POLITECNICO DI MILANO
Master degree in Computer Science and Engineering
Dipartimento di Elettronica e Informazione

A Methodology for Error Simulation in Convolutional Neural Networks Executed on GPU

Supervisor: Antonio Rosario Miele
Co-supervisor: Luca Maria Cassano

SJTU Supervisor[1]: Jingwen Leng

Author:
Alessandro Toschi, matr. 894350

Academic year 2018-2019

[1]This thesis has been conducted within the double master degree program between Politecnico di Milano and Shanghai Jiao Tong University.

To my family.

Abstract

Nowadays, there is growing interest in employing Convolutional Neural Networks (CNNs) in safety-critical systems. CNNs achieve higher accuracy in perception tasks than the traditional Computer Vision (CV) algorithms. CNNs are generally executed on Graphic Process Units (GPUs) because the Single Instruction Multiple Data (SIMD) architecture of these units is particularly well-suited to speed up the highly data-parallel elaborations of such applications. The acceleration enables the application to meet the strict requirements imposed by safety-critical systems, especially time requirements. The combination, composed of a CNN executed on GPU, is becoming more and more used in safety-critical systems. Therefore, we must ensure the proper functioning of such a combination in any possible situation, also in the presence of faults in the digital systems.

The reliability analysis aims at studying the behavior of systems under the occurrence of faults; the goal is to determine whether the system is able to work correctly by autonomously handling the occurred errors, or it fails, thus producing a wrong result. The most insidious threats in our context are the faults caused by environmental conditions, called *soft errors*. Soft errors do not have disruptive effects, but they induce transient effects that corrupt the state of the system. Indeed, a soft error may change the value of a bit stored in a memory cell, thus inducing an error when that cell is read. Therefore, the activation of a soft error may induce the application to deviate its behavior from its expected functioning. As a matter of fact, it is necessary to understand how the CNN behaves when the soft errors are activated and how much it deviates from the nominal behavior. In fact, the outcomes produced by the CNN can be used by decision-making systems, which choices may have a direct impact on the safety of the users.

Traditionally, the literature is headed towards the reliability analysis of CNNs executed on GPUs through the architectural fault injection. The architectural fault injection for GPU is a technique that emulates the ac-

tivation of soft errors within the architecture of the device. The emulation occurs by injecting bitflips in the GPU data-path, with effects similar to those caused by the physical event. Although it is very accurate, the architectural fault injection poses severe constraints on the system under analysis. The implementation techniques, exploited by the fault injector to emulate the faults, slow down the execution of the application. The slow down may lead the application to not comply with the time constraints to which the whole system is subject. Secondly, the integration of the fault injector and the application is challenging because the application needs to be modified to allow the fault injector to operate. The modification may require to recompile the source code, which can be difficult if the application uses closed-source libraries. Rather than fault injection, the reliability analysis can also be performed through error simulation. The error simulation is a technique that simulates the effects of the activation of soft errors directly in the source code of the application running on the GPU. This happens by corrupting one or more values of the application according to error models. The implementation of an error simulator is far easier than a fault injector because the error simulator can be directly integrated within the Machine Learning (ML) frameworks, which are commonly used to develop CNNs. The main issue related to the error simulation regards the error models with which corrupting the application. The error models must be capable of reproducing the effects of physical faults that occurred in the underlying hardware. Thus, the error models are required to be validated. When the error models are not validated, there is a risk to introduce errors within the system that do not correspond to reality, leading to incorrect outcomes. In the literature, we do not find any validated error models for CNNs executed on GPU since the majority of the works are focused on the architectural fault injection.

For these reasons, the purpose of this thesis is to define a methodological framework for the error simulation using validated models in a CNN at the application level. The goal of the framework is to connect the abstraction level of the GPU architecture, where faults are generally emulated, and the abstraction level of the CNN, where the behavior of the program is analyzed to evaluate the effects of the faults. At first, we have designed a methodology to define the error models that enables us to derive validated error models for the single operator of the CNN. We have performed several architectural fault injection campaigns, targeting the single CNN operator, obtaining thousands of faulty outputs. The faulty outputs are

originated by the activation of soft errors injected during the fault injection campaigns. The error models are built by analyzing these faulty outputs according to three parameters: the number and domains of corrupted values and spatial patterns. These three parameters are defined statistically. The statistic approach enables us to recreate any of the observed faulty outputs by drawing each parameter from its distribution. The error models are thus validated by construction because derived upon the analysis of the faulty outputs. Nonetheless, their effects will be further compared to the ones obtained using the state-of-the-art GPU fault injector publicly available. Besides the error modeling, the framework proposes an approach for performing error simulation campaigns on a CNNs executed on GPU. Such an approach enables us to sabotage the output of a CNN operator, according to the error models defined above. The error simulation allows a higher degree of integration with the application. The higher integration leads us to speed up the execution of error campaigns compared to the current practice.

The framework has been then implemented, bringing us to obtain a repository of error models and an error simulator tool. For the sake of demonstration, the error models repository contains the error models of 11 CNN operators, such as Convolution, Batch Normalization, or Leaky ReLU. Nonetheless, the repository is extendable by applying the same error modeling approach to the other operators. Each model is thus composed of the three probability distributions, one for each parameter. The error simulator is a tool designed for corrupting the outputs of the CNN operators. The corruption, i.e., the insertion of errors, is performed according to the error models present in the repository. The tool is built upon the TensorFlow ML framework, with which the CNN is developed. The error simulator also features some advanced injection techniques, such as checkpointing, or the extensive usage of the cache. These optimizations enable the tool to reuse the intermediate computations, achieving execution times close to the native execution.

Finally, we have compared our framework to two baselines in real case studies. The first comparison regards SASSIFI that is the state-of-the-art GPU fault injector developed by NVIDIA. With that tool, we have compared the execution times and accuracy of our error models. For this comparison, we have used the TensorFlow implementation of the YOLO V3 CNN, which is the state-of-the-art network for object detection. We have simulated 137,000 errors with our error simulator tool in 15 hours. We

have injected 360,000 faults with SASSIFI to obtain the same amount of errors because most of the faults have not been activated. The overall time required by SASSIFI has been 92 hours; thus, the same campaign through our error simulator is 6.1x times faster than the one using SASSIFI. Among these 137,000 errors produced by our error simulator, we have analyzed the effects generated by them in the output of YOLO V3. The obtained effects are equal to the ones generated by SASSIFI in 98.72% of the cases.

The second comparison regards a novel error simulator, TensorFI, that enables us to perform reliability analysis of CNNs. For this comparison, we were not able to use the YOLO V3 network due to technical limitations and design flaws present in TensorFI. Thus, we have used the LeNet-5 model for the MNIST dataset and a model for the CIFAR10 dataset, both performing object classification within images. The models are significantly smaller than YOLO V3 and enable us to test TensorFI. We have simulated 10,000 errors with our error simulator tool in 24.74 and 37.62 seconds for LeNet-5 and CIFAR10 models, respectively. TensorFI has simulated the same amount of errors in 1098.71 and 2409.47 seconds for LeNet-5 and CIFAR10 models, respectively. The speedup induced by our tool compared to TensorFI ranges from $44.41x$ to $64.04x$ times. TensorFI embeds error models that are not validated and are far different from the ones we have observed. The error models embedded in TensorFI are not probabilistic and directly inherited from the fault models used in the architectural fault injection. Therefore, the reliability analysis performed through TensorFI cannot be trusted. The errors observed in the output of the single CNN operator are far complicated than the single bitflip used in the architectural fault injection. Hence, the fault models of the architectural fault injection are not valid for the application level.

In conclusion, we have proved that our error models are validated, either by construction and by comparison. Besides that, our error simulator is faster than the current state-of-the-art tools, achieving execution times close to the native executions.

Sommario

Al giorno d'oggi, si registra un ricorso crescente alle Convolutional Neural Network (CNN) nei sistemi critici in quanto queste raggiungono accuratezze più elevate rispetto ai tradizionali algoritmi della Computer Vision (CV). Le CNN sono eseguite sulle Graphic Process Unit (GPU) poiché l'architettura Single Instruction Multiple Data (SIMD) di queste unità è particolarmente adatta ad accelerare il lavoro di tali applicazioni. L'accelerazione delle CNN sulle GPU è necessaria per rispettare gli stringenti requisiti imposti dai sistemi critici, soprattutto per quanto riguarda i vincoli temporali. Il binomio costituito dalla CNN eseguita sulla GPU è sempre più presente nei sistemi critici, e, per via della loro natura complessa, è necessario assicurare il corretto funzionamento in ogni situazione possibile, anche di fronte a guasti nei sistemi digitali.

L'analisi di affidabilità studia il comportamento dei sistemi in presenza di guasti. L'obbiettivo è determinare se il sistema sia autonomamente in grado di gestire l'occorrenza di guasti, oppure fallisce producendo un risultato errato. Nel nostro contesto, le insidie maggiori sono rappresentate dai guasti originati da fattori e condizioni ambientali, chiamati *soft errors*. I soft errors non hanno effetti distruttivi o permanenti, ma generano guasti transitori che corrompono lo stato del sistema. Infatti, l'occorrenza di un soft error può commutare il valore di un bit contenuto in una cella di memoria che, qualora sia letto, può produrre un errore. L'attivazione di un soft error, cioè la sua lettura, può indurre l'applicazione a comportarsi in modo diverso da quanto atteso. La necessità è quindi comprendere il comportamento delle CNN quando si attivano i soft errors, quantificandone la deviazione rispetto al funzionamento atteso. Questo è indispensabile poiché i risultati prodotti dalla CNN potrebbero essere usati da sistemi decisionali, le cui scelte hanno impatto sulla sicurezza degli utilizzatori.

Tradizionalmente, la letteratura è sempre stata orientata ad eseguire analisi di affidabilità delle CNN eseguite su GPU attraverso tecniche di

iniezioni guasti architetturali. Le iniezioni guasti architetturali su GPU emulano l'attivazione dei soft errors all'interno della sua architettura. L'emulazione avviene iniettando bitflip nel percorso dati della GPU, con effetti analoghi a quelli causati dall'evento fisico. Per quanto molto accurata, l'iniezione guasti architetturale impone numerosi vincoli al sistema in oggetto. Le tecniche usate dagli iniettori per emulare i guasti hanno come effetto collaterale quello di rallentare l'esecuzione dell'applicazione. Il rallentamento può portare l'applicazione a non rispettare più i vincoli temporali a cui è soggetto il sistema complessivo. In secondo luogo, l'integrazione dell'iniettore guasti con l'applicazione è complessa poiché questa necessita di essere modificata per permettere all'iniettore di operare. La modifica dell'applicazione può richiedere di modificare il codice sorgente, non sempre attuabile con librerie di codice a sorgente chiuso. In alternativa all'iniezione guasti, l'analisi di affidabilità può essere effettuata tramite simulazione d'errore. La simulazione d'errore è una tecnica che simula gli effetti delle attivazioni dei soft errors direttamente nel codice sorgente dell'applicazione eseguita su GPU.

La simulazione avviene corrompendo uno o più dati dell'applicazione secondo dei modelli d'errore. La realizzazione di un simulatore d'errore è molto più semplice rispetto a quella di un iniettore guasti architetturali perché il simulatore può integrarsi direttamente nelle librerie di Machine Learning (ML) con cui vengono scritte le CNN. Il problema principale legato al simulatore d'errore riguarda la validazione dei suoi modelli d'errore, poiché devono essere in grado di riprodurre gli effetti fisici che si verificano nel dispositivo sottostante. Se i modelli d'errore non sono validati, si corre il rischio di introdurre errori nel sistema che non corrispondono alla realtà, portando ad una analisi incorretta. Nella letteratura, non troviamo riscontri di modelli d'errore validati per le CNN eseguite su GPU dato che la maggior parte dei lavori è focalizzata sull'iniezione guasti architetturali.

Lo scopo di questa tesi è definire un framework metodologico per la simulazione d'errore a livello applicativo su una CNN, attraverso modelli d'errore validati. L'obbiettivo del framework è connettere il livello di astrazione della GPU, dove vengono emulati i guasti, a quello della CNN, dove viene analizzato il comportamento in presenza di tali guasti. In primo luogo, abbiamo definito una metodologia per creare modelli d'errore validati sul singolo operatore della CNN. Successivamente, abbiamo eseguito numerose campagne di iniezioni guasti architetturali sui singoli operatori della CNN ottenendo migliaia di risultati corrotti. Questi risultati sono

originati dall'attivazione dei soft errors iniettati nelle campagne guasti. I modelli d'errore sono definiti analizzando i risultati corrotti secondo tre parametri: numero e domini dei valori corrotti e motivo spaziale. Il modello così descritto segue un approccio statistico, con cui è possibile ricreare i risultati corrotti osservati secondo le distribuzioni di probabilità di ogni parametro. I modelli d'errore sono perciò validati per costruzione poiché derivati dall'analisi dei risultati corrotti. Nonostante questo, vogliamo offrire un'ulteriore comparazione dei nostri modelli d'errore confrontando gli effetti che essi generano con quelli generati dal migliore iniettore guasti architetturali per GPU. La seconda contribuzione del framework è un approccio per realizzare campagne di simulazione d'errore sulle CNN eseguite su GPU. L'approccio consiste nel sabotare l'uscita di un operatore della CNN, secondo i modelli d'errore definiti sopra. La simulazione d'errore raggiunge un grado di integrazione maggiore con l'applicazione, riuscendo perciò a velocizzare l'esecuzione delle campagne di errore rispetto alle pratiche attuali.

Il framework è stato poi implementato, ottenendo una collezione di modelli d'errore e uno strumento di simulazione d'errore. A titolo dimostrativo, abbiamo popolato la collezione con modelli d'errore basati su 11 operatori della CNN, quali la Convolution, Batch Norm, oppure Leaky ReLU. La collezione rimane aperta ad estensioni future per tutti gli ulteriori operatori. Il simulatore d'errori è uno strumento progettato per corrompere l'uscita di un operatore della CNN, inserendo errori secondo i modelli presenti nella collezione. Il simulatore è basato sulla libreria di ML TensorFlow, e integra alcune tecniche avanzate di iniezione, come il check-pointing e l'uso estensivo di cache, che gli assicurano tempi d'esecuzione vicini a quelli nativi.

In chiusura, abbiamo comparato il nostro framework a due punti di riferimento del settore su casi d'uso reali. La prima comparazione riguarda SASSIFI che è lo stato dell'arte nel contesto di iniettori guasti architetturali per GPU, sviluppato da NVIDIA. Con SASSIFI abbiamo testato i tempi d'esecuzione nello svolgere la stessa campagna di errori e l'accuratezza dei modelli d'errore. Il soggetto di questo confronto è stato YOLO V3 che è una CNN stato dell'arte nel contesto dell'identificazione di oggetti. Abbiamo simulato col nostro strumento 137,000 errori in 15 ore. Per ottenere lo stesso numero di errori con SASSIFI, abbiamo iniettato 360,000 guasti poiché la maggior parte di essi non si è attivato. Il tempo totale richiesto da SASSIFI è stato di 92 ore, evidenziando che il nostro simulatore è stato

6.1 volte più veloce nel fare la stessa campagna di errori. Di questi 137000 errori abbiamo analizzato gli effetti nell'uscita di YOLO V3 sia per il nostro simulatore che per SASSIFI, risultando capaci di generare il 98.72% degli effetti di SASSIFI.

Il secondo confronto riguarda un emergente simulatore d'errore per CNN chiamato TensorFI. Per via di limitazioni tecniche ed errori progettuali, non è stato possibile utilizzare YOLO V3 con TensorFI. Al suo posto abbiamo impiegato due CNN, LeNet-5 per il MNIST dataset e una implementazione personale per CIFAR10. Entrambe le reti effettuano classificazione d'oggetti in immagini. Abbiamo simulato 10,000 errori in entrambe le reti in 24.74 e 37.62 secondi rispettivamente per LeNet-5 e CIFAR10. Le stesse campagne con TensorFI hanno richiesto 1098.71 e 2409.47 secondi rispettivamente per LeNet-5 e CIFAR10. TensorFI integra modelli d'errore che non sono probabilistici, importandoli direttamente dai modelli di guasto usati nell'iniezione architetturale. Perciò, le analisi di affidabilità attraverso TensorFI non sono verosimili poiché gli errori osservati nell'uscita di un singolo operatore della CNN sono molto più complessi dei bitflip usati a livello architetturale. Quindi, i modelli di guasto architetturali non sono validi per l'applicazione.

In conclusione, abbiamo dimostrato che i nostri modelli d'errore sono validati sia per costruzione che per confronto. Inoltre, il nostro simulatore è più veloce degli strumenti che attualmente rappresentano lo stato dell'arte, ottenendo tempi di esecuzione molto prossimi a quelli nativi.

Ringraziamenti

The greatest enemy of knowledge is not ignorance,
it is the illusion of knowledge.
Stephen William Hawking

Ringrazio innanzitutto i professori Antonio Rosario Miele e Luca Maria Cassano che mi hanno sapientemente guidato nella realizzazione di questa tesi, curando meticolosamente ogni dettaglio portando al compimento di questo lavoro di cui sono fiero ed orgoglioso.

Ringrazio i miei genitori, mia mamma e mio babbo, che mi hanno donato il loro supporto incondizionato e tranquillità che mi hanno permesso di raggiungere questo obiettivo e per questo vi sono eternamente grato.

Ringrazio mia sorella, i miei nipoti Sofia e Riccardo, che con la vostra bellissima famiglia, a cui sono molto legato, mi siete stati sempre vicino.

Ringrazio la mia fidanzata Elena, per essere stata un faro lungo il mio cammino nonchè compagna di vita. Grazie perchè se sono arrivato qui oggi lo devo alla serenità e felicità che derivano dall'averti al mio fianco.

Ringrazio Francesco e Laura, compagni d'avventura dal 1995 coi quali so di poter contare sempre e siete un punto fisso nella mia vita. Ringrazio anche Federico, Ossama, e Pierpaolo per camminare al mio fianco avermi reso lieta questa avventura.

Ringrazio gli amici "cinesi" per avermi sostenuto nella grande avventura della Cina e per avermi aiutato in tutti quei momenti difficili regalandomi la forza di concludere il viaggio. In particolare, Edoardo, Giulia, e Giacomo, perchè siete stata una delle scoperte più preziose fatte in Cina.

Infine, ringrazio tutti per il vostro supporto e che se sono arrivato qui oggi lo devo a tutti voi.

Grazie ancora!
Alessandro

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Digital systems are nowadays widely employed in all the activities of our life. Depending on the working scenario and the application role, digital systems may assume a certain level of criticality for the success of the mission of the overall appliance/facility they are integrated into and for the safety of people and the environment they interact with. More precisely, systems are dubbed as *safety-critical systems* if their failure or malfunctioning may cause injuries on the people that are around it or working with it.

Among the safety-critical systems, Autonomous Driving System (ADS) represents the ultimate challenge in recent years [1]. An ADS is composed of a set of high-level functionalities aimed at (partially) replacing the human driver with electronics and machinery in the various driving tasks, such as perception, planning, and control of the vehicle. For instance, an ADS is capable of detecting the lanes of the current track or identifying the pedestrians and other obstacles by exploiting Computer Vision (CV) and Machine Learning (ML) algorithms to automate the perception functionalities. Once the lanes and obstacles are identified, they are consumed by the planner, which is the functionality having in charge of the decision-making activities. The planner generates the feasible trajectories and selects the best one by solving an optimization problem. Finally, further ML algorithms are also employed in control to automate the car's operations.

The functionalities offered by an ADS are classified and ruled by the Society of Automotive Engineers (SAE), which is a standard developing organization that produces the standards and regulations adopted by any interested party in the field of autonomous driving. According to the SAE standards [2], autonomous driving functionalities are classified according to

the provided level of automation in five levels, from the lowest automation level 1 to the highest 5. A car equipped with an ADS of level 5 implies that the car can drive under all conditions and everywhere without requiring any intervention from the driver, which is now considered as a passenger. Therefore, ADS of level 5 must expose a very high degree of reliability to be able to face any situation, even the most critical since there are human lives at stake.

At the same time, one of the most relevant problems in digital circuits is that they may be subject to physical faults, leading to failures and malfunctioning in the system's behavior. The main causes of faults may be internal to the devices, such as premature break-downs or aging of some component, or external, mainly due to mechanical or thermal stress, or exposition to radiations. Therefore, devices have considerable chances to experience *hard faults*, such as permanent break down, or, with a higher frequency, *soft errors*, causing temporary memory state change or data corruption. The overall effect of a fault affecting a system is a possible deviation of the executed application from its nominal behaviour to an erroneous one, thus leading to the computation of a wrong result or even a system crash.

Radiation-induced faults have historically been considered a concern by companies and researchers belonging to the aerospace domain. Nevertheless, it has been demonstrated that a (small) number of such faults may also occur at the ground level [3]. According to this analysis, 2 faults every thousand billion hours occur on average at ground-level. If we consider that in 2019 the number of cars traveling in Europe has been 268 million (see [4]), we can estimate that a fault would be observed on every single car every 3.7 hours, which may be a concern. A fault that causes a wrong result may pose the system into a critical state. The criticality of a wrong result is far highly severe when it happens on information that is considered the entry-point for many other following modules. In autonomous driving, it is, therefore, necessary to focus primarily on the perception module because it is in charge of detecting the surrounding environment, assisted by several sensors, such as cameras, radars, and LIDARs, by identifying the present obstacles and many other elements, like traffic signs, lanes, and semaphore's lights. The information produced by the perception module is consumed by other modules, such as the planner or the prediction modules, which are responsible for decision-making tasks. As a consequence, the wrong information that is issued from the perception module can lead to a wrong decision and next to dangerous repercussions, like the emer-

(a) Correct classification *(b) Misclassification* *(c) Wrong placement*

Figure 1.1: Two hypothetical situations originated by a soft error.

gency stop of the vehicle or, in the worst cases, the crash of the car. Let us concretize this situation with the following example.

Consider an autonomous vehicle that is traveling at a constant speed of 130 km/h on the highway, employing a level 5 ADS. The front camera is capturing frames at 10 fps, which are then processed by the perception module to detect the obstacles within the frames. Suddenly, a soft error occurs while the perception module was assigning the label to the detected obstacle that is in front of the car. Instead of labeling the obstacle as "Truck" (Figure 1.1(a)), the obstacle has been classified as "Traffic Cone", as shown in Figure 1.1(b). Another possibility is that a soft error is manifested as the wrong placement of the bounding box that identifies the truck, as shown in Figure 1.1(c). Both situations represent a danger for the passengers. In the first case, the planner may select a trajectory to overtake the traffic cone, but due to sudden change, the maneuver results sharp and perilous. The only viable countermeasure for commercial ADS that faces the second case is to trigger the watchdog and activates an emergency brake. Both situations are not optimal from the passengers' point of view, even if not catastrophic, but a further effort should be spent to define countermeasures that improve the management of critical situations and soft errors.

Given these discussions, it is clear how it is mandatory to investigate the reliability issues of the considered ADS, and, in particular, of the perception functionalities, to assess if they are robust enough or they need to be hardened with some reliability-oriented mechanism. Such a kind of investigation is generally carried out by evaluating how the system reacts to the occurrence of faults and how its behavior deviates from the nominal one.

The perception module exploits various CV algorithms to detect obstacles and objects. Nowadays, the state-of-the-art techniques used in CV rely on Convolutional Neural Networks (CNNs), which overtake traditional image processing pipelines. CNNs are data-intensive computational models that present a high degree of data parallelism, which is appropriately exploited by accelerating them using Graphic Process Units (GPUs). The combination of CNNs and GPUs is successful because it is possible to achieve high performance that meets the real-time constraints of the car's ADS; on the other hand, it is mandatory to assess the effects of faults on this specific system.

Reliability analysis is generally performed in the later phases of the design flow of a system as a final assessment. Unfortunately, this common practice is not effective in the scenario of ADSs due to the high complexity of both the applications and the underlying architectural platforms and the too many strict requirements on the system (in terms of performance, power consumption) and on the design activity (design time, costs and effort). As a consequence, we claim that there is a need for novel reliability-related design and analysis approaches to be tightly integrated with the rest of the design flow and providing early feedback also on reliability issues.

1.1 Goal

The goal of this thesis is to propose a novel framework for the automatic analysis of the reliability properties of CNNs executed on GPU devices. The idea is to replace the classical reliability analysis performed through physical fault injection or architectural fault emulation by means of an error simulation approach acting at the functional level. This will ease the reliability analysis both in terms of design effort and execution time and will offer a better visibility/identification of the reliability issues of the system, or sub-parts of the application pipeline, till to the early phases of the design flow. In this way, the framework will support the designer in understanding if a CNN is sufficiently resilient against faults, or it needs to be hardened; in the latter case, whether it suffices to focus only on some parts/step of the application pipeline, to keep overheads to a minimum (to save power, time, area) or it is mandatory to harden the entire system.

In order to achieve such a result, the framework will be provided with a methodology to properly define error models to be applied directly in the application execution. Such models have to accurately represent the effects

caused by faults occurring in the underlying GPU device. The second part of the proposed idea is the actual error simulator capable of automating extensive experimental campaigns on the CNN under analysis by exploiting the outcomes of the first phase. Such a simulator will be integrated with the most popular ML frameworks; in such a way, the reliability analysis of a CNN can be performed in the same environment in which it is designed, thus offering fast feedback to the design activities.

