Politecnico di Torino

Fault Injection techniques for GPU Reliability Evaluation

Master's degree in Electronic Engineering

Supervisors:
Prof. Matteo Sonza Reorda
Dr. Juan David Guerrero Balaguera

Candidate:
Luigi Galasso
s267302

Academic Year 2021/2022

Acknowledgments

I would like to thank Professor Matteo Sonza Reorda for proposing me an interesting topic for my thesis work. I would like to thank Doctor Juan David Guerrero Balaguera for supporting me and concretely helping in the last months.
I would like to thank all my university colleagues I encountered in this journey, for helping me in many ways.
I would like to thank Rafael, for friendship and hard work in all the challenges we have faced together.
I would like to thank my parents and my brothers because with their support and inspiration I have been able to keep going even when I thought I could not make it, cheering me up every time I needed.
I would like to thank Cora for the support, patience and love she has always given me, even when it has been more difficult, during these years.
I would like to thank all people who believed in me.

Abstract

A Graphical Processing Unit (GPU) is a computer chip that renders graphics and images by performing rapid mathematical calculations. In recent years, GPUs are exploited for reasons beyond graphics processing as General Purpose GPUs (GPGPUs); they work as hardware accelerators for high-performance computing in many different fields, including safety-critical applications. In these domains, Convolutional Neural Networks (CNNs) represent a widely used computing approach, which is well supported by GPU, since they leverage data and thread-level parallelism. Considering this information, the reliability evaluation of GPUs is needed to meet desired requirements. To achieve this objective, it is necessary to study the GPU behavior in presence of hardware faults. In this thesis project, in particular, the presence of permanent faults affecting GPU functionalities has been analyzed. A permanent fault persists indefinitely after its occurrence: it manifests as stuck-at bits in the architecture that is, lines that always carry the logical signal "0" or "1". Those faults can be mimicked by injecting via software errors in the code running on the GPU; this could be obtained masking at assembly level one or more bit of a selected register before or after the corresponding instruction is executed. Therefore, in this work, it has been developed a framework, based on a binary instrumentation tool (NVBitFI), designed to properly perform permanent fault injection campaigns. Some injection techniques were elaborated to target distinct elements inside a GPU Streaming Multiprocessor: the Register Files and the Functional Units (Floating Point, Integer and Special Function Units). The presented environment has been used to test an NVIDIA GPU with a specific CNN target application, i.e., the LeNet model available in Darknet environment. To support the framework, many fault simulations were performed, and the obtained results were analyzed and compared.

Contents

List of Figures

List of Tables

Acronyms

ANN Artificial Neural Network.

API Application Programming Interface.

CNN Convolutional Neural Network.

CPU Central Processing Unit.

CUDA Compute Unified Device Architecture.

DUE Detected Unrecoverable Error.

FP Floating Point.

FU Functional Unit.

GPGPU General-Purpose GPU.

GPU Graphical Processing Unit.

INT Integer.

RF Register File.

SASS Source and Assembly.

SDC Silent Data Corruption.

SFU Special Function Unit.

SIMD Single-Instruction Multiple-Data.

SIMT Single-Instruction Multiple-Threads.

SM Streaming Multiprocessor.

CHAPTER 1

Introduction

1.1 Motivations and Thesis Contribution

Currently, Deep Learning and especially Convolutional Neural Networks (CNNs) have become a fundamental computational approach applied in a wide range of domains, including some safety-critical applications (e.g., automotive, robotics, and healthcare equipment). In these domains GPUs are used as hardware accelerators. Therefore, the reliability evaluation of those computational systems is mandatory.

Reliability is a property of a system, and it is defined as the probability that it will perform its intended function adequately for a specified period of time, or will operate in a defined environment without failure. A failure is generated, as effect of error propagation up to the output of the system. Meaning that the results generated are in someway wrong. The internal errors are activated by faults, depending on various conditions.

A Fault is, instead, a defect in the electronic system. The faults can be divided in two categories: permanent and transient. Permanent faults are stable defects in the system's components, such as wires tied to fixed logic values (stuck-at 0 or 1). Their permanence could affect all the system computations. Transient faults are temporary, meaning that after a certain period of time, during which they could affect the system behavior, they could disappear.

The main goal of this project is to evaluate the reliability of Graphical Processing Units (GPUs), executing high computational demanding applications, as Convolutional Neural Networks (CNNs). In many previous works, CNNs were tested on GPUs mostly for the presence of transient faults through the use of instrumentation tools, as SASSIFI or NVBitFI, able to soft inject fault in specific location imitating the presence of hardware random misbehavior due to radiations. Permanent faults testing is, instead, more complex; it requires that the effect generated by the error injection is kept for a longer time

with respect to transient faults. For this research work, it has been chosen to focus on permanent fault injections in NVIDIA GPUs. The presence of a hard fault is simulated by a tool, able to dynamically perform the hard errors injection at assembly instruction level. Starting from the NVBitFI basic functionality to perform single instruction permanent fault injections, a complete environment was developed in order to support accurate injections campaigns. The framework functionalities are based on the usage of a profile function to collect meaningful information about the application behavior on the target GPU, and an inject function able to perform fault injections targeting register files and functional units. The tool is automated by means of custom scripts, able to generate a proper fault list, gather information about injection results and perform a fault classification. A specific target application was chosen to perform the tests. The LeNet [1] Convolutional Neural Network (CNN) model available in the Darknet environment [2] is tested, using the MNIST dataset for training and inference [3].

A brief description of the content of each chapter of this work follows.

In Chapter 1 - Introduction, the main objectives and motivations of the thesis work are described.

In Chapter 2 - Backgrounds, the needed information for understanding the concepts in this research is explained in details, with particular emphasis on NVIDIA GPU architectures elements description and functioning, general-purpose computing on GPU based on CNNs, a bibliographic overview about CNN reliability evaluation and NVBitFI tool description.

In Chapter 3 - Permanent Fault Injection Framework, the developed tool is described justifying the taken choices for the implementation and all its functionalities.

In Chapter 4 - Experimental Results, some faults simulations results in several cases are shown and analyzed.

In Chapter 5 - Conclusions, the conclusions of the thesis are presented, showing what could be highlighted from the overall results.

CHAPTER 2

Backgrounds

In this chapter, some information required to understand the concepts, illustrated in next sections, are explained in details. The chapter proposes a description of NVIDIA GPU functioning and Convolutional Neural Networks, a bibliographic overview about CNN reliability evaluation on hardware accelerators and NVBitFI tool description.

2.1 GPU: Architectures and Functioning

In recent years, in many different applications, as safety-critical ones, the computational power demand is significantly increased and the traditional single-core architectures are not able to meet the required performance. As a result, the trend of last years is to exploit parallel processing in those fields. A program able to run in a parallel way, using several hardware units, is able to be much more faster than a single sequential based one; parallelism means trying to execute the same instruction, hence the same operation, independently on separate computational units. This solution provides systems with a significant increase in processing speed, however there are many issues to be faced with this strategy. In particular, these are: the code partitioning into sections to be distributed or issued in parallel, scheduling the sections and allowing intra-sections communication. From this idea, it is necessary to establish how to generate a parallel enabled code. The smallest sequence of programmed instructions, that can be managed independently, and so in parallel, by a scheduler is a thread of execution. A thread is the fundamental building block of a parallel program; it is like a process, in that it has state and current program counter; but typically, threads of the same process, share the address space, allowing a thread to easily access data of other threads within the same process. Multithreading is a technique whereby multiple threads share a processor without requiring an intervening process switch. The ability to rapidly switch between threads is what enables multithreading to be used to hide pipeline and memory latency issues, usually affecting CPUs

[4].

While single, double or quad core CPUs are optimized for applications with a limited number of threads, executing several type of different possible operations, and characterized by an high percentage of conditional branches; Graphical Processing Units (GPUs) are instead suited for applications with multiple threads that are dominated by longer sequences of computational instructions. As illustrated in Figure 2.1, a GPU consists of many functional units with only a few support elements, allowing an high level of parallel processing [5]. These characteristics of modern GPUs are used for intensive computations applications.

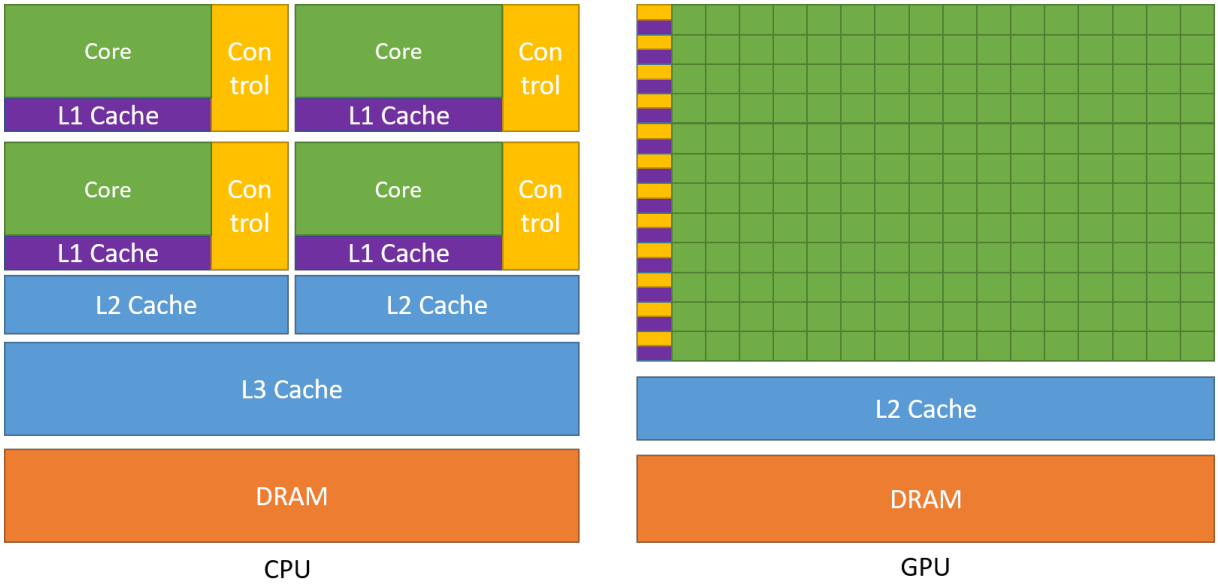


Figure 2.1: CPU and GPU general architectures, taken from [6]

The parallel programming model adopted for NVIDIA GPUs consists in writing special functions, executing parallel operations, named Kernels. A kernel is just a name for a function that executes on the GPU.

Each Kernel is issued by the Host (CPU) on a Device (GPU). It is launched as a grid of elements, that is simply a 2D set of blocks. Each block is composed of a defined number of threads [7]. In this way threads are packed together and GPU is responsible to assign them to internal cores (Figure 2.2).

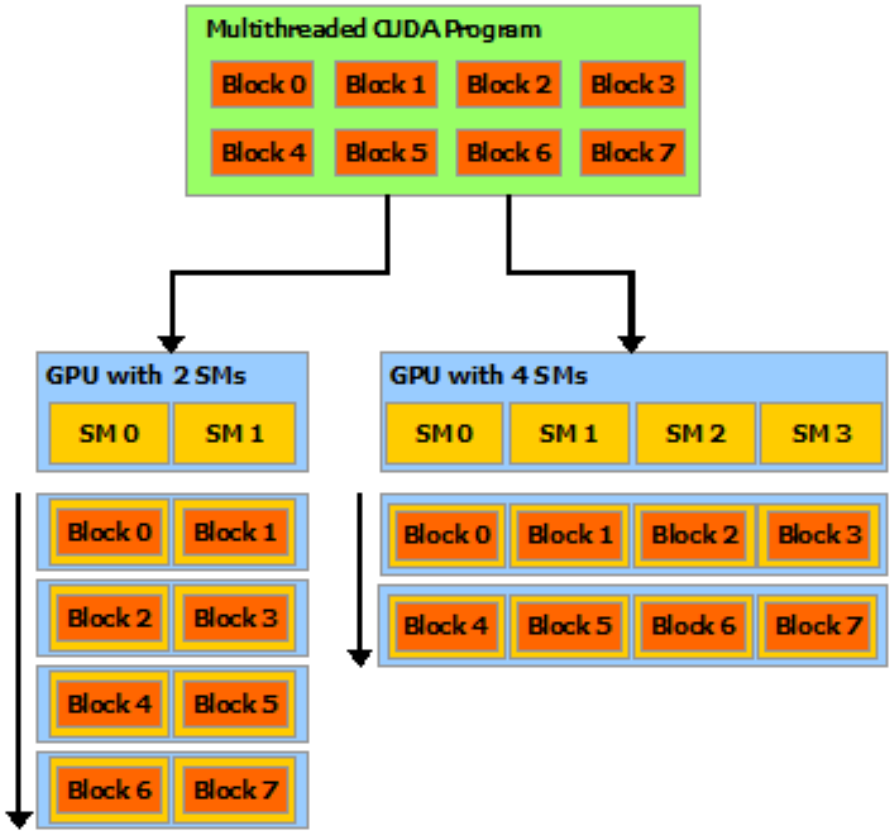


Figure 2.2: Blocks distribution, taken from [6]

NVIDIA GPUs are, in fact, used as GPGPU adopting the idea of Single-Instruction Multiple-Threads (SIMT), introduced by NVIDIA itself, consisting in applying one instruction to multiple independent threads in parallel, not just multiple data lanes as enabled by Single-Instruction Multiple-Data (SIMD) approach. SIMT is basically an execution model used in parallel computing where single instruction, multiple data (SIMD) is combined with multithreading [4]. Each SIMT instruction controls the execution and branching behavior of one thread while in SIMD design, it is applied one instruction to a vector of multiple data lanes together [8].

