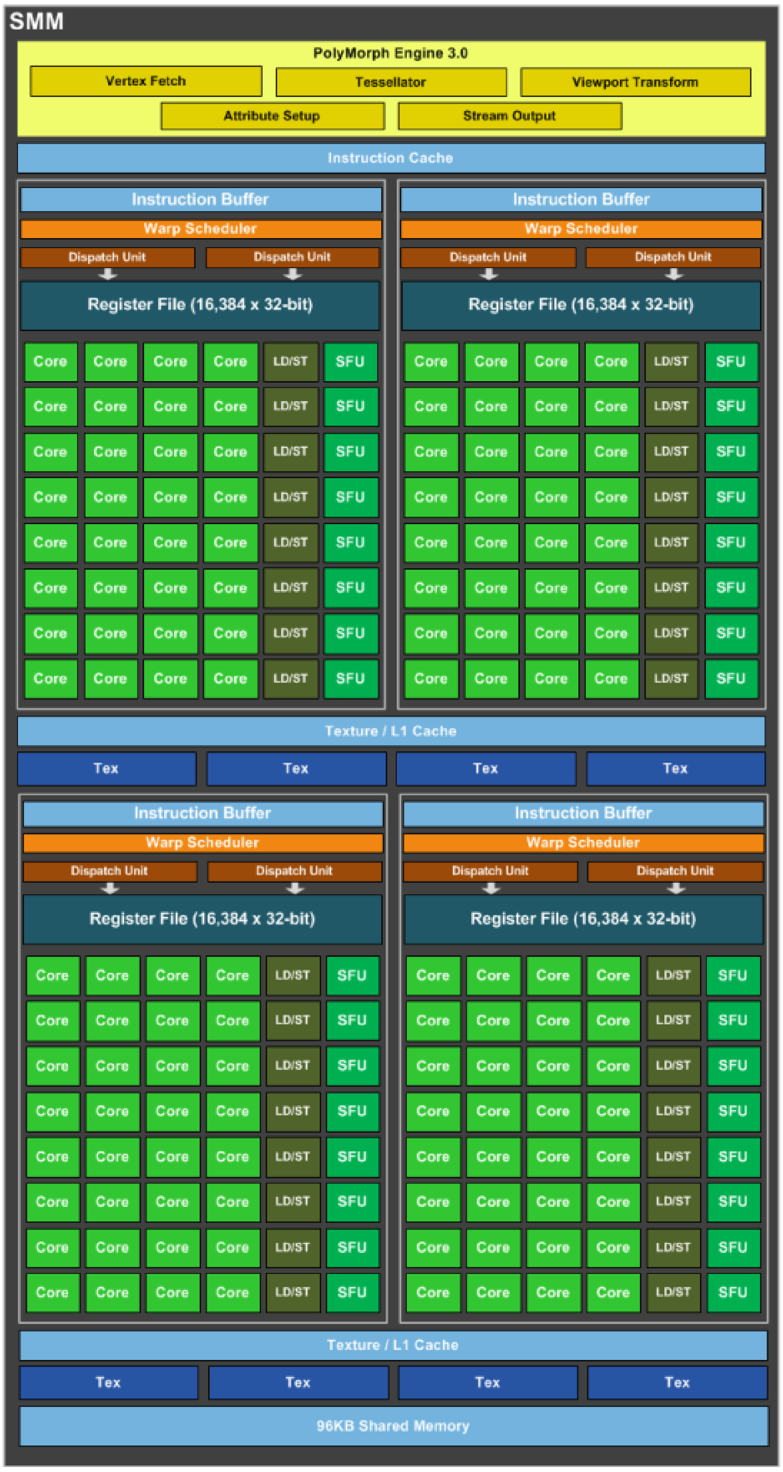


Figure 4.1: Nvidia Maxwell GM204 SM. Source: Nvidia Geforce GTX 980 Whitepaper [45]

Multiplier For the sake of reproducibility, an open source multiplier unit from OpenCores has been used [48]. This particular unit is compliant with IEEE-754 standards and is a single-precision, signed 32-bit floating-point (FP32) multiplier. The RTL design has been synthesized using the 45nm NangateOpenCell library [49] and a frequency of 50 MHz. The ATPG process was carried out using the Synopsys TetraMAX tool. The synthetized gate-level unit features a total of 12,510 stuck-at faults.