1.2 Thesis Outline

The thesis is organized as follows:

- Chapter 2 introduces all the knowledge required to understand this work. The background presents the theoretical discussion about the CNNs, the ML frameworks, GPUs, and faults. A review of the literature is presented in the second part of the chapter, in which we compare the current state-of-the-art GPU fault injector and the methodologies adopted in terms of reliability analysis.
- Chapter 3 states the current limitations and constraints of the literature, and we present the points that enhance the current state-of-the-art methodologies.
- Chapter 4 presents the framework and the general methodologies behind it. The methodologies are developed in the context of CNN application executed on GPU, but the discussion is also offered in a general way, applicable to a class of image processing applications that are accelerated on a computing device.
- Chapter 5 presents the steps that characterize the error modeling phase. Such a modeling phase aims to build an error model that is representative of the architectural faults that occur on GPU. Several GPU fault injection campaigns are executed targeting a CNN operator, obtaining many faulty outputs. The steps provided by the methodology enable us to analyze those faulty outputs and to derive an error model built on them that is meaningful for the application level.
- Chapter 6 presents the tool that can inject the errors modeled in the previous phase into a CNN running in a ML framework, outperforming traditional architectural fault injectors.

- Chapter 7 presents the evaluation and validation metrics of both the error simulator and error models compared to the state-of-the-art works.
- Chapter 8 encloses all the results obtained and our contributions. Besides, we present the possible future works that extend the working scenario.

Chapter 2

Background and Related Work

This chapter introduces the fundamental concepts, topics, and foreknowledge needed to understand the working scenario considered in this thesis. The system under analysis is a Convolutional Neural Network (CNN), which is an application widely employed in the context of image processing and computer vision, executed on a Graphic Process Unit (GPU) device. CNNs consist of a graph of connected operations, which purpose and meaning will be described in Section 2.1. CNNs are generally developed by means of specific Machine Learning (ML) frameworks. Among the available ones, TensorFlow [5] is the most widespread and popular framework that will be adopted by the application. Caffe [6] is an alternative ML framework, mostly specialized in CNNs, that will be used to validate our work. Both frameworks will be described in Section 2.2. CNNs are heavy computational tasks that take great benefit from the parallel architecture of GPUs. GPUs are parallel computing devices, which represent the target architecture of this work, and its overview will be provided in Section 2.3. Finally, the presentation of the background is completed by Section 2.4 where reliability issues related to *faults* possibly affecting the digital systems are briefly discussed. The second part of the chapter is dedicated to the review of the literature and the analysis of the state-of-the-art works in the context of reliability analysis of GPUs running CNN applications. Section 2.5.1 presents the available fault injectors targeting GPU devices and, later, Section 2.5.2 focuses on the methodologies to evaluate the vulnerability of CNNs against faults. Benefits and limitations of the past works are discussed to introduce the necessity of the approach proposed in this thesis.

2.1 Convolutional Neural Networks - CNNs

Convolutional Neural Network (CNN) [7] is a computing model that manages multidimensional data, also known as *tensors*, and aims to derive a semantic representation from the input to accomplish a high-end task. CNNs are generally employed in the field of image processing and computer vision. Images are the best data to be fed into a CNN and represent the best candidates to which apply feature extraction. The CNN can perform several tasks; it can classify items within an image, detects objects through bounding boxes individuation, i.e., a rectangle of pixels, and performs image segmentation, i.e., assigning to each pixel the most probable label. A brief overview of the various operators or layers composing a CNN will be carried out in the following subsections.

The standard topology of a CNN is composed of a series of blocks, as shown as an example in Figure 2.1. The blocks follow a common pattern repeated various times and consist of: one or more convolutional layers, a batch normalization, an activation function, and then max-pooling. Depending on the task to accomplish, the CNN can have different terminal blocks. The terminal blocks can be constituted of a traditional fully-connected neural network if we are dealing with a classification task, like in Figure 2.1. Otherwise, the output of the CNN can be a tensor itself, which in this case, the network is referred to as fully convolutional, and this situation is typical for image segmentation or object identification. Each mentioned block has a different purpose. The convolutional layer is used to learn and extract features from the input by learning the appropriate set of weights. Such features are ones that maximize the network's accuracy against a loss function but are likely to be not human manageable or understandable. The batch normalization is a mathematical operation that helps the training phase by fixing the data distribution, speeding up the

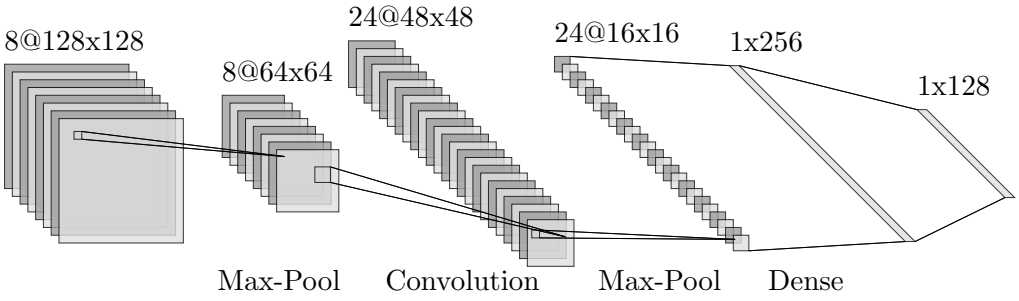


Figure 2.1: The typical topology of a CNN, credits to [8].

learning. The activation function is a mathematical function that is applied element-wise to emulate the biological activation of a neuron. The max-pooling layer belongs to the category of dimensionality reduction operators, which reduces the size of the tensor to increase the degree of generalization. Recapping, CNN aims to learn a wide number of meaningful and semantic features, up to thousands, while reducing the size of the tensor to generalize its representation. In the following sections, the various blocks composing the CNN are discussed in details.

2.1.1 Tensor

A *tensor* [9] is a data structure in which elements are arranged on regular grids stacked along a common axis, as shown in Figure 2.2. It is the basic unit element of any CNN, representing the input and output of each layer or operator. An intuitive representation of a tensor is an RGB image because it is a grid of pixels, and each pixel is composed of three components, i.e., the color components. The regular grids are either called *channels* or *feature-maps*, depending on the context. The spatial representation of a tensor is identified by the triplet $(C \times H \times W)$, in which C represents the number of channels or feature-maps and $H \times W$ is the grid size. This representation is known as *channels-first* because channels are prefixed to grid size. The dual representation, known as *channels-last*, places the channels after the grid size. These representations are equivalent, except for low-level memory allocation layout, but they are useful to interpret data with different semantic within the scope. ML applications usually use the channels-first representation and the grids are referred to feature-maps, whereas Computer Vision (CV) applications use channels-last representation and channels.

2.1.2 Convolution

Convolution [9] is a linear operation that maps each element of a regular grid, i.e., feature-map, to a linear combination of the element itself and its neighbors. The coefficients of the linear combination take different names depending on the application field. In ML literature is common to find them named as kernel, weights, or mask. The convolution, viewed as a generalization of filter operation, introduces the concept of weights sharing because the kernel is shared among the elements of the grid and helps to keep the model general and lightweight.

The convolution can be mathematically formulated as a function $g(x, y)$, which operates on a grid $f(x, y)$ and a kernel K of size (K_W, K_H):

$$g(x, y) = \sum_i \sum_j f(x - i, y - j) * K(i, j)$$

The convolutional layer is used in CNNs to learn different filters, capable of exploiting high-level aspects. It maps an input space $(C_i \times H_i \times W_i)$ to $(C_o \times H_o \times W_o)$ output space, in which C_o is the number of filters learnt by the layer. It is worth noting that the grid size in the output space can change, letting the layer not only performing convolution but also dimensionality reduction. This behavior can be obtained by setting the *stride* parameter to a value greater than one. Instead of sliding the kernel over each element of the grid, a non-unitary stride allows defining the step between two successive elements, diminishing the number of processed elements and so the size of the output grid. The layer learns a weight tensor of shape $(C_i \times, C_o \times K_H \times K_W)$, in which there is a kernel of size $(K_H \times K_W)$ for each input and output feature map. C_i convolutions are executed, and then the intermediary outputs are summed together, obtaining the *ith* output feature map, as shown in Figure 2.3.

It is worth mentioning that the convolutional layer is implemented using matrix multiplication, in which the input tensor is suitably rearranged into a data structure where each element and its neighbors are placed in columns, converting a slide operation into a matrix multiplication. This rearrangement operation puts each element and its neighbors into a column vector, projecting the two-dimensional space into a linearized one, as shown in Figure 2.4. Column vectors are then stacked to form a matrix,

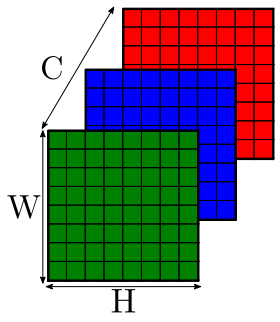


Figure 2.2: Graphical representation of a channels-first tensor, highlighting C channels and $H \times W$ grid size.

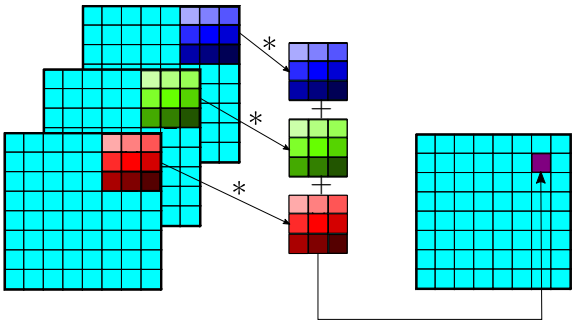


Figure 2.3: Example of 3x3 convolution.

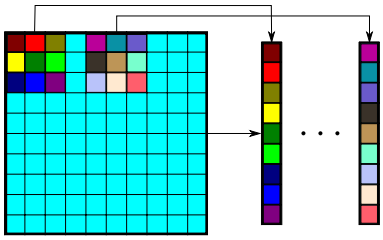


Figure 2.4: Tensor rearrangement into columns format.

and the kernel's weights are linearized into a row vector. The two linearizations allow the convolution to be transformed into a matrix multiplication, which is executed faster than a sliding operation because it can benefit the presence of device accelerators, such as the GPU.

2.1.3 Dimensionality Reduction

Dimensionality reduction is a necessary operation to reduce the sensitivity of the model. The CNN learns representations that are sensitive to the precise position of features in the input image. Small movements, i.e., translations, on the input image can lead to different feature maps and different classification, for example. To overcome this issue, it is necessary to down-sampling the current representation because a lower resolution version of the input still contains most of the significance. The dimensionality reduction can be achieved either by setting a stride greater than one in a convolutional layer, as described in the previous section or by using a pooling operator. There exist many pooling operators, but the most used one is the max pooling operator [9] that applies a max filter to non-overlapping sub-regions of the input tensor.

2.1.4 Batch Normalization

Batch normalization [10] is an operation that aims to reduce the *internal co-variance shift* and induces a dramatic acceleration of the CNN training phase. The internal co-variance shift can be defined as the continuous change in the distribution of input data. This issue represents a challenging problem because layers have to adapt to new data distributions while learning the appropriate filters, which minimize the loss. This can cause the learning algorithm to forever chase a moving target. Whitening the input data helps layers to handle only consistent data, to avoid the insidious issue of vanishing gradient, and to focus only on optimizing the loss, achieving substantial speedup.

The normalization consists of scaling the input data to have zero mean and unitary variance. The scaling is not performed globally over the whole input, but it is applied grid-wise. Suppose of having an input tensor $\boldsymbol{x}$ with shape $(C \times H \times W)$, and $\boldsymbol{x}^{(c)}$ indicates the *c-ith* channel, then:

$$\hat{\boldsymbol{x}}^{(c)} = \frac{\boldsymbol{x}^{(c)} - \mu_B^{(c)}}{\sqrt{\sigma_B^{2(c)} + \varepsilon}} \qquad \mu_B^{(c)} = \frac{1}{m}\sum_{i=0}^{m} \boldsymbol{x}_m^{(c)} \qquad \sigma_B^{2(c)} = \frac{1}{m}\sum_{i=0}^{m}\left(\boldsymbol{x}_i^{(c)} - \mu_B^{(c)}\right)$$

The training phase is assumed to be executed on batches of size m, so the mean and variance are computed over a batch of input tensors. Two additional trainable parameters, $\gamma^{(c)}$, and $\beta^{(c)}$, are introduced to let the normalized data to be re-scaled over the original domain, ensuring flexibility and avoiding to lose representation power induced by scaling.

$$\boldsymbol{y}^{(c)} = \gamma^{(c)}\hat{\boldsymbol{x}}^{(c)} + \beta^{(c)}$$

This layer learns C scaling factors $\gamma^{(c)}$, C biases $\beta^{(c)}$ whereas C means $\mu_B^{(c)}$, and C variances $\sigma^{2(c)}$ are computed during the training phase and remain constants over the inference phase.

2.1.5 Activation Functions

An activation function is a mathematical function that is applied element-wise to a tensor. The role of the activation function is to emulate the biological activation of neurons by abstracting the rate of action potential firing in cells [11].

Among the possible activation functions, we here mention the *Rectified Units* class of functions, and then the *Sigmoid* one.

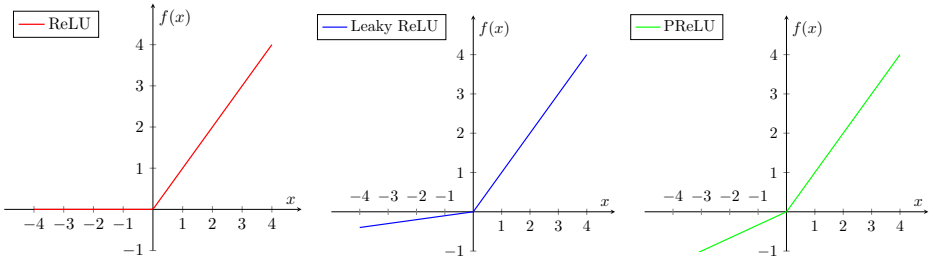
Table 2.1: Functions belonging to the class of Rectified Units.

Name	Math Formulation	Figure
Rectified Linear Unit (ReLU) [14]	$f(x) = max(0, x)$	2.5(1)
Leaky ReLU [15]	$f(x) = \begin{cases} x, & \text{if } x \geq 0 \\ 0.01x, & \text{if } x < 0 \end{cases}$	2.5(2)
Parametric ReLU (PReLU) [16]	$f(x) = \begin{cases} x, & \text{if } x \geq 0 \\ a\,x, & \text{if } x < 0,\ a \in \mathbb{R}^+ \end{cases}$	2.5(3)

Rectified Units. Rectified units [12] is a class of activation functions that have gained great success in CNNs. They all share the common behavior to be linear in the right half plane ($x > 0$) and *zero* or *almost zero* in the left half plane ($x < 0$). Their main advantages [13] are:

1. Easy to compute, speeding up the execution.
2. The gradient of such functions allows us to write a deeper network because it overcomes the *vanishing gradient* issue.
3. Produce a sparse output, which is closer to what happens in a human brain since not all the neurons activate simultaneously.

Table 2.1 shows the most popular examples of the Rectified Units and their mathematical formulation. Leaky ReLU and PReLU are a relaxation of the ReLU, in which they do not crop negative values to zero but allows some leakage, preventing the neuron from dying. The only difference between Leaky Relu and PReLU is that the negative slope present in the Leaky ReLU is fixed, while that parameter is trainable for the PReLU.



((1)) ReLU. *((2)) Leaky ReLU.* *((3)) PReLU with $a = 0.33$.*

Figure 2.5: List of Rectified Units plots.

Sigmoid. The Sigmoid [17] (Figure 2.6) activation function is a monotonic real function defined as:

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

It is a popular activation function because it ranges between 0 and 1, exhibiting a smooth exponential transition between the boundaries, and usually is used to represent probabilities.

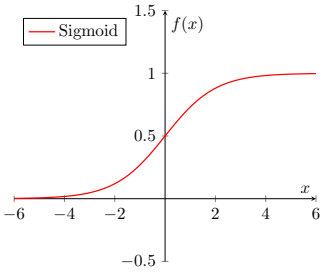


Figure 2.6: Sigmoid activation function.

2.1.6 Element-wise Operators

It worth mention a set of simple element-wise operators that are widely used in CNNs in the top-most graph or internally in the various already-discussed operators. They represent arithmetic transformation of a tensor, either by adding two tensors, by applying a function element-wise, or by scaling by a factor. These operators are listed in the following list:

1. **Add**: adds two tensors, element-wise.
2. **Mul**: scales each tensor's feature maps, likewise the BiasAdd operation.
3. **Div**: divides the tensor by a constant dividend.
4. **Exp**: applies the exponential function to a tensor, element-wise.
5. **Biasadd**: C scalar biases are added to the C feature maps of a tensor. The biases are trainable and broadcasted at real-time to fill the feature map size.

2.2 Machine Learning Frameworks

Designing a Machine Learning (ML) software is a challenging activity since algorithms are very complex, requiring researchers and engineers to spend months trying to replicate published papers results or by defining a model. Such applications are benefiting the GPU acceleration, but writing custom code that targets the GPU requires a considerable amount of time and specific knowledge of the underlying architecture. ML frameworks have been developed to ease such a design activity, exploiting the fact that, as shown for CNN in Section 2.1, most of these algorithms are built upon a set of standard operators.

The publicly available frameworks allow forgetting all the implementations details because they provide general and extensible interfaces that would suit the majority of workloads. The frameworks are also optimized to deliver the computation on the best available device, achieving high performances. These aspects enable the user only to focus on the CNN model, for instance, demanding all the implementation details to the framework, shortening the time to deploy the ML model. Among these publicly available frameworks, we will present in the following sub-sections two of them, i.e. TensorFlow and Caffe, since both of them have been employed in this work with different purposes.

2.2.1 TensorFlow

TensorFlow [5] is a framework for designing ML models and executing them on a wide variety of heterogeneous systems, ranging from mobile devices to large distributed systems with numerous GPUs. It is developed by Google and enables to write almost any ML model.

Among the many available models, it is particularly suitable for developing CNN models, by either supporting training and inference. TensorFlow describes each computation using a state-full data-flow graph, as shown in the example in Figure 2.7, in which each operation is mapped to a wide variety of hardware platforms. TensorFlow expresses each operator of a CNN as a node of the data-flow graph, having zero or more edges, both incoming and outgoing, in which the values that flow in such edges are tensors objects. The data-flow graph results to be a directed graph but with the peculiarity of being append-only, which means that it is not possible to remove a node from the graph once inserted nor altering the structure or connection of an already existing node. What makes TensorFlow differ-

ent from the other ML frameworks is that it decouples the definition of a CNN, or any other models, from its instantiation and execution, through the data-flow graph. When a CNN's operator is added to the data-flow graph, it represents an abstract computation, called *operation*. Afterward, when the data-flow graph is executed, TensorFlow assigns the best available implementation to each operation, called *kernel*, which can be run on a specific device, such as GPU or CPU.

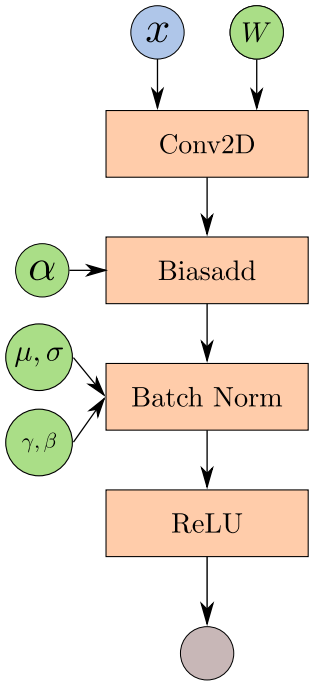


Figure 2.7: Example of data-flow computational graph. Blue objects are user's inputs, green objects are model parameters, and orange boxes are operators.

TensorFlow provides different levels of Application Program Interfaces (APIs) with which it is possible to define the model. Such APIs range from low-level APIs, which have a one-to-one match with the data-flow graph operations, to high-level APIs, such as Keras [18], which have a higher degree of abstraction and their usage can lead to the insertion of tens of operation within the data-flow graph. The availability of such high-level APIs is what makes TensorFlow so popular and used because with few lines of codes is possible to create a complete CNN model capable of being executed on the GPU.

The instantiation and execution of the data-flow graph are demanded to a *session* interface, which is in charge of allocating the memory required

by each operation and identifies the best implementation kernel for each operation. Usually, the session is created once, and then the data-flow graph is fully or partially executed by providing the input tensors and the outputs to be computed.

Finally, it worth mentioning two special components of the data-flow graph, which are necessary for the later discussions on the error simulator designed in this work:

- *Control Dependency*: it is a special edge that introduces priority among operations. If the operation A is connected through a control dependency edge to the operation B, then the operation A will be executed before the operation B.
- *Variable*: it is a special operation that holds a reference to a mutable persistent tensor that survives across multiple executions of the data-flow graph, while most tensors do not. Variables are used to store the weights and attributes of operations because they need to persist during all the executions.

2.2.2 Caffe

Caffe [6] is an open-source modifiable framework for developing state-of-the-art deep learning algorithm and possibly accelerating them on GPU. Caffe focuses on CNNs by expressing them as directed acyclic graphs of layers. The fundamental basic blocks of which Caffe is composed are *blobs* and *layers*. Blobs are four-dimensional arrays used to store data, either batch of images, layers' parameters, or any other attributes. Blobs are allocated on demand and provide a unified interface that handles all the synchronization operations in heterogeneous contexts, for example copying the memory from the GPU to the CPU and vice-versa when it is needed. The layers are the essence of a neural network, and they provide two operations: the *forward pass* and *backward pass*. The forward pass, also known as the inference phase, takes a set of input blobs and produces a set of output blobs, while the backward pass is the inverse operation in the opposite direction, typical of the training phase. Each layer provides either a CPU and a GPU implementation of each operation. The Caffe Network is a data structure in charge of keeping the layout of the CNN, calling the forward pass on layers in order, and determining which implementation has to be executed according to the global settings. The global settings contain a switch attribute that indicates if it has to use the GPU or CPU

implementation. Caffe is designed to separate the model definition from its implementation. The model is defined using the Protocol Buffer language [19] with which the user outlines the network by stating the layers and their connections. The model is then loaded by the Caffe Network that instantiates the network as presented in the configuration file. This decoupling leads the user to fast prototyping the CNN models without caring too much about the implementations. However, the separation between the model definition and the implementation is not that strong as in TensorFlow. From the definition of a layer to its implementation is just a function call, and this enables a greater degree of inspection and control of what is executing. This closeness allows the framework to achieve high performances, especially using the GPU because the whole abstraction level is bare-metal to the hardware platform.

2.3 Graphic Process Units - GPUs

Graphic Process Units (GPUs) are computing devices designed to accelerate the rendering of 3D scenes. Such devices embed in hardware the graphics pipeline, which contains all the processing steps that turn a 3D model into a pixel buffer to be displayed on a screen. The processing steps within the graphics pipeline are extremely regular tasks on the various basic chunks of the input data [20]; this means that the same operation is applied to every pixel or element, without involving complex control logic typical of the general-purpose tasks. In other words, the behavior exposed by the graphics pipeline tasks, such as pixel manipulation, lighting, camera transformation, and many more, is highly data parallel. The operations performed on every pixel can be performed in parallel threads, which follow a regular structure, with low presence of branches and divergencies, and without occurring in complex data-race conditions. All these conditions bring to the creation of a dedicated type of devices; in fact GPUs are designed to accelerate such tasks and to provide a many-core architecture able to leveraging the data-parallel nature of the graphics pipeline, achieving outstanding performances, which are not comparable with traditional CPUs [21]. In the latest years, GPUs have been started to be also applied in traditional High Performance Computing (HPC) tasks that exhibit a high level of data-parallelism, similar to the graphics pipeline. This shift has brought GPUs to approach more general-purpose tasks, like many operations performed in ML, such as the General Matrix Multiplications

(GEMM) [22], which is the basic block of many operators, like the convolution or dense layer. GPUs have therefore evolved to the General Purpose GPU (GPGPU) paradigm by integrating new many-core architectures that would also suit tasks not only confined to graphics pipelines, such as ML and CV. Indeed, the two ML frameworks presented in the previous sections, TensorFlow and Caffe, are both accelerated through GPU.

In the next two sections, we will briefly describe the hardware architecture and the programming model that characterize the GPU, using the NVIDIA nomenclature.

2.3.1 Hardware Architecture

The GPU integrates a many-core architecture organized in two levels [23]; it is composed of an array of Streaming Multiprocessors (SMs), each of them, in its turn, is a multi-core processor, as shown in Figure 2.8[1]. Each SM is a Single Instruction Multiple Data (SIMD) multi-core processor that leverages on the data-parallelism by executing multiple threads in parallel that share the same instructions stream and operate on different data. As shown in Figure 2.8 right side, the internal structure of a SM presents the following units:

- *Scheduler*: this unit is in charge of performing the fetch and decode operation, which loads the instruction for the stream.
- *Register File*: this unit contains the registers used by each thread. The maximum number of registers per thread is 255, while the maximum number of 32-bit register per SM is 32K or 64K [21], depending on the architecture.
- *Execution Units*: each SM is composed of hundreds of execution units that execute in parallel the instruction loaded by the schedule unit. There exists three types of execution unit:
  - *ALU*: arithmetic logic unit that manages either 32-bit floating-point and integer values, with support of 64-bit values.
  - *LD/ST*: memory unit, which performs loads and stores instructions.