To program a GPU it is necessary to write in OpenCL or CUDA language the kernels to be executed. The compiler first transforms the code into PTX form. Successively, a code generator and translator maps PTX to specific target machines' assembly language. SASS is the low-level assembly language that compiles to binary microcode, which executes natively on NVIDIA GPU hardware. The Parallel Thread Execution (PTX) programming model is explicitly parallel: a PTX program specifies the execution of a given thread of a parallel thread array (CTA, or Cooperative Thread Array). PTX provides a stable ISA (Instruction Set Architecture) that spans multiple GPU generations, meaning that it is machine-independent and scalable [9]. This intermediate step is crucial to guarantee the portability of the code among different GPU's micro-architectures. The

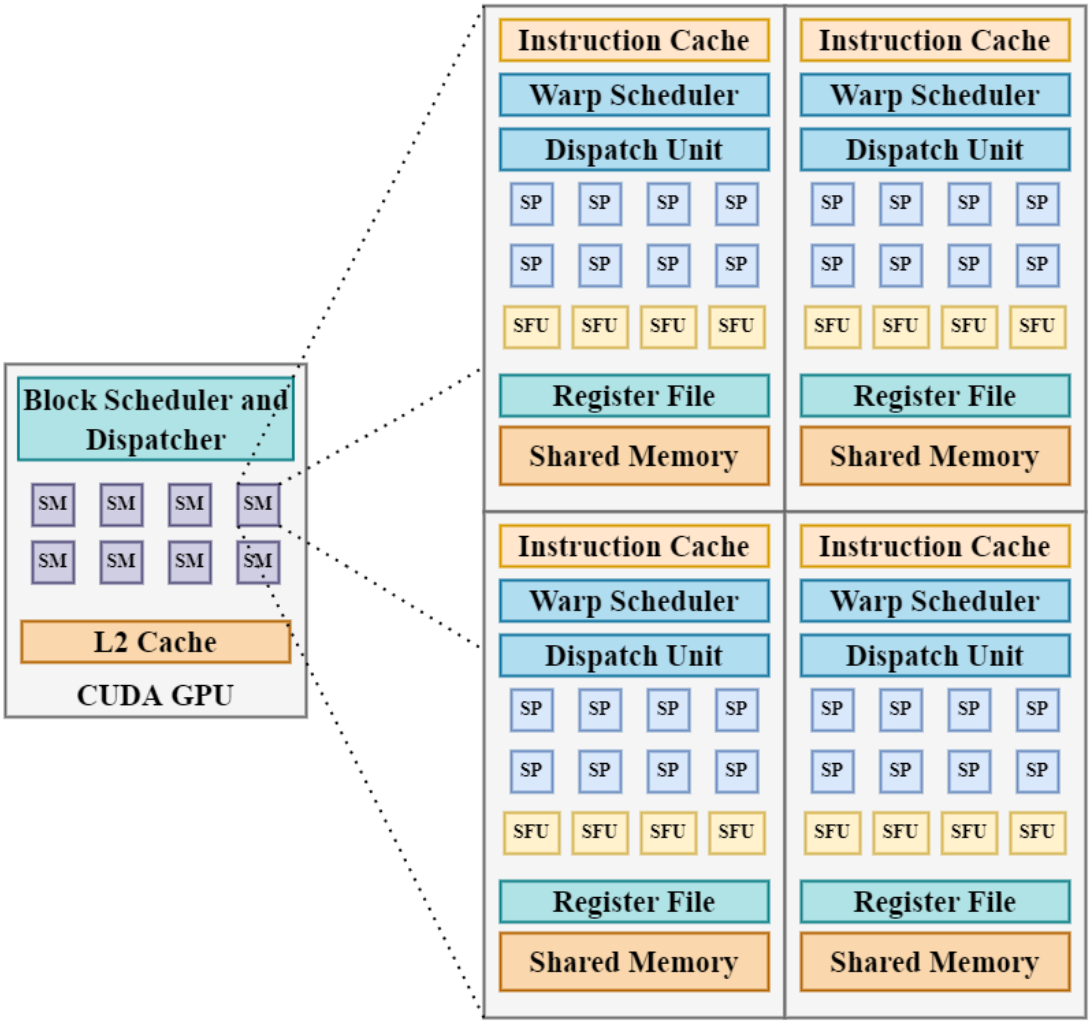


Figure 2.3: General CUDA GPU architecture and SM detail

language mostly adopted in NVIDIA GPU programming is Compute Unified Device Architecture (CUDA) which is an extension to the C language used to invoke a kernel and set up device-specific operations. The CUDA software stack consists of CUDA hardware driver, mathematical libraries and CUDA API [10].

A basic NVIDIA GPU consists of an array of computing unit blocks known as Streaming Multiprocessors (SMs), which are able to execute multiple threads on separate execution units independently; having their own context, with its own instruction address and register state. The main parts composing a GPU are shown in the Figure 2.3. It is possible to observe the presence of the Block Scheduler used to assign block of threads to different SMs and the SMs consisting of independent Warp Schedulers, Dispatch Units, Shared Memory, Register File and a set of functional units (Streaming Processors or CUDA cores) able to execute integer, floating-point, load-store and transcendental function operations [8] [11].

The Block scheduler is in charge of distributing, on the available SMs, blocks of

threads to be executed in parallel following the maximum occupancy of the GPU. This allows to maximize the parallelization of tasks. Each block of threads is divided in groups of 32 threads called *Warps*. Many Warps are then assigned to execute concurrently on a single GPU SM in a bench of execution called cooperative thread array (CTA). The Warp Scheduler, internal to the SM, issues successive instructions to the Warp's active threads.

Consequently, the SM maps the Warp's threads to the internal cores, and each thread executes independently on a selected core.

Since each SM is divided in four sub-blocks each containing several cores, the allocation is done by mapping the Warp identifier to the range (0-3); whereas the Lane identifier is used to define the used core, in dependence on the type of instruction.

The SIMT strategy works with full efficiency when all 32 threads of a warp take the same execution path. Some individual threads can be inactive due to independent branching or predication. If threads of a Warp diverge via a data dependent conditional branch, the Warp executes each branch taken path serially, disabling execution of threads that are not on that path; and when all paths complete, the threads "reconverge" to the original execution path. Branch divergence can only occurs within a Warp and different Warps execute independently regardless of whether they are executing common or separated code paths.

In Figure 4.3a and 4.3b, it is possible to observe two different NVIDIA SMs' microarchitecture. The SMs differ for the number and the type of Functional-Units available. These characteristics, with the number of present SMs in the GPU defines the Compute Capability of the GPU. The compute capability is the "feature set" (both hardware and software features) of the device. NVIDIA GPU architecture names are related to that parameter: "Maxwell", "Tesla", "Fermi", "Kepler" or "Ampere". Each of those architectures have features that previous versions might not have. For example Ampere architecture based GPU compute capability is 8.x while Maxwell architecture based GPU compute capability is 5.x [12] [13].

The Figure 2.4 shows that each SM is divided in four equal sub-blocks. Each of them contains a set of cores; in particular Maxwell SM provides 32 Integer-Floating Point (FP) cores and 8 SFUs (Special Function Units) while Ampere SM contains 32 Floating Point (FP), 16 Integer (INT) cores and 4 SFUs. Ampere and other recent architectures as Tesla, also include Tensor cores to perform matrix multiplications very quickly. Ampere architecture GPUs consists in many SMs getting them an higher compute capability with respected to Maxwell. This reflects in higher performances thanks to the greater parallelization of tasks. As an example, while Maxwell architecture GPU contains one SM, recent GeForce 30 series GPUs based on Ampere architectures, provide tens of SMs working in parallel, hence executing much more blocks of threads simultaneously.

(a) Maxwell Architecture SM. (b) Ampere Architecture SM.

Figure 2.4: NVIDIA Streaming Multiprocessor (SM), taken from [12] and [13]

2.2 General-purpose computing on GPU: Convolutional Neural Networks

In this section the basic idea behind CNN algorithms are described, then it is provided a view on the CNN adopted for the developed environment, namely LeNet.

An Artificial Neural Network is an algorithm model widely adopted for machine learning. The field of machine learning is concerned with the question of how to construct computer programs that automatically improve with experience [14]. Deep learning is machine learning with deep artificial neural networks, that are neural networks composed of at least two or three 'layers'. Each layer of a ANN is composed of set of simple elements called *neurons*. In Figure 2.5 it is possible to observe a simple neural network composed of three layers (one input layer, one output layer and one hidden layer between the previous two) and the neuron 3 of layer 2.

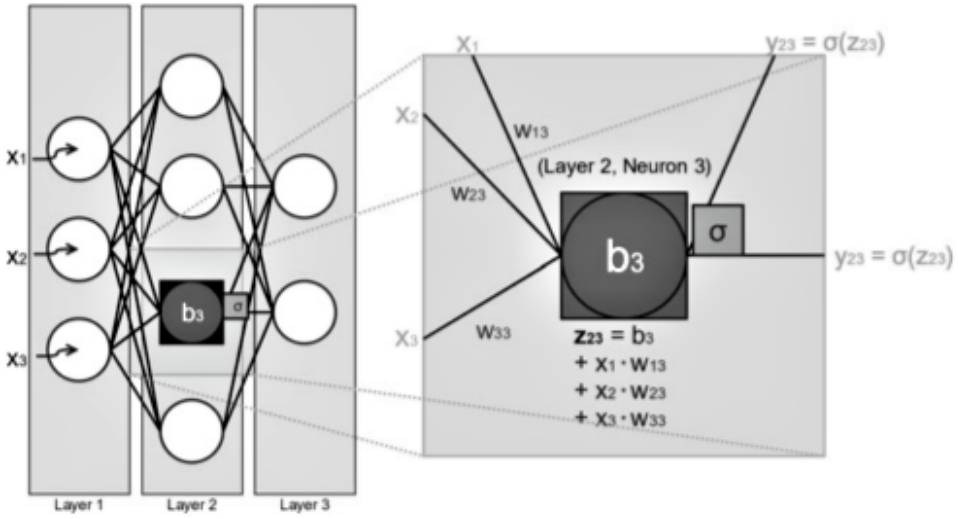


Figure 2.5: Three Layers Neural Network, taken from [15]

The neurons of the input layer can accept one input value(x_i), whereas neurons of other layers have more than one input corresponding to previous layer output. Referring to the equation 2.1, each Neuron gets the input which is the sum of the products of the inputs from the previous layer and respective *weight* (w_{ij}). To this sum a value called *bias* (b_j) is added and the final value passes through an activation function σ, that can be linear or non-linear.

$$y_{ij} = \sigma \left[\left(\sum_{i=0}^{n} w_{ij} * x_i \right) + b_j \right] \tag{2.1}$$

The weight regulates how much of the initial value will be forwarded to a given

neuron, while the bias is a modifiable value. One of the purposes of deep learning is to perform image prediction. The weights and biases play a crucial role to make correct predictions: the whole point of a neural network is to find a good set of weights and biases, and this is done through 'training' via backpropagation, which is the reverse of a forward pass. The idea is to measure the error the network makes when classifying and then modify the weights so that this error becomes very small [16].

With respect to ANN, a Convolutional Neural Network (CNN) consists in a neural network in which hidden layers perform convolutional operations. A CNN is composed at least of three types of layer:

Layer: it performs convolution operations as illustrated in Equation 2.1; each neuron performs the same operation on the complete set of input data;Max-Pooling Layer: it performs an arithmetic operation on the previous layer output to reduce the data size. It takes the output of each convolution kernel and reduces it by a defined algorithm, which can be to return the maximum value, the average value, or others; Fully Connected Layer: it generates the output of the network; each neuron in that layer is connected to all the outputs of the previous layer.

2D image prediction takes the use of CNNs, since they perfectly fit to pattern recognition and image classification. A 2D image can be modeled as a matrix representing pixels values; so the value for each pixel ranges between 0 (black) and 255 (white) for one channel image and 0-255 for each colour in RGB scale for three channels (coloured image), meaning that a 2D convolution, between weights kernel and input matrix local receptive field, is performed. Referring to Figure 2.6, a convolutional layer is usually followed by a max-pooling layer used to get a smaller image but denser of information. This obtained compacting input matrix resorting to near-elements (pixel values) information. It is possible that multiple convolutional layers and max-pooling layers alternate. Then, it is present a flattening layer, used to get a 1D vector representation of the output matrix; and fully connected layer used to get final result trough a single neuron.

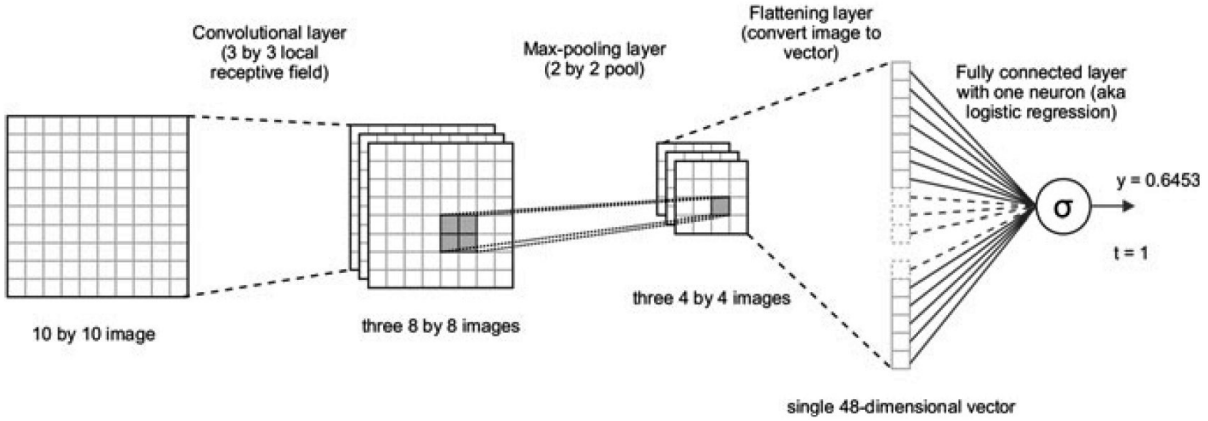


Figure 2.6: A convolutional neural network with convolutional, max-pooling, flattening and fully connected layers with one neuron, taken from [15]

Notable CNN models are AlexNet [17], GoogLeNet [18] and LeNet [1]. AlexNet is composed of many stacked convolutional layers; its peculiarities are: the use of Rectified Linear Units (ReLU), with respect to the non linear tanh activation function, allows to reduce the training time; and elements overlapping pooling allows achieving an error reduction at the cost of an increase of total number of parameters needed to execute the network. GoogLeNet architecture solved most of the problems that large networks faced, mainly through the Inception module's utilization. The Inception module is a neural network architecture that leverages feature detection at different scales through convolutions with different filters and reduced the computational cost of training an extensive network through dimensional reduction [17]. The GoogLeNet architecture consists of 27 layers, and part of these layers are a total of 9 inception modules. The expensive Fully-connected layers at the end of the model of common CNN is substituted with a simple global average-pooling layer, which averages out the given layer input values of each feature map. This change has dramatically reduced the number of parameters used in the model, which made it a faster in the training phase, lighter in size, and higher in performance, compared to other architectures.

In this thesis the LeNet CNN model available in the Darknet [2] environment was used. In addition, MNIST dataset [3] for training and inference is adopted.

LeNet (Figure 2.7) is one of the most famous CNN architecture able to infer 32x32 pixels gray-scale images. LeNet CNN architecture is made up of 7 layers. The layer composition consists of 3 convolutional layers, 2 subsampling (or max-pool) layers and 2 fully connected layers.

The input layer should take image in 32x32 form, and these are the dimensions of images that are passed into the next layer. If an input image has smaller dimensions, as in MNIST dataset [3], to meet the requirements of the input layer, the 28x28 images

are padded, meaning that some zero are added in the edges, not changing image content [1][15].

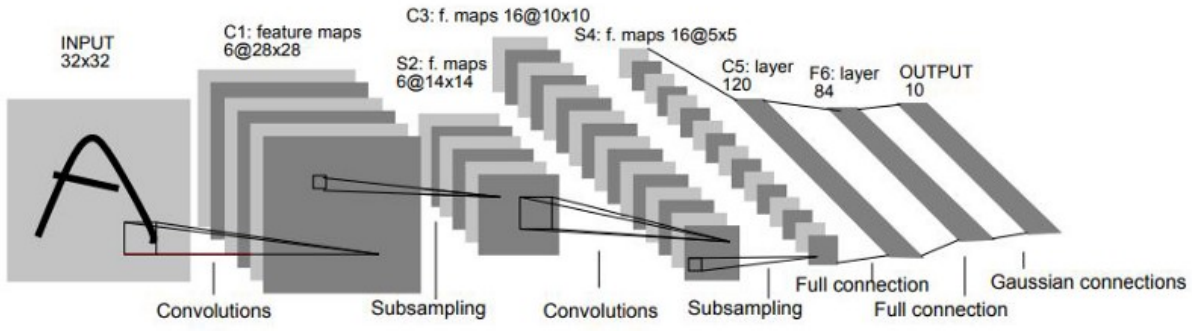


Figure 2.7: LeNet CNN, taken from [1]

The implementation of this CNN model, in the Darknet environment, on a Maxwell architecture GPU, resorts to eight different kernels executed in the various layers. In the Table 2.1 for each type of layer present in the CNN the belonging kernels are listed and the tasks of each layer are reported. It is worthy to highlight that to perform matrix multiplication cuBLAS library is employed; therefore, during the compilation stage of convolutional layers the most suitable kernel is implemented. This is done according to the GPU micro-architecture in order to use efficiently the device.