CNNs Two ITLs have been generated for two different CNNs (ResNet-20 and DenseNet-121) trained and tested on CIFAR-10 using PyTorch.

The first layer of ResNet-20 performs a convolution with 'same' padding and stride 1 between a $3 \times 32 \times 32$ input image and a filter tensor of 16 filters of size $16 \times 3 \times 3 \times 3$, resulting in 432 FP32 weights. The first layer of DenseNet-121, on the other hand, uses 'same' padding and stride 2, and convolves a $3 \times 32 \times 32$ input image with a filter of size $64 \times 3 \times 7 \times 7$, i.e., 64 filters, for a total of 9,408 FP32 weights.

4.2 ITL generation

The initial stage of the suggested approach involves the extraction of a list of weight-input pairs processed by each multiplier for every GPU core. This extraction is performed as explained in section 3. In this scenario, the convolutional algorithm has been fixed to GEMM, and the observation of thread-core mapping has been performed through profiling operations. Note that only a few seconds are required to generate the weight-input list, once convolution algorithm and mappings are known.

It is important to note that the only phase that required a fair amount of manual work is the Dataflow Algorithm Extraction phase, since the convolution algorithm and the workload-thread mapping had to be manually fixed and the thread-core mapping had to be profiled. The other two phases can be completely automated once the first has been concluded.

4.2.1 Dataflow algorithm extraction

Convolution algorithm The utilized convolution algorithm, outlined previously, is GEMM. This means that every output element is computed exactly as a direct convolution, so the arithmetic operations performed are exactly those described by Equation 2.7. To construct the list of input indices $[n, c, h, w]$ for an output index $[i, j, k, l]$, it sufficient to enumerate the indices from the summations in Equation 2.7.

Workload-thread mapping To have a knowledge of the *workload-thread mapping*, a C++ PyTorch extension has been added to force the usage of the GEMM implementation provided by CUTLASS (described in Section 2.2.3), since it is publicly available. Furthermore, this implementation has been customized to have warp tiles with the same height as the weight matrix (i.e., the number of filters/output channels M) and force each thread to compute a whole output column (i.e., M output elements, 1 for each output channel).

For ResNet-20, with a threadblock size of 128 threads (i.e., 4 warps), an input matrix of size 27×1024 and a weight matrix of size 16×27 result in:

- 8 thread block tiles of size 16×128
- 4 warp tiles per thread block, of size 16×32
- 32 thread tiles per warp, of size 16×1.

This means that a single core is in charge of computing $16 \cdot 8$ elements of the output feature map, performing 3,456 multiplications. The custom *workload-thread* mapping ensures that every core processes all the weights at least once. Due to the constraints being the same across different cores, it is possible to launch one single ATPG process valid for all the cores.

Since the GEMM tiling parameters can be controlled, it is straightforward to associate each thread to the position of the outputs it computes. Threads are identified in CUDA by a 3D block index $\mathbf{B}_{\text{idx}}$ and a 3D thread index $\mathbf{T}_{\text{idx}}$. It is assumed that block-level tiles of the output matrix are assigned to thread blocks in left-to-right, top-to-bottom order. The same is assumed for warp-level and thread-level tiles.

Note that changing this mapping could change the fault propagation and affect the test coverage (TC) of the targeted units.

Thread-core mapping The thread-core mapping of the Nvidia Jetson Nano GPU has been determined using a profiling program that launches a customizable number of thread blocks, each consisting of a customizable number of threads. To uniquely identify a CUDA core, the triple

$$\langle \text{SM}, \text{Warp scheduler}, \text{lane}^1 \rangle$$

is employed. Thus, each thread records the SM, warp scheduler and CUDA core on which it is executing, as well as its block and thread IDs.

[1]The term "lane" refers to a CUDA core within a processing block, i.e., a warp

While the SM and CUDA core can be identified by reading the GPU's special registers `%smid` and `%laneid`, identifying the warp scheduler is less trivial. Reading the special register `%warpid` provides a warp identifier within the thread block, but it does not offer any definitive information about the scheduler. Profiling results indicate that the least-significant two bits of the warp ID are indicative of the warp scheduler. Warps are distributed among schedulers in round-robin fashion. For instance, warp 0 will be sent to warp scheduler 0, warp 1 to warp scheduler 1, and so on. Therefore, a warp scheduler can be identified by computing

$$\text{WS} = \text{Warp ID} \bmod 4 \tag{4.1}$$

Concerning lanes, threads within a warp are assigned to lanes sequentially in ascending order of thread ID.

