UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

FERNANDO FERNANDES DOS SANTOS

Understanding and Improving GPUs' Reliability Combining Beam Experiments with Fault Simulation

Thesis presented in partial fulfillment
of the requirements for the degree of
Doctor of Computer Science

Advisor: Prof. Dr. Paolo Rech
Coadvisor: Prof. Dr. Luigi Carro

Porto Alegre
December 2021

CIP — CATALOGING-IN-PUBLICATION

dos Santos, Fernando Fernandes

Understanding and Improving GPUs' Reliability Combining Beam Experiments with Fault Simulation / Fernando Fernandes dos Santos. – Porto Alegre: PPGC da UFRGS, 2021.

134 f.: il.

Thesis (Ph.D.) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2021. Advisor: Paolo Rech; Coadvisor: Luigi Carro.

1. GPUs. 2. Reliability. 3. High Performance Computing. 4. Safety critical systems. I. Rech, Paolo. II. Carro, Luigi. III. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos André Bulhões
Vice-Reitora: Prof. Patricia Helena Lucas Pranke
Pró-Reitor de Pós-Graduação: Prof. Júlio Otávio Jardim Barcelos
Diretora do Instituto de Informática: Prof. Carla Maria Dal Sasso Freitas
Coordenador do PPGC: Prof. Claudio Rosito Jung
Bibliotecário-chefe do Instituto de Informática: Alexsander Borges Ribeiro

ABSTRACT

Graphics Processing Units (GPUs) have moved from being dedicated devices for multimedia and gaming applications to general-purpose accelerators, employed in High Performance Computing (HPC) and safety-critical applications, such as autonomous vehicles. This market shift led to a burst in the GPU's computing capabilities and efficiency, significant improvements in the programming frameworks and performance evaluation tools, and a sudden concern about their hardware reliability.

In order to evaluate the GPU reliability, researchers expose a device to a neutron beam and perform fault injection to simulate the fault propagation. While beam experiments provide a very realistic error rate of the device, it lacks fault propagation visibility. Contrarily, fault injection allows the complete visibility of the fault propagation, but the fault simulation and the error model are often limited to user-accessible resources and may lead to unrealistic results. Consequently, a methodology to accurately estimate the error rate of a device is necessary to answer two of the fundamental open questions in GPU reliability evaluation: (1) whether fault simulation provides representative results and can be used to predict the Failure In Time (FIT) rates of codes running on GPUs. (2) are the single and double bit-flip accurate error models to simulate faults on a GPU.

This thesis presents a novel FIT estimation approach to predict the NVIDIA GPUs' error rate. The proposed FIT estimation is achieved by comparing and combining high-energy neutron beam experiments that account for more than 13 million natural terrestrial exposure years, an extensive architectural-level fault simulation (using SASSIFI and NVBitFI), and detailed application-level profiling, requiring more than *1,000* GPU hours. Results show that, in most cases, the estimated Silent Data Corruption (SDC) rate is sufficiently close (differences lower than $5\times$) to the experimentally measured SDC rates. The knowledge from the FIT estimation is then used to present a new error model based on the relative error in opposition to single/double bit flip. The relative error is based on a new method that extracts the relative error differences from a fault injection at the Register-Transfer Level (RTL).

Using the experimental, architectural, and algorithmic analysis, this work presents also two novel hardening solutions for HPC and safety-critical applications: (1) Reduced Precision Duplication With Comparison (RP-DWC). RP-DWC's primary goal is to lower the overhead of Duplication With Comparison (DWC) by executing the redundant copy in reduced precision. RP-DWC achieves an excellent coverage (up to 86%) with

minimal overheads (as low as 0.1% time and 24% energy consumption overhead). (2) Dedicated software solutions for hardening Convolutional Neural Networks (CNNs). The Algorithm-Based Fault Tolerance (ABFT) employed to the matrix multiplication (the core of the CNNs) can correct more than 60% of the critical SDCs in a CNN, while re-designing the CNN's max pool layers leads to a detection up to 98% of SDCs. Additionally, this work is the first to evaluate the CNNs' error rate and CNNs' hardening efficiency on neutron beam experiments.

Keywords: GPUs. Reliability. High Performance Computing. safety critical systems.

Entendendo e Melhorando a Confiabilidade das GPUs Combinando Experimentos com Feixe e Injeção de Falhas

RESUMO

Graphics Processing Units (GPUs) passaram de dispositivos dedicados a aplicações multimidia e *gaming*, para se tornarem aceleradores de propósito geral usados em *High Performance Computing* (HPC) e aplicações críticas como carros autônomos. Tal mudança no mercado das GPUs levou a um aumento nas capacidades computacionais, eficiência energética, melhoras nas ferramentas de programação e de análise de performance, e também um aumento na preocupação com a confiabilidade do hardware das GPUs.

Com o objetivo de avaliar a confiabilidade das GPUs, pesquisadores expõe o dispositivo a um feixe de nêutrons e realizam injeções de falhas para simular a propagação das falhas. Se por um lado experimentos de radiação provêm uma taxa de falhas realista, por outro lado eles não permitem visualizar a propagação das falhas no hardware e na aplicação. Contrariamente, a injeção de falhas permite a completa visualização da propagação de uma falha injetada, porém, na maioria das vezes os modelos de falhas são por sua vez limitados ao que o pesquisador consegue acessar e modificar, o que pode levar a resultados não realísticos. Consequentemente, uma metodologia para estimar com precisão a taxa de falhas de um dispositivo é necessária para responder duas questões fundamentais na avaliação da confiabilidade das GPUs: Se a injeção de falhas consegue prover resultados representativos que podem ser usados para estimar a taxa *Failure In Time* (FIT) de códigos executando em GPUs, e se os modelos de falhas que consideram a modificação de um único bit ou dois bits são modelos acurados para simular falhas em uma GPU. Sendo assim, essa tese propõe uma nova metodologia para estimar a taxa *Failure In Time* de GPUs NVIDIA. A metodologia proposta é possível através da comparação e combinação dos resultados de experimentos de radiação realizados em um feixe de nêutrons de alta energia, que correspondem por mais de 13 milhões de anos de exposição no fluxo terrestre natural, e extensivos experimentos utilizando simulação de falhas (usando SASSIFI e NVBITFI), e *profiling* de aplicações que requerem mais de *1,000* horas de GPU. Os resultados mostram que, para a maioria dos casos, as taxas de *Silent Data Corruptions* (SDCs) estimadas são suficientemente perto (diferenças menores que $5\times$) das estimadas experimentalmente nos testes de radiação. O conhecimento extraído da estimação do FIT é então usado para propor um novo modelo de falhas em oposição ao bit flip único ou

duplo. O modelo de falhas proposto é baseado no erro relativo extraído de injeção de falhas em *Register Transfer Level* (RTL) comparando as diferenças observadas na saída das injeções.

Usando uma análise experimental, arquitetural, e algorítmica, esse trabalho apresenta também duas novas soluções de tolerância a falhas para HPC e aplicações críticas. A primeira solução proposta é a *Reduced Precision Duplication With Comparison* (RP-DWC), onde o principal objetivo é diminuir a sobrecarga causada pela *Duplication With Comparison* (DWC) executando a cópia redundante em uma precisão reduzida. A técnica RP-DWC consegue uma taxa de detecção excelente, 86%, com sobrecarga mínima, podendo chegar aumento de tempo de execução mínimos de 0.1%, e em alguns casos somente 24% de aumento no consumo de energia. O segundo tipo de solução proposta é voltado para *Convolutional Neural Network*, onde duas modificações foram apresentadas. A técnica já conhecida, *Algorithm Based Fault Tolerance* (ABFT) empregada as multiplicações de matrizes (maior parte do processamento das CNNs) conseguem corrigir mais de 60% dos SDCs críticos em uma CNN, enquanto recriando camadas especificas de uma CNN, *max-pool*, foi capaz de detectar 98% dos SDC. Adicionalmente, esse trabalho também é o primeiro a validar a taxa FIT de CNNs, como também a eficiência de tolerância a falhas aplicadas a uma CNN em experimentos de radiação.

Palavras-chave: GPUs, confiabilidade, Computação de Alta Performance, sistemas críticos.

LIST OF ABBREVIATIONS AND ACRONYMS

ABFT	Algorithm-Based Fault Tolerance
AVF	Architecture Vulnerability Factor
BFS	Breadth-first search
CCL	Connected Component Labeling
CFD	Computational Fluid Dynamics
CNN	Convolutional Neural Network
COTS	Commercial Of The Shelf
CUDA	Compute Unified Device Architecture
DMR	Double Modular Redundancy
DUE	Detected Unrecoverable Error
DWC	Duplication With Comparison
ECC	Error Correction Code
EPL	Expected Precision Loss
FIT	Failure In Time
GPU	Graphics Processing Unit
HPC	High Performance Computing
IOV	Instruction Output Value model
IPC	Instructions Per Cycle
ISA	Instruction Set Architecture
LUD	LU Decomposition
MWBF	Mean Work Between Failures
MxM	Matrix Multiplication
NW	Needleman–Wunsch
RP-DWC	Reduced Precision Duplication With Comparison
RTL	Register-Transfer Level
SASS	Source And Assembly

SDC	Silent Data Corruption
SECDED	Single Error Correction Double Error Detection
SM	Streaming Multiprocessor
TMR	Triple Modular Redundancy
TRE	Tolerated Relative Error
YOLO	You Only Look Once

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

Graphics Processing Units (GPUs) have evolved from supporting hardware for user applications and graphics rendering to general-purpose accelerators extensively employed in High Performance Computing (HPC) and safety-critical applications such as autonomous vehicles and aerospace markets. The highly parallel architecture of GPUs, in fact, perfectly fits the computational characteristic of most HPC codes and is incredibly efficient in executing matrix multiplication, which is the computing core of Convolutional Neural Networks (CNNs) used to detect objects in autonomous vehicles. The most recent GPU architecture advances, such as tensor core and mixed-precision functional units, move toward improving the architecture performances and software flexibility for HPC and deep learning applications.

Consequently, GPU vendors have been working on many architecture modifications to improve the GPUs reliability while maintaining high performances (HARI; ADVE; NAEIMI, 2012; RECH et al., 2014; WADDEN et al., 2014a; HARI et al., 2021). The researchers have proposed hardening at different levels of the GPU hardware/software, such as in the memory cell (RECH et al., 2014), Error Correction Code (ECC) (HARI; ADVE; NAEIMI, 2012), Redundant Multithreading execution (WADDEN et al., 2014a), and Algorithm Based Fault Tolerance for deep learning applications (HARI et al., 2021). Additionally, GPU vendors are working on the design of platforms compliant with strict automotive reliability standards as the ISO26262 (ISO, 2011; NVIDIA, 2018). As shown in Section 5.2, this thesis demonstrates that it is possible to detect 98% of the SDCs with a software-level hardening technique for CNNs executing on a GPU.

The research community has been carefully studying GPU reliability with both fault-injection (Wei et al., 2014; Hari et al., 2017; TSELONIS; GIZOPOULOS, 2016; TSAI et al., 2021), and beam experiments (Goncalves de Oliveira et al., 2016; SULLIVAN et al., 2021). Beam experiments provide a very realistic analysis but lack visibility. As errors are observed only when they manifest at the program/chip output, it is impossible to associate observed behaviors with their source of the fault (without specialized hardware for observability) and identify the most vulnerable GPUs' resources. In opposition, fault simulation provides complete visibility of the fault propagation and allows a detailed analysis of the propagation through the micro-architecture (i.e., the Architecture Vulnerability Factor (AVF)). This methodology enables identifying the resources or code portions that, once corrupted, are more likely to affect the computation. However, faults

usually can be injected only on a subset of the available resources, and the adopted fault-model risks to be unrealistic if not tuned with experiments.

The comparison between beam experiments and fault simulation is an essential missing piece in the GPU reliability evaluation puzzle that this research intends to find. Unfortunately, it is still mostly unclear whether a reliability evaluation based only on fault simulation is realistic. Evaluating the effectiveness of many error mitigation techniques requires fault-injection experiments. However, data based on fault simulations alone does not imply that the method will be useful in the field. This thesis investigates and compares the programs' Failure In Time (FIT) rates measured with beam experiments that count for more than $13 * 10^6$ years of natural exposure with the failure rates estimated from *400,000* fault injections using SASSIFI (Hari et al., 2017) and NVBitFI (TSAI et al., 2021), evaluating at which level and under which assumptions fault simulation can provide a realistic reliability evaluation for GPUs. Through beam experiments, this research presents the FIT rates of the main functional units (including mixed-precision and tensor core), register file, and shared memory of Kepler and Volta GPUs. The FIT rates of 15 representative codes for HPC and safety-critical applications, including three CNNs, are also presented. Some codes have been executed using different data types (integer, float-, single-, or half-precision) to understand the impact of mixed-precision on code's reliability.

Based on the data obtained from the beam experiments, profiling, and fault injection, this thesis proposes a multi-level fault simulation methodology to improve the GPU error model. The new fault injection method consists of performing fault injection at Register-Transfer Level (RTL) main structures and propagating the fault *syndrome* at the software level. The proposed method can join the accuracy of the RTL simulations with the performance of software fault injection. Then, for the first time, it is possible not only to unveil the effects of faults on otherwise hidden GPU resources but also to propose **a more detailed fault model** to be used in the reliability evaluation of complex codes. This information is essential, as it helps researchers focus on designing a hardening solution to a subset of critical resources.

The NVIDIA mixed-precision architectures are extremely interesting for the HPC and safety-critical markets. On GPUs, good object detection accuracy can be achieved through neural network representing data in half-precision float point (16 bits) or even in short integer (8 bits) (COURBARIAUX; BENGIO; DAVID, 2014; GUPTA et al., 2015). Several HPC applications could also be executed in reduced-precision, significantly im-

proving the computing efficiency (BREUER, 2005; PUENTE et al., 2014). As reliability is a primary concern for both safety-critical and HPC applications, it is mandatory to understand if and how mixed-precision influences the reliability of devices and codes. To take advantage of mixed-precision GPU hardware, this work moves a step forward in the performance efficiency of Duplication With Comparison (DWC) by presenting **Reduced Precision Duplication With Comparison (RP-DWC)**, an improvement over the traditional DWC approach, which consists of executing the replica in a lower precision. The results show that RP-DWC achieves an excellent coverage (up to 86%) with minimal overheads. The time overhead can be as low as 0.1%, while the energy consumption overhead can be as low as 24%.

In this thesis, two generations of NVIDIA GPUs are evaluated, Kepler and Volta. The two GPUs architectures have almost 10 years of release difference, which is as an interesting case study to understand where the architectural improvements can benefit the GPU reliability. Additionally, for both architectures and most codes, experiments are presented with Error Correction Code (ECC) enabled and disabled to evaluate the efficacy of GPUs built-in reliability solutions and distinguish between the contribution of logic and memory faults to the codes error rate.

1.1 Motivation

This thesis is about **understanding and improving the GPU reliability by combining the knowledge extracted from beam experiments, fault simulation, and profiling**. Currently, the difference between the data from the fault simulators and the error rates from radiation experiments is one of the most significant issues for GPU reliability analysis. This is the primary motivation of this research. In order to better characterize the investigation developed in this work, the following topics are also covered in this thesis:

The reliability of machine learning on GPUs: CNNs algorithms can exploit GPU's ability to support data and thread-level parallelism. However, researchers have been overly focused on the performance while neglecting other critical aspects, particularly reliability. While performance is vital in these applications, reliability needs to be paramount. It is not possible to tradeoff performance for reliability in safety-critical applications.

The reliability of mixed-precision GPUs: While it has been shown that mixed-precision operations are very beneficial in terms of computing and power efficiency, their

impact on the devices and applications reliability has not been thoroughly investigated yet.

1.2 Goals and organization

By performing multiple levels of fault injection, beam experiments, RTL, and software fault simulation, this work proposes a systematic and quantitative analysis of the error rate on NVIDIA GPUs. The main contributions of this work are:

- Combining beam experiments and fault simulation to deeply understand GPUs' reliability. This research also presents a comparison between the GPU's FIT rate measured with beam experiments with the FIT rate predicted using fault simulation and kernel profiling. This study provides essential information to ensure that fault simulation provides a realistic reliability evaluation.
- This work advances GPU reliability by characterizing how microarchitecture vulnerabilities in a GPU can undermine a CNN's reliability. Most previous works focused on HPC application reliability on a GPU.
- A new approach of hardening is proposed for mixed-precision architectures. This thesis goes a step forward in the performance efficiency of DWC by presenting Reduced-Precision DWC (RP-DWC), an improvement over the traditional DWC approach, which consists of executing the replica in a lower precision.
- For the first time for GPUs, a fine grain *RTL fault injection* (using FlexGripPlus) is combined with the flexibility and efficiency of *software fault injection in real GPUs*. With the RTL analysis, this research gathers the syndrome induced by faults in the micro-instruction output value, and produces an accurate fault model for the most common machine operations. As all GPU modules are accessible in the RTL model, presenting a reliability characterization that considers most GPU resources is possible.

The remainder of the document is organized as follows. Chapter 2 gives a background on the reliability of electronic devices and a short introduction of GPU's uses on machine learning and its impact on reliability. Chapter 3 presents some error rate concepts, the radiation tests setup, the fault simulators tools, and other artifacts used in this work, such as devices and codes. Chapter 4 presents the reliability evaluation using beam experiments and software fault simulation. New software fault tolerance techniques are

presented in Chapter 5. A methodology to estimate the FIT rate based on profiling and beam experiments is presented in Chapter 6. Chapter 7 presents the new fault injection methodology. Chapter 8 summarizes the main achievements and concludes this research.

2 BACKGROUND AND RELATED WORK

This chapter presents the background and the related work in radiation effects in computing devices, Graphics Processing Unit (GPU) reliability, Convolutional Neural Networks (CNNs), error rate estimation, and mixed-precision architectures necessary for this research.

2.1 Radiation Effects on Computing Devices

Galactic cosmic rays, interacting with the terrestrial atmosphere, trigger a flux of high-energy particles, mainly neutrons, that reach the ground. The natural flux of high-energy neutrons at sea level has been estimated to be about 13 $neutrons/((cm^2) \times h)$ (JEDEC, 2006). A terrestrial neutron strike may perturb a transistor's state, generating bit-flips in memory or current spikes in logic circuits that, if latched, lead to an error (BUCHNER et al., 1997; MAHATME et al., 2011).

Neutron-induced events are typically *soft* errors in the sense that the device is not permanently damaged. A new write operation will correctly store the value on the struck memory cell, and a new operation using the struck logic gate will provide the correct result. Soft errors are the worst kind of errors since they are harder to detect and are transient. Given the shrinking dimensions of transistors, the pursuit of lower power consumption, and the integration of several resources in a single chip, the probability of neutron-induced faults, in both memory and logic resources, has increased significantly (BAUMANN, 2005; SRIDHARAN et al., 2015). On a GPU, a soft error leads to three different main outcomes:

1. **No effect** on the program output, i.e., the fault is masked, the corrupted data is not used, or the circuit functionality is not affected;
2. A **Silent Data Corruption (SDC)** is an incorrect program output. The application finishes correctly, but its output does not contain correct data;
3. **Multiple Data Corruption** is an SDC caused by multiple errors in a GPU. For instance, an error in the Streaming Multiprocessor (SM) scheduler can corrupt the output of multiple threads;
4. A **Detected Unrecoverable Error (DUE)** is a detected but uncorrectable event, a crash, or device reboot that make the system abort abruptly.

Given a particle flux, the error rate of code being executed on a microprocessor depends on both the memory/logic sensitivity (BAUMANN, 2005; NOH et al., 2015), the number of resources adopted for computation, and on the probabilities for the fault to be propagated through the hardware design (the microarchitecture) and the program (MUKHERJEE et al., 2003; SRIDHARAN; KAELI, 2010).

Modern parallel computing architectures such as GPUs have some inherent reliability weaknesses (WUNDERLICH; BRAUN; HALDER, 2013; GOMEZ et al., 2014; Goncalves de Oliveira et al., 2016). A single particle-induced fault in the scheduler or shared memory is likely to affect the correctness of several parallel threads or kernels, leading to the corruption of multiple output values (RECH et al., 2013). Additionally, a single corrupted thread could feed thousands of future parallel threads with erroneous data, again leading to multiple errors (LI et al., 2016). Previous studies have already evaluated the reliability of FPGA, GPUs, and Xeon Phi running various codes through radiation experiments (BAUMANN, 2005; QUINN et al., 2005; OLIVEIRA et al., 2017c) or fault simulation (FANG et al., 2014; KALIORAKIS et al., 2015; LU et al., 2015; Hari et al., 2017; OLIVEIRA et al., 2017a; TSAI et al., 2021). By the time this work was done, this is the first research that combines beam experiments and fault simulation and compares the GPU's FIT rate measured with beam experiments with the FIT rate predicted using fault simulation and kernel profiling. This study provides essential information to ensure that fault simulation provides a realistic reliability evaluation.

