POLITECNICO DI TORINO

Master's degree in
Computer Engineering

Master's degree thesis

Reliability Assessment and Software-Based Hardening of a Hyperspectral Image Classifier for GPUs

Supervisors
Josie Esteban Rodriguez Condia
Matteo Sonza Reorda
Juan David Guerrero Balaguera

Candidate
Sergiu-Mohamed Abed

December 2024

Abstract

Over the last few years, we have seen Artificial Intelligence being adopted across many domains and sectors. One such domain is Edge Computing, where Internet of Things (IoT) devices are now being designed to handle neural networks to provide computation close to the source of data in order to reduce latency and throughput on the network. Unfortunately, modern applications of Artificial Intelligence (e.g., hyperspectal image classification in domains such as precision agriculture) face reliability challenges, which are frequently encountered in embedded systems being deployed in uncontrolled and rough environments. In particular, many methods have been studied to assess the reliability of neural network applications. However, more research is still needed to understand the effects of faults in the underlying hardware used for execution of neural networks.

The focus of this thesis work is to study the impact of transient faults affecting the hardware of the device on the performance of a Hyperspectral Image Classifier. To do so, a tool developed by NVIDIA called NVBitFI has been adapted and used to evaluate the effects of corruption and its main impacts by injecting faults at the instruction-set level. After a large amount of simulations performed under different configurations of the classifier, it was possible to achieve some important results: the amount of outcomes that led to changes in the output without affecting the classification (15.89%), the amount of outcomes that led to changes in the output and altered the classification (24.89%) and crashes (or hangs) of the model (11.47%). In addition, the experiments allow the identification of the most sensitive parts (e.g., code blocks) of the classifier, i.e., the parts that, when subjected to faults, contributed to the majority of the changes in the behavior of the model.

Lastly, the experimental results and the identification of those vulnerable parts are employed to develop a software-based hardening technique applied to the critical parts of the classifier to mitigate the effects of transient faults, hence increasing its fault tolerance.

In future work, further analysis similar to this work should be performed on different kinds of applications making use of the same libraries used by the hyperspectral image classifier studied here (i.e., cuBLAS and PyTorch) to understand if similar trends can be noticed across different scenarios and so, potentially improve the fault tolerance of the particularly sensitive functions.

Summary

Thesis Objectives

The recent advances in Artificial Intelligence (AI) made it possible for its adoption across many domains, one such domain being Edge Computing. With the deployment of AI models on low-power and high-performance devices situated in rough and uncontrolled environments, reliability has become an important aspect when designing AI applications for critical domains where a failure can lead to loss of valuable resources or disasters. The goal of this thesis work is the evaluation of the reliability of a Hyperspectral Image (HSI) classifier considering transient fault effects arising from the underlying hardware architecture of two GPUs. The classifier is then modified by applying a software-based hardening technique in order to increase its fault tolerance.

Methodology

The HSI classifier tested in this work consists of a pre-processing step implemented by means of an iterative version of the Principal Component Analysis (PCA) dimensionality reduction method and an inference step represented by a 3D Convolutional Neural Network (3D CNN).

To simulate faults, this work made use of a software-based fault injection tool developed by NVIDIA called NVBitFI that simulates transient faults in the hardware by corrupting the destination register of assembly code instructions. The tool is able to corrupt a program on-the-fly, i.e., during the execution it intercepts the instruction (opcode) targeted and applies a bit-flip model on the result. NVBitFI consists of two main tools: a profiler and an injector. The profiler generates a file called profile containing information about the kernels and opcodes used during the execution of the program. The injector, as the name suggests, injects faults in the destination register. NVBitFI categorizes the outcomes of faults in three different ways: Masked (the fault did not propagate to the output of the classifier), SDC (Silent Data Corruption, where the fault affected the output) and DUE (Detected Unrecoverable Error, where the fault caused a hang or a crash of the application). For this analysis, SDC category was further divided in two disjoint ones: SDC-safe, where the output of the classifier was changed but the classification outcome is the same as the non-faulty one, and SDC-critical, where also the classification is different. The main focus of the analysis conducted is on SDC-critical.

To evaluate the reliability of the HSI classifier, this work made use of two metrics: Program Vulnerability Factor (PVF) and Mean Execution Between Failures (MEBF). PVF represents the frequency of SDC-critical outcomes among the total number of fault injections performed (lower is better). MEBF quantifies the number of times the application was executed correctly between two critical SDCs (higher is better).

The software-based hardening technique used is a method based on Duplication with Comparison (DwC), where the sensitive part of the application is executed twice and if the two results differ, the same code block is executed a third time and the third result is considered.

Experimental configurations

The simulations are performed on the HSI classifier during the inference operation on three different datasets: Indian Pines, Salinas and Pavia University. The three datasets are hyperspectral images where each pixel corresponds to a certain surface area and is described by intensities of various wavelengths in the electromagnetic spectrum.

To understand the inherent fault tolerance of the PCA step, this work tested three configurations: PCA 7, PCA 10 and PCA 50, i.e., the number of principal components used for reducing the dimension of the datasets are 7, 10 and 50, respectively.

The two GPUs on which the experiments were conducted are NVIDIA GeForce GTX 1050 (low-end) and NVIDIA GeForce RTX 3060TI (mid-range).

Results

Based on the profile, it was possible to identify that some kernels and opcodes are more frequently used than other. For example, *FFMA*, *XMAD*, *IADD*, *LDG* and so on are the most frequently executed opcodes, and *computeBOffsetsKernel*, *elementwise* and *gemv2N* are among the most used kernels.

The evaluation of the fault injection campaigns showed several trends across different configurations. Figure 5.7 shows that when moving from a lower to a higher number of principal components, one can notice that the amount of SDC-critical outcomes increases, indicating that the pre-processing step implemented with PCA could be the most sensitive part of the application to faults. This fact is then proven when looking at the number of SDC-critical caused by each static kernel. The results showed that the vast majority of corrupted classifications were caused by just a handful of the static kernels, namely *gemv2N*, *gemv2T*, *enable_if* and *gemvNSP*, all belonging to the PCA part of the classifier.

Another trend discovered is that the sensitivity to transient faults of the inference operation also depends on the input data. On the dataset Indian Pines, which is the smallest among the ones tested, the classifier experiences a larger amount of SDC-critical compared to on the other two. On the other hand, Pavia University (the largest one) sees fewer SDC-critical. When looking at the profile corresponding to the inference on each dataset, the number of kernels of the CNN part is larger when the size of the dataset is larger. This pattern also indicates that the pre-processing stage of the classifier is the most sensitive one.

Figure 5.8 shows the impact of considering the execution time in the analysis of reliability. In figure 5.7 we see a high discrepancy between the PVFs of inference on Indian Pines and the other datasets, whereas figure 5.8(a) shows that MEBF values across the three datasets have values that don't differ significantly. However, when considering only the execution time of the PCA stage (which is almost the same for all three datasets), as shown in figure 5.8(b), the results once again show a high difference, with Indian Pines showing lower MEBF.

Finally, the hardening technique based on DwC applied on PCA increased the fault tolerance of the HSI classifier for most of the experimental settings (four out of six) as seen in table 5.4, where one can see a drop of SDC-critical up to 63%. For one of the settings, hardening did not have any effect, while for another, hardening actually caused an increase of SDC-critical by 10%.

Acknowledgements

My deepest thanks go to my supervisors who provided me with guidance and invaluable insights throughout the work carried out in this Master's degree thesis. Their great passion for the domain in the context of this work played a crucial role in my decision to pursue a career in the same field.

I'm extremely grateful to my family for their love, care and support. Special thanks go to my mother, father and grandmother for their hard work and invaluable support without which my academic journey would not have been possible.

Last but not least, I would like to thank all the friends I've made since moving to Italy for all the good times, for their help and for enriching my general knowledge with stories and facts about their background and culture.

Contents

Thesis Objectives . . . 2
Methodology . . . 2
Experimental configurations . . . 3
Results . . . 3

List of Tables 8

List of Figures 9

1 Introduction 11

2 Background 13
2.1 Hyperspectral Imaging (HSI) . . . 13
2.1.1 Hyperspectral Image Classification . . . 14
2.2 GPU programming model . . . 14
2.2.1 Kernel . . . 14
2.2.2 Thread hierarchy . . . 15
2.2.3 Key terminologies . . . 16
2.3 Reliability assessment and estimation . . . 16
2.3.1 Permanent and transient faults . . . 16
2.3.2 Experimental-based reliability assessment strategies . . . 17
2.4 Software-based injections and tools for edge computing and GPUs (NVBitFI) 19
2.4.1 NVBitFI software architecture . . . 20
2.4.2 (Transient) Fault injection procedure . . . 20
2.4.3 Instruction groups targeted . . . 21
2.4.4 Fault outcome categorization . . . 22
2.5 Reliability metrics . . . 22
2.5.1 Program Vulnerability Factor (PVF) . . . 23
2.5.2 Mean Execution Between Failure (MEBF) . . . 23

3 Study case: Hyperspectral Image Classification 25
3.1 Pipeline . . . 25
3.1.1 Pre-processing . . . 26
3.1.2 Inference . . . 26
3.2 Datasets . . . 27

3.2.1 Indian Pines . . . 27
3.2.2 Pavia University . . . 28
3.2.3 Salinas . . . 29

4 Fault injection framework adaptation (NVBitFI) 31
4.1 NVBit and NVBitFI installation . . . 31
4.2 Directory structure and framework configuration . . . 31
4.2.1 nvbitfi/injector &nvbitfi/profiler . . . 32
4.2.2 nvbitfi/scripts/params.py . . . 33
4.2.3 nvbitfi/test-apps/PCAHyperspectralClassifier . . . 34
4.2.4 nvbitfi/test_pca.sh . . . 36
4.2.5 nvbitfi/logs . . . 37
4.3 Fault and Error Classification . . . 37
4.4 Workflow of reliability assessment and software-based hardening . . . 38

5 Experimental results 41
5.1 Setting of experiments . . . 41
5.1.1 GPUs . . . 41
5.1.2 Datasets . . . 41
5.1.3 PCA configurations . . . 42
5.1.4 Targeted instruction groups . . . 42
5.2 Profile analysis . . . 42
5.3 Structural analysis of HSI classifier under transient faults . . . 46
5.3.1 SDC-safe and SDC-critical study at opcode level . . . 46
5.3.2 SDC-safe analysis at kernel level . . . 48
5.3.3 Program Vulnerability Factor (PVF) . . . 49
5.3.4 Mean Execution Between Failure (MEBF) . . . 49
5.3.5 Classification corruption examples . . . 52
5.3.6 Effects of transient faults on different PCA configurations . . . 52
5.3.7 Identification of SDC-critical prone kernels . . . 55
5.4 Hardening . . . 56

6 Conclusions 61

List of Tables

5.1 Percentages of PCA and CNN dynamic kernels 42
5.2 Experimental configurations of fault injections campaigns 46
5.3 Results before and after hardening for PCA 10 on RTX3060TI 57
5.4 Hardening results for PCA 10 on RTX3060TI 58

List of Figures

2.1 2D example of thread hierarchy. Source here . . . 15
3.1 Li et al. HSI classifier comprising image pre-processing (PCA) and AI algorithms (CNN). . . . 25
3.2 3D CNN classifier workflow . . . 27
3.3 Indian Pines . . . 28
3.4 Pavia University . . . 28
3.5 Salinas . . . 29
4.1 NVBitFI directory structure . . . 32
4.2 Reliability assessment and software-based hardening workflow . . . 38
5.1 Instruction groups executions distribution. Dataset: Salinas . . . 43
5.2 Dynamic opcodes per instruction group. Dataset: Salinas . . . 44
5.3 Kernels executions. Dataset: Salinas . . . 45
5.4 Opcodes executions per kernel for two of the most representative kernels. Dataset: Salinas . . . 47
5.5 Opcodes targeted that led to SDCs. Dataset: Indian Pines . . . 48
5.6 Kernels targeted that led to SDC-safe. Dataset: Indian Pines . . . 49
5.7 Relative PVF . . . 50
5.8 MEBF on (a) complete application; (b) PCA part only . . . 51
5.9 Example Indian Pines faulty outcome (accuracy drop: 62.58%) . . . 53
5.10 Example of Salinas faulty outcome (accuracy drop: 59.30%) . . . 54
5.11 Example of Pavia University faulty outcome (accuracy drop: 6.90%) . . . 55
5.12 SDC-critical distribution over kernels for Indian Pines. Each plot depicts two different distributions: one for register files and one for functional units 56
5.13 SDC-critical distribution over kernels. Each plot depicts two different distributions: one for register files and one for functional units . . . 57
5.14 cuBLAS implementation of PCA . . . 58

Reliability is the precondition for trust.
[Wolfgang Schäuble]

Chapter 1

Introduction

The development and integration of computationally complex applications (e.g., Artificial Intelligence) on embedded/edge computing systems is boosted by modern advances in computer architecture and power-efficient platforms, including Graphics Processing Units (GPUs) and specialized AI-accelerators (Dally [2023]). This availability of flexible and computationally powerful platforms allows the massive adoption of edge computing in several domains, including precision agriculture, healthcare, autonomous robotics, surveillance, and environmental monitoring, to distribute computational costs and process information under low-power consumption restrictions (i.e., General Purpose Computing on Edge Nodes for distributed and High-Performance Computing domains, Varghese et al. [2016]). In particular, embedded computer vision and image processing applications (e.g., Hyperspectral Imaging or HSI, De Lucia et al. [2022]) are boosted by AI algorithms that also speed up their execution while providing effective results (e.g., classification/detection).

In safety-/mission-critical domains, the reliability and integrity of AI-powered edge systems are crucial to guarantee their correct operation, especially in the presence of failures. For this purpose, experimental evaluations characterize and identify the vulnerable and sensitive (hardware/software) parts inside a system. The results are later used for design improvements or to develop fault countermeasure mechanisms (Rech [2024]). Unfortunately, the increasing computational complexity, the large amount of data of edge applications, and the transistor density of new platforms impose challenges and impede the straightforward adoption of reliability assessment strategies (Meuser et al. [2024]).

In literature, some works focused their assessment and evaluations on the application (e.g., AI model architecture) and aimed to improve the application description against vulnerable or fault-sensitive parts in the code (e.g., layers). Unfortunately, this level of abstraction can hardly identify and accurately represent corruption sources from the underlying hardware. Thus, fine-grain (hardware-aware) evaluations are mostly neglected (Li et al. [2017b], Sabbagh et al. [2019], Goldstein et al. [2020], Ruospo et al. [2021]). In contrast, other works Santos et al. [2019, 2021a], Condia et al. [2022], Santos et al. [2023] mainly evaluate the system's hardware micro-architecture (e.g., by resorting to RT-/Gate-level description of the hardware accelerators or multi-processors) running ML workloads. In these cases, the analysis identifies the vulnerable hardware structures impacting the

workload and how they propagate for later improvement by introducing hardware/software mitigation mechanisms. In Condia et al. [2022], the authors evaluated the impacts of hardware faults inside GPUs and their impact on CNN workloads. Unfortunately, these strategies are computationally intensive and can hardly be used in large applications due to their considerable evaluation time (e.g., evaluating a CNN's layer for a faulty GPU might require around 1,000 hours, Condia et al. [2022]). However, other works assess and characterize large ML workloads by efficiently using software-based error strategies that involve the fine-grain use of the hardware/software interface and the architecture of a system (i.e., the association between the underlying hardware in a system and the application structure) by employing instrumentation frameworks to modify and corrupt targeted code instructions to represent errors arising from the underlying hardware (Hari et al. [2017b], Tsai et al. [2021b], Villa et al. [2019b]). In addition, the real system is used at speed to characterize and assess an application.

The aim of this work is to present an efficient, yet feasible way of evaluating and determining the impact of transient faults on a HSI classifier. As mentioned earlier, performing simulations at the hardware level by identifying the more sensitive physical modules and seeing their impact on the software is too computationally and time demanding, whereas assessing the reliability considering the application structure (e.g., the neural network architecture) ignores the hardware details. To take advantage of the benefits of both worlds, this analysis uses a software-based approach called *Hardware Injection Through Program Transformation* (HITPT) which simulates faults on the hardware by performing corruptions at the instruction set level. This way, the effect of a fault can be directly mapped to both the hardware and software locations.

This Master's thesis work assesses the reliability of a large hyperspectral image classifier for edge computing platforms when faults impact its implementation platform. In detail, two GPU-based systems are evaluated by employing the HITPT strategy to analyze the effects of transient faults. The analysis proved that some code routines (17% of the kernels) in the classifier are highly sensitive to corruption independently of the underlying hardware platform. Then, I developed and validated a software-based hardening mechanism (based on Duplication with Comparison fault detection technique) to reduce the impact effects from those identified highly sensitive kernels.

The present document is structured as follows: Chapter 2 (Background) provides the reader with all the necessary prerequisites needed for understanding the context of this work. It first introduces the Hyperspectral Imaging domain, followed by a brief overview of the GPU programming model. Next, it presents the reliability assessment domain describing the types of faults and experiments that can be conducted, focusing primarily on software-based simulations. Then, the chapter concludes with the definition of two metrics for quantifying the reliability of a system. Chapter 3 presents the study case of this work, describing the HSI classifier architecture and the datasets used. Chapter 4 provides an in-depth description of the fault injection tool used for conducting the experiments and a guide on how to configure it for the given study case. Chapter 5 presents the setting for the experiments, showcases and discusses the results obtained. Chapter 6 concludes this Master's thesis by summarizing the work performed, drawing a conclusion based on the obtained results and describes some future works.

Chapter 2

Background

This chapter serves as a brief overview of the essential background information required to understand the problem addressed by this work, the experiments conducted and the results obtained. It begins by introducing the field of Hyperspectral Imaging, followed by a short description of the GPU programming model. The chapter then explores the domain of reliability assessment, outlining the various types of faults and experiments that can be performed, with an emphasis on software-based simulations. Finally, it concludes by defining two metrics used to quantify the reliability of a system.

2.1 Hyperspectral Imaging (HSI)

The advanced and practical benefits of computer vision and signal processing algorithms on several applications promote their adoption and mapping on edge computing devices with strong energy and computational power restrictions.

In particular, *Hyperspectral Imaging* (HSI), or imaging spectroscopy, is a powerful technique used to analyze and extract extended information (e.g., features) from a given scenario, area, or geographical region with relevance in the analysis of information collected remotely for a wide variety of applications, including precision agriculture, geophysics, robotics, environmental monitoring, and surveillance (Arce et al. [2014], Khan et al. [2018], Imani and Ghassemian [2020], Ahmad et al. [2022, 2024]).

In detail, hyperspectral sensors attached to satellites or drones simultaneously collect many images (from tens to hundreds) of ground scenes at different electromagnetic wavelengths, organized in structured data (i.e., stacks of 2D images), representing a scene at different wavelengths and constituting a 3D hyperspectral data cube. HSIs are data which are very similar to RGB images, with the difference that along the third dimension (i.e., depth of the image), instead of having three channels corresponding to intensities of colors red, green and blue, we have channels corresponding to radiance at different wavelength bands of the electromagnetic spectrum.