Layer types	Kernels	Task
Convolutional and Fully Connected	sgemm_NN, sgemm_NT_vec, im2col_gpu_kernel, fill_kernel, add_bias_kernel, activate_array_kernel	Operation to matrix multiplication operation and add biases to the necessary parameters after the matrix multiplication. Apply non linearity to the feature maps to reduce the input linearity for the next layer
Max Pooling	forward_maxpool_layer_kernel	To reduce the spatial dimension of the input volume for next layers
Softmax	softmax_kernel	To calculating the probabilities of each class

Table 2.1: LeNet kernels

As previously anticipated the environment developed take the use of the MNIST dataset. The MNIST database, which stands for Modified National Institute of Stan-

dards and Technology database, is a large dataset of handwritten digits. It contains 60,000 training images and 10,000 testing images. As shown in Figure 2.8, each image is a scan of a handwritten digit from 0 to 9, and the differences between different images are a result of the differences in the handwriting of different individuals [3]. Each image consists in 28x28 pixels, and each pixel value takes on a value from 0 to 255 (greyscale). The labels associated with the images correspond to the ten digit values.

Figure 2.8: Some handwritten digits in the MNIST dataset, taken from [3]

2.3 Previous Work

The main purposes of Graphical Processing Unit, until a few years ago, were related to video encoding and graphics rendering; in which possible faults do not generate catastrophic behaviors. Nowadays GPUs are used to accelerate CNNs models in many safety-critical applications, in which it is crucial to guarantee a certain level of reliability. One of the major sources of unreliability in GPUs is soft errors, typically caused by high energy particles striking electronic devices and causing them misclassification (e.g., flip a single bit). As mentioned, these errors can impact the behaviors of the GPU dramatically. Therefore, in safety-critical applications, it is important to ensure that potential data corruption is avoided and failure rates must be reduced to the minimum and should not exceed, for example, 10 Failures in Time (FITs), which equals to a single error in 10^9 hours of operations.

Reliability evaluation of CNN has been treated in previous works in different fields. Many works take use of compiler-based instrumentation tools to perform transient fault injections campaigns.

The CNN AlexNet model has been studied in healthcare field for image classification in disease detection [19] and for autonomous drive accident prevention [20]. In the first case, the AlexNet model was trained and evaluated with a Cholec80 dataset for surgical tool detection. In the two papers soft injections campaigns were performed on NVIDIA GPUs using the compiler-based fault injector SASSIFI [21], able to perform transient fault injection campaigns, targeting instruction output value (IOV), instruction output address (IOA) or randomly selected register (RF), at low level. The tool functions were used to evaluate the Program Vulnerability Factor (PVF) in terms of Kernel Vulnerability Factor (KVF). This is achieved by dividing the fault simulation in different phases; one for each individual kernel type. Therefore, injections are performed for separate instructions of the same kernel, belonging to two categories depending on the type of operation performed: GPR (use of general purpose registers) or LOAD/STORE. The study was dealt establishing a number threshold of critical faults; and some Soft hardening techniques were proposed, by identifying the most vulnerable kernels to misbehavior for the CNN. In [20] it is proposed an alternative strategy for GPU design; consisting in properly select just the instructions with higher Instruction Vulnerability Factor (IVF) replicating them to reduce the presence of critical errors; reducing even more the overhead.

Similar works report the reliability evaluation of VGG-16 [22], GoogLeNet [23] and ResNet [24] CNN models on NVIDIA GPUs, resorting to SASSIFI fault injector, again for transient fault injection campaigns. In those papers Layer Vulnerability Analysis is carried out and complete layers of kernels were instrumented to perform error evaluation. The results show that the early convolutional layers are more susceptible to faults than later layers, while fully-connected layers do not generate data corruption errors.

Another remarkable investigation, consisting in comparing soft fault injection experiment of a CNN running on a GPU accelerator with a particle beam radiation-induced errors campaign affecting elements in GPUs, is reported [25]. This study confirms the possibility to consider software fault injection approaches as a valid emulation of the presence of hardware fault in GPUs' internal locations; since the outcomes obtained with the two techniques result to be similar.

In the light of this information, this research work is motivated by the demand of exploring the idea of performing permanent fault injection campaigns, not yet investigated, in hardware accelerators for CNNs execution. This last fault model is challenging to implement at architectural level on a GPU, since the effect of this kind of fault must be kept stable propagated for the complete duration of application execution.

2.4 NVBit binary instrumentation tools

Binary instrumentation is a technique of hardware injection through program transformation that allows transformation of application binary code for a variety of purposes, including application profiling, error checking, and fault injection. It is used to develop custom tools used to apply specific transformations to a code.

Many instrumentation tools, as SASSIFI [21], are compiler-based, meaning that they are tied to a specific compiler version. Furthermore, they need the source code recompilation, since they cannot target the code generated by the GPU driver via PTX translation. In addition, they are not compatible with common used machine learning libraries as cuBLAS. This issue is avoided by NVBit tools; which are able to dynamically perform code instrumentation, allowing to instrument unknown libraries during the build time.

The NVBit core is composed of a single static library (libnvbit.a) and header file (nvbit.h) that provides all the APIs required to implement an NVBit tool. An NVBit tool is created by:

- developing a .cu file, implementing a GPU device function which is injected in an application's GPU kernels according to user defined injection points. This process make use of NVBit APIs: Callback, Inspection, Instrumentation, Control, and Device [26];
- compiling it with NVIDIA NVCC (Nvidia CUDA compiler);
- linking it with the static library libnvbit.a.

This process generates a shared library (.so extension); which is than injected into applications using the CUDA driver. This mechanism is based on the usage of standard Linux LD_PRELOAD environment variable, that contains paths to shared libraries, to be loaded before any other library. The Figure 2.9 illustrates the flow for NVBit tool compilation and run-time preloading; as described above.

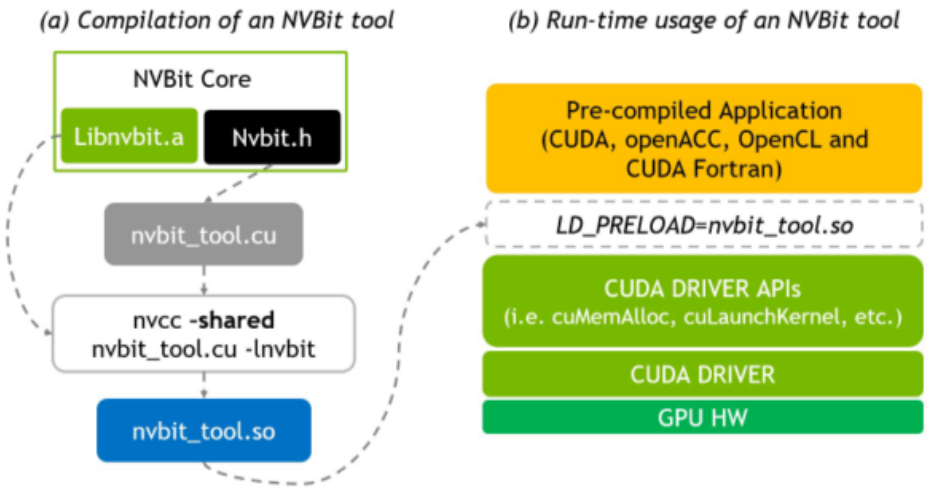


Figure 2.9: NVBit core overview, taken from [27]

The Callback APIs are triggered by the NVBit core when a particular event in the target application is encountered. These events are application start or termination, and entry/exit of any CUDA driver API call. The mechanism behind the instrumentation of single instructions is illustrated in Figure 2.10. At the exit of the CUDA driver callback, if instrumentation was applied through instrumentation API, a Code Generation starts.

The original code is first inspected to find the position in which perform the instrumentation, through an *Instrumentation Function*. In the Figure 2.10 the instrumentation is performed before the highlighted instruction. An instrumented code is saved in system memory; note that the selected instruction is substituted to a new instruction (JMP) used for subroutine call. A new piece of code named *Trampoline* is saved in GPU memory. The Trampoline function saves the state of the thread, before the instrumentation execution, pass the argument needed for instrumentation, call an instrumentation function (in the Figure 2.10 JMP "foo"), restore the state and finally execute the original instruction previously substituted before jump back to the original code execution [27].

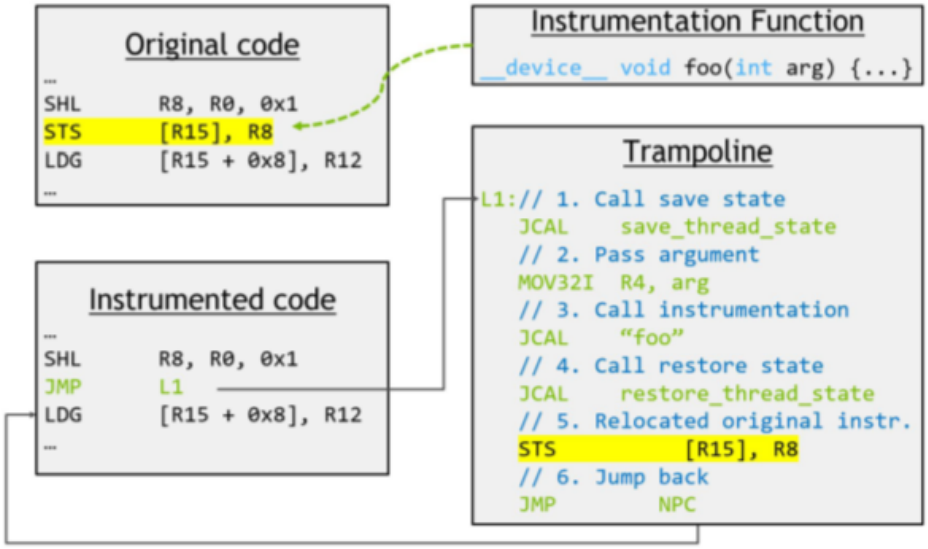


Figure 2.10: Instrumented code generation process, taken from [27]

With the approach described, in a previous work, a tool called NVBitFI was developed [28]. The key advantages of NVBitFI with respect to compiler-based tools, are due to dynamic instrumentation of program binaries and the use of a generalized GPU architecture abstraction. It doesn't require any pre-compiled source code; this makes it portable across NVIDIA GPUs because even if instruction set architecture (ISA) is the same their encoding can change across generations; and can inject errors into dynamically selected kernels.

The tool mainly support the transient fault model that occurs in the GPU compute pipeline or memory read subsystem [29]. NVBitFI mostly consists in a profiler and a transient fault injector. The tool is able to perform injections, specifying the type of instructions that should be injected and the bit-flip model indicating the type of bit-level corruption. The injections are performed in a specified kernel instance, applying to selected instruction destination registers a mask randomly selected or user-specified.

The tool supports, also permanent fault injection to a limited extent; it is possible in fact to perform injections in all threads that execute in the target SM and lane, for a single instruction Opcode type, modifying the target destination register, with a XOR bit mask.

The framework illustrated in this report was built using the NVIDIA NVBitFI tool. This tool was adapted and customized, extending its capabilities to support the permanent fault injection and propagation of hard errors that mimic the presence of permanent faults in the hardware component of the GPU: Register File and Functional Units.

CHAPTER 3

Performed Activities: Permanent Fault Injection Framework

3.1 Injection Campaign Environment

In this thesis work it is proposed an environment based on NVBitFI instrumentation tool adapted to perform permanent fault injection campaigns in NVIDIA GPU executing, in the Darknet framework [2], the LeNet CNN model described in the previous chapter, using the Training & Inference MNIST dataset [3]. To run the injection campaign it is used a set of images not used during the training phase. In particular, 100 images were selected as input data to be inferred concurrently. In the Figure 3.1 it is illustrated the complete environment; consisting in Profile and Injection Blocks (written in CUDA resorting the NVBitFI's Application Programming Interface (API)), Permanent Fault List Generator and Classifier; all included in a global Controller. This element was made in Python is composed by four main scripts run_profiler.py, Neural_list.py, Neural_classifier.py, Neural_parser.py.

The Controller manages the whole environment; it is responsible of performing several tasks:

- Profile the target application to collect information, like the number of used SMs, threads in each kernel, registers used by each thread in kernels and also the opcode types used by the target application;
- Perform an initial fault-free simulation to obtain a reference scenario, used to perform output data comparison;
- Generate a fault list resorting to the information gathered during the profile phase, according to the permanent fault injection defined target; this process can also be

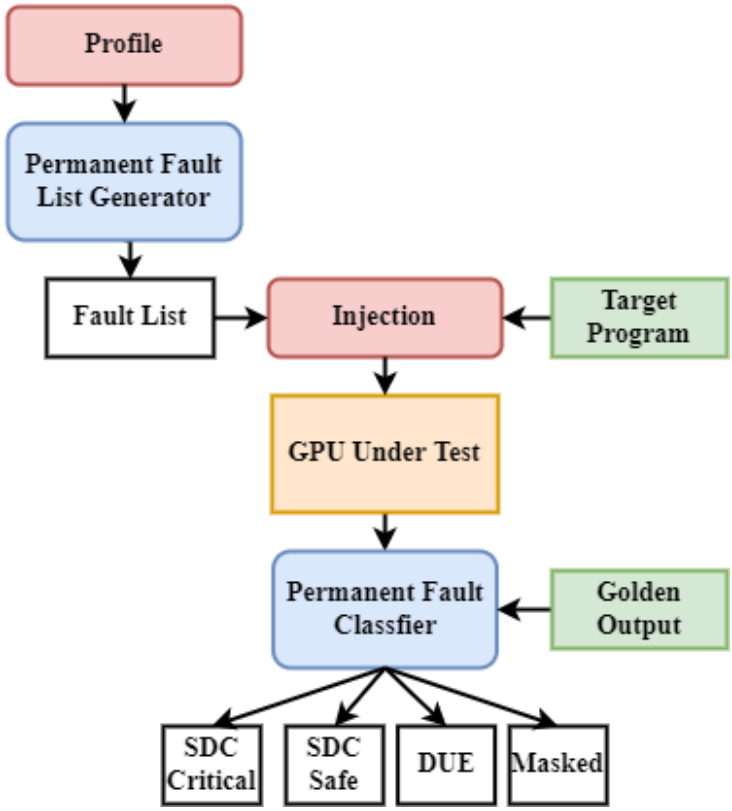


Figure 3.1: Permanent Fault Injection Environment

skipped if a previously generated fault list is adopted;

- Perform the permanent fault injections campaign by injecting one fault at a time, propagated through all the executing kernels, up the application process conclusion;
- Collect the results, and compare them with the golden reference model, to classify the injected error.

A permanent fault is defined as an error that persists indefinitely (or at least until repair) after its occurrence [30]. The presence of the permanent fault generates different possible effects on the CNN behavior. The output generated by the CNN for a single image is in the form illustrated in the table 3.1. Each inference class representing a decimal digit from 0 to 9, it is associated to a percentage indicative of the prediction probability. The highest percentage is associated to the first prediction class. Therefore, the classification process consists in comparing for each inference class the prediction percentages of faulty and fault-free scenarios, according to the expected order.

Image 0_Five.png	
Percentage	Inference Class
77.40%	Five
19.95%	Three
2.45%	Two
0.15%	Seven
0.05%	Zero
0.00%	Eight
0.00%	One
0.00%	Six
0.00%	Nine
0.00%	Four

Table 3.1: Darknet Output Format example

A fault can generate a Silent Data Corruption (SDC) when the error's propagation produces a different inference result, for at least one image of the tested set of images, with respect to fault-free case; or it is possible that the fault generates a Detected Unrecoverable Error (DUE) causing the CNN not to be able to produce the inference result at all. For each image the fault can have a specific impact on the final classification; in particular, for each of them a permanent fault can be classified in four possible categories:

- **SDC-Critical**: the fault propagates to the CNN output, modifying the probabilities vector during the inference calculation producing a wrong classification result.
- **SDC-Safe**: the fault propagates to the CNN output, modifying the probabilities vector during the inference calculation, but the classification output is still correct, meaning that the inference percentage is different with respect to golden model.
- **DUE**: the fault produces a crash or stuck. This error interrupts the execution of the CNN at any time. The causes of this behavior can be memory access violation, memory misalignment violation, or timeout (the error blocks the CNN model in an infinite loop).
- **Masked**: the fault does not have any impact on the output, meaning that, the output is the same as the fault-free scenario.