2.2 Reliability evaluation

The error rate of computing devices, including GPUs, running specific applications has already been measured through-beam experiments in previous work (BAUMANN, 2005; ZIEGLER; PUCHNER, 2010; SEIFERT; ZHU; MASSENGILL, 2002; NGUYEN et al., 2005; CONSTANTINESCU, 2002; OLIVEIRA et al., 2017b; SULLIVAN et al., 2021). By exposing the device running a code to an accelerated particle beam, it is possible to induce transient faults in the hardware and, counting the manifestations of errors at the output, measure the realistic error rate. While providing the realistic error rate, beam experiments jointly consider all the factors that influence the device error rate, impeding the distinction of each factor's contribution and making it challenging to identify the most vulnerable parts of the system. Additionally, beam experiments are performed on a silicon prototype or the final Commercial Of The Shelf (COTS) prod-

uct, making any modification to the project, including improving the device reliability, extremely expensive.

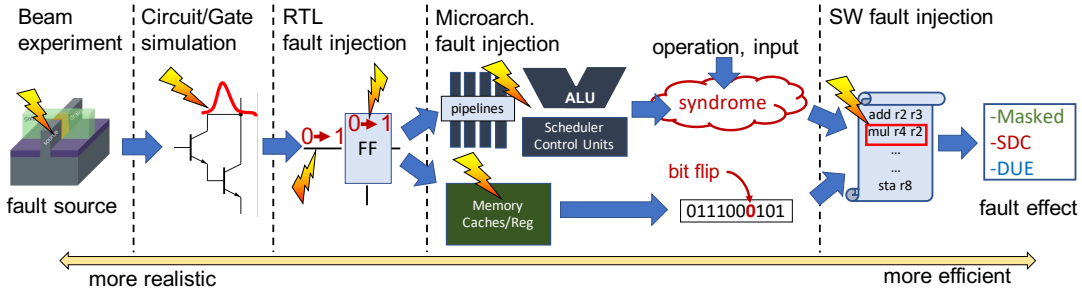
Fault simulation, contrarily, is used to understand the probability for a fault to propagate, generating an error. Figure 2.1 illustrates a view of the available fault injection methodologies across different levels of abstractions. Each evaluation methodology has some benefits and limitations, which are summarized next:

- **Circuit or Gate Level Simulations** induce analog current spikes or digital faults in the lowest abstraction level that still allows to track fault propagation (not available with beam tests). There are two main issues with the level of details required to perform this analysis on GPUs: (1) a circuit, or gate-level description of GPUs is not publicly available and, even if it was, (2) the time required to evaluate the whole circuit would definitely be excessive (the characterization of a small circuit takes weeks (Kochte et al., 2010)).
- **Register-Transfer Level (RTL) fault injection** accesses all resources (flip flops and signals) and provides a more realistic fault model, given the proximity of the RTL description with the actual implementation of the final hardware (Ejlali et al., 2003; SUBASI et al., 2018; CONDIA et al., 2020). However, the time required to inject a statistically significant number of faults makes RTL injections impractical. The massive amount of modules and units in a GPU and the complexity of modern HPC and safety-critical applications exacerbate the time needed to have an exhaustive RTL fault injection (hundreds of hours for small codes), making it unfeasible.
- **Micro-architecture fault injection** provides a higher fault coverage than software fault injection as faults can, in principle, be injected in most modules. A preliminary work, based on Multi2Sim (Ubal et al., 2012) GPU description, presented micro-architectural fault injection data. However, most micro-architectural analysis is limited to just memories (KALIORAKIS et al., 2015; TSELONIS; GIZOPOULOS, 2016; Vallero; Gizopoulos; Di Carlo, 2017; YANG et al., 2021b). One of the issues of micro-architectural fault injection in GPUs is that the description of some modules (including the scheduler and pipelines) is behavioral. Thus, their implementation is not necessarily similar to the realistic one.
- **Software fault injection** is performed at the highest level of abstraction and, on GPUs, it was proved efficient and helpful in identifying those instructions or code portions that, once corrupted, are more likely to affect computation (Wei et al.,

2014; FANG et al., 2014; Hari et al., 2017; YANG et al., 2021a; TSAI et al., 2021). However, the analysis is limited as faults can be injected only on that subset of visible resources to the programmer. Unfortunately, critical resources for highly parallel devices (i.e., hardware scheduler, threads control units, etc.) are not accessible to the programmer and cannot be characterized via high-level fault injection.

- **Hybrid or combined fault injections** at different levels of abstraction have been adopted to increase the efficiency of the reliability evaluation without jeopardizing its accuracy. Some studies have proposed to use a detailed RTL fault injection in specific portions of the circuit and a fast fault simulation in others (Ejlali et al., 2003; SARTOR; BECKER; BECK, 2019). Recent works combined an extremely detailed gate-level fault injection in tandem with a faster (but still impracticable for complex devices) RTL evaluation (Kochte et al., 2010; NIMARA et al., 2016). Cho et al. used high-level simulation (not using real hardware), triggering a RTL model when the fault needs to be injected (Cho et al., 2015). Subasi et al. focus on RTL injection to provide a more detailed fault model but limited to embedded processors ALU (SUBASI et al., 2018). While this work takes inspiration from the two-level fault injection concept, none of these works address GPUs (nor parallel devices in general), but mainly embedded processors, with a completely different complexity scenario. Additionally, none of the previous works provide, as this work does, a fault model database that could be used in future evaluations (SANTOS et al., 2021).

Figure 2.1: Abstract view of fault injection and propagation. Reliability evaluations closer to the fault source (i.e., the silicon implementation) are more realistic by extremely costly. Evaluations closer to the fault manifestation at the output are more efficient but risk to be unrealistic. Single or double bit-flips injections at software level, for instance, accurately simulate memory faults, only. Faults in other resources have a syndrome that depends on operation and input value.



In an effort to improve the analysis made using fault simulation, the concept of SDC probability is proposed by some works (YIM et al., 2011; FENG et al., 2010; LI;

PATTABIRAMAN, 2018; PALAZZI et al., 2019; ANWER et al., 2020). The SDC probability is the AVF normalized by the instruction probability to be sampled from the code's instructions. The SDC probability is a metric that connects the AVF (fault propagation) and the *instruction profiling* of the application to determine the dependence of the AVF on the instruction distribution. However, the SDC probability still lacks information on the hardware sensitivity, which still presents as a model limitation.

Recent studies (Chatzidimitriou et al., 2019; Serrano-Cases et al., 2020) tried to predict application SDC rate using micro-architectural fault injection on ARM CPUs. For GPUs, only one study (HARI et al., 2020) attempted to predict the error rate at low- and application-level. The low-level implementation considered beam experiments, and application-level analysis employed fault injection. The results show that the SDC prediction is plausible. However, the paper did not provide insights into the impact of hidden GPU resources (parallelism management) on the SDC rate or identifying the code/architecture characteristics/metrics that significantly impact GPUs. It focussed on only one GPU architecture with a single ECC setting and did not study sensitivity to compiler versions. This work is the first to demonstrate that analyzing multiple GPU architectures and compiler versions are crucial for application failure rate analysis, and investigating with both the ECC models reveals new insights.

2.3 Fault tolerance in artificial neural networks

Prior works, from the late 1990's, have shown that ANNs are highly tolerant to transient faults (ALIPPI; PIURI; SAMI, 1995; BETTOLA; PIURI, 1998; PIURI, 2001; DISTANTE; PIURI, 1991; PHATAK; KOREN, 1995; DISTANTE et al., 1991; NETI; SCHNEIDER; YOUNG, 1992). Software fault injection was performed at different levels of abstraction to understand the vulnerability of various networks. Unfortunately, GPUs tend to propagate a single fault to multiple output elements, a behavior not studied in previous research. Additionally, as the structure and topology of neural networks have significantly evolved in recent years, most prior results cannot be easily extended to today's deep learning complex frameworks. Specifically, neural networks for object detection have the peculiarity of using convolution to extract features from the data (LECUN et al., 1998), a class of computations where reliability has not been deeply studied to date. A study on CNN reliability, executed on dedicated accelerators, is presented by Li et al. (LI et al., 2017), showing that each layer and flip-flop has a different injec-

tion sensitivity. The authors also identify which flip-flops to harden to increase the CNN reliability.

This work significantly advances the knowledge on CNNs reliability by considering realistic error models (provided by beam experiments) and understanding the propagation of errors injected in memory elements and computing resources. This thesis considers and compares three commercially available frameworks and three different GPU architectures. Finally, hardening strategies proposed in this work do not require costly hardware modifications and are experimentally proven to be very useful and efficient.

Given the results of previous studies, it is possible to expect that the fault-tolerance of neural networks will be dominated by subsequent filtering operations. This was suggested through the study of the mathematical algorithms present in general neural paradigms, and by evaluating specific implementations (DISTANTE; PIURI, 1991; DISTANTE et al., 1991; NETI; SCHNEIDER; YOUNG, 1992; ALIPPI; PIURI; SAMI, 1995; PIURI, 2001).

Prior work has also considered how to improve the reliability of neural networks. Most of the available solutions rely on partial or full duplication (or even triplication) of operations (PHATAK; KOREN, 1995). Some solutions require specific hardware modifications (BETTOLA; PIURI, 1998; LI et al., 2017). These approaches are less than ideal for real-time object detection, a task commonly performed in automotive applications, due to processing overhead and added costs.

Some studies have used fault injection during network training (SEQUIN; CLAY, 1990; BOLT, 1992; KULAKOV; ZWOLINSKI; REEVE, 2015). Once the neuron values are adjusted to support errors, the network can maintain the correct classification, even if a fault happens. The main limitation of this approach is that it is hard to inject a representative set of faults. The fault model set, unfortunately, is determined by the network dimension. For deep CNNs, injecting a statistically significant set of fault models is a considerable amount of work, infeasible for most frameworks.

2.3.1 The performance of CNNs on GPUs

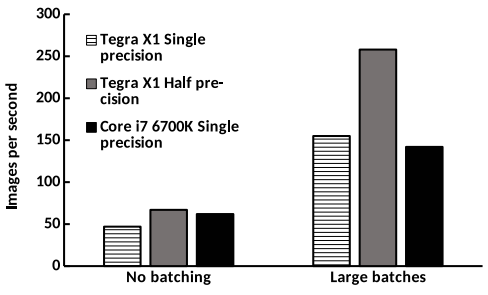
In 1989, a new algorithm in machine learning proposed by LeCun *et. Al* placed a breakthrough for image recognition in AI (LeCun et al., 1989). The authors proposed a machine learning method called CNN to detect handwritten digits, which achieved more than 98% of accuracy (LECUN et al., 1998). Since then, machine learning methods

have increased their accuracy and complexity. However, with deeper neural networks and the increasing number of classification method parameters, machine learning became costly in performance for that time processors. Only in 2009, the machine learning methods were implemented using GPUs (RAINA; MADHAVAN; NG, 2009) with a single NVIDIA GPU Raina *et. Al* achieved more than $72\times$ speedup over a dual-core CPU. Ever since, the machine learning algorithms get lots of improvement on NVIDIA GPUs in terms of performance and power consumption (CHETLUR et al., 2014; KRIZHEVSKY, 2014). Nowadays, the standard COTS platform to train and execute machine learning algorithms is through NVIDIA GPUs (GAWANDE et al., 2018).

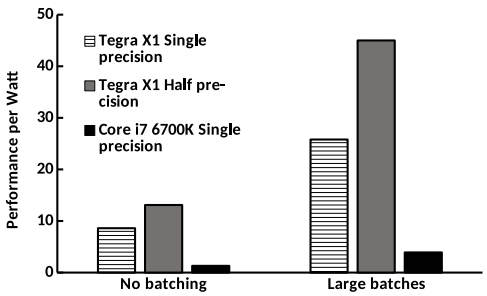
As an example of how NVIDIA overpower the rivals in the deep learning market, figure 2.2 shows a result adapted from (NVIDIA, 2015b). Two platforms are evaluated, an embedded NVIDIA GPU, Tegra X1, and a general-purpose CPU, Intel i7 6700K, running AlexNet Convolutional Neural Network object classification. For Tegra X1, the results are shown for two float precisions, single and half (i.e., FP32 and FP16), without losing classification accuracy. The batch sizes are plotted for *No batching* (batch size equals to 1) and *Large batches* (batch size equals to 128 for Tegra X1 and 48 for Intel i7). The batch size defines the number of samples seen before updating the neural network model in the training process. For images per second result, Intel i7 is only better than Tegra X1 using a batch size of 1, while for large batches, Tegra X1 outperforms Intel i7 for single and half precision. If the power consumption is taken into the analysis, Tegra X1 half precision exceeds Intel i7 up to $6.6\times$ without batching and $11.5\times$ with large batches. Today, NVIDIA GPUs are the most used commercial device for Deep Learning researchers. Consequently, this work evaluated the NVIDIA GPU reliability running three of the most accurate CNN when this work is done, YOLO, Faster R-CNN, and Resnet.

Figure 2.2: The images per second and performance per Watt achieved for NVIDIA Tegra X1 and Intel i7 6700K. Adapted from (NVIDIA, 2015b)

(a) Images per second processed in the inference process.



(b) The ratio between performance and power consumption



Convolutional Neural Networks (CNNs) are one of the most efficient ways to perform image classification (CIREGAN; MEIER; SCHMIDHUBER, 2012), segmentation (CIRESAN et al., 2012), and object detection (REDMON et al., 2015; Mao et al., 2018; REN et al., 2015; GKIOXARI; MALIK; JOHNSON, 2019). Prior work has adopted CNNs for real-time object detection,showing excellent results (ANGELOVA et al., 2015; RIBEIRO et al., 2016; REDMON; FARHADI, 2018). Figure 2.3 shows an example of how CNNs works, most of them are composed of two stages: Feature extraction and classification (ALOM et al., 2019).

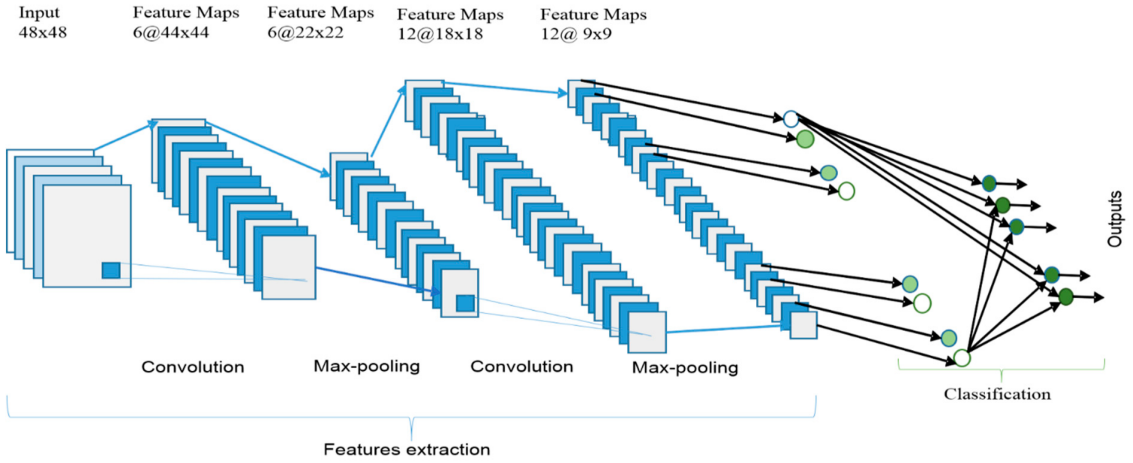
One of the basic steps when using CNNs for object detection is convolution. A kernel filter is convolved with a matrix to extract specific features of the image. The kernel filter slides over the input matrix, multiplying and accumulating products at every position of the input with every position of the kernel. This process can be mapped to a matrix multiplication operation. Each block is reorganized as a row of matrix A, and the filter kernel is replicated as columns of a matrix B. Convolution is then computed as AxB. Other methods consider convolutional algorithms or Fast Fourier Transforms. However, they are less efficient than GEMM-based convolutions (CHETLUR et al., 2014; MATHIEU; HENAFF; LECUN, 2013). As GPUs are highly efficient for accelerating matrix multiplication, they are an excellent target for CNNs. Specific processors can be designed for CNN, achieving significant performance and low power consumption (LUO et al., 2017; JOUPPI et al., 2017). However, the design of dedicated processors is exceptionally costly and less than ideal for embedded applications.

A CNN has several layers that perform convolution on a raw image or feature map (i.e., the output of an upstream convolutional layer). This work considers three modern frameworks: i.) You Only Look Once (YOLO) versions, one, two and three (REDMON et al., 2015; REDMON; FARHADI, 2016; REDMON; FARHADI, 2018), ii.) a 2) a Faster Region-based Convolution Neural Network (Faster R-CNN) (REN et al., 2015), and iii.) a Residual Network (Resnet) (HE et al., 2016). **YOLO** is based on *Darknet*, which is an open-source CNN used for object classification and detection written in C and Compute Unified Device Architecture (CUDA) (REDMON et al., 2015). **Faster R-CNN** is written in C++ and Python, based on Caffe's (JIA et al., 2014) deep learning framework. **ResNet** is a CNN for classification based on the Torch7 deep learning framework (COLLOBERT; KAVUKCUOGLU; FARABET, 2011). ResNet only performs object classification, while YOLO and Faster R-CNN also provide detection (i.e., bounding boxes that highlight the classified objects in the frame). However, the networks pipeline and operation performed

are similar once all the CNNs share the same structure. There are other differences between YOLO, Faster R-CNN, and ResNet that are likely to impact their reliability as demonstrated in (Santos et al., 2019).

The GPU microarchitecture can propagate a single fault to affect several output elements, and this behavior significantly impacts CNN reliability. This work demonstrates in Chapter 4.1 that, unfortunately, ECC is insufficient to ensure high reliability in CNNs, as it does not reduce the number of critical errors. Chapter 5 presents more efficient hardening techniques that can improve the fault-tolerance on CNNs.

Figure 2.3: An example of a CNN's architecture, most of CNNs are split into two stages feature extraction and classification. Figure extracted from (ALOM et al., 2019).



2.4 Mixed precision and error criticality

In recent years major hardware vendors are making available devices that enable the execution of operations in mixed-precision. The market needs for more efficient architectures and applications in terms of execution time and power consumption pushed the adoption of approximate computing in a growing number of applications. Consequently, the demand for mixed-precision platforms increased since they are extremely interesting for the HPC and safety-critical markets.

Some applications do not require the full precision provided by IEEE754's double or single precisions (STRZODKA; GODDEKE, 2006). As the float point functional units area grows quadratically as the precision increase, using double and single-precision data even if lower precision is sufficient would add unnecessary overhead to the application (GODDEKE; STRZODKA; TUREK, 2007). Previous studies have shown significant performance improvement using mixed precision on HPC (GODDEKE; STR-

ZODKA; TUREK, 2007; CLARK et al., 2010), Deep Learning (COURBARIAUX; BENGIO; DAVID, 2014; VENKATESH; NURVITADHI; MARR, 2017), physics simulation (GRAND; GöTZ; WALKER, 2013), and approximate computing (HO et al., 2017; MICIKEVICIUS et al., 2017).

Mixed-precision architectures can be used in low power devices (TONG; NAGLE; RUTENBAR, 2000), FPGAs (MINHAS; BAYLISS; CONSTANTINIDES, 2014), GPUs (NVIDIA, 2017; MINHAS; BAYLISS; CONSTANTINIDES, 2014), general purpose CPUs (LOMONT, 2011), etc. The architecture is said to be mixed precision if it has support for at least two of the five float point arithmetics defined by the IEEE 754 (16 , 32, 64, 128, and 256 bits) (IEEE, 2008). Modern devices such as NVIDIA GPUs, Intel accelerators, and FPGAs, have support for half, single, and double precisions (MINHAS; BAYLISS; CONSTANTINIDES, 2014; NVIDIA, 2017; INTEL, 2016). Nevertheless, other architectures support different precisions, such as 8 bits operations (GUPTA et al., 2015).

Many fields can benefit from float point mixed-precision arithmetic, such as Neural Networks, image processing, HPC, etc (CLARK et al., 2010; HWANG; SUNG, 2014; COURBARIAUX; BENGIO; DAVID, 2014). Recently, efforts have been to improve the performance of mixed-precision applications on modern devices. Most recent research does not cover how mixed-precision arithmetics impact the device or application reliability. This work also covers different precision reliability on Volta architecture.