If we slice an HSI image (which can be visualized as a 3D parallelepiped) along a certain wavelength, we would obtain a 2D plane where each point (pixel) corresponds to a geographical position and has a numerical value indicating the radiance of that specific

location at the selected wavelength.

2.1.1 Hyperspectral Image Classification

HSI classification represents the task of assigning a label to each pixel in the hyperspectral image. To classify a pixel, a deep learning model such as a Convolutional Neural Network (CNN) makes use of the convolution operation in order to take in consideration the spectral information of the pixel and its neighboring pixels (De Lucia et al. [2023]).

HSI data generally tend to be high dimensional, making processing them computationally demanding. Pre-processing strategies such as dimensionality reduction aim to remove redundant features from HSI data in order to avoid overfitting (i.e., to avoid that a model memorizes the dataset rather than learning the hidden patterns) and to reduce the computational complexity associated to training neural networks. Principal Component Analysis (PCA) is a dimensionality reduction technique that works by reducing the size of the dataset of N features by finding a vector space basis consisting of principal components (i.e., vectors that better describe the variance of the dataset) of dimension $M < N$ such that when the dataset is projected on this space, the information is preserved as much as possible. Working on a lower dimensional dataset leads to fewer computations, hence reducing execution time and energy consumption, while having a minimal impact on performance.

Typically, HSI applications involve matrix-vector operations, meaning that the computations can be parallelized for faster and more efficient execution. To achieve this, edge computing devices nowadays are equipped with power-efficient hardware accelerators such as Graphics Processing Units (GPUs).

2.2 GPU programming model

A GPU (Graphics Processing Unit) is a specialized hardware accelerator device designed to handle data-intensive and highly parallel tasks. Modern GPU generations are flexible enough to target the execution of many complex and data-intensive applications, including Artificial Intelligence workloads (Dally et al. [2021], Choquette et al. [2021]). In particular, tasks are parallelized by splitting the workload into smaller and independent tasks and having them executed simultaneously. In a GPU, those parallel tasks are known as *Kernels*. However, these *independent* tasks may depend on the results of other tasks at a certain point of their execution and so, synchronization strategies are needed for a correct workflow.

2.2.1 Kernel

Simply put, a kernel is a function defined in a high-level programming language like C++ describing the workflow of one "smaller and independent" task mentioned earlier (i.e., the main purpose of a kernel is to describe the operation of one parallel thread). When a kernel is executed, it instantiates N different *threads* running in parallel.

2.2.2 Thread hierarchy

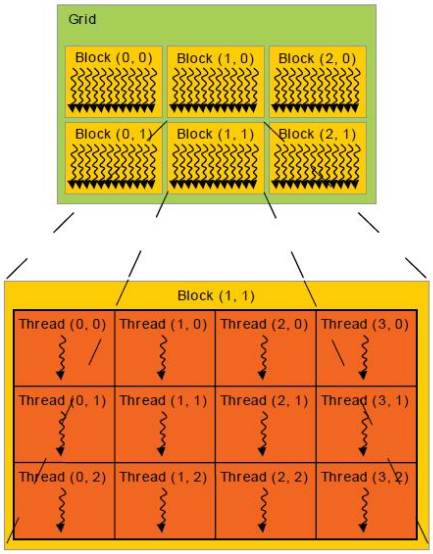


Figure 2.1: 2D example of thread hierarchy. Source here

Depending on the problem being solved, a kernel execution can have its threads grouped logically in a geometric way (i.e., 1-dimensional, 2-dimensional and 3-dimensional) under different granularity and hierarchies. Figure 2.1 depicts the 2D scenario. Please note that the following concepts being introduced use terminologies defined by NVIDIA, but the concepts are the same as the ones you may see from other GPU vendors, such as AMD.

Thread

A thread is the basic operative execution flow in the GPU programming model. Threads are supported by hardware mechanisms to dispatch them efficiently with minimal interaction from the programmer.Reasoning about a kernel execution in terms of threads means looking at the problem at a fine granularity, seeing it at a "low-level".

Block

A block is a group of threads that can have its threads organized one-dimensionally, two-dimensionally or three-dimensionally. Some examples to illustrate the usage of blocks are the operations vector addition and matrix multiplication. In the case of vector addition, you can define a 1-dimensional block of size N, where the i^{th} thread sums the two numbers located on the i^{th} position on the two vectors. For matrix multiplication, a non-optimal way of performing this operation is by defining a two-dimensional block of threads, where each thread at position (i, j) computes the dot product between the i^{th} row of the first matrix and j^{th} column of the second one.

Grid

A grid is a group of blocks. Just like blocks with threads, a grid can be organized one-dimensionally, two-dimensionally or three-dimensionally. A grid represents the highest-level at which you can look at the problem (i.e., coarse granularity).

2.2.3 Key terminologies

Some terminologies needed later on in this chapter are the following:

- **SASS**: GPU assembly language code on Nvidia GPUs generated by compiling a high-level programming source code;
- **opcode**: assembly language instruction. Most opcodes have one or more source registers and one destination register;
- **static kernel/opcode**: indicates a certain type of kernel/opcode. For example, you can have the instruction ADD present in the SASS of a program;
- **dynamic kernel/opcode**: indicates an instance of a certain static kernel/opcode. For example, you have the ADD instruction present in the SASS 12 times. Each 12 invocations of the instruction ADD represents a dynamic opcode. Analogous for kernels.

2.3 Reliability assessment and estimation

Generally, neural networks are trained and tested in a controlled environment, assuming that the final product will be deployed in ideal scenarios, but that is definitely not the case with edge computing. IoT devices in domains such as robotics, automotive, and aerospace are characterized by rough, non-deterministic environments and a high likelihood of disasters in case of application misbehavior. Because of this, artificial intelligence, due to its adoption in safety-critical domains, faces new reliability challenges (Ahmadilivani et al. [2024], Ruospo et al. [2023b]).

Reliability is the subject that ensures that a system performs its intended task correctly even under the presence of faults. Reliability and fault tolerance have been of great research interest for many years, and they are still so nowadays. In recent years, many studies have been done on developing strategies for assessing the reliability of AI models in uncontrolled environments, but there is still a significant need for further analysis (Ahmadilivani et al. [2023], Ruospo et al. [2023a], Santos et al. [2023], Ruospo [2022], Guerrero-Balaguera et al. [2022], Condia et al. [2022, 2021], Ruospo et al. [2021]).

2.3.1 Permanent and transient faults

Reliability assessments of a system or application are intended to verify their operative behaviors when faults arise from the underlying hardware (propagated as software errors) to corrupt the system operation and compromise its execution. In detail, a fault represents

a defect or anomaly that can potentially cause a system to misbehave. In the hardware context, there are several fault models. However, the two most relevant fault models, which are highly studied and mandatory by industrial standards (Gosavi et al. [2018], Pöhls [2023]), are permanent and transient faults.

Permanent faults

A permanent fault represents a physical defect in the hardware. For example, a permanent fault in a register can cause one of its bits to be stuck at 0 or 1. In fact, the stuck-at-1 faults and stuck-at-0 faults are part of the classical stuck-at-fault model, which is used to represent and analyze permanent faults in a system's hardware.

Transient faults

Transient faults might also arise at the hardware level, but these effects are non-permanent. They can be caused by environmental factors, such as external radiation and cosmic rays that impact and propagate across a circuit, changing one or more logic states and affecting the memories or flip-flops of a system (i.e., transient faults can modify the value stored in a register at a certain moment in time by flipping one of its bits). However, once a new value is written on the register, the transient fault vanishes, i.e., it does not propagate to further writings on the register.

In particular, modern systems that include many transistors and memories, such as GPUs, might be vulnerable to the impacts of transient faults during their in-field execution.

2.3.2 Experimental-based reliability assessment strategies

This subsection describes the experimental assessment strategies used in academia and industry to analyze the reliability of a system. In general, there are several types of strategies for studying a system's resilience and identifying the vulnerable structures of AI hardware.

Formal evaluations

These kinds of works focus on the analysis of neural networks to understand how structural irregularities in the architecture or at the inputs may impact the resilience performance of the models. Bhatti et al. [2022] studied the effects of real-world noise from the input data on the output of a hidden single layer. More specifically, they evaluated the model based on three properties: *i)* the first one is robustness, which represents the fact that when an input is correctly classified without any noise present, the model should correctly label the input also in the presence of noise, *ii)* the second property is training bias, which means that samples from some classes will more likely still be correctly classified after some amount of noise compared to other classes. Lastly, *iii)* the third property is node sensitivity, describing the noise sensitivity of each input node of the model.

While this kind of evaluation is definitely of great interest, it completely ignores the hardware details of the device running the model.

Application-based evaluations

Application-based evaluations study the effects of changes made to a neural network's inputs, parameters, feature maps, etc. Unlike formal evaluations, they require an understanding of the neural network architecture to understand how faults (e.g., bit-flip on a parameter) propagate through the model and to identify the parts of the architecture that need to be hardened (Chen et al. [2020], Mahmoud et al. [2020]).

Just like formal methods, application-based evaluations do not take in consideration the underlying hardware.

Software-based experiments

Software-based experiments (a.k.a. Hardware Injection Through Program Transformation or 'HITPT') consist of a paradigm of reliability evaluation that modifies the application code by introducing fine-grain software operations/routines representing error effects from fault corruptions occurring in the underlying hardware. Then, the instrumented code is executed on the platform to determine impact effects during the real application's operation. The HITPT strategy is economical and can effectively analyze fault corruption effects from a system's operative data path (e.g., memories and execution units). In fact, the strategy has been successfully adopted by several works on the evaluation of large workloads, including AI algorithms on CPUs (Vargas et al. [2014, 2018]) and hardware accelerator platforms, including GPUs (Guerrero Balaguera et al. [2023]Hari et al. [2017b], Tsai et al. [2021a]). However, well-defined software error models are required to provide affordable system characterizations against faults and errors.

Software-based experiments are the main focus of this thesis work. In Section 2.4, NVBitFI is introduced and described.

Simulation-based experiments

Simulation-based experiments consist of a set of fault simulation campaigns, with the support of logic simulators, injecting one or several faults into a representative fine-grain system model (e.g., structural/functional simulators or a hardware abstraction model, including RT or gate-level) (Condia et al. [2020], Pinto-Salamanca et al. [2024], Sierra et al. [2023b, 2024], Li et al. [2017a], Hari et al. [2012], Sierra et al. [2023a], Pessia et al. [2024], Bosio et al. [2019], Limas Sierra et al. [2024]). This method can provide fine-grain identification of those vulnerable hardware/software modules in a system. However, a representative and accurate model is required. In addition, the injection of faults is limited to a selected fault model (e.g., transient or permanent fault models) and its accuracy to represent errors. Unfortunately, the large evaluation times and the required computational power restrict their adoption into complex and large systems, such as AI-powered ones (e.g., the simulation-based reliability evaluation of a simple *Convolutional Neural Network* 'CNN' on a GPU might require more than 10,000 days Condia et al. [2022]).

Physical experiments

Physical experiments (e.g., beam experiments) evaluate the system by exposing it to an external source able to produce effects/faults during the execution of the system's application (e.g., radiation/electromagnetic source)Fernandes dos Santos et al. [2017]. These analyses effectively evaluate a system's overall reliability to transient fault effects (e.g., *Single Event Upsets* or SEUs)Santos et al. [2019], Fernandes dos Santos et al. [2019], Oliveira et al. [2020b]. However, the strategy is costly since it requires specialized facilities and can hardly provide fine-grain identification of the most vulnerable modules in a system due to implicit limitations (e.g., observation restricted to the outputs of the application)Gnad et al. [2024], Oliveira et al. [2020a].

Hybrid mechanisms

Hybrid mechanisms represent a combination of two or more of the strategies mentioned above to benefit of their advantages (Santos et al. [2021b]). For example, you may want to assess the reliability of a program from both hardware and software perspectives. To achieve this, one could perform simulation-based experiments to evaluate the fault tolerance of the device when executing the application, and application-based evaluations to understand the effects of corruptions occurring on the architecture of a neural network on the output of the model.

2.4 Software-based injections and tools for edge computing and GPUs (NVBitFI)

There are many HITPT frameworks for performing software-based injections on real GPUs, such as SASSIFI (Hari et al. [2017a]), Hauberk (Yim et al. [2011]), and GPU-Qin (Fang et al. [2014]). However, only a subset inject faults at the assembly code level (SASS in the case of NVIDIA GPUs).

On the one hand, SASSIFI is a framework that represents errors by replacing any targeted instruction in the intermediate-level (PTX) code while compiling the higher-level language code. In particular, this framework adds a subroutine with several instructions to modify a destiny register with corrupted information to mimic the effect of hardware faults. For instance, the instruction command `"MOV R1, R2"` uses `R2` as the source register and `R1` as the destiny register. The framework adds a function/routine to corrupt the final state of `R1` when targeted for fault injection.

On the other hand, GPU-Qin is a framework that employs a different approach and takes advantage of the cuda-gdb debugger to inject faults at locations indicated by breakpoints.

In this thesis work, the tool NVBitFI (Tsai et al. [2021a]) developed by NVIDIA and based on another framework called NVBit (Villa et al. [2019a]) has been adapted and used, since this is an evolved version of SASSIFI and provides further fine-grain control when injecting and profiling fault effects. In particular, NVBitFI is an instrumentation tool that performs fault injection at the assembly (SASS) code level, allowing the assessment

of the reliability and availability of a GPU application. Unlike the previously mentioned methodologies, NVBitFI is able to inject faults during the execution of the program, meaning that it does not need access to the source code, nor does it need to recompile the program. This makes NVBit a flexible and efficient tool without adding too much overhead during simulations. This is a great advantage since fault simulations in general need an exhaustive amount of simulations.

2.4.1 NVBitFI software architecture

This subsection introduces and briefly describes the main features of the NVBitFI framework for software-based error injection. NVBitFI consists of two main tools: profiler and injector.

Profiler

The profiler is a tool that analyses (profiles) an executing GPU program and builds a file called *profile* describing the program's workflow. More specifically, the profile contains a line for each dynamic kernel the program executes. Each line, besides the kernel name, also includes some statistics about the SASS code of the executed kernel, i.e., for each opcode (SASS instruction), it reports the number of times the opcode was executed.

The impressive aspect of NVBitFI, compared to other frameworks that work at the SASS level, is that its profiler can gather all the information (e.g., executed SASS instruction per thread, warp, block, and grid) without having access to internal details of the running application on the GPU. This feature makes NVBitFI a powerful tool since it allows the study of the reliability of proprietary GPU applications without violating the secrecy of a company's software.

Injector

NVBitFI comes with two types of injectors: an injector for transient faults and an injector for permanent faults. The injector for transient faults injects a fault on a single dynamic opcode. On the other hand, the permanent fault injector will target all the dynamic instructions belonging to the same opcode, simulating a physical defect on the part of the hardware responsible for executing that specific type of instruction.

The strength of the injector of NVBitFI is that it is capable to inject errors directly into the fault-free compilation of the program without needing to recompile the source code for every fault simulation.

2.4.2 (Transient) Fault injection procedure

The workflow of performing transient fault simulations can be described in four steps.

Step 1: Generate target program profile

By means of the profiler, NVBitFI builds the profile to find all the possible fault injection sites of the program. The profile is later used as a uniform probability distribution in

which the samples to be drawn from it are the dynamic SASS instructions.

Step 2: Select single-injection parameters

With the profile of the application ready, the framework can now randomly select a predefined number of dynamic opcodes to be subject to faults.

The dynamic opcodes within the profile are identified by a tuple *(kernel_name, kernel_count, opcode_count)* which helps the injector understand at what point of the execution of the program it should interfere and insert the fault.

Step 3: Inject fault

In NVBitFI, when a transient fault is injected, it simulates it by modifying the value on the destination register according to an initially selected fault model (e.g., bit-flip, write zero, write random value etc.).

Step 4: Analyze target program output

Finally, once the fault has been injected and the corrupted execution has finished, NVBitFI will analyze the output of the program by comparing it to the output of the same program to the same input under a fault-free scenario (in this case, the output is called golden output).

2.4.3 Instruction groups targeted

As previously mentioned, NVBitFI simulates hardware faults by performing various bit-flip models on the software at instruction set level. The instruction groups supported by NVBitFI as possible targets for corruptions are the following:

- G_FP64: floating-point operations in 64 bits;
- G_FP32: floating-point operations in 32 bits;
- G_LD: instructions reading from memory;
- G_PR: instructions writing to predicate registers;
- G_NODEST: instructions without destination registers;
- G_OTHERS;
- G_GPPR: group consisting of all instruction, except for G_NODEST;
- G_GP: instructions writing on general purpose registers.

The bit-flip models supported by NVBitFI are:

- FLIP_SINGLE_BIT;

- FLIP_TWO_BITS;
- RANDOM_VALUE;
- ZERO_VALUE.

2.4.4 Fault outcome categorization

By default, NVBitFI classifies a transient fault outcome assigning one of the following categories: Masked, SDC and DUE.

Masked

A masked outcome represents a fault occurrence that did not cause any change in the application. When a specific part of the hardware architecture is targeted for fault injections and most of the times it resulted in a masked outcome, it means that that component of the device has a high fault tolerance.

SDC

SDC, which stands for *Silent Data Corruption*, is an outcome indicating that the fault led to changes in the behavior of the system, but did not cause the program to crash (hence why *silent*).

DUE

DUE, which stands for *Detected Unrecoverable Error*, is an outcome representing a crash or a hang of the application (e.g., an exception in the application execution).

When a DUE occurs, a user or monitor system can detect it and possibly fix the cause of the error since flags, interruptions, or exceptions are triggered.

In contrast, SDCs occur without interrupting the system. Still, it corrupts the operations and the results, so if the source of SDC is not addressed soon, it could potentially lead to application hazards.

2.5 Reliability metrics

Several evaluation metrics have been proposed to evaluate the impact of faults and errors in the operation of complex systems and applications. Among them, two metrics that are often employed in the reliability assessment of software are Program Vulnerability Factor and Mean Execution Between Failure.

2.5.1 Program Vulnerability Factor (PVF)

Program Vulnerability Factor (PVF) is a metric indicating how likely a fault occurring in a program will lead to an error or wrong result (Sridharan and Kaeli [2008], Fang et al. [2016]).

$$PVF = \frac{number\ of\ faults\ resulting\ in\ an\ error(SDCs)}{total\ number\ of\ faults\ injected}$$

2.5.2 Mean Execution Between Failure (MEBF)

This metric is an adaptation of a metric used in hardware domain. Tambara et al. [2015] used this metric to quantify the effects of radiation-induced errors on the device. They first began by computing a metric called cross-section (σ).

$$\sigma = \frac{number\ of\ errors}{fluence}$$

where $fluence$, in this work, represents the amount of faults injected in the application and $number\ of\ errors$ means the number of times a fault led to an error (failure).