[1]It is worth mentioning that in this background chapter we will present simplified schemes of the GPU architecture for the sake of brief overview. For more detailed and accurate schemes we suggest the reader to refer to NVIDIA documentation.

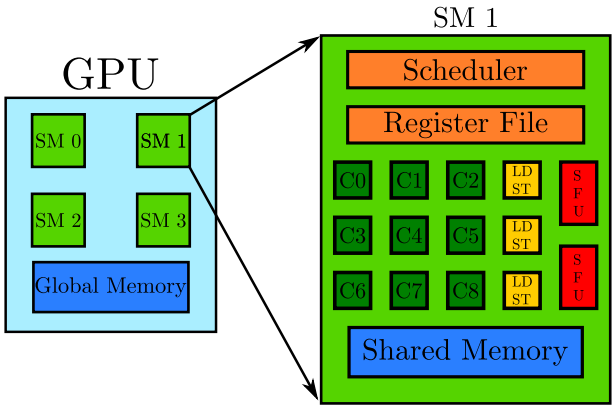


Figure 2.8: Simplified version of the GPU hardware architecture.

– *SFU*: special function unit that executes transcendental functions, such as sine, cosine, square root, and others.

The overall workflow of the SM is the following. The scheduler unit does the fetch and decode of each instruction; then, the elaborations required by the instruction are performed by N execution units in parallel on N different chunks of data. The parallel execution of the N threads is an enhanced version of the SIMD paradigm called Single Instruction Multiple Thread (SIMT), in which each thread has its private context, stored separately in the register file, but it shares the execution flow with a set of other threads. The scheduler of each SM is not able to manage an arbitrary number of parallel threads at a time. In fact, the scheduler partitions the threads into groups of threads, called *warps*; in the NVIDIA architecture, the warp has a fixed size of 32 threads each, and then the scheduler manages them accordingly. The SM is also able to manage the task-parallelism because the scheduler unit can dispatch multiple instruction streams in time-multiplexing. This enables the interleaved execution of multiple warps, maximizing the throughput, while hiding the latencies of slow operations, such as memory loads/stores. The task-parallelism is also addressed t the higher level of the GPU architecture (left side of Figure 2.8, which can schedule different warps among the available SMs, achieving a two-level task-parallelism that is similar to a general many-core system.

The GPU features a complex hierarchy of memories shown in Figure 2.9. The memory hierarchy is directly exposed to the program, thus, it has to be explicitly managed by the software engineer. Such a memory hierarchy consists of the following levels:

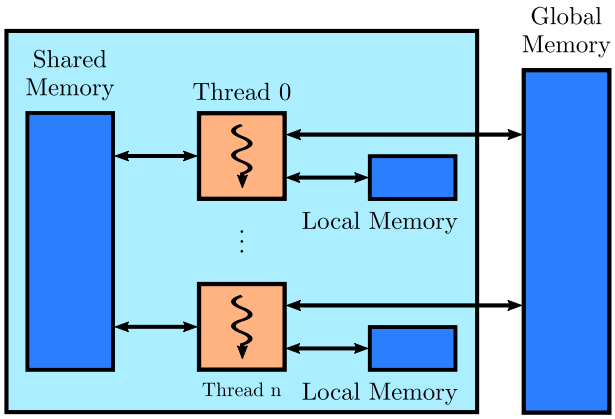


Figure 2.9: GPU memory hierarchy.

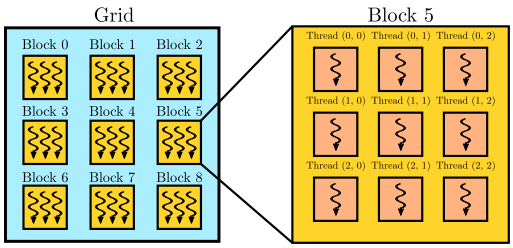


Figure 2.10: Grid of thread blocks.

- *Register File*: the fastest and closest memory available directly in the SM. It holds all the local variables allocated by each thread.

- *Local Memory*: it is another private memory for each threads placed again in the SM. It contains the variables that do not fit the register file.

- *Shared Memory*: the shared memory is a user-defined cache that is used to let threads in the same SM to cooperate by sharing data and achieving the maximum performances.

- *Global Memory*: this is the largest available memory in the GPU. Such a large memory cannot be installed directly in the SM but it resides off-chip. This involves a slower access to this memory compared to the shared memory.

2.3.2 Programming Model

The basic program unit that can be accelerated on the GPU is called *kernel*, according to the NVIDIA CUDA framework [21]. The kernel is actually a traditional function, that, differently from the classical CPU sequential execution, is automatically parallelized with a large set of threads by following the SIMT paradigm. The threads executing a kernel are structured into a matrix directly mapping the input/output data to be processed; each thread is identified by an unique index, as shown in Figure 2.10. Moreover, the overall grid may be partitioned in sub-parts called blocks. The threads' block has the peculiarity to be executed on the same SM, potentially in different warps; running on the same SM, therefore, the block's threads share the same shared memory to exchange data.

```
void vec_scalar(float* vec, float scalar)
{
    for(int i = 0; i < N; i++)
    {
        vec[i] = vec[i] * scalar;
    }
}
```

Sequential version.

```
__kernel__ void vec_scalar_gpu(float* vec, float scalar)
{
    int i = get_thread_index();
    vec[i] = vec[i] * scalar;
}
⋮
vec_scalar_gpu<<<N>>>(...);
```

GPU parallel version.

Figure 2.11: Linear scaling algorithm presented either in sequential and GPU parallel version.

Figure 2.11 presents two versions of the same algorithm, performing a linear scaling of a vector of N floating-point numbers by means of a constant. The traditional version targeting a CPU contains a loop that

performs the scaling operation on each single element in the vector. Instead, the GPU kernel function specifies only the body of the operation, i.e., the scaling, not presenting the loop, but acting on a single position of the vector.At runtime, this code will be concurrently executed by a number N of threads specified at the time of the kernel invocation. Each thread will therefore access on a single vector position, by using its identifier as the index. The effect is that the loop in the sequential code is transformed in a parallel execution of its various iterations, each one performed by a different thread.

2.4 Faults in Digital Systems

In the context of safety-critical or mission-critical systems, as the Autonomous Driving Systems (ADSs) considered as the working scenario in this thesis, resilience and reliability are two fundamental properties that such systems must comply with. As a consequence, it is fundamental to study the occurrence of faults in the devices and the effects they may have.

A *fault* is defined as a defect in the circuit causing a deviation of the system from its nominal behavior. In the classic literature [24], faults are classified according to their duration and persistence into three classes:

- *Permanent Faults*: represent irreversible physical changes in the device.
- *Intermittent Faults*: are usually caused by hardware instability activated by the variation of the working conditions, such as chip temperature or supply voltage. In general, they do not affect the system permanently but usually signal that a permanent fault is likely to happen soon.
- *Transient Faults*: temporary and reversible modifications generated by environmental conditions such as radiations, electromagnetic interference, power supply, and electrostatic discharge.

On the one hand, semiconductor manufacturing technology has significantly reduced the risk of occurrence of permanent and intermittent faults. On the other hand, technology scaling makes electronic circuits more susceptible to radiations, cross-talk or other noise sources because the noise margins have been reduced due to voltage scaling, and high operating frequencies. These phenomena can deposit unwanted charges in electronic

devices, thus producing glitches in the circuits, commonly referred to as *soft errors*. When soft errors occur, then it is possible to restore the device by resetting it or rewriting the interesting part. In particular, they mainly cause bit-flips in the memory cells, i.e., a bit value erroneously changes from 0 to 1 or vice-versa, thus representing a corruption of the processed data.

Historically, these phenomena have appeared in space-born electronics due to cosmic rays in 1975 [25], and then such phenomena have also been experienced at ground level in 1978 [26]. In both cases, the states of some bits had randomly changed, although the memory was not damaged. Among all the possible soft errors, this work focuses on Single Event Upsets (SEUs) that is the commutation of a single memory bit due to a particle strike.

As a matter of fact, transient faults nowadays represent one of the most relevant failure phenomena in modern electronic devices [27]. When considering ADSs and their need for safety-critical properties, this is a particularly challenging issue due to the large employment of GPU devices for accelerating computations. Indeed, modern GPU architectures have been the result of the discussed aggressive technology progress. It is worth mentioning, for example, the number of transistors in two top-class GPUs in different years; the NVIDIA GTX Titan [28], released in 2013, has 7.08 billion of transistors and fabrication process of 28 nm. In contrast, NVIDIA Titan RTX [29], released in 2019, has 18.6 billion of transistors and a fabrication process of 12 nm. Thus, GPU-based system must be accurately investigated in terms of susceptibility to soft errors and reliability. It is worth mentioning that NVIDIA devotes a large effort for the qualification of its products by means of radiation tests in order to allow their employment in automotive systems [30, 31].

Similarly to a CPU-based system, when a soft error occurs in a GPU accelerating an application, its final effect on the application can be classified as follows:

- *Masked*, if the output of the application does not differ from a golden reference. The soft error has not been activated since it may have occurred in a dead portion of the data that is not used by the program, leaving no trace of it.

- *Application Timeout*, if the application hangs forever without any possible recovery action except the restart.

- *Application Crash*, if the application suddenly terminates.
- *Silent Data Corruptions (SDCs)* [32], if the application ends successfully, yielding an incorrect output that differs from the golden reference.

Among these effects, masked effect can be neglected since they has no effect. Moreover, application crash and timeout do not represent a major hazard; indeed, they are detectable by the operating system that may apply proper countermeasures such as restarting the application. On the other hand, SDCs represent a critical issue since they can put the system into an incorrect state without any signal. As a conclusion, this thesis will focus on SDCs caused by SEUs within the GPU.

2.5 Related Work

In this second part of the chapter, we will review the literature related to the reliability analysis applied to the considered working scenario. We will first discuss fault injection tools for GPUs and later the methodologies for automating the analysis of the effects of faults in CNNs executed on GPU.

2.5.1 Fault Injectors for GPU devices

The classical approach to perform reliability analysis in digital systems is based on the fault injection. The strategy consists in injecting a fault in the circuit and analyzing its effects by monitoring the subsequent activity of the system. Fault injection may be performed by different means, by emulating the fault in the real system or simulating its effects in a model of the system.

Fault injectors have similar structures and follow common steps. The injector needs first to discover where to inject faults, creating a list of inject-able sites, which, depending on the extent, can be executable trace, assembly instructions, or source-code lines. This phase is either called *instrumentation phase* or *profiling phase*. From that list, inject-able sites are drawn to populate the injection list and mapped with the proper error value. This injection list can be either generated ahead of time and then reused for multiple inputs or generated during the execution of the campaign. The campaign is performed by selecting one record from the injection list, executing the program up to the injection site, replacing the

value, and then resuming the execution. The last optional part is the output analysis, which is context-dependent, and several implementations are possible. The injector can either return the raw output or can provide a classification of the fault according to user-defined comparisons. We here review the literature on fault injectors for GPUs; it is worth mentioning that we will present two of the discussed tools, SASSIFI and TensorFI, with much more details since they will represent the reference points for the work in this thesis.

2.5.1.1 CUDA-GDB

CUDA-GDB [33] is the NVIDIA tool for debugging CUDA applications. Although it has been designed for other purposes, it can be used to implement a fault injector. CUDA-GDB can freeze the execution of a program at any point, let the developer choose the variable of interest, replace its value to mimic the effect of the fault, and then resume the execution. Before executing the fault injections campaign, a profiling routine should retrieve the list of kernels and all their local variables. From the list, a kernel is randomly selected, and a breakpoint is set in its correspondence. The debugger stops the execution at the breakpoint and retrieves the variable's value, chosen randomly. According to the developer-defined policy, the value is replaced, and the execution continues. This technique is easy and fast to implement, has a great extent, and is capable of targeting the RF and the output of instructions. However, the involvement of a debugger requires the program to be compiled with the debug symbols, and it slows down the execution.

2.5.1.2 GPU-Qin

GPU-Qin [34] is a fault injector built using the CUDA-GDB framework introduced above, from which inherits the pros and cons. What makes it different from CUDA-GDB is the grouping phase, which is executed first of all. The threads within a kernel do not always execute the exact amount of instructions, causing divergence. This issue limits the GPU to express its full potential and should be minimized. Under this assumption, it is reasonable to group threads according to their divergence behavior. A thread is selected from each group and is executed in a simulator, GPU-Sim [35], which is able to extract a complete trace of execution. The profiling phase takes the traces and tries to map the executable instructions with their

corresponding source-code lines. The injector implements only the single bitflip as the fault model and is able to target the outputs of instructions, the Register File (RF), and the source operands of LD/ST operations, i.e., predicate instructions. The performances achieved are consistent with the methodology implemented. The grouping phase results to be slow since the debugger executes instructions step-by-step that degrades the performances since the step-by-step execution has user response time.

2.5.1.3 SASSIFI

SASSIFI [36] is the fault injector built by NVIDIA for its GPUs. It is the best tool so far targeting NVIDIA's GPUs, following all the methodological steps presented at the beginning of Section 2.5.1. It uses low-level instrumentation that classifies it as a micro-architectural fault injector. SASSIFI employs SASSI [36], another NVIDIA's tool, which can instrument GPU assembly instructions (SASS). The instrumentation tool inserts callbacks to the user-space function before or after assembly instruction, i.e., the micro-architecture. These callbacks allow retrieving the executed kernel's names, the execution count, and the assembly instruction codes together with their register or memory information. Instruction codes are grouped into classes of instructions, i.e., floating-points, loads, or instructions that write to the RF, to provide coarser inject-able sites.

SASSIFI proposes three modes of injections:

1. *RF Mode*: selects a random instruction, from any within a thread, and inserts a fault into one of the allocated registers for that thread.

2. *Instruction Output Value (IOV) Mode*: selects a random instruction, from the ones that write to the RF within a thread, and inserts a fault in the destination register *after* its execution.

3. *Instruction Output Address (IOA) Mode*: selects a random instruction, from the ones that write to the RF or memory within a thread, and inserts a fault in the destination register or memory address (ST) *before* its execution.

The injection sites list is made up of the injection mode, the error model (single or double bitflip, random or zero value), and the class of instructions to target, as shown in Figure 2.12. In the injection step, one injection site is selected from the list per application run and injected using SASSI. Then,

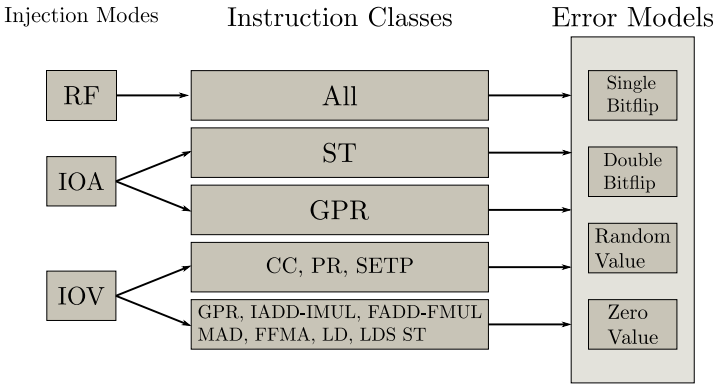


Figure 2.12: Injection modes, error models and instruction classes in SASSIFI.

the application is monitored to classify the execution's behavior, i.e., crash, hangs, SDCs, or masked results.

SASSIFI is the architectural GPU fault injector with the broader scope. It is one of the fastest and its low-level extent, leading it to become the first choice when it comes to GPU fault injector. On the contrary, it suffers from several limitations: to be instrumented by SASSI, the program is required to be recompiled for each injection mode (RF, IOV, and IOA). Compilation might be not so trivial on large projects and cannot be performed on closed-source libraries, like cuBLAS, excluding them from being injected. The injector is compatible up to GPUs of the Maxwell series (2014) and works only with Ubuntu 14 and CUDA 7.

2.5.1.4 LLFI-GPU

LLFI-GPU [37] is a GPU fault injector based on the open-source CPU LLFI fault injector [38]. It is a compiler-based fault injector that uses the LLVM compiler [39] to instrument the program and injects faults. The instrumentation phase is done by intercepting the calls of the CUDA compiler to LLVM. The number of kernel calls, threads per kernel, and the instructions executed are extracted during this phase. Besides, the code necessary to inject faults is added before being passed to LLVM. After the LLVM compilation, the *Intermediate Representation (IR)* is returned to the CUDA compiler to be converted into SASS assembly. The injection phase is common to the one described in Section 2.5.1. LLFI-GPU is faster than injectors based on debugging but suffers from the need to recompile the code and a narrower extent than SASSIFI.

2.5.1.5 TensorFI

TensorFI [40] is a high level error simulator for TensorFlow that evaluates the resilience of ML applications. This simulator differs from others because it is built to be high level, abstracting the whole computing architecture and not relying on any platform. The injection of ML applications presents different challenges than traditional architectural injection on pure CNNs. The operators used in CNNs are a subset of the ones available in an ML framework, and a ML application can include abstract operators that give no clue about their implementations or architectural details. Usually, such applications run on heterogeneous systems, operators target the best execution device available, such as GPUs, CPUs, or even TPUs. The heterogeneity makes it difficult to perform architectural fault injection because too dependent on the implementation rather than functionality. TensorFI moves the scope of the fault injection from the architectural level to a higher level, and this enables the possibility to target a wider range of applications that otherwise will be excluded from traditional fault injection because of their implementation details or complexity. TensorFI has been designed to meet three goals:

- *Ease of Use and Compatibility*: the injector should be as transparent as possible to either the developer and TensorFlow.
- *Portability*: the injector should be attachable to TensorFlow without requiring any recompilation or modification of the framework itself by the user.
- *Speed of Execution*: the injector's behavior should not interfere with the normal execution of the TensorFlow's graph. The operators should be executed on the best available device, so campaigns should be reasonably fast time.

The injector is built as a two-phase injector, and its granularity is the operator of the TensorFlow's graph. During the *instrumentation phase*, the original graph is replicated operator-by-operator, which ensures the compatibility goal and provides sufficient automation that abstracts the graph details to the user. The replication inserts the necessary code to inject faults within the output of such an operator and the control logic to trigger the injection. At the end of this phase, two graphs coexist, the original and the faulty one, which will be used during the *execution phase*. The injector

relies on a configuration file for the campaign targets and settings and also serves the purpose of minimizing the source-code intervention performed by the user. The fault models, which are customizable in the configuration file, are random number, zero, single and, multiple bitflips. These fault models are either applied to a scalar, an element of the output tensor, or the whole output tensor. The fault modes are not the only parameters required to set up the campaign. The user is also required to specify the injection's probability of any operator it is interested in injecting together with the number of instances in the model of such an operator, necessary for the control logic that triggers the injection. The execution phase selects at each iteration an injection site, i.e., an operator, according to the probabilities and instances defined in the configuration file, executes it in the faulty graph, replaces its output with the fault model chosen and then resume the execution, returning the model's output.

The fault modes embedded in TensorFI represent a design flaw because these are valid within the architectural level while no validation is provided that these are still accurate for the functional level. Our experiments will show how architectural faults, such as bitflips, are propagated to the functional level, i.e., ML operators' output, contesting the TensorFI methodology.

Other flaws have been detected in TensorFI from an implementation point of view, and these mines the design goals on which is built. Certain operators, mainly the arithmetic ones (multiplication, addition, argmin, etc.), are replicated with their NumPy's implementations. Although the semantics of the operation is preserved, two implementations of the same algorithm, especially when managing floating-point values, can lead to two slightly different outputs that may represent an issue if the context demands high accuracy. The set of operators that are replicated by TensorFi are a subset of the ones available in TensorFlow and are hard-coded within the tool, so using this injector for a real-world model may not be possible if the operators are not inject-able by TensorFI. This does not represent a remote possibility because popular operators, such as LeakyReLU or Batch Normalization, are not available, and the tool is not able to bypass unknown operators to allow at least to run the model. Indeed it will crash because not recognizing such operators. The developer is required to modify the tool's source-code if wants to enable new operators because the tool is not designed to be extended but only internally modified. These two limitations compromise the compatibility, transparency, and portability claimed by the

tool.

The execution phase is based on the environment set up by the instrumentation phase and heavily relies on records that are listed in the configuration file. The tool to properly work requires the user to insert in the configuration file the exact number of instances for each inject-able operator, from the ones and only from the ones that are hard-coded in the tool, as explained above. An incorrect number of instances leads to a wrong injection policy that will fail to select the right injection site bringing two possible situations. A sub-sampling error reduces the pool of inject-able operators when the instances count is underestimated. A super-sampling error wastes iterations, not injecting any operators when the instances count is overestimated. Large models can have hundreds or thousands of operators, and accurately counting them, considering only the operators that are managed by TensorFI, is not so trivial if not automatized and may vanishing the ease of use property.

The speed of execution will be quantitatively analyzed in a further chapter. Still, TensorFI is faster than any architectural fault injection, even SASSIFI. Its speed is due to the fact that it belongs to the application domain. Operating at the application domain does not involve any of the strict constraints of the architectural fault injection, mostly time and memory overheads. However, the performances are better than the GPU fault injectors, but, as we will demonstrate, are still far from being optimal. The replication of the graph is done in a way that each time an operator needs to be executed, TensorFI opens a TensorFlow's session. The opening of a TensorFlow's session is a non-negligible operation that has a cost, quantified in time. The time overhead derived from opening the TensorFlow's session is even more emphasized if the operator is accelerated on GPU because it also has to occur the memory transfer between CPU and GPU.

2.5.2 Methodologies for Reliability Analysis

The reliability analysis of ML applications accelerated on GPU is used to identify the most brittle operators or GPU kernels to faults when such applications are deployed in safety-critical systems. The typical analysis tries to correlate the injection of an architectural fault at the *i-th* layer, operator, or GPU kernel, with the output of the ML model. This analysis leads to identifying the parts that, if corrupted, bring the output of the model to deviate from its nominal behavior. The outcome of the reliability analysis is usually a statistical model that associates each layer with its

probability of deviating from the nominal behavior, or, if the accomplished task allows it, with the probability of a custom error class. The reliability analysis represents the first step that allows us to develop custom hardening strategies that try to mitigate the effects of faults when such applications are used in safety-critical systems.

In the next sub-sections, we will describe the current state-of-the-art works on the reliability analysis of CNN accelerated through GPU.

2.5.2.1 Kernel and layer vulnerability factor to evaluate object detection reliability in GPUs

The work in [41] proposes two new vulnerability factors to evaluate object detection applications on GPUs. *Kernel Vulnerability Factor (KVF)* and *Layer Vulnerability Factor (LVF)* indicate the probability of faults in a kernel or layer to affect the output. These factors are used to define a new hardening strategy that cleverly aims to replicate only the most corruptible kernels or layers. Instead of replicating the whole program or hardware components, only the most brittle and weak tiers can be replicated, increasing the performances and reducing the waste. These metrics are assessed for two algorithms: *Histogram of Oriented Gradients (HOG)* for KVF [42] and *You Only Look Once (YOLO)* [43] for LVF. HOG is a gradient-based algorithm that can detect objects by grouping adjacent spatial regions, based on their gradients. The KFV for HOG is evaluated using fault injections at both architectural and high level, using either NVIDIA SASSIFI and CUDA-GDB. YOLO is a popular CNN for object detection that achieves high accuracy and low inference time. The LVF is evaluated using fault injections only at the architectural level, using NVIDIA SASSIFI. The error model used in CUDA-GDB is the random value, whereas NVIDIA SASSIFI uses the bitflip and two modes: RF and IOV. The IOV campaign resulted in a higher impact than the RF campaign, but both are overtaken by CUDA-GDB, meaning that the high level faults are likely to produce an SDC. Based on the KVF and LVF analysis, a *Double Module Redundancy (DMR)* strategy is used to replicate the sensitive kernels, and layers achieving a high percentage of SDCs detected, up to 80%.

2.5.2.2 Understanding Error Propagation in Deep Learning Neural Network (DNN) Accelerators and Applications

The work in [44] studies the impact of faults in DNNs and proposes solutions to mitigate their effects. It targets four popular convolutional neural networks: ConvNet, AlexNet, CaffeNet, and NiN. These networks are executed on a DNN simulator, Tiny-CNN [45], which has been appropriately modified for fault injections. The work aims to examine the effects of faults in DNNs, executed on DNN accelerators, and classifies the error propagation according to the network topologies, data types, layer positions, and types. The analysis highlighted that corrupting the exponent bits of a floating-point value are likely to induce an SDC, while not for the mantissa and sign bits. However, the SDC probability is higher if the data type is wider, i.e., 32 bits floating-point against 16 bits floating-point. Bitflips are not symmetrical; a transition from 0 to 1 is likely to impact more than a transition from 1 to 0. Finally, the work identifies in the *Local Response Layer (LRN)*, introduced in AlexNet [46], a valid helper that normalizing values and can mitigate the effect of large deviations in the values' domain.

2.5.2.3 Analyzing and Increasing the Reliability of Convolutional Neural Networks on GPUs

The work in [47] evaluates the reliability of GPUs, executing three CNN benchmarks: YOLO, Faster R-CNN, and ResNet, exposed to a real radiation test through neutron beams. The GPUs under test belong to three architectures: Kepler, Maxwell, and Pascal, both developed by NVIDIA. The radiation test poses attention to the electronic implementation of GPUs: FinFET devices (Pascal series) have an error rate that is one order of magnitude lower than CMOS devices (Kepler and Maxwell series). The radiation test also results in a higher percentage of crashes than SDCs, which is the opposite of the software-based fault injections. The experiments confirm the tendency of fault to spread across several threads within the architecture of GPUs and that deeper networks (ResNet and Faster R-CNN) are more likely to experience SDCs and crashes than YOLO. The evidence that from 67% to 82% of GPU processing is spent on GEMM has suggested adopting an *algorithm-based fault tolerance (ABFT)* strategy for matrix multiplication [48] that can correct up to 87% of critical errors. The second part of the work is focused on software-based fault injections, through SASSIFI, targeting YOLO, which leads to a redesign of

the max-pool layer that detects up to 89% of critical SDCs.