The error injection is performed by the instrumentation mechanism by NVBitFI. The tool is able to instrument the kernels of the target application, injecting an error at Source and Assembly (SASS) instruction level. Each executing instruction is checked,

than eventually instrumented. The instructions are examined in order to detect meaningful information as the instruction opcode, the destination and source registers. If the instruction characteristics correspond to the injection specification, it is instrumented according to the injection model adopted. The instrumentation consists in applying a bit-flips or a single bit stuck-at mask to an identified register. After that, the kernels execution continue, repeating this process for each instruction of each kernel. The developed environment is able to perform permanent fault injections campaigns targeting different GPU internal modules. These are: the Register File (RF) and the Functional Unit (FU). As described in Section 2.1 when a group of Threads is issued, they are assigned to a specific group of GPU units; and depending on the fault injection target these units need to be in someway identified. In the following the various scenarios taken into account are described. For each of them a specific permanent fault injector is designed.

3.2 Permanent Fault injection in the Register Files

Fault injections are performed at assembly code level; for this reason, it necessary to illustrate SASS instructions' composition. The NVIDIA GPU architectures have the following instruction set format:

$$(instruction\ opcode)(destination)(source1),(source2),(source3) \tag{3.1}$$

Valid destination and source locations include: RX for registers, SRX for special system-controlled registers, PX for condition registers, c[X][Y] for constant memory [31]. During the execution of a kernel each thread has access to a private set of registers (RX), used to perform the needed operations. The Figure 3.2 illustrates that each warp resident in a SM (in Maxwell Architecture the max number of resident warps is 64) is composed of 32 threads issued together, each thread of a warp access privately to a set of registers in the SMs' Register File, used to execute the instructions. Since each GPU has a maximum number of threads running in parallel; to target a single thread is necessary to select a Warp and the position of thread in the Warp (Lane identifier). Therefore, in order to perform a fault injection in a target register is necessary to select a register number, a warp identifier and a lane identifier. This is mandatory to guarantee the error injected is present permanently during all the application's kernels execution. In addition, it is worthy to highlight that, if the number of threads is large enough to require several blocks to be executed on the same SM, the permanent fault on one particular register affects more than one thread.

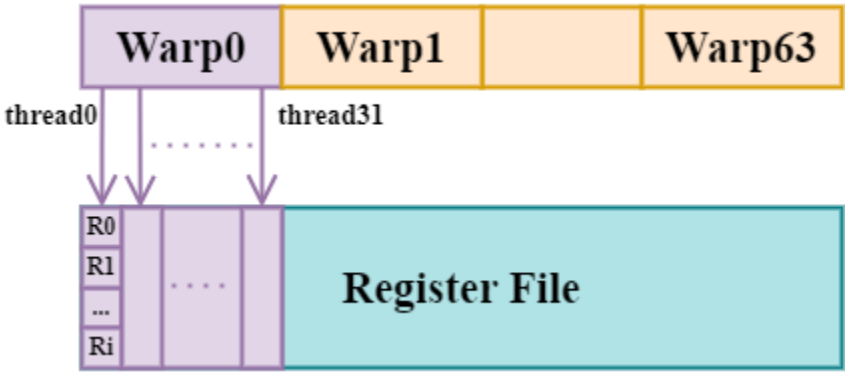


Figure 3.2: Threads-Registers assignment

Therefore, each fault is defined by the quintuple <SMID, threadID, Register, Mask, stuck-at> where SMID represents the SM where the injection should be performed out of the many possibly available; threadID is the resident thread selected in which to perform the injection; Register is the target destination register to be injected; Mask is the single bitmask to be applied to the target register value; stuck-at can be 1 or 0 depending on the value to be forced in the defined bit. In Figure 3.3 they are illustrated the phase of the injection campaign. First on the host side the pf_injector go through all the instructions to be executed in each kernel, checking for each of them if the destination register (see instruction format 3.1) corresponds to the fault's target register; if the two match the injection function (inject_funcs) is launched on device side. The injection propagates thorough all the kernels composing the program. Each kernel is inspected in this way.

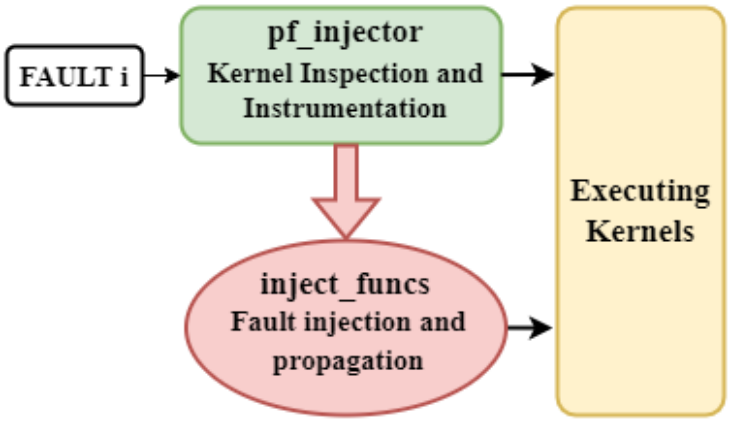


Figure 3.3: Permanent Fault Injector

To perform the real injection on GPU side, three conditions must be satisfied:

1. The Straming Multiprocessor where the injection is performed corresponds to the fault target one;

2. The Warp of the injected thread is the same of the target one; it is calculated as the ThreadID divided for Warp size equal to 32 in each of the NVIDIA architectures;

3. The Lane of the injected thread is the same as the target one; it is obtained performing modulus operation between the ThreadID and the Warp size (32).

In Figure 3.4 it is depicted the injection process at assembly code level. In the Figure the faulty register is R0 and the a specific bit is stuck-at 0 or 1. Therefore, after the execution of an instruction using R0 as destination register the faulty bit is masked. The masking is implemented by applying a bitwise AND, in case of stuck-at 0, and a bitwise OR, in case of stuck-at 1.

```
MOV R1, c[0x0][0x20] ;
S2R R0, SR_CTAID.X ;
Fault Injection in R0
S2R R2, SR_CTAID.Y ;
XMAD R0, R2.reuse, c[0x0] [0x14], R0 ;
Fault Injection in R0
XMAD.MRG R5, R2.reuse, c[0x0] [0x14].H1, RZ ;
```

Figure 3.4: Injection Process RF: fault in R0

3.3 Permanent Fault injection in the Functional Units

The fault injection in Functional Unit (FU) differs from RF injections, since for each FU many different instructions opcodes execute on the same core. For this reason it is crucial, to guarantee the permanence of the error, to instrument all the matching instructions executing on the same core. In Figure 3.5 it is possible to observe a simplified view of a SM organization and the interior of a Streaming Processor (SP). It is worth noting the presence of three types of blocks addressed by the developed injector: Integer Unit, Floating Point Unit inside a SP and Special Function Unit (SFU) which executes transcendental functions. For this reason, the three corresponding opcode categories are taken into account.

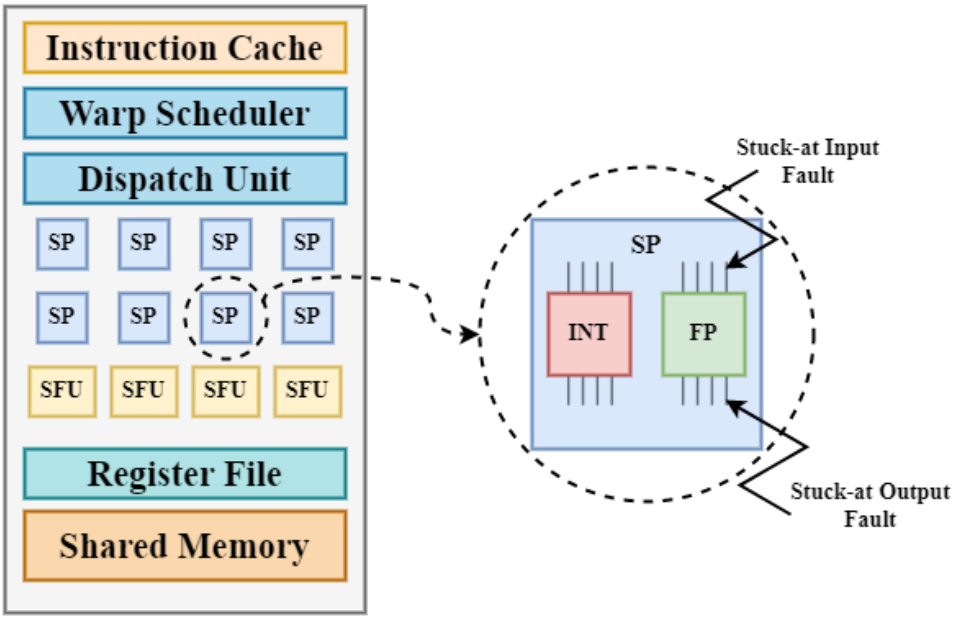


Figure 3.5: Streaming Processor view

In the Table 3.2 a set of opcodes of the three selected categories is reported. The table reports the instructions, belonging to the three categories, present in the LeNet target application, in the Darknet environment, executed on a Maxwell architecture GPU.

Instruction Opcodes		
Opcode	Type	Description
FADD	FP	FP32 Addition
FCMP	FP	FP32 Compare
FFMA	FP	FP32 Fused Multiply Addition
FMUL	FP	FP32 Multiply
FMUL32I	FP	FP32 Multiply Immediate
FSET	FP	FP32 Compare And Set
MUFU	SFU	FP32 Multi Function Operation
RRO	SFU	Range Reduction Operator FP
BFE	INT	Bit Field Extract
IADD	INT	Integer Addition
IADD3	INT	3-input Integer Addition
IADD32I	INT	Integer Addition Immediate
ICMP	INT	Integer Compare and Select
IMNMX	INT	Integer Minimum/Maximum
ISCADD	INT	Scaled Integer Addition
ISCADD32I	INT	Scaled Integer Addition Immediate
LOP	INT	Logic Operation
LOP3	INT	Logic Operation 3-operands
LOP32I	INT	Logic Operation Immediate
SHL	INT	Shift Left
SHR	INT	Shift Right
XMAD	INT	Integer Short Multiply Addition

Table 3.2: Opcodes list

In Figure 3.6 it is possible to observe the input and output interconnections of a generic Functional Unit (FU).

For each instruction execution, the input interconnections are fed with values coming from the source registers in the Register File, used by the instruction itself, as operands. For the selected application the maximum number of input operands resident in general purpose register is three. The destination register, instead, is fed by the Output interconnections.

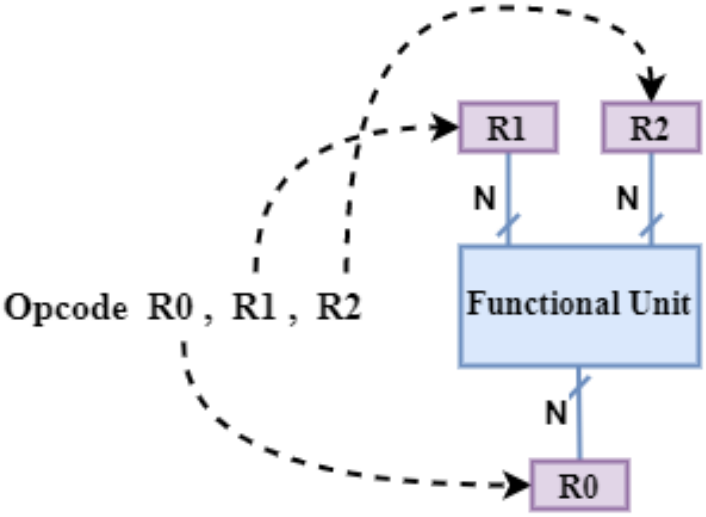


Figure 3.6: Functional Unit Interconnections

Hence, to mimic the presence of a hardware permanent fault in FU, we need the instrumentation of destination or source registers of a specific instruction opcode type. Since each instruction can use up to three source registers (as operands), and mandatorily one destination register, the permanent fault injection is obtained by masking for each selected instruction one register, destination or source. This is implemented targeting a single opcode type, so that all matching instructions are instrumented.

As a consequence, it is needed to define, for each fault, seven parameters <SMID, SubSM, Core, Opcode, Mask, Register Position, Stuck-at>. SMID and Stuck-at parameters follow the same explanation given in the previous section. SubSM represents the subpart of the SM where the injection should be performed; Core is the internal unit number selected in which to perform the injection; Opcode is the target instruction opcode type to be injected; Register Position, representing the correspondent interconnection, is the target register out of the possible four for each instruction to be injected. If the selected register is the destination one, the fault injection is performed after the instruction execution, meaning that the output interconnections is affected (Figure 3.7). If, instead, the register selected for the fault injection is a source, the injection is performed before instruction execution (Figure 3.8).

```
MOV R1, c[0x0][0x20] ;
S2R R15, SR_CTAID.X ;
S2R R0, SR_TID.X ;
XMAD.MRG R2, R15.reuse, c[0x0] [0x8].H1, RZ ;
Fault Injection in R2
XMAD R0, R13.reuse, c[0x0] [0x8], R0 ;
Fault Injection in R0
XMAD.PSL.CBCC R15, R14.H1, R2.H1, R0 ;
Fault Injection in R15
ISETP.GE.AND P0, PT, R15, c[0x0][0x140], PT ;
```

Figure 3.7: Integer Unit Fault Injection: XMAD instruction

```
MOV R1, c[0x0][0x20] ;
S2R R15, SR_CTAID.X ;
S2R R0, SR_TID.X ;
Fault Injection in R15
XMAD.MRG R2, R15.reuse, c[0x0] [0x8].H1, RZ ;
Fault Injection in R13
XMAD R0, R13.reuse, c[0x0] [0x8], R0 ;
Fault Injection in R14
XMAD.PSL.CBCC R15, R14.H1, R2.H1, R0 ;
ISETP.GE.AND P0, PT, R15, c[0x0][0x140], PT ;
```

Figure 3.8: Integer Unit Fault Injection: XMAD instruction

In that case, all the instructions opcode are checked by the pf_injector on CPU side; if they match with the opcode pointed by the fault the injection is performed if three conditions are fulfilled:

1. The Streaming Multiprocessor where the injection is performed corresponds to the fault target one;

2. The SubSM of the injected thread corresponds to fault objective; it is computed by retrieving the thread Warp and applying modulus four operation;

3. The target core of the fault injection is the same of core in which the thread is executing computed as the Lane modulus the number of Functional Units of the

opcode type, present in the sub-part of the SM.

Finally, the injection is performed masking the selected bit location.

An alternative approach, supported by the developed framework, consists of performing single bit-flip masking through the usage of a XOR instruction. That fault model, already proposed by NVBitFI, has been integrated in a dedicated injector. This model is intended to mime the presence of an hard fault inside the GPU FUs; however, this model is considered less realistic, with respect to the stuck-at model, of hardware faults.

The Functional Unit targeting is performed resorting to some allocation parameters. In fact, the fault injector function is adaptable to different micro-architectures, depending on the number of available internal cores for each type (INT, FP, SFU). The injector is informed about the compute capability of the GPU and according to the injected opcode category, the core number is selected out of the available ones, for the architecture family.

The environment contemplates the possibility of injecting one or two instruction opcodes (belonging to the same category) for each fault; in both cases, only the opcodes present in the application code instructions are used for the fault list generation and injections. This strategy reduces the presence of Masked faults, since not present opcode types would not be injected at all.