Considering the Jetson Nano, it is important to note that only one SM is available, resulting in `%smid` always being equal to 0.

The warp scheduler and lane to which a thread will be assigned can be predicted. With a thread block size of 128 threads/4 warps, it follows that:

$$\begin{cases} \text{Warp scheduler} & = \left\lfloor \dfrac{t_{\text{id}}}{32} \right\rfloor \\ \text{Lane} & = t_{\text{id}} \mod 32 \end{cases}$$

where t_{id} is the linearized thread index[2].

Padding Depending on convolution parameters, convolution algorithm and GPU architecture, certain cores may always multiply some weights by padding values. As there is no control over padding values, only weights multiplied by a non-padding input element at least once by each core should be selected. In other words, every pair containing a weight that is solely multiplied by padding in at least one core must be excluded from the list.

For ResNet-20, only the weights in the center column of each channel of each filter are multiplied at least once by a 'real' input element. As a result of this selection, the ATPG process was launched considering only 144 FP32 weights.

As for DenseNet-121, the last 32 filters have been excluded from the ATPG process and a 3×3 region around the central element has been selected for each filter channel, resulting in the reduction of candidate weights from 9,408 to 864. Then, as for ResNet-20, only weights in the central column have been selected, for a total of 288 weights. However, since the convolution is strided, utilizing only weights in center columns results in wasted space on the images (half the pixels in

[2]Since thread indices are 3-dimensional, they have to be flattened to a single index

CNN	**N. of weights**	**Selected weights**	**N. of ATPG patterns**	**Test coverage [%]**
ResNet-20	432	144	128	93.58%
DenseNet-121	9,408	576	135	94.28%

Table 4.1: Details about the ATPG process

the best case). Hence, also weights in the right columns have been selected, for a final total of 576 weights.

4.2.2 ATPG-based Pattern Generation

Details on the ATPG process are provided in Table 4.1. The second column displays the total number of weights of the first convolutional layer. For both CNNs, all the weights are employed at least once by every core during the first convolution. By eliminating weights that in certain cores are exclusively multiplied by 0-padding (2/3 of all weights for ResNet-20, 1/3 of reduced weights for DenseNet-121), the final list of candidate weights has been obtained (Column 3, Table 4.1). An ATPG process was set up by imposing the selected weights as constraints and searching a single ATPG pattern for each. However, some weights did not originate patterns able to increase the test coverage. The TetraMAX process required about 0.17s of CPU time to find a single pattern, while the total time required to generate the final set of input patterns was approximately 2 minutes for ResNet-20 and 10 minutes for DenseNet-121. The final number of ATPG patterns is given in Column 4, and the final test coverage is equal to 93.58% for ResNet-20, and 94.28% for DenseNet-121. Due to the constrained weights, 3.89% and 4.01% of stuck-at faults were classified as ATPG Untestable, respectively.

The achieved percentages are lower than those attained in [5] on NVDLA's computational units. The ATPG process used in that study did not impose any constraints and altered not only input values, but also the actual weights of the neural network. Consequently, it is not suitable for on-line testing, while the ITL method deliberately uses the actual weights of the CNN to alternate on-line inferences of "normal" images with self-test ones.

4.2.3 Self-test Images Generation

The ATPG patterns are then utilized to construct the self-test images, as described in Algorithm 2. The construction process requires knowing which pixel in the input image is processed by which core: this information is retrieved during the dataflow algorithm extraction. Ultimately, this procedure yielded 6 self-test images

Proposed ITLs	**N. of images**	**Avg. TC [%]**	**Self-test time [ms]**	**ITL storage requirements [kB]**
ResNet-20	6	94.74	0.35	467
DenseNet-121	8	95.46	0.36	623

Table 4.2: Details of the ITLs developed for testing on-line the FP32 multiplier.

for ResNet-20 and 8 for DenseNet-121, with a generation time of approximately 3 minutes for both ITLs.

The real ITLs are illustrated in Figures 4.3 and 4.2, respectively[3].

4.2.4 Results

Next, starting from these ITLs, a logic simulation has been performed resorting to Modelsim® HDL Simulation, by simulating the exact $\langle \text{input}, \text{weight} \rangle$ pairs of the obtained images entering into each of the core's multipliers.