2.4.1 SDC criticality

SDCs are not always critical, as some output errors can be tolerated. For instance, if the corruption affects only the least significant positions of the mantissa of a float point number, the results could still be inside float point operations' inherent variance. Some physical simulations accept as correct values in a range that can be as high as 4% for wave simulations (PUENTE et al., 2014; OLIVEIRA et al., 2017b). Additionally, approximate computing is gaining interest in various HPC applications (BREUER, 2005; NVIDIA, 2017). If the presence of SDCs does not impact the application output, SDCs could be considered tolerable. Moreover, the output of most neural-networks-based object detection frameworks is a vector of *tensors* containing the probability of eligible objects. Objects identified with a sufficiently high probability are classified and eventually detected. Transient faults that modify the probability without altering an object's rank or

change the coordinates of low-probability objects are not considered critical. Intuitively, the reduction of data and operation precision is likely to increase the criticality of transient errors, as demonstrated in a recent study (SANTOS et al., 2019).

3 METRICS AND EVALUATION METHODOLOGY

This chapter describes the devices and codes characterized, the metrics adopted for the reliability evaluation, and how they are measured for GPUs.

3.1 Devices

Devices: For this work, two NVIDIA GPU micro-architectures, Kepler (Tesla K40c) and Volta (Titan V and Tesla V100), are considered. NVIDIA Kepler micro-architecture, designed in 2013, was one of the most popular GPUs for HPC. Kepler introduces significant changes compared to previous micro-architectures, such as introducing dynamic kernel parallelism and launching concurrent kernels in the same GPU. The tested NVIDIA K40 (**Kepler**) is built with the $Kepler$ ISA and fabricated in a $28nm$ TSMC standard CMOS technology. This model has 2880 CUDA cores divided in 15 Streaming Multiprocessors (SMs). Each K40 SM has 64K registers, 64KB of L1/shared memory, 1.5MB of L2 cache, and 6GB GDDR5 memory. Single Error Correction Double Error Detection (SECDED) Error Correcting Code (ECC) protects the register file, shared memory, and caches while read-only data cache is parity protected. K40 is used with both the ECC enabled and disabled to evaluate the register file error rate's impact on the prediction.

Titan V and Tesla V100 (**Volta**) are designed with the Volta micro-architecture and built with TSMC FinFET 12nm. Volta GPUs feature hardware acceleration for three IEEE754 float point precisions: double, float, and half. Each of the 80 Volta SMs has 64 FP32 cores, 64 INT32 cores, 32 FP64 cores (HO; WONG, 2017; NVIDIA, 2018). Volta also includes eight *tensor cores*, i.e., specific hardware that performs the Matrix Multiplication and Accumulate (MMA) operation on $4x4$ matrices. While the V100 operates at 1246MHz as a base clock, Titan V operates at 1200MHz. Tesla V100 has 16GB of HBM2 RAM, and Titan V has 12GB of HBM2 RAM. Finally, only the V100 has SECDED ECC on the main memories.

It is worth noting this evaluation considers errors occurring in the GPU core, not in the main memory. That is, injecting faults in the main memory is not studied. For Kepler, the beam spot is sufficiently small (2cm of diameter) to not hit the onboard DDR when ECC is disabled. For Volta, as HBM2 memories are on top of the chip when ECC is disabled, all the global memory accesses are made through Triple Modular Redundancy

(TMR). The choice of not considering main memory reliability that has already been extensively studied (Saeng-Hwan Kim et al., 2007; Chen et al., 2016) is dictated by the fact that bitflips in it would mask effects in the GPU core.

3.2 Tested Codes

Twelve representative codes listed in Table 3.1 (Kepler) and Table 3.2 (Volta) are chosen for this work. A set of codes that comes from broad domains, from HPC to deep learning. As a side contribution, this work measures which are the codes that are more vulnerable or reliable. Additionally, all the codes, data, and tools used in this work are available on GitHub to allow reproducibility. The radiation experiments setup tools and codes are available in (SANTOS et al., 2014). The data extracted on CNNs and discussed in Chapter 4.1 is available in (SANTOS et al., 2018). FIT prediction, discussed in Chapter 6, data is available in (SANTOS et al., 2021). Finally, the data for RTL/Software fault injections and the new error model discussed on Chapter 7 are available at (SANTOS et al., 2021).

A detailed description of the chosen benchmarks are as follows:

1. **Connected Component Labeling (CCL)** is a labeling algorithm that is commonly used for object detection. CCL scans the image in parallel in a row-wise fashion to find contiguous pixels using child threads through dynamic parallelism in the same row that belong to the same label. Then, CCL merges the components previously found and updates the respective labels using child threads through dynamic parallelism (UKIDAVE et al., 2015).
2. **Breadth-first search (BFS)** is a search in graphs algorithm that is widely used in GPS Navigation Systems. BFS kernel searches an undirected and unweighted graph to find the minimum edges reaching all vertexes of the graph. The chosen BFS algorithm uses each thread to represent each vertex of the graph them store the results of visited vertices in an array of indexes, and another array to keep the costs (HARISH; NARAYANAN, 2007). BFS is a well-known example of a not well-suitable algorithm for GPUs since each thread has a very irregular memory access pattern.
3. **Lava** simulates particle interactions in a large 3D space. It calculates the particle's potential and relocation in a large 3D space due to mutual forces between

them (CHE et al., 2009). Lava kernel computation is mostly dot-products with float point data. Each thread performs one particle's interactions with all particles in neighboring boxes. Lava is representative of Multi-physics Particle Dynamics Code applications which solve a series of ordinary differential equations using Finite Difference Methods (SZAFARYN et al., 2010).

4. **Hotspot** estimates a processor temperature using an architectural floor plan and simulated power measurements (CHE et al., 2009). The computation is represented by a grid, where each output cell represents the average temperature value of the corresponding area of the chip. Hotspot is a 2D stencil highly parallelizable code that achieves a high occupancy level. Hotspot is representative of stencil solvers, used in applications from geophysics to molecular dynamics (PUENTE et al., 2014; BIYIKLI; YANG; TO, 2014) and intensely studied by the community (NGUYEN et al., 2010).
5. **Gaussian** elimination is a linear algebra algorithm that solves a system of equations and computes the result for all of the variables in a linear system, row-by-row (CHE et al., 2009).
6. **LU Decomposition (LUD)** is a linear algebra method that calculates solutions for a square system of linear equations. LUD kernel factors a matrix as the product of a lower triangular matrix and an upper triangular matrix. LUD is representative of highly CPU-bound codes (CHE et al., 2009). Additionally, this application has many row-wise/column-wise interdependencies, significant inter-thread sharing, and row/column dependencies.
7. **Needleman–Wunsch (NW)** is a bioinformatics algorithm that aligns DNA sequences. Possible pairs of sequences are organized in a 2D integer matrix, then NW deduces the optimal alignment by finding the maximum path in the matrix. NW uses smaller problems to solve the original problem (CHE et al., 2009), which leads the NW kernel to have many control flow instructions.
8. **Matrix Multiplication (MxM)** is an essential tool for HPC and is also one reason for GPUs' success in CNNs training and execution. For this reason, this study pays particular attention to this algorithm and test both the naive version (*MxM*) and the optimized version that digest data in the most suitable way for GPUs as General Matrix Multiplication (*GEMM*). The GEMM version considered is part of the NVIDIA CUBLAS libraries, which is to be highly efficient, has a dedicated kernel code for each group of size, precision, and device configuration.

9. **Mergesort** is a divide and conquer sorting algorithm of $O(n \log n)$ complexity. CUDA toolkit Mergesort sorts divided arrays with Bitonic sorting algorithm. Then the final step organizes the data segments previously ordered. CUDA toolkit Mergesort creates an array of indexes for each input element. If an element on input data is changed, its index is moved along (NVIDIA, 2015a). Sorting algorithms have a high memory device utilization as also lots of control flow instructions.
10. **Quicksort** is one of the most known sorting algorithms. CUDA toolkit implementation chooses a random pivot between the two split segments, then each element in the segment vector is sorted relative to the pivot. CUDA toolkit Quicksort uses Cuda Dynamic Parallelism, which allows the main kernel to create child kernels reducing the host-GPU communications overhead (NVIDIA, 2015a).
11. **CFD Solver** is an unstructured grid solver for Computational Fluid Dynamics (CFD) (CHE et al., 2009). CFD is a type of numerical algorithms commonly used to solve problems that involve fluids. The algorithm used in this work solves an Euler equation for compressible flow.
12. **You Only Look Once (YOLO)** is an open-source high accurate real-time CNN-based object detection framework for automotive applications (REDMON; FARHADI, 2018). This work uses three versions of YOLO, version one, two, and three. The newer version of YOLO is always different from the former one. For instance, they all differ in the layer numbers, YOLOv1 has 31 layers, YOLOv2 has 32 layers, and YOLOv3 has 106 layers (REDMON et al., 2015; REDMON; FARHADI, 2016; REDMON; FARHADI, 2018). The differences between the versions are mainly upgrades in the original algorithms that improved the object detection's accuracy and performance. In this work, all YOLO versions are tested with Caltech dataset (DOLLÁR et al., 2012). Exceptionally, YOLOv3 is executed in single and half precision on NVIDIA Volta. It is worth noting that, to focus only on mixed-precision operation effects on CNN reliability, YOLOv3 is not retrained. The weights of the single-precision version are converted to half-precision precision.

This work also discusses if the code reliability characteristic is due to the resources' sensitivity, the number of resources used for computation, or the probability of propagating faults (AVF). Tables 3.1 and 3.2 also list the amount of shared memory and the average number of registers used for computation together with the execution time. They also include the IPC and occupancy metrics used to consider the GPU parallel

management in the FIT rate prediction (details in Section 6.2).

Table 3.1: Codes used for reliability evaluation on Kepler.

Benchmark	Precision	SHARED (KB)	RF (AVG)	Execution Time [s]	IPC	Occupancy
CCL	Integer	0.12	33.1	1.19	0.14	0.11
BFS	Integer	0	20.3	0.27	1.22	0.81
Lava	Float	7.04	37	2.43	4.12	0.57
Hotspot	Float	3	23	1.7	3.89	0.94
Gaussian	Float	0	13.9	0.27	0.51	0.34
LUD	Float	8.61	26.1	1.44	0.58	0.37
NW	Integer	8.26	32	0.33	0.2	0.08
MXM	Float	8	25	0.38	1.5	1
GEMM	Float	31.01	247.8	0.07	4.94	0.19
Mergesort	Integer	2.51	15.4	0.75	2.11	0.97
Quicksort	Integer	0.32	26.1	1.81	1.97	0.96
Yolov2	Float	8.02	96.2	0.1	2.84	0.59
Yolov3	Float	9.07	99.6	0.44	3.11	0.65
Faster R-CNN	Float	5.4	67.7	0.09	2.40	0.58
Resnet	Float	1.9	63.9	0.03	1.54	0.49

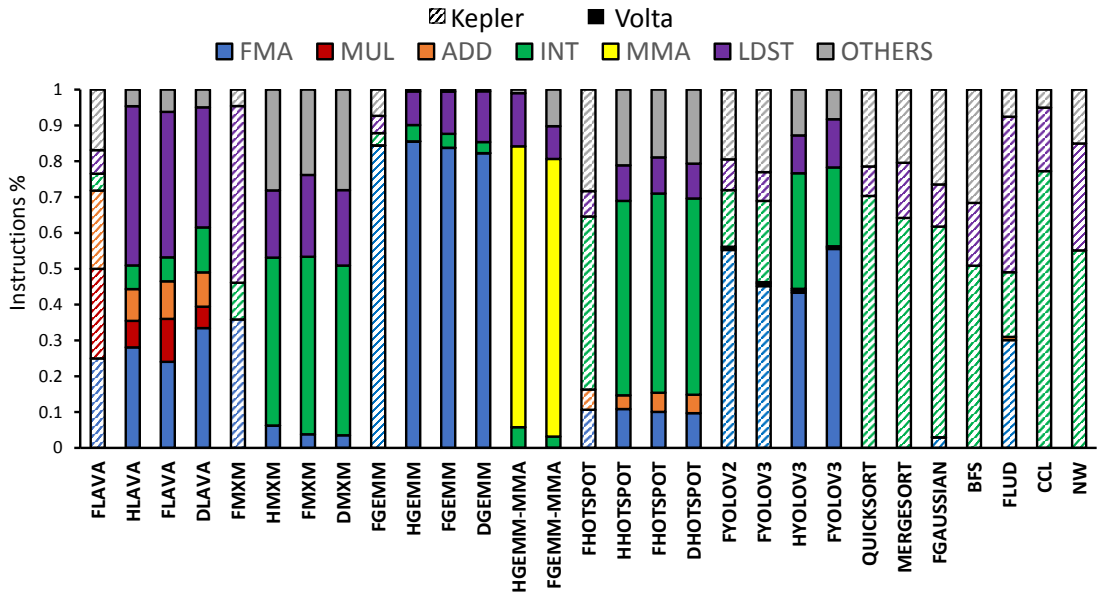
Table 3.2: Codes used for reliability evaluation on Volta.

Benchmark	Precision	SHARED (KB)	RF (AVG)	Execution Time	IPC	Occupancy
Lava	Half	8	255	0.31	0.26	0.1
	Float	8	255	0.56	0.12	0.1
	Double	16	254	1.07	0.07	0.1
Hotspot	Half	16	26	0.44	0.48	0.94
	Float	32	27	0.65	0.32	0.95
	Double	64	30	1.3	0.18	0.96
MXM	Half	0	27	1.17	2.84	1
	Float	0	25	1.97	2.62	1
	Double	0	29	2.4	2.3	1
GEMM	Half	64	127	0.4	2.34	0.25
	Float	64	134	0.77	2.36	0.13
	Double	64	234	1.3	1.22	0.13
Yolov3	Half	21.52	69.0	0.4	1.23	0.7
	Float	34.18	122	0.46	1.74	0.7

The dynamic instructions are profiled for each code listed in Table 3.1 and 3.2 using NVPROF and NSIGHT-COMPUTE, respectively. Profiling the codes provide insights into the micro-instructions that significantly contribute to the benchmark execution. Figure 3.1 shows, in percentage, the micro-instructions that compose each code. Each float point code has the precision made explicit in the first letter of its name, D for double-precision (64 bits), F for single-precision (32 bits), and H for half-precision (16 bits). That is, *HHOTSPOT* is Hotspot executed in half-precision, *DGEMM* is GEMM executed in double-precision. Codes that only operate with INT32 data do not have their names modified.

The instructions in the kernel profile are divided into two classes: (1) the arithmetic instructions commonly used on the benchmarks (i.e., FMA, MUL, ADD, INT, MMA), and the primary instruction used for data movement (LDST). (2) "OTHERS" are the ones that have a minor contribution to the final benchmark. (i.e., transcendental functions, branch, inter-thread communication, thread barrier, NOP, and atomic directives). The FIT rate of only the former set of instructions is measured through beam experiments, as they are the most likely to be corrupted and the most common in a wide range of codes. Testing all the micro-instructions would be unfeasible due to beam time restrictions (more than 20 different micro-instructions types in the NVIDIA ISA). Nevertheless, as further demonstrated in Chapter 6, even considering a (large but not exhaustive) subset of instructions allows a reasonable estimation of the code's FIT rates. The information provided by the profile tools is required to map how the application uses the GPU's resources. Thus, a detailed profile is crucial for a correct FIT prediction. Figure 3.1 shows that the arithmetic instructions dominate the composition of chosen benchmarks, information that will be useful later in this work.

Figure 3.1: Instruction type per code for Kepler and Volta.



3.3 Synthetic Micro Benchmarks implementation

This section describes the implementation of the microbenchmarks used in this work. Microbenchmarks are a set of synthetic applications designed to stress specific

components of the GPU architecture. Using microbenchmarks, it is possible to evaluate the error rate of a particular resource. Table 3.3 shows the main characteristics of each microbenchmark. The microbenchmarks are tuned to stress mostly the unit that is being evaluated. That is, in each microbenchmark composition, 99% of the instructions is the one that is being assessed. For example, 99% of the FADD microbenchmark is FP32 ADD instructions. In sequence, each microbenchmark is explained in detail.

RF and SHARED micro-benchmarks measure the FIT rate of the Register File and Shared Memory, respectively (L1 cache has the same technology as Shared Memory). Each thread per SM that writes a known pattern in all accessible registers or shared memory (255 registers per thread and 48KB of static shared memory among all SM threads) and, after a pre-defined time, read-backs the values counting bit-flips. The micro-benchmark instantiates the lowest possible number of threads to reduce the probability of having errors in other resources besides the memories. The time between a write and a read should be long enough to ensure that the setup/read-back time is negligible and short enough to prevent more than one neutron from generating faults. This latter constrain is necessary to detect eventual Multiple Bit Upsets (MBUs, more than one bit corrupted in a single word). The exposure time is heuristically set to $1s$. It is anticipable that, for RF, the MBU rate is lower than 2% and, for SHARED, is lower than 0.9%.

LDST performs a sequence of memory movements on the global memory (Load followed by Store) with ECC enabled. The LDST kernel reads a memory region from global memory that contains a unique pattern and stores it in another global memory location. Each kernel consists of 4M threads, each performing 2^{10} memory movements. In total, this micro-benchmark allocates 2GB of memory. The host CPU setup compares the expected pattern on the output memory and counts the number of corruptions. CPU verification time is not considered for FIT calculation.

A group of specific micro-benchmarks was created to evaluate the arithmetic functional units. Each thread in **FMA** (Fused Multiply and Add), **ADD** (Addition), **MUL** (Multiplication), and **MAD** (Integer Multiply and Accumulate) micro-benchmarks executes 10^8 operations, while **MMA** performs 10^7 16x16 (the matrix is sliced into $4x4$ smaller matrices, see Table 3.3) matrix multiplications (with FP16 on HMMA or FP32 casted to FP16 for FMMA). A lower of operations MMA was chosen to keep the exposure time, and so the statistic, similar to the other micro-benchmarks. The inputs are pre-defined and have been randomly generated off-line, ensuring to avoid overflow. The integer and float versions of the micro-benchmarks have been tested on the Kepler and

Table 3.3: Summarized micro-benchmarks details. The information present on the table for the latency is extracted from (JIA et al., 2018; ARAFA et al., 2019). On Volta, the FP16 corresponds to two operations performed in the same hardware (x2). Kepler supports FP64 instruction. However, as Kepler FP64 instructions are not used in this work, they are not listed.

		Instruction latency (cycles)				Operand description
		INT32	**FP16**	**FP32**	**FP64**	
Addition (ADD)	Kepler	9	–	9	–	2 input registers + 1 output register
	Volta	4	6 (x2)	4	8	
Multiply (MUL)	Kepler	9	–	9	–	2 input registers + 1 output register
	Volta	4	6 (x2)	4	8	
Multiply and Add (FMA/MAD)	Kepler	9	–	9	–	3 input registers + 1 output register
	Volta	4	6 (x2)	4	8	
Shared memory (SHARED)	Kepler	26 - 55	–			Thread Block load followed by store
	Volta	18 - 49				
Matrix Multiply and Accumulate (MMA)	Volta	–	4x4 matrix 1 cycle	–	–	A warp of threads perform 16x16 matrix multiply
Load Followed by Store (LDST)	Kepler	331 - 382	–	–	–	4 Bytes load followed by 4 Bytes store on global memory
Register File (RF)	Kepler and Volta	–				255 loads followed by 255 stores

the integer (INT32), double (FP64), float (FP32), and half (FP16) versions of the micro-benchmarks on the Volta. The number of instantiated threads is tuned to occupy all the GPU's available functional units (3,840 threads for Kepler, 20,480 threads for Volta). The micro-benchmarks can be easily adaptable to any GPU.