CHAPTER 4

Experimental Results

The developed fault simulator tool is evaluated using a LeNet pretrained model available in the Darknet environment. The Darknet functions were adapted in order to support the possibility to predict 100 images concurrently. The test_classifier function of Darknet was used for that purpose. The trained LeNet model can classify images of handwritten digits (0 to 9). For the experiments, we used the MNIST dataset described in Chapter 2.

The CNN was evaluated using the embedded platform Jetson Nano, which has an NVIDIA Maxwell architecture and Compute Capability 5.3. For the experiments, the LeNet model was evaluated by performing fault injections on the register file and the functional units on one Streaming Multiprocessor.

4.1 Register File permanent fault simulations

The RF fault injections campaign was divided in two phases. In the first phase, an exhaustive fault injection was performed; meaning that for a single thread, each register is tested for all possible stuck at fault. The thread simulated is thread 0 executing on the only SM. Using 152 registers, the total amount of fault tested was 9,792 and the simulation took 28 hours. The results are shown in the Table 4.1; from there, it is possible to observe the predominance of SDC faults. This result justifies the possibility of performing permanent fault simulation at register level, guarantying a substantial fault coverage.

Fault Classification			
SDC-Critical	SDC-Safe	Masked	DUE
7.96%	39.43%	21.10%	31.51%

Table 4.1: Register Files Exhaustive Fault Simulation results (thread 0, SM 0), Fault Classification

Faults propagated to the Output Classification		
SDC-Critical	SDC-Safe	Masked
11.62%	57.57%	30.82%

Table 4.2: Register Files Exhaustive Fault Simulation results Only faults propagated to the Output (thread 0, SM 0), Fault Classification

In the Figure 4.1 the simulation outcome for the complete set of registers used by the application is shown. The fault classification has been performed considering as reference model the golden model, meaning that, if in the fault free scenario, a certain image is classified wrongly, it is still considered the correct outcome. In addition, if a fault generates a critical result for at least one image, out of the 100 inferred, the fault is labeled as critical. From that image, it is possible to confirm that the first ten registers are affected by a lot of critical failures and DUEs. In Figure 4.2 it is reported the accuracy degradation with respect to the reference accuracy, equal to 92%. This values basically means that on 100 images, 92 are correctly inferred. In Figure 4.2a the accuracy of faulty scenarios is calculated considering only faults which doesn't generate unrecoverable errors (DUE); in other words, the faults considered are the ones propagating until the CNN output. Instead, in the Figure 4.2b all the faults are included, for the average accuracy calculation. The Images highlight that the first set of registers (R0-R9) are the most critical; since faults would affect dramatically the CNN accuracy, with a large degradation from the fault free model.

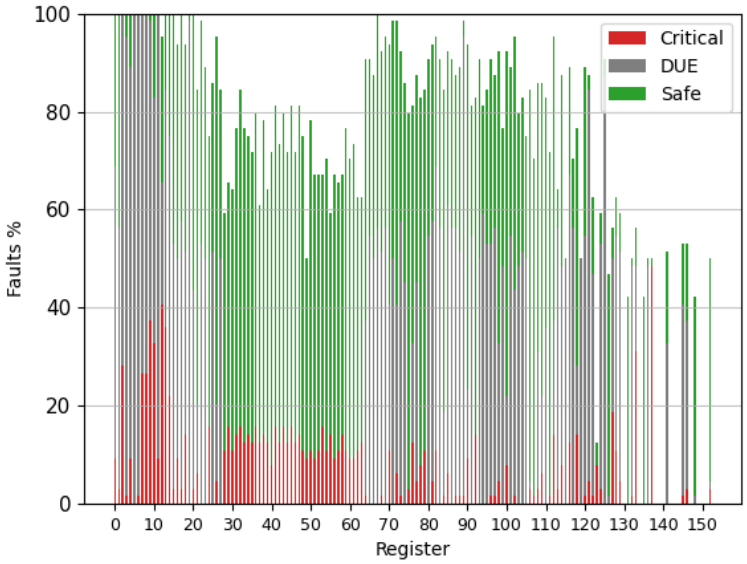
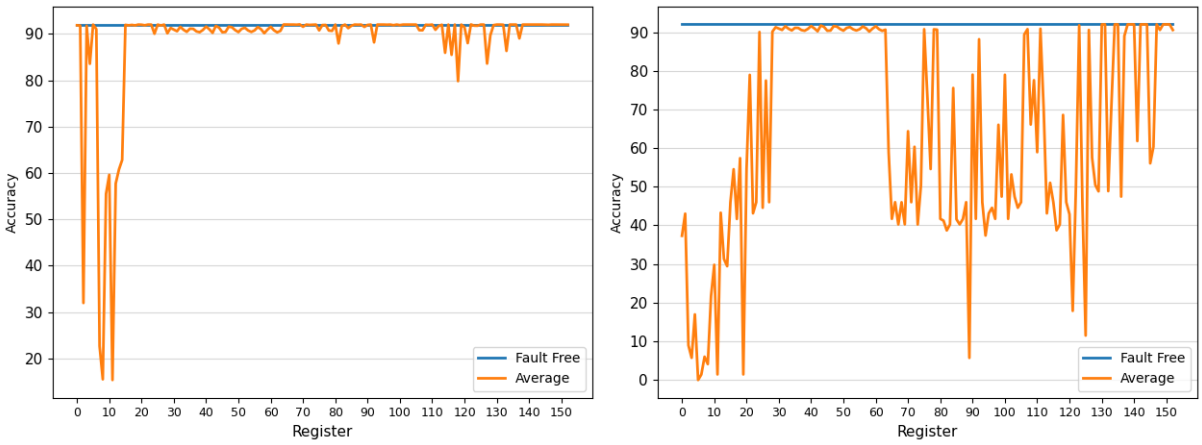


Figure 4.1: Register Files Exhaustive Fault Simulation, Fault Classification (thread 0, SM 0)



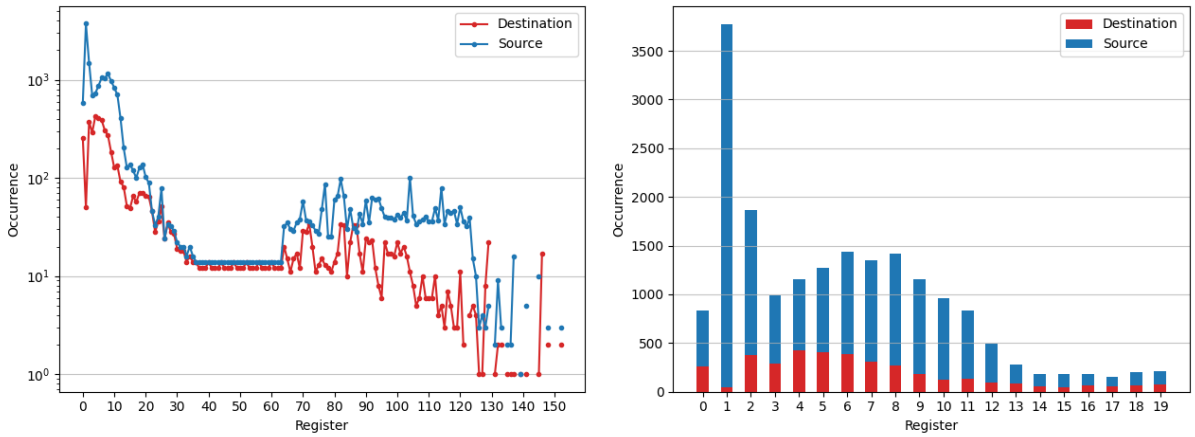
(a) Faults propagated to the Output

(b) All Faults

Figure 4.2: Register Files Exhaustive Fault Simulation, CNN Average Accuracy (thread 0, SM 0)

During the second phase, the strategy adopted to perform RF injections campaigns, was to select just a set of registers to target. The first phase of exhaustive faults simulations suggested that some registers are more critical than others, because they are used more deeply in the code. Using profile techniques, from the source code, it was extracted the registers' utilization parameter, representing the amount of time each register is present in the code, as source operand or destination for a single thread of execution. The Figure 4.3 demonstrates that the register in the range R0-R9 are the most used by

the application, as confirmed by the CNN accuracy degradation. On the other hand, some registers are used rarely; for example, registers in the range R30-R60 are present less than 20 times.



(a) Full view.

(b) Zoom.

Figure 4.3: Register Usage extracted with application profiling

For these motivations, registers R0-R9 were chosen to perform error injection campaigns with a random approach; meaning that for each fault the thread and faulty bit of specific register were randomly selected. A total of 8,000 faults were injected during the simulation, with a total duration of 18 hours. The fault simulations results are reported in the Figure 4.4 and they are synthesized in the Tables 4.5. For each of the registers it has been experienced a lot of crashes responsible for DUE. The most problematic Registers are number 5 since it is characterized by the 100% of DUE faults and 9 which is associated with the largest amount of critical faults equal to 40% of the injected faults. In addition, R2, R7 and R8 experience, in almost the 30% of the cases, critical failures. The Tables 4.4 and 4.6 confirm that many faults generate critical results; removing the faults crushing the application, for R7, R8 and R9 the portion of critical faults exceed 90% of the total injected ones propagating to the CNN output. The Figure 4.5 depicts the accuracy degradation of the CNN associated to this campaign. The two figures show that register 7 and 8 bring to the largest breakdown. The reason, behind these values, is that registers do not experience any SDC-Safe faults, but only critical. R5 is instead characterized by the largest accuracy reduction, considering all the faults type.

Fault Classification			
SDC-Critical	SDC-Safe	Masked	DUE
14.44%	10.38%	7.59%	67.60%

Table 4.3: Register Files Fault Simulation results, Fault Classification

Faults propagated to the Output Classification		
SDC-Critical	SDC-Safe	Masked
44.56%	32.02%	7.59%

Table 4.4: Register Files Fault Simulation results only faults propagated to the Output, Fault Classification

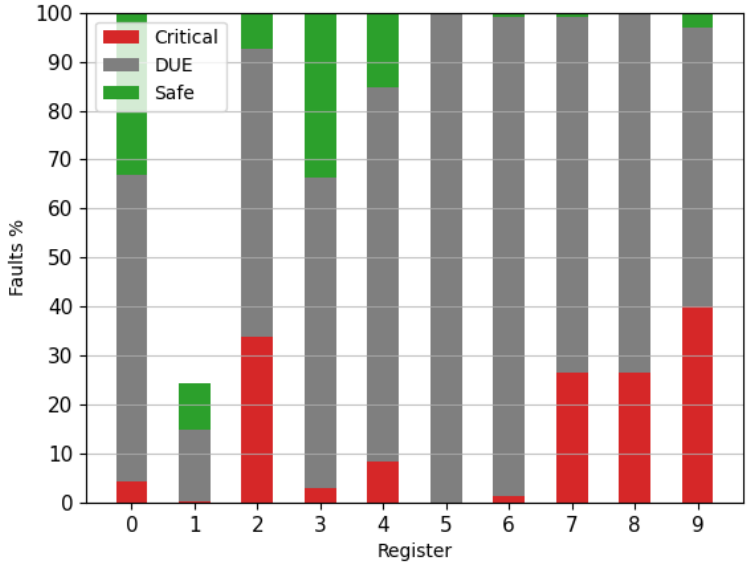
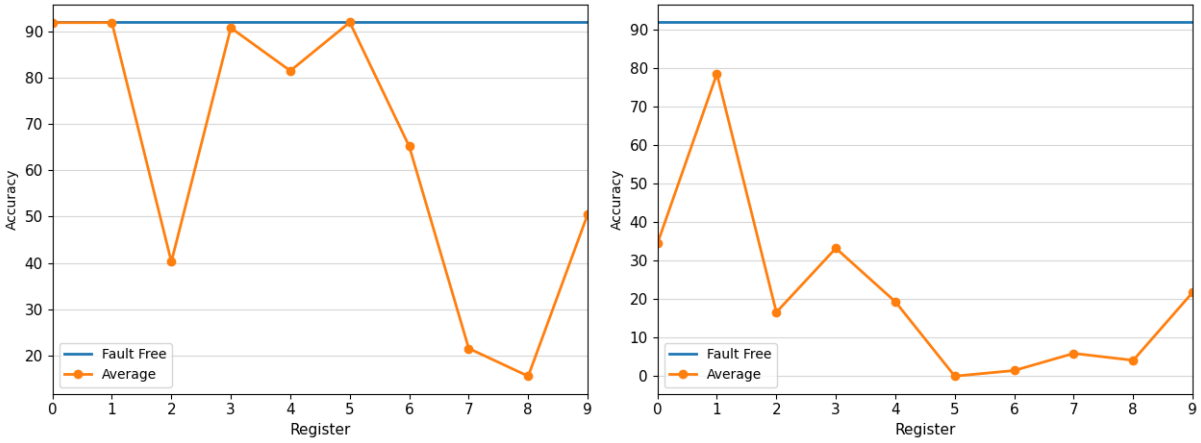


Figure 4.4: Fault Classification of faults in the Register Files



(a) Faults propagated to the Output (b) All Faults

Figure 4.5: Register Files Fault Simulation, CNN average Accuracy

Fault Classification				
Register	SDC-Critical	SDC-Safe	Masked	DUE
0	4.25%	33.25%	0.0%	62.5%
1	0.375%	9.5%	75.625%	14.5%
2	33.875%	7.375%	0.0%	58.75%
3	3.0%	33.375%	0.25%	63.375%
4	8.5%	15.25%	0.0%	76.25%
5	0.0%	0.0%	0.0%	100.0%
6	1.375%	0.875%	0.0%	97.75%
7	26.5%	1.0%	0.0%	72.5%
8	26.5%	0.0%	0.0%	73.5%
9	40.0%	3.125%	0.0%	56.875%

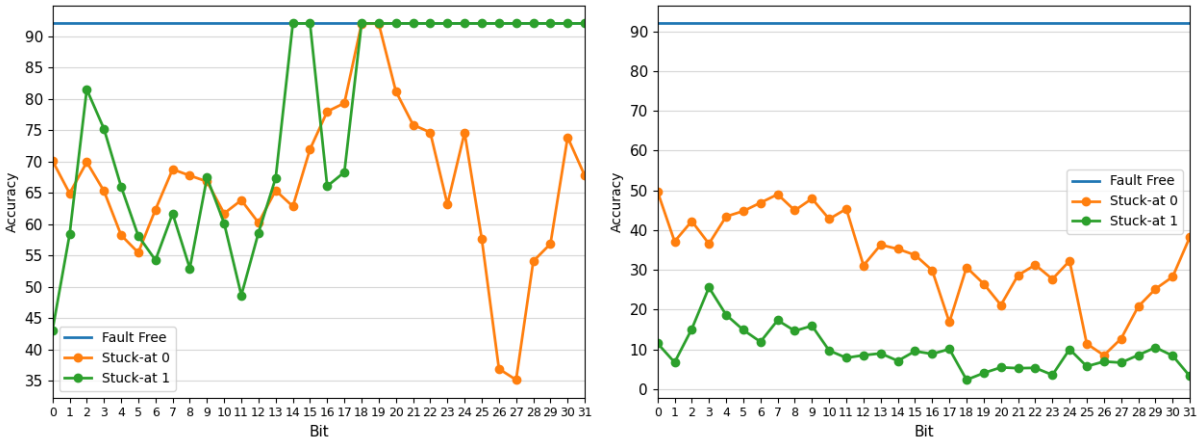
Table 4.5: Register Files Fault Simulation results, Fault Classification

In Figure 4.6 it is reported the CNN accuracy degradation, associated to each bit averaged for the complete set of registers targeted. Some bits appear to be more critical than others. For stuck-at 1 faults the figure depicts that the first half-word is more critical than the last, as the accuracy appears to be nearby the fault free. Whereas, in the first half-word more minimum peaks are present; above all, bit 0 and 11 affect the CNN accuracy with more than a 40% degradation. For what concerns stuck-at 0 faults, a more diversified trend is visible; in that circumstance the most critical bits are 26 and 27 reaching the minimum accuracy value of approximately 35%. Considering instead the accuracy, calculated including application crushing faults, stuck-at 1 faults appear to be

Faults propagated to the Output Classification			
Register	SDC-Critical	SDC-Safe	Masked
0	11.33%	88.67%	0.0%
1	0.44%	11.11%	88.45%
2	82.12%	17.88%	0.0%
3	8.19%	91.12%	0.69%
4	35.79%	64.21%	0.0%
5	0.0%	0.0%	0.0%
6	61.11%	38.89%	0.0%
7	96.36%	3.64%	0.0%
8	100.0%	0.0%	0.0%
9	92.75%	7.25%	0.0%

Table 4.6: Register Files Fault Simulation results only faults propagated to the Output, Fault Classification

more critical than stuck-at 0, with almost a steady trend near 10%.