For each test image, 128 (the total number of multipliers) Value Change Dump (VCD) files have been collected. These VCDs have been used to run gate-level fault simulations with TetraMAX, to compute the exact test coverage that each self-test image achieves on each core's multipliers. It has been observed that the actual test coverage is always higher than the one computed at the end of the ATPG process. Indeed, apart from the operations involving test patterns (3rd Column, Table 4.1), given a filter tensor $W \in \mathbb{R}^{M\times C\times f\times f}$, each core executes a number of additional multiplications equal to

$$\text{n. of images} \times \text{n. patterns/image per core} \times (M \cdot C \cdot f \cdot f - 1)$$

including several multiplications by zero (the 0-padding). It means that the proposed ITL generation guarantees a minimum test coverage among all the GPU cores. The exclusion of those weights that, in some cores, are multiplied exclusively by padding, avoids penalising individual cores.

Table 4.2 reports details of the two ITLs, in terms of (i) number of self-test images, (ii) average test coverage over all the 128 cores, (iii) time required to run the ITLs and compare the golden signature output feature map with the computed one, and the (iv) total storage required to support this self-test approach.

As for the self-test time, it is worth underlining that the time required to run the inference of 6 CIFAR10 images is 0.35 ms for ResNet-20, and 0.36 ms for

[3]Note that the ITLs are meant to be used in tensor, not image, form and may contain floating point values that do not correspond to RGB colors. To convert them to images the values have been normalized