Next, another metric called Mean Time Between Failure ($MTBF$) is defined as the ratio between the inverse of the cross section and the flux.

$$MTBF = \frac{1}{\sigma \times flux}$$

The $flux$ represents the fluence per time unit, i.e., it represents the number of fault occurrences per unit of time. In the scenario of software-based fault simulations (which is the case in this thesis work), unlike in the case of physical experiments, the $flux$ is a controlled and fixed value. For convenience, $flux$ is set to 1.

Finally, based on the cross section and $MTBF$, $MEBF$ is defined as the ratio between $MTBF$ and the execution time of the application.

$$MEBF = \frac{MTBF}{t} = \frac{fluence}{t \times number\ of\ errors}$$

$MEBF$ is a measure describing the interval between two failures (two SDCs in this case) in terms of number of executions.

When performing simulations, it is assumed that all experimental configurations are subjected to the same conditions. For example, on average a transient fault is injected every T seconds (this T value is the same for all experiments and it is absorbed in the $flux$). MEBF describes the interval between failures in terms of number of executions and so, a workload with a lower execution time will have a higher number of executions in between two failures than a workload with a larger execution time.

MEBF, as defined here, is a relative metric that helps in comparing the number of executions of two programs. To obtain the actual number of executions between two failures, one must provide a real-world value for the $flux$.

Chapter 3

Study case: Hyperspectral Image Classification

This work builds on top of the work performed by De Lucia et al. [2023], where the authors trained and tested different 3D Convolutional Neural Networks (3D CNNs) for the task of Hyperspectral Image Classification. Their goal was to evaluate the performance, energy consumption and time efficiency of the models, which are crucial aspects in edge computing. For our analysis, we selected the model by Li et al. [2017c].

The goal of this Master's degree thesis is to assess the reliability of the HSI classifier and to harden at software level the sensitive parts of the model in order to increase its fault tolerance.

3.1 Pipeline

The pipeline of the HSI classifier consists of two phases: pre-processing and inference.

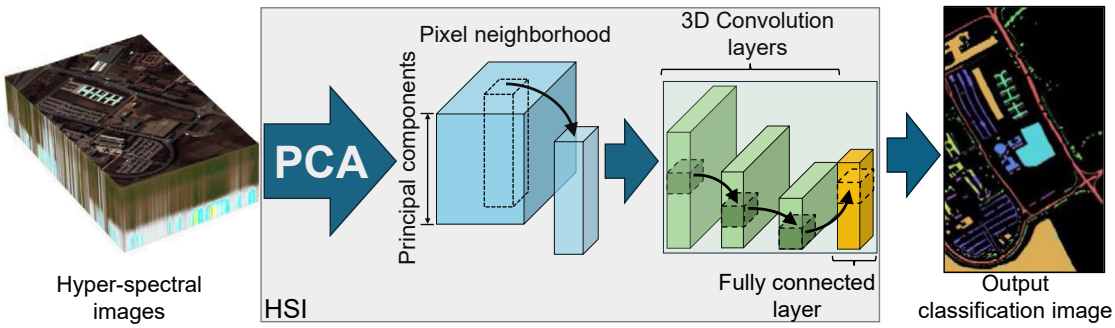


Figure 3.1: Li et al. HSI classifier comprising image pre-processing (PCA) and AI algorithms (CNN).

3.1.1 Pre-processing

As described in the Background chapter, HSI data tends to be high dimensional and this can be a problem in edge computing. In order to reduce energy consumption and computational complexity while also improving the performance of the classifier, De Lucia et al. [2023] employed an unsupervised dimensionality reduction strategy called Principal Component Analysis (PCA).

In its "classical" form, PCA finds the principal components by calculating the eigenvalues and eigenvectors of the covariance matrix of the dataset. The problem is, when the dataset is too highly dimensional (i.e., the number of features of a sample in the dataset is high) like in the case of HSI, the covariance matrix can be too large to fit in memory. To overcome this, De Lucia et al. [2023] implemented PCA using Gram-Schmidt PCA (GS-PCA) by Andrecut [2009], which is an iterative version of PCA based on NIPALS-PCA and Gram-Schmidt orthogonalization process to obtain orthogonality among the principal components.

The output of GS-PCA is a lower dimensional dataset to be fed to the 3D CNN in the inference phase. Depending on the computational capability and the electric power at disposal, one may choose a certain number of principal components to use for projecting the HSI dataset on a lower dimension. If the device has stringent requirements for power consumption or low compute performance, a low number of principal components can be selected. Whereas, if these requirements are not too strict, one may choose a larger number of principal components.

cuBLAS

To speed-up preprocessing, GS-PCA was implemented in cuBLAS to make it executable on GPU. cuBLAS is an NVIDIA CUDA adaptation of the BLAS library for performing basic linear algebra operations.

De Lucia et al. [2023] implemented GS-PCA making use of the following cuBLAS functions:

- **cublasSgemv**: performs single-precision general matrix-vector operations;
- **cublasSaxpy**: multiplies an input single-precision vector by a single-precision scalar and adds the result to another single-precision vector;
- **cublasSnrm2**: computes the Euclidian norm of a single-precision vector;
- **cublasSger**: performs the symmetric rank 1 operation.

3.1.2 Inference

The classifier being used is Li et al. [xx], which is a CNN model leveraging 3D convolution layers implemented in PyTorch. More specifically, the model consists of:

- 2x 3D Convolution layers
- 1 Fully Connected layer

The difference between 3D convolution layers and the traditional 2D convolution layers, which are widely used in the field of computer vision, is that the filters applied on the feature map slide also depth-wise through the wavelength bands, as opposed to just along the width and height of the input image.

After each convolution layer, the model has an activation function called ReLU (Rectified Linear Unit, Agarap [2018]).

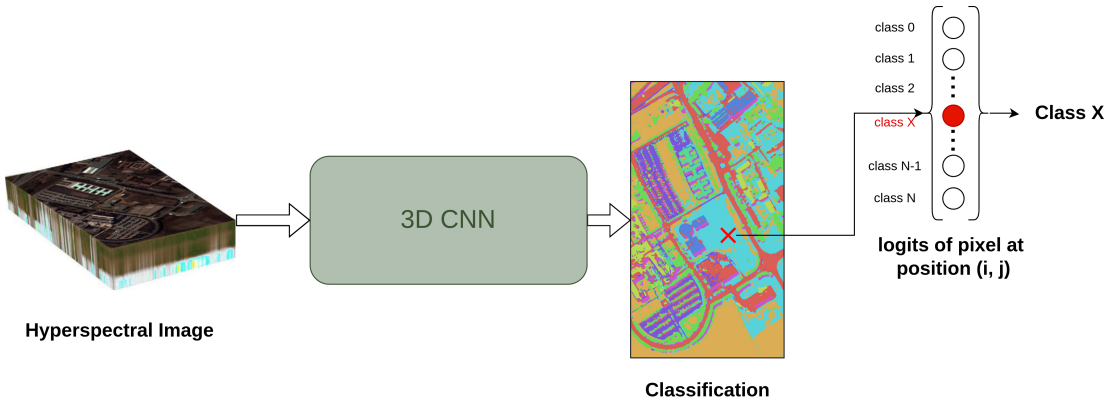


Figure 3.2: 3D CNN classifier workflow

In figure 3.2, the output of the 3D CNN depicts the classification of the HSI image. The output is actually a visual representation of the classification in which the label assigned to a pixel is illustrated by a color indicating the type of material that pixel represents.

Logits

In general, for a certain input, Neural Networks give as output a vector of size N called **logits**, where N is the number of classes that could be assigned to the input sample. A numerical value at the i^{th} position within the logits vector corresponds to how likely the input sample belongs to the class i and so, the assigned label corresponds to the position within the logits vector of the highest value.

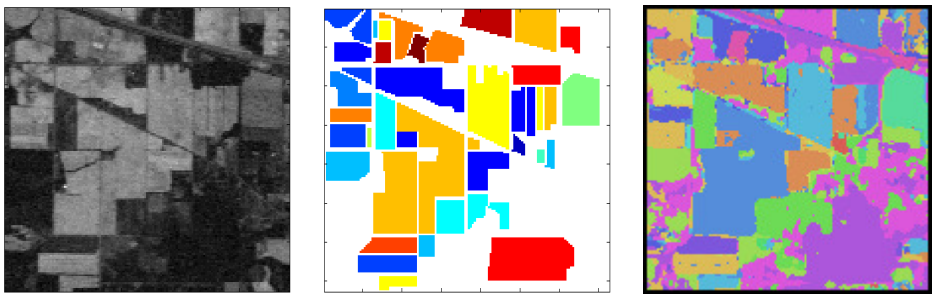
Figure 3.2 illustrates the meaning of logits in the context of HSI classification, where each pixel is assigned a label. The classifier will output a logits vector for each pixel and based on it the classification is performed.

3.2 Datasets

De Lucia et al. [2023] trained and tested the model on three HSI datasets.

3.2.1 Indian Pines

Indian Pines is a HSI dataset collected by the AVIRIS sensor of NASA attached to an aircraft while flying over a site in North-Western Indiana. It has a resolution of 145×145

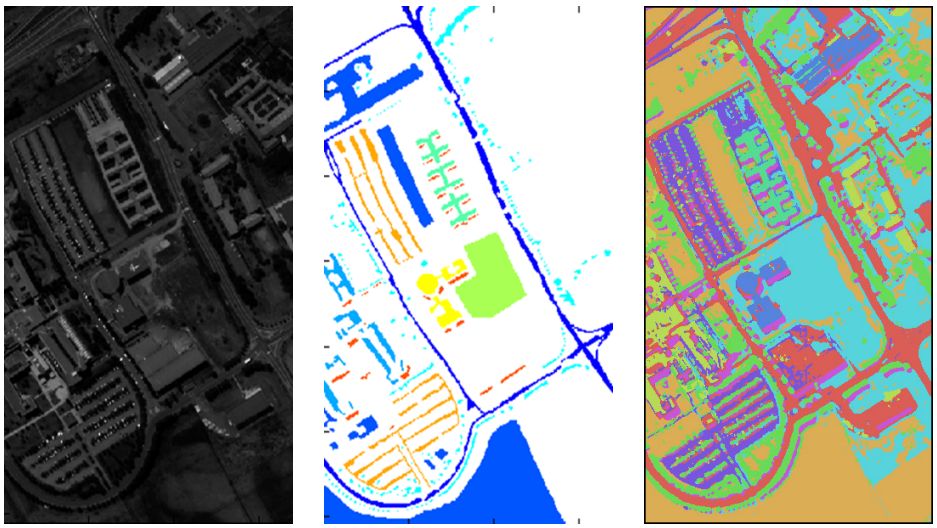


(a) black & white (b) ground truth (c) classifier inference

Figure 3.3: Indian Pines

and 224 spectral reflectance bands with wavelengths in the range $[0.4 \times 10^{-6}, 2.5 \times 10^{-6}]$ and a ground resolution of 17 meters. However, for the experiments performed in this work, the water absorption bands were removed, remaining with 200 bands. The ground truth of this dataset consists of 16 classes.

3.2.2 Pavia University



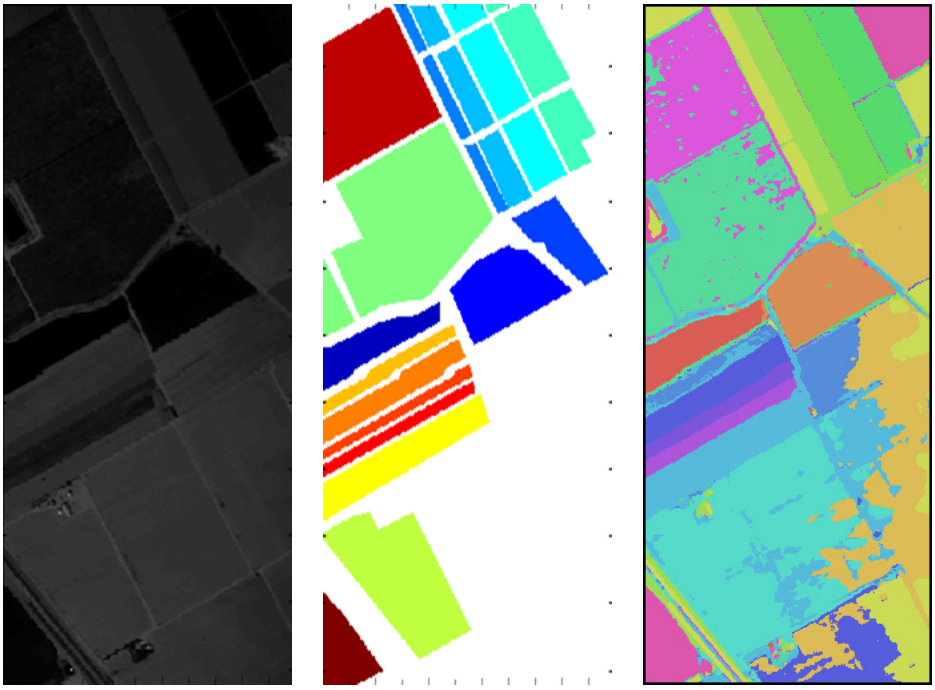
(a) black & white (b) ground truth (c) classifier inference

Figure 3.4: Pavia University

This dataset was obtained by means of a ROSIS sensor during a flight over Pavia, Italy in 2002. Some of the samples recorded over the territory have no information and

so, after removing them, the final HSI image has a resolution of 610×340, with a ground pixel resolution of 1.3 meters. The samples contain 103 spectral bands. The ground truth of Pavia University dataset consists of 9 classes.

3.2.3 Salinas



(a) black & white (b) ground truth (c) classifier inference

Figure 3.5: Salinas

Like Indian Pines, Salinas dataset was obtained using the AVIRIS sensor. The scene depicted in the dataset corresponds to Salinas Valley, California. The image resolution is 512×217, with a ground pixel resolution of 3.7 meters. Again, just like for Indian Pines, the water absorption bands were removed, leaving a total of 204 bands. The ground truth of this dataset consists of 16 classes.

Chapter 4

Fault injection framework adaptation (NVBitFI)

This chapter presents a detailed explanation for setting up NVBitFI for conducting transient fault injection simulations on the HSI classifier by Li et al. [2017c]. First, it presents a very brief description of the installation process, then it shows the directory structure and provides a thorough framework configuration description. Finally, the fault outcome classification is explained, since for this specific work the fault injection outcome categories differ with respect to the default ones proposed by NVBitFI and presented in the background chapter.

4.1 NVBit and NVBitFI installation

NVBitFI was built on top of the NVBit framework so, a user should install the latter (NVBit framework) before being able to use the fault injection tool. A guide for installing both tools may be found in the README of this github repository.

4.2 Directory structure and framework configuration

Once both tools have been installed correctly, NVBitFI source code can be found by moving to the directory `nvbit_release/tools/nvbitfi` inside the directory where NVBit was installed (e.g., `~/nvbit`).

Figure 4.1 shows the directory structure of NVBitFI source code. The files and folders highlighted in red are the most relevant parts for setting-up the framework for performing transient fault injections on the HSI classifier. Please note that the folder `PCAHyperspectralClassifier` does not come with NVBitFI. It must be copied from the this GitHub repository.

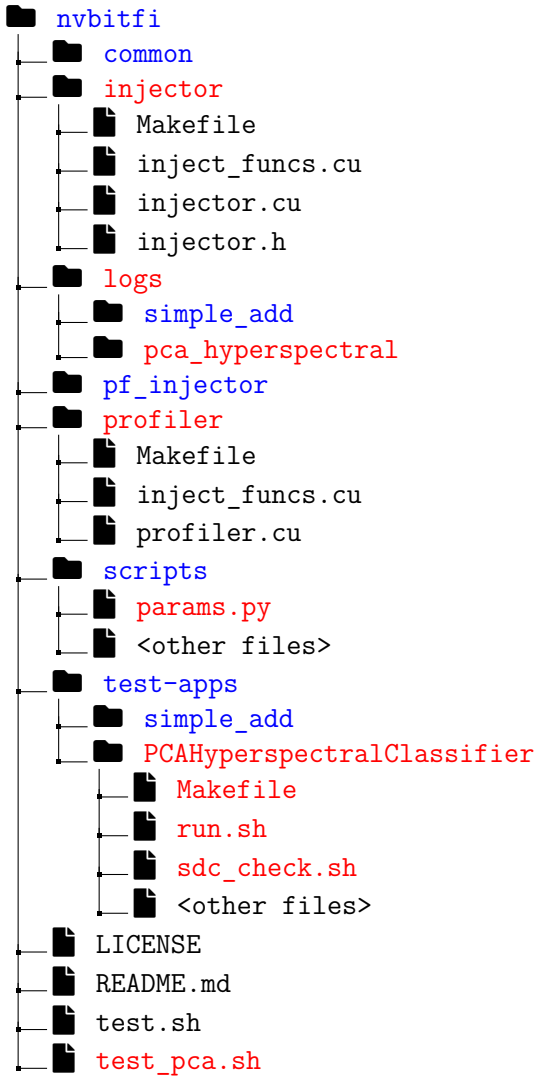


Figure 4.1: NVBitFI directory structure

4.2.1 nvbitfi/injector & nvbitfi/profiler

The two directories contain the source code of the injector and profiler. They must be compiled the first time you run a fault injection campaign. To do so, change directory to each path and run the `make` command. These are done automatically when executing `test_pca.sh` (more on this one later), however sometimes you may need to do them manually when getting strange errors from the `make` command later on during the adaptation of the tool for the HSI classifier.

4.2.2 nvbitfi/scripts/params.py

As the name suggests, `params.py` contains the parameters for configuring the tool for running the experiments accordingly. There are many parameters defined in this script, but the most important ones to be set are the following:

inst_value_igid_bfm_map

This is a dictionary consisting of one or more key-value pairs of the form

```
instruction_group: [bitflip_model_1, bitflip_model_2, ...]
```

`instruction_group` is a variable that should assume an integer value between 0 and 7 corresponding to the instruction group on which the fault should be injected.