3.4 Beam Experiment Setup

Beam experiments are the most effective way to measure the Failure In Time (FIT) rate of code running on a computing device. By dividing the number of observed errors by the received particles fluence η ($neutrons/cm^2$) it is possible to calculate the *cross section*:

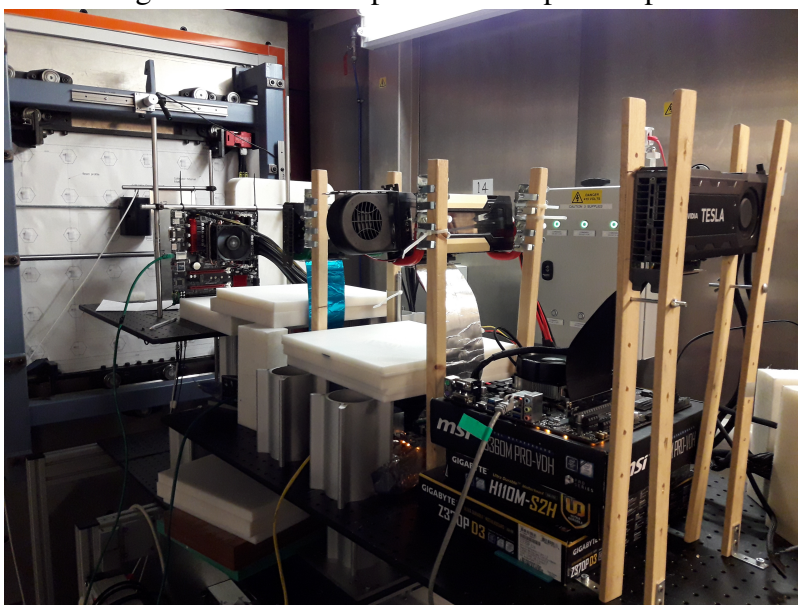
$$\sigma[cm^2] = \frac{\#errors}{\eta} \tag{3.1}$$

The fluence is obtained by multiplying the average neutron flux provided by the test facility ($neutrons/(cm^2 \cdot s)$) by the effective execution time. The cross-section (cm^2) represents the circuit area that will generate an output error if hit by a particle. The higher

the number of computation resources, the higher the cross-section, and the higher the probability for an impinging particle to generate an error.

The experiments are performed at the ChipIR facility of the Rutherford Appleton Laboratory, UK, and at the LANSCE facility of the Los Alamos National Laboratory, USA. Figure 3.2 shows the setup mounted in the ChipIR facility. Both facilities deliver a beam of neutrons with a spectrum of energies that resembles the atmospheric neutron one (CAZZANIGA; FROST, 2018), the probability of generating an error of a neutron produced in the experimental facilities is similar to a terrestrial neutron one.

Figure 3.2: Beam experiment setup at ChipIR



The available neutron flux was about $3.5 \times 10^6 n/(cm^2/s)$, ~8 orders of magnitude higher than the terrestrial flux ($13 neutrons/(cm^2 \cdot h)$ at sea level (JEDEC, 2006)). Since the terrestrial neutron flux is low, it is improbable to see more than a single corruption during program execution in a realistic application. Thus, the experiments have been carefully designed to maintain this property (observed error rates were lower than 1 error per 1,000 executions). Experimental data can then be scaled to the natural radioactive environment without introducing artifacts. At least ten codes and seven micro-benchmarks have been tested per device. Each code was tested for at least 72 effective hours (i.e., the actual time the device was running the tested code, without considering the setup, result check, initialization, and recovery from the crash time). When scaled to the natural exposure, the more than 1,224 hours of the test account for more than 13 million years.

When multiplied with the expected neutron flux at which the device will operate ($13 \times 10^9 neutrons/(cm^2 \cdot h)$ at sea level), the cross-section estimates the realistic error rate, expressed in FIT, i.e., errors per 10^9 hours of operation. Not to reveal business-

sensitive data, this work reports the FIT rate normalized by a constant factor of β, as shown in equation 3.2.

$$FIT_{norm} = \frac{\sigma \times 13 \times 10^9}{\beta} \tag{3.2}$$

It is worth noting that neither the cross-section nor the FIT rates depend on the execution time, but only on the number of resources used for computation, their sensitivity (probability for the fault to occur), and criticality (probability for the fault in the resource to affect the calculation). If the same amount of memory is exposed for a given time t or $2 \times t$ its FIT rate would not change. In fact, in $2 \times t$ it is expected twice the error and twice the neutrons (i.e., twice the fluence). Similarly, under the correct assumption that at most one fault can affect the GPU during code execution (the natural flux is very low), executing x *sequential* ADDs or $2 \times x$ sequential ADDs does not change the probability of having one ADD corrupted by neutrons. However, what can change is the probability of the error in one of the ADDs to propagate to the output of the sequence of the operations (i.e., the AVF). If the additional x ADDs are executed in *parallel* with the original sequence, the FIT rate is expected to double (same execution time, same fluence, but doubled error rate). These observations in Section 6.2 are used to account for GPU parallelism management in the FIT rate prediction based on fault injection. Code modifications that improve or reduce performances and change the number of resources or their criticality are also expected to impact the FIT rate.

3.5 Fault Simulation Frameworks

For this work, the fault injection is performed in two levels, at RTL and Software levels. Both methodologies are described in this section.

3.5.1 Software level fault injection

Fault simulation can help us understand how the fault propagates and which resource is more critical for the whole application. Fault simulation provides the Architectural Vulnerability Factor (AVF) (MUKHERJEE et al., 2003) of the hardware components where faults are injected, which expresses the probability of a fault leading to a failure. The AVF identifies which resource, once corrupted, is more likely to affect the GPU com-

putation.

SASSIFI and NVBitFI frameworks are used in this work, which can inject transient errors in the GPU's Instruction Set Architecture (ISA) visible states, general-purpose registers, and predicate registers, condition instructions, and arithmetic instructions (Hari et al., 2017; TSAI et al., 2021). SASSIFI does not support ISAs newer than Maxwell, while NVBitFI works for any NVIDIA GPU, including Volta. SASSIFI and NVBitFI are the most suitable fault simulators for this work since it is possible to instruct the kernels at the Source And Assembly (SASS) level. Other fault injectors such as GPUQin, CAROL-FI (OLIVEIRA et al., 2017a), Kayotee (JHA et al., 2019), GPGPU-SIM (FANG et al., 2014), neither allow to inject faults at the SASS level, nor they offer support for Kepler and Volta architectures. At least 4,000 single bit flips faults are simulated per code on NVBitIF, and at least 10,000 bit flips faults per application on SASSIFI (1,000 for each instruction kind), for a total of more than 50,000 faults per ISA, ensuring 95% confidence intervals to be lower than 5% (Hari et al., 2017).

SASSIFI is an older fault simulator than NVBitFI. SASSIFI has 14 injection sites for the Instruction Output Value model (IOV), while NVBitFI has only four injection sites for IOV. SASSIFI can inject faults on the output of the floating-point, integer, load, and branch instructions. SASSIFI also inject faults in instructions that write in the predicate registers, the register file, and the instruction address. In contrast, NVBitFI can inject faults only at double, float, load instructions, and instructions that write in the general-purpose registers. That is, SASSIFI is likely to be more refined than NVBitFI. The differences between the fault injectors are also expected to lead to differences in the global AVF, as shown in Chapter 4.

Unfortunately, neither SASSIFI nor NVBitFI (nor any other fault injector) supports fault injection on NVIDIA proprietary libraries such as cuDNN and CUBLAS on Kepler. However, NVBitFI can inject faults in proprietary libraries on Volta. For the NVIDIA libraries, then, the AVF measured with NVBitFI on Volta was chosen to calculate the AVF for the Kepler prediction. Using NVBitFI on Volta is a simplification. As shown in Chapter 4, on the codes that do not use proprietary libraries executed on Kepler, the AVF measured with SASSIFI is 21% smaller than NVBitFI, on average. As shown in Chapter 6, this still allows the accurate prediction of the SDC FIT rates.

3.5.2 RTL level fault injection

At Register-Transfer Level (RTL) this work uses FlexGripPlus fault injector (CONDIA et al., 2020). FlexGripPlus is an open-source VHDL-based GPU model, which implements NVIDIA Tesla micro-architecture (G80) (Lindholm et al., 2008). NVIDIA proposed tesla micro-architecture in 2006. However, FlexGripPlus's most representative modules are very similar to current commercial CUDA devices. This model can use three different configurations (8, 16, or 32) per Streaming Multiprocessor, selected before simulation or synthesis.

A custom RT-level framework (Du et al., 2019) performs the fault injection through a general controller that manages the *ModelSim* environment, which hosts FlexGripPlus. According to a faults list, the controller injects one fault (as a single transient) in the targeted GPU module. For the analysis presented in this work, errors are injected in: the warps scheduler, the pipeline registers, the Integer Functional Units (INT FUs), the Single Precision Floating Point FUs (FP32 FUs), the Special FUs (SFUs) used for transcendental functions, and the control logic (see Figure 7.1).

The RTL fault injection does not consider faults in the main memory structures (caches, register file, shared memory). Table 3.4 lists the characterized modules, their size, and the micro-instructions that use each module. Memory errors are not considered for two reasons: (1) the syndrome of faults in memory is already established (single or double bit-flips (BAUMANN, 2005)) and does not need an RT-level injection to be characterized. (2) these resources are easily protected with ECC, making their reliability evaluation less attractive. Overall, this study characterization covers 84% of the resources (flip flops) involved in the computation of the characterized micro-instructions, excluding memories (24% if considering ECC-protected memories).

Table 3.4: Evaluated modules, sizes and instructions used per module

Module	RTL Size (Flip-Flops)	Type	Instructions
FP32	4,451	Execution/Data	FADD, FMUL, FFMA
INT	1,542	Execution/Data	IADD, IMUL, IMAD
SFU	3,231	Execution/Data	FSIN, FEXP
SFU controller	190	Control	FSIN, FEXP
Scheduler controller	3,358	Control	ALL
Pipeline Registers	10,949	Control/Data	ALL

4 GPU RELIABILITY EVALUATION

This Chapter shows the error rate for Convolutional Neural Networks (CNNs) and a set of High Performance Computing (HPC) codes for two NVIDIA architectures. Up to 15 applications are evaluated, providing an overview of the reliability of the GPUs. Then, the Architecture Vulnerability Factor (AVF) is shown for all applications evaluated in the beam.

4.1 Evaluating CNNs reliability

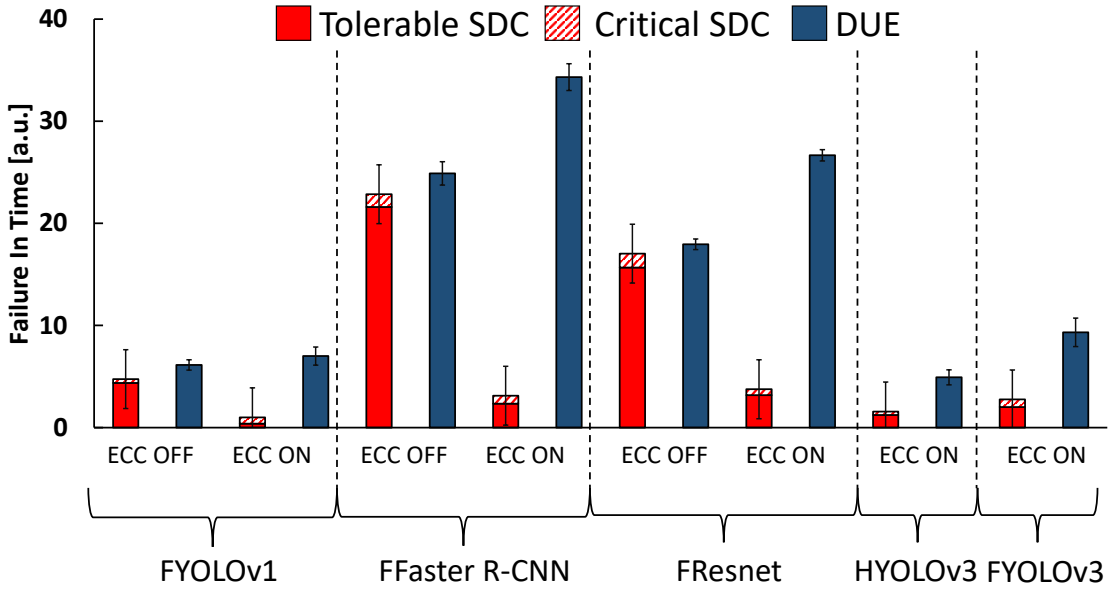
The most realist way to evaluate CNNs reliability is through beam experiments. Beam experiment setup allows evaluating the realistic radiation-induced error rate of the CNNs. Additionally, this work will draw general conclusions on the reliability of the tested neural networks.

Figure 4.1 shows the normalized FIT rate of YOLOv1, YOLOv3, Faster R-CNN, and Resnet executed on a Kepler and Volta GPUs (the later only YOLOv3 is evaluated). All reported values are relative to the SDC FIT rate for YOLOv1 ECC ON. The reported results are normalized not to reveal business-sensitive information. Even if the data is normalized, it still allows a direct comparison among configurations.

The results are presented for Silent Data Corruption (SDC) and Detected Unrecoverable Error (DUE) FIT rate. The SDC FIT is divided into Tolerable errors (i.e., that do not impact classification/detection) and Critical errors (i.e., that impact classification/detection). The output of YOLO, Faster R-CNN, and ResNet is a vector of *tensors* containing the probability of eligible objects. Objects identified with a sufficiently high probability are classified. In YOLO and Faster R-CNN, a tensor also contains the coordinates of a BB (i.e., potential object), which is then used to describe the detected object. SDCs that modify the probability such that they do not impact an object's rank, or change the coordinates of a low-probability BB for a detection framework, are not considered critical. When radiation significantly impacts classification or detection, the *Precision* and *Recall* (FAWCETT, 2006) of the corrupted output are measured. Recall is the fraction of existing objects that were detected (or classified), even in the event of a radiation-induced error ($Recall < 1$ means that some objects were not detected). Precision measures the fraction of the detections produced by the classifier that relate to an existing object ($Precision < 1$ means that some additional objects were detected). When

the Precision or Recall is different from 1, then the output is considered a critical error.

Figure 4.1: Normalized FIT rates for CNNs on Kepler and Volta architectures. FYOLOv1, FFaster R-CNN, and FResnet run on Kepler with ECC OFF and ECC ON configurations. HYOLOv3 and FYOLOv3 run only on Volta with ECC.



Experimental data is presented with a 95% confidence interval. DUEs are more probable than SDCs for all of the tested configurations, this result is in contrast to the trends observed for HPC applications, as presented in prior work (Goncalves de Oliveira et al., 2016; TIWARI et al., 2015), where the Crash rate is lower than the SDC rate. Neural networks have already been demonstrated to be intrinsically resilient to SDCs (SEQUIN; CLAY, 1990; PROTZEL; PALUMBO; ARRAS, 1993; TCHERNEV; MULVANEY; PHATAK, 2005). Even if one or more neurons are compromised, a neural network could potentially provide a correct output.

DUEs, on the contrary, have a component that depends on the GPU control logic sensitivity, and on the CPU-GPU context change, which does not benefit from the SDC tolerance of neural networks. CNNs kernels have a high level of reuse that requires several device-host synchronizations. A transient fault during those synchronizations could potentially result in a GPU crash. As a result, while a significant portion of SDCs could be masked, crashes could still undermine the device reliability. For the same reason, Faster R-CNN and Resnet, which requires a much larger number of CPU-GPU synchronizations, shows up to $5\times$ higher Crash rate than YOLO.

ECC has the drawback of increasing the DUE rate for Faster R-CNN, Resnet, and YOLO. The ECC Crash rate increases up to 30%. ECC is able to correct one-bit flip in

the memories, and when a double bit flip is detected, it throws a system exception. As CNNs use lots of memory to perform classification/detection, multiple errors on memories are expected to happen, resulting in a higher DUE rate when ECC is enabled. The experimental data attests that for YOLO, Faster R-CNN, and Resnet, radiation-induced Crashes are a severe limitation of GPU reliability. A detection system that uses GPUs in vehicles must have a watchdog or other fast Crash detection system to ensure availability and avoid missed deadlines. Even if crashes are more frequent than SDCs, they could be considered less critical since crashes could be, at least, detected (PATTABIRAMAN et al., 2006). However, the system must be able to recover before causing any harm to the environment (CANDEA; FOX, 2001).

SDC trends across GPUs are shown in Figure 4.1. The SDC rates are related to framework complexity and accuracy. The complex structure of Faster R-CNN and the deeper network of Resnet increases their SDC rate as compared to YOLO, more than $10x$ higher for Resnet and Faster R-CNN. Figure 4.1 shows that Resnet has a higher FIT rate than YOLO (i.e., v1 and v3), though the rate is similar when compared to Faster R-CNN. The higher efficiency and simpler convolution are insufficient to compensate for the higher error rate associated with a deeper neural network. The results suggest that the higher complexity of the CNN, unfortunately, the more likely that output errors will be seen. Hardening solutions, as presented later in section 5.2, are then mandatory for safety-critical applications. On the Kepler, the execution time for Faster R-CNN is $2.7x$ longer than for YOLOv1. For Faster R-CNN, due to the limited performance, data remains in caches and registers longer. This data is exposed and critical since it is likely to be used in future operations.

For all the tested CNNs, a significant number of the radiation-induced errors that propagate to the output are not considered critical, as they do not impact the detection (see Figure 4.1). The main reasons Critical SDCs occur less often versus Tolerable SDCs are: (1) some SDCs modify the object probabilities in such a way that the ranking is unaffected, (2) Not all output data will be used for detection, and (3) even if an identified object is corrupted, its shape could be sufficiently similar to the expected one, and thus, be correctly detected. Additionally, all CNN operations use floating point operands. Some of those floats represent the object coordinates, which need to be cast into integers. Errors affecting the low-order bit positions are not expected to impact detection.

For object detection, the percentage of critical SDCs is much lower for Faster R-CNN than for YOLOv1, independent of the architecture. For YOLOv1, the percentage

of critical SDCs is 8% for ECC OFF, 61% for the Kepler with ECC ON. For Faster R-CNN, the critical SDCs are 5% for ECC OFF and 25% for the Kepler with ECC ON. The differences in the critical errors percentages between the CNNs come from the detection mechanism and object representation used in Faster R-CNN. Several anchor boxes are used to define an object, and corruption in one of the coordinates of a vertex does not significantly impact the detected object shape.

Figure 4.1 shows the error rates for the Kepler with ECC enabled. ECC can reduce the error rate, but the proportion of critical errors is not reduced, which is a symptom of the poor resiliency provided by ECC. Based on the GPU-Qin analysis in (LI et al., 2016), it is known that ECC does not mask all the faults as the error in computing elements could propagate to the output. The beam tests provide, additionally, the realistic probability of experiencing an SDC when ECC is enabled or not, which is the only way to evaluate the effectiveness of ECC. A novel and worrying insight from beam tests is that ECC does not reduce (or reduce only slightly) CNNs Critical SDC rate.

The SDC rate for YOLOv1, Faster R-CNN, and Resnet on the ECC ON is 21%, 14% and 22% the SDC rate seen for ECC OFF, respectively. ECC is not as effective on these workloads as it is for other codes, mainly because neural networks are intrinsically resilient to data errors. Similar to results from previous studies (Goncalves de Oliveira et al., 2016), shows that ECC reduces the GEMM SDC rate by about one order of magnitude. ECC is less efficient in protecting a CNN, as some of the SDCs that are masked by ECC would not have affected the CNN execution. Valuable insight from the above results is that ECC does not reduce the number of critical SDCs for YOLO, Faster R-CNN or Resnet.

The portion of SDCs that impact detection is less than 8% for ECC OFF Kepler and can be more than 60% in the ECC-protected Kepler. This confirms that most of the critical SDCs come from faults outside of the main memory structures. The data strongly suggests that faults in caches or registers are not as severe for CNNs, as compared to corruptions in logic, the scheduler, or flip-flops. ECC reduces the absolute number of SDCs but has the side effect of increasing the portion of multiple errors, which are more likely to propagate through CNN layers and affect detection. Additionally, the average difference between a corrupted element's value and the expected value is higher when ECC is enabled. It is then more likely for the faults not masked by ECC to significantly impact the correctness of CNN.

On Volta GPU, YOLOv3 half precision version has lower FIT than float precision

one. The amount of per-core resources required to perform the operations depends on the chosen data precision, single precision use 152 registers per kernel, and half 126 registers. The half precision performance masks the problems of having fewer bits for data representation. Namely, more executions will finish before YOLOv3 on half precision experience a failure. Then the FIT rate decrease as the precision decrease.

The DUE rate grows as the data representation increase. A CNN has lots of PCI-express bus interactions, like CPU-GPU communication. Data movement through the PCI-express bus has been proved to generate more DUEs than SDCs (FRATIN et al., 2018). As float needs a higher amount of memory transfer, it is expected as a larger representation, the more significant is the DUE rate for YOLOv3. Comparing half and single versions of YOLOv3 the size of memory grows $2x$. As stated in previous work (FRATIN et al., 2018), the DUE FIT is also influenced by the number of control-flow, branch operations, ECC, and GPU-CPU communication. Consequently, object detection CNNs have a much higher probability of experiencing DUEs when compared to arithmetic codes.