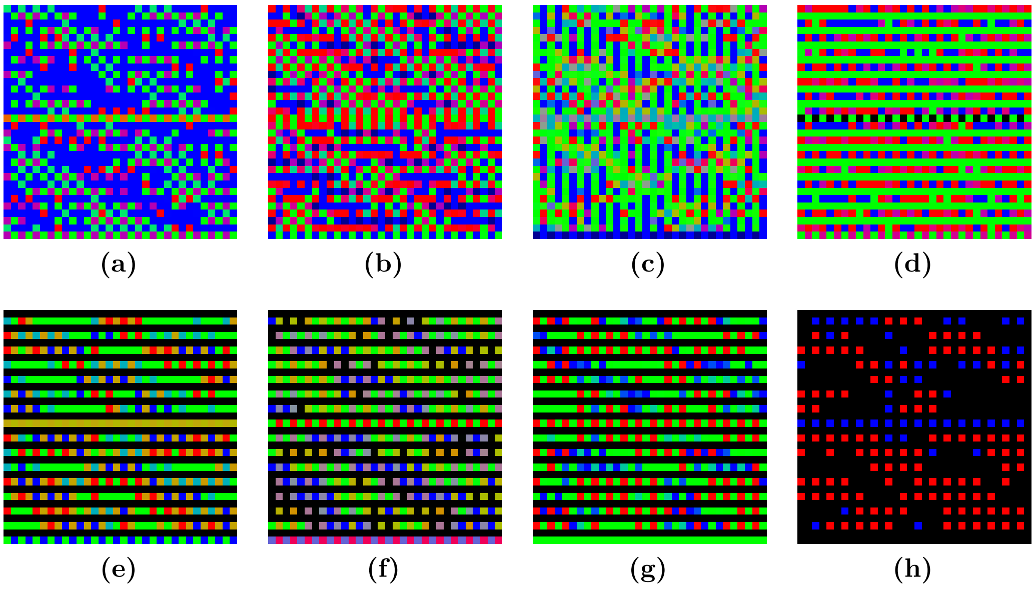


Figure 4.2: DenseNet-121 ITL

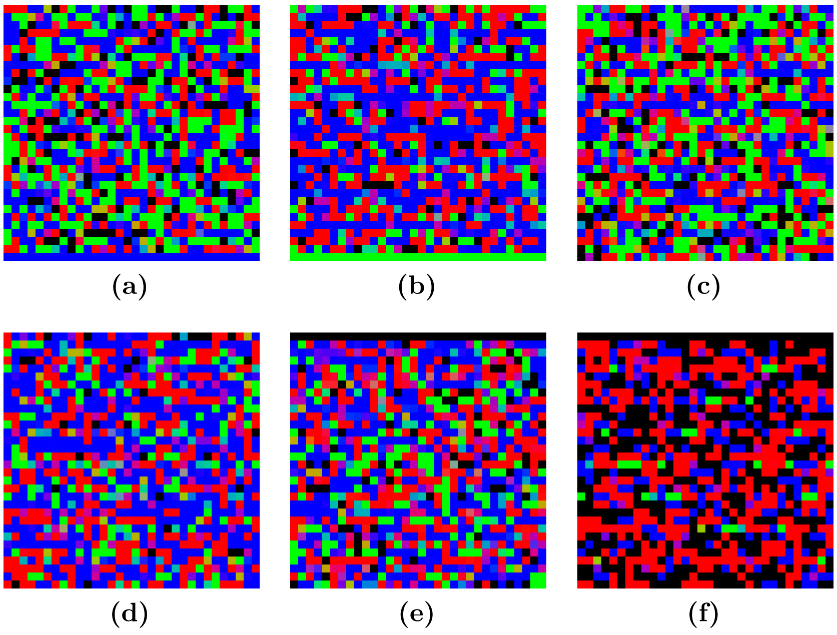


Figure 4.3: ResNet-20 ITL

ITL type	**Num. of images**	**Avg. TC on FP32 mul. [%]**			
		warp0	**warp1**	**warp2**	**warp3**
Proposed ITL	6	94.72	94.76	94.76	94.72
Checkerboard ITL	6	81.37	81.12	81.12	81.46
Random ITL	6	85.23	84.73	85.01	85.13
CIFAR10 ITL	6	84.42	84.85	84.56	84.61

(a) ResNet-20

ITL type	**Num. of images**	**Avg. TC on FP32 mul. [%]**			
		warp0	**warp1**	**warp2**	**warp3**
Proposed ITL	8	95.45	95.47	95.47	95.45
Checkerboard ITL	8	81.69	81.58	81.22	81.68
Random ITL	8	86.01	85.38	85.4	85.91
CIFAR10 ITL	8	84.88	85.39	85.53	85.07

(b) DenseNet-121

Table 4.3: Comparison of the proposed ITLs with Checkerboard, Random, and CIFAR10 images

DenseNet-121. Furthermore, considering the memory space needed to store the ITLs, the space to store the self-test images (e.g., for ResNet-20, 6 test images multiplied by $3 \cdot 32 \cdot 32 \cdot 4$ bytes) is summed to the space needed to store the golden test responses, i.e., the signature output feature map for each image (e.g., for ResNet-20 6 test images multiplied by $16 \cdot 32 \cdot 32 \cdot 4$ bytes).

For the sake of completeness, Tables 4.3a and 4.3b compare the test coverage of the proposed ITLs (in each warp) with the ones obtained by running the inference of checkerboard images, random images, and CIFAR10 images. An example of checkerboard image is given in Figure 4.4; checkerboard images seek to reproduce the well-known testing technique of applying specific checkerboard test patterns in assembly programs (e.g., `0xa5a5a5a5`).

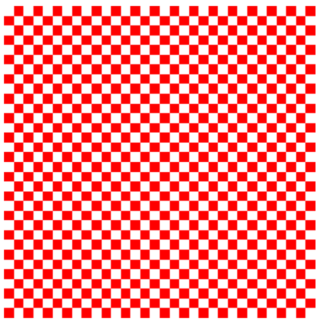


Figure 4.4: Checkerboard image

Then, the same quantity of test images was selected for each type of ITL (proposed, checkerboard, random, and CIFAR10), and gate-level fault simulations for each core in each warp have been performed. Each test image was fault simulated separately, and at the end, the 6 or 8 fault lists have been merged through TetraMAX. The final value is reported as the average over the 32 CUDA cores' multipliers in each warp. It can be observed that test coverage values oscillate depending on the amount of padding processed by each core, but it is interesting to note that the proposed ITL's values are always higher than ones obtained at the end of the ATPG process (Table 4.1).

As shown in Tables 4.3a and 4.3b, the main advantage of the proposed test images consists in the achieved test coverage: it is $\sim 13\%$, $\sim 9\%$, and $\sim 10\%$ higher than the checkerboard, random, and CIFAR10 ITLs, respectively. It means that with a very low number of inferences, it is possible to cover about 95% of stuck-at faults of the GPU's multipliers, without modifying the current CNN or undertaking costly memory operations.

4.3 ITL Validation

Instead of performing low-level fault injections and propagating faulty values at the software level, hardware faults affecting the multipliers are injected by building a set of carefully modified images which mimic the same faulty output of the multiplier, without injecting it at gate level.