`bitflip_model_i` represents the type of corruption we want to perform (e.g., flip one bit, write random value etc.). We can specify more than one bit-flip model for the same instruction groups. For instance, a configuration of the framework to target the General Purpose Registers (injecting single-bit flips and random values), the Load instructions (injecting single-bit flips and random values), and the 32 bits Floating point instructions (injecting single-bit flips, random values, and random values on warps) can be set as:

Listing 4.1: Example of configuration for several target fault groups

```
# values for params.py script
inst_value_igid_bfm_map = {
    # General Purpose Registers
       G_GP: [FLIP_SINGLE_BIT, RANDOM_VALUE],
    # Load instructions
       G_LD: [FLIP_SINGLE_BIT, RANDOM_VALUE],
    # 32 bits Floating point instructions
      G_FP32: [FLIP_SINGLE_BIT, RANDOM_VALUE,
               WARP_RANDOM_VALUE]}
```

apps

`apps` is a dictionary storing key-value pairs of the form

```
app_name: [
            app_path,           # path to the application's
                                # directory
            executable_name,    # name of the executable
            executable_path,    # path to the directory of
                                # the executable
            expected_runtime,   # runtime in seconds of
                                # the application
                                # running on a specific machine
```

```
        app_params          # list of parameters to be
                            # passed to run.sh as a string
]
```

This dictionary can consist of more than one key-value pair for configuring the tool to run fault injection campaigns on multiple applications. Note that `app_params` must be a string where the arguments are separated by blank spaces. If no parameters needed, `app_params` should be an empty string (i.e., `""`).

NUM_INJECTIONS

Indicates the number of injections to create randomly. These injection sites are created prior to the beginning of the injection campaigns. This number of injection sites is done for each combination of *(instruction group, bit-flip model)* specified in `inst_value_igid_bfm_map`. For example, if you have `inst_value_igid_bfm_map` as

```
inst_value_igid_bfm_map = {
                    G_FP32: [FLIP_SINGLE_BIT, ZERO_VALUE],
                    G_GP: [FLIP_SINGLE_BIT, FLIP_TWO_BITS]
                }
```

and `NUM_INJECTIONS=1000`, then NVBitFI will generate a total of 4000 fault sites (1000 for `(G_FP32,FLIP_SINGLE_BIT)`, 1000 for `(G_FP32,FLIP_SINGLE_BIT)` etc.).

THRESHOLD_JOBS

This parameter indicates the actual number of faults to be simulated in each *(instruction group, bit-flip model)* combination. Out of the `NUM_INJECTIONS` injection sites created, NVBitFI will randomly select a number equal to `THRESHOLD_JOBS` to target.

TIMEOUT_THRESHOLD

This is a number for letting NVBitFI know how long should it wait for a simulation (i.e., an execution of the application under the presence of a fault) before interrupting it and assign it a DUE outcome due to a hang. The actual time the framework will wait is equal to `TIMEOUT_THRESHOLD*expected_runtime` (recall: `expected_runtime` is defined in the `app` dictionary mentioned earlier).

4.2.3 nvbitfi/test-apps/PCAHyperspectralClassifier

This directory contains the source code of the HSI classifier. This directory must be copied inside `nvbitfi/test-apps/PCAHyperspectralClassifier`. Once copied, for NVBitFI to work, three extra files are needed: `Makefile`, `run.sh` and `sdc_check.sh`. You can simply copy the examples reported in `nvbitfi/test-apps/simple_add` and modify them to fit your use case.

nvbitfi/test-apps/PCAHyperspectralClassifier/Makefile

The make file is needed for automatically setting up the project from within `nvbitfi/test_pca.sh` (more on this file later). There are some rules and environmental variables that need careful attention.

Following are some important environmental variables:

- `TARGET`: must be set to the name of the application directory (in our case, PCAHyperspectralClassifier)
- `ARCH`: must be set to an integer corresponding to the compute capability of the GPU on which NVBitFI is executed. For example, the NVIDIA GeForce GTX1050 GPU has compute capability 6.1 and so, we must set `ARCH=61` (i.e., compute capability multiplied by 10). Here you can find the compute capabilities of all NVIDIA GPUs available on the market.

Following are some important make rules:

- `PCAHyperspectralClassifier`: this rule defines the compilation of the application. In this specific case, only the pre-processing part of the HSI classifier must be compiled, since the neural network part is implemented in Python using PyTorch;
- `golden`: this rule is called within `test_pca.sh` to generate the golden outputs, i.e., the output of the HSI classifier without any faults injected. The command associated to this rule is the same as the one you would use for executing the application outside the fault injection tool with the addition that the standard output and standard error are redirected to two files: `golden_stdout.txt` and `golden_stderr.txt`, respectively;
- `clean`: use the same `clean` rule from `nvbitfi/test-apps/simple_add` and modify it by removing extra outputs generated by the application.

nvbitfi/test-apps/PCAHyperspectralClassifier/run.sh

`run.sh` is executed by the profiler and injector. It contains the same command as the one associated to the `golden` rule in the Makefile previously described. NVBitFI will run this command for initiating the profiling process and for each fault injected. During the execution, the injector will intervene to corrupt the SASS instruction that was initially selected as target when it is reached;

The best thing to do is to copy the version of this file present inside `nvbitfi/test-apps/simple_add` and modify it.

nvbitfi/test-apps/PCAHyperspectralClassifier/sdc_check.sh

As the name suggest, this shell script defines a set of commands for determining whether a fault injection led to an SDC outcome or not. To do so, it checks whether there are some differences between the output of the model under corruption and the golden output.

The `sdc_check.sh` file in `nvbitfi/test-apps/simple_add` looks only at the differences between the standard output and standard error and their golden counterparts. In the case of the HSI classifier, the main output is not the standard output (like in the case of `simple_add`), but it has a third output file called `prediction_inference.tif`. `sdc_check.sh` must also check the difference between this output and its golden counterpart.

One very important step that must be done is to remove any non-deterministic values printed on standard output and error (e.g., execution runtime). This can be done by combining commands such as `grep` and `tr`.

The workflow of `sdc_check.sh` is as follows:

1. Compute `diff.log` by comparing `prediction_inference.tif` with its golden version (use shell command `diff`);
2. Compute `stdout_diff.log` by comparing `stdout.txt` with `golden_stdout.txt`
3. Compute `stderr_diff.log` by comparing `stderr.txt` with `golden_stderr.txt`
4. Compute `special_check.log` by **appending** to it the three "diff" files mentioned above. This file is checked by NVBitFI to understand if something happened. If the file is not empty, NVBitFI will be triggered to check what exactly went wrong (i.e., which of the three files has changed).

4.2.4 nvbitfi/test_pca.sh

Executing this bash script initiates the fault injection campaigns. It can be divided in several steps and sub-steps. I definitely encourage to copy the version of this script from the `simple_add` example and modify it accordingly. Below the steps are reported, with a comment mentioning which ones are the same as the `simple_add` example.

- Step 0: Setup
    1. Add execution permissions to the shell scripts located in `nvbitfi` directory (e.g., `nvbitfi/test_pca.sh`; no difference with respect to `simple_add`);
    2. Setup environment variables. Here you may add new variables missing from the `simple_add` needed for the HSI classifier;
    3. Compile the injector and profiler (needed to be done only one time; no difference compared `simple_add`)
    4. Run the application without instrumentation. Here you must change the change directory command argument to the path of the application (i.e., `test-apps/PCAHyperspectralClassifier/`)
- Step 1: Profile and generate injection list
    1. Profile the HSI classifier (no changes)
    2. Generate injection list (no changes)

- Step 2: Run fault injection simulations (no changes)
- Step 3: Parse the results (no changes)

4.2.5 nvbitfi/logs

This directory is automatically created once the first fault injection campaign is run. It contains a directory for each application tested (e.g., `simple_add` and `PCAHyperspectralClassifier`) and each directory contains the logs for each fault injection performed.

Besides the logs, `nvbitfi/logs/pca_hyperspectral` contains also the injection list generated at the beginning of the campaign and the profile of the application called `nvbitfi-igprofile.txt`.

It is important to remember that everytime you decide to rerun some experiments, you must empty the `nvbitfi/logs/pca_hyperspectral` directory. Otherwise, the new results will overlap with the old ones.

The logs stored here are used for the analysis described in the Experimental Results chapter. These logs contain the output of the application after a fault injection, the `diff*.txt` files explained earlier and two files `nvbitfi-injection-info.txt` and `nvbitfi-injection-log-temp.txt` indicating the targeted dynamic kernel, dynamic opcode and before and after value in the destination register.

4.3 Fault and Error Classification

As explained in section 2.4.4, a fault outcome can be classified as either masked, SDC or DUE. In this work, the SDC category has been split in two sub-categories:

- **SDC-safe**: represents the case in which the fault injected caused a change in the output logits (see section 3.1.2) of the classifier but did not change the final classification with respect to the golden one. This means that the fault caused changes in the numerical value in the logits vector, but it did not change the position of the maximum value within the logits.
- **SDC-critical**: indicates the situation in which both the logits output and the classification performed by the model changed due to the fault.

The simulations assigned as SDC by NVBitFI correspond to SDC-critical, since the script `sdc_check.sh` checks whether the "faulty" classification and the golden one are different or not without taking into consideration any changes at logits level.

As a consequence, all SDC-safe outcomes have been classified as masked by NVBitFI and so a post-processing step is required in order to separate SDC-safe cases from the masked ones. To be able to do that, before running the experiments, the application of De Lucia et al. [2023] was slightly modified to output, besides a file showing the classification outcome called `prediction_inference.tif`, a numpy object storing the logits of all the pixels in the HSI image. Then, when looking among the cases considered masked by NVBitFI, if an execution of the application under a fault generated logits that are different with respect to the golden logits, that execution is considered SDC-safe.

4.4 Workflow of reliability assessment and software-based hardening

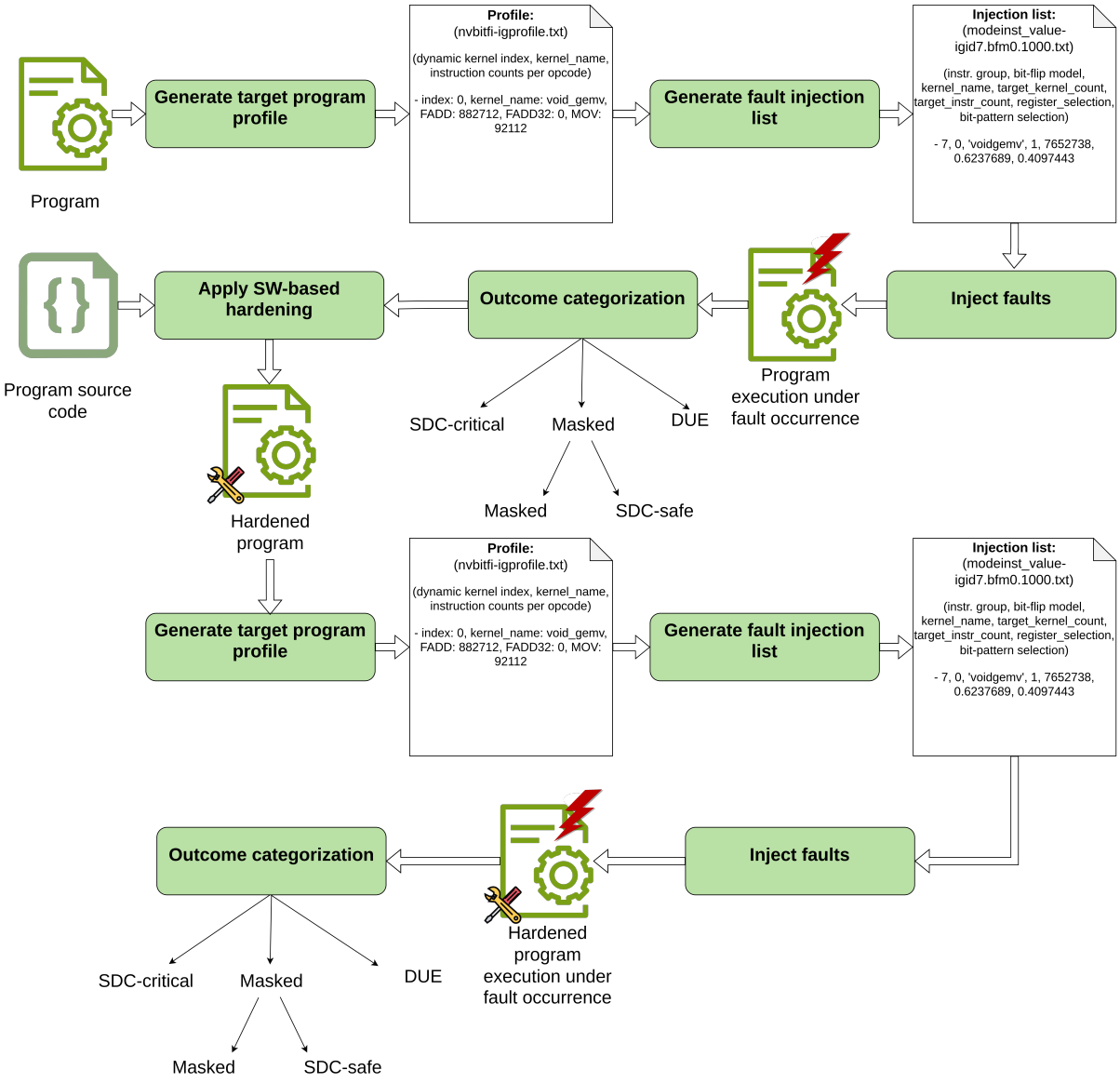


Figure 4.2: Reliability assessment and software-based hardening workflow

With all the scripts and parameters mentioned in section 4.2 set up, it is now possible to reproduce the experiments done for this Master's thesis. Figure 4.2 depicts a general scheme of the flow used to evaluate the reliability of the HSI application and improve its resilience by developing a hardening mechanism. In particular, the framework works as follows: first, the profiler will build a profile of the application in a fault-free scenario,

then the injector, based on the profile, will randomly select locations in the application for corruption. Next, transient fault injections corrupt the program at the previously chosen locations (e.g., register files). Then, based on the outcome, NVBitFI will categorize the corruption as either Masked, SDC or DUE.

As described in section 4.3, NVBitFI will label some outcomes as masked even though there are changes at the output (i.e., the logits) of the HSI classifier. The reason for this is because NVBitFI looks only at the final classification, which can be the same as the golden one under different logits value (see 3.1.2). To account for this, a post-processing step was implemented to extract SDC-safe outcomes from the ones initially labeled as masked.

The next step of this workflow consists of applying a software-based hardening technique to reduce the susceptibility of the HSI classifier to SDC-critical outcomes. In detail, I used the most critical SDC cases from the fault characterization as main target candidates for hardening.

Finally, the same steps performed previously for reliability assessment are again performed, this time on the hardened model to observe the improvements.

The following chapter describes and discusses the experiments and results of the HSI application's fault characterization and reliability assessment.

Chapter 5

Experimental results

This chapter addresses the experiments conducted and discusses the results obtained. It first presents the experimental settings; then, it shows a description of the application at the opcodes and kernels level obtained by making use of the profile generated by the profiler of NVBitFI. Next, it analyzes the impact of transient faults on the systems, highlighting the percentages of the simulations leading to each possible outcome categorization. Then, based on the analysis conducted, it was possible to identify the part of the HSI classifier that is the most sensitive to faults.

Lastly, a software-based hardening was implemented, and its effects are presented.

5.1 Setting of experiments

5.1.1 GPUs

The experiments in this work have been carried out on two different NVIDIA GPUs: GeForce GTX1050 and GeForce RTX3060TI. The GTX1050 is a low-performance GPU, similar to what can be seen in edge AI devices. On the other hand, the RTX3060TI is a mid-range GPU targeted for computationally intensive applications. The two GPUs differ in both computational power and architecture, making it possible to understand what impact transient faults have on different hardware and whether common patterns can be seen among different devices.

5.1.2 Datasets

The fault characterization has been performed on the HSI classifier while making inferences on the three datasets used by De Lucia et al. [2023] in their work: Indian Pines, Salinas, and Pavia University. The reason for analyzing different datasets is that a program, in general, behaves differently under different inputs, i.e., it may use different code structures and their equivalent parts of the hardware depending on the input. In particular, a preliminary analysis of the datasets shows that Indian Pines is much smaller compared to the other two. However, on Indian Pines dataset, the pre-processing step executes more dynamic kernels than the other two datasets when the PCA configuration is the same.

5.1.3 PCA configurations

To analyze the effects of transient faults under different application configurations, I decided to test the HSI classifier considering different numbers of principal components for the PCA pre-processing step. Changing the number of principal components changes the amount of dynamic kernels being executed by the program, hence possibly increasing or reducing reliability. I focus on the three main ranges of the PCA configuration that might affect or benefit the reliability of the HSI classifier: one with a compacted and reduced number of components (PCA 7), one with an average and representative amount of components (PCA 10), and one with a large number of components (PCA 50).

5.1.4 Targeted instruction groups

In this work, the fault injection campaigns targeted two instruction groups: the floating point operations in 32 bits (G_FP32) and the instructions writing on the general purpose registers (G_GP).

5.2 Profile analysis

PCA	GPU	Dataset	Nr. PCA kernels	Nr. CNN kernels	Tot. kernels
PCA7	GTX1050	Indian Pines	45,91%	54,09%	4,785
		Pavia University	6,22%	93,78%	28,236
		Salinas	8,49%	91,51%	15,388
	RTX3060TI	Indian Pines	46,51%	53,49%	4,466
		Pavia University	6,62%	93,38%	26,178
		Salinas	9,06%	90,94%	14,295
PCA10	GTX1050	Indian Pines	69,96%	30,04%	9,277
		Pavia University	8,66%	91,34%	31,219
		Salinas	12,38%	87,62%	17,305
	RTX3060TI	Indian Pines	71,74%	28,26%	8,453
		Pavia University	9,88%	90,12%	27,125
		Salinas	13,84%	86,16%	15,088
PCA50	RTX3060TI	Indian Pines	95,68%	4,32%	59,909
		Pavia University	68,60%	31,40%	77,855
		Salinas	82,76%	17,24%	81,704

Table 5.1: Percentages of PCA and CNN dynamic kernels

Before diving into the fault injection campaign results, let us have a look at what kind of information we can gather with the profiler of NVBitFI. Table 5.1 shows the percentages of the total number of dynamic kernels corresponding either to the pre-processing step implemented with PCA or to the CNN part of the HSI classifier. What can be noticed is that across all PCA configurations considered (i.e., PCA7, PCA10 and PCA50), the inference operation of the HSI classifier on the dataset Indian Pines has the majority of its dynamic kernels belonging to the PCA part. Whereas, for Salinas and Pavia University,

most of the dynamic kernels belong to the CNN part. This has to do with the fact that Indian Pines is a significantly smaller dataset compared to the other two and so, the convolutional network has a smaller workload, hence fewer kernels. The GS-PCA algorithm consists of two nested loops with the outer one having a number of iterations equal to the number of principal components chosen (i.e., in this table, 7, 10 or 50). however, the inner loop strongly depends on the dataset, since the loop performs an undefined number of iterations until an error margin is met and judging by the percentages provided in Table 5.1, Indian Pines always needs more inner loop iterations to arrive to the adequate error margin.

Figure 5.1 shows a logarithmically scaled representation of the amount of dynamic opcodes executed by the application during inference on the Salinas dataset. Figure 5.2 provides a more fine-grained overview, containing a plot for each instruction group illustrating the number of times each opcode has been executed during inference. According to the results, the HSI application uses the opcodes BRA, FFMA, XMAD, LDG, LDS, DEPBAR, and MOV most frequently.