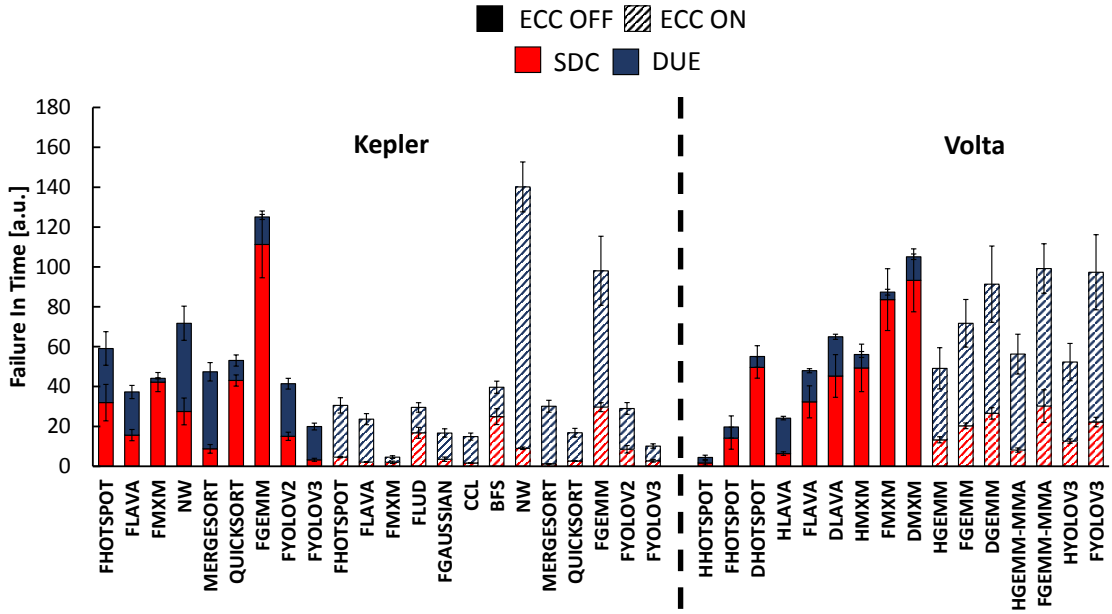
Figure 4.1 also shows the percentage of SDCs for YOLOv3 that are tolerable, that change detection. On beam experiments, the percentage of critical errors is similar comparing the two data types. It is worth noting that detection errors are less dependent on data type as coordinates are expressed integer values and, thus, are not influenced by the number of corrupted digits in the mantissa. Since on object detection, the coordinates are integers, an error that corrupts the last representation of the float could be masked by the rounding process. Section 4.1 shows that critical errors depend on which layer and the weight value of the neural network modified by the fault.

4.2 GPU FIT rate

In order to generalize the reliability analysis, this work goes beyond the study of the CNNs' reliability and evaluates a set of High Performance Computing (HPC) codes. Thus, the following section presents an extensive evaluation of 11 applications on two NVIDIA GPUs. This work first characterizes the Failure In Time (FIT) rate for the tested codes, then for these codes, the Architecture Vulnerability Factor (AVF) is presented.

Figure 4.2 shows the experimentally measured SDC and DUE normalized FIT rates for the GPUs executing the codes with ECC OFF and ON. Normalized values are reported to not reveal business-sensitive data. Values are reported with 95% confidence intervals considering a Poisson distribution.

Figure 4.2: Normalized FIT rates for Kepler and Volta. Not all configurations could be tested due to beam time limitations.



Not surprisingly, the ECC reduces the SDC FIT rate significantly. For Kepler, the average SDC FIT rate with ECC OFF is up to 21× higher than with ECC ON. For Volta, it is not possible to test the same codes with ECC enabled and disabled for beam time restrictions. The DUE FIT rate increases of up to 5× when ECC is ON. The DUE increase is exacerbated for NW and FGEMM, because of the high number of kernel calls and access to the main memory: NW is a component labeling algorithm that launches many kernels concurrently to process different parts of the input simultaneously and FGEMM uses lots of global memory and highly utilizes the memory bandwidth.

Matrix multiplication (either MxM or GEMM) is the code with the highest SDC FIT rate for Kepler and Volta. The SDC FIT rate of matrix multiplication is particularly significant when ECC is OFF (2 to 3× higher than the other codes). Matrix multiplication heavily relies on FMA operation which, according to the data in Section 6.3, is among the most vulnerable functional units. Moreover, as the code is easily parallelizable, most GPU functional units are used for computation, which exacerbates the probability of faults. Additionally, as shown in Figure 4.3, matrix multiplication has the highest AVF. As a result, the higher FIT rate of matrix multiplication is caused by the use of highly sensitive functional units, the parallel use of most of the available units in parallel, and the high probability for a fault in one unit to affect the result.

In CNNs, as YOLOV2 and YOLOV3, more than 75% of the operations are matrix multiplication related (REDMON; FARHADI, 2016). CNNs share with matrix multipli-

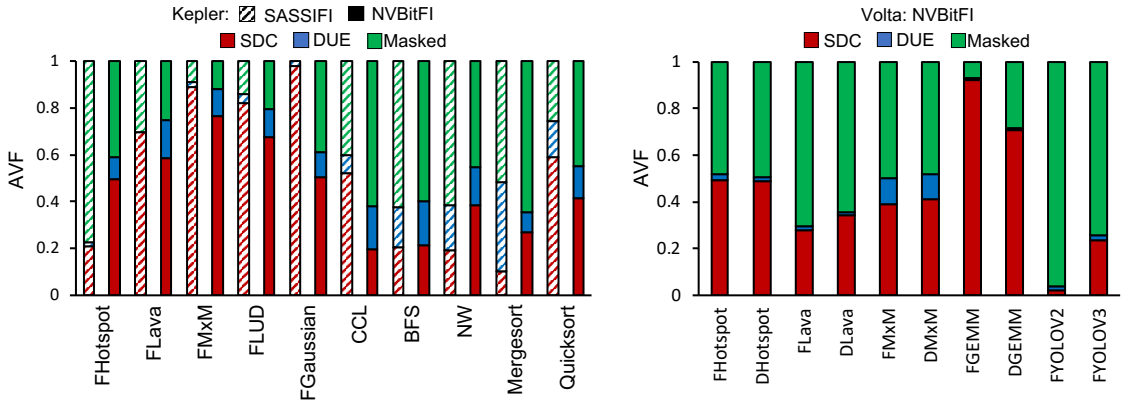
cation the problem of using a high amount of the most sensitive functional units. However, as shown in Figure 4.3, CNN's AVF is low, reducing the probability of the (likely) faults to propagate to the output.

For the Volta GPU, the focus of this analysis is comparing the FIT rates of codes executed with different precisions (double, single, and half). For all the codes, independently on the ECC status, increasing the precision increases the code FIT rate. A higher precision functional unit has a higher area and, thus, a higher probability of being hit by a neutron (see Figure 6.1). When ECC is disabled, the trend is exacerbated by the fact that higher precision implies a higher number of bits to store data, and this has a linear dependence with the FIT rate.

4.3 Architectural Vulnerability Factor

Figure 4.3 shows the AVF for all the codes considered. On Kepler, faults with both SASSIFI and NVBitFI are injected, while on Volta, faults can be injected only with NVBitFI. It is reasonable to recall that a higher AVF (probability for a fault to affect computation) does not necessarily imply a higher error rate.

Figure 4.3: Architectural Vulnerability Factor for Kepler on the left (using SASSIFI and NVBitFI) and Volta on the right (NVBitFI).



Comparing the SASSIFI and NVBitFI results on Kepler in Figure 4.3, it seems that AVF depends on the fault injector. For most of the benchmarks, the AVF is higher on NVBitFI than on SASSIFI. On average, NVBitFI provides an 18% higher AVF than SASSIFI. The differences between SASSIFI and NVBitFI on Kepler is particularly remarkable for CCL (2.6×), FGaussian (1.9×), Quicksort (1.4×), and FHotspot (0.4×). SASSIFI is older than NVBitFI and supports CUDA 7.0, while NVBitFI supports CUDA 10.1 and higher versions. As the NVIDIA compiler has been improved throughout the

versions, even for older architectures, the generated SASS code changes with the compiler versions when compiled with default options. NVIDIA compiler has two main parts. The front-end compiler takes the code written in a high-level language (e.g., CUDA) and generates e intermediate code in a virtual ISA called parallel thread execution (PTX). The back-end compiler takes the target-independent PTX code and applies many code optimizations (e.g., unrolling, loop-invariant code motion, dead code elimination, register allocation) before generating SASS code, which can run on the target GPU. Significant updates are often made to the front-end and back-end compiler infrastructure to support new features and future target SASS versions. As a result, while both fault injectors similarly instrument the SASS code, the generated code can be different due to the compiler version, with a significant impact on the code AVF.

One of the prior studies conducted on CPUs concludes that Intermediate Representation-level fault injections are as accurate as assembly-level fault injections for SDCs across different optimization levels (Palazzi et al., 2019). Given that the back-end compiler performs many optimizations, the combinations (and the exact order of which) can significantly change the sequence of the SASS instructions that execute on the GPU. Such a conclusion will likely not hold for GPUs (a detailed analysis is presented at Section 6.5). FGaussian, FLUD, FMxM, and Lava have the highest AVF on Kepler, for both SASSIFI and NVBitFI. All these benchmarks are float-based applications (see Figure 3.1).

Figure 4.3 shows the AVF for Volta obtained with NVBitFI, focusing on mixed-precision hardware. Half precision fault injection is not shown because, until the moment of this work, NVBitFI does not offer support to half instructions. As FHotspot, FLava, and FMxM execute the same kernel for all precisions, their SDC AVF is independent of data precision (the variation between double and float is lower than 4%).

GEMM (and thus YOLO that relies on GEMM for convolution) executes a different kernel for each input and precision configuration. For each GEMM configuration, the data is organized to fit each CUDA group (block, warp, and threads). For each group level, there is a different memory tile configuration to better use the caches, maximizing the pressure in the memory and the functional units. Consequently, different kernels will generate different instrumentation, which will impact the AVF. As shown in Figure 4.3, FGEMM has 30% higher AVF than DGEMM. YOLOv3 is an improved version of YOLOv2, being more accurate and complex (YOLOv3 has 113 layers, YOLOv2 only 32). On CNNs, some faults that propagate to the output are not considered errors since

they do not modify the classification result. Thus, the framework's complexity will impact the AVF, as a less precise CNN will tolerate more incorrect results than a more precise one. Faults in YOLOv2 are then less likely to affect the output as it is less accurate than YOLOv3.

4.4 Discussion on GPU reliability

The FIT rates of 11 codes obtained from beam experiments have been presented, plus an in-depth reliability evaluation of CNNs is shown. The results from beam experiments are complemented by two fault injection frameworks (SASSIFI and NVBitFI), using two NVIDIA GPUs (Kepler and Volta). All the data presented in this Chapter gives a strong motivation and background to paver the analysis that will be considered in the following Chapters. It is worth noting that the FIT and AVF evaluation are only the first steps to make conclusions on GPU reliability. The following Chapters will present other concepts and propose new solutions to improve reliability at the software level.

5 HARDENING TECHNIQUES

This chapter presents the hardening techniques proposed to improve the fault-tolerance on GPUs. At first, this chapter gives a background on existing GPU software and hardware fault tolerance methods. Then, the hardening for Convolutional Neural Networks (CNNs) is shown. The software level fault tolerance is compared with available Error Correction Code (ECC) for GPUs.

This chapter also presents a more generic software-level fault tolerance to applications on mixed-precision GPUs. The proposed technique, Reduced Precision Duplication With Comparison (RP-DWC), is a new way to implement Double Modular Redundancy (DMR) for mixed-precision GPU architectures.

5.1 Available hardening techniques for GPUs

At different levels of abstractions, several strategies have been proposed to mitigate the effects of transient faults. Memory arrays can be efficiently protected with Error Correcting Codes (ECC), which have already been shown to improve the device reliability significantly (BAUMANN, 2005). High-end GPUs protect their main storage structures with Single Error Correction Double Error Detection (SECDED) ECC. The memory hierarchy includes caches, register files, and global and shared memory. Some major resources are left uncovered, including flip-flops in pipeline queues, logic gates, block/warp schedulers, instruction dispatch units, and the interconnect network. ECC has been shown to reduce the error rate by one order of magnitude for HPC applications executed on a GPU but increases the number of crashes (Goncalves de Oliveira et al., 2016). There are several ways to mitigate SDCs in software, such as using code replication (WADDEN et al., 2014b). Unfortunately, while duplication in GPUs has been demonstrated to be even more effective than ECC (Goncalves de Oliveira et al., 2016), in real-time systems that come with strict deadlines, the overhead associated with duplication is unacceptable.

On the software level, the computation can be protected using Algorithm-Based Fault Tolerance (ABFT) approaches (HUANG; ABRAHAM, 1984; Chen; Li; Chen, 2016; RECH et al., 2013; BRAUN; HALDER; WUNDERLICH, 2014), which are generally efficient but limited to a subset of applications only, such as matrix multiplication and FFT. A more generic approach for software fault tolerance is Triple Modular Redundancy (TMR), which consists of replicating the modules of a system three times. Then,

in the end, a comparison between the modules' results is made, and the fault correction is done based on the values of two modules that are equal. The TMR has demonstrated the most feasible way to perform error detection and correction (Lyons; Vanderkulk, 1962; Chen; Li; Chen, 2016). However, TMR costs in terms of area, power consumption, and performance are very high (Kastensmidt et al., 2005), in some cases, much more than $3\times$ (Chen; Li; Chen, 2016).

In contrast, when it comes to computation, one of the most effective solutions to detect and mitigate transient faults is duplication or, in general, replication. By comparing the output of two independent copies of an instruction, it is possible to detect most of the transient faults (ZIEGLER; PUCHNER, 2010). Duplication can be implemented in software or hardware, in time (by executing the duplicated operations sequentially), or in space (by running the duplicated operations in parallel).

Instruction Duplication With Comparison (DWC), a software-level approach, can be easily implemented in highly parallel architectures such as GPUs. As threads are executed in separated computing units, it is improbable for a fault in a copy to affect also the duplicated one. Thus, as demonstrated in (OLIVEIRA et al., 2014; WADDEN et al., 2014b; MAHMOUD et al., 2018), DWC in GPUs can detect more than 80% of the transient faults. However, errors in shared resources (like caches, if unprotected) or in the scheduler can potentially affect multiple threads and, thus, remain undetected. While being very useful, DWC is far from being efficient as the introduced overhead in terms of power consumption, silicon area, or execution time can be too high (costing at least 100% more than the unhardened version). As a result, DWC is impractical for HPC and real-time systems such as autonomous vehicles.

Recent work has proposed solutions to reduce the cost of DWC in GPUs significantly. With redundant multithreading, for a particular set of applications, a slowdown of less than 10% can be achieved but, for others, because of the cost of communication, the overhead is still higher than 2x (WADDEN et al., 2014b). Software-managed Instruction Replication for GPUs (SInRG), on the contrary, can reach an overhead that is, on the average, 69% the unhardened code execution time (MAHMOUD et al., 2018).

All the efficient DWC approaches proposed to date work under the assumption that sufficient computing resources are available to run the duplicated copy in parallel with the original one. Otherwise, the DWC overhead would still be forced to be at least 2x. This work proposes to leverage the concept of approximate computing and the available resources in modern mixed-precision architecture to reduce the overhead of DWC even

further.

Many previous works have investigated computations in half- or single-precision floating-point as an effective way of approximating an application and improving both performance and energy efficiency (LAM et al., 2013; GRAND; GöTZ; WALKER, 2013). Past work on approximate computing and mixed-precision architectures have demonstrated the feasibility of exploiting the inherent error-tolerance of particular computational patterns for improved performance or energy efficiency. In this work, the available forms of controlled approximation to implement Instruction Reduced-Precision Duplicate with Comparison (RP-DWC) is exploited, amortizing the overheads traditionally associated with duplication.

5.2 Fault Tolerance for CNNs

This work advances the knowledge of GPU reliability by characterizing how microarchitecture vulnerabilities in a GPU can undermine a CNN's reliability. Most previous works have focused on HPC application reliability on a GPU, leaving CNNs reliability unstudied. CNNs are a class of applications of extreme importance for autonomous vehicles, making their reliability evaluation fundamental.

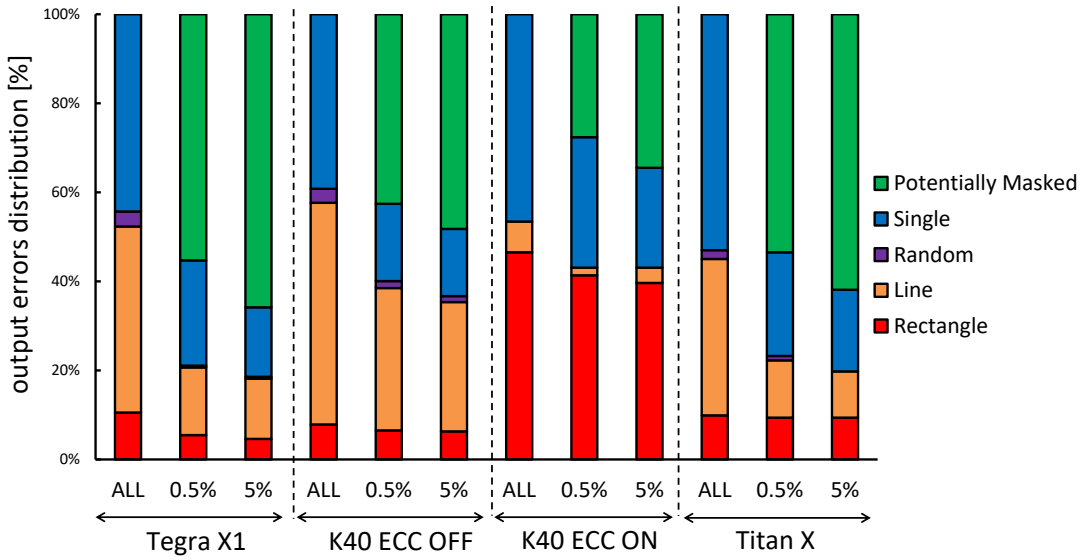
Based on past analysis (LI et al., 2016), ECC does not mask all the faults as an error in computing elements could propagate to the output. The beam tests provide the realistic probability of experiencing an SDC when ECC is enabled or not, which is the only way to evaluate ECC's effectiveness. Not all the errors affect the CNNs classification/detection, so they are considered Tolerable errors. Otherwise, errors that modify the CNN's output would be characterized as Critical errors. As shown in Section 2.3.1 and detailed better in (Santos et al., 2019), a novel and worrying insight derived from the beam tests is that ECC does not reduce (or reduce only slightly) CNNs Critical SDC rate. So the following two Sections, 5.2.1 and 5.2.2, detail improvements on CNN's reliability by using software based fault tolerance.

5.2.1 GEMM ABFT

Figure 5.1 shows the percentage of corrupted executions that are affected by single, line (i.e., multiple corrupted elements on the same row/column of the output ma-

trix), rectangle (four or more elements distributed in a rectangle/square), and randomly distributed errors (random errors are grouped single errors that do not fit in the other categories). The result also shows the spatial error distributions for 0.5% and 5% acceptable error margins. When a faulty execution has all of its corrupted elements inside the tolerance margin are tagged as *Potentially Masked*. In the vast majority of cases, GPU corruption affects more than a single output element. It is worth noting that multiple corrupted elements are not necessarily caused by multiple impinging particles but instead inherent in the GPU computation to *spread* low-level transient faults across multiple output elements. Considering multiple corrupted elements, the reliability of the neural network is going to be impacted. Interestingly, most of the potentially masked executions are those affected by a single corrupted element, while the number of rectangle errors remains almost unchanged. It is possible to conclude that those errors that spread through the GPU microarchitecture and the GEMM program are the same ones that have a greater difference from the expected value.

Figure 5.1: SDCs spatial distribution in GEMM. All SDCs are extracted on beam experiments.



Rectangle errors are potentially the most severe for CNNs. Rectangle errors in GEMM are caused by faults that impact the scheduling of the execution of multiple threads in a Streaming Multiprocessor (SM). GEMM increased execution efficiency if compared to a naive matrix multiplication implementation since it can divide the input matrices into chunks that fit nicely in the cache of a SM, reducing memory latency. If a fault can cause a thread to be incorrectly assigned or scheduled to a SM, or if some

threads fail to synchronize, the whole SM output matrix portion is likely to be corrupted, leading to a rectangular error. Errors in memory elements protected by ECC (e.g., registers and caches) have been reported to manifest into either single corrupted element or line errors (RECH et al., 2013). When ECC is turned ON, single and line errors (which are less critical for CNNs) are corrected, but all the other errors (including rectangular errors) are not corrected. As a result, the percentage of rectangular errors increases when ECC is enabled.