Permanent faults affecting MUL_0 have been considered, and the faulty images have been created by following Algorithm 3 for ResNet-20. For a single image, MUL_0 executes a total of 3,456 multiplications by using all weights more than once. Of all the weights in the first layer (16 filters of size $3 \times 3 \times 3$), MUL_0 is responsible for performing 216 multiplications per filter. This means that there are 16 patches (3,456/216) in total containing faulty input values for the first layer only. These patches must be overlaid on each of the ITL images. Therefore, in total, each individual stuck-at fault corresponds to $6 \cdot 16 = 96$ faulty images. The time needed to generate the faulty images for all faults was approximately 20 minutes.

To compute $\widehat{O}$ and $\widehat{I}$, a Modelsim HDL simulation was performed by injecting the stuck-at faults that have been marked as detected at the end of the gate-level fault simulation. The simulation required approximately 8 hours. Then, given $\widehat{I}$ and the respective ITL, the faulty images have been created using Algorithm 3. Finally, inferences on the faulty images have been performed with a PyTorch simulation, without changing the ResNet-20 CNN model. To verify that the faulty ITLs can propagate the multiplier's faults up to the first convolution output feature map, Equation 3.2 was used to check for differences between tensors. They all produced a difference in the output feature maps: all the detected faults (after the gate level simulation) are propagated and observed through the output tensor of the first convolutional layer.

Chapter 5

Conclusions

This thesis outlines a technique for creating test images that can identify the presence of stuck-at faults in GPU multipliers in real time. Furthermore, it is demonstrated that with a minimal set of images, one can detect around 95% of permanent stuck-at faults with low self-test times and minimal memory occupation for the ITL storage.

Further research is needed to investigate the extension of this methodology to other GPU units. The primary concern may be that, in GPUs, information regarding thread-core and workload-thread mappings is not always accessible publicly. The dataflow algorithm can be extracted in particular instances, such as CUTLASS, and by executing profilation routines.

Additionally, the ITLs developed in this study are specific to multiplier units, with plans to expand to other computational and logic units in the future. As a matter of fact, convolutional algorithms in GPUs often exploit fused multiply-and-add units instead of multipliers.

One final consideration concerns the observability point when comparing output feature maps within the CNN (after the first layer in this case). To do so, the CNN needs to be considered as a white box since the comparison must be done on an intermediate result. The optimal solution might involve fixing the observability point at the output. However, in this case, a comprehensive analysis of fault propagation must be conducted to account for the inherent masking ability of CNNs, caused by the presence of non-linear activation functions, normalizations and pooling layers.

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 p. 1).

[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 GPUs». In: *2013 18th IEEE European Test Symposium (ETS)*. 2013, pp. 1–6. DOI: 10.1109/ETS.2013.6569353 (cit. on p. 1).

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

[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 p. 1).

[5] Yi He, Takumi Uezono, and Yanjing Li. «Efficient Functional In-Field Self-Test for Deep Learning Accelerators». In: *2021 IEEE International Test Conference (ITC)*. 2021, pp. 93–102. DOI: 10.1109/ITC50571.2021.00017 (cit. on pp. 1, 46).

[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 p. 2).

[7] Jens Krüger and Rüdiger Westermann. «Linear Algebra Operators for GPU Implementation of Numerical Algorithms». In: *ACM Trans. Graph.* 22.3 (July 2003), pp. 908–916. ISSN: 0730-0301. DOI: 10.1145/882262.882363 (cit. on p. 3).

[8] Wen mei W. Hwu, David B. Kirk, and Izzat El Hajj. *Programming Massively Parallel Processors: A Hands-on Approach, Fourth Edition.* Publisher Copyright: © 2023 Elsevier Inc. All rights reserved. Elsevier, Jan. 2022. ISBN: 9780323984638. DOI: `10.1016/C2020-0-02969-5` (cit. on p. 4).

[9] *CUDA C++ Programming Guide.* `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`. Accessed: 2023-09-27 (cit. on p. 4).

[10] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* `http://www.deeplearningbook.org`. MIT Press, 2016 (cit. on pp. 9, 16, 19).

[11] Fionn Murtagh. «Multilayer perceptrons for classification and regression». In: *Neurocomputing* 2.5 (1991), pp. 183–197. ISSN: 0925-2312. DOI: `https://doi.org/10.1016/0925-2312(91)90023-5` (cit. on pp. 9, 16).

[12] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. «Backpropagation Applied to Handwritten Zip Code Recognition». In: *Neural Computation* 1.4 (1989), pp. 541–551. DOI: `10.1162/neco.1989.1.4.541` (cit. on p. 19).