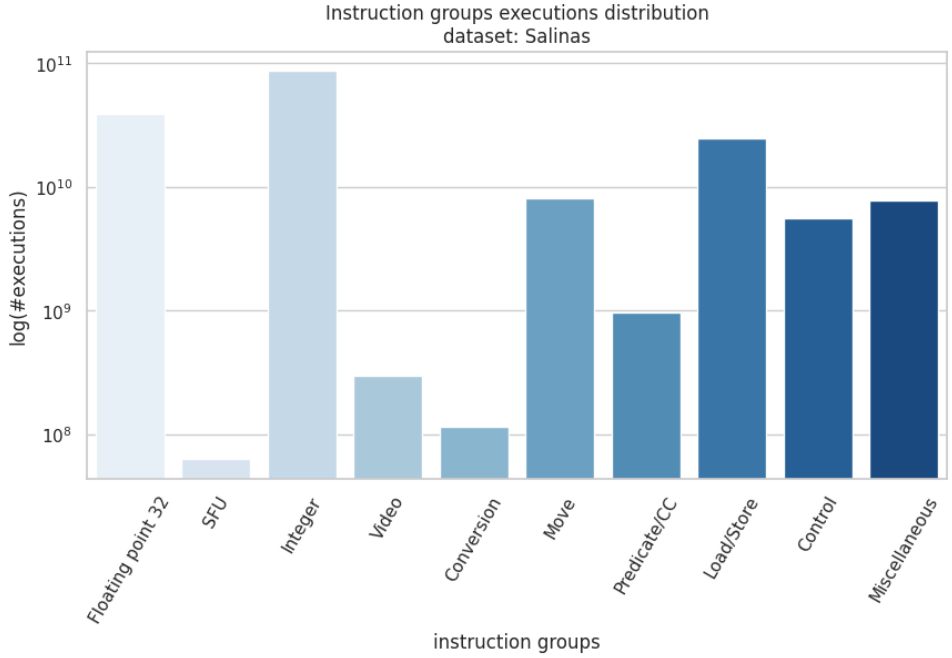
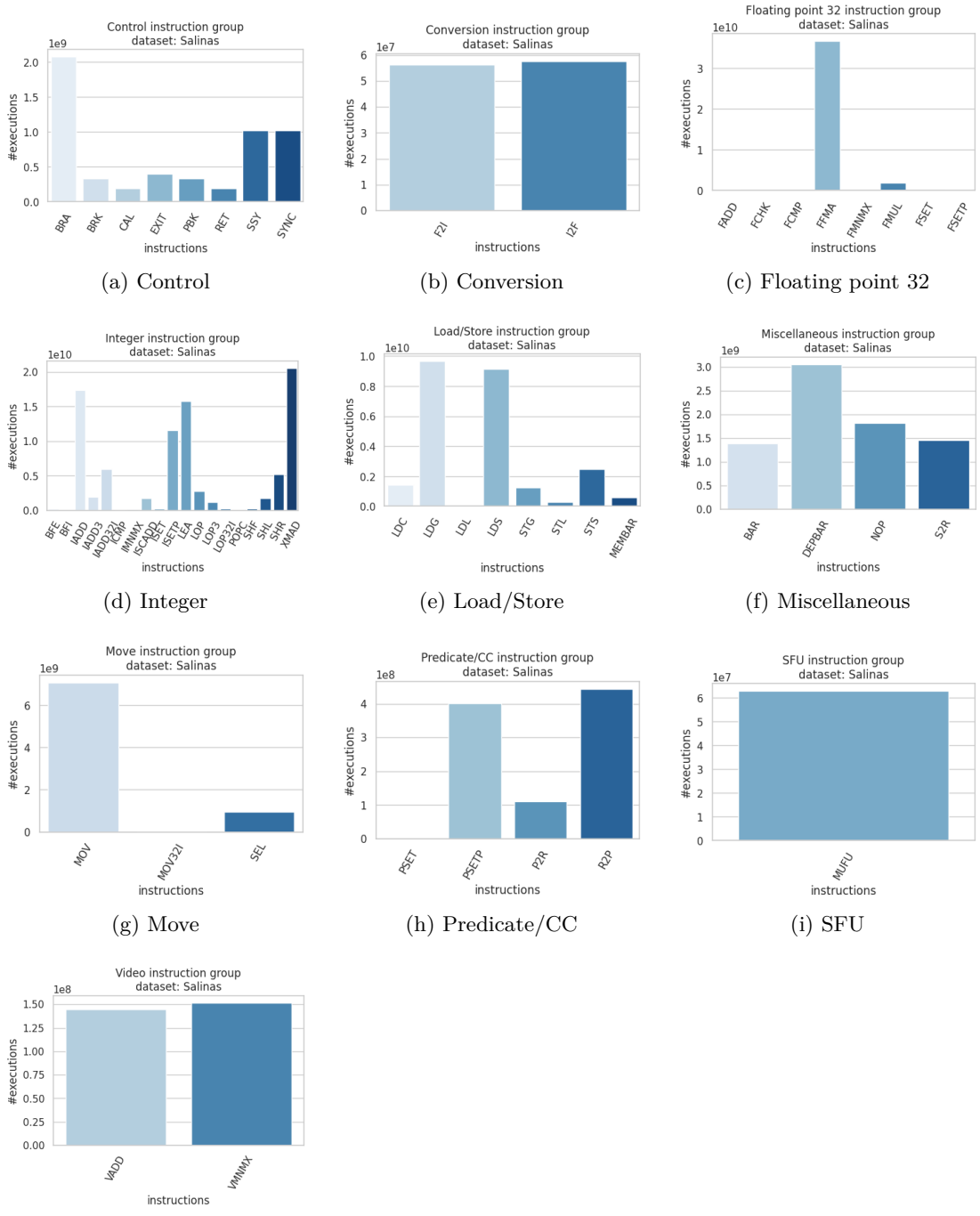


Figure 5.1: Instruction groups executions distribution. Dataset: Salinas

Figure 5.3 shows a logarithmically scaled representation of the amount of dynamic kernels executed per each static kernel. A figure like this helps us understand which kernels are very likely to be targeted by the injector for simulating corruptions, since the injector selects the fault sites under a uniform distributions, meaning that dynamic kernels have the same probability of being chosen. This also means that a static kernel with a large number of corresponding dynamic kernels have a high likelihood of being selected.



(a) Control (b) Conversion (c) Floating point 32

(d) Integer (e) Load/Store (f) Miscellaneous

(g) Move (h) Predicate/CC (i) SFU

(j) Video

Figure 5.2: Dynamic opcodes per instruction group. Dataset: Salinas

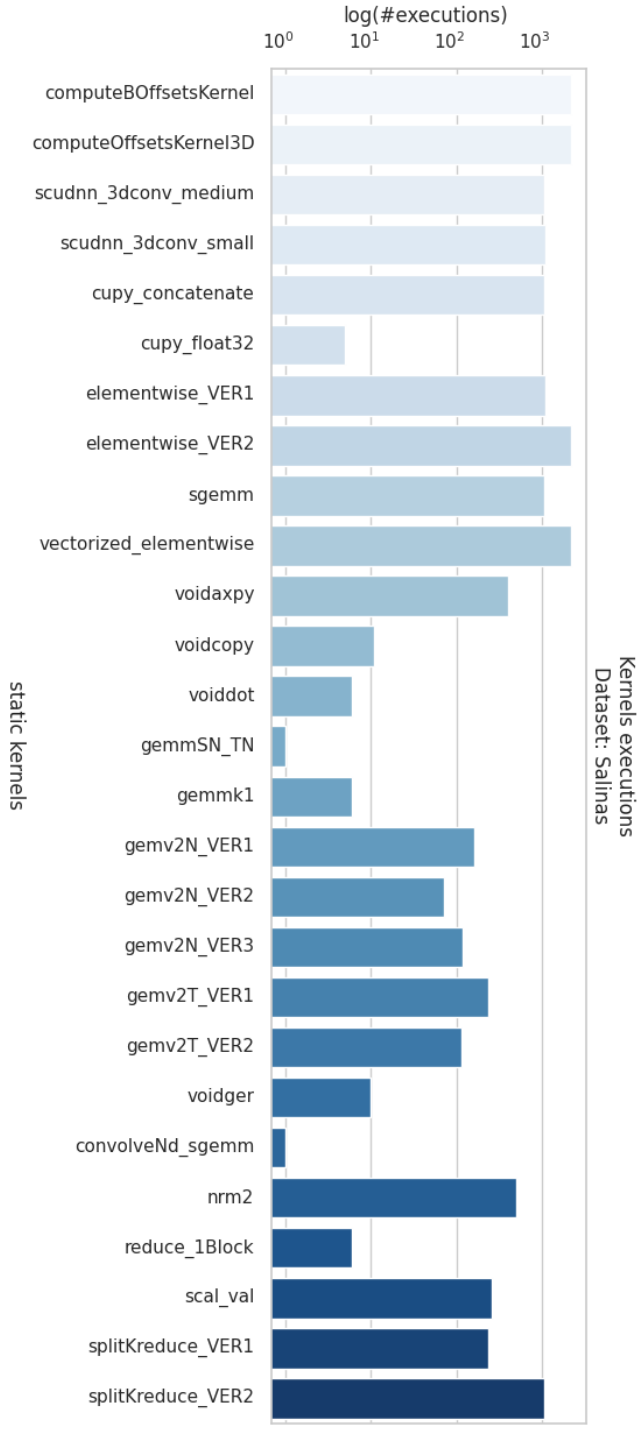


Figure 5.3: Kernels executions. Dataset: Salinas

Finally, figure 5.4 reports the amount of dynamic opcodes occurring during the execution of the dynamic kernels of the kernels *cudnn_maxwell_scudnn_128x32_3dconv_fprop_medium_nn_v0* and *voidgemv2T_kernel_val_VER1* (suffixes such as *VER1* were appended to the names of some kernels to differentiate between kernels with the same name but different prototypes). These two kernels are among the most representative ones in our study, as it will be discussed in the next section.

5.3 Structural analysis of HSI classifier under transient faults

			Instruction groups		
PCA size	GPU	Dataset	Functional units (G_FP32)	Register file (G_GP)	Total
PCA7	RTX3060TI	Pavia University	1000	1000	2000
		Salinas	1000	1000	2000
		Indian Pines	1000	1000	2000
	GTX1050	Pavia University	200	200	400
		Salinas	200	200	400
		Indian Pines	1000	1000	2000
PCA10	RTX3060TI	Pavia University	1000	1000	2000
		Salinas	1000	1000	2000
		Indian Pines	1000	1000	2000
	GTX1050	Pavia University	200	200	400
		Salinas	200	200	400
		Indian Pines	1000	1000	2000
PCA50	RTX3060TI	Pavia University	1000	1000	2000
		Salinas	1000	1000	2000
		Indian Pines	1000	1000	2000

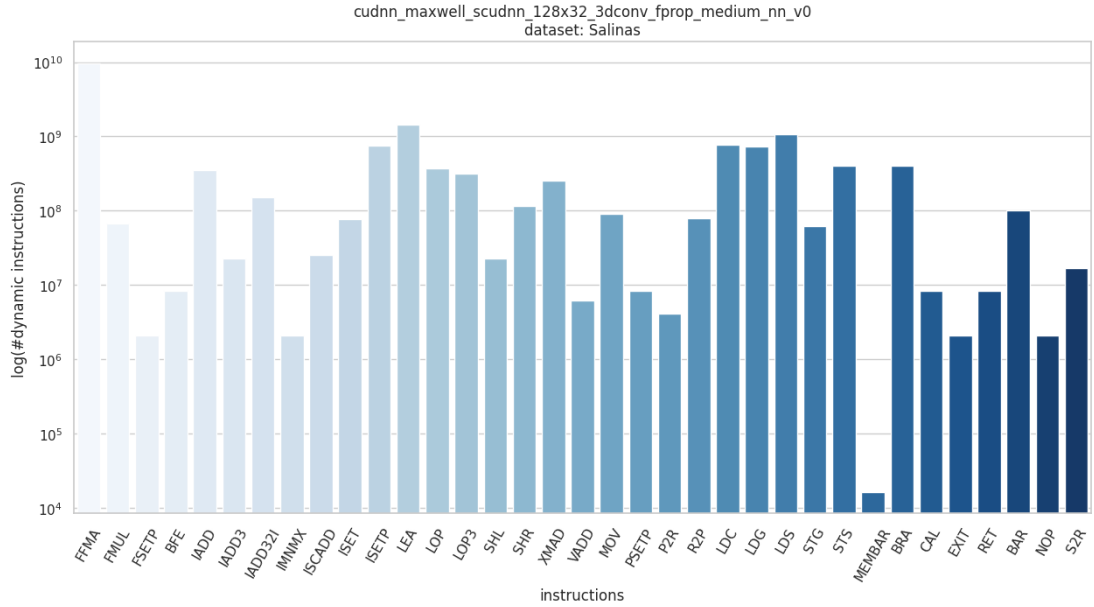
Table 5.2: Experimental configurations of fault injections campaigns

This section provides the analysis of the effects of transient faults on the application, primarily focusing on the trends of SDC-critical cases among the kernels.

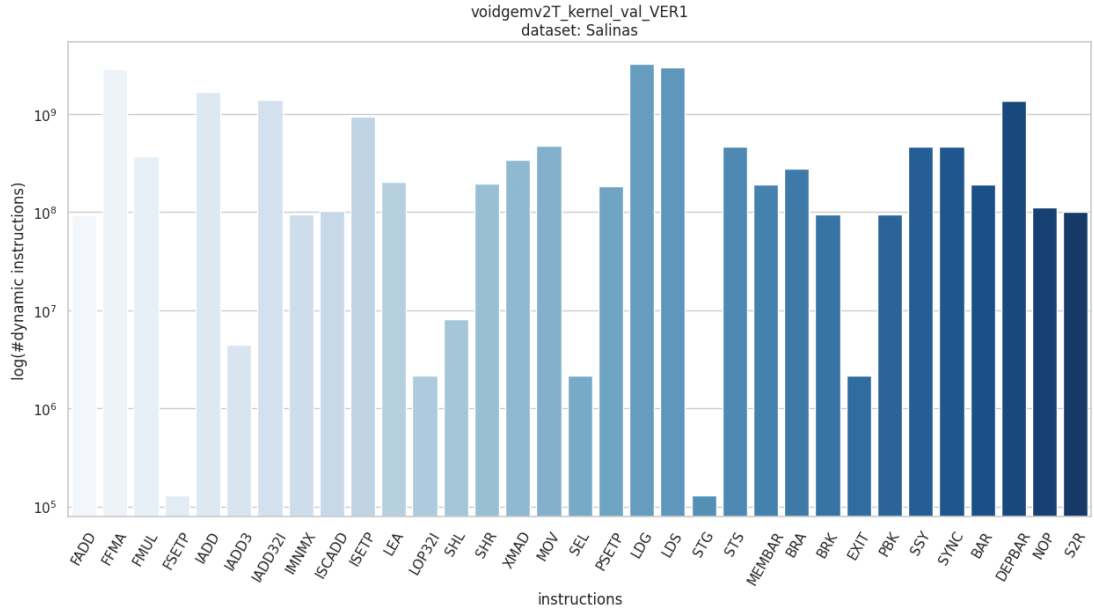
Table 5.2 reports the fault injection simulations performed for each configuration of the type (PCA components, GPU, Dataset, Instruction group). Some configurations consisted of 200 fault injections instead of 1000 because the system using the GTX1050 faced some thermal cooling issues after 200 simulations when working on the larger datasets (i.e., Pavia University and Salinas), making all the simulations after the first 200 leading to DUEs due to timeouts. It became impractical to perform more than 200.

5.3.1 SDC-safe and SDC-critical study at opcode level

Figure 5.5 describes the amount of opcodes leading to SDC-safe and SDC-critical when targeted for transient fault injection during inference on Indian Pines dataset. The static



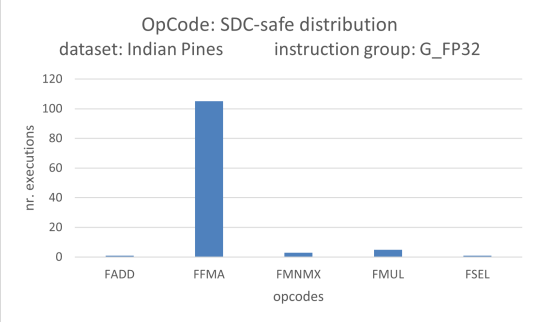
(a) Convolution



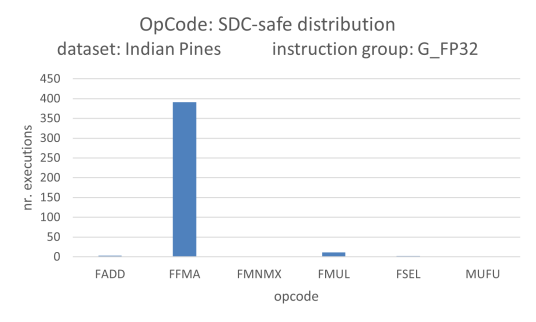
(b) GEMV

Figure 5.4: Opcodes executions per kernel for two of the most representative kernels. Dataset: Salinas

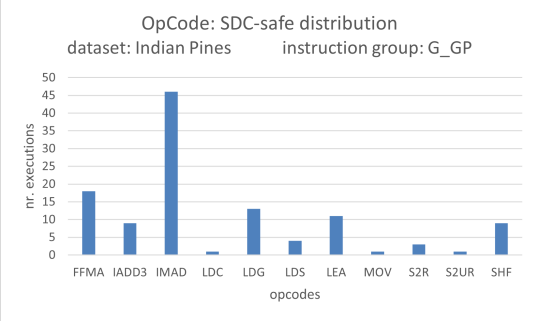
opcodes contributing to the most SDC-safe and SDC-critical outcomes are FFMA, IADD3, IMAD, LDG and LEA. FFMA, IMAD and IADD3 exhibit a similar trend in both SDC-safe and SDC-critical. That is, one cannot really say about one of them that it contributes more to SDC-critical outcomes than to SDC-safe or vice-versa. LDG, on the other hand, seems to be more prone to SDC-critical outcomes when targeted by faults. Lastly, LEA and SHF contribute with a higher percentage in the SDC-safe outcomes compared to the percentages of SDC-critical for which they are responsible.