Figure 5.1 helps to explain why ECC is not very effective in reducing SDCs and most critical SDCs in GEMM, consequently on CNNs. ECC reduces the absolute number of SDCs but has the side effect of increasing the portion of rectangle errors, which are more likely to propagate through CNN layers and affect detection. Additionally, the average difference between a corrupted element's value and the expected value is higher when ECC is enabled. It is then more likely for the faults not masked by ECC to significantly impact the correctness of CNN.

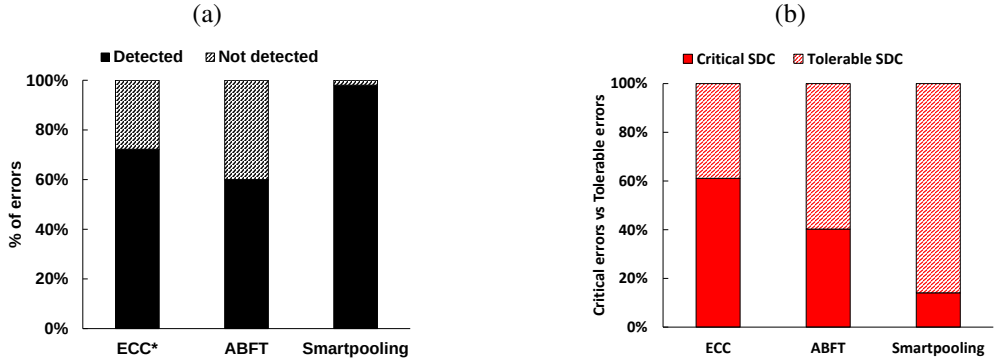
Since ECC does not seem to be effective for CNNs, this work leverages ABFT to protect YOLOv1, protecting matrix multiplication operations, as described by Huang et al. (HUANG; ABRAHAM, 1984), and extended by Rech et al. (RECH et al., 2013), to correct line and random errors in $O(N)$ time. For FFT-based convolutions, a promising ABFT able to detect more than 80% of faults has been described by Pilla et al. (PILLA et al., 2014). Unfortunately, it is not possible to implement and evaluate ABFT on Faster R-CNN and Resnet as they are built with NVIDIA proprietary cuDNN libraries, based on Caffe and Torch, respectively. Nevertheless, the GEMM core used to perform convolution in Faster R-CNN and Resnet is the same algorithm used in YOLOv1. The portion of SDCs affecting GEMM that ABFT can correct will be very similar between the three frameworks. Since the percentage of GEMM operations is higher in Faster R-CNN and Resnet (82% and 80%, respectively) than in YOLOv1 (67%), ABFT could even be more effective for the former frameworks.

It has been decided not to add a dimension to the input matrices for the checksum to adapt ABFT for the CNNs, but instead to *substitute* the last row of the first input matrix and the last column of the second input matrix with column/row checksums. The GEMM kernels are tuned to fully use caches and registers. Adding an extra dimension would compromise execution time and significantly increase data movement and memory latency. This would result in a different behavior under radiation not solely related to ABFT. It is worth noting that the evicted row/column does not significantly reduce YOLOv1 detec-

tion capabilities (differences from the ECC OFF and ABFT protected versions are lower than 10%). It is possible to believe that performing the training of YOLOv1 with the evicted rows/columns would reduce the detection differences. To focus on the efficacy of the hardening strategy, the same weights are used (i.e., the same non-evicted data) to keep the protected and unprotected versions similar.

The figure 5.1a shows the percentages of the detection for each fault tolerance tested on YOLOv1 running on NVIDIA Kepler GPU. Figure 5.1a shows the percentage of Critical and Tolerable SDCs produced for each technique. For comparison, the ECC detection efficacy is extracted using the relative difference between the ECC OFF version vs. the ECC ON version of YOLOv1. That is, figure 5.1a the ECC percentage of detection is the relative difference between the execution under beam with ECC turned on vs. ECC turned off. The ABFT procedure does not significantly affect YOLOv1's SDCs or Crash error rate.

Figure 5.2: Figure 5.1a shows the Detected vs Undetected errors for the three tested fault tolerance for YOLOv1. Figure 5.1b shows the Critical vs Tolerable of undected SDCs on YOLOv1 coparing the fault tolerance approaches



It is clear that ABFT can correct about 60% of the SDCs. If considering only the critical SDCs, an ABFT-protected YOLOv1 is more resilient than an ECC-protected version. This is because ABFT corrects all the detected errors that affect GEMM computation. As shown in Figure 5.1, 80% to 90% of the observed errors for GEMM are single, line, and random errors, which ABFT can correct.

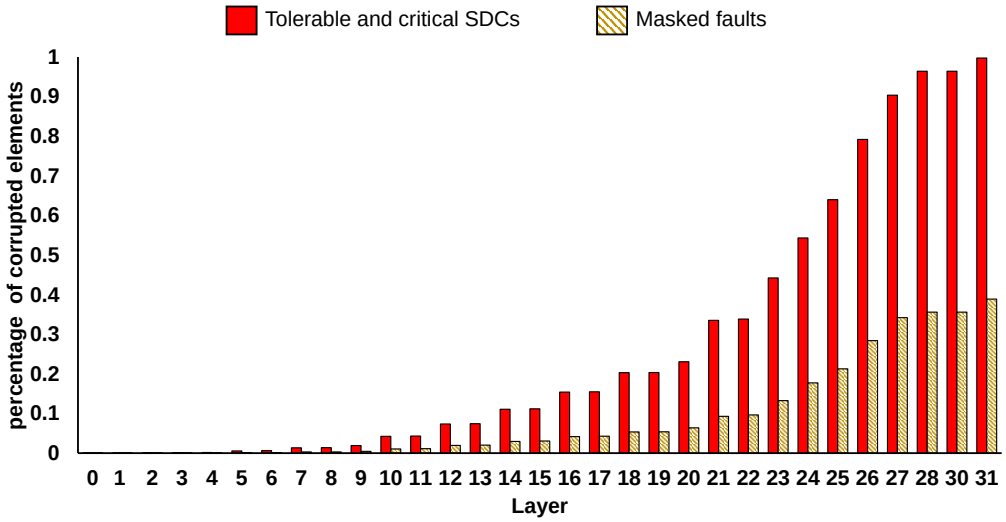
As noted earlier, on GPUs, the ABFT code is run in parallel and executed in linear time. However, since ABFT performs multiplication/accumulation, it can be implemented using the same hardware used for matrix multiplication. That is, almost no hardware changes are required to implement the proposed hardening. ECC, on the contrary, has a logarithmic memory cost.

5.2.2 Reliable Max-pooling

For this research, a utility that captures all of the outputs from each YOLOv1 layer has been created to track faults propagation through the layers. This feature adds a small execution overhead but does not modify the execution flow. When a fault is injected, each layer's data is compared with the fault-free data. Then it is possible to identify the layer in which the fault occurred and how the fault modifies the data matrix in each layer. It can also track how that fault propagates through the CNN pipeline until the fault is masked or reaches a visible program output.

Next, based on the results, this work proposes to re-design the max pool layers to detect, and possibly correct, faults. Figure 5.3 shows the percentage of the corrupted elements across the CNN. As shown in Figure 5.3, errors spread quickly in a CNN. Injections leading to SDCs or Critical SDCs tend to have all the last layer's output elements corrupted. Alternatively, injections that are masked corrupt less than half of the elements of the CNN output. To make CNN execution more reliable, it is fundamental to promptly detect errors to prevent errors from propagating and reaching the fully connected layers.

Figure 5.3: Average percentage of corrupted elements at the output of each layer.



Additionally, the results show that in a fault-free execution, the maximum absolute value for elements entering the maxpool layer, considering all of the frames in the Caltech and VOC datasets, is 21.15 for 1P, 12.23 for 3P, 8.39 for 8P, and 5.72 for 19P. These values are extremely small, considering the full range of FP16 or FP32 that can be represented and that radiation can produce. Instead of simply propagating the element

with the highest value, the max-pooling operation should also evaluate if the value of the element to propagate is *reasonable*.

A more reliable maxpool layer should evaluate if the value of the max element is greater than a threshold (to be conservative, the threshold is set to be $10x$ the max value of a fault-free execution) and, if so, halt the processing of the current frame and move to the next frame. This solution will detect those faults in GPUs that affect multiple elements that, as demonstrated in Section 5.2.1, are also the faults with the greatest impact on the final value.

It is feasible to go a step further and correct errors by designing a maxpool layer that, when detecting a faulty max value, propagates the second greatest element, if *reasonable*. As the maxpool layer is intrinsically imprecise, propagating the second-highest value does not significantly undermine detection quality (RIESENHUBER; POGGIO., 1999; SCHERER; MÜLLER; BEHNKE, 2010). The overhead introduced to implement detection/correction is limited to 4 variables that hold the thresholds and the potential error detection overhead, which is done in O(1). This overhead is much lower than the overhead for ECC, which is $O(\log n)$.

Figure 5.1b reports the percentage of SDCs detected, and undetected with the proposed max pool layer. Data was obtained with beam experiments. Smartpool detected 98% of SDCs under the beam. The most promising result is that the radiation experiments demonstrate that only 2% of the SDCs remain undetected, which is extremely close to the 99% detection limit ASIL-D imposes for self-driving vehicles.

5.3 Reduced Precision Duplication With Comparison

The traditional Duplication With Comparison (DWC) is a generic system modular redundancy with a result comparison at the end. RP-DWC consists of duplicating the instruction flow for execution in a lower precision. Using a reduced precision replica to implement DWC has three main benefits compared to a traditional DWC:

Smaller overhead. For example, an FP32 operation imposes about half the overheads of the equivalent FP64 operation. In the specific case of modern GPUs addressed here, since spare mixed-precision units are available, these overheads can be reduced even further. When no dependence between the two replicas exists (intrinsically true for any DWC-based strategy), and the code is not shared-memory or register limited (i.e., if the original code does not use all of the available registers or memory), then two replicas can

be executed in parallel.

Lower probability of having the replica corrupted, as reducing precision has the effect of reducing the code error rate (see Section 4.2). It is expected that half the chance to have detections caused by errors in the replica when using RP-DWC concerning traditional DWC.

Diversity of the copies, since the replicated operations will now execute in a different precision and use different processing resources, reducing the chances for a fault to have the same impact in both copies and remain undetected.

Next, a few definitions are shown to formalize the RP-DWC approach. For simplicity and ease of understanding the implementation on modern GPUs, it is considered, without loss of generality, an FP64 original code and an FP32 redundant copy. The same definitions and considerations apply to other precisions.

Considering two real numbers x and y, which can be encoded either as FP64 (x_{64} and y_{64}) or in FP32 (x_{32} and y_{32}). Operating x and y together yields value z_{64} (the output of the original application) or z_{32} (used only for fault-detection). Even in the absence of faults, due to the reduced precision, $z_{64} \neq z_{32}$. RP-DWC defines a tolerable interval for the difference between z_{64} and z_{32}, named *Expected Precision Loss (EPL)*, over which error detection is triggered. EPL acts as a threshold for triggering fault detection.

The intrinsic difference between z_{64} and z_{32} does not allow for detecting all the faults. However, as shown in the results, most of the faults are still detected with RP-DWC. The undetectable faults might be tolerated as they produce an error inside the precision difference between FP64 and FP32 operations.

5.3.1 Overview of the Implementation

This work explicitly targets modern GPU architectures to show RP-DWC implementation and leverages the mixed-precision hardware to further amortize DWC costs. RP-DWC is a software technique to be applied at compiler time. To implement RP-DWC, five steps must be taken. The steps must be applied to each existing full precision instruction from the original program flow:

Step 1. Casting the inputs to Reduced Precision. The 64 bits input data (x_{64} and y_{64}) needs to be reduced to 32 bits (x_{32} and y_{32}) to feed the reduced-precision replica. There are various ways to perform the cast operation. For instance, NVIDIA features four different casts (round up, round zero, round to nearest, round towards zero) (NVIDIA,

2017). The cast operation does not significantly impact the Expected Precision Loss, as in the experiments performed, the differences between the casts were, on average, orders of magnitude smaller than 1%.

The default *round to nearest* cast is selected. The value is taken from the input registers of the FP64 copy, and the casted value is stored in registers that are serving as an input of the FP32 copy. While RP-DWC does not increase the pressure on caches or main memory, it increases the pressure on the register file, possibly increasing the overhead if there are insufficient registers to hold the values for both the FP64 and FP32 copies. This property is intrinsic of any software DWC for GPUs, but, as the FP32 copy requires only half the registers compared to the FP64 copy (each FP64 register occupies two FP32 registers), RP-DWC is less likely to reach register saturation than a traditional DWC. Additionally, as in any DWC, if the fault affects the data *before* replication, both copies will receive an erroneous input, preventing detection. On GPUs that include ECC in the main memory structures, it is convenient to assume that the probability of having such a corruption in the input data is very low.

Step 2. Executing the original and reduced-precision instructions. The two copies are executed, either sequentially or in parallel in case dedicated mixed-precision hardware is available. RP-DWC for GPUs duplicates the instructions inside a thread rather than duplicating threads, warps, blocks, or kernels. By keeping the duplicated instructions inside the same thread of the original copy, the programmer ensures that those are executed in the same core using the idle mixed-precision hardware, avoiding dependencies, duplicated caches, and synchronization issues. Since it is only necessary to duplicate the instructions, then add a comparison at the end of the thread or after an instruction code block, the duplication process can be automated at the compiling stage.

Step 3. Casting the high-precision result to reduced precision. Once both instructions complete their execution, z_{64} is cast down to FP32 to prepare the comparison. The z_{64} is cast to FP32 as it is needed only to detect an error, and more accurate error detection by performing the comparison in FP64 will not be achieved.

Step 4. Performing the error detection operation. Dissimilarity to a traditional DWC, as the two copies z_{64} and z_{32} are naturally different, in RP-DWC a simple a bit-wise comparison does not suffice to detect faults. It is necessary to decide if the difference between the two copies is within the Expected Precision Loss (EPL), which, for now, is considered as given. The two outputs can be compared using the relative difference $\delta_r = \frac{z_{32}}{z_{64}}$ (with z_{64} casted to FP32, in Step 4a). The operation performed in RP-DWC (a

division) is much more complex than in a traditional DWC (a comparison). The absolute difference would reduce the error detection overhead, but the result would depend on the exponent (subtracting similar numbers with a high exponent provides a huge absolute difference).

In the specific case of NVIDIA Volta GPUs, there are twice as many FP32 cores as FP64 cores (see Chapter 3.1 for more details). As RP-DWC uses one FP32 core per each FP64 operation in the original copy, there are indeed enough FP32 cores available to perform the division in parallel with the next FP64-FP32 execution. NVIDIA also features a fast division operation (*fdividef*) that approximates the division results and takes, on average, half the time of a standard division. The use of *fdividef* rather than a normal division to detect faults is suggested as the relative difference between the results is negligible.

An alternative comparison is the use of unsigned integers. To compare z_{64} (now cast to FP32) and z_{32}, it is possible to consider their representation as an unsigned integer (UINT32) and subtract them, which produces the difference δ_{UINT}. Re-interpreting the FP32 values as UINT is not a cast operation since it requires no modification in the representation and does not significantly increase the overhead. By subtracting the two 32 bits representations interpreted as UINT gives a fast (and accurate) evaluation of the magnitude of the difference between the two numbers, which is exactly what is needed to detect errors. The higher the result of the subtraction, the more significant is the difference between the two representations. If the two representations differ only for some bits in the least significant digit of the mantissa, their values as UINT are very close. On the contrary, if the two floating-point values have different exponent or sign bit, their values as UINT are very different. As for δ_r calculation, the UINT subtraction can also be performed in parallel with the next FP64-FP32 as GPUs have dedicated integer cores (see Chapter 3.1).

The user can choose the use of δ_{UINT} and δ_r. From the past experience, δ_{UINT} is faster but might detect fewer errors than δ_r if the exponent is very low.

Step 5. Comparing and taking action on the result. Once z_{64} and z_{32} are compared, the difference between z_{64} and z_{32} can be either (1) within or (2) outside the EPL.

In case (1), it is assumed that the instruction execution is successful or, if a fault happened, it produced an undetectable error. The execution can then proceed after casting the next FP64 input to FP32. It is necessary always to feed the FP32 copy with the FP64

values to reduce the divergence of reduced-precision executions. In case (2), an error flag is raised to inform the application that an error has been detected.

It is worth noting that, as in any other duplication strategy, the comparison operation could be a single point of failure, i.e., a fault in this operation can lead to undetectable errors. Nevertheless, as experimentally shown in Section 5.3.4, this does not undermine the error coverage of RP-DWC as the comparison operation is less likely to be corrupted than the duplicated operation. With the granularity discussed in Section 5.3.2 the probability of a fault in the comparison operation is reduced even further.

Depending on the system implementation, once the error flag is detected, the execution can be terminated or rolled-back to the previous check. The roll-backed execution should be performed in FP64 by both copies to guarantee, for instance, that the detected error does not come from a round-off error. However, one can decide to restrict the threshold if inputs are very well structured (for example, in image processing applications, the pixels' colors are well limited, and in physical simulations, the possible values are known). If an unexpected input is received, a false-positive can trigger error detection, and re-execution will not solve the problem. Additionally, roll-back also can deal with the unlikely catastrophic cancellations. While executing both copies in FP64 might seem to undermine the efficiency of RP-DWC, that only happens when an error (or an unlikely false-positive) is detected. In these few cases, the overhead would be identical to that of a traditional DWC.

5.3.2 Granularity of the Approach

DWC can be implemented with different granularities, i.e., performing the correctness check at each instruction or after a block of instructions. In a coarse-grained RP-DWC, a sequence of FP64 instructions is duplicated with a series of FP32 instructions. The replicated sequence receives the FP64 input cast to FP32. The two sequences are then executed in parallel without interacting until reaching the correctness check.

To detect errors as early as possible, the correctness should be checked at each instruction. However, this requires a cast and a comparison operation at each operation, increasing the introduced overhead (which, as shown in Section 5.3.5, is still lower than a traditional DWC) and the probability for the error to occur in the comparison itself.

To evaluate the trade-off between the introduced overhead and the achieved error detection, the correctness check with various granularities have been implemented: at

each instruction, at the end of the computation, and after some or several instructions (10, 100, 1,000). Longer sequences are not considered as basic blocks in GPUs have, on the average, between 20 to 100 instructions (CHAKRABARTI et al., 2012). It is worth noting that a correctness check should be performed independently on the chosen granularity before any data-driven condition statement. A small divergence between FP64 and FP32, eventually caused by intrinsic differences, could force the two copies to take different paths.

A longer block of instructions can potentially increase the two copies' intrinsic difference and, thus, the EPL. A bigger EPL, in principle, implies a higher number of undetectable errors. However, as the error propagates in the sequence of instruction, it may increase in magnitude, thus becoming detectable even with a bigger EPL.

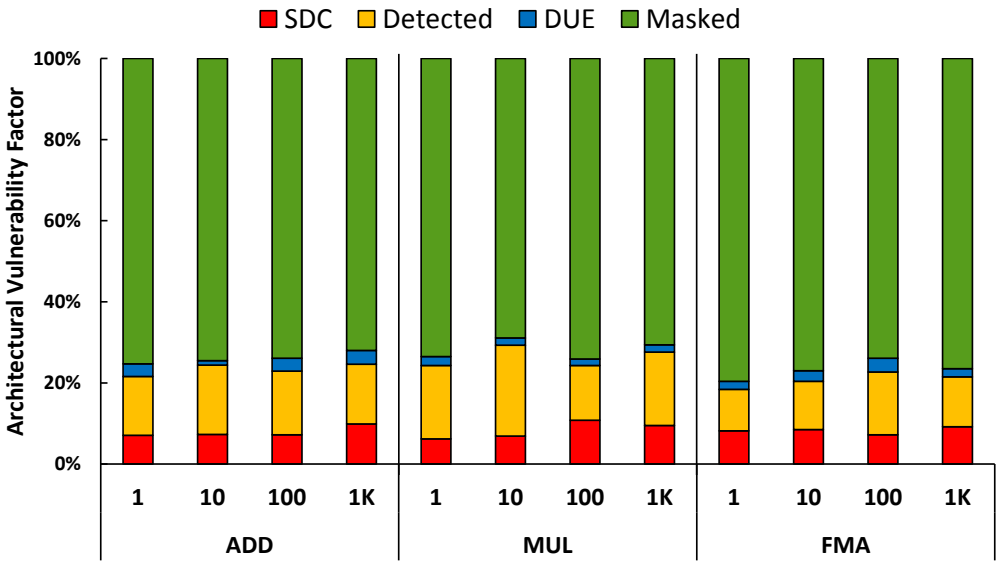
5.3.3 Architectural Vulneratibility Factor

First, the AVF is analyzed, i.e., the probability for an injected fault (random single bit flip) to propagate and generate a DUE (crash/hang) or a (detected or undetected) SDC. The results are presented using δ_{UINT} to compare the two copies. The use of δ_r would not affect the reported data significantly, having an impact on overhead lower than 5% in the fine-grain RP-DWC and negligible in the coarse-grain RP-DWC.