[13] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. «ImageNet Classification with Deep Convolutional Neural Networks». In: *Commun. ACM* 60.6 (May 2017), pp. 84–90. DOI: `10.1145/3065386` (cit. on p. 19).

[14] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. «You Only Look Once: Unified, Real-Time Object Detection». In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR).* 2016, pp. 779–788. DOI: `10.1109/CVPR.2016.91` (cit. on p. 19).

[15] Alec Radford, Luke Metz, and Soumith Chintala. «Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks». In: Nov. 2016 (cit. on p. 19).

[16] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. «ImageNet: A large-scale hierarchical image database». In: *2009 IEEE Conference on Computer Vision and Pattern Recognition.* 2009, pp. 248–255. DOI: `10.1109/CVPR.2009.5206848` (cit. on p. 19).

[17] Mark Everingham, Luc Gool, Christopher K. Williams, John Winn, and Andrew Zisserman. «The Pascal Visual Object Classes (VOC) Challenge». In: *Int. J. Comput. Vision* 88.2 (June 2010), pp. 303–338. ISSN: 0920-5691. DOI: `10.1007/s11263-009-0275-4` (cit. on p. 19).

[18] Olga Russakovsky et al. «ImageNet Large Scale Visual Recognition Challenge». In: *International Journal of Computer Vision (IJCV)* 115.3 (2015), pp. 211–252. DOI: `10.1007/s11263-015-0816-y` (cit. on p. 19).

[19] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. «Going deeper with convolutions». In: *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2015, pp. 1–9. DOI: `10.1109/CVPR.2015.7298594` (cit. on p. 19).

[20] Karen Simonyan and Andrew Zisserman. «Very Deep Convolutional Networks for Large-Scale Image Recognition». In: *International Conference on Learning Representations*. 2015 (cit. on p. 19).

[21] 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 p. 19).

[22] *TensorFlow*. `https://www.tensorflow.org/`. Accessed: 2023-09-25 (cit. on p. 20).

[23] *PyTorch*. `https://pytorch.org/`. Accessed: 2023-09-25 (cit. on p. 20).

[24] Heehoon Kim, Hyoungwook Nam, Wookeun Jung, and Jaejin Lee. «Performance analysis of CNN frameworks for GPUs». In: *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2017, pp. 55–64. DOI: `10.1109/ISPASS.2017.7975270` (cit. on p. 20).

[25] *Nvidia cuDNN*. `https://developer.nvidia.com/cudnn`. Accessed: 2023-09-24 (cit. on p. 20).

[26] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. *cuDNN: Efficient Primitives for Deep Learning*. 2014. arXiv: `1410.0759 [cs.NE]` (cit. on pp. 20, 22, 23).

[27] Kumar Chellapilla, Sidd Puri, and Patrice Simard. «High performance convolutional neural networks for document processing». In: *Tenth international workshop on frontiers in handwriting recognition*. Suvisoft. 2006 (cit. on p. 20).

[28] Michael Mathieu, Mikael Henaff, and Yann LeCun. *Fast Training of Convolutional Networks through FFTs*. 2014. arXiv: `1312.5851 [cs.CV]` (cit. on p. 20).

[29] Andrew Lavin and Scott Gray. «Fast Algorithms for Convolutional Neural Networks». In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 4013–4021. DOI: `10.1109/CVPR.2016.435` (cit. on p. 20).

[30] *Netlib BLAS*. `http://www.netlib.org/blas/`. Accessed: 2023-09-24 (cit. on p. 20).

[31] *Nvidia cuBLAS*. `https://developer.nvidia.com/cublas`. Accessed: 2023-09-24 (cit. on p. 20).

[32] *CUTLASS Convolution*. https://github.com/NVIDIA/cutlass/blob/main/media/docs/implicit_gemm_convolution.md. Accessed: 2023-09-24 (cit. on p. 23).

[33] *Efficient GEMM in CUDA*. https://github.com/NVIDIA/cutlass/blob/main/media/docs/efficient_gemm.md. Accessed: 2023-09-25 (cit. on p. 24).

[34] *CUTLASS*. https://github.com/NVIDIA/cutlass. Accessed: 2022-12-15 (cit. on p. 23).

[35] M. Bushnell and Vishwani Agrawal. *Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits*. Springer Publishing Company, Incorporated, 2013. ISBN: 1475781423 (cit. on pp. 25, 26, 29).