2.5.2.4 A Reliability Analysis of a Deep Neural Network

The work in [49] is a straightforward experiment that tests the reliability of two CNNs, YOLO and LeNET, against faults. The experiment is based on the open-source framework *darknet* [50] and uses a custom fault injector that is built upon it. Unlike traditional GPU fault injector, this version targets only layers' weights by inserting a fault (bitflip) into a weights' element of a specific layer before its execution and restoring it before the next run. The classification of the faulty output follows a traditional structure: *Masked* if no differences are reported w.r.t. the golden referece; *Safe* if the absolute difference is less than 5%; and *Unsafe* if the two previous conditions are unmet. The results of the experiment confirm that exponent bits are more sensitive to faults and cause critical behaviors.

2.5.2.5 Increasing the Efficiency and Efficacy of Selective-Hardening for Parallel Applications

The work in [51] proposes a methodology to perform selective hardening instead of performing traditional hardening techniques like DMR and *Triple Module Redundancy (TMR)*. The aim is to increase the reliability of parallel codes while minimizing the cost and the overhead induced by replication. The strategy focuses on the source-codes and ranks the code portions based on the probability that corrupting them leads to impact the output of the program (PVF) and on the memory overhead. The ratio between PVF and the memory overhead allows us to find a good balance between efficiency and efficacy since considering only PVF may introduce excessive overhead. The rank helps to identify the most sensitive portions, and their replication can protect up to 60% of the source-code costing only 3% of overhead.

2.5.2.6 Evaluation of Histogram of Oriented Gradients Soft Errors Criticality for Automotive Applications

The work in [52] targets Pedestrian Detectors, implemented through a HOG algorithm and *Support Vector Machine (SVM)* classifier, that runs on two different GPU architectures. The output of this task is a set of bounding boxes, a structure that contains the size and position of the identified object within the image in pixel units. The paper proposes a classification of such bounding boxes when objectives of fault injection campaign. When

comparing a faulty output with a golden reference, the first parameter to evaluate is if the number of bounding boxes differs. That difference can be either 0, > 0, or < 0. The case where the difference is zero is a *necessary but not sufficient* condition of masked execution. The second parameter is the center of mass, which catches the presence of shifts along the x-axis and y-axis among faulty outputs and the golden reference. The last parameters are *Precision* and *Recall* that two common metrics to evaluate the accuracy of a detector. The GPUs are under test for either a radiation test and software-based fault injection, through a CUDA-GDB-style injector. Both experiments agree on the conclusion: the algorithm is robust against SDCs, whereas it experiences a lot of crashes, especially in the radiation test. The software-based highlighted the code sections that should be hardened to reduce the number of critical errors.

Before proceeding, let us sum up what has been mentioned above about GPU fault injectors and the reliability methodologies. GPU fault injectors are accurate and integrate fault models as closest as possible to the physical event. However, they are costly because the architectural fault injection is slow, and the application must respect all the constraints exhibited by the fault injector. Therefore, it is interesting the error simulation because it overcomes the constraints of the architectural fault injection, especially the speed with which it is possible to obtain usable results, but the error simulators need to employ validated error models; otherwise, the analysis will be useless. Thereby arises the need for building error models built upon the architectural fault injection, which are valid at the application layer, resorting to the architectural fault injection once and then reiterating the error models built.

For what concerns the reliability methodologies, they are always focused on classifying the errors in terms of ML model's output. This vision leads to produce results that are difficult to port to other applications than the one that has originated it because it is too dependent on the target. What can really help the reliability analysis of safety-critical systems is to define a reliability analysis and methodology focused on each operator of the ML model, instead of the output. Achieving that, it is possible to port such analysis also to other applications, gaining generality and abstraction, which, if combined with error simulation, can open scenarios difficult to reach using the traditional approaches.

In conclusion, in this chapter, we have presented the background on

the target system we would like to consider in this work, consisting of the CNN application running on a GPU. Then, we have analyzed the tools and methodologies for the reliability analysis of this kind of system, discussing benefits and limitations. Given these basic, in the next chapter, we will present the goal of this thesis and then, in the rest of the work, the proposed solution.

Chapter 3

Goals and Requirements

Based on the background discussed in the previous chapter, we here present the working scenario addressed in this thesis aiming at focusing on the reliability analysis of Convolutional Neural Networks (CNNs) executed on Graphic Process Units (GPUs). After that, we will discuss first the requirements of this task, also commenting on the limitation of the current practice in reliability analysis through fault/error injection we noticed during the review of the literature. Finally, we will present the goals of this work, and we define the Key Performance Indicators (KPIs) and baselines we will use to evaluate the achieved results.

3.1 Working Scenario

In Chapter 1, we described a motivating example, in which a soft error was presented in the perception module, leading to the wrong classification and the wrong placement of an obstacle. The perception module is backed by several CNNs running on GPU. The example described may be originated by the following chain of events:

- There is a CNN, executed on a GPU, that processes an image.
- Suddenly, a Single Event Upset (SEU) manifest itself within the GPU architecture, by flipping the state of a memory cell.
- The corrupted memory cell is currently used and consumed by the CNN application, leading the output of the CNN to deviate from its nominal behavior, i.e., misclassifying.

- In such a scenario, the SEU has turned into a Silent Data Corruption (SDC).

- Finally, the SDC is propagated to the application level, posing the system into a critical state.

CNNs and GPUs are, respectively, computing models and accelerator devices that are not safe by design. CNNs are designed to achieve the highest accuracy, and their models are not intrinsic robust [53, 54, 55], although there exists a shared belief that they are. GPUs are designed for performances, and only in recent years have been started to be deployed in safety-critical systems, but they present some weaknesses for what concerns the faults in their architectures, as reported in works [47, 56].

The duo CNN and GPU, when used in safety-critical systems, like autonomous driving, require an accurate evaluation of its behavior when faults occur, in particular SDCs. Therefore, in this thesis, we consider a CNN application, executed on GPU, managing images. There is the quest, therefore, of new frameworks capable of evaluating the reliability of such a combination in its application domain, providing a validated error model that is representative of the GPU and is built upon the observed SDCs obtained through GPU fault injection campaigns. The framework should anticipate the reliability analysis already from the development stage with a faster timing, compatible with the requirements necessary for the system, and with lower complexity.

3.2 Constraints and Current Limitations

The reliability analysis of a complex safety-critical system is generally challenging because current methodologies and tools do not properly connect the architecture/device level where faults are generally modeled and the application one where the error propagates in the functionality. The main reason is that the evaluation of the hardware device, through architectural fault injection, poses severe and numerous constraints that make it difficult to port the results at the abstraction level of the application execution. On the other hand, when simulating errors directly in the application, the lack of error models makes the analysis not effective nor significant. In the next sections, we discuss in detail these limitations of each of these worlds, describing the implementation and methodological limitations of

either architecture-level fault injection and application-level error simulation.

3.2.1 GPU Fault Injection

The fault injection is the current practice for the reliability assessment of systems; it is supported by a strong background and is closed to the reality of events. However, the fault injection in GPUs has few shortcomings, mainly related to execution times and flexibility. In the next paragraphs, we will detail each of these factors.

Recompilation The integration of many GPU fault injectors, either debugger- or compiler-based, with their target applications requires the target applications to be re-compiled along with the GPU fault injector. In the case of closed-source or legacy applications, such an integration cannot be performed. Many external libraries that are widely used in Machine Learning (ML) frameworks, such as cuDNN [22] or cuBLAS [57], are provided as closed-source. These situations severely limit the applicability and integrability of GPU fault injectors with their target applications.

Execution Times and Limits Execution time is the principal cost of the architectural fault injection because the time required to perform a campaign is higher due to the broader scope and so the broader injection space. The techniques used to inject faults, either by instrumentation callbacks or debugger stop-and-resume, are not negligible and executed for every instruction to discriminate if the injection site has been reached or not. For instance, in SASSIFI [58], the time overhead, quantified as the ratio between the instrumented and clean execution time, can range from 1.02x up to 166x. The increased execution time is problematic not so much for the single execution but for the execution of the campaigns, in which the same instance is executed hundreds or thousands of times. In this case, the time overhead accumulated by each execution can become prohibitive, and the duration of a campaign can last up to hours, days, or weeks. Another limitation is due to the slowness with which the GPU fault injectors load and allocate the memory from the CPU to the GPU. Such a slowness makes the architectural fault injection to be likely unfeasible with CNN containing gigabytes of data, requiring to analyze each layer or operator in an isolated environment.

Flexibility The traditional approach adopted in the analysis of CNNs, executed on GPU, through fault injection campaigns, is to compare the output of the network with a golden reference. This approach, even though it is valid, lacks flexibility because we can only correlate the *ith* operator of the CNN, in which it is inserted the fault, and the output of the network. This lack of flexibility addresses the following issues:

- It is not possible to derive a typical behavior of the single operator that is portable or reproducible to other networks.
- There exist errors that are manifested in the output of a specific operator but are then absorbed by the other layers of the network, not resulting in a deviation of the network's output.
- It is not possible to force the generation of a specific error in the output of the network because fault injectors are not so responsive.

All these reasons lead the results of the analysis of a CNN executed on GPU to not be portable or reproducible to other networks because all the results are linked to the output of the network, which changes from network to network.

3.2.2 Error Simulation

As discussed in the literature review, the error simulation is an alternative approach for the reliability analysis of safety-critical systems, generally adopted in other research fields such as the ones on middle-wares and distributed systems. It aims at overcoming all the constraints imposed by the architectural fault injection by directly injecting errors in the application execution. The error simulation works at the application level with a coarser granularity than the architectural fault injection. Thus, the errors used in the error simulation need to be different from the well-known and validated fault models used in the architectural fault injection, but still, have to be representative of the behavior of the GPU when affected by a fault.

Lack of Error Models The error simulation and the assessment of the application can be disjoint from the architectural fault injection. The literature has exhaustively studied the architectural fault injection, so the fault models used in that field are validated and representative of a physical event. Porting those fault models as-is (or some variants of them, as

in [40]) at the application level is not legit. In fact, the effects of the faults at the functional level on the application's output or in some intermediate step are different and more elaborate than a simple corruption of scalar values, as generally occurs within the registers at the architecture level.

In our scenario, in order to replace the classical fault injection with the error simulation, there is the quest of validated error models capable of describing the effects of the faults in a GPU running a CNN application. Such error models need to be defined in terms of corruptions of tensors, being the intermediate data exchanged by the various operators of a CNN in a more sophisticated way than the classical single or multiple scalar value corruption, as in [40], since this strategy is not representative of the real effects.

Lack of Available Tools The current state of availability of error simulator tools for CNNs executed on GPU is lacking. Well-designed errors simulators for this context, capable of exploiting all the execution techniques widely used in many architectural fault injector, do not exist yet. TensorFI [40] is the only error simulator that is publicly available for this context. However, its implementation is not well designed because it does not take advantage of the ML framework, which is built upon and does not include typical strategies used by traditional fault injectors to speedup the execution of the campaigns, such as check-pointing or cache. The result is a tool that finds it hard to be employed in real case scenarios because its current state is closer to a prototype than a complete and usable product.

3.3 Contributions

The goal of this thesis is to provide a methodological framework for the reliability assessment of CNNs, executed on GPU. The framework is composed of two parts for the error modeling, and the error simulation, respectively, and it is directly integrated into a popular ML framework.

The error modeling is a methodology that shifts the errors obtained though GPU fault injection campaigns towards the application level. The error modeling methodology operates on each CNN operator, characterizing the errors that appear in its output through GPU fault injection campaigns. This approach creates a correlation between the fault injected within the operator and its output. The whole error modeling is performed once because once extracted, the error models are representative of the GPU

behavior, and the architectural fault injection is no longer required. The error simulation is a methodology that enables the reliability assessment of CNNs in the application domain, by injecting errors in the output of CNN's operators.

The error simulation relies on the errors modeled according to the proposed methodology, and integrates itself directly in the application, overcoming all the issues derived from the architectural fault injection. The error simulator is well-designed, completely integrated with the ML framework, and employing techniques, like the check-pointing, to accelerate the execution of the campaigns.

The advantage offered by this approach is the capability to inject errors with the same accuracy of the architectural fault injection, and, at the same time, with the higher flexibility and simplicity offered by the error simulation, and with shorter execution times.

3.4 KPI and Baseline Approaches

The proposed and developed framework is evaluated against the following KPIs, which we consider essential to overcomes the most critical issues among the ones presented in Section 3.2:

- *Accuracy*: we aim that the errors, injected with the error simulator and modelled according to our methodology, produce the same effects in the output of the whole CNN network like the ones produced by the architectural fault injection campaigns. The error models are representative of the effects of the SEUs injected in the architecture of the GPU. We have modeled these effects upon the single CNN operator's output, which represents the granularity achievable at the application level. Therefore, the error models are validated by construction in the case of the single operator's output. Although that, we want to ensure that the effects in the output of the whole CNN network are the same, or at least comparable.

- *Execution Times*: We aim to obtain for our error simulator execution times that are lower than the architectural fault injection and other error simulators when performing the same campaign. Given a fixed number of faults or errors, depending on the context, to insert in a CNN, the execution time is the elapsed time from the first injection to the last one. We want that our error simulator is capable of inserting

that number of errors faster than the baseline for the architectural fault injection and error simulation.

Our framework is evaluated on that KPIs and compared to two baselines, which represent the state-of-the-art for the architectural fault injection and error simulation:

- SASSIFI [58] is the best available for what concerns the GPU fault injection. Its fault models are validated, and its micro-architectural scope makes it the most precise available tool.
- TensorFI [40] is one of the newest solutions in the field of error simulation for CNNs. Its scope and approach make it the direct competitor of our error simulator.

In this chapter, we have introduced the limitations of the current state-of-the-art approach and presented the KPIs on which we evaluate our framework. In the next chapter, we are going to introduce the methodological framework proposed in this thesis.

Chapter 4

The Proposed Framework for Error Modeling and Simulation

This chapter presents the main proposal of this thesis, which is the methodological framework for the reliability analysis of Convolutional Neural Networks (CNNs) applications executed on Graphic Process Units (GPUs) based on error simulation. We first provide an overall view of the framework by discussing the key ideas at the basis of it. Then, we will detail the various phases of the designed methodology and, finally, we will present the prototypical implementation. This implementation will be employed in the next chapters in real-world cases.

4.1 An Overview of the Methodology

The discussion of the previous chapters has shown how the reliability analysis of a CNN running on a GPU (and, more in general, of any software-based system) relies on two different abstraction levels, the architecture level and the application one. Indeed, fault models have been widely validated at the architecture level, thus offering a realistic approach for their emulation. On the other hand, the analysis of the fault effects has to be performed at the application level since the interest is in understanding how the system functionality deviates from the nominal one. The main limitation of the current practice is, therefore, the fact that the two abstraction levels are unconnected, thus limiting the capabilities of performing a reliability analysis.

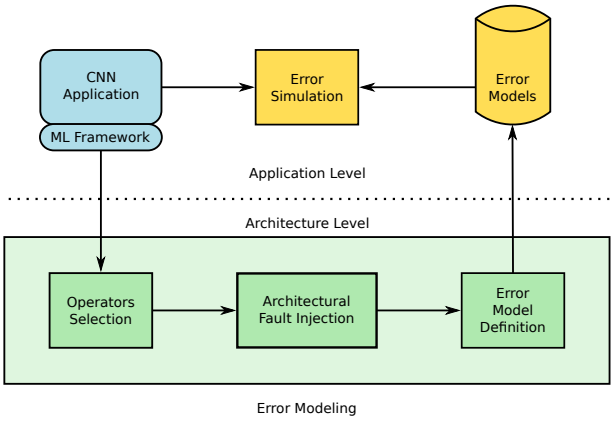


Figure 4.1: Methodological Framework

The goal of the methodology we propose here is to connect these two abstraction levels by means of a methodology for a systematic analysis of the effects of the faults injected at the architecture level into the application behavior by defining specific error models, which are validated by design. These error models enable, therefore, an accurate and realistic error simulation directly on the CNN application under analysis. As a result, this approach is able to overcome the lack of error simulation, thus offering the possibility to the benefit of its higher flexibility in the analysis and performance in terms of execution time than the classical fault injection.

The first step in this error modeling and simulation is, therefore, the identification of the basic element in the CNN model to consequently define the error models as the corruptions of its nominal behavior. As discussed in Section 2.2, all the Machine Learning (ML) frameworks express the ML models, and consequently, the CNNs models, in the form of a data-flow graph. Each data-flow graph is populated by many CNN operators, each one performing an elaboration on a set of input tensors (multidimensional matrices of data), and producing one or more output tensors. Therefore, we identify the CNN operator as the basic element we will consider in the error simulation. Then, due to the fact that we aim at focusing on Silent Data Corruptions (SDCs) as discussed in Section 3.1, we define error models based on the corruptions of the output tensor of the CNN operators.

Figure 4.1 presents the overall methodological framework we propose in this work. The framework is horizontally divided into two parts according to the two considered abstraction levels, the bottom part devoted to the error modeling while the top part to the error simulation. The **error**

modeling aims at defining the error models for each CNN operator used in CNN applications. This part is composed of the following activities:

- **Operator selection.** The input of this part of the framework is the list of all operators available in a ML framework. Being the list of all ML operators quite large, and each operator parametric, the first step is devoted to the identification of the minimal representative set of operators and their corresponding parametrization.
- **Architectural fault injection.** An extensive fault injection campaign with a GPU fault injector is performed on each CNN operator of the set identified in the first step. Then, being interested in considering SDCs, we collected the raw results for all the runs producing in a corrupted output tensor.
- **Error model definition.** Finally, the raw results of the fault injection campaigns performed on each CNN operator are analyzed to extract recurrent corruption patters and consequently defining the corresponding error models. The final output is a repository of error models.

The output of the error modeling is a repository of error models. Such a repository is then used in the top part of the methodological framework by an **error simulator** implemented on the top of a widely-used design framework for ML applications and directly acting on the CNN model.

Within the discussed methodological framework, the error modeling is the most onerous and time-consuming activity. However, this step is executed once to set up the repository. On the other hand, the error simulation can be performed countless times using the models in the repository and can target any possible CNN design. The error modeling is the transitioning step that allows the two levels to communicate because it takes as input the faulty tensors obtained through the architectural fault injection and returns the application-level error model. The models derived using this methodology are specific of the operator under analysis and not of the network they belong to. This peculiarity enables us to reproduce the same error model for a CNN operator to different instances of it even belonging to different networks, assuming the input data distribution as normalized or at least comparable. Therefore, this framework allows us to overcome the limits presented in the state-of-the-art works on the reliability analysis of CNNs executed on GPU since their methodologies are difficult to be applied in a different network from where it is applied.

In the next sections, we will describe in detail each step of the framework.

4.2 Operators Selection

The fault injection is not performed to carpet upon each operator of the CNN. Indeed, we select the smallest subset of operators that is meaningful for error modeling under the same conditions. Among the instances of the same operator type, the selection criterion is to privilege the instance that manages the smallest data in terms of shape. The reasons why we prefer the instance that manages the smallest data are:

- Lower loading and execution times of the operator's instance, which facilitate and reduce the execution time of the whole fault injection campaign.
- Smaller outputs to analyze because the error modeling phase works better by generalizing towards bigger tensors than the vice-versa.

The instances of the same operator's type but with parameters, for example, a convolution with 3×3 kernels and a convolution with 5×5 kernels, should be considered as two different operator types because they may generate two different error models.

After having identified the set of operator's instances to inject, they should be isolated and restricted to a confined environment. It is necessary to extract for each operator's instance its inputs, outputs, and parameters in order to be executed in the environment. Therefore, such an environment is considered as a stub, composed of the following steps:

- *Preamble*: loads of the inputs, outputs, and parameters of the operator's instance.
- *Execution*: the run of the CNN operator's instance.
- *Epilogue*: computes the difference between the computed output tensor and the golden output loaded before.

4.3 Architectural Fault Injection

The architectural fault injection aims to characterize the output of the CNN operators to faults inserted in the GPU kernels that make up their

implementation. This phase presents all the steps to follow to be able to execute a fault injection campaign over a CNN operator.

4.3.1 Campaign Sizing

Sizing the fault injection campaigns is crucial to obtain relevant results for the error modeling. An underestimate number of faults can lead to results that are not statistically relevant or accurate; for instance, with a small fault list, some error may be not generated, thus causing an imprecise error model characterization. On the opposite, an overestimated number of faults may not provide any additional information and results in a waste of time and resources. In our context, the inject-able sites are the GPU assembly instructions, which make part of the GPU kernels of the operator's implementation. The number of inject-able instructions, although it is finite, can be enormous, up to be considered infinitely large, if considered with multiple inputs.

Therefore, we adopted an ad-hoc approach for properly defined the campaign size, taking into account explicitly the architecture of the GPU. The data and thread parallelism are two mechanisms that the user is aware of. Each thread does not manage the whole available memory, but only a small fraction of it and executes the same instructions of other threads. The reasonable number of GPU assembly instructions to consider is not the whole number of instructions executed by all the threads but only the number of instructions executed by one thread, assuming the computation equal and uniform for any thread, according to the Single Instruction Multiple Thread (SIMT) paradigm. Depending on the flexibility of the GPU fault injector, the assembly instructions considered for the campaign can be restricted to a subclass or family of instructions, for example, considering only the floating-point instructions or only the memory instructions (LD/ST). This flexibility allows defining different campaign sizes for each of the family of instructions. Given the following parameters:

1. T: number of threads.
2. M: global memory allocated for the kernel.
3. N: number of instructions either all or of one family of instructions.

The estimated campaign size n provided by the heuristic evaluation is:

$$n = \frac{M}{T} \times N$$

This estimation, however, can be adjusted and rounded up to the upper thousand according to the user's preferences.

4.3.2 Fault List Definition

Once the operators have been selected and the campaigns are sized, all that remains is to execute those campaigns. To execute a campaign, the GPU fault injector needs to generate the fault or injection list. This action is nearly always dependant on the chosen tool and varies with it. However, the generation of the fault list requires the chosen campaign size and the fault model that we want to inject. In all the methodology, we consider the fault model the single bitflip for two reasons:

1. It is the most used and validates model for the representation of a particle strike. This work [59] shows how little is the difference between a single and double bitflip, so we stick with the single bitflip.

2. Other fault models, like the random or zero models, have not been considered because the input distribution of each operator does not require such models. The ranges of values are confined, and they do not span all the possible values within the floating-point domain. The ranges are controlled either by the normalization layers, which help to have a zero mean and unitary variance tensor, and the activation functions, which have narrow co-domains.

The high number of values that are present within an input tensor combined with the SIMT architecture of the GPU induces a high degree of randomness that, given the high symmetry of threads within the GPU, justifies the usage of few datasets of inputs and outputs for each execution of the operators.

4.3.3 Campaign Execution

After having sized the campaigns, extracted the datasets, and generated the fault lists, the campaigns for the various CNN operators are executed one at a time, and all the faulty tensors are collected and stored for the subsequent analysis.

4.4 Error Model Definition

The definition of the error models is the core of the proposed methodology. It represents the way of characterizing the errors, derived from the architectural fault injection, towards the application-level by classifying them with the parameters that the application is able to understand. The basic unit of any CNN is the tensor, so every result to be managed and interpretable by the CNN must be expressed in the form of a tensor. We aim to derive classes of errors that are common to all the instances of the same type. Before jumping to the formal specification of the errors' classes is necessary to state formally what we consider an error, which definition will hold in whole work. Given two floating-point values v and v', which represent the golden value and the output value respectively, then v' is considered as an *error* if the following inequality holds:

$$|v - v'| \geq \varepsilon \qquad \varepsilon > 0$$

The term ε is a positive small number that defines when two floating-point are assumed to be equal. This threshold is context-sensitive, so the correct value depends on the context and the accuracy required, always keeping in mind that the floating-point arithmetic is not associative, as presented in the well-known document in [60], thus different implementations may lead to two results that are slightly different. In this work, we have considered being reasonable to choose a ε equal to 10^{-3} for the CNN application, justified by the high regularity of the data because all the tensors are normalized, i.e., they exhibit zero mean and unitary variance. Given the definition of what an error is, in its turn, we define a faulty tensor or output, a tensor that has *at least* one error with respect to the golden reference.

The classification of the faulty tensors is performed according to three parameters: the cardinality, the spatial pattern, and the domains of corrupted values. These three parameters allow to classify each faulty tensor and to build a repository of errors that are representative of the ones observed in the architectural fault injection but apply to any operator's instance of the same type, providing enough abstraction and generalization to apply them also to tensors that do not have the same shape as the observed ones. More important, these three parameters can be described in the form of an algorithm so the models are representable as three functions that applied to a tensor, it generates a new tensor according to the

observed errors.