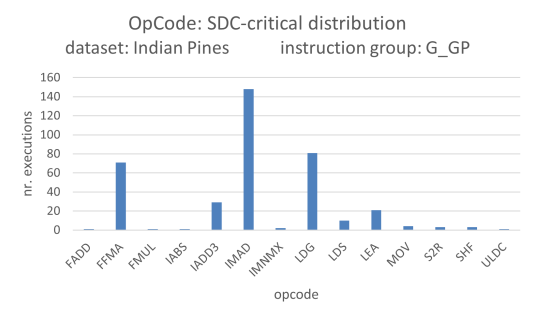
(a) SDC-safe; Instruction group: G_FP32



(b) SDC-critical; Instruction group: G_FP32



(c) SDC-safe; Instruction group: G_GP



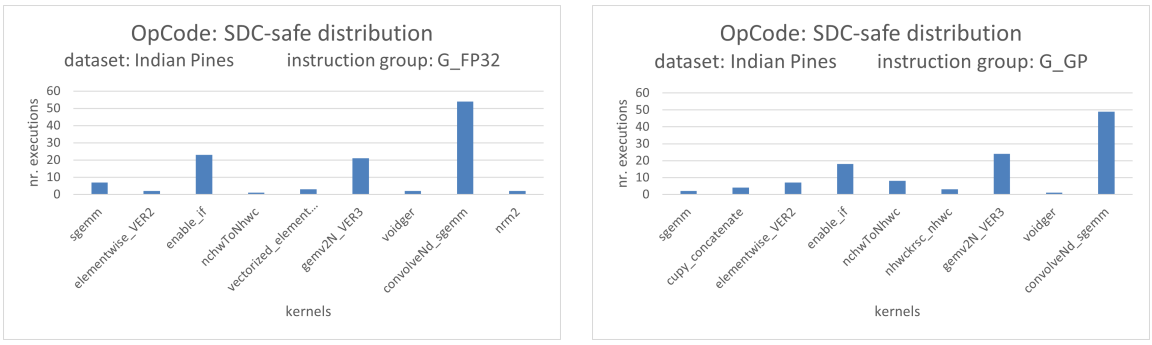
(d) SDC-critical; Instruction group: G_GP

Figure 5.5: Opcodes targeted that led to SDCs. Dataset: Indian Pines

5.3.2 SDC-safe analysis at kernel level

This subsection very briefly discusses the trends of SDC-safe across the kernels. We can see in figure 5.6 the contribution of various kernels to the SDC-safe outcomes. In particular, *convolveNd_sgemm* contributes to a high amount of SDC-safe and it belongs to the CNN part of the HSI classifier pipeline. The *enable_if* and *gemv2N_VER3* kernels belong to the pre-processing stage (PCA) part of the application.

Since SDC-safe effects do not have any effects on the final classification results, this work prioritized the further analysis of SDC-critical.



(a) G_FP32 (b) G_GP

Figure 5.6: Kernels targeted that led to SDC-safe. Dataset: Indian Pines

5.3.3 Program Vulnerability Factor (PVF)

PVF tells us how frequently a fault may lead to a failure or wrong result in an application. As it can be seen in figure 5.7, PVF varies depending on the input to the application and the configuration of the pre-processing step.

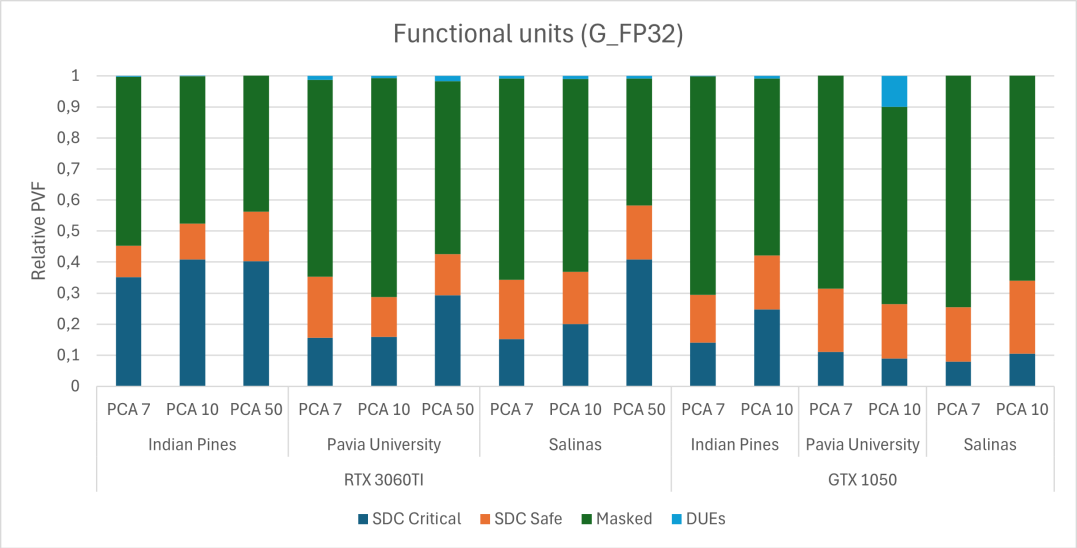
In the case of PCA 10 under fault injections at the register file (G_GP) of RTX3060TI, the HSI classifier sees relative PVFs of SDC-critical outcomes of 0.376, 0.218 and 0.174 for Indian Pines, Salinas and Pavia University, respectively. When injecting faults at the functional units, we see relative PVFs of 0.409, 0.200 and 0.159 for Indian Pines, Salinas and Pavia University, respectively.

The trend that can be noticed in figure 5.7 is that the number of SDC-critical outcomes is larger when working on Indian Pines dataset compared to the other two ones. This is because Indian Pines is a much smaller dataset and because of this, the profile of the application for this specific dataset will consist mostly of kernels belonging to PCA algorithm and of fewer kernels corresponding to the CNN part. For the other two datasets, the profile consists mostly of kernels of CNN and the minority of them belong to PCA.

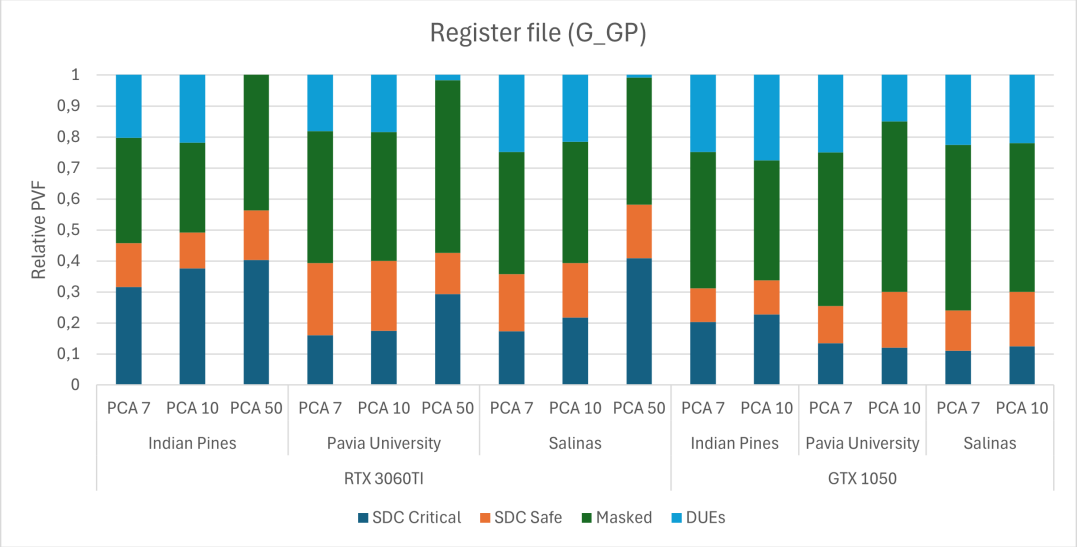
5.3.4 Mean Execution Between Failure (MEBF)

According to Figures 5.8, the MEBF shows, for each dataset, the impact of the execution time and the corruption effects of faults (i.e., higher values are better for resilience). As it was seen in Figure 5.7 (where lower PVF is better), inference on Indian Pines is significantly more sensitive compared to the other two datasets. However, in Figure 5.8(a), after factoring in the execution time, the corruption effects described by MEBF on each dataset have values closer to each other. The reason for this is because even though inference on Indian Pines sees a higher number of critical SDCs, being a smaller dataset makes the HSI classifier perform the task in a shorter amount of time than in the case of the other two datasets (especially at the CNN part).

Figure 5.8(b) depicts MEBF when factoring in the execution time of the PCA step only, excluding the workload of the neural network. In this case it can be seen again



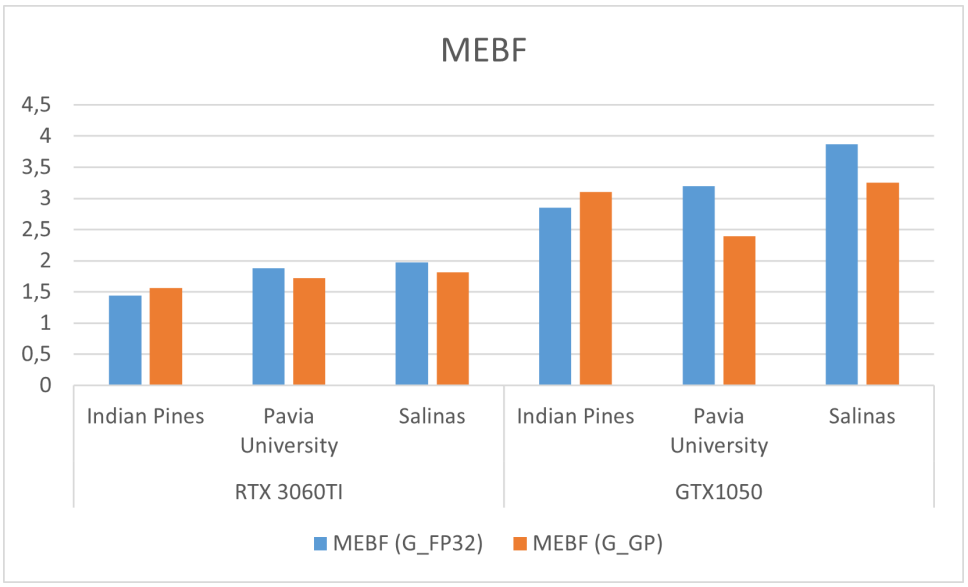
(a) Functional units



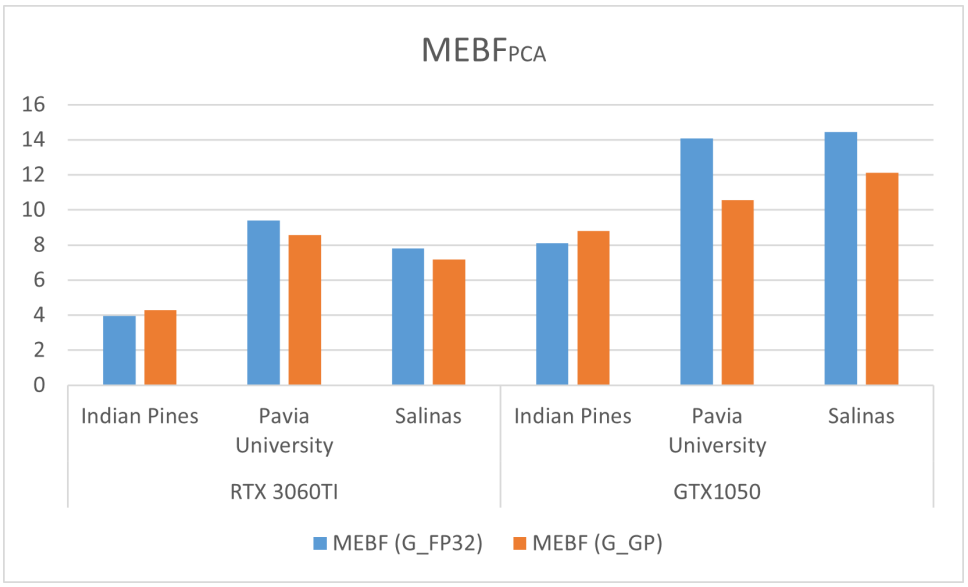
(b) Register file

Figure 5.7: Relative PVF

the high discrepancy between the inferences on the three datasets, as originally seen in the PVF plots in Figure 5.7. We see again this discrepancy because the PCA step of the inference process has an execution time that does not differ by a lot across the three datasets. For GTX1050, the execution times for PCA are 0.5 , 0.66 and 0.79 seconds for Indian Pines, Salinas and Pavia University, respectivelty. Whereas, the execution times of the entire application are 1.42, 2.46 and 3.48 seconds for Indian Pines, Salinas and Pavia



(a) Whole application



(b) PCA only

Figure 5.8: MEBF on (a) complete application; (b) PCA part only

University. For PCA, the difference between Pavia University and Indian Pines is only 0.29 seconds, whereas for the entire application execution the difference is 2.06 seconds (working on Pavia University takes more than twice as long as on the smallest one).

These results show the importance of considering the execution time of a program

when assessing its reliability. To sum up, even though when working on Indian Pines the HSI classifier sees more frequent critical SDCs (as seen in Figure 5.7), the fact that the execution time on this dataset is significantly lower than on the other two makes the number of executions of the program between two occurrences of critical SDCs almost the same as for the other two datasets.

5.3.5 Classification corruption examples

As mentioned in 4.3, SDC-critical are corruptions that propagate to the final classification, affecting the HSI classifier's performance operation. For the most part, the transient faults cause minimal corruption, in which just a few pixels are wrongly labeled. However, depending on the scenario, just one-pixel misclassification can cause a loss of resources and money (e.g., one pixel could correspond to a large field portion of valuable crops, and so wrongly detecting a lack of water in this can cause the irrigation system to flood the crops and wastewater).

Figures 5.9, 5.10, 5.11 report some of the more extreme outcomes in which a high percentage of pixels have been misclassified. In figure 5.9 we see a drastic drop in accuracy from 96.66% to 34.08%, leading to the loss of diagonality in the confusion matrix. The confusion matrix tells us how the misclassified samples are being labeled and generally, a good classifier has a diagonal confusion matrix.

In figure 5.10 we see again a huge drop of accuracy of 59.30% causing some loss of diagonality. Looking at the images, we see entire fields being misclassified.

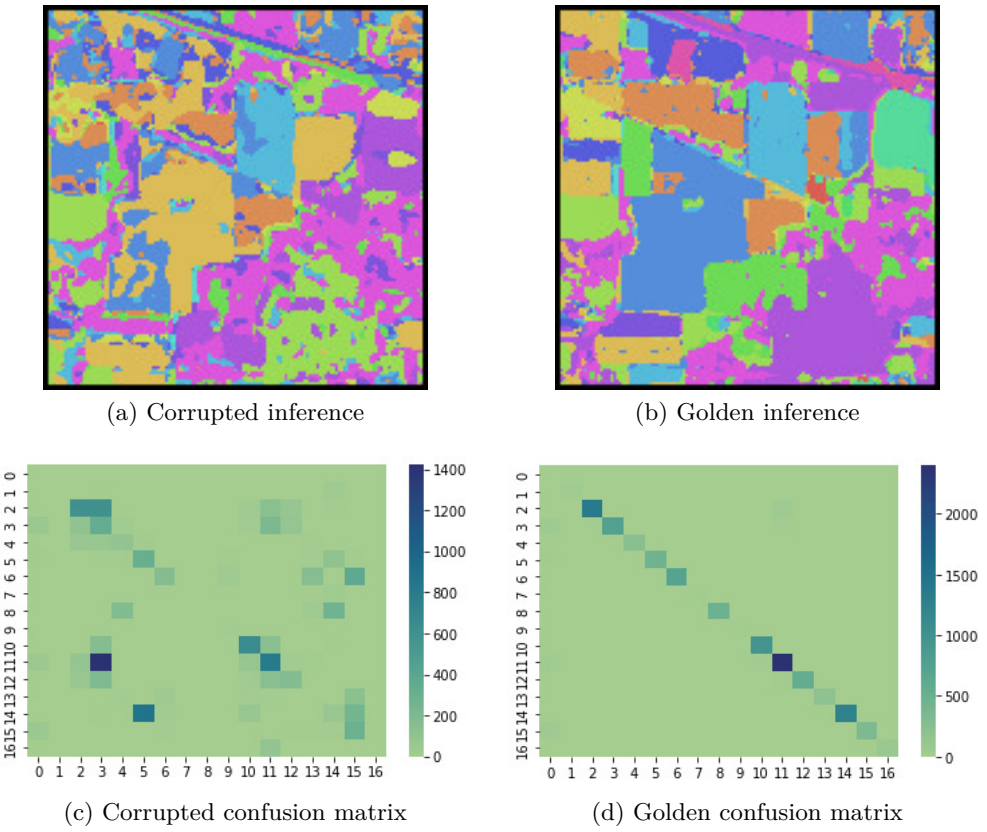
5.3.6 Effects of transient faults on different PCA configurations

After obtaining the results from the simulations run on the application using PCA 10, it was decided to further investigate the effects of transient faults on the same model using fewer principal components for projecting the datasets. More specifically, I tested the PCA 7 and PCA 50 cases.

According to the experimental results, I anticipate that the PCA part of the HSI application is critical and is the most sensitive part of the HSI classifier.

As it can be seen in the figure 5.7, the general trend seems to be that the larger the number of principal components used is, the higher the amount of SDC-critical is. This makes sense because the PCA algorithm used by De Lucia et al. [2023] is an iterative algorithm with the number of the outer loop iterations being equal to the number of principal components to be used. NVBitFI works by first generating a profile containing all the kernel calls done by the program and then randomly selecting one kernel call to corrupt for each fault performed. In the case of PCA 7, the profile will contain fewer kernel calls corresponding to the PCA algorithm compared to PCA 10 and 50 and so it contains fewer sensitive kernels.

Even though PCA 7 seems to be more fault tolerant, it comes at the cost of lower performance of the classifier, specifically on the dataset Salinas. On this specific case, a drop of around 23% in accuracy can be seen without any fault injections. One can conclude that PCA 10 is the best trade-off in terms of fault tolerance and performance of the application.