[36] Annachiara Ruospo, Ernesto Sanchez, Lucas Matana Luza, Luigi Dilillo, Marcello Traiola, and Alberto Bosio. «A Survey on Deep Learning Resilience Assessment Methodologies». In: *Computer* 56.2 (2023), pp. 57–66. DOI: 10.1109/MC.2022.3217841 (cit. on pp. 26, 30).

[37] P. Bernardi, M. Grosso, E. Sanchez, and O. Ballan. «Fault grading of software-based self-test procedures for dependable automotive applications». In: *2011 Design, Automation & Test in Europe*. 2011, pp. 1–2. DOI: 10.1109/DATE.2011.5763092 (cit. on p. 28).

[38] G. Xenoulis, D. Gizopoulos, N. Kranitis, and A. Paschalis. «Low-cost, on-line software-based self-testing of embedded processor cores». In: *9th IEEE On-Line Testing Symposium, 2003. IOLTS 2003*. 2003, pp. 149–154. DOI: 10.1109/OLT.2003.1214382 (cit. on pp. 28, 29).

[39] Annachiara Ruospo, Angelo Balaara, Alberto Bosio, and Ernesto Sanchez. «A Pipelined Multi-Level Fault Injector for Deep Neural Networks». In: *2020 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. 2020, pp. 1–6. DOI: 10.1109/DFT50435.2020.9250866 (cit. on p. 30).

[40] Lucas Matana Luza, Annachiara Ruospo, Daniel Söderström, Carlo Cazzaniga, Maria Kastriotou, Ernesto Sanchez, Alberto Bosio, and Luigi Dilillo. «Emulating the Effects of Radiation-Induced Soft-Errors for the Reliability Assessment of Neural Networks». In: *IEEE Transactions on Emerging Topics in Computing* 10.4 (2022), pp. 1867–1882. DOI: 10.1109/TETC.2021.3116999 (cit. on p. 30).

[41] Siva Kumar Sastry Hari, Timothy Tsai, Mark Stephenson, Stephen W. Keckler, and Joel Emer. «SASSIFI: An architecture-level fault injection tool for GPU application resilience evaluation». In: *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2017, pp. 249–258. DOI: 10.1109/ISPASS.2017.7975296 (cit. on p. 37).

[42] Timothy Tsai, Siva Kumar Sastry Hari, Michael Sullivan, Oreste Villa, and Stephen W. Keckler. «NVBitFI: Dynamic Fault Injection for GPUs». In: *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2021, pp. 284–291. DOI: `10.1109/DSN48987.2021.00041` (cit. on p. 37).

[43] Cristiana Bolchini, Luca Cassano, Antonio Miele, and Alessandro Toschi. «Fast and Accurate Error Simulation for CNNs Against Soft Errors». In: *IEEE Transactions on Computers* (2022), pp. 1–14. DOI: `10.1109/tc.2022.3184274` (cit. on p. 37).

[44] Josie E. Rodriguez Condia et al. «A Multi-level Approach to Evaluate the Impact of GPU Permanent Faults on CNN's Reliability». In: *2022 IEEE International Test Conference (ITC)*. 2022, pp. 278–287. DOI: `10.1109/ITC50671.2022.00036` (cit. on p. 37).

[45] Nvidia. *Nvidia Geforce GTX 980 Whitepaper*. `https://web.archive.org/web/20141014141630/https://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_980_Whitepaper_FINAL.PDF`. Accessed: 2023-09-27 (cit. on pp. 41, 42).

[46] *NVIDIA Xavier Achieves Industry First with Expert Safety Assessment*. `https://blogs.nvidia.com/blog/2020/05/20/xavier-achieves-industry-first-safety-assessment/`. Accessed: 2022-12-23] (cit. on p. 41).

[47] *Safe Travels: NVIDIA DRIVE OS Receives Premier Safety Certification*. `https://blogs.nvidia.com/blog/2022/12/16/nvidia-drive-os-tuv-sud-safety-certification/`. Accessed: 2022-12-23 (cit. on p. 41).

[48] *OpenCores, Floating Point Adder and Multiplier*. `https://opencores.org/projects/fpuvhdl`. Accessed: 2022-12-15 (cit. on p. 43).

[49] *Open-Cell Library*. `https://si2.org/open-cell-library/`. Accessed: 2022-12-22 (cit. on p. 43).