Figure 5.4 shows the percentage of injected faults that are masked, that became an SDC that is detected or undetected, or that produce a DUE. The data is obtained for the micro-benchmarks with different RP-DWC granularities. That is, performing the correctness check at each instruction, every 10, 100, or 1,000 instructions.

Figure 5.4 demonstrates that most of the injected faults are either masked or detected. Additionally, the AVF for DUE is very small compared to the AVF for SDC. This is because DUEs are mainly produced by errors in control logic or interfaces (FRATIN et al., 2018; Chatzidimitriou et al., 2019), while the injections target mostly the datapath. Figure 5.4 also shows that there is no significant difference in the AVF between operations and configurations (fine or coarse grain RP-DWC). Having similar AVF attests that the configurations and operations have a similar probability for a fault to propagate or to be masked. Fault injection does not include any information about the probability of a fault during the different computations. In other words, it is not possible to evaluate from Figure 5.4 alone if the additional instructions required to implement duplication affect the code error rate. Beam experiment data, presented in the next subsection, will also

Figure 5.4: AVF results for the micro-benchmarks.



consider this aspect.

Table 5.1 provides additional details on the efficiency and efficacy of RP-DWC by listing the percentage of errors that are detected and the imposed overheads (execution time and energy consumption). The detection rate and the overheads of a traditional DWC are also included to ease the comparison with RP-DWC. Please recall that, while for RP-DWC micro-benchmarks, the data is shown for different granularities. For the traditional DWC and the realistic codes (MXM, Lava, FWT, and BlackScholes), only the entire code's duplication is shown, which is the best-case scenario overhead.

5.3.4 Error Detection

As shown in Table 5.1, the percentage of detected errors for RP-DWC ranges from 57% for 1,000 op FMA to 76% for 10 op MUL. While increasing the granularity increases the threshold, there seems to be no correlation between the detection rate and the granularity. As discussed in Section 5.3.2, there are two reasons for that: (1) Fine-grain RP-DWC requires a check at every executed instruction, increasing the probability of having an (undetectable) check operation corruption. (2) An undetectable error in the first operations of a block in a coarse-grain RP-DWC can increase in magnitude as it propagates, eventually becoming detectable when the correctness check is performed. The longer the block of instruction, the higher the probability for the undetectable errors to become detectable. That implies a tradeoff is present between the performance overhead

Table 5.1: Error detection and overhead (Fault Injection).

Benchmark	Granularity	Overhead Time	Overhead Energy	Detection
ADD (RP-DWC)	1 op	34,9%	51,2%	67.1%
	10 op	3,9%	41,0%	70.1%
	100 op	0,3%	28,0%	68.6%
	1,000 op	0,1%	23,9%	59.8%
MUL (RP-DWC)	1 op	35,3%	52,3%	74.5%
	10 op	3,6%	41,0%	76.5%
	100 op	0,3%	33,0%	59.3%
	1,000 op	0,1%	31,9%	70,0%
FMA (RP-DWC)	1 op	35,3%	55,9%	55.4%
	10 op	3,6%	43,0%	58.3%
	100 op	0,3%	35,0%	68.3%
	1,000 op	0,1%	32,9%	57.2%
MXM (RP-DWC)		13,0%	62,0%	63.4%
LavaMD (RP-DWC)		18,1%	56,0%	83.4%
BlackScholes (RP-DWC)		5,8%	13,8%	66.2%
FWT (RP-DWC)		37,4%	50,1%	75.6%
ADD (trad. DWC)		94,2%	111,0%	>95%
MUL (trad. DWC)		94,4%	124,0%	>95%
FMA (trad. DWC)		99,1%	124,0%	>95%
MXM (trad. DWC)		70,3%	108,0%	>95%
LavaMD (trad. DWC)		83,6%	107,0%	>95%
BlackScholes (trad. DWC)		50,5%	76,7%	>95%
FWT (trad. DWC)		94,6%	106,6%	>95%

and the time to detection, but not between the performance and the detection rate.

The slightly lower detection rate compared to traditional DWC (over 95% from Table 5.1) or state-of-the-art DWC from previous work (Maniatakos et al., 2011; Seetharam et al., 2013; KUDVA et al., 2013; WADDEN et al., 2014b; SULLIVAN, 2015; Goncalves de Oliveira et al., 2016; MAHMOUD et al., 2018; Traiola et al., 2018; RODRIGUES et al., 2019) is not surprising, since, as any faults hitting or propagating to the less-significant bits of an FP64 number are not detectable. Even though this limitation of RP-DWC provides an upper bound of erroneous bits coverage it can achieve, these wrong least significant bits are precisely the ones that will provide a smaller impact to the application output, as shown in Section 5.3.7.

RP-DWC has a higher detection capability than previous work that approximates the algorithm or proposes approximated hardware for error detection ((Maniatakos et al., 2011; ZHANG; NATHAN; SORIN, 2015; RODRIGUES et al., 2019)), which is 55%-76% for RP-DWC and 20%-40% for previous work. This data attests that approximating the algorithm might not be as effective as approximating the hardware. The higher error

detection of RP-DWC compared to the use of dedicated approximated hardware could be caused by the higher approximation chosen in (Seetharam et al., 2013; Maniatakos et al., 2011) and by intrinsically more reliable hardware designed by NVIDIA.

5.3.5 Overhead

Data in Table 5.1 shows that the execution time overhead of RP-DWC (35% in the worst case of a fine-grain RP-DWC) is much lower than traditional DWC (70%-90%) and of recent efficient DWC (39% in (MAHMOUD et al., 2018)). The implementation created for this work of conventional DWC has a lower overhead compared to some previous studies that showed an overhead of 2x (Goncalves de Oliveira et al., 2016), as it duplicates operations inside a thread rather than threads or blocks of threads. If FP64 cores are available, the GPU could then be able to schedule some of the two FP64 copies in parallel. As expected, as the granularity of the correctness check is reduced, the overhead is reduced. When the granularity is increased, as the comparison operations are less frequent, the overhead is reduced accordingly, eventually becoming nearly-zero.

The Instructions Per Cycle (IPC) and the number of executed instructions of the RP-DWC version are approximately 2x the unhardened versions, for all configurations. On the contrary, for the traditional DWC versions, while the number of executed instructions is slightly higher than 2x, the IPC is very similar to the unhardened version (lower than 1.2x). This also proves that the available resources now contribute to the reliability improvement with a nearly zero overhead.

To further investigate the overhead differences between RP-DWC and traditional DWC, the register file pressure is considered. The higher register file usage is found for MXM traditional DWC, for which 41 registers per thread are instantiated. As the register limit on NVIDIA Volta is 256 per thread, none of the tested configurations saturates the register file. The higher overhead for traditional DWC, then, comes from the saturation of computing units.

Table 5.1 shows that the energy overhead of RP-DWC is significantly reduced when the correctness checks are less frequent, reaching a value that can be as low as 24% to 32% BlackScholes reaches an even lower energy overhead (13.8%). However, such a low overhead is not solely justified by RP-DWC, but also by the simplicity of the code (few global memory access, no shared memory utilization, and executes only simple operations) (Yazdanbakhsh et al., 2017). For the same reasons, the energy overhead of the

traditional DWC is lower than for the other codes. The energy consumption overhead of RP-DWC is comparable to the traditional DWC only for the fine-grain implementation (1 check every operation for RP-DWC vs. one check at the end of the application for traditional DWC). The energy consumption overhead of a fine grain RP-DWC is, then, higher than 100%. This is justified because BlackScholes is actually executing 3x the instructions of the unhardened version (FP64 and FP32 copies plus the error detection operation). Traditional DWC has, even with the check only at the end of the computation, a higher energy consumption overhead, which ranges from 111% to 124%. This favorable energy consumption result of RP-DWC is achieved by leveraging the FP32 cores to execute the redundant copy in parallel.

Finally, for RP-DWC, the energy consumption overhead is always higher than the time overhead as the error detection operation (UINT or δ_r) can be done in parallel with floating-point operations in GPUs, reducing the time but not the energy overhead of activating the idle cores for error detection.

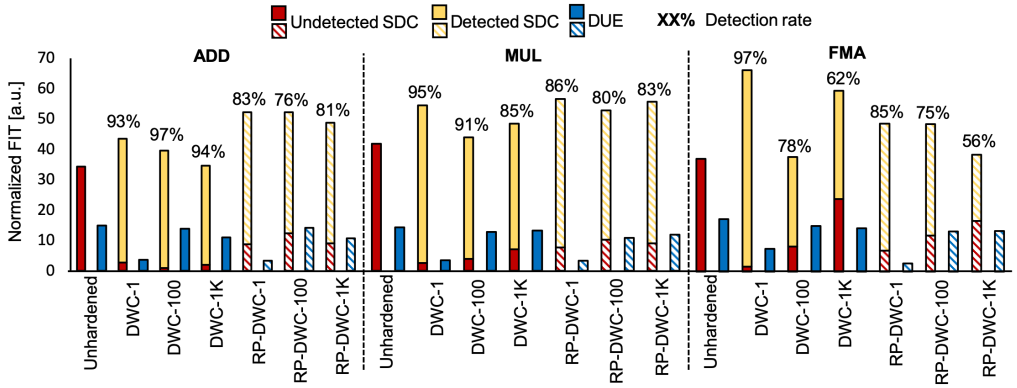
5.3.6 Neutron Beam Experiments

To have an even more realistic evaluation of the effectiveness of RP-DWC and a direct comparison with traditional DWC, the GPUs were exposed to accelerated neutron beams. Dissimilarity to fault injection, during beam experiments, all the GPU resources are irradiated and could be corrupted, and the fault model is as close as possible to the real one.

Figure 5.5 reports the beam experiment results for the microbenchmark in the unhardened version (no duplication) and protected with a traditional DWC and with RP-DWC, in three different granularity (correctness check after 1, 100, or 1,000 operations). The detection rates of DWC and RP-DWC are made explicit in the Figure to ease comparing the effectiveness of the two techniques. All the reported values are affected by a 15% experimental error due to statistic and neutrons count uncertainty. FIT rates have been normalized to the lowest measured value (DUE rate of DWC-1 for ADD) not to reveal business-sensitive data but still to allow a direct comparison between configurations. The dashed lines are used just to differentiate the RP-DWC results from the traditional DWC ones.

From Figure 5.5, it is possible to notice that the SDC FIT rate for the unhardened version is lower than the FIT rate of the protected versions (considering the combination

Figure 5.5: Normalized beam experimental data for the unhardened, the traditional DWC, and RP-DWC versions of the micro-benchmarks. Dashed lines are used just to highlight the RP-DWC versions.



of detected and undetected SDCs). This is expected, as the check operation introduces a computation and memory overhead that can increase the protected versions error rate. The increased FIT rate is higher when the check is performed at each instruction, as more instructions are being executed. RP-DWC has a higher increase in the SDC rate for all configurations but FMA. This is because the cast and error detection operations (detailed in Section 5.3.1) are computationally more costly than ADD and MUL, but not of FMA. Nevertheless, both traditional and RP-DWC detects most of the SDCs, resulting in a much lower undetected SDC rate than the unhardened versions.

As observed with fault injection, and for the same reason, the detection rate of the traditional DWC is always higher than the RP-DWC. On average, under the beam, the detection rate of RP-DWC is 9% lower than DWC. The fact that the detection rates of RP-DWC are higher than those reported in Table 5.1 should not surprise. Data in Table 5.1 is obtained injecting only single bit flips, which are harder to detect with RP-DWC. When even one corrupted bit is outside of EPL, RP-DWC triggers the detection. The higher the number of bits flipped, the higher the probability of detection. As already studied in previous work (RECH et al., 2014; Goncalves de Oliveira et al., 2016), radiation may induce complex fault models in GPUs (multiple bit flips, the corruption of several threads, and so on) that are easier to detect with RP-DWC. Detection rates in Table 5.1 are then a conservative estimation of RP-DWC detection capabilities. Nevertheless, radiation during beam experiment can also corrupt data before it is assigned to the two copies and, thus, resulting in undetected errors. ECC's presence would remove this possibility (the tested Titan V does not have ECC), possibly improving the detection rates of both DWC and RP-DWC.

Then, the use of RP-DWC results in a slightly lower error detection but makes

both overheads much smaller than a traditional DWC and state-of-the-art DWC for GPUs as reported in (WADDEN et al., 2014b; MAHMOUD et al., 2018). As shown in the next subsection, the errors RP-DWC does not have a minimal impact on the output correctness and could potentially be tolerated, mitigating the observed lower detection's impact.

5.3.7 Detected vs Undetected Errors

As discussed in Section 5.3.1, in a real application of RP-DWC, once an error is detected, the computation would either be terminated or the roll-back mechanism would be triggered. In the experiments, the execution is allowed to complete even if an error is detected in order to compare the impact of detected and undetected errors in the output correctness.

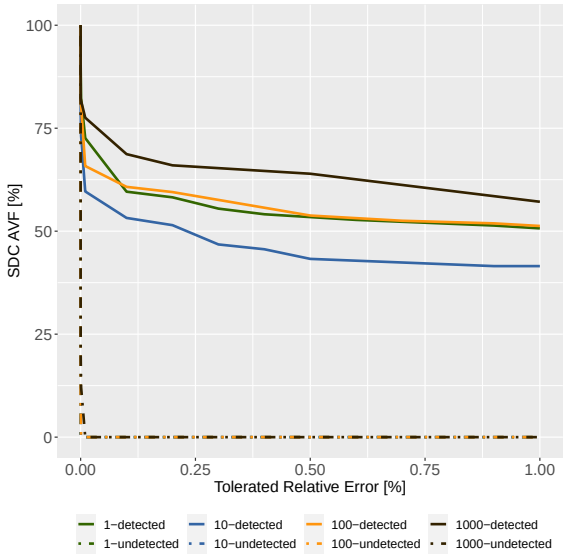
To measure the impact of these errors, the concept of *Tolerated Relative Error (TRE)* is used as introduced in (Fernandes dos Santos et al., 2019). A TRE of 0% implies no tolerance in the output correctness, i.e., the computed output must match the expected output. In other words, any mismatch between the computed output and the expected value is considered a critical error. Increasing the TRE relaxes the correctness constraint accepting (corrupted) values in a given range as tolerable (corrected). As an example, considering a TRE of 10%, any output value between 90% and 110% of the expected value will be considered as correct or, at least, tolerable. If the output is composed of multiple elements, as in the tests, *all elements* must have a tolerable value for the execution not to be considered corrupted.

Figure 5.6 shows how detected and undetected errors affect the code output. Only fault-injection data is reported. Beam experiment results are similar and not shown. In the y-axis, how much the SDC AVF rate would be reduced is plotted as a function of TRE (that varies from 0% to 1% on the x-axis). When TRE is 0%, the AVF will be precisely the experimental one (100%). As the TRE is increased, some corrupted executions eventually become tolerable and, then, the AVF will be reduced accordingly. A small TRE is sufficient to significantly reduce the AVF, indicating that most of the errors have a small impact on the output value.

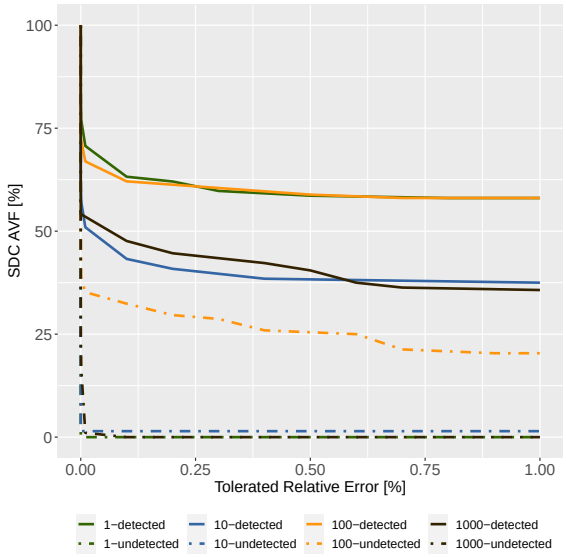
Data in Figure 5.6 demonstrates that, for all codes, the critical error rate reduction is much faster for the undetected faults (dashed lines in Figure 5.6). This indicates that undetected faults have a much lower impact on the output correctness than the impact the detected errors would have. Even with a TRE as small as 0.1%, all the undetected errors

Figure 5.6: TRE for the microbenchmarks, ADD 5.5a, MUL 5.5b, and FMA 5.5c. Figure 5.5d shows the TRE for MXM, Lava, FWT, and BlackScholes.

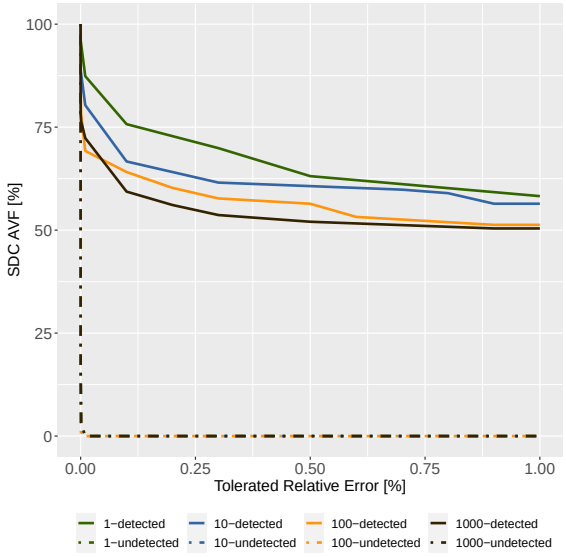
(a) ADD



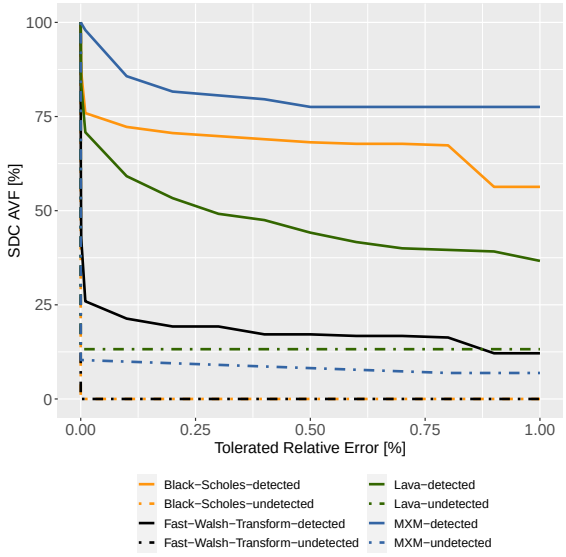
(b) MUL



(c) FMA



(d) Benchmarks



would be tolerated for all the micro-benchmarks (but MUL with 100 ops) while at least 60% of the detected errors would still be considered critical. This means that the errors RP-DWC is unable to detect have an impact on the output value that is lower than 0.1%. For MUL, as said, it is necessary to use a smaller input to avoid overflow in the FP32 replica. A small error might then induce a higher TRE. However, with a TRE of less than 5% (not shown), all the undetected errors for MUL would be considered tolerable. For MXM, Lava, FWT, and BlackScholes, shown in Figure 5.5d, the trend is maintained, but the impact of undetected faults is slightly higher (∼90% of the undetected errors modify

the output of less than 1%). This is because MXM, Lava, FWT, and BlackScholes include several instructions of different nature (i.e., ADD, MUL, FMA, and other instructions). It is worth noting that in a traditional DWC (not shown), the detected and undetected error effects in the application output overlap as the detection is independent of the relative value of the error.