(a) Corrupted inference

(b) Golden inference

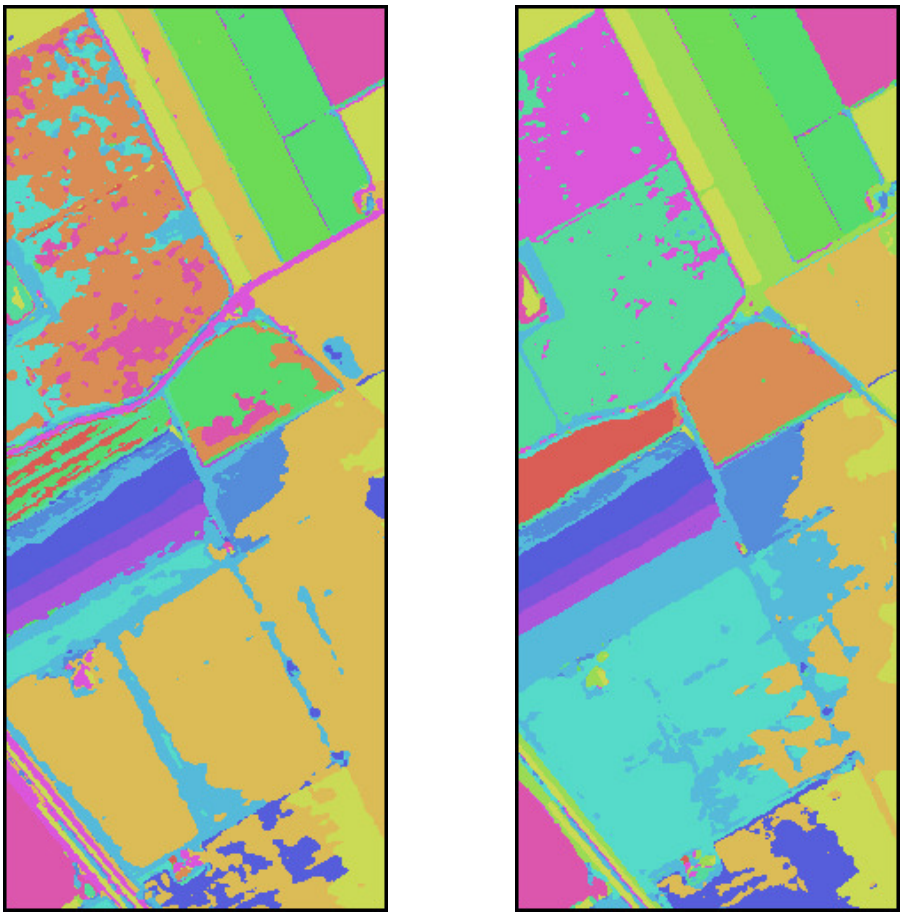
(c) Corrupted confusion matrix

(d) Golden confusion matrix

Figure 5.9: Example Indian Pines faulty outcome (accuracy drop: 62.58%)

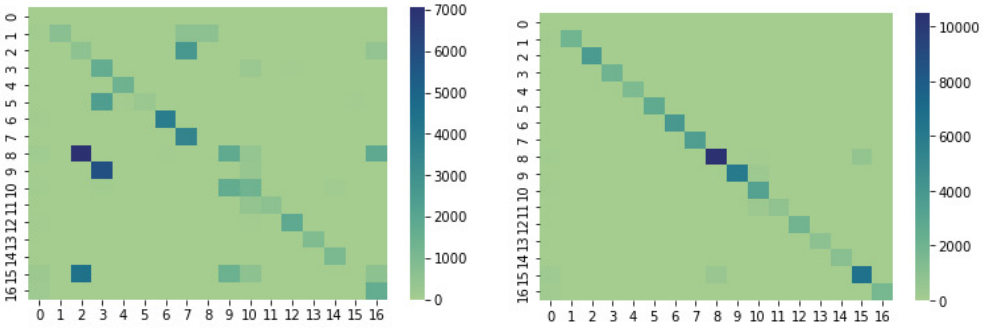
One strange observation when looking at figure 5.7 is that across all configurations, RTX3060TI experienced a higher amount SDC-critical compared to GTX1050, despite the general belief that larger GPUs (in terms of number of streaming multiprocessors) have a higher fault tolerance than smaller ones. The general belief relies on the fact that a higher number of streaming multiprocessors means that there will be a smaller amount of dynamic kernels sharing the same physical streaming multiprocessor and so, if a hardware fault occurs in one streaming multiprocessors, the fault won't propagate to many dynamic kernels.

In this analysis this trend cannot be seen because the two GPUs have different architectures and because of this, the compilers compile the source code in different manners. Even though the CNN part of the classifier is implemented using the same PyTorch functions, the kernels being executed in the GPUs differ between the two architectures. The same happens with the pre-processing step. The two GPUs use the same cuBLAS implementation of PCA, but when looking at the profiles, there are many kernels that are present only in one of the two devices. More specifically, RTX3060TI uses a kernel called *enable_if* that is not present in the profile of GTX1050 and this specific kernel is responsible for the majority of SDC-critical outcomes seen in the analysis of the larger



(a) Corrupted inference

(b) Golden inference

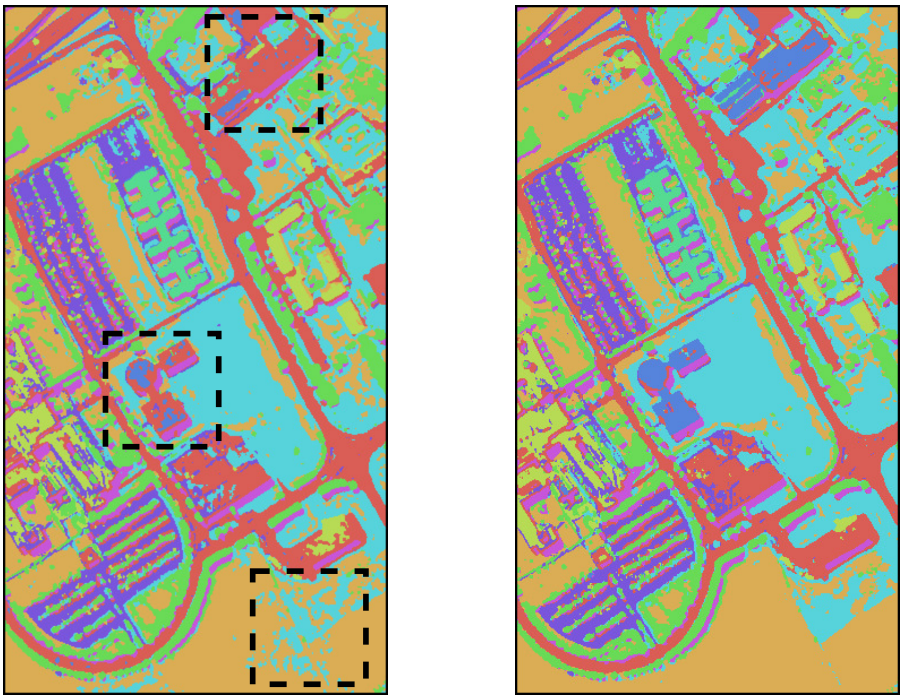


(c) Corrupted confusion matrix

(d) Golden confusion matrix

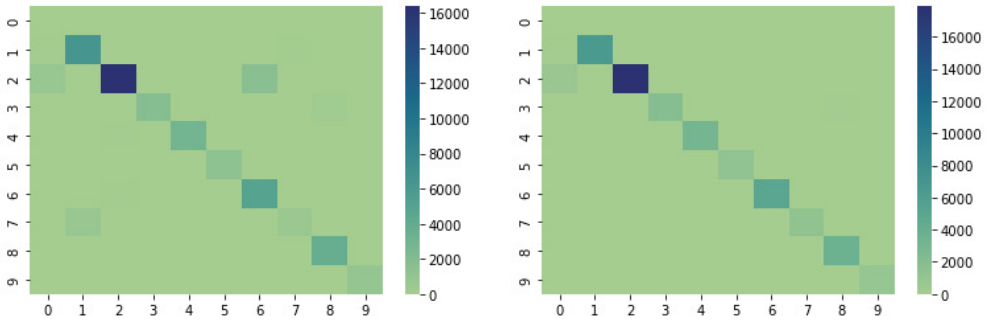
Figure 5.10: Example of Salinas faulty outcome (accuracy drop: 59.30%)

GPU.



(a) Corrupted inference

(b) Golden inference



(c) Corrupted confusion matrix

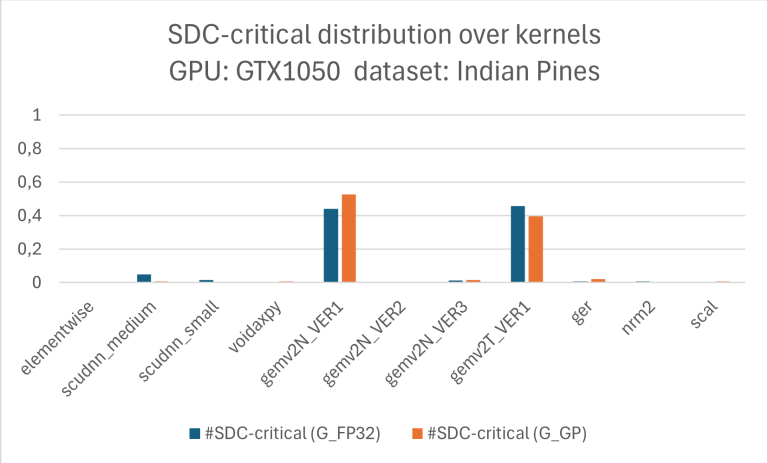
(d) Golden confusion matrix

Figure 5.11: Example of Pavia University faulty outcome (accuracy drop: 6.90%)

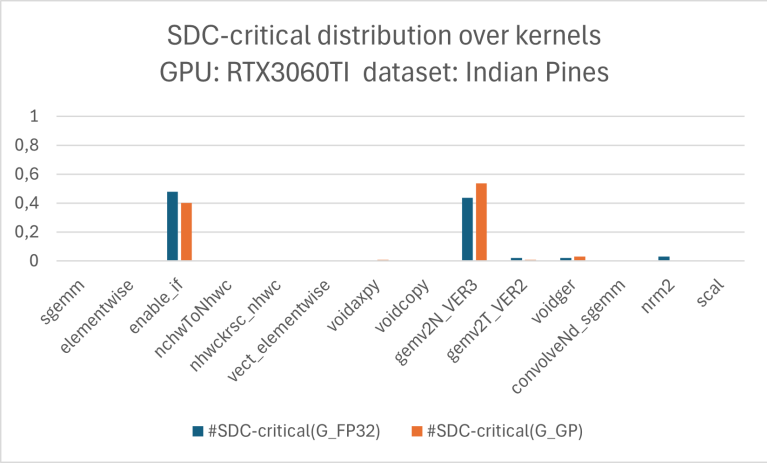
5.3.7 Identification of SDC-critical prone kernels

When focusing on the occurrences of SDC-critical for each kernel, it becomes clear that some kernels are more susceptible to SDC-critical than other. Looking at figures 5.12 and 5.13, one can observe that the most sensitive kernels are *gemv2N*, *gemv2T*, *enable_if* and *gemvNSP*. The common characteristic of all these four kernels is that they correspond to the PCA part of the HSI classifier pipeline. Figure 5.14 reports the source code snippet of the pre-processing step making use of the general matrix-vector product algorithm implementation available in cuBLAS. It can be concluded that this part of the application

is a good candidate for hardening in order to significantly increase fault tolerance.



(a) Indian Pines, GTX1050

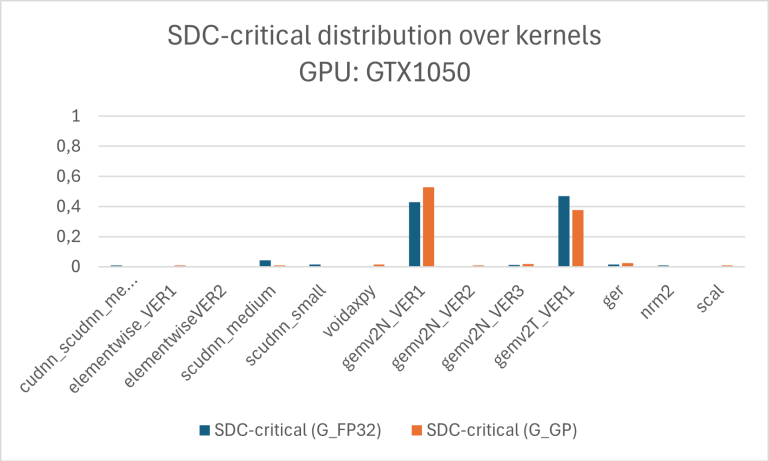


(b) Indian Pines, RTX3060TI

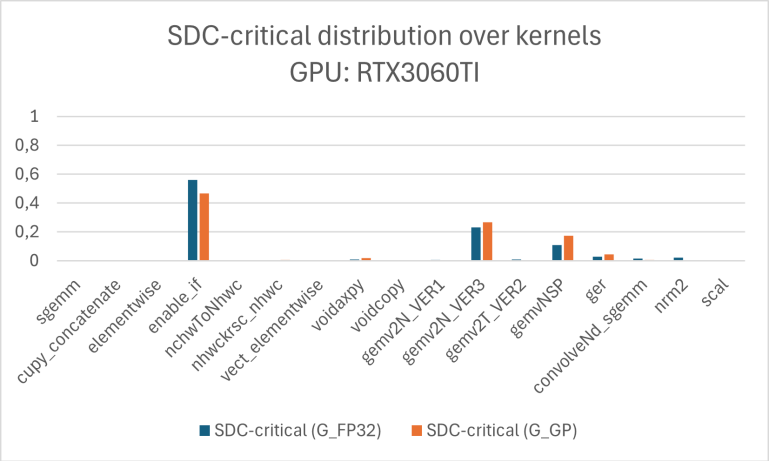
Figure 5.12: SDC-critical distribution over kernels for Indian Pines. Each plot depicts two different distributions: one for register files and one for functional units

5.4 Hardening

Given the findings stated at the end of the previous section, I applied a hardening technique on the cuBLAS implementation of PCA using a method based on Duplication with Comparison (DwC). DwC is actually a fault detection mechanism, meaning that it can only determine whether a fault has occurred and it does so by executing the same module twice and checking if the two executions gave different outputs. Based on this information,



(a) Across all datasets, GTX1050



(b) Across all datasets, RTX3060TI

Figure 5.13: SDC-critical distribution over kernels. Each plot depicts two different distributions: one for register files and one for functional units

a fault masking technique can be implemented by triggering a third execution of the same module when the first two executions have different results.

		Before hardening				After hardening			
Datasets	**Instruction Group**	**SDC-critical**	**SDC-safe**	**Masked**	**DUE**	**SDC-critical**	**SDC-safe**	**Masked**	**DUE**
Indian Pines	G_GP	376	116	290	218	136	101	550	213
	G_FP32	409	115	475	1	273	161	561	5
Pavia University	G_GP	174	226	416	184	130	150	526	194
	G_FP32	159	128	705	8	176	151	668	5
Salinas	G_GP	218	175	391	216	147	102	532	219
	G_FP32	200	169	621	10	201	179	614	6

Table 5.3: Results before and after hardening for PCA 10 on RTX3060TI

```
void KernelPCA::fit_transform(int M, int N, float *R, bool verbose, float* imgT)
    // GS-PCA
    float a;
    for(k=0; k<K; k++)
    {
        cublasScopy (M, &dR[k*M], 1, &dT[k*M], 1);
        a = 0.0;
        for(j=0; j<J; j++)
        {
            cublasSgemv ('t', M, N, 1.0, dR, M, &dT[k*M], 1, 0.0, &dP[k*N], 1);
            if(k>0)
            {
                cublasSgemv ('t', N, k, 1.0, dP, N, &dP[k*N], 1, 0.0, dU, 1);
                cublasSgemv ('n', N, k, -1.0, dP, N, dU, 1, 1.0, &dP[k*N], 1);
            }
            cublasSscal (N, 1.0/cublasSnrm2(N, &dP[k*N], 1), &dP[k*N], 1);
            cublasSgemv ('n', M, N, 1.0, dR, M, &dP[k*N], 1, 0.0, &dT[k*M], 1);
            if(k>0)
            {
                cublasSgemv ('t', M, k, 1.0, dT, M, &dT[k*M], 1, 0.0, dU, 1);
                cublasSgemv ('n', M, k, -1.0, dT, M, dU, 1, 1.0, &dT[k*M], 1);
            }

            L[k] = cublasSnrm2(M, &dT[k*M], 1);
            cublasSscal(M, 1.0/L[k], &dT[k*M], 1);

            if(fabs(a - L[k]) < er*L[k]) break;

            a = L[k];

        }

        cublasSger (M, N, - L[k], &dT[k*M], 1, &dP[k*N], 1, dR, M);
```

Figure 5.14: cuBLAS implementation of PCA

Datasets	Instruction Group	SDC-critical drop (%)	SDC-safe drop (%)	Masked increase (%)	DUE drop
Indian Pines	G_GP	63,83%	12,93%	89,66%	5
	G_FP32	33,25%	-40,00%	18,11%	-4
Pavia University	G_GP	25,29%	33,63%	26,44%	-10
	G_FP32	-10,69%	-17,97%	-5,25%	3
Salinas	G_GP	32,57%	41,71%	36,06%	-3
	G_FP32	-0,50%	-5,92%	-1,13%	4

Table 5.4: Hardening results for PCA 10 on RTX3060TI

Tables 5.3 and 5.4 show the results of hardening. For the majority of cases, hardening led to improved fault tolerance. Out of the six experimental settings, four of them see a significant drop in occurrence of SDC-critical (i.e., 63.83%, 33.25%, 25.29% and 32.57%). The other two, namely [Salinas, G_FP32] and [Pavia University, G_FP32] either see virtually no changes or see actual worse results.

When injecting transient faults on the functional units while working on the Salinas dataset, the number of SDC-critical increased by 1 after applying hardening. On the same instruction group but on Pavia University dataset, the amount of SDC-critical actually

increased by 10.69% after hardening, i.e., an increase of 17 more corrupted classifications.

The general trend of these results seems to be that hardening is more successful when the target for transient faults is the register file compared to functional units. When focusing on the register file alone, the HSI classifier sees improvements in terms of fault tolerance across all three datasets. Not only have the SDC-critical decreased, but also the masked outcomes have increased by as much as 89.66%. The functional units group, on the other hand, experiences improvements only in the case of Indian Pines.

The effectiveness of hardening seems to be dependent also on the dataset tested. On Indian Pines, which is the smallest of the three analyzed, it experiences the highest improvement with a decrease of SDC-critical and SDC-safe by 63.83% and 12.93%, respectively, and an increase of masked outcomes by 89.66%. On Salinas, which is in the middle of the other two datasets in terms of size, hardening provides significant improvements when the target of the fault injection campaign is the register files, but has almost no effect on functional units. Finally, hardening gives significant improvements on Pavia University on register files (albeit, not as much as on the other two datasets), but it worsens the fault tolerance of the functional units.

Chapter 6

Conclusions

This work assessed and evaluated the reliability of a Hyperspectral Imaging (HSI) application comprising a pre-processing step (PCA) and a neural network based inference step (3D Convolutional Neural Network), considering transient fault effects arising from the underlying hardware architecture of two system configurations. To simulate these faults, I used a software-based fault injection tool developed by NVIDIA called NVBitFI that simulates faults in the hardware by corrupting the destination register of assembly code instructions. The tool is able to corrupt a program on-the-fly, i.e., during the execution it intercepts the instruction (opcode) targeted and applies a bit-flip model on the result.