(a) Faults propagated to the Output

(b) All Faults

Figure 4.6: Register Files Fault Simulation results, per bit CNN average Accuracy

4.2 Functional Units permanent fault simulations

The FU permanent fault simulation was divided in two steps. During the first phase the stuck-at fault model is used to test Functional Units, injecting errors in at input and output interconnections level. In the second part, instead, the bit-flip fault model is adopted to perform the injections campaign; with this model, the attempt done is to

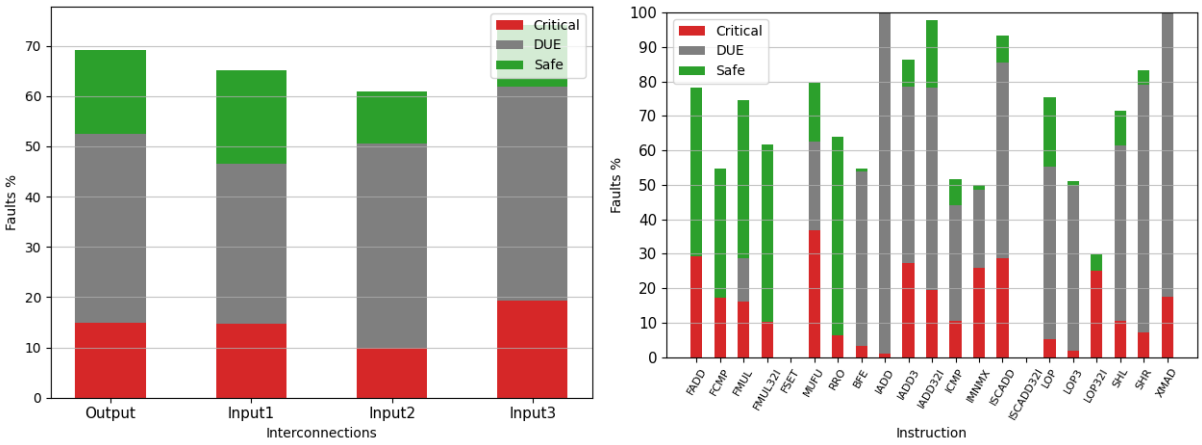
mimic misbehavior internal to the selected FU.

4.2.1 Stuck-at Model

As for the RF case, using the stuck-at fault model, a first exhaustive fault simulation was performed targeting for each present opcode, a single sub-SM and a single core in the same SM. The simulation of the 6,656 faults took 21 hours. In the Table 4.7, it is possible to observe the fault classification results; in that case the results are cumulative of all the FUs' interconnections (input and output). The Figure 4.7b reports, instead the fault classification for each injected opcode instruction.

Fault Classification			
SDC-Critical	SDC-Safe	Masked	DUE
14.09%	15.57%	33.70%	36.64%

Table 4.7: Functional Units Exhaustive Fault Simulation results, Fault Classification



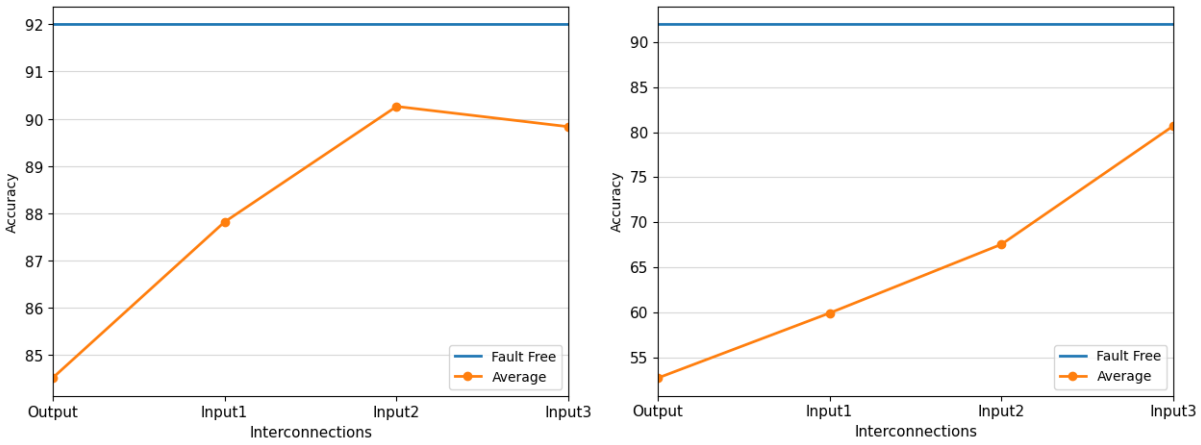
(a) Fault Classification

(b) Instruction Fault Classification

Figure 4.7: Functional Units Exhaustive Fault Simulation results, Fault Classification

The Figure 4.8 depicts the accuracy degradation of the CNN. This exhaustive simulation shows that, even in that case, the CNN exhibits a consistent accuracy reduction; in particular, the functional units output injections lead to the largest breakdown. Considering, all the faults, the degradation, is from 92% of the reference scenario to 54%, whereas taking into account only faults propagating to the output, a 8% reduction is obtained.

Successively, a random fault simulation is performed injecting separately the three functional unit types; randomly choosing for each fault the opcode, a sub-SM and a single



(a) Faults propagated to the Output

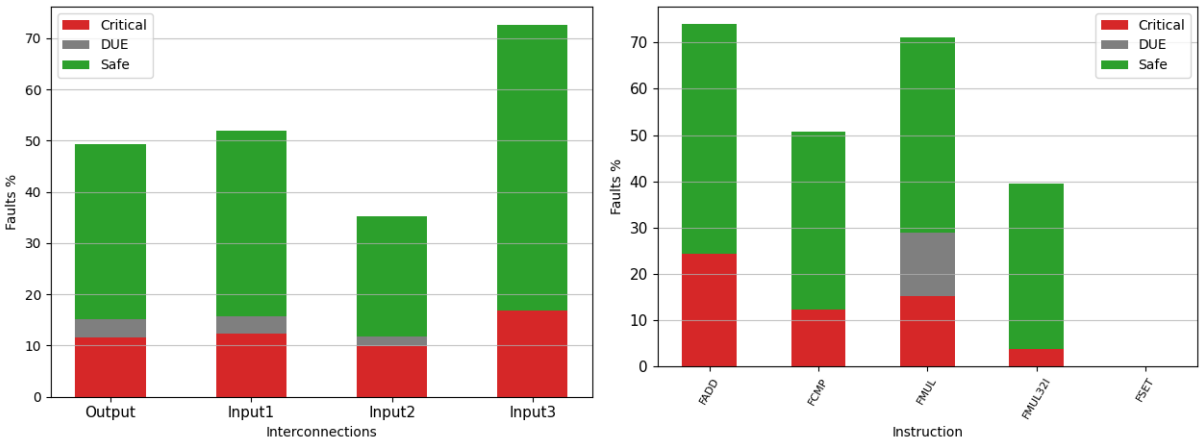
(b) All Faults

Figure 4.8: Functional Units Exhaustive Fault Simulation, CNN average Accuracy

core in the same SM. A total of 10,500 faults were injected and the simulation time was equal to 31 hours.

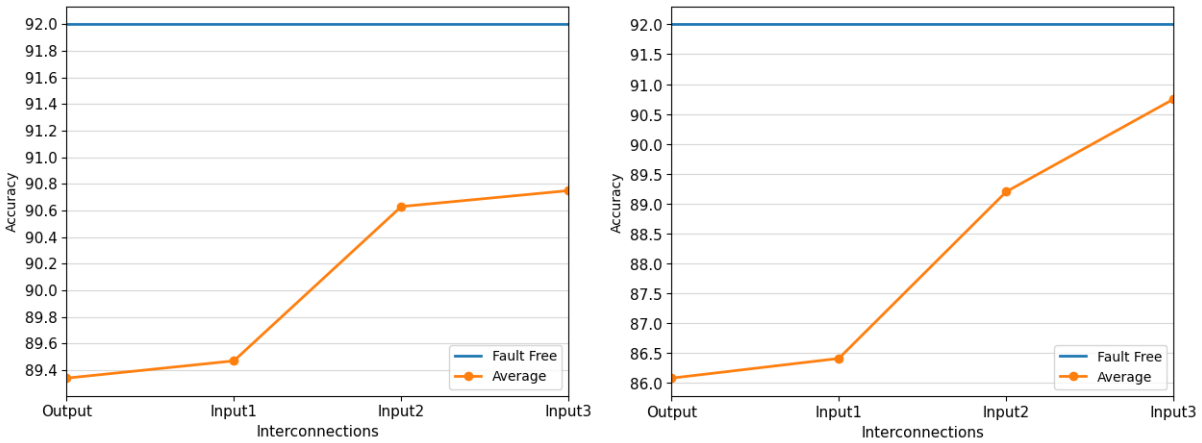
The Floating Point units fault simulation results are presented in the Figure 4.9 and 4.10. Floating Point units do not experience so many DUE faults as in Integer units. With respect to integer case, there are many more faults classified as Masked and SDC-Safe, as expected in all the FU's interconnections. The CNN accuracy does not degrade so much; in all the cases it is near the golden accuracy, for the faults do not crush the application's execution. To understand this result it is needed to study the Figure 4.9b and 4.11. These images prove that, among all the instruction set, the floating point instructions do not lead to many critical failures. This behavior is directly correlated to the running application type. For a CNN, FP operations are meanly used to perform 2D convolutions; meaning that, usually, the faults injection in those FUs lead to image misclassification (SDC-Critical), or variations in inference percentages (SDC-Safe). As it is possible to observe from the Figure 4.11, Stuck-at 0 faults do not generate a real degradation. In addition, it is sometimes experienced that the CNN accuracy increases above the fault free reference value. On the other hand, stuck-at 1 faults generate almost all critical faults, for the instructions affected. The maximum accuracy breakdown is obtained, for faults propagated to the Output case, for FADD and FCMP case, with a 10% reduction; whereas in the other case, FMUL lead to a 20% degradation. It is worth noting that, FSET instruction is affected only by masked faults; whereas the instruction FFMA is not present in the figures since it was impossible to obtain a meaningful result in the time assigned for the fault injection, using NVBit on the board used. Finally, the Figure 4.12 confirms that stuck-at 1 Faults are more critical than 0 and, as it is illustrated Most Significant Bits (MSBs) appear to be more critical; with the lowers peak assumed

for bit 30 in both Figures 4.12a and 4.12b.



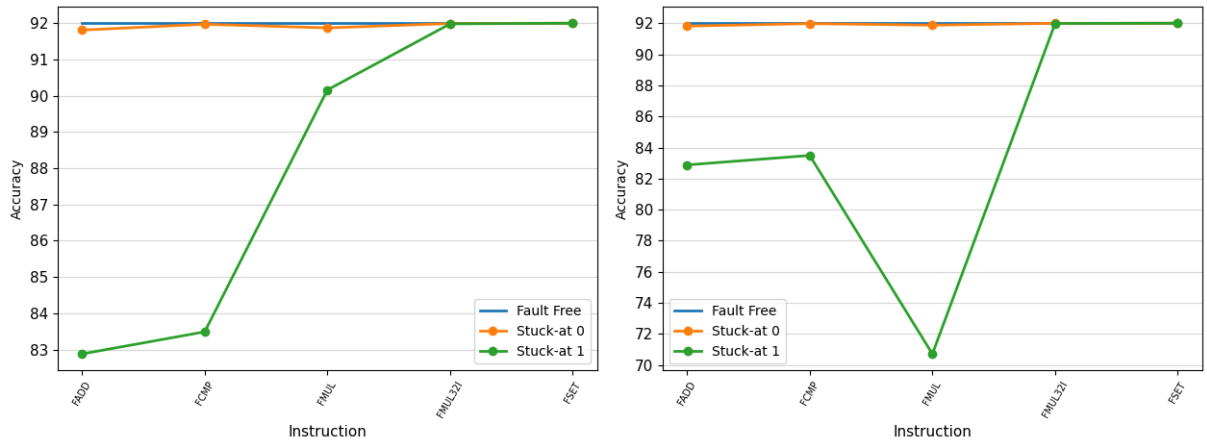
(a) Fault Classification (b) Instruction Fault Classification

Figure 4.9: Floating Point Units Fault Simulation results



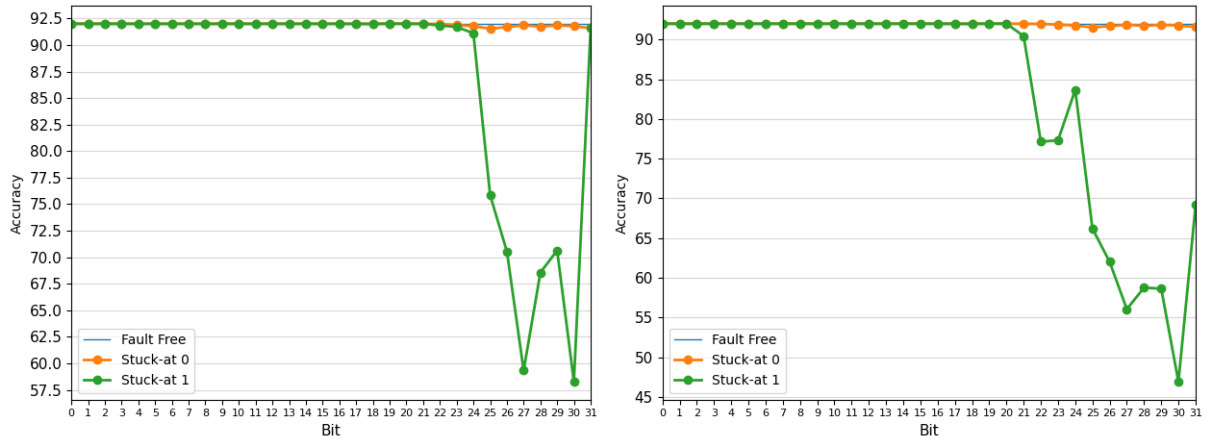
(a) Faults propagated to the Output (b) All Faults

Figure 4.10: Floating Point Units Fault Simulation, CNN average Accuracy



(a) Faults propagated to the Output (b) All Faults

Figure 4.11: Floating Point Units Fault Simulation, per instruction CNN average Accuracy



(a) Faults propagated to the Output (b) All Faults

Figure 4.12: Floating Point Units Fault Simulation, per bit CNN average Accuracy

The SFU fault simulation results are reported in the Figure 4.13 and 4.14. In that case, since the instruction executed take use of only one source operand, it is clear that, interconnections two and three are not subject to fault injections, as it is confirmed by CNN accuracy degradation. The accuracy degrades for both the output interconnections and first input interconnection of 10%, in DUE-free case, and 25 % for the case in which DUE are included. The Tables 4.8 and 4.9 illustrate that, the only two instructions tested, have a very different behavior, when an error is injected. The MUFU instruction is much more critical than the RRO instruction. This is due to the fact that it is deeply used in matrix multiplication kernels and fully connected kernels; whereas the RRO instruction is present in max-pooling layer's kernels. In that case Figure 4.15 shows that the trend for

accuracy degradation per faulty bit injected is variable, with a major impact associated to most significant bits.