4.4.1 Cardinalities

The first parameter by which the faulty tensors are analyzed is the cardinality. The cardinality is an intuitive concept that counts the number of errors that are present in the faulty tensor with respect to the golden reference, according to the definition of error provided at the beginning of this section. From analyzing all the faulty tensors of a CNN's operator is possible to build a histogram in which each cardinality is assigned its probability considering the total number of cases in which that cardinality has appeared compared to the total number of faulty outputs. The probability map, represented as a histogram, is the first evidence of the differences between the architectural fault injection and the functional error simulation. The single fault, i.e., the single bitflip, at the architectural level, can result in one or more than one error in the output of the tensor. Finally, the probability map of cardinalities allows a higher degree of flexibility of the system evaluation because the analysis can be focused on a specific cardinality or a subset of them since it is a statistical model.

Algorithm 1 presents the steps to extract the cardinality given a faulty tensor and the golden tensor. The first step is to compute the absolute difference between the faulty tensor and the golden tensor, obtaining a difference tensor. Recalling the definition of error given at the beginning of this chapter, each element of the difference tensor is checked if greater than the threshold, obtaining a boolean equality tensor. If the comparison holds, then it represents an error, and its location in the equality tensor is set to True, otherwise False. The cardinality represents the number of True elements, which is obtainable by summing up the equality tensor. Then each cardinality will feed into a dictionary that represents the histogram for the operator.

4.4.2 Domains of Corrupted Values

The analysis of the domains of corrupted values aims to determine, in a qualitative way, how the faulty value deviates from the golden value in terms of magnitude. This classification is necessary because the error simulation must be able to reproduce the same error, so knowing statistically which kind of domains have assumed the errors w.r.t. the golden reference is essential. The classification we proposed is either based on specific

Algorithm 1: Extraction of the cardinality of a faulty tensor.

Input: faulty_tensor and golden_tensor.
Output: Cardinality of the faulty_tensor.

```
1 begin
2 |   differences_tensor = ||golden_tensor - faulty_tensor||;
3 |   equality_tensor = differences_tensor > ε;
4 |   return equality_tensor.sum()
5 end
```

values, such as NaN or zero values, and a range of values represents some domains. The domains are presented in the following list, each associated with an explanation of its origin:

1. *NaN*: the faulty value has become a NaN value, while the corresponding golden value is not. The NaN value may originate as a consequence of an illegal computation in computing units of the GPU.

2. *Zero*: the faulty value is zero, while the corresponding golden value is not. A zero value is likely to appear when a write instruction is not executed due to the bitflip, by changing the address or the register of destination for the example.

3. *Bitflip*: the faulty value differs by only one bit with respect to the golden value. This is the case when the bitflip injected by the GPU fault injector is presented as-is in the output of the operator. In this situation, the bitflip has not been absorbed by other computations and is likely to be injected just before a write operation close to the end of the kernel.

4. $[-1; 1] \setminus \{0\}$: the difference of the golden value and faulty value ranges between the closed interval -1 and 1, zero excluded because it will not be an error otherwise. This class models a fault that is presented in the significand bits of an IEEE 754 floating-point value. It is a qualitative class that is justified by the high regularity of the data in the CNN.

5. *Random*: this is the case when the faulty value does not lie in the previous domains. Thus, the error that will be generated is a random bit-string of the same size of the targeted data-type.

Algorithm 2: Domains classification of errors for a CNN's operator.

```
1  begin
2      nan_count = 0;
3      zero_count = 0;
4      bitflip_count = 0;
5      between_interval_count = 0;
6      random_count = 0;
7      foreach faulty_tensor ∈ faulty_tensors do
8          error_locations = find_errors(golden_tensor, faulty_tensor);
9          foreach error_location ∈ error_locations do
10             f_value = faulty_tensor[error_location];
11             g_value = golden_tensor[error_location];
12             if is_nan(f_value) and not is_nan(g_value) then
13                 nan_count++;
14             else if is_zero(f_value) and not is_zero(g_value) then
15                 zero_count++;
16             else if is_bitflip(f_value, g_value) then
17                 bitflip_count++;
18             else
19                 if -1 ≤ g_value - f_value ≤ 1 then
20                     between_interval_count++;
21                 else
22                     random_count++;
23                 end if
24             end if
25         end foreach
26     end foreach
27     return all the counters
28 end
```

Algorithm 2 presents the classification described above and is applied to all the faulty outputs of a CNN's operator. The order of the comparisons is not casual but enforces the correct classification of the bitflip and the closed interval $[-1; 1]$. Basically, it is tested if the faulty value belongs to one of the domains, and if so, the corresponding counter is increased. In the end, each counter is divided by the total number of faulty values analyzed, obtaining a probability map for each domain.

4.4.3 Spatial Patterns

The last parameter that is part of our error modeling methodology is the spatial pattern, which models the spatial locations of the indexes of the errors within the faulty tensors. The locations of errors are not random, but they follow patterns that are highly dependant on the computing device, the GPU, and the implementation of CNN's operators. In the previous two parameters, the cardinality and the domains of corrupted values, can be considered as independent from the shape of the analyzed tensor and can be reproduced to other shaped tensors without efforts. This is not valid for the spatial patterns because they are dependant on the shape of the analyzed tensor. Therefore, it is necessary to spend the effort to classify the spatial patterns according to a model that is reproducible and applicable also to tensors with different shapes from the observed ones. Another difference with the respect to the previous two parameters is that it is not possible to derive a classification for the spatial patterns in advance without having manually inspected the faulty tensors at first.

The indexes of the errors' locations within the tensor can be:

- *linear*: if we consider the tensor as a uni-dimensional vector and an index is the offset from the first element.
- *multi-dimensional*: if we consider the index as a triple belonging to the space $featuremaps \times height \times width$ of the tensor.

The choice of using one or another representation for the indexes is irrelevant, and we will use the one that better simply the discussion. The indexes of the errors in the first instance are absolute positions, but to be applied to other shaped tensors, we need to find another representation that enables the indexes to be freed from their absolute locations. Starting from the absolute locations of the errors, we define two additional vectors that will help the classification:

- *Offset Vector*: the indexes are sorted, and the first index is subtracted to all the others, obtaining a vector that is relative to the first index.
- *Stride Vector*: contains the strides between consecutive pairs of items of the offset vector.

Both representations will be used to represent different patterns.

To better clarify this methodology, let us introduce a practice example of classification, which helps to illustrate each step. Suppose of having experienced five errors, which locations expressed as linear indexes are

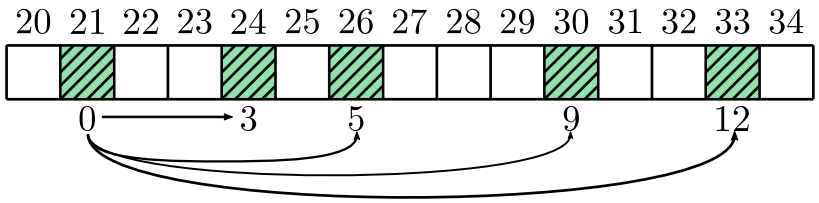


Figure 4.2: Offsets vector.

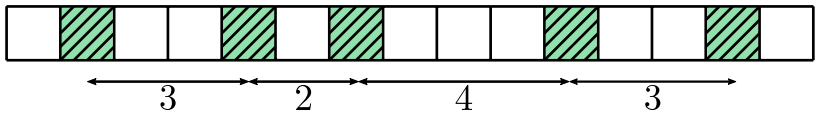


Figure 4.3: Strides vector.

$[21, 24, 26, 30, 33]$. The offsets vector is $[0, 3, 5, 9, 12]$, and the strides vector is $[3, 2, 4, 3]$, as shown in Figure 4.2 and Figure 4.3. Suppose of wanting to determine if that pattern is a row pattern, i.e., all the errors are located on the same row, and the width of the tensor's feature map is 4. That pattern is not a row pattern for the tensor under analysis because there are more errors than elements in a row. However, when classifying, it is important to generalize the pattern, and under the assumption of analyzing the smallest possible tensor, that pattern is a row pattern for a tensor of width 8. To catch this abstraction is possible to analyze the strides vector and for the row pattern discriminate if all the strides are less than the tensor's width. This condition holds even for the case of width 4, and under this logic, it will be classified as a row pattern.

4.5 Error Simulation

The error simulation is the evaluation of the reliability of the system by inserting the errors in the output of the CNN's operators through a tool called error simulator. The error simulation is entirely done at the application level, as shown in Figure 4.1, and uses an approach based on the *saboteur* that split into two sections the execution of the CNN to modify the output of a CNN operator and resuming the execution. Using the error models repository, build according to the previous methodology, the error simulation is not linked with the device level. Thus it is possible to test the system even without the physical GPU because the error models embed its behavior against faults. This feature is essential to properly evaluate the system already from the early stage of the development, even in sim-

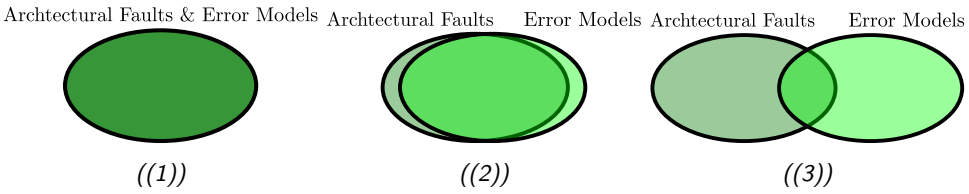


Figure 4.4: Intersection of faults set and errors set.

ulated environments that are not equipped with the full hardware, when it is tested for correctness. To ensure that the early assessment is possible, the error simulation should comply with the following implementation constraints:

- *Overheads*: the error simulator must not introduce excessive overheads, either related to time and memory. An excessive time overhead is not compatible with real-time systems with strict temporal deadlines, at the risk of testing a distorted system not more representative of the original one. Excessive memory usage by the error simulator may lead to the saturation of the available memory, risking compromising the nominal operativity of the systems.

- *Transparency*: the error simulator should be as transparent as possible, either respect to the user and the ML framework used to define the CNN. This constraint requires the error simulator to be easy to integrate with the system without demanding for the upheaval of it and not to change the behavior of the ML framework.

The error models used by the error simulator need to be trusted and validated, i.e., they need to be representative of the GPU behavior against faults and as accurate as of the observed faulty tensors. Otherwise, the whole error simulation is useless because we are testing the system with errors that have no foundation. In the following list, we will describe three situations that represent different possibilities of overlap between the set of architectural faults and the error models. The set of the architectural faults represents all the errors observed in the fault injections campaigns. On the other hand, the error models set represents all the errors that our modeling is capable of reproducing in the target CNN, through the error simulation. The intersection between these two sets identifies all the errors that have been experienced during the architectural fault injection campaigns and are reproducible by the error simulator. The left uncovered part, i.e., the portion of the architectural faults set that is not covered by the error models

set, represents all the errors observed during the fault injection campaigns that are not reproducible by the error models, due to its limited expression capabilities. The right uncovered part, i.e., the portion of the error models set that is not covered by the architectural faults set, represents all the errors that are generated by the error models but have not been observed in the architectural faults. This latter case is more dangerous than the previous one because we have modeled errors that have no foundation in reality, risk of compromising the reliability and robustness of our error models.

- *Figure 4.4(1)*: in this situation, the two sets completely overlap, representing the best achievable. This means that the error models have the same expressive power of the architectural fault injection.

- *Figure 4.4(2)*: in this situation, the two sets are overlapped for the majority but not completely in both sets. This means that the error models are able to catch the majority of the architectural faults, but not all. On the other side, the modeling introduces some artifacts, i.e., errors that have not been seen in the architectural fault campaigns, which are not be representative of the GPU behaviors. This situation is almost unavoidable because the modeling cannot be 100% accurate, but we should try to minimize the uncovered parts.

- *Figure 4.4(3)*: in this situation, the two sets are little overlapped. This represents the worst situation possible because the modeling is able to catch a minority of architectural faults and introduces many errors that are not linked to the GPU. This case should be avoided as much as possible, and a reliability analysis performed using this error model is almost useless because we are testing for a problem that is not reflected in reality.

In the next section, we will introduce the implementation of the framework here presented, specifying which tools have been used and which ones have been created in-house for this purpose.

4.6 Framework Implementation

We have designed and implemented a prototype of the proposed methodological framework by using both state-of-the-art tools and in-house ones.

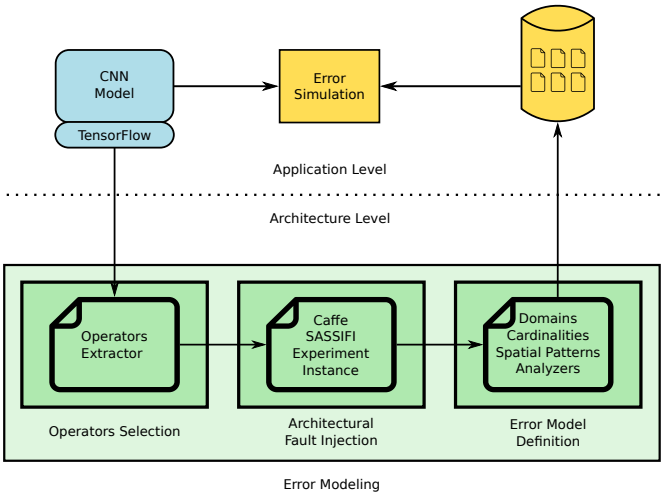


Figure 4.5: Instantiation of the methodological flow.

The structure of the designed software infrastructure is presented in Figure 4.5. The scheme mimics the one in Figure 4.1, in which each box has been expanded with the tools used or implemented to achieve that functionality. We have employed TensorFlow for modeling CNNs since it is one of the most popular frameworks for designing and training framework for this kind of ML applications. Therefore, we have implemented an in-house error simulator tool, which is integrated into the TensorFlow framework, to perform the application-level reliability analysis based on the repository of available error models. Indeed, as we will show later in Chapter 6, we discarded the publicly-available state-of-the-art tool, i.e., TensorFI [40], due to its limitations both in terms of setup effort and execution time required for the error injection campaigns. Finally, it is worth mentioning that, even if we selected TensorFlow, the proposed idea at the basis of the methodological framework and the error simulation is compliant with various other ML frameworks, such as Caffe or PyTorch. Future work is devoted to the porting of the error simulator in such frameworks. The lower part of the framework, which consists of the architectural fault injection on GPUs and the definition of the error models, has been designed around the state-of-the-art tool for NVIDIA devices that is SASSIFI [58]. Unfortunately, TensorFlow is not compliant with SASSIFI due to different requirements in terms of versions of the CUDA library. SASSIFI requires CUDA 7.0, while TensorFlow requires at least the subsequent CUDA 8.0. Therefore, the Caffe framework [6] has been employed as the replacement of Tensor-

Flow for the architectural fault injection and the next error modeling phase. Caffe is designed as bare to metal, directly exposing the GPU implementation of operators. The proposed approach is technically sound because there exists a one-to-one mapping between CNN's operators in TensorFlow and Caffe. Unless small implementation details, the operators' functionalities are preserved between these two ML frameworks, and it has been tested, for each operator, to obtain the same output under the same clean execution. The framework employs a custom Python script, named "Operators Extractor", to extract all the CNN's operators from TensorFlow, with which it is possible to select the relative Caffe operators by manual inspecting both ML frameworks. This mapping represents the connection between the application and the architectural levels. Besides extracting the operators' list, the "Operators Extractor" script can also retrieve an instance of the desired operator in terms of input, output, and parameter tensors. Once the operators are extracted, and the mapping between TensorFlow and Caffe is set, we create a C++ script, called "Experiment Instance" that embeds the GPU implementation of the operator's instance, exposed by the Caffe framework. The "Experiment Instance" represents the unit that will be executed numerous times by SASSIFI, as specified by the campaign size. The campaign sizing is a semi-supervised task because we rely on an automatic functionality offered by SASSIFI that counts all the assembly instructions executed by an operator, with which we manually choose the best size. After having performed all the fault injection campaigns with SASSIFI, the framework uses three Python scripts to build the error models repository, which scripts classify the faulty tensors. One script for each parameter of the error model, i.e., cardinalities, domains of corrupted values, and spatial patterns, that extract, classify and model the errors. The experimental setup and the execution of the architecture-level fault injection campaign for executing the error modeling methodology are provided in the next section, while the results and the analysis of the outcomes are discussed later.

Here in this chapter, we have presented the general methodology, which is behind the framework. All the details for what concerns the error modeling and the error simulator will be detailed in the next two chapters.

Chapter 5

Error Modeling

The methodological framework presented in the previous chapter is here implemented and demonstrated. In this chapter, we will discuss the application of the error modeling applied for the single operators of a Convolutional Neural Network (CNN), presenting a detailed implementation about each step of the methodological framework. Then, the discussion on the design and implementation of the error simulator is presented in the next chapter.

5.1 Operators Selection

The error modeling methodology has been applied only on a subset of all the possible CNN operators, which is a well-formed and representative subset, for the sake of demonstration and to validate the methodology; nonetheless, the experiments here performed can also be repeated for the remaining operators.

The CNN's operators that we consider for our experiments are taken from a TensorFlow implementation of the YOLO V3 CNN [43, 61, 62]. YOLO V3 is the case study used in the validation and evaluation part of this work. Table 5.1 lists the operator types extracted from YOLO V3, and the considered instances, in terms of input and output sizes. The instances have been chosen following the considerations made in Section 4.2. We privileged and chosen the smallest instance for each operator's type. This choice has been motivated by the fact that managing smaller data implies lower execution times, enabling us to perform the architectural fault injection in reasonable times. Smaller data translates into producing smaller objects to analyze. The whole error modeling methodology is easier

Table 5.1: Set of operators that are considered in this framework. The table shows the operator's type, the sizes of the input and the outputs, and some optionally note.

Operator	**Input**	**Output**	**Note**
Convolution 1	$512 \times 13 \times 13$	$256 \times 13 \times 13$	Kernel size = 1 and strides = 1
Convolution 2	$128 \times 52 \times 52$	$256 \times 52 \times 52$	Kernel size = 3 and strides = 1
Convolution 3	$256 \times 52 \times 52$	$512 \times 26 \times 26$	Kernel size = 3 and strides = 2
Add	$1024 \times 13 \times 13$	$1024 \times 13 \times 13$	
Batch Norm	$256 \times 13 \times 13$	$256 \times 13 \times 13$	
Biasadd	$256 \times 13 \times 13$	$256 \times 13 \times 13$	
Div	1×10647	1×10647	
Exp	$1 \times 8112 \times 2$	$1 \times 8112 \times 2$	
Leaky ReLU	$256 \times 26 \times 26$	$256 \times 26 \times 26$	Negative slope = 0.1
Mul	$1 \times 8112 \times 2$	$1 \times 8112 \times 2$	
Sigmoid	$1 \times 2028 \times 80$	$1 \times 2028 \times 80$	

to be applied to a small tensor and then generalized towards a bigger one than vice-versa. Because of these reasons and under the same conditions, there is no excuse to choose a tensor different from the smallest one. The "Operators Extractor" script retrieves the list of operators from TensorFlow from which we selected the smallest instance for each operator, and in Table 5.1 are reported all the selected instances for this framework.

It is worth noting that we choose three instances of the convolutional operator because they perform three slightly different operations, although they both implement the same functionality, as described in Section 2.1. Therefore, from the analysis point of view, their differences can lead to different error models for the three instances.

- Convolution 1 uses 1×1 kernels. A convolution operation with 1×1 kernels is no longer a neighbor operation because each element is multiplied by the kernel without involving any neighbor. The operation degenerates to an operation that scales each feature map by a factor. The implementation of such an operation is still carried out through General Matrix Multiplications (GEMM) but does not require any rearrangement of the data because no neighbors are involved as in

Table 5.2: Mapping of TensorFlow and Caffe operators.

Operator	**TensorFlow**	**Caffe**
Convolution 1	tf.nn.conv2d	conv_layer.hpp
Convolution 2	tf.nn.conv2d	conv_layer.hpp
Convolution 3	tf.nn.conv2d	conv_layer.hpp
Add	tf.add	caffe_gpu_add
Batch Norm	tf.nn.batch_normalization	batch_norm_layer.hpp scale_layer.hpp
Biasadd	tf.nn.bias_add	bias_layer.hpp
Div	tf.div	scale_layer.hpp
Exp	tf.math.exp	caffe_gpu_exp
Leaky ReLU	tf.nn.leaky_relu	rely_layer.hpp
Mul	tf.mul	scale_layer.hpp
Sigmoid	tf.math.sigmoid	sigmoid_layer.hpp

the other cases.

- Convolution 2 is a traditional convolution with a kernel of size 3. The implementation of this operation requires that each element and its neighbors are placed into column vectors before executing the GEMM. This displacement involves a custom Graphic Process Unit (GPU) kernel that creates the column vectors. So, the operation is composed of two kernels instead of one, as in the previous case.

- Convolution 3 is like the Convolution 2 but has the strides parameter set to 2. As mentioned in Section 2.1, a strides parameter set to 2 lets the operation also perform dimensionality reduction, halving the output space because not all the elements are considered in the computation.

From the list of operators, we manually build the one-to-one mapping between Caffe and TensorFlow, displayed in Table 5.2 since, as discussed, we are using two different Machine Learning (ML) frameworks in the top-most and bottom-most parts of the methodological framework.

For each operator's instance, it is necessary to define and extract the input, output, and parameter tensors from the TensorFlow model, using the "Operators Extractor" script. The output represents the golden reference,

and it has a double purpose. It is used first to align the Caffe implementation of the operator with TensorFlow to ensure and guarantee to obtain the same result. The golden reference will also be used by the "Experiment Instance" script to provide a first coarse classification during the fault injection campaigns. All those tensors are saved as NumPy archives (.npy) [63], which is the standard binary file format to store NumPy arrays on disk. It is worth mentioning that we extracted for each operator two different input/output datasets, while the parameters remain constant over the experiments. The choice of extracting only two different datasets is motivated by the fact that each tensor contains thousand of values, which are valid inputs for the GPU kernels. This as well as the regularity of data induced by normalization, which narrows the domain of the tensor's elements, and the extremely symmetric architecture of the GPU and the task performed, allows exploring the injection space extensively.

The "Experiment Instance" is the C++ script that allows us to execute the CNN operator's instance, following the guidelines provided in Section 4.2. The structure of each script is regular and described as follows:

- *Preamble*: during this step, the script loads the input, output, and parameter tensors required by the instance. The script uses the open-source Cnpy library [64] to load a NumPy archive into a C++ vector. After loading the tensors, it setup and allocates the memory required by Caffe.
- *Execution*: this step is different for each instance, and here it is created the Caffe layer with the parameters load in the previous phase. The layer is then executed on the GPU.
- *Epilogue*: the last step provides a first coarse classification of the output tensor. If the output tensor, computed by Caffe, differs w.r.t. the golden reference of at least one tensor's element, then such an output tensor is saved on disk as a NumPy archive, otherwise is discarded.

5.2 Architectural Fault Injection

We here discuss in detail how the tools in Figure 4.5 have been configured, and the architecture-level fault injection campaign has been performed.

As presented in Section 2.5.1.3, SASSIFI offers three modes of injection: Register File (RF), Instruction Output Value (IOV), and Instruction Out-

Table 5.3: Combinations of injection modes, instructions' classes and fault model used in the fault injection campaigns.

Injection Mode	Instructions' Classes	Fault Model
IOV	GPR, STORE_OP, PR_OP	Single Bitflip
IOA	GPR, STORE_OP	Single Bitflip
RF	GPR	Single Bitflip

put Address (IOA). Each CNN operator's instance has been tested with every mode. Therefore, we are able to target the register file, the output register of instructions, and the memory addresses.

Such a scope is sufficient enough to derive a comprehensive view of the underlying architecture and the behavior of each CNN's operator. SASSIFI provides quite fine control to choose the instructions to target for each injection mode, so it is possible to specify for each injection mode which class of instructions to target. The classes of instructions are described in the following list:

1. General Purpose Register (GPR): this class contains all the instructions that write to any register, such as integer, floating-point, move, and load instructions.

2. Store Operation (STORE_OP): this class contains all the store instructions, either global and shared.

3. Predicate Operation (PR_OP): this class contains all the instructions that write to the predicate registers, which are used for branching and control flow.

Table 5.3 presents the partition of injection modes, instructions' classes, and fault models, which are used for the campaigns.

As anticipated in Section 5.2, the campaign sizing is a semi-supervised task. SASSIFI provides a functionality that counts the assembly instructions contained in an executable program. Thus, using that function on each "Experiment Instance", we retrieve the instruction counts for each instance. That counters are manually inspected, and knowing how many memory manages each instance (input and output tensors), we derive the campaign size using the heuristic presented in Section 4.3. Table 5.4 and Table 5.5 show for each injection mode and instruction class the number of faults injected.

After having estimated the size of each campaign, SASSIFI provides another script that generates the fault list, given the injection mode, the instruction class, and the number of faults to insert (campaign size). Such a list is generated ahead-of-time, and then SASSIFI consumes it and the "Experiment Instance", performing the GPU fault injection campaign.

5.3 Error Model Definition

The outcomes produced by the fault injection campaigns have been subjected to three automatic scripts for error modeling implementing the approach presented in Section 4.4, and results are here discussed and commented.

5.3.1 Cardinalities

Cardinality is an index we defined to represent how many times the same error count has appeared in all the faulty tensor within an operator's experiment. This information defines in a probabilistic way how many values the error simulator has to corrupt in the output tensor to mimic the effects of faults corrupting the execution of an operator on GPU.