By means of a profiler, NVBitFI is capable to analyze the underlying dynamic kernels and opcodes executed when running the classifier and reports all of these information in a file called profile. Based on the profile, it was possible to discover that some kernels and opcodes are more frequently used than other. For example, *FFMA*, *XMAD*, *IADD*, *LDG* and so on are the most frequently executed opcodes, and *computeBOffsetsKernel*, *elementwise* and *gemv2N* are among the most used kernels.

The evaluation of the fault injection campaigns showed several trends across different configurations. When moving from a lower to a higher number of principal components, one can notice that the amount of SDC-critical outcomes (i.e., outcomes in which the fault changed the classification outcome of the HSI model) increases, indicating that the pre-processing step implemented with PCA could be the most sensitive part of the application to faults. This fact is then proven when looking at the number of SDC-critical caused by each static kernel. The results showed that the vast majority of corrupted classifications were caused by just a handful of the static kernels, namely *gemv2N*, *gemv2T*, *enable_if* and *gemvNSP*, all belonging to the PCA part of the classifier.

Another trend discovered is that the sensitivity to transient faults of the inference operation also depends on the input data. On the dataset Indian Pines, which is the smallest among the ones tested, the classifier experiences a larger amount of SDC-critical faults compared to on the other two. On the other hand, the Pavia University dataset (the largest one) shows fewer SDC-critical faults. When looking at the profile corresponding to the inference on each dataset, the number of kernels of the CNN part is larger when the size of the dataset is larger. This pattern also indicates that the pre-processing stage of the classifier is the most sensitive one.

The MEBF results showed the impact of considering the execution time in the analysis of reliability. Looking at PVF, one can see a high discrepancy between the PVFs of inference on Indian Pines and the other datasets, whereas the MEBF values across the three datasets have values that don't differ significantly. However, when considering only the execution time of the PCA stage (which is almost the same for all three datasets) the results once again show a high difference, with Indian Pines showing lower MEBF.

The experiments were performed on two GPU architectures. One peculiarity found during the analysis is that the larger GPU (RTX3060TI) has experienced a larger amount of classification degradation due to transient faults, despite the common trend that larger GPUs generally have higher fault tolerance. The two NVIDIA GPUs are of different generations and so this tells that the architecture of an accelerator can play a crucial role in terms of fault tolerance. Because of the different architectures and generations, the two GPUs use two different sets of kernels, hence with potentially different inherent fault tolerance. One particular kernel encountered only in the RTX3060TI called *enable_if* accounts for the majority of SDC-critical faults seen.

Finally, the hardening technique based on Duplication with Comparison (DwC) fault detection increased the fault tolerance of the HSI classifier for most of the experimental settings (four out of six), where one can see a drop of SDC-critical faults up to 63%. For one of the settings, hardening did not have any effect, while for another, hardening actually caused an increase of SDC-critical faults by 10%.

In future work, further analysis similar to this work should be performed on different kinds of applications making use of the same libraries used by the HSI classifier studied here (i.e., cuBLAS and PyTorch) to understand if similar trends can be noticed across different scenarios and so, potentially improve the fault tolerance of the particularly sensitive functions if the source code is available. Another important aspect to be studied is the analysis of other hardening techniques in increasing the fault tolerance of HSI classifiers.

Bibliography

Abien Fred Agarap. Deep learning using rectified linear units (relu). *CoRR*, abs/1803.08375, 2018. URL http://arxiv.org/abs/1803.08375.

Muhammad Ahmad et al. Hyperspectral image classification—traditional to deep models: A survey for future prospects. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 15:968–999, 2022.

Muhammad Ahmad et al. A comprehensive survey for hyperspectral image classification: The evolution from conventional to transformers, 2024.

Mohammad Hasan Ahmadilivani, Mario Barbareschi, Salvatore Barone, Alberto Bosio, Masoud Daneshtalab, Salvatore Della Torca, Gabriele Gavarini, Maksim Jenihhin, Jaan Raik, Annachiara Ruospo, Ernesto Sanchez, and Mahdi Taheri. Special session: Approximation and fault resiliency of dnn accelerators. In *2023 IEEE 41st VLSI Test Symposium (VTS)*, pages 1–10, 2023.

Mohammad Hasan Ahmadilivani, Alberto Bosio, Bastien Deveautour, Fernando Fernandes Dos Santos, Juan-David Guerrero-Balaguera, Maksim Jenihhin, Angeliki Kritikakou, Robert Limas Sierra, Salvatore Pappalardo, Jaan Raik, Josie E. Rodriguez Condia, Matteo Sonza Reorda, Mahdi Taheri, and Marcello Traiola. Special session: Reliability assessment recipes for dnn accelerators. In *2024 IEEE 42nd VLSI Test Symposium (VTS)*, pages 1–11, 2024.

Mircea Andrecut. Parallel gpu implementation of iterative pca algorithms. *Journal of computational biology : a journal of computational molecular cell biology*, 16:1593–9, 09 2009. doi: 10.1089/cmb.2008.0221.

Gonzalo R. Arce et al. Compressive coded aperture spectral imaging: An introduction. *IEEE Signal Processing Magazine*, 31(1):105–115, 2014.

Iram Tariq Bhatti, Mahum Naseer, Muhammad Shafique, and Osman Hasan. A formal approach to identifying the impact of noise on neural networks. *Commun. ACM*, 65 (11):70–73, October 2022. ISSN 0001-0782. doi: 10.1145/3550492. URL https://doi.org/10.1145/3550492.

Alberto Bosio, Paolo Bernardi, Annachiara Ruospo, and Ernesto Sanchez. A reliability analysis of a deep neural network. In *2019 IEEE Latin American Test Symposium (LATS)*, pages 1–6, 2019. doi: 10.1109/LATW.2019.8704548.

Zitao Chen, Niranjhana Narayanan, Bo Fang, Guanpeng Li, Karthik Pattabiraman, and Nathan DeBardeleben. Tensorfi: A flexible fault injection framework for tensorflow applications. In *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, pages 426–435, 2020.

Jack Choquette, Edward Lee, Ronny Krashinsky, Vishnu Balan, and Brucek Khailany. 3.2 the a100 datacenter gpu and ampere architecture. In *2021 IEEE International Solid-State Circuits Conference (ISSCC)*, volume 64, pages 48–50, 2021.

Josie E. Rodriguez Condia, Boyang Du, Matteo Sonza Reorda, and Luca Sterpone. Flexgripplus: An improved gpgpu model to support reliability analysis. *Microelectronics Reliability*, 109:113660, 2020. ISSN 0026-2714.

Josie E. Rodriguez Condia, Fernando Fernandes dos Santos, Matteo Sonza Reorda, and Paolo Rech. Combining architectural simulation and software fault injection for a fast and accurate cnns reliability evaluation on gpus. In *2021 IEEE 39th VLSI Test Symposium (VTS)*, pages 1–7, 2021.

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 Int. Test Conf. (TC)*, pages 278–287, 2022. doi: 10.1109/ITC50671.2022.00036.

B. Dally. Hardware for deep learning. In *IEEE Hot Chips 35 Symp. (HCS)*, pages 1–58. IEEE Computer Society, 2023.

William J. Dally, Stephen W. Keckler, and David B. Kirk. Evolution of the graphics processing unit (gpu). *IEEE Micro*, 41(6):42–51, 2021.

Gianluca De Lucia, Marco Lapegna, and Diego Romano. Unlocking the potential of edge computing for hyperspectral image classification: An efficient low-energy strategy. *Future Generation Computer Systems*, 147:207–218, 2023. ISSN 0167-739X.

Gianluca De Lucia et al. Towards explainable ai for hyperspectral image classification in edge computing environments. *Comput. Electr. Eng.*, 103:108381, 2022. ISSN 0045-7906.

Bo Fang, Karthik Pattabiraman, Matei Ripeanu, and Sudhanva Gurumurthi. Gpu-qin: A methodology for evaluating the error resilience of gpgpu applications. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 221–230, 2014. doi: 10.1109/ISPASS.2014.6844486.

Bo Fang et al. epvf: An enhanced program vulnerability factor methodology for cross-layer resilience analysis. In *46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 168–179, 2016.

Fernando Fernandes dos Santos, Lucas Draghetti, Lucas Weigel, Luigi Carro, Philippe Navaux, and Paolo Rech. Evaluation and mitigation of soft-errors in neural network-based object detection in three gpu architectures. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 169–176, 2017. doi: 10.1109/DSN-W.2017.47.

Fernando Fernandes dos Santos, Caio Lunardi, Daniel Oliveira, Fabiano Libano, and Paolo Rech. Reliability evaluation of mixed-precision architectures. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 238–249, 2019. doi: 10.1109/HPCA.2019.00041.

github. Nvbitfi installation guide. https://github.com/NVlabs/nvbitfi. Accessed: 2024-11-19.

Dennis Gnad et al. Reliability and security of ai hardware. In *2024 IEEE European Test Symposium (ETS)*, pages 1–10, 2024.

Brunno F. Goldstein et al. Reliability evaluation of compressed deep learning models. In *IEEE 11th Latin American Symposium on Circuits Systems (LASCAS)*, pages 1–5, 2020.

Mukul Anil Gosavi et al. Application of functional safety in autonomous vehicles using iso 26262 standard: A survey. In *SoutheastCon*, 2018.

Juan-David Guerrero-Balaguera, Josie E. Rodriguez Condia, and Matteo Sonza Reorda. Neural network's reliability to permanent faults: Analyzing the impact of performance optimizations in gpus. In *2022 29th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, 2022.

Juan David Guerrero Balaguera et al. Understanding the effects of permanent faults in gpu's parallelism management and control units. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '23, 2023.

Siva Kumar Sastry Hari, Sarita V. Adve, Helia Naeimi, and Pradeep Ramachandran. Relyzer: exploiting application-level fault equivalence to analyze application resiliency to transient faults. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, page 123–134, 2012. ISBN 9781450307598.

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)*, pages 249–258, 2017a. doi: 10.1109/ISPASS.2017.7975296.

Siva Kumar Sastry Hari et al. 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)*, pages 249–258, 2017b.

Maryam Imani and Hassan Ghassemian. An overview on spectral and spatial information fusion for hyperspectral image classification: Current trends and challenges. *Information Fusion*, 59:59–83, 2020. ISSN 1566-2535.

Muhammad Jaleed Khan et al. Modern trends in hyperspectral image analysis: A review. *IEEE Access*, 6:14118–14129, 2018.

Guanpeng Li, Siva Kumar Sastry Hari, Michael Sullivan, Timothy Tsai, Karthik Pattabiraman, Joel Emer, and Stephen W. Keckler. Understanding error propagation in deep learning neural network (dnn) accelerators and applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, 2017a. ISBN 9781450351140.

Guanpeng Li et al. Understanding error propagation in deep learning neural network (dnn) accelerators and applications. SC '17, 2017b. ISBN 9781450351140.

Ying Li et al. Spectral–spatial classification of hyperspectral imagery with 3d convolutional neural network. *Remote Sensing*, 9(1):67, 2017c.

Robert Limas Sierra, Juan-David Guerrero-Balaguera, Josie E. Rodriguez Condia, and Matteo Sonza Reorda. Exploring hardware fault impacts on different real number representations of the structural resilience of tcus in gpus. *Electronics*, 13(3), 2024. ISSN 2079-9292. doi: 10.3390/electronics13030578. URL https://www.mdpi.com/2079-9292/13/3/578.

Abdulrahman Mahmoud, Neeraj Aggarwal, Alex Nobbe, Jose Rodrigo Sanchez Vicarte, Sarita V. Adve, Christopher W. Fletcher, Iuri Frosio, and Siva Kumar Sastry Hari. Pytorchfi: A runtime perturbation tool for dnns. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 25–31, 2020.

Tobias Meuser et al. Revisiting edge ai: Opportunities and challenges. *IEEE Internet Computing*, 28(4):49–59, 2024.

Daniel Oliveira, Sean Blanchard, Nathan Debardeleben, Fernando F. Dos Santos, Gabriel Piscoya Dávila, Philippe Navaux, Carlo Cazzaniga, Christopher Frost, Robert C. Baumann, and Paolo Rech. Thermal neutrons: a possible threat for supercomputers and safety critical applications. In *2020 IEEE European Test Symposium (ETS)*, pages 1–6, 2020a. doi: 10.1109/ETS48528.2020.9131597.

Daniel Oliveira, Fernando F. dos Santos, Gabriel Piscoya Dávila, Carlo Cazzaniga, Christopher Frost, Robert C. Baumann, and Paolo Rech. High-energy versus thermal neutron contribution to processor and memory error rates. *IEEE Transactions on Nuclear Science*, 67(6):1161–1168, 2020b. doi: 10.1109/TNS.2020.2970535.

Francesco Pessia, Juan-David Guerrero-Balaguera, Robert Limas Sierra, Josie E. Rodriguez Condia, Marco Levorato, and Matteo Sonza Reorda. Effective application-level error modeling of permanent faults on ai accelerators. In *2024 IEEE 30th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, pages 1–7, 2024. doi: 10.1109/IOLTS60994.2024.10616087.

María L. Pinto-Salamanca, Josie E. Rodriguez Condia, José A. Hidalgo-López, and Wilson J. Pérez-Holguín. Analyzing the reliability of stream sparse matrix-vector multiplication accelerators: A high-level approach. In *2024 IEEE 25th Latin American Test Symposium (LATS)*, pages 1–4, 2024.

Henrich C. Pöhls. Towards a unified abstract architecture to coherently and generically describe security goals and risks of ai systems. In Ruben Rios and Joachim Posegga, editors, *Security and Trust Management*, pages 85–94, 2023.

Paolo Rech. Artificial neural networks for space and safety-critical applications: Reliability issues and potential solutions. *IEEE Trans. Nucl. Sci.*, 71(4):377–404, 2024.

A. Ruospo, G. Gavarini, C. de Sio, J. Guerrero, L. Sterpone, M. Sonza Reorda, E. Sanchez, R. Mariani, J. Aribido, and J. Athavale. Assessing convolutional neural networks reliability through statistical fault injections. In *2023 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1–6, 2023a.

Annachiara Ruospo. Reliability assessment methodologies for ann-based systems. In *2022 IEEE 23rd Latin American Test Symposium (LATS)*, pages 1–4, 2022.

Annachiara Ruospo, Ernesto Sanchez, Lucas Matana Luza, Luigi Dilillo, Marcello Traiola, and Alberto Bosio. A survey on deep learning resilience assessment methodologies. *Computer*, 56(2):57–66, 2023b.

Annachiara Ruospo et al. On the reliability assessment of artificial neural networks running on ai-oriented mpsocs. *Applied Sciences*, 11(14), 2021.

Majid Sabbagh et al. Evaluating fault resiliency of compressed deep neural networks. In *IEEE International Conference on Embedded Software and Systems (ICESS)*, pages 1–7, 2019.

Fernando F. dos Santos et al. Revealing gpus vulnerabilities by combining register-transfer and software-level fault injection. In *51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 292–304, 2021a. doi: 10.1109/DSN48987.2021.00042.

Fernando Fernandes dos Santos, Siva Kumar Sastry Hari, Pedro Martins Basso, Luigi Carro, and Paolo Rech. Demystifying gpu reliability: Comparing and combining beam experiments, fault simulation, and profiling. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 289–298, 2021b. doi: 10.1109/IPDPS49936.2021.00037.

Fernando Fernandes dos Santos et al. Analyzing and increasing the reliability of convolutional neural networks on gpus. *IEEE Transactions on Reliability*, 68(2):663–677, 2019.

Fernando Fernandes dos Santos et al. Characterizing a neutron-induced fault model for deep neural networks. *IEEE Transactions on Nuclear Science*, 70(4):370–380, 2023.

Robert Limas Sierra, Juan-David Guerrero-Balaguera, Josie E. Rodriguez Condia, and Matteo Sonza Reorda. Optimizing the analysis and evaluation of logic simulation workloads in hpc systems. In *2023 IEEE 17th International Conference on Application of Information and Communication Technologies (AICT)*, pages 1–6, 2023a. doi: 10.1109/AICT59525.2023.10313156.

Robert Limas Sierra, Juan-David Guerrero-Balaguera, Josie E. Rodriguez Condia, and Matteo Sonza Reorda. Analyzing the impact of different real number formats on the structural reliability of tcus in gpus. In *2023 IFIP/IEEE 31st International Conference on Very Large Scale Integration (VLSI-SoC)*, pages 1–6, 2023b.

Robert Limas Sierra, Juan-David Guerrero-Balaguera, Francesco Pessia, Josie E. Rodriguez Condia, and Matteo Sonza Reorda. Analyzing the impact of scheduling policies on the reliability of gpus running cnn operations. In *2024 IEEE 42nd VLSI Test Symposium (VTS)*, pages 1–7, 2024.

Vilas Sridharan and David R. Kaeli. Quantifying software vulnerability. WREFT '08, 2008.

Lucas Antunes Tambara, Paolo Rech, Eduardo Chielle, and Fernanda Lima Kastensmidt. Analyzing the failure impact of using hard- and soft-cores in all programmable soc under neutron-induced upsets. In *2015 15th European Conference on Radiation and Its Effects on Components and Systems (RADECS)*, pages 1–5, 2015. doi: 10.1109/RADECS.2015.7365586.

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)*, pages 284–291, 2021a. doi: 10.1109/DSN48987.2021.00041.

Timothy Tsai et al. Nvbitfi: Dynamic fault injection for gpus. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 284–291, 2021b.

Vanessa Vargas, Pablo Ramos, Wassim Mansour, Raoul Velazco, Nacer-Edinne Zergainoh, and Jean-François Mehaut. Preliminary results of seu fault-injection on multicore processors in amp mode. In *2014 IEEE 20th International On-Line Testing Symposium (IOLTS)*, pages 194–197, 2014.

Vanessa Vargas, Pablo Ramos, Jean-François Méhaut, and Raoul Velazco. Swifi fault injector for heterogeneous many-core processors. *revistapuce*, (106), 2018.

Blesson Varghese et al. Challenges and opportunities in edge computing. In *IEEE Int. Conf. on Smart Cloud (SmartCloud)*, pages 20–26, 2016.

Oreste Villa, Mark Stephenson, David Nellans, and Stephen W. Keckler. Nvbit: A dynamic binary instrumentation framework for nvidia gpus. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 372–383, New York, NY, USA, 2019a. Association for Computing Machinery. ISBN 9781450369381. doi: 10.1145/3352460.3358307. URL https://doi.org/10.1145/3352460.3358307.

Oreste Villa et al. Nvbit: A dynamic binary instrumentation framework for nvidia gpus. In *52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO

'52, page 372–383, New York, NY, USA, 2019b. Association for Computing Machinery. ISBN 9781450369381.

Keun Soo Yim, Cuong Pham, Mushfiq Saleheen, Zbigniew Kalbarczyk, and Ravishankar Iyer. Hauberk: Lightweight silent data corruption error detector for gpgpu. In *2011 IEEE International Parallel Distributed Processing Symposium*, pages 287–300, 2011. doi: 10.1109/IPDPS.2011.36.