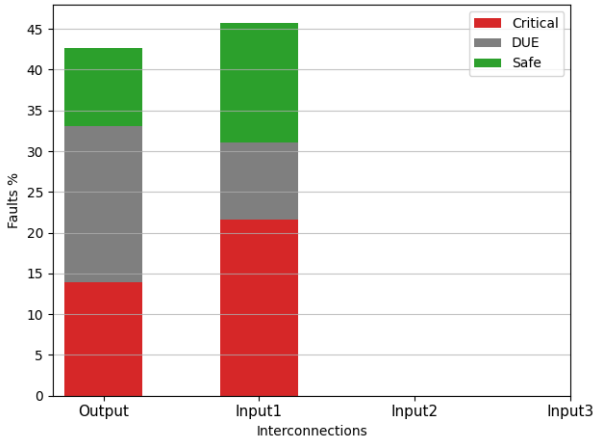
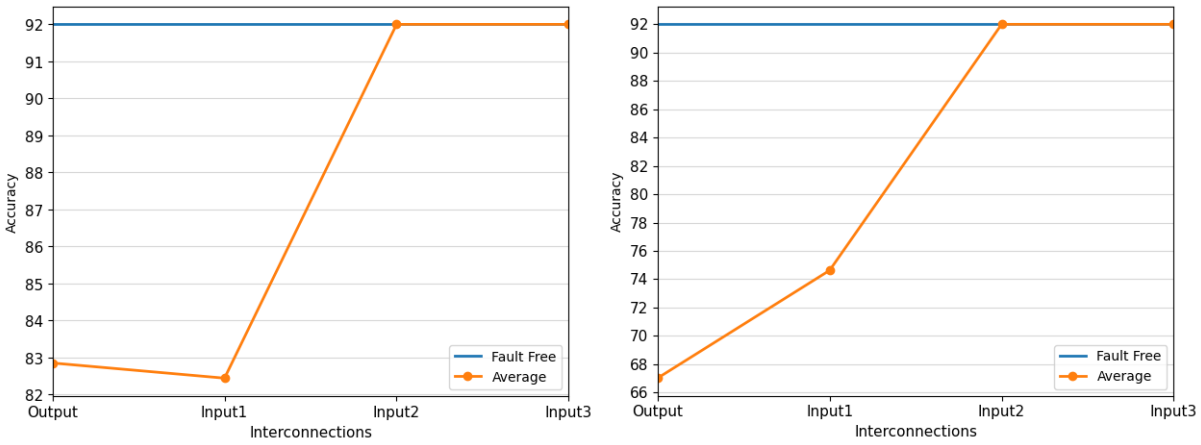


Figure 4.13: Special Function Units Fault Simulation results, Fault Classification

Fault Classification				
Instruction	SDC-Critical	SDC-Safe	Masked	DUE
MUFU	30.2%	13.2%	29.4%	27.2%
RRO	2.6%	10.4%	87.0%	0.00%

Table 4.8: Special Functions Units Fault Simulation results, per instruction Fault Classification



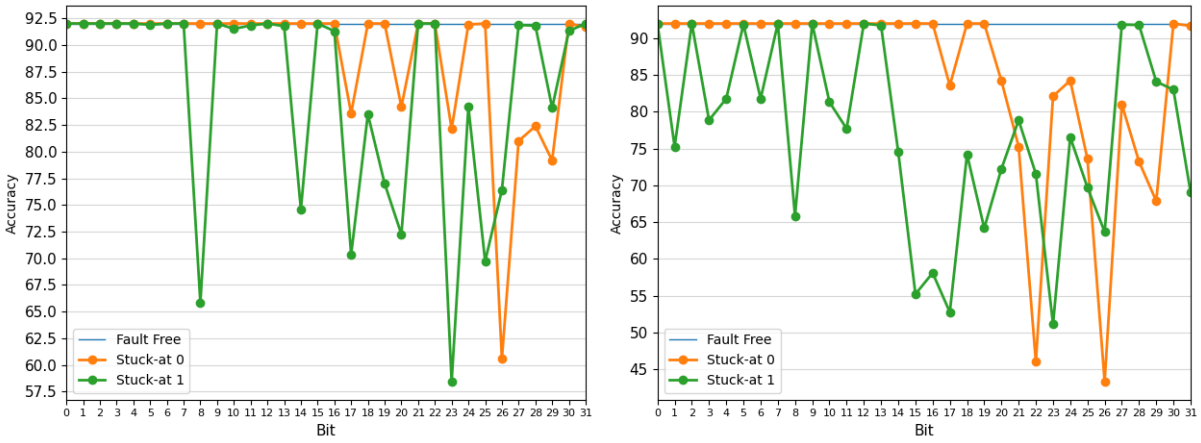
(a) Faults propagated to the Output

(b) All Faults

Figure 4.14: Special Function Units Fault Simulation, CNN average Accuracy

CNN Accuracy: stuck-at 0, stuck-at 1		
Instruction	Faults Propagated to the Output	All faults
MUFU	86.82%, 77.83%	79.63%, 61.44%
RRO	92.00%, 91.95%	92.00%, 91.95%

Table 4.9: Special Functions Units Fault Simulation results: per instruction CNN Accuracy



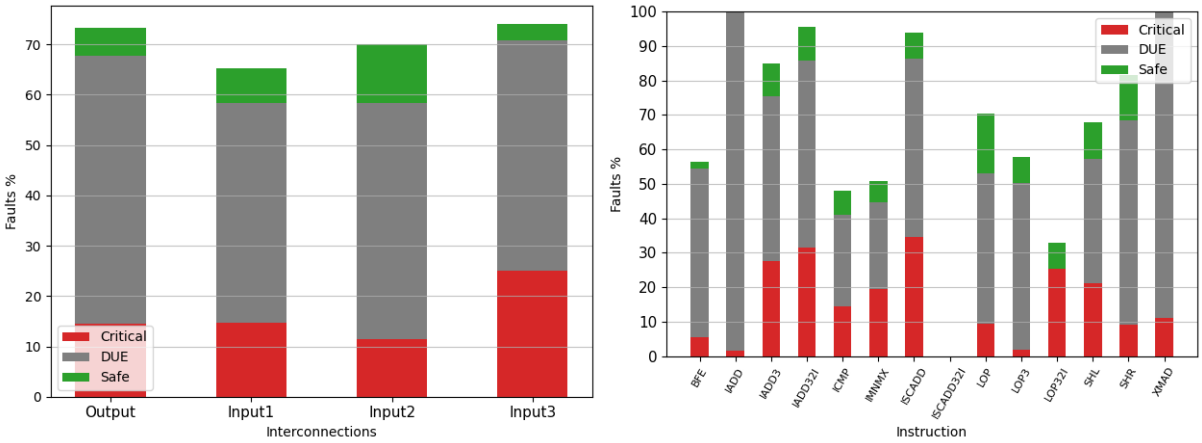
(a) Faults propagated to the Output

(b) All Faults

Figure 4.15: Special Function Units Fault Simulation, per bit CNN average Accuracy

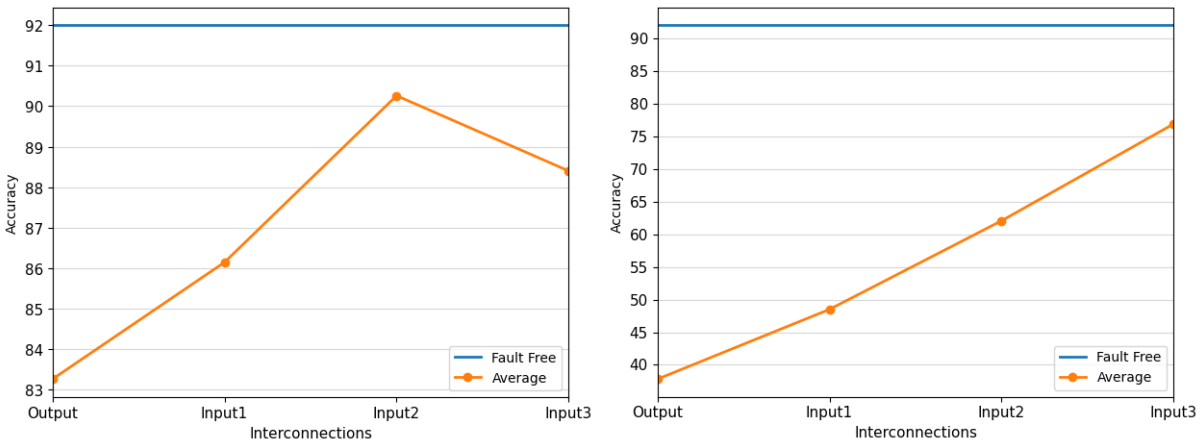
The results of the fault simulations campaign, in Integer units, are shown in the Figure 4.16 and 4.17. As it is illustrated, injections in FU's input and output interconnections generate similar behaviors. With respect to FP and SFU injections, in that scenario, many more faults are classified as DUE. This is confirmed by Figure 4.17, where it is possible to observe that in presence of this kind of fault the impact on the CNN accuracy is dramatic. In general, for faults effect propagating for all application execution, the degradation is higher for the first input of the integer FU, responsible for almost 10% of accuracy reduction. The Figure 4.18 depicts the performance of the CNN accuracy particularized for each instruction opcode tested. Among all, the instruction IADD3, ICMP, ISCADD and XMAD are the most critical if we consider only scenarios in which the fault effect is completely propagated, with the minimum accuracy of 30% reached by the last opcode. Considering all the faults for the accuracy calculation, the impact of unrecoverable faults is huge. For many opcodes, the accuracy is less than 30%, for the stuck-at 1 faults, which are, therefore more critical than stuck-at 0. Integer FU are used to perform operations related to memory access, and the presence of application crush is often connected to memory access violations. This justifies the presence of so

many DUEs. Figure 4.19, reporting the accuracy degradation averaged per bit, informs that stuck-at 1 faults generates unrecoverable errors, affecting the accuracy degradation in the right sub-figure, while stuck-at 0 faults reduce the accuracy in all the other case, as depicted in sub-figure on the left.



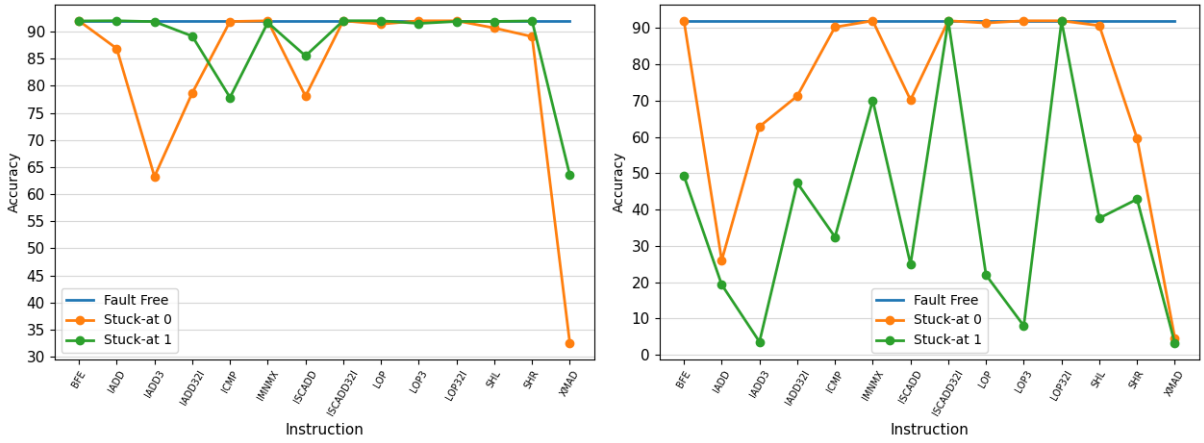
(a) Fault Classification (b) Instruction Fault Classification

Figure 4.16: Integer Units Fault Simulation results



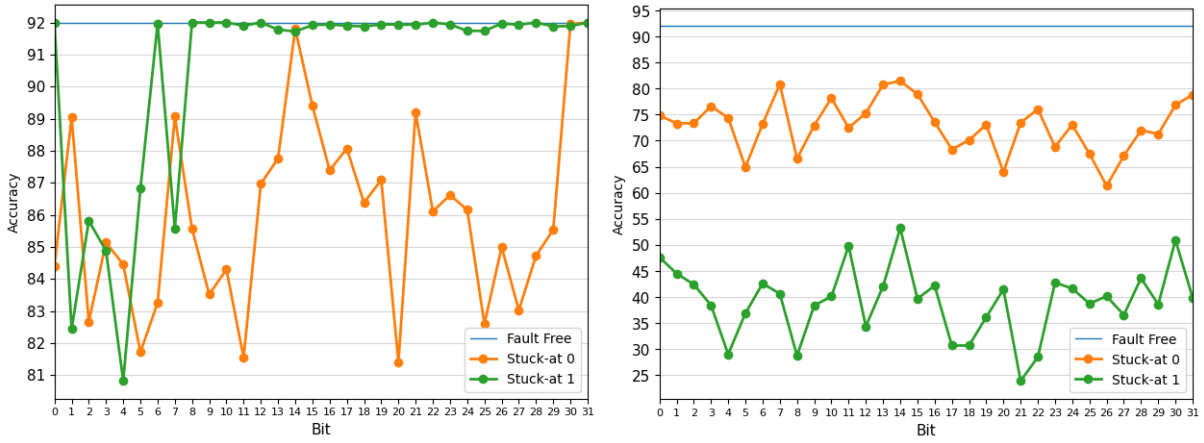
(a) Faults propagated to the Output (b) All Faults

Figure 4.17: Integer Units Fault Simulation, CNN average Accuracy



(a) Faults propagated to the Output (b) All Faults

Figure 4.18: Integer Units Fault Simulation, per instruction CNN average Accuracy



(a) Faults propagated to the Output (b) All Faults

Figure 4.19: Integer Units Fault Simulation, per bit CNN average Accuracy

Lastly, the overall results are illustrated in Figure 4.20, 4.21 and synthesized in the Table 4.10. It is evident that, Floating Point Unit is the less critical among the three functional units; as mentioned, injections in FP units, still allow the CNN code to conclude and generate an inference result, eventually with a different inference percentage. In addition, it is possible to affirm that stuck-at 1 faults generate frequently critical results or application crush, whereas stuck-at 0 impact is present to a lesser extent.