5.3.8 Impact of Undetected Errors in HPC and Safety-Critical Applications

While both the detected and undetected errors are considered different from the expected FP64 result, data reported in Figure 5.6 shows that the RP-DWC technique detects those errors that are more likely to generate a significant deviation from the application output. Previous works have demonstrated that small fluctuations in the output value (up to 4%) have a low or negligible impact and can, potentially, be even tolerated for several HPC applications such as particles or physical simulation, weather forecast, heat distribution, wave propagation, earthquakes prediction, etc. (GODDEKE; STRZODKA; TUREK, 2007; GRAND; GöTZ; WALKER, 2013; PUENTE et al., 2014). The realistic codes tested (MXM, LavaMD, FWT, BlackScholes) are among these HPC applications. As shown in Table 5.1 and Figure 5.5, RP-DWC, while being more efficient, has a lower error detection rate than traditional DWC (of $\sim$15% for fault injection and $\sim$9% for beam experiments). Nevertheless, as shown in Figure 5.6, even considering a fluctuation in the output value (i.e., a TRE) of 1%, more than 90% of the undetected errors with RP-DWC are considered tolerable (99% for the micro-benchmarks). As a result, almost all the RP-DWC undetectable errors, according to (PUENTE et al., 2014), can be tolerated in several HPC applications.

Even for some safety-critical applications, such as CNNs for objects detection, small variations in the output values are unlikely to induce critical faults (LI et al., 2017) (see Section 4.1). A fault is critical for a CNN when it induces a misdetection. On the other hand, a fault that only slightly modifies the CNN output but *does not* impact detection will not cause vehicle misbehaviors and, thus, can be considered tolerable. An additional fault injection campaign is run to evaluate the percentage of detected vs. undetected errors that are critical for CNNs. The analysis focus on RP-DWC for MXM, as more than 70% of operations in a CNN are related to matrix multiplications and, as shown in Section 4.1, most of CNN errors are caused by MXM corruptions. To understand if the undetected errors would cause critical errors for a CNN, a fault injection campaign is per-

formed on YOLOv3 processing frames from VOC2012 dataset (REDMON; FARHADI, 2018; EVERINGHAM et al., 2012). The output of a random matrix in a random layer of YOLOv3 is modified. First, a relative error of 1% is injected as, according to Figure 5.6, the vast majority ($\sim$90%) of MXM errors RP-DWC is unable to detect have a TRE lower than 1%. To consider the worst-case scenario for RP-DWC, *all the elements* or *a row of elements* of the matrix are corrupted. *None* of the 1,000 injections induced misdetections in YOLOv3. This result confirms that the faults RP-DWC is unable to detect have a negligible impact even in CNNs, which is not sufficient to cause misdetections. Then, when a relative error is injected in the range 1% to 100% and about 12% of these errors (i.e., the ones RP-DWC is highly likely to detect) induced misdetections.

It is possible to derive that, while RP-DWC has a lower detection rate than a traditional DWC, only $\sim$10% of the errors RP-DWC is unable to detect (i.e., the ones with a TRE higher than 1%) can potentially be critical for HPC or safety-critical applications. If the other $\sim$90% of undetectable errors are considered as actually tolerable errors, then the detection rate of RP-DWC increases from the 55%-83% (see Table 5.1) to 95%-98%, which is comparable with the traditional DWC. The decision to consider or not the tolerable errors and the tolerance threshold depends on the application. In any case, RP-DWC shows a better cost-benefit trade-off (overhead-detection) than traditional DWC and could be particularly useful for HPC applications.

6 FAILURE IN TIME ESTIMATION

This Chapter presents one of the main contribution of this work, the Failure In Time (FIT) rate estimation combining kernel profiling, fault injection, and error rate. The results help to understand the impact of hidden GPU resources (parallelism management, scheduler, dispatcher, queues, etc.) and identify the code/architecture characteristics/metrics that have a significant impact on the GPUs error rate. Finally, the Chapter presents a case study about the differences of an analysis made strictly on the software level and the proposed error rate estimation that also considers the hardware level.

6.1 FIT rate prediction through fault simulation

It is possible to assume that the cause of the observed code output error is one and only one original neutron strike (that can produce a single or multiple bit-flip, as evaluated in Section 6.3) in one and only one resource. This is justified by the observation that, with the current technology and the low intensity of natural flux of particles, it is possible to consider negligible the probability for more than one neutron to generate a fault during one code execution. The cross-section of a bit of a 28nm SRAM cache memory, for instance, is in the order of 10^{-16} - $10^{-17}cm^2$ (BAGGIO et al., 2004). As the flux of neutron at sea level is $13neutrons/(cm^2 \cdot h)$, the fault rate of a memory bit is in the order of 10^{-15} - $10^{-16}errors/h$. Even on a hypothetical GPU with 1GB of these SRAM internal memories (caches, shared memory, etc.), it would be expected to see at most $1 \times 10^{-6}errors/h$, making it unlikely for more than one fault to occur during one application execution. Additionally, all neutron-induced events are transient and not cumulative: the sensitivity of a resource does not depend on the number of faults it has undergone (BAUMANN, 2005). This guarantees that a code can be affected only by a corruption that has happened during its execution, independently of what happened in previous computations (unless the code uses a corrupted output as input, in which case the error must be attributed to the previous computation and should not be counted in the code FIT rate).

Neutron-induced errors are then uncorrelated and stochastic events. Thus, a device's probability of being corrupted by a neutron is equal to the sum of the probabilities of having a neutron-induced corruption in one of its resources, i.e., the probability of disjoint events ($P(Event_{resource_1} \cup Event_{resource_2} \cup \ldots\ Event_{resource_n})$). Con-

sequently, the Cross-Section of a code (and informally the FIT rate) is the sum of the probabilities of having a neutron-induced fault in each of the resources used for its computation multiplied by the probability for the fault in that resource to propagate and manifest at the output (the resource AVF), i.e., the probability of independent events ($P(Propagation_{resource_1} \cap Propagation_{resource_2} \cap \ldots\ Propagation_{resource_n})$).

Knowing the AVF and FIT rate of every resource used for computation, in principle, would allow a perfect estimation of the FIT rate of a code. Unfortunately, even if each GPU resource were accessible by the user, it would be unfeasible to measure the FIT and AVF of each resource since the GPU is a very complex device. It has been decided to limit this study to the contribution of GPUs' main functional units and memories. Beam experiments measured the FIT rates of most common functional units (arithmetical micro-instructions), register file, and shared memory (details in Section 6.3). Then, the probability for a fault in each micro-instruction or used memory to affect the code output is calculated through fault injection.

The estimated FIT rate of a code (†FIT) can be calculated adding the expected contribution of each micro-instruction $P(E_{INST_i})$ and memory level $P(E_{MEM_i})$ to the code error rate, as shown in Equation 6.1.

$$\dagger FIT = \sum_{i=1}^{n} P(E_{INST_i}) + \sum_{i=1}^{m} P(E_{MEM_i}) \tag{6.1}$$

The contributions to the FIT rate of the code, $P(E_{INST_i})$ and $P(E_{MEM_i})$, as said, depend on the number of resources used for computation, the probability of a fault to be generated (the resource cross section or FIT), and the probability for the fault in that resource to affect the computation (AVF) as formalized in the following Equations 6.2 and 6.3.

$$P(E_{INST_i}) = f(INST_i) \cdot AVF_{INST_i} \cdot FIT_{INST_i} \tag{6.2}$$

$$P(E_{MEM_i}) = f(MEM_i) \cdot AVF_{MEM_i} \cdot FIT_{MEM_i} \tag{6.3}$$

Where $f(INST_i)$ and $f(MEM_i)$ are the probability of having one instance of a micro-instruction $INST_i$ or a bit of memory level MEM_i used in the computation of the benchmark, AVF_{INST_i} and AVF_{MEM_i}, FIT_{INST_i} and FIT_{MEM_i} are the AVF and FIT of a micro-instruction $INST_i$ and a bit of memory MEM_i, respectively. The FIT_{INST_i} and FIT_{MEM_i} are informally used as the probability of the error in the hardware in a given re-

source, as the FIT rate in this case is obtained from multiplying the $\sigma[cm^2]$ (the hardware probability) by a constant flux (13×10^9).

On a GPU, as on any other computing device, $f(MEM_i)$ is the number of bits of memory level i instantiated for computation. It is worth noting that, when ECC is enabled in memory MEM_i and if all memories have ECC it is possible to assume $AVF_{MEM_i} \approx 0$ and as consequence $P(E_{MEM_i}) \approx 0$, simplifying Equation 6.1 to only its first summation.

$f(INST_i)$ is the percentage of instructions of the type $INST_i$ to be executed in the code. On a GPU, the FIT rate has the peculiarity of varying significantly based on the code degree of parallelism and on how the GPU scheduler can allocate the available functional units. The next section discusses a possible way to consider the GPU peculiarity using kernel profiling.

6.2 Profiling kernel dynamic instructions

On GPUs, the probability for a neutron to corrupt an operation inside a thread depends on how many threads are active and how many parallel operations are being executed. The higher the number of instructions a thread is allowed to schedule or the higher the number of active threads in an SM, the higher the number of functional units that can be corrupted. To understand how many computing resources are exposed and should be considered in Equation 6.2, it is necessary to profile the codes to measure the code parallelism.

The smaller threads working group in a GPU is a warp (i.e., 32 threads). The alive warps in an SM can be in three different states: stalled, eligible (ready but waiting), and selected (executing). Each SM has four warp schedulers that select the instructions from warps based on their state. Then, each scheduler picks the eligible warp to execute up to 2 instructions, limiting the instruction issue parallelism to 4 per SM. The number of cycles a warp requires to be ready to execute the next instruction is the latency of the warp. If the latency cannot be hidden, the resources will be underutilized. During latencies, the alive threads are exposed and could be corrupted. Thus, it is necessary to consider the number of active warps (i.e., the *Achieved Occupancy* in NVIDIA profiling tools) to model the number of functional units that could be corrupted.

The Achieved Occupancy alone is not sufficient to model the number of resources used on GPUs. The number of active threads could be limited by resource utilization (commonly, the amount of registers and shared memory). If the instructions in these (few)

active threads do not have dependencies, they could be scheduled in parallel, saturating the available functional units. This is the case of GEMM, for instance, that has a very low occupancy (see Tables 3.1 and 3.2) but imposes massive stress in the functional units. Other codes (as sort) have high occupancy but suffer from long latencies. This work also considers the *Instructions Per Cycle (IPC)* of the code in the prediction model to account for (un)efficient utilization of resources. A high IPC means that a thread can execute a high number of instructions, implying that it has the allocation of a high number of functional units (increasing the number of resources used for computation that can be corrupted).

To consider the contribution of parallelism of GPUs, then, $P(E_{INST_i})$ is multiplied in Equation 6.2 by a factor (φ_{INST}) defined as follows:

$$\varphi_{INST} = AchievedOccupancy * IPC \tag{6.4}$$

High occupancy and a high IPC indicate that many resources are employed for computation. The lower the occupancy and the IPC, the lower the resources used for computation.

6.3 Synthetic Micro benchmarks

As described on Equation 6.2 the model needs the FIT_{INST_i} and FIT_{MEM_i} for the FIT estimation. To measure the FIT rate of the functional units and main atomic instructions of Kepler and Volta architectures, seven classes of synthetic micro-benchmarks have been designed for this work. Reported results are used for predicting the FIT rates of codes and are valuable to compare the reliability of the different funcional units that compose GPUs architecture.

6.3.1 Micro-benchmarks profile and error rate

Errors are identified by comparing the result with the pre-computed fault-free output when each thread's operations have been completed. For errors at each operation are not checked to avoid excessive overhead and make the probability of a fault in the comparison negligible. However, if two different neutrons corrupt two different instructions during the 10^8 operations, it would be counted as one error underestimating the functional

unit FIT rate. Nevertheless, in each hour, it has been observed <10 events. Considering that an average of 2 seconds is necessary to execute 10^8 operations, in the worst case of having each of the errors generated in the first of the 10^8 operations, the probability of having two corruptions is lower than 1%. Also, more than one thread corrupted (more than one sequence of operations with incorrect data) has happened in less than 1% of corrupted execution. If each instruction is checked, all the errors would be caught but, as only after a sequence of operations are checked, some errors might be masked, reducing the FIT rate. To consider the masking effect of subsequent operations, a fault injection on the micro-benchmarks were run and found that the AVF is always higher than 70% (being 100% for integer). To have the most accurate prediction, in Section 6.1, the FIT rate of micro-benchmarks by their AVF are multiplied.

Figure 6.1 shows the SDC and DUE normalized FIT rate measured with beam experiments for all micro-benchmarks on Kepler and Volta. FIT rates are normalized and shown in arbitrary units (a.u.) to allow comparison without revealing business-sensitive data. FMA, ADD, MUL, and MAD are tested, both with ECC ON and OFF and found that the error rates are comparable (differences lower than 20%). In fact, these mico-benchmarks use very little memory (few registers).

Figure 6.1: Micro-benchmarks experimental FIT rates, normalized to each device's lowest measured value: FADD's DUE on Kepler, HFMA's DUE on Volta. Memories values are also normalized by the minimum memory FIT for each architecture.

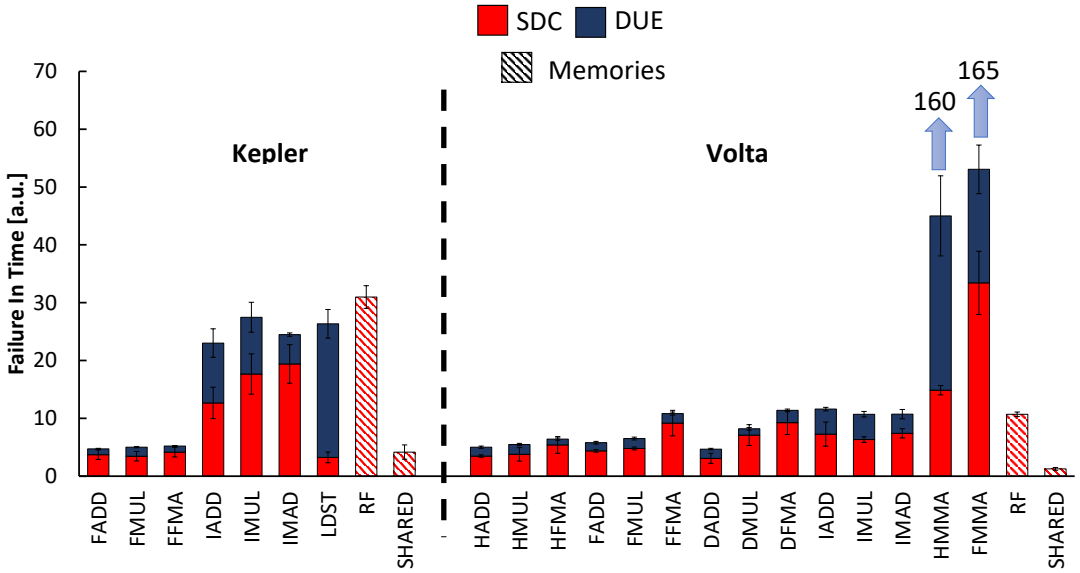


Figure 6.1 shows that for all the tested float instructions (FADD, FMUL, FFMA) on Kepler, both SDC and DUE rates are very similar. When the instructions are executed

using INT32, the FIT rate is, on average, 4× higher than FP32. This is probably because the integer operations are executed in the same hardware as the FP32 operations with evident lower efficiency that can increase the vulnerability. The INT32 error rates vary according to the type of instruction, following the operation's complexity. IMUL's FIT rate is approximately 30% higher than that of IADD. These results suggest that an integer addition is less complex and demands less hardware than integer multiplication. Since IMAD performs integer multiplication and addition, its FIT rate is higher than both IMUL (10% higher) and IADD.

LDST is the only micro-benchmark for which the DUE rate is higher (7.1×) than the SDC rate, which is expected because the critical operand in the LDST micro-benchmark is a memory address. An incorrect address can either be valid or invalid. The likelihood of a corrupted address to be valid is low if the total memory allocated is small, which is the case. Hence, the chances of invalid addresses, such accesses trigger a device or CUDA API exception, is higher.

Figure 6.1 also shows the SDC and DUE error rate for Volta micro-architecture. This work focus on mixed-precision cores reliability as this is the crucial novelty in GPU architectures. The differences in the FIT rates between int, double, float, and half precision operations in Figure 6.1 rely on the different Volta mixed-precision cores' complexities. Since a multiply requires more resources than an addition, its FIT rate is expected to be higher, and FMA (fused multiply and addition) is expected to have FIT rate higher than ADD and MUL, which is following the results. Additionally, the higher the operation precision, the higher the FIT rate (again, higher precision implies more resource utilization). It is worth noting that, dissimilarly to Kepler, integer operations on Volta are executed on dedicated cores. The FIT rate depends on the complexity of the hardware resources.

Figure 6.1 also shows the FIT rate of micro-benchmark focussing on Matrix Multiplication and Addition (MMA), also known as *Tensor Core*, for Volta architecture (MMA is not available for Kepler). Hardware MMA operations can be used via CUDA on $16x16$ input matrices. The introduction of specific (sophisticated) hardware to execute matrix multiplications in mixed-precision was driven by the importance of this operation in DNN training and interference.

As the complexity of MMA is higher than those of other functional units and so is the utilization, its FIT rate is expected to be higher than all other micro-benchmarks on Volta. As shown in Figure 6.1, Half and Float MMA have FIT rates that are 12× higher

than that of DFMA, which has the highest FIT rate among the others micro-benchmarks. It is worth noting that, as one and only one neutron can generate an error in the 10^7 or 10^8 operations each thread performs (details in Section 3.4), the number of operations each thread executes does not impact the FIT rate. A higher number of sequential operations increases the number of errors because it increases the execution (i.e., exposure) time and, thus, the neutron fluence, not because the hardware is more vulnerable. If more operations are executed in parallel, the FIT rate is expected to increase as more operations are performed but roughly the same execution time.

One interesting observation is that, while being more sensitive, the MMA core performs, in one operation, the equivalent of $4x4$ FMA or $4x4$ ADD, MUL, and the loop control variables needed to implement MxM in software (these latter instructions can economize with a loop-unrolling). From the data, the FIT of each HMMA and FMMA micro-benchmarks is $9\times$ and $12\times$ higher than an FMA micro-benchmark (it is worth recalling that FMMA uses the HMMA core after a cast). As 64 MMA instructions are required to multiply two $16x16$ matrices, and for each warp-wide MMA instruction, it is feasible instead execute a warp of 32 FMAs, it is possible to deduce that the use of MMA is $2\times$ (64/32, where 32 is the number of threads in a warp) more reliable than the combination of operations needed to execute a software MxM. The use of MMA eliminates repeated fetches of the multiply-and-add operations and reduces activity in instruction memory and pipelines. As the size of the matrix multiplication supported by the MMA increases the reliability (and performance), the benefit will also increase.

Figure 6.1 shows the error rate of a byte in the Register File (RF) and Shared memory (SHARED), also normalized by the minimum FIT of each memory. The result clearly shows RF as the most critical memory resource at the SM level on a GPU. This result confirms previously published research (Goncalves de Oliveira et al., 2016). RF and SHARED are implemented with different interleaving, which affects the probability of MBUs. The experiments show that the MBU rate is less than 2% for RF and less than 0.9% for SHARED. While the Kepler vs. Volta FIT rates are not shown (different normalization for the two boards in Figure 6.1 were used), it is possible to see that the fabrication process plays a significant role. Kepler RF (28nm planar) has an approximately an order of magnitude higher error rate than Volta RF (16nm FinFET). The difference between planar and FinFET error rate is a known and documented phenomenon (NOH et al., 2015).

6.4 Beam vs Fault injection

After having detailed the reliability characteristics of Kepler and Volta architectures and several codes, the codes' FIT rates measured with beam experiments are compared with those predicted with fault simulation and profiling. Following the methodology described in Section 6.1, to predict the codes FIT rate the fault-injection (details in Section 4.3), application profiling, and beam experiments on functional units (details in Section 3.3) are combined. This comparison's main scope is to evaluate at which level a reliability analysis based on fault simulation can be considered realistic. It is good to recall that on Kepler, both SASSIFI and NVBitFI are tested, and on Volta, only NVBitFI, as SASSIFI is not supported. Moreover, on Kepler, neither SASSIFI nor NVBitFI supports fault injection on NVIDIA proprietary libraries such as cuDNN and CUBLAS. For the codes based on NVIDIA libraries (GEMM and YOLO), the AVF measured with NVBitFI on Volta is used for the Kepler prediction. As discussed in Section 4.3, NVBitFI and SASSIFI provide SDC AVFs that differ, on average, by $\sim$18%. The difference then slightly reduces the accuracy of the prediction for GEMM and YOLO on Kepler.