Bar charts in Figure 5.1 present for each considered operator the probability distributions for the various cardinalities extracted from the experiments' outcomes. From a coarse analysis of the cardinalities obtained is possible to highlight two clusters of cardinalities. The first set presents only low cardinalities, i.e., the maximum number of errors within the same tensor is 4. By contrary, the second set contains both low and high cardinalities, i.e., ranging from 1 error up to tens of errors within the same tensor. The operators that exhibit low cardinalities, which are Add, Biasadd, Div, Exp, Leaky ReLU, Mul, and Sigmoid, are implemented as "linear kernels". Linear kernels are straightforward GPU kernels, in which each thread processes one and only one element without involving any shared memory or cooperation between threads or blocks. The insertion of a fault in a thread of a "linear kernel" is rare to propagate to multiple elements, so it is reasonable to observe such cardinalities.

The operators that exhibit both low and high cardinalities are the three Convolutions and the Batch Norm. Such operators are composed of several kernels and heavily rely on GEMM, which benefits from the use of shared memory and threads cooperation. The corruption of one of those

threads can propagate to other threads and result in many errors in the output. However, the probabilities of high cardinalities are far lower than the low cardinalities. The number of instructions that have effects on other threads, i.e., the shared instructions, is lower than the number of instructions confined to the current thread. Such a condition is then reflected in their probability distributions.

Two other cardinalities pop up on the radar and are linked to the architecture of the GPU:

- *16*: it represents the block size used in Caffe, so the corruption of a block of threads leads to 16 errors in the output. Notice that also 15 is a cardinality tied to the block size because one location can be corrupted twice within the same thread block.
- *32*: it represents the size of the warp on NVIDIA GPUs. So, 32 errors correspond to the corruption of an entire warp. The same reason for 15 applies to 31 because the same location can be corrupted twice.

Table 5.4: Campaigns sizes for the IOV mode. Values are expressed in thousands.

Operator	**IOV**		
	GPR	**PR_OP**	**STORE_OP**
Convolution 1	20K	6K	2K
Convolution 2	10K	2K	4K
Convolution 3	20K	2K	2K
Add	4K	2K	1K
Batch Norm	30K	10K	4K
Biasadd	4K	2K	1K
Div	4K	2K	1K
Exp	4K	2K	1K
Leaky ReLU	4K	2K	1K
Mul	4K	2K	1K
Sigmoid	4K	2K	1K
Partial Sums	108K	34K	19K
Total	161K		

Table 5.5: Campaigns sizes for the IOV and RF mode. Values are expressed in thousands.

Operator	**IOA**		**RF**
	GPR	**STORE_OP**	**GPR**
Convolution 1	13K	2K	13K
Convolution 2	10K	2K	10K
Convolution 3	20K	2K	20K
Add	4K	1K	4K
Batch Norm	20K	4K	20K
Biasadd	4K	1K	4K
Div	4K	1K	4K
Exp	4K	1K	4K
Leaky ReLU	4K	1K	4K
Mul	4K	1K	4K
Sigmoid	4K	1K	4K
Partial Sums	91K	17K	91K
Total	108K		91K

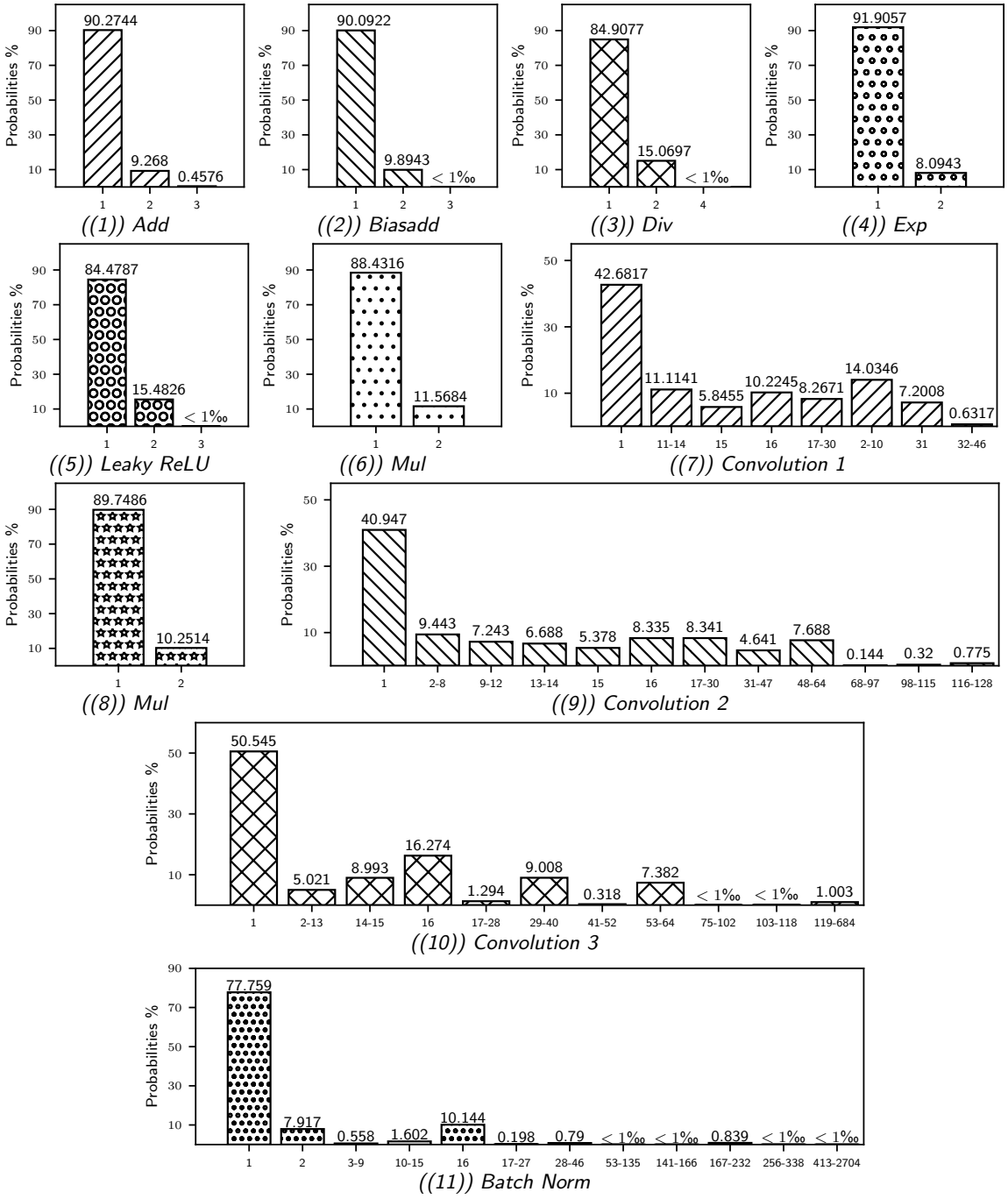


Figure 5.1: Cardinalities of all the CNN's operators. For the sake of brevity, some cardinalities have been grouped into ranges for a better visualization.

5.3.2 Domains of Corrupted Values

In a second step, we have analyzed how each single corrupted value varies from the golden counterpart, i.e., we have analyzed the domains of the corrupted values. Different from the cardinalities, it is not possible to link the results obtained to a specific architectural detail of the GPU; moreover such analysis only determined five coarse grained ranges of values in which each corrupted value falls into. Figure 5.2 shows the domains for each CNN's operator, showing only the percentage of occurrence of each case.

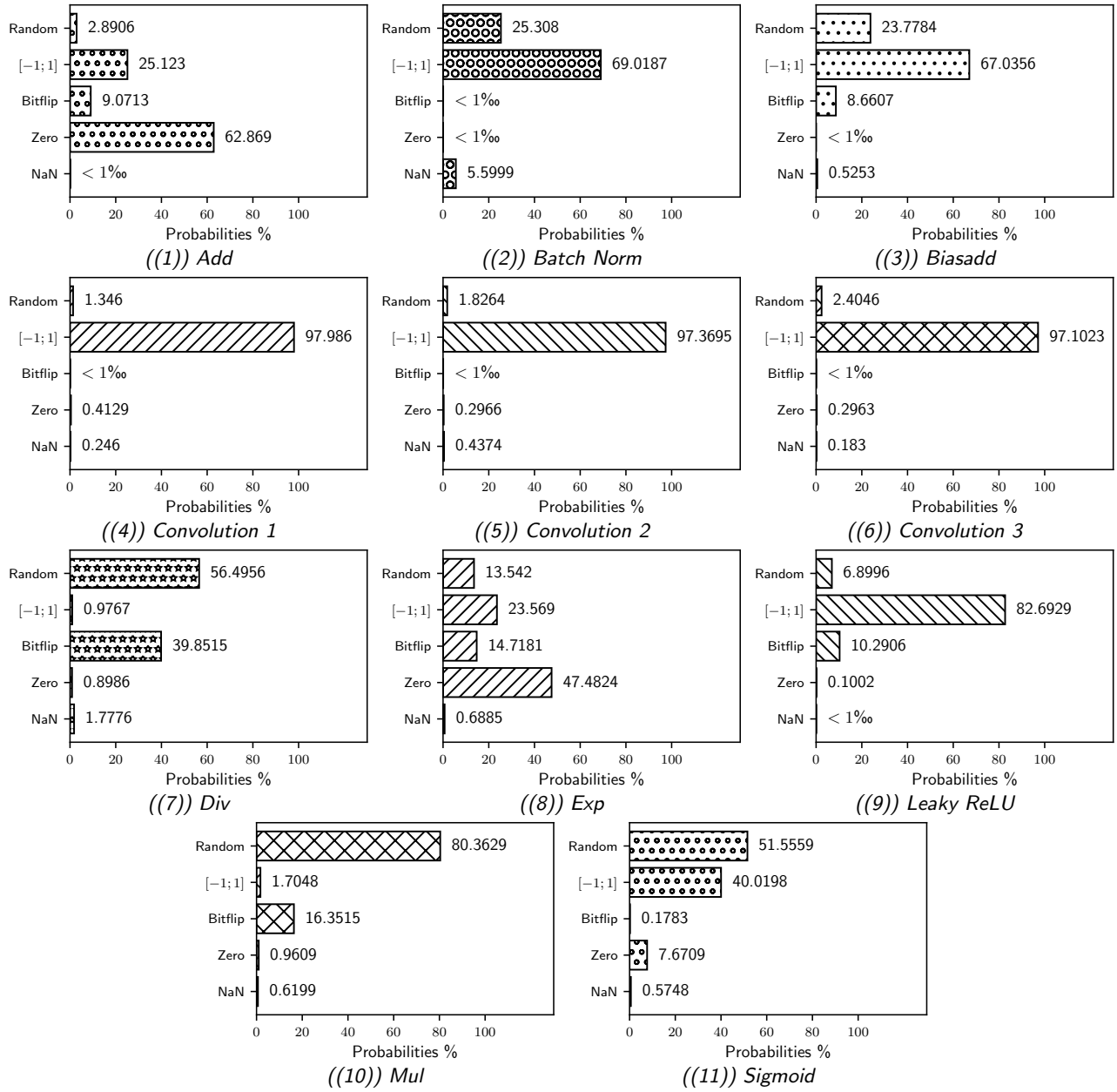


Figure 5.2: Domains of all the CNN's operators. Here are presented the results only in relative terms for the sake of brevity.

5.3.3 Spatial Patterns

Finally, we have analyzed the spatial patters of the errors, i.e., how the corrupted values are spatially distributed in the output tensor. The previous two parameters, the cardinalities and domains of corrupted values, do not require specific knowledge of the context for their formulations. Indeed, it has been possible to define the classes for both parameters in advance. This condition does not hold for the spatial patterns because the classes of locations of the corrupted values are explicitly dependant on the GPU architecture and the CNN. Thus, it is not possible to formulate in advance the classes for the spatial patterns, and their formulation has been a semi-supervised task.

Convolution 1 has been the first operator to be considered for this analysis. The faulty tensors of the Convolution 1 operator have been manually inspected to derive the spatial pattern classification. Then, the classification has been automatized and converted into an algorithmic procedure, which has been applied and validated to the faulty tensors of all the other operators. We can state that the classification derived by inspecting the faulty tensors of the Convolution 1 operator and then automatically applied to all the other tensors is sound and accurate because we are able to recognize the 98.83% of spatial patterns, leaving only the 1.17% of them unclassified.

The analysis of spatial patterns must be executed explicitly resorting to the knowledge of tensor, by classifying the patterns in the three-dimensional space instead of considering the linearized version. Such an abstraction degree allows classifying the patterns in the context of CNN, directly interpretable and manageable by the application level. The corrupted values are also located according to patterns that are linked to the GPU and its parallel architecture. The first coarse classification of the patterns concerns if the locations are placed on the same feature map or spread among several feature maps. In the next sections, we will detail each case presenting then a further fine classification.

5.3.3.1 Same Feature Map

This class of patterns considers only those locations that lie within the same feature map. This specific condition lets the problem be reconsidered and analyzed in terms of matrices instead of the full tensor because the tensor is just composed of several matrices stacked along a common axis. Therefore, the locations are classified according to the matrices notations, looking for

matching the locations of the corrupted values to specific anchors, such as rows and columns. From our experimental results, it has been noted that only faulty tensors with low cardinalities (≤ 16) present the errors that lie within the same feature map. Two factors motivate this observation. Despite the convolution and the batch normalization, all the other operators are implemented as linear kernels, which involves no cooperation among threads nor usage of the shared memory, so faults tend to propagate to one or two elements mostly. Complex operators like the convolution or batch normalization highly resort to GEMM, which is implemented using shared memory and threads cooperation to achieve high performances. The insertion of a fault within a code portion that manages shared data is likely to propagate to other threads belonging to the same block, according to the GPU programming model presented in Section 2.3. Hence, it is possible, according to the event presented above, that a set of elements in number near to the GPU block size result corrupted, lying within the same feature map.

Spatial patterns in the same feature map are presented in the following list:

- *Single Point*: this is the simplest case, in which only one value is corrupted, as shown in Figure 5.3. It has been proved and checked that there no exist a location for the single point that has a probability of appearing higher than the uniform probability. For this reason, this class will generate one corrupted value in a random location within the tensor.

- *Same Row*: the locations are placed on the same row of the feature map, as shown in Figure 5.4(1). Despite the trivial case, this pattern applies also for the cases shown in Figure 5.4(2) and Figure 5.4(3). In the first case the row is not complete, whereas the second case presents a scatter pattern but in both cases the errors lies within the same row. For the last case, shown in Figure 5.4(4), apparently seems that the locations are not on the same row, which is true for the example in figure. However, that pattern has to be generalized to any possible shaped tensor, so for a possible larger tensor that pattern is on the same row. This pattern happens when all the strides within the strides vector are strictly less then the width of the tensor.

- *Same Block*: the locations are spaced in multiples of the GPU block size.

- *Unclassified*: it is not possible to classify the locations according to the previous three classes, so they are assumed to be random within the feature map.

The "Same Block" and "Unclassified" classes are not shown in figures because it is not possible to represent them in a meaningful way.

5.3.3.2 Multiple Feature Maps

In contrast to the previous class, the locations are spread across multiple feature maps. This class applies only to "complex" operators, such as the three convolutions and the batch normalization, and only to those faulty tensors that present high cardinalities (≥ 16). The intuitive explanation of these locations is always related to the shared memory and threads cooperation but also to the control of the GPU. If a fault targets a warp or its execution, then it is likely to observe a high number of errors in the output. Despite the previous case, it is necessary to take full advantage of the notion of the tensor.

Spatial patterns in multiple feature map are presented in the following list:

- *Bullet Wake*: the same location within a feature map is corrupted in multiple feature maps, as shown in Figure 5.5(1) and 5.5(2). To detect this behavior, it is checked if the locations are spaced in multiples of the feature map size, i.e., H $\times$ width. The feature maps involved can be sequential, or they can have any scatter pattern within a starting and ending feature map.

- *Same Block*: similar to the case for the same feature maps, this case is when locations are spaced in multiples of the GPU block size and they are spread across multiple feature maps.

- *Shatter Glass*: this class is an evolution of the bullet wake in which there is a common shared location among all the feature maps,but then in one or more feature maps, the errors can spread over the rows or columns following the patterns presented for the same feature map case classification, including the common location. The examples of such a class are shown in Figure 5.6(1), 5.6(2), and 5.6(3).

- *Quasi-Shatter Glass*: this class is a relaxation of the previous one in which some times the shared location cannot be present in all the

feature maps, but the row or column pattern is present, as shown in Figure 5.7(1), 5.7(2), and 5.7(3).

- *Unclassified*: it is not possible to classify the errors' locations according to the previous three classes, so they are assumed to be random within the whole tensor.

The "Same Block" and "Unclassified" classes are not shown in figures because it is not possible to represent them in a meaningful way.

The probabilities of occurrence of the various spatial patterns for each considered CNN operator are reported in Table 5.6. It is worth mentioning that the type and the probability of occurrence of the various spatial patterns are highly dependent on the algorithm of the analyzed operator. For this reason for linear operators, such as Add or Biasadd, the single point failure has a very high probability of occurrence; on the opposite, for the more complex Convolution 1 operator, other spatial patterns are more frequent. It is not excluded that for other operators we have not analyzed here, different new patterns can be observed; for instance, in case of operators having a column-based organization of the data matrices, a *Same Column* spatial pattern is observed in place of the *Same Row* one. Indeed, as a final note, such a pattern has already been noticed for the considered operators, but with such a low frequency (close to zero) that we have decided not to report it as a new pattern but in the *Unclassified* one.

5.3.3.3 Generality and Parametrization

The outcome of the error modeling phase is a set of probability distributions, one per each operator and parameter. The first two parameters, the cardinalities and the domains of corrupted values, are fixed, which means the model is the same for all the operators, to which are assigned different probabilities according to the operator type. These parameters already provide a sufficient degree of generality and completeness. The spatial patterns have a more complex model and representation than the other two parameters. The spatial patterns are different for each operator but also for each cardinality within the same operator. This happens because the spatial patterns and cardinalities are related to each other. Thus, the structure of the spatial pattern model is indexed by cardinalities, and, for each cardinality, we have a different probability distribution following the classification provided at the beginning of this section. The probability distribution is not sufficient to represent a spatial pattern because we also need to store the pattern itself in terms of tensor's indexes or locations

Table 5.6: Distribution of the various spatial patterns on each considered operator.

	Same Feature Map				Multiple Feature Maps				
Operator	**Single Point**	**Same Row**	**Same Block**	**Unclas.**	**Bullet Wake**	**Same Block**	**Shatter Glass**	**Quasi Shatter Glass**	**Unclas.**
Add	0.903	0.018	0.0	0.008	0.0	0.059	0.0	0.0	0.012
Batch Norm	0.778	0.025	0.01	0.001	0.127	0.028	0.011	0.0	0.02
Biasadd	0.901	0.012	0.0	0.01	0.002	0.037	0.0	0.0	0.038
Convolution 1	0.376	0.217	0.0	0.0	0.226	0.0	0.174	0.004	0.003
Convolution 2	0.409	0.131	0.001	0.0	0.329	0.012	0.103	0.012	0.003
Convolution 3	0.505	0.124	0.001	0.0	0.254	0.0	0.11	0.005	0.001
Div	0.849	0.067	0.084	0.0	0.0	0.0	0.0	0.0	0.0
Exp	0.919	0.005	0.0	0.0	0.0	0.043	0.0	0.0	0.033
Leaky ReLU	0.845	0.015	0.008	0.03	0.0	0.048	0.0	0.0	0.054
Mul	0.884	0.003	0.0	0.0	0.0	0.04	0.0	0.0	0.073
Sigmoid	0.897	0.013	0.0	0.0	0.0	0.05	0.0	0.0	0.04

Table 5.7: Table of parameters for each spatial pattern class.

Class	Unit	Parameters
Same Row	offset vector	maximum distance in linear index
Same Column	vector of relative column indexes	maximum relative column index
Bullet Wake	vector of relative feature map indexes	maximum of the relative feature map indexes
Shatter Glass	list of pairs feature map indexes and linear indexes w.r.t. the shared element within each feature map	# of different feature maps involved, maximum of the relative feature maps indexes, minimum and maximum linear indexes within the feature map w.r.t. the shared element
Quasi Shatter Glass	the same as Shatter Glass	the same as Shatter Glass
Same Block	vector of relative indexes of the GPU block size	maximum relative index of the GPU block size.
Random	offset vector	–

to be reproducible. Hence, alongside the probability distribution, we store the pattern using the parametrization provided in Table 5.7. Therefore, a spatial pattern is identified either by a probability distribution and a numerical representation that contains the information referred to as the assigned pattern class.

5.4 Definition of the Error Models

The analysis of the fault injection campaigns, according to the three parameters, has resulted in a repository of probability distributions, one per each parameter and operator. Such a repository represents our error model with which we define an error algorithmically. Given the output tensor, the operator's type that has originated it, and the error model, we can generate an error for that tensor using a saboteur approach. The generation of an error follows the following steps in order:

1. Draw of the cardinality from the cardinalities probability distribution

of the given operator.

2. Draw of the corrupted values, as many as the drawn cardinality, from the distribution of the domains of the given operator.

3. Draw and generation of the spatial pattern according to the drawn cardinality and the given operator.

Following these three steps, we can corrupt the output of any CNN operator through an algorithm and, therefore, being able to inject errors.

In this chapter, we have first presented the overall structure of the proposed methodological framework for the error simulation in CNNs executed on GPU devices. Then, we have focused the discussion on the execution of the error modeling methodology. The result is a repository of error models for the various operators of a CNN. In the next chapter, we will continue the discussion on the implementation of the methodological framework presenting the designed and prototyped error simulator where the defined error models repository has been integrated into.

An analytic view of all the results presented in this chapter, regarding the error models, is available in the GitHub repository of this thesis [65]. There you can find either absolute and probabilistic results for each operator and for each model's parameter, which are not particularly amenable to be displayed in their entirety.

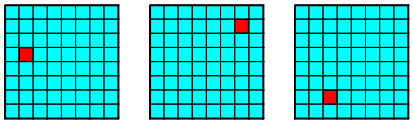
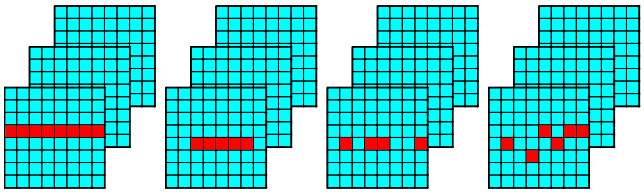
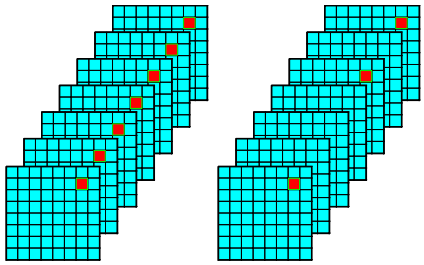


Figure 5.3: Spatial Patterns - Same Feature Map - Single Point



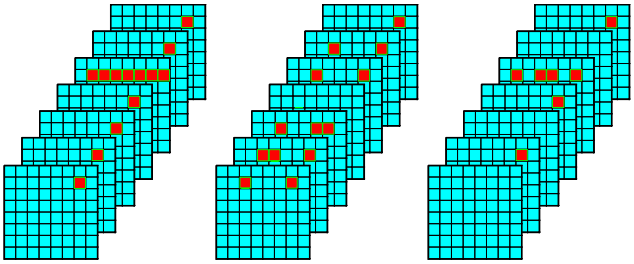
((1)) *((2))* *((3))* *((4))*

Figure 5.4: Spatial Patterns - Same Feature Map - Same Row



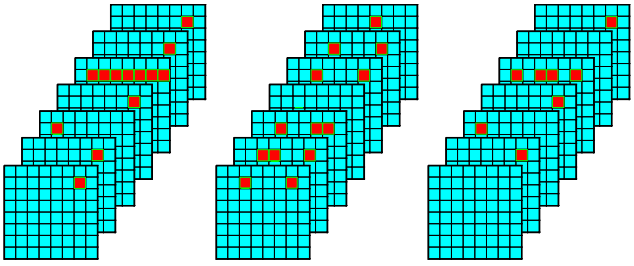
((1)) *((2))*

Figure 5.5: Spatial Patterns - Multiple Feature Maps - Bullet Wake



((1)) *((2))* *((3))*

Figure 5.6: Spatial Patterns - Multiple Feature Maps - Shatter Glass



((1)) *((2))* *((3))*

Figure 5.7: Spatial Patterns - Multiple Feature Maps - Quasi-Shatter Glass

Chapter 6

Error Simulation

In the previous chapter, we presented the error modeling and the instantiation of the methodological framework neglecting the error simulation, which is here designed and implemented. The first section presents the overall structure of the error simulator, whose phases will be individually detailed in the next sections. Finally, we compare our error simulator tool to the literature, highlighting the strengths and weaknesses.