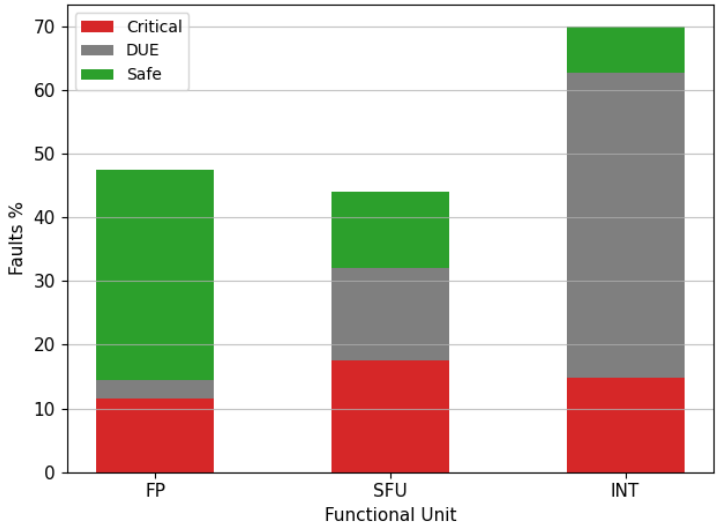
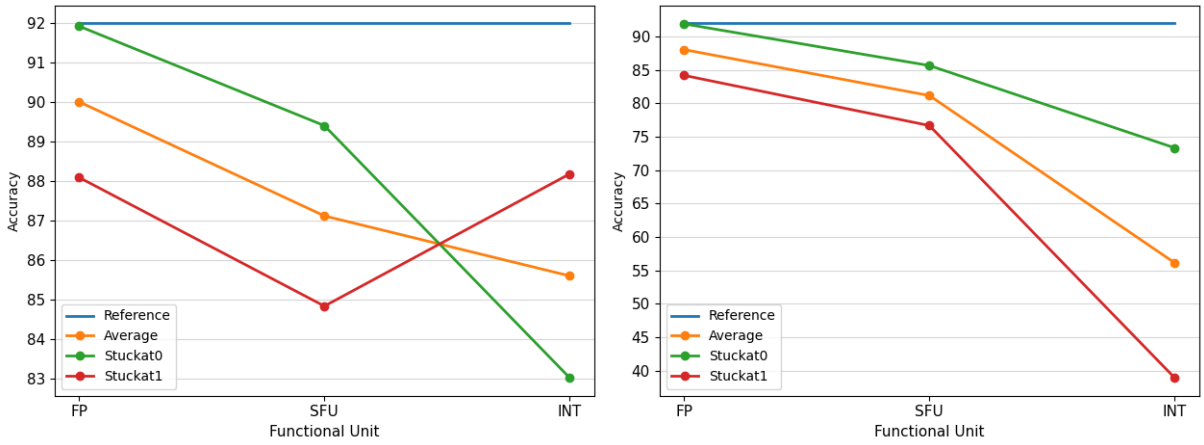


Figure 4.20: Functional Units Fault Classification, Stuck-at model



(a) Faults propagated to the Output

(b) All Faults

Figure 4.21: Functional Units CNN Average Accuracy Degradation, Stuck-at model

Fault Classification				
Functional Unit	SDC-Critical	SDC-Safe	Masked	DUE
FP	11.64%	32.98%	52.52%	2.90%
SFU	17.46%	11.90%	55.95%	14.68%
INT	14.90%	7.18%	30.08%	47.85%

Table 4.10: Functional Units Global Fault Classification, Stuck-at model

4.2.2 Bit-flip Model

The Bit-flip fault model, as anticipated in the previous chapter, is not considered realistic of an hardware fault. The model is tested resorting to the same figure of merit used previously. In that case, a total of 4,200 faults were injected using a random approach, injecting one instruction opcode type at a time, and randomly selecting the sub-part of SM and Core where to perform the injection. For each fault a single bit-flip is applied. The Figure 4.22 and Table 4.11 show the fault classification related to the three functional units tested: Floating Point (FP), Special Function Unit (SFU) and Integer (INT).

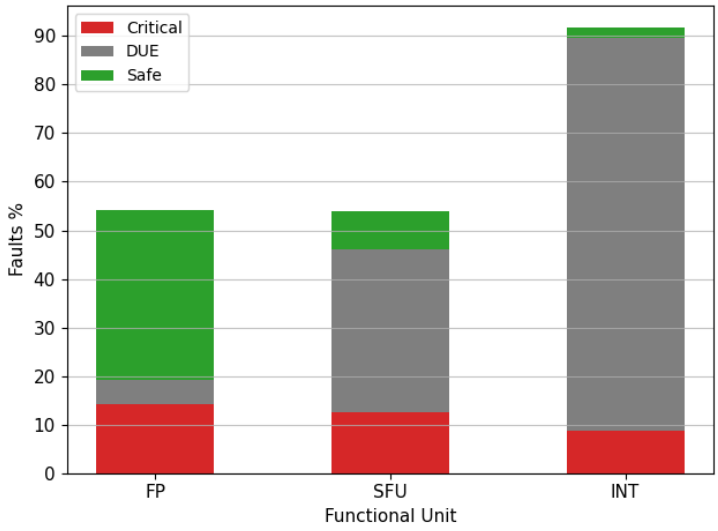


Figure 4.22: Functional Units Fault Classification, Bit-flip model

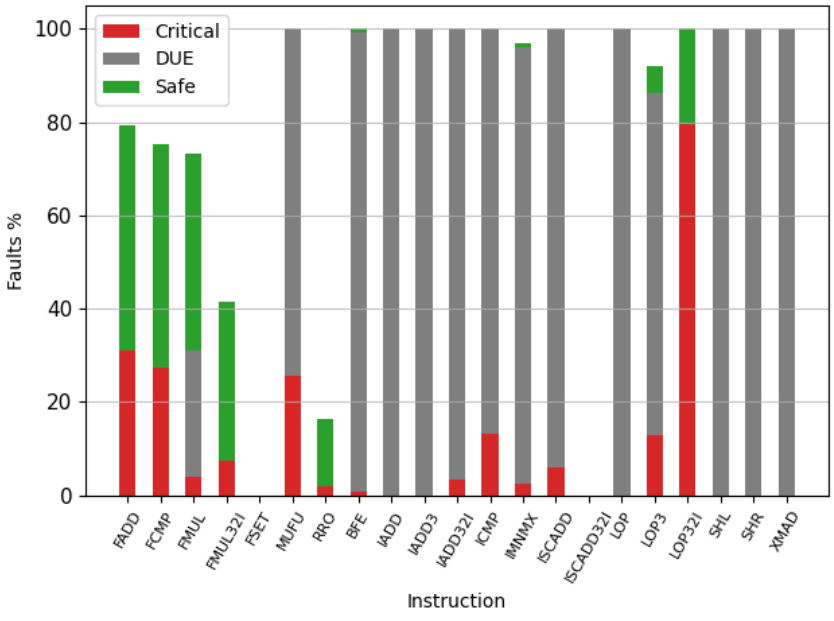
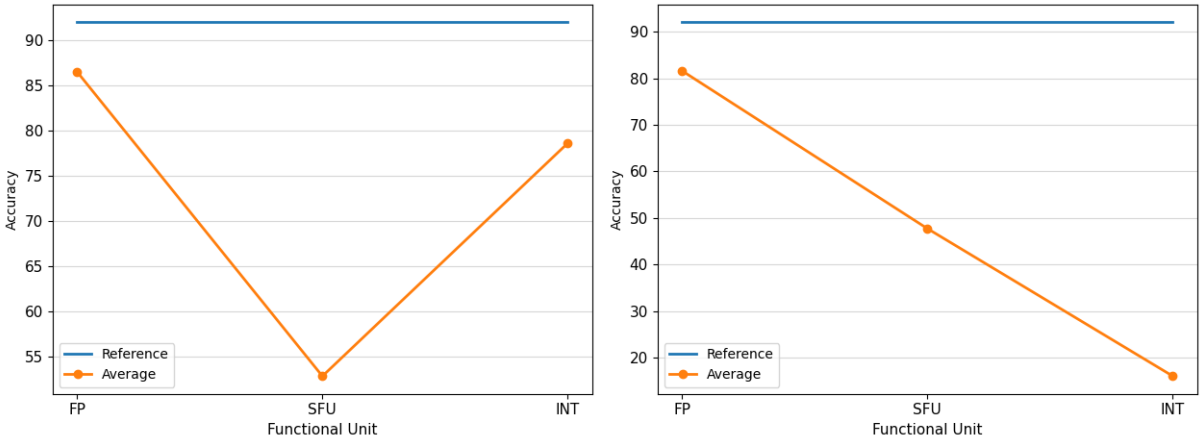


Figure 4.23: Functional Units Fault Classification per instruction, Bit-flip model



(a) Faults propagated to the Output (b) All Faults

Figure 4.24: Functional Units CNN average Accuracy, Bit-flip model

From there, it is possible to observe that the integer unit is subjected to many more DUE faults than SFU and FP, while floating point units are less sensible to DUE. Floating Point Unit is, instead, more prone to experience SDC-Safe and masked faults. For the FP and SFU units the masked faults are more than in the integer unit case. This characteristic of floating point instructions, can be related to the fact that they are really responsible for many CNN operations, in the convolutional kernels. On the other hand, Integer unit faults generate, often DUE failures since they are related to manage memory access and data distribution among threads. The results obtained highlight that this model doesn't allow to test deeply the Integer and SF units, since a lot of DUE are present. Instead, the results associated to the FP units are pretty similar for the two fault models.

Fault Classification				
Functional Unit	SDC-Critical	SDC-Safe	Masked	DUE
FP	13.98%	34.64%	45.39%	5.99%
SFU	12.50%	8.00%	46.00%	33.50%
INT	8.27%	1.93%	7.79%	82.01%

Table 4.11: Functional Units Global Fault Classification, Bit-flip model

CHAPTER 5

Conclusions

In this thesis work, the impact of permanent faults in a GPGPU (General Purpose Graphics Processing Unit) is evaluated. This project represents something new in the academic world, since the other researches mostly focused on transient faults. A Permanent Fault simulator framework was setup to perform the fault injection campaigns. The Framework, based on NVBitFI functionalities, was developed to support errors injections in two different locations inside the GPU. These are the Register File and the Functional Units. The fault simulator is tied to a global controller, able to automate the process of application profiling, generation of the faults list, fault injection and classification. The framework is used to assess the reliability of the LeNet CNN model, available in the Darknet environment. The test_classifier function of Darknet was modified in order to enable the inference of 100 images concurrently. The simulations were divided in two main steps. The Register File was tested first for evaluating the impact of single bit stuck-at faults, starting from an exhaustive simulation and then concentrating the random injections in the first ten registers (R0-R9), which are the most used by the application. A not negligible percentage of faults produced Silent Data Corruption (SDC), demonstrating the impact of hard faults on the GPU computations; these errors are responsible for a substantial accuracy degradation of the CNN.

Subsequently, the Functional Units were tested, adopting again the single bit stuck-at fault model. That approach is considered more realistic compared to the bit-flip model proposed by NVBitFI. With the stuck-at model, it has been proven that faults effecting Special Function and Integer units generate CNN misclassifications (Critical SDC). Whereas, faults injected in Floating Point units allow the CNN to produce the correct prediction results with percentage of inference different from the one obtained in the fault-free case (Safe SDC). The overall results rely on the specific application running on the GPU. Floating Point operations are less critical than the others; in that case, injection and error propagation still allow the CNN application to conclude the execution pro-

ducing a prediction result, eventually with a different inference percentage. In that case too, stuck-at 1 faults generated more frequently unrecoverable failures than stuck-at 0. Adopting then the bit-flip model, functional units were tested again. In that case, many fault injections crushed the application's execution, specifically this happened with errors' insertion in Integer and Special Function units, confirming that the stuck-at model allows a clearer observation of internal misbehavior. On the contrary, for Floating Point units injections the results obtained were similar to those retrieved with the stuck-at model.

Future activities aim to extend the set of locations where to inject permanent faults. In addition, it could be possible to evaluate more CNN architecture models. To improve the reliability of this type of computational system, hardening techniques could be proposed to counteract the vulnerabilities caused by permanent faults in CNN architectures. Finally, future works could be focused on the development of design strategy to improve GPU resilience to errors.

References

[1] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[2] J. Redmon, "Darknet: Open source neural networks in c." `http://pjreddie.com/darknet/`, 2013–2016.

[3] Y. LeCun, C. Cortes, and C. Burges, "The MNIST database of handwritten digits." `http://yann.lecun.com/exdb/mnist/`, 1998.

[4] J. L. Hennessy and D. A. Patterson, *Computer Architecture A Quantitative Approach*. United States of America: The Morgan Kaufmann Series in Computer Architecture and Design, 2017.

[5] P. Glaskowsky, "Nvidia's fermi: the first complete gpu computing architecture," *NVIDIA*, vol. 18, 2009.

[6] "CUDA C++ Programming Guide, CUDA Toolkit v11.6.1." `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`, 2022. [Accessed: 2022-03-02].

[7] S. Cook, *CUDA Programming A Developer's Guide to Parallel Computing with GPUs*. United States of America: Morgan Kaufmann, 2013.

[8] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, 2008.

[9] NVIDIA, "Parallel thread execution isa version 7.6." `https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#goals-of-ptx`, 2022.

[10] NVIDIA, "About cuda." https://developer.nvidia.com/about-cuda, 2021.

[11] J. E. R. Condia, B. Du, M. Sonza Reorda, and L. Sterpone, "Flexgripplus: An improved gpgpu model to support reliability analysis," *Microelectronics Reliability*, vol. 109, p. 113660, 2020.

[12] "Nvidia geforce gtx 980 whitepaper, featuring maxwell, the most advanced gpu ever made," *NVIDIA*, 2014.

[13] "Nvidia ampere ga102 gpu architecture whitepaper, second-generation rtx," *NVIDIA*, 2021.

[14] T. Mitchell, *Machine Learning*. McGraw-Hill Education, 1997.

[15] C. C. Aggarwal, *Neural Networks and Deep Learning*. Springer, Cham, 2018.

[16] S. Skansi, *Introduction to Deep Learning, From Logical Calculus to Artificial Intelligence*. Undergraduate Topics in Computer Science, Springer, Cham, 2018.

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

[18] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1–9, 2015.

[19] K. Adam, I. I. Mohd, and Y. Ibrahim, "Analyzing the resilience of convolutional neural networks implemented on gpus: Alexnet as a case study," *International Journal of Electrical and Computer Engineering Systems*, 2021-06-21.

[20] K. Adam, I. Mohamed, and Y. Ibrahim, "Analyzing the soft error reliability of convolutional neural networks on graphics processing unit," *Journal of Physics: Conference Series*, vol. 1933, p. 012045, 06 2021.

[21] S. K. S. Hari, T. Tsai, M. Stephenson, S. W. Keckler, and J. 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)*, pp. 249–258, 2017.

[22] J. Wei, Y. Ibrahim, S. Qian, H. Wang, G. Liu, Q. Yu, R. Qian, and J. Shi, "Analyzing the impact of soft errors in vgg networks implemented on gpus," *Microelectronics Reliability*, vol. 110, p. 113648, 2020.

[23] Y. Ibrahim, H. Wang, and K. Adam, "Analyzing the reliability of convolutional neural networks on gpus: Googlenet as a case study," in *2020 International Conference on Computing and Information Technology (ICCIT-1441)*, pp. 1–6, 2020.

[24] Y. Ibrahim, H. Wang, M. Bai, Z. Liu, J. Wang, Z. Yang, and Z. Chen, "Soft error resilience of deep residual networks for object recognition," *IEEE Access*, vol. 8, pp. 19490–19503, 2020.

[25] F. dos Santos and P. Rech, "Analyzing the criticality of transient faults-induced sdcs on gpu applications," pp. 1–7, 11 2017.

[26] "NVIDIA CUDA Driver APIs." `https://docs.nvidia.com/cuda/cuda-driver-api/index.html`, 2022. [Accessed: 2022-03-05].

[27] O. Villa, M. Stephenson, D. Nellans, and W. S. Keckler, "Nvbit: A dynamic binary instrumentation framework for nvidia gpus," *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 372–383, 2019.

[28] T. Tsai, S. K. S. Hari, M. Sullivan, O. Villa, and S. W. Keckler, "Nvbitfi: An architecture-level fault injection tool for gpu application resilience evaluations." `https://github.com/NVlabs/nvbitfi`, 2020.

[29] T. Tsai, S. K. S. Hari, M. Sullivan, O. Villa, and S. W. Keckler, "Nvbitfi: Dynamic fault injection for gpus," in *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 284–291, 2021.

[30] J. L. Boulanger, "3 - principle of dependability," in *Certifiable Software Applications 1* (J.-L. Boulanger, ed.), pp. 43–64, Elsevier, 2016.

[31] "CUDA Binary Utilities v11.6.1." `https://docs.nvidia.com/cuda/cuda-binary-utilities/index.html#nvdisasm`, 2022. [Accessed: 2022-03-06].