6.1 Overall Structure

The error simulator is a tool developed in Python that allows us to inject errors, according to the models extracted in the previous chapter, in the output of any Convolutional Neural Network (CNN)'s operator, within the TensorFlow framework. This tool falls in the category of functional error simulators because it works at the application level, ignoring the underlying architecture, in which the CNN's operators are executed. Nevertheless, the workflow of this simulator does not differ from the traditional organization

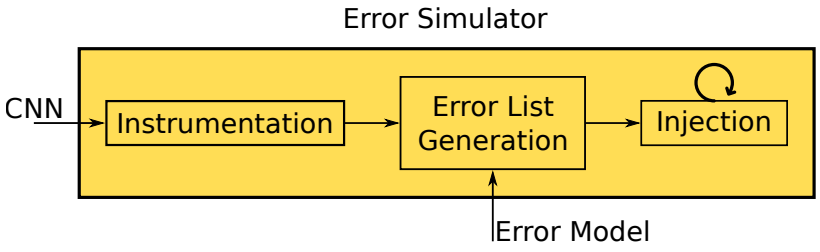


Figure 6.1: The phases of which the error simulator is composed of.

of an architecture-level fault injector, as discussed presented in Section 2.5.1 for the Graphic Process Units (GPUs). The error simulator is composed of three main phases, as shown in Figure 6.1, introduced in the following list:

- *Instrumentation*: given the CNN model, expressed as the TensorFlow data-flow graph, the error simulator replicates it and inserts the instrumentation logic that is needed to extract all the operators that are inject-able.
  - **Input**: the CNN data-flow graph and TensorFlow's session.
  - **Output**: the list of inject-able operators and replicated data-flow graph.
- *Error List Generation*: knowing the inject-able operators and the campaign size, the error simulator generates the list of errors, which form the campaign.
  - **Input**: the list of inject-able operators, the error model, the campaign size, and the injection policy.
  - **Output**: the error list.
- *Injection*: the error simulator consumes the error list. This phase can be reiterated many times because it is possible to inject several inputs using the same error list.
  - **Input**: the error list, the CNN input.
  - **Output**: the list of output tensors after the campaign execution.

6.2 Instrumentation Phase

The core of the instrumentation phase is to extract the inject-able sites, i.e., the graph's operators, in which it is possible to inject the errors. This whole phase is automatic and does not require any user intervention, easing the complexity, especially in the case of large networks. The extraction of the inject-able sites needs to take into account the following aspects concerning the TensorFlow data-flow graph:

- The data-flow graph is append-only, which means we can append operators to the graph, but we cannot modify the already existing relationships among the operators, i.e., the edges, nor remove them. Therefore, any modifications of the actual data-flow graph require the building of a replica of it.

- The data-flow graph may contain operators that are not relevant to the inference phase, such as training optimizer or save/restore operations. Therefore, the inject-able operators are the ones who are activated and triggered during the inference phase, discarding the ones who are not executed during this phase.

The extraction of the inject-able operators, according to the definition above, cannot be done using the original data-flow graph because the public Application Program Interfaces (APIs) of TensorFlow do not allow us to retrieve the execution trace. Therefore, it is necessary to insert within the data-flow graph the instrumentation code to be able to determine if a certain operator has been executed or not. The instrumentation code is a Python function, which is inserted before each operator, that, if executed, it will append to a global list the signature of the operator, which contains the identifier and the shape of the output of the operator. The insertion of such an instrumentation code directly in the original data-flow graph is not possible because of three reasons:

1. The data-flow graph is not modifiable, if not building a new one.
2. Even if it was possible to modify the original data-flow graph, we want to design our error simulator to be as transparent as possible to the user, leaving the data and objects provided by the user untouched.
3. The execution of the instrumentation code is within the Python environment, while the TensorFlow's operators are executed from a C++ library, which represents a different application environment. Since the instrumentation code is inserted before each operator, this continuous shift of environment induces a time overhead because the context is switched from Python to C++ back and forth. This time overhead induced by a single execution of the instrumented data-flow graph is negligible because the extraction is performed only once while using the instrumented graph also for the injection would lead to an unsustainable and avoidable time overhead.

Thus, the insertion of the instrumentation code requires to create a new data-flow graph, which is a replica of the original one. Figure 6.2 shows an example of data-flow graph replication.

The steps performed during the instrumentation phase are shown in the form of pseudo-code in Algorithm 3, and are explained in the following list:

- The inputs of the instrumentation phase are the original data-flow

Algorithm 3: Pseudo-code of the instrumentation phase.

```
   Input: old_graph, fetches, and input_feed_dict
   Output: List of activated operators
 1 begin
 2     activated_operators = [];
 3     new_graph = tf.Graph();
 4     foreach op ∈ old_graph.operators() do
 5         new_op = create_op_replica(op);
 6         new_graph.append_with_control_dependency(new_op, instr_func);
 7     end foreach
 8     assign_variables(old_graph, new_graph);
 9     session.run(fetches, feed_dict=input_feed_dict);
10     return activated_operators;
11 end
```

graph ("old_graph"), the output of the CNN model ("fetches"), and the input parameters of the CNN model ("input_feed_dict").

- Line 2 and Line 3: we create the new TensorFlow's data-flow graph and the list that will contain the operator's signatures.

- Line 5: we replicate each operator of the original data-flow graph using only the public APIs of TensorFlow.

- Line 6: the replicated operator is appended to the new data-flow graph, having set the instrumentation function as control dependency. As detailed in Section 2.2.1, the control dependency is a mechanism that forces the execution of the operation before the current one. In our case, we force the execution of the instrumentation function before the current operator, so when the current operator is executed necessarily, the instrumentation has already been executed. The instrumentation function inserts in the activation list the signature of the operator, which is the operator's name and the shape of the output tensor.

- Line 8: the replication of an operator does not involve the transferring of its parameters. To do so, we have to copy the parameters one at a time using the TensorFlow's assignment function to transfer them from one graph to the other.

- Line 9: we perform an inference run with the new data-flow graph, using the session object of TensorFlow.
- Line 10: the list contains all the activated operators, which have executed during the inference run.

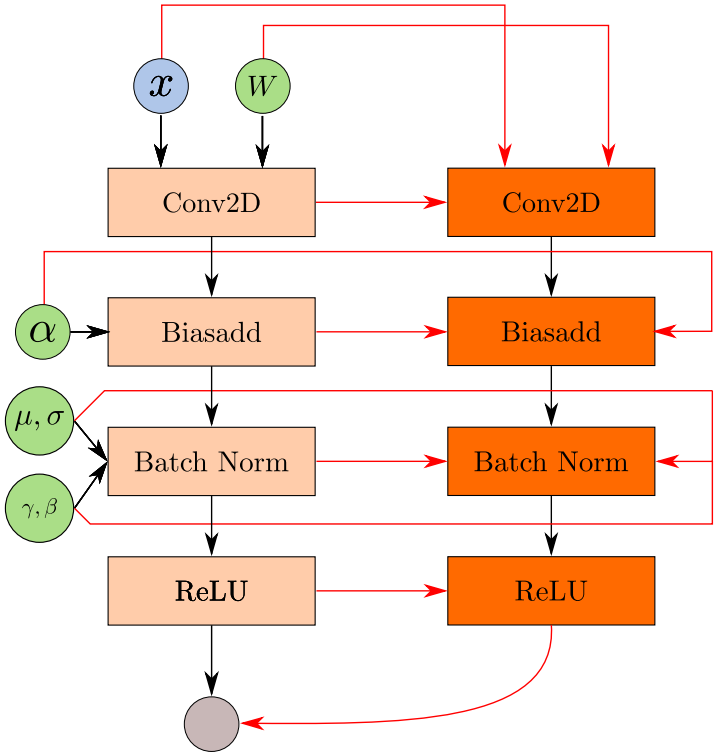


Figure 6.2: Example of replication of the data-flow graph. The left graph is the original one, while the right graph is the replicated one. All the parameters are shared between the two graphs, as well as the input tensors.

6.3 Error List Generation

The error simulation campaign can be performed on all the inject-able operators or a subset of them. The user can specify the focus of the campaign by indicating with which injection policy the error simulator has to generate the error list. The injection policies are:

- *Random*: no restriction is applied; all the operators are considered.
- *Operator Type*: this policy focuses only on a specific operator type.

- *Operator Specific*: this policy focuses on a specific instance of an operator.

The policies are applied by filtering the list activated operators and removing all those who do not meet the condition specified by the user.

The error model repository is presented as a set of JSON files, one for each operator type and for each parameter, which the error simulator loads during this phase. The cardinalities and domains of corrupted values are stored as lookup tables, in which, for each operator type, we have the domain or the cardinality associated with its probability, as shown in Figure 6.3. The spatial patterns are stored as a double-entry lookup table, as shown in Figure 6.3(c), in which, for each operator and cardinality, we have a first probability distribution that identifies the pattern class and then, for each class, the patterns themselves, each associated with their intra-class probability.

As anticipated in the previous chapter in Section 5.4, the error model can be algorithmically described. Therefore, the error list generation is outlined as a procedure, shown in the form of pseudo-code in Algorithm 4, and explained in the following list:

- Line 3: the inject-able operators are filtered according to the user-defined injection policy.
- Line 4: we generate as many errors as the user-defined campaign size.
- Line 5: we draw a random operator from the filtered ones. The selection is performed with replacement because the same operator can be injected more than once.
- Line 6: we draw a cardinality using the probabilities associated with the selected operator type.
- Line 7: we draw as many domains as the cardinality using the probabilities associated with the selected operator type.
- Line 8: we draw the spatial pattern for the selected operator and cardinality.
- Line 9: we append to the error list all the extracted objects.
- Line 11: the error list is generated and returned.

```
''Convolution 1'' :
{
    1 : 0.972,
    4 : 0.02,
    8 : 0.008,
    ⋮
},
⋮
}
```

(a)

```
''Leaky ReLU'' :
{
    NaN : 0.005,
    Zero : 0.01,
    Bitflip : 0.1,
    [−1;1] : 0.85,
    Random : 0.035
},
⋮
```

(b)

```
''Biasadd'' :
{
  2: {
      ''Class'' : {
          ''SAME_FEATURE_MAP_SAME_ROW'': 0.32,
          ''SAME_FEATURE_MAP_SAME_COLUMN'': 0.17,
          ⋮
      }
      ''Pattern'' : {
          ''SAME_FEATURE_MAP_SAME_ROW'' : {
              (0, 2) : 0.57,
              (0, 4) : 0.33,
              RANDOM(64) : 0.1
          }
      }
  }
  ⋮
}
```

(c)

Figure 6.3: Box (a) represents the structure of the cardinality, Box (b) represents the structure of the domains of corrupted values, and Box (c) represents the structure of the spatial patterns.

Algorithm 4: Pseudo-code of the error list generation phase.

```
   Input: operators, injection_policy, campaign_size, error_models
   Output: Error list
 1 begin
 2     error_list = [];
 3     filtered_op = filter_operators(operators, injection_policy);
 4     for i ← 0 to campaign_size do
 5         op = select_operator(filtered_op);
 6         cardinality = select_cardinality(error_models, op);
 7         domains = select_domains(error_models, cardinality, op);
 8         patterns = select_patterns(error_models, cardinality, op);
 9         error_list.append(op, cardinality, domains, patterns);
10     end for
11     return error_list;
12 end
```

6.4 Injection Phase

The error simulation exploits a technique based on saboteurs, to actually inject the corrupted values, and execution check-pointing to speedup executions by "jumping" directly to the operator to be corrupted. The check-pointing splits the inference execution of the CNN into two parts, as shown in Figure 6.4. In the first part, the CNN is executed up to the operator we want to inject, retrieving its output, using the CNN input provided by the user. Then, such an output is modified according to the domains and the pattern extracted for the considered error. The second part reintroduces the modified output in the network, and the CNN is executed up to the end. This approach is described for the injection of one operator, and it is simply repeated for each error in the list when performing an error campaign. Whenever the same operator is targeted more than once, then its output is cached, so we can avoid to perform the first part of execution and to reuse the already extracted output.

Once again, this phase can be described in the form of an algorithm, as shown in the pseudo-code in Algorithm 5, and the following list details each step:

- Line 2: this list will contain all the outputs produced during the campaign execution.
- Line 3: the cache is a dictionary that contains the operator as key

Algorithm 5: Pseudo-code of the error injection phase.

```
   Input: error_list, fetches, input_feed_dict
   Output: Output list
 1 begin
 2     output_list = [];
 3     cache = {};
 4     foreach (op, cardinality, domains, pattern) ∈ error_list do
 5         if op ∉ cache then
 6             output = session.run(op, feed_dict=input_feed_dict);
 7             cache[op] = output;
 8         output = cache[op];
 9         modify_output(output, cardinality, domains, pattern);
10         net_output = session.run(fetches, feed_dict={op: output});
11         output_list.append(net_output);
12     end foreach
13     return output_list
14 end
```

and its output as value.

- Line 4: we iterate over the error list, extracting, for each error, the operator, the cardinality, the domains, and the spatial pattern.

- Line 5: we check if the current operator is in the cache or not.

- Line 6, 7: if the current operator is not in cache, then we compute its output and store it on the cache for further reuse.

- Line 8, 9: we extract a copy of the output from the cache, and we modify it in-place according to the cardinality, domains, and spatial pattern.

- Line 10, 11: we complete the second part of the execution, providing the modified output, and then we append the model's output in the list.

- Line 13: we return to the user the list of outputs.

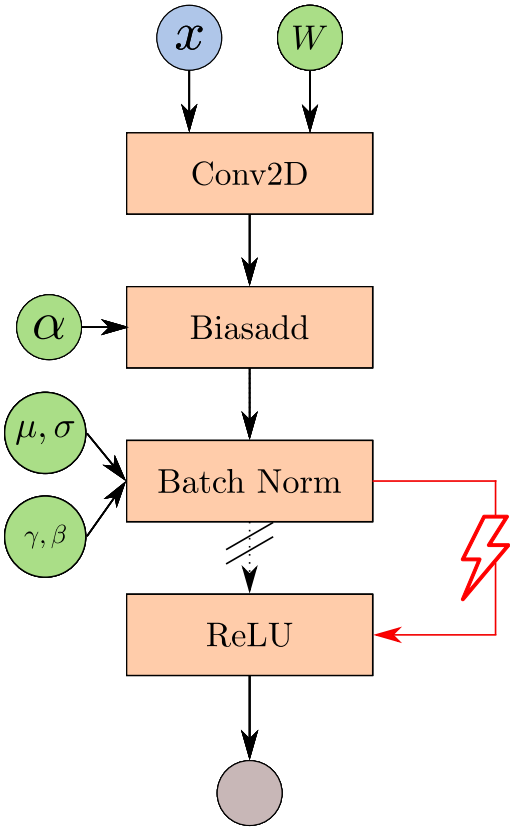


Figure 6.4: Example of injection using the check-pointing technique. The model is evaluated until the Batch Norm operator, then its output is returned to the application domain, in which the errors are injected. After that step, the modified output is brought back within the network, resuming the execution.

6.5 Methodological and Implementation Flaws

During the development and design of our error simulator, we have noticed some differences between our approach and our direct competitor, TensorFI. Such differences regard either methodological and implementation details, which have been taken into account during the development, which are not found in our tool. The next two sub-sections describe in detail each difference.

6.5.1 Error Models

The biggest methodological flaw presents in TensorFI is related to the error models it embeds and their validation. TensorFI does not provide any justification nor validation of its error models, which are directly inherited from the architectural fault injection. The first difference between the error models of the two tools is that our error models are probabilistic, so we are able to inject at least one corrupted value up to tens of them, while TensorFI can corrupt one element or the whole tensor. In our analysis of more than 100,000 faulty tensors, we have never experienced the case in which the whole tensor has been corrupted. Thus, the only common point between the two models is considering only the corruption of one value, and in that case, it does not make sense to discuss its spatial location because it is random within the tensor. Therefore, the only comparison between the two models regards one corrupted value and its domain, and Table 6.1 presents the domains offered by TensorFI and our framework. Figure 6.5 shows how the domains overlap, highlighting that TensorFI provides a small set of errors compared to our simulator and the architectural fault injection. TensorFI does not consider at all more than one corrupted value, losing all the patterns typical of the GPU and also introduces errors that are not validated. Therefore, the error models embedded in TensorFI are not significant, and one should avoid relying on them for the reliability analysis of a system because they do not represent the GPU behavior against faults and introduce errors that are not related to the hardware or other behavior.

6.5.2 Minor Differences and Setup Effort

TensorFI does not replicate the data-flow graph one-to-one, but it copies each operator trying to guess also its parameters but fails to achieve this task because some parameters are hard-coded or unsupported. This is the case of the convolution operator or other operators that do no support

Table 6.1: Comparison of the domains of corrupted values offered by TensorFI and our error simulator.

Domain	TensorFI	Our Approach
NaN	Not present	Present
Zero	Present	Present
Bitflip	Present	Present
$[-1; 1]$	Achievable but not by default (requires code modification)	Present
Random	Present, but restricted to the range $[0, 1)$	Present, can generate any 32-bit floating-point value.

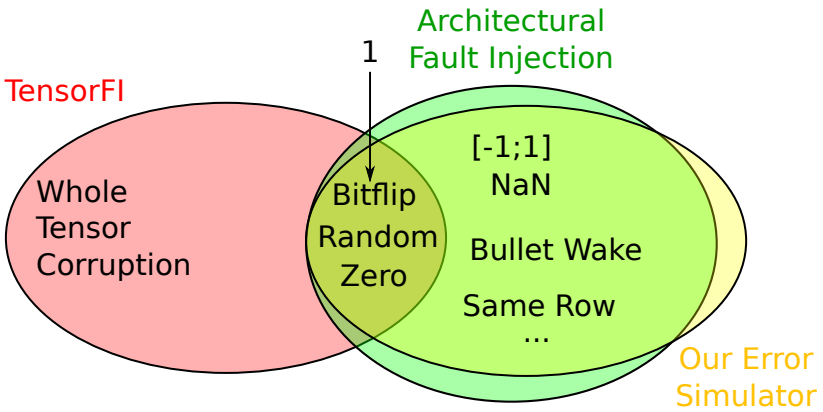


Figure 6.5: Overlap of the domains of TensorFI, our error simulator, and the architectural fault injection.

the typical representation of tensor used in Machine Learning (ML), i.e., $C \times H \times W$, but support only $H \times W \times C$, making difficult to support real models, since the majority of them are expressed using the former form. Such an approach can lead to incomplete representations or dark errors related to the guessing of parameters. The technique used to replicate the graph by our implementation relies on the public APIs of TensorFlow. It provides an identical copy of the data-flow graph, so it implicitly supports and copies all the parameters and tensor's formats. The approach adopted by TensorFI has also reflected in the number of supported operators, which is relegated to only those who are covered in the source codes. On the contrary, our replication is total and covers any possible operator with any possible configuration because our replication is deep and accurate.

TensorFI has a partially-automated instrumentation phase that replicates the original data-flow graph but relies on a configuration file, filled by

the user, to generate the injection sites at run-time. The configuration file contains a section in which the user must express the number of instances of each operator present in the data-flow graph. However, as pointed out in Section 2.5.1.5, the extraction of the number of instances is not trivial because only the operators that are target-able by TensorFI need to be counted and only the instances that are used by the inference phase. This approach fails to scale for a network with tens or hundreds of operators. The instrumentation phase provided by our tool is completely automated and does not require any user interaction. The difference is quantifiable in at most a pair of minutes for our automatic instrumentation, while it can take up to hours to configure TensorFI properly.

6.6 Porting to Other ML Frameworks

This error simulation approach has been proved to work also for other ML frameworks, such as Caffe and PyTorch. We have built two proof of concept experiments that, using the same methodology of this error simulator, achieve the goal of injecting errors in a CNN, meeting all the requirements presented so far. Summing up, the check-pointing method is portable to other ML frameworks, but every implementation requires specific adaptation of the instrumentation phase to adhere to the framework under analysis. Caffe and PyTorch do not rely on data-flow graph computation. Instead, they make use of the eager execution, an imperative environment that evaluates operation immediately without decoupling the definition and the execution like in TensorFlow. The flexibility of ML frameworks has allowed to develop the tool in just over 1000 lines of code, 584 for the error simulator and 667 for the model and injection sites generation.

We have here presented the error simulator and the methodology behind it. In the next section, we compare our error simulator against a GPU fault injector to assess the correctness of the simulator in a real case scenario.

Chapter 7

Experimental Evaluation

This chapter presents an experimental evaluation of the proposed methodological framework against the baselines and using the Key Performance Indicators (KPIs), both identified in Chapter 3. In the first section of the chapter, we present the case studies, which are three Convolutional Neural Networks (CNNs) and their datasets, which will be used as the targets of the experiments. In the second section, we will provide the accuracy validation of our approach against the current practice, which is the architectural fault injection, considering the state-of-the-art Graphic Process Unit (GPU) fault injector, SASSIFI [58]. In the third section, we will provide comparisons of the execution times of our framework with respect to SASSIFI and TensorFI [40]. We will also highlight some additional notes about TensorFI ans its flaws. The final section sums up all the obtained results, providing a comprehensive view of them.

All the experiments and comparisons have been performed on the same machine, which is a MacBook Pro 2014 equipped with an Intel Core® i7-4870HQ as CPU and a NVIDIA GeForce GT 750M as GPU, running on Ubuntu 18.04 LTS.

7.1 Case Studies

We have used three different CNN models and three different datasets for the evaluation of our framework. The following list presents the three different CNN models, while, after that, we will motivate the reasons for using three different models.

- **YOLO V3 [43, 61, 62]**. It represents the state-of-the-art in the object detection task enough to be employed in commercial autonomous

driving systems like Apollo [66] and Autoware [67]. We have considered one implementation for TensorFlow trained upon the COCO dataset [68]. The network falls in the category of deep neural networks because it contains 45 different operator types and more than 6000 operator's instances. The COCO dataset with which is trained contains more than 330 thousand RGB images and 80 object categories. In the accuracy validation, we will compare the output of this network to check if the error simulation is capable of producing the same effects as the architectural fault injection. The output of the network is a list of detection for each of the 80 object categories. Detection is a tuple of five elements (x_1, y_1, x_2, y_2, p), in which the first four elements are the pixel coordinates of the bounding box, which marks the object within the image, and the last element is the probability assigned to this object.

- **LeNet-5 [7]**. It is a CNN used for hand-written digits classification for the MNIST dataset [69], capable of achieving 99.05% accuracy. It is a simple network composed of two convolutional layers and three dense layers. The MNIST dataset contains 60 thousand black-and-white images, representing hand-written digits.

- **CIFAR10**. It is a CNN used for object classification for the CIFAR10 dataset [70]. The implementation we have chosen is taken from a Keras tutorial [71], which is capable of achieving 78% accuracy. The CIFAR10 dataset contains 60 thousand RGB images and 10 object categories.

YOLO V3 has been our primary target because it is a deep model, containing several different operator types and instances, and it is a state-of-the-art model employed in real safety-critical systems. For these reasons, it was our primary choice because it represents an excellent with which test our methodology and with which we have designed and performed all the experiments regarding the architectural fault injections. Therefore, it has been possible to execute YOLO V3 with SASSIFI and Caffe, adopting some precautions that will be explained in the next section. The same does not apply for TensorFI because it has not been possible to execute YOLO V3 with it due to its technical limitations, as pointed in Section 2.5.1.5. Faced with this fact, we have chosen to use two other CNN models, which are significantly smaller, CIFAR10 and LeNet-5, and allow us to compare the execution times of our approach and TensorFI. In this way, we are still able to compare our approach with the two baselines.

7.2 Accuracy Validation

In Section 3.4, we have defined the accuracy as the capability of reproducing in the network's output the same effects of the architectural fault injection campaigns. The baseline for this validation is SASSIFI because it is the state-of-the-art of GPU fault injection and the most accurate tool since its fault models are highly accurate. The network used for this validation is the TensorFlow implementation of YOLO V3, as explained above.

The rules with which we compare the effects of the injected errors and faults, respectively, are presented as follows. The output of the YOLO V3 network is a list of detection for each object category. Thus, we have 80 lists, possibly empty if no object associated with that list has been detected, containing the detection expressed as tuples. Each list is associated with an object category; for instance, the list 0 is associated with the object category 0 that represents the "person" class, and so on. The effects of a fault/error can be the removal or modification of a detection or the appearance of a new detection in any possible object category. We assume that the detection obtained though the architectural fault injection is our golden reference, and we check if the detection obtained trough error simulation report the same effects. We count how many times we can match the detection and, if divided by the total number of detection, it expresses the percentage of accuracy of our detection. The complement of that percentage express how many detection have not been matched on both sides.

Theoretically, to compare the two tools, we should have performed two campaigns, one with SASSIFI, and one with our error simulator, both targeting the whole YOLO V3 network, by injecting the same number of faults/errors and comparing the outputs. Unfortunately, a network-wide campaign cannot be performed with SASSIFI due to technical limits; SASSIFI is not able to load the whole CNN during the execution of the campaigns. Indeed, the functioning of SASSIFI introduces a huge time overhead when it comes to loading and allocating the GPU memory required by the CNN, which demands more than 1 Gb of memory.

To overcome this issue, we have modified the experimental framework to run with SASSIFI only the specific operator of the CNN to be corrupted while the rest of the data-flow graph is executed in TensorFlow by following this execution flow:

1. The subpart of the data-flow graph before the considered operator is executed in the standard TensorFlow environment to compute the

input tensors.

2. The fault injection campaign is performed by means of SASSIFI on the sole operator to be corrupted to collect, for each run, the output tensor.

3. For each collected output tensor, the subpart of the data-flow graph, after the considered operator, is executed in the standard TensorFlow environment by providing in input such a tensor.

In practice, for the sake of time, we have reused the outcomes of the fault injection campaign discussed in Chapter 5. For what concerns the error simulator, we injected only in the instances targeted by the architectural fault injection by inserting the same amount of errors like the one observed in the experiment. In this way, the two tools are comparable, and we are able to check if we obtain the same errors referred to as the output of the network having injected in the same locations of the network.

The architectural fault injection campaigns have resulted in 137,004 faulty tensors, which then are inserted in TensorFlow in the same operator's instance that has originated that faulty tensor. With the error simulation, we have performed the same error campaign by inserting 137004 errors in the same operator's instances and then collecting all the outputs. The analysis of the outputs, according to the procedure described above, has highlighted that our approach is 98.72% accurate, which means that we can reproduce the same effects of the architectural fault injection in the 98.72% of the cases. This result leads us to consider that our error models are validated and accurate as of the architectural fault injection.

7.3 Execution Times Analysis

The execution time is the time elapsed for performing an error simulation or fault injection campaign. The execution times are comparable only if the two tools are performing a campaign of the same size. This analysis is divided into two sections, one for each tool, which sections are presented as follows.

7.3.1 SASSIFI

During the experiments carried out for the realization of the framework, we have injected 360 thousand faults with SASSIFI, which have resulted in 137,004 faulty tensors.

Table 7.1: Execution times of the error simulation and SASSIFI, either in hybrid and full version. The values are expressed in hours.

Campaign of 360k faults resulted in 137k faulty tensors		
SASSIFI + TensorFlow	**SASSIFI**	**Our Approach**
92 h	833 h	**15** h

In the previous section, we stated that it was not possible to execute the full network of YOLO V3 because SASSIFI induces a severe time overhead, due to the memory loading. The time required for a single execution of the whole network requires 12 minutes for the Instruction Output Value (IOV) and Register File (RF) modes, and 19 minutes for the Instruction Output Address (IOA) mode. If we wanted to execute the same amount of injections with the full network as done in the experiments, the estimated time required is quantifiable as 833 hours, which is more than a month ($\approx$ 34 days).

The time required by SASSIFI to inject 360 thousand faults has been 277,880 seconds ($>$ 77 hours). The 360 thousand faults have generated 137,004 faulty tensors, which have been reinserted in TensorFlow, requiring 54,281 seconds ($>$15 hours). In total, the full experiment has required 332,081 seconds ($>$ 92 hours).

In total, we have injected 360,000 faults resulting at a time of 277,880 seconds ($>$ 77 hours), obtaining 137,004 faulty tensors to be reinserted in the CNN, which represents the second component. The reinsertion of the 137,004 faulty tensors has required 54,281 seconds ($>$ 15 hours). The overall time required by the architectural fault injection is 332,081 seconds ($>$ 92 hours).

In the context of the error simulation, we are able to generate as many faulty tensors as we want at run-time with no additional time overhead. Thus, we need to insert, with our error simulator, in the same instances of the CNN used in the architectural fault injection, the same amount of faulty tensors. The insertion with our tool of 137,004 faulty tensors has required 54,414 seconds ($>$ 15 hours).

The results highlight that our error simulator induces a speedup of $6.1x$ times than the traditional approach with a saving of 277,667 seconds ($>$ 77 hours) for injecting the same amount of faulty tensors in the CNN. In both cases, the setup and pre-processing times have not been considered because they are constant and performed once, not changing the overall outcome. The execution times are summarized in Table 7.1.

Table 7.2: Comparison of execution times between TensorFI and our approach. The values are expressed in seconds.

Campaign of 10,000 error simulations		
Dataset	**TensorFI**	**Our Approach**
MNIST	1098.71 s	**24.74 s**
CIFAR10	2409.47 s	**37.62 s**

Table 7.3: The subject of the analysis is our tool and the symbols are the evaluation of the current tool for that KPI compared to our tool.

KPI	**SASSIFI**	**TensorFI**
Execution Times	6.1x	44.41x - 64.04x
Accuracy	$\approx$	$>>$

7.3.2 TensorFI

As anticipated at the beginning of this chapter, we cannot use the YOLO V3 network for TensorFI, due to its technical limits. Therefore, we designed two ad-hoc experiments to test the execution times of TensorFI, each composed of 10,000 errors to inject in LeNet-5 and CIFAR10, respectively. Our tool has performed the MNIST experiment in 24.74 seconds, while the CIFAR10 in 37.62 seconds. TensorFI has performed the two experiments in 1098.71 and 2409.47 seconds, respectively. Our tool induces a speedup compared to TensorFI, which varies from $44.41x$ for the MNIST dataset to $64.04x$ for the CIFAR10 dataset. If we compare the results to a plain execution, i.e., the time duration of 10,000 executions in clean conditions, our simulator induces a mean time overhead of $1.55x$ while TensorFI exhibits a time overhead of $67.79x$ for the MNIST dataset and $101.52x$ for the CIFAR10 dataset. Table 7.2 shows the different execution times obtained by our error simulator and TensorFI.

The results obtained by TensorFI are due to implementation flaws because it does not provide any advanced injection technique, like the checkpointing, nor employs optimizations, such as the reusing of the intermediate computations.

7.4 Concluding Remarks

After having compared our framework with the current state-of-the-art tools, we can outline the concluding remarks regarding our tool. Table 7.3

includes all the results in a schematic form, which are described as follows.

SASSIFI is the current state-of-the-art tool for what concerns the GPU fault injection, being the most accurate tool available. However, it has severe limitations for what concerns the applicability and integration with the target application and the considerable amount of time required for the execution of the campaigns. Our error simulator overcomes the limitations addressed by SASSIFI. The tool has validated error models, either by construction and comparison, which makes it significant and relevant, with an accuracy that is comparable to the one achievable with SASSIFI, being able to reproduce the 98.72% of the errors obtained with SASSIFI. The execution times achieved by our error simulator enables us to perform all the experiments executed in this thesis is only 15 hours compared to the 92 hours required by SASSIFI. This is transformed to a speedup of the $6.1x$, which makes it preferable as a tool for testing the reliability of such systems.

TensorFI is the first tool that tries to join two worlds by applying the error simulation in the context of the reliability analysis of CNNs executed on GPU. Although the idea is heading in the right direction and has been a source of inspiration for this work, the realization leaves much to be desired. The error models it employs are not validated; thus, the whole reliability analysis performed with this tool is not significant and useless because it responds to a problem that does not exist. The execution times are far from being optimal and acceptable, introducing an excessive time overhead and a poorly designed implementation of the tool with no efficient injection techniques like check-pointing or any sort of cache. Our tool induces a speedup of the $44.41x$ and $64.04x$ for the MNIST and CIFAR10 models, respectively.

Chapter 8

Conclusions and Future Work

Nowadays, Convolutional Neural Networks (CNNs) are widely employed for perception functionalities in many safety-critical systems because of their high accuracy, which overcomes traditional Computer Vision (CV) algorithms. Among all these systems, CNNs are highly engaged in Autonomous Driving Systems (ADSs), in which they perform many machine vision tasks within the perception modules in such systems. When deployed in safety-critical systems, the CNNs must deal and comply with temporal constraints, which make it necessary to execute them fast. Graphic Process Units (GPUs) are computing devices that, thanks to their parallel architecture and the Single Instruction Multiple Data (SIMD) paradigm, are capable of accelerating the execution of CNNs up to make them compliant with the temporal requirements of safety-critical systems. Therefore, it is very common to find in safety-critical systems the duo composed of CNN executed on GPU.

This couple must ensure the proper functioning in any possible situation, so it is necessary to carry out the reliability analysis on them to outline their behavior against faults, which may make the system to deviate from its nominal behavior. Traditionally, this type of analysis is complex to carry out because the CNN and the GPU are placed in two different abstraction levels, respectively, the application level and the architectural level. The currently available tools and state-of-the-art methodologies still make it difficult to connect these two levels, limiting the efficacy of the reliability analysis.

In this thesis, we have proposed a novel methodological framework for the reliability analysis of CNNs executed on GPUs by facilitating the connection between the architecture level where faults are classically emulated and the application level where the produced errors are analyzed. The

framework is composed of two parts, the error modeling, and the error simulation, respectively.

The error modeling is a methodology that enables us to characterize the errors appearing in each CNN operator's output in response to GPU fault injection campaigns, performed through the state-of-the-art GPU fault injector, SASSIFI [58]. The characterization leads us to create an error model repository that contains the errors modeled according to three parameters, the cardinalities, the spatial patterns, and the domains of corrupted values. These three parameters are the application-oriented representation of the errors originated by architectural faults. Therefore, error modeling is the connection link between the two abstraction levels, which makes possible the reliability analysis of the overall system.

The error simulation is a methodology that enables us to assess the reliability of a target CNN by sabotaging the outputs of its operators according to the models defined in the error models repository. This methodology is implemented through an error simulator tool, built upon TensorFlow, capable of targeting CNN models in their application domain, exploiting advanced injection techniques, like the check-pointing, or optimizations, like the caching of intermediate computations.

Either the error modeling and error simulation have been validated through several experiments against the current state-of-the-art tools and best practices. The metrics with which we have evaluated our framework are execution times, the times spent on performing an error simulation campaign, and the accuracy of the error models. The framework has been compared with SASSIFI, which is the state-of-the-art GPU fault injector and the most accurate tool for the GPU. The experiments highlighted that our framework is $6.1x$ times faster than SASSIFI and is almost accurate as it since it is able to reproduce the 98.72% of the errors obtained during the fault injection campaigns. The second tool with which we have compared our framework is TensorFI [40], which is the only error simulator publicly available for the reliability analysis of CNNs. Although TensorFI is a novel contribution in this field, it has evident methodological and implementation flaws. Our framework compared to TensorFI, it turns out that our error simulator is from $44.41x$ to $64.04x$ times faster than TensorFI when performing the same error simulation campaign. The accuracy of TensorFI can be evaluated only qualitatively. However, it embeds error models that are not validated nor similar to our error models, making TensorFI a tool of doubtful value since its error models are not accurate and so the reliability analysis is not truthful.

8.1 Future work

The discussed work presents various future directions aimed at extending the considered working scenario, as discussed in the following.

Finalizing the error modeling. The error modeling is a general methodology for characterizing the errors in the output of CNN operators. In this work, we have applied this methodology only on a relevant subset of operators included in the YOLO V3 CNNs considered in the experimental sessions. Therefore, it is worth to extend the set of analyzed operators with the missing ones (for instance, the max pooling operator) to derive a comprehensive repository of error models, which is applicable to any CNN model.

Porting to other Machine Learning (ML) frameworks. TensorFlow has been considered in this thesis as the framework to integrate the error simulator because it is the reference point of the field. However, there exist other popular ML frameworks that we may consider in the future to integrate the error simulator. Caffe is one of them, and although it is discontinued, it still covers a niche demanding raw performances, which justifies our interest in this framework. PyTorch [72] is another high-level ML framework, which is qualified as the direct competitor of TensorFlow for popularity. The porting of the proposed error simulator to these frameworks would enable us to cover the majority of the tools used by the ML community, thus extending the benefits of our reliability analysis methodology in the case of design of CNNs for mission-/safety-critical applications.

Considering other types of devices. In this work, the system under analysis was the duo CNN executed on GPU. However, there are also alternative solutions to execute CNNs, for instance by accelerating on Field Programmable Gate Array (FPGA) devices, or in case of unavailability of any accelerator, on CPU. One of the most interesting future work is to consider such alternative devices. Indeed, the proposed methodological framework is highly flexible; it will require only to re-perform a new error modeling activity on the newly considered device, in the same way we have shown for the GPU. The new output of such an activity will be a new list of error models that will be integrated into the repository so that the error simulator can also be employed in these new scenarios without requiring any relevant modification.

Considering other types of ML and image processing applications. The CNNs are just a subset of all the models present in the ML world, particularly suitable for managing images, and, therefore, they are widely employed in the CV field. However, there exist other ML models and image processing applications widely employed in safety-critical systems, such as Recurrent Neural Networks (RNNs), Histogram of Oriented Gradients method, Bayesian Networks, and many more. It would be worth applying the proposed methodological framework to analyze all these other applications to study their intrinsic error resilience.

Integration of the proposed framework in reliability-driven design flows. The output of the methodological framework applied on a given case study is a report of the susceptibility to faults of the various parts of the application under analysis. Such output is highly valuable in the cases we aim at performing a cost-aware hardening of a safety-critical system, for instance by applying the selective duplication as in [51]. Therefore, a last interesting future work consists in the definition of comprehensive reliability-driver design flows capable at exploiting the outcome of our reliability analysis for a cost-effective hardening; the integration of all these separate methods in a single picture will offer the possibility to improve the quality of both the obtained system implementation, in terms of higher performance and reliability, and the design flow, in terms of a reduced design time and effort.

Bibliography

[1] M. Campbell, M. Egerstedt, J. P. How, and R. M. Murray, "Autonomous driving in urban environments: approaches, lessons and challenges," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 368, no. 1928, pp. 4649–4672, 2010.

[2] S. International, "Automated driving: levels of driving automation are defined in new sae international standard j3016," 2014.

[3] E. Normand, "Single event upset at ground level," *IEEE transactions on Nuclear Science*, vol. 43, no. 6, pp. 2742–2750, 1996.

[4] "Report: Vehicles in use - europe 2019 acea - european automobile manufacturers' association." `https://www.acea.be/publications/article/report-vehicles-in-use-europe-2019`. (Accessed on 03/20/2020).

[5] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. Software available from tensorflow.org.

[6] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," *arXiv preprint arXiv:1408.5093*, 2014.

[7] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation applied to handwrit-

ten zip code recognition," *Neural computation*, vol. 1, no. 4, pp. 541–551, 1989.

[8] "Nn svg." `http://alexlenail.me/NN-SVG/LeNet.html`. (Accessed on 12/18/2019).

[9] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[10] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *CoRR*, vol. abs/1502.03167, 2015.

[11] A. L. Hodgkin and A. F. Huxley, "A quantitative description of membrane current and its application to conduction and excitation in nerve," *The Journal of physiology*, vol. 117, no. 4, pp. 500–544, 1952.

[12] B. Xu, N. Wang, T. Chen, and M. Li, "Empirical evaluation of rectified activations in convolutional network," *CoRR*, vol. abs/1505.00853, 2015.

[13] D. Liu, "A practical guide to relu - danqing liu - medium." `https://medium.com/@danqing/a-practical-guide-to-relu-b83ca804f1f7`, 11 2017. (Accessed on 12/17/2019).

[14] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th international conference on machine learning (ICML-10)*, pp. 807–814, 2010.

[15] A. L. Maas, A. Y. Hannun, and A. Y. Ng, "Rectifier nonlinearities improve neural network acoustic models," in *Proc. icml*, vol. 30, p. 3, 2013.

[16] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *Proceedings of the IEEE international conference on computer vision*, pp. 1026–1034, 2015.

[17] J. Han and C. Moraga, "The influence of the sigmoid function parameters on the speed of backpropagation learning," in *International Workshop on Artificial Neural Networks*, pp. 195–201, Springer, 1995.

[18] F. Chollet *et al.*, "Keras." `https://keras.io`, 2015.

[19] "Protocol buffers — google developers." https://developers.google.com/protocol-buffers/. (Accessed on 02/26/2020).

[20] "Graphics pipeline - win32 apps — microsoft docs." https://docs.microsoft.com/en-us/windows/win32/direct3d11/overviews-direct3d-11-graphics-pipeline. (Accessed on 02/26/2020).

[21] NVIDIA, "Cuda c programming guide." https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[22] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," *arXiv preprint arXiv:1410.0759*, 2014.

[23] "Nvidia-kepler-gk110-gk210-architecture-whitepaper.pdf." https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-product-literature/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf. (Accessed on 02/26/2020).

[24] C. Constantinescu, "Trends and challenges in vlsi circuit reliability," *IEEE micro*, vol. 23, no. 4, pp. 14–19, 2003.

[25] D. Binder, E. C. Smith, and A. Holman, "Satellite anomalies from galactic cosmic rays," *IEEE Transactions on Nuclear Science*, vol. 22, no. 6, pp. 2675–2680, 1975.

[26] T. C. May and M. H. Woods, "A new physical mechanism for soft errors in dynamic memories," in *16th International Reliability Physics Symposium*, pp. 33–40, IEEE, 1978.

[27] J. Maiz and N. Seifert, "Introduction to the special issue on soft errors and data integrity in terrestrial computer systems," *IEEE Transactions on Device and Materials Reliability*, vol. 5, pp. 303–304, Sep. 2005.

[28] "Geforce gtx titan — specifications — geforce." https://www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan/specifications. (Accessed on 01/09/2020).

[29] "Scheda grafica definitiva per pc titan rtx con turing — nvidia." https://www.nvidia.com/it-it/deep-learning-ai/products/titan-rtx/. (Accessed on 01/09/2020).

[30] E. J. Wyrwas, "Proton testing of nvidia gtx 1050 gpu," 2017.

[31] E. J. Wyrwas, C. Szabo, K. A. LaBel, M. Campola, and M. O'Bryan, "Standardizing gpu radiation test approaches," 2018.

[32] S. K. S. Hari, S. V. Adve, and H. Naeimi, "Low-cost program-level detectors for reducing silent data corruptions," in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, pp. 1–12, IEEE, 2012.

[33] "Cuda-gdb :: Cuda toolkit documentation." `https://docs.nvidia.com/cuda/cuda-gdb/index.html`. (Accessed on 01/14/2020).

[34] B. Fang, K. Pattabiraman, M. Ripeanu, and S. 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)*, pp. 221–230, IEEE, 2014.

[35] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, pp. 163–174, IEEE, 2009.

[36] M. Stephenson, S. K. Sastry Hari, Y. Lee, E. Ebrahimi, D. R. Johnson, D. Nellans, M. O'Connor, and S. W. Keckler, "Flexible software profiling of gpu architectures," in *ACM SIGARCH Computer Architecture News*, vol. 43, pp. 185–197, ACM, 2015.

[37] G. Li, K. Pattabiraman, C.-Y. Cher, and P. Bose, "Understanding error propagation in gpgpu applications," in *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 240–251, IEEE, 2016.

[38] J. Wei, A. Thomas, G. Li, and K. Pattabiraman, "Quantifying the accuracy of high-level fault injection techniques for hardware faults," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pp. 375–382, IEEE, 2014.

[39] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, p. 75, IEEE Computer Society, 2004.

[40] G. Li, K. Pattabiraman, and N. DeBardeleben, "Tensorfi: A configurable fault injector for tensorflow applications," in *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pp. 313–320, IEEE, 2018.

[41] F. F. dos Santos, L. Carro, and P. Rech, "Kernel and layer vulnerability factor to evaluate object detection reliability in gpus," *IET Computers & Digital Techniques*, vol. 13, no. 3, pp. 178–186, 2018.

[42] N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," 2005.

[43] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 779–788, 2016.

[44] G. Li, S. K. S. Hari, M. Sullivan, T. Tsai, K. Pattabiraman, J. Emer, and S. 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*, p. 8, ACM, 2017.

[45] "tiny-dnn/tiny-dnn: header only, dependency-free deep learning framework in c++14." `https://github.com/tiny-dnn/tiny-dnn`. (Accessed on 01/16/2020).

[46] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, pp. 1097–1105, 2012.

[47] F. F. dos Santos, P. F. Pimenta, C. Lunardi, L. Draghetti, L. Carro, D. Kaeli, and P. Rech, "Analyzing and increasing the reliability of convolutional neural networks on gpus," *IEEE Transactions on Reliability*, vol. 68, no. 2, pp. 663–677, 2018.

[48] K.-H. Huang and J. A. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE transactions on computers*, vol. 100, no. 6, pp. 518–528, 1984.

[49] A. Bosio, P. Bernardi, A. Ruospo, and E. Sanchez, "A reliability analysis of a deep neural network," in *2019 IEEE Latin American Test Symposium (LATS)*, pp. 1–6, IEEE, 2019.

[50] "Darknet: Open source neural networks in c." `https://pjreddie.com/darknet/`. (Accessed on 01/16/2020).

[51] D. Oliveira, P. Navaux, and P. Rech, "Increasing the efficiency and efficacy of selective-hardening for parallel applications," in *2019 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pp. 1–6, IEEE, 2019.

[52] F. Fernandes, L. Weigel, C. Jung, P. Navaux, L. Carro, and P. Rech, "Evaluation of histogram of oriented gradients soft errors criticality for automotive applications," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 13, no. 4, p. 38, 2016.

[53] D. Hendrycks and T. Dietterich, "Benchmarking neural network robustness to common corruptions and perturbations," *arXiv preprint arXiv:1903.12261*, 2019.

[54] A. Azulay and Y. Weiss, "Why do deep convolutional networks generalize so poorly to small image transformations?," *Journal of Machine Learning Research*, vol. 20, no. 184, pp. 1–25, 2019.

[55] O. Bastani, Y. Ioannou, L. Lampropoulos, D. Vytiniotis, A. Nori, and A. Criminisi, "Measuring neural net robustness with constraints," in *Advances in neural information processing systems*, pp. 2613–2621, 2016.

[56] D. A. G. De Oliveira, L. L. Pilla, M. Hanzich, V. Fratin, F. Fernandes, C. Lunardi, J. M. Cela, P. O. A. Navaux, L. Carro, and P. Rech, "Radiation-induced error criticality in modern hpc parallel accelerators," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 577–588, IEEE, 2017.

[57] "cublas — nvidia developer." `https://developer.nvidia.com/cublas`. (Accessed on 03/17/2020).

[58] S. K. S. Hari, T. Tsai, M. Stephenson, S. W. Keckler, and J. Emer, "Sassifi: An architecture-level fault injection tool for gpu application resilience evaluation," in *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 249–258, IEEE, 2017.

[59] F. Ayatolahi, B. Sangchoolie, R. Johansson, and J. Karlsson, "A study of the impact of single bit-flip and double bit-flip errors on program ex-

ecution," in *International Conference on Computer Safety, Reliability, and Security*, pp. 265–276, Springer, 2013.

[60] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Computing Surveys (CSUR)*, vol. 23, no. 1, pp. 5–48, 1991.

[61] J. Redmon and A. Farhadi, "Yolo9000: Better, faster, stronger," *arXiv preprint arXiv:1612.08242*, 2016.

[62] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement," *arXiv*, 2018.

[63] "numpy.lib.format — numpy v1.17 manual." `https://docs.scipy.org/doc/numpy/reference/generated/numpy.lib.format.html#module-numpy.lib.format`. (Accessed on 03/24/2020).

[64] "rogersce/cnpy: library to read/write .npy and .npz files in c/c++." `https://github.com/rogersce/cnpy`. (Accessed on 03/24/2020).

[65] "Alessandrotoschi/a-methodology-for-error-simulation-in-cnns-executed-on-gpu-results." `https://github.com/AlessandroToschi/A-Methodology-for-Error-Simulation-in-CNNs-Executed-on-GPU-Results`. (Accessed on 03/29/2020).

[66] "Apollo." `http://apollo.auto/`. (Accessed on 01/22/2020).

[67] "Autoware.ai." `https://www.autoware.ai/`. (Accessed on 02/12/2020).

[68] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft coco: Common objects in context," in *European conference on computer vision*, pp. 740–755, Springer, 2014.

[69] "Mnist handwritten digit database, yann lecun, corinna cortes and chris burges." `http://yann.lecun.com/exdb/mnist/`. (Accessed on 03/23/2020).

[70] "Cifar-10 image classification in tensorflow - towards data science." `https://towardsdatascience.com/cifar-10-image-classification-in-tensorflow-5b501f7dc77c`. (Accessed on 03/24/2020).

[71] "keras-apache-mxnet/cifar10_cnn.py at master · awslabs/keras-apache-mxnet." `https://github.com/awslabs/keras-apache-mxnet/blob/master/examples/cifar10_cnn.py`. (Accessed on 03/27/2020).

[72] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems*, pp. 8024–8035, 2019.

Acronyms

ADS Autonomous Driving System. 1–4, 23, 24, 101

API Application Program Interface. 16, 81, 82, 90

CNN Convolutional Neural Network. V–XII, XIX, 4, 5, 7–18, 25, 29, 32–34, 36–43, 45–57, 59–61, 64, 65, 69–71, 74, 77, 79, 80, 82, 86, 91, 93–95, 97, 99, 101–104

CV Computer Vision. V, IX, 1, 4, 9, 19, 101, 104

FPGA Field Programmable Gate Array. 103

GEMM General Matrix Multiplications. 18, 33, 62, 63, 66, 72

GPGPU General Purpose GPU. 19

GPR General Purpose Register. 65, 68

GPU Graphic Process Unit. V–VII, IX–XI, XV, XVI, XIX, 4, 5, 7, 11, 15–29, 31–43, 45, 47–50, 53, 55–60, 63, 64, 66, 67, 70–73, 76, 77, 80, 89, 91, 93, 95, 99, 101–103

HPC High Performance Computing. 18

IOA Instruction Output Address. 27, 64, 65, 68, 97

IOV Instruction Output Value. XXI, 27, 32, 64, 65, 68, 97

KPI Key Performance Indicator. XVI, 37, 42, 43, 93, 98

ML Machine Learning. VI, VII, X, XI, XVII, 1, 5, 7, 9, 15, 16, 18, 19, 29–31, 35, 39, 41, 42, 46, 47, 57, 59, 60, 63, 90, 91, 103, 104

PR_OP Predicate Operation. 65, 68

RF Register File. XXI, 27, 32, 64, 65, 68, 97

RNN Recurrent Neural Network. 104

SAE Society of Automotive Engineers. 1

SDC Silent Data Corruption. 25, 28, 32–35, 38, 46, 47

SEU Single Event Upset. 24, 25, 37, 38, 42

SIMD Single Instruction Multiple Data. V, IX, 19, 20, 101

SIMT Single Instruction Multiple Thread. 20, 22, 49, 50

SM Streaming Multiprocessor. 19–22

STORE_OP Store Operation. 65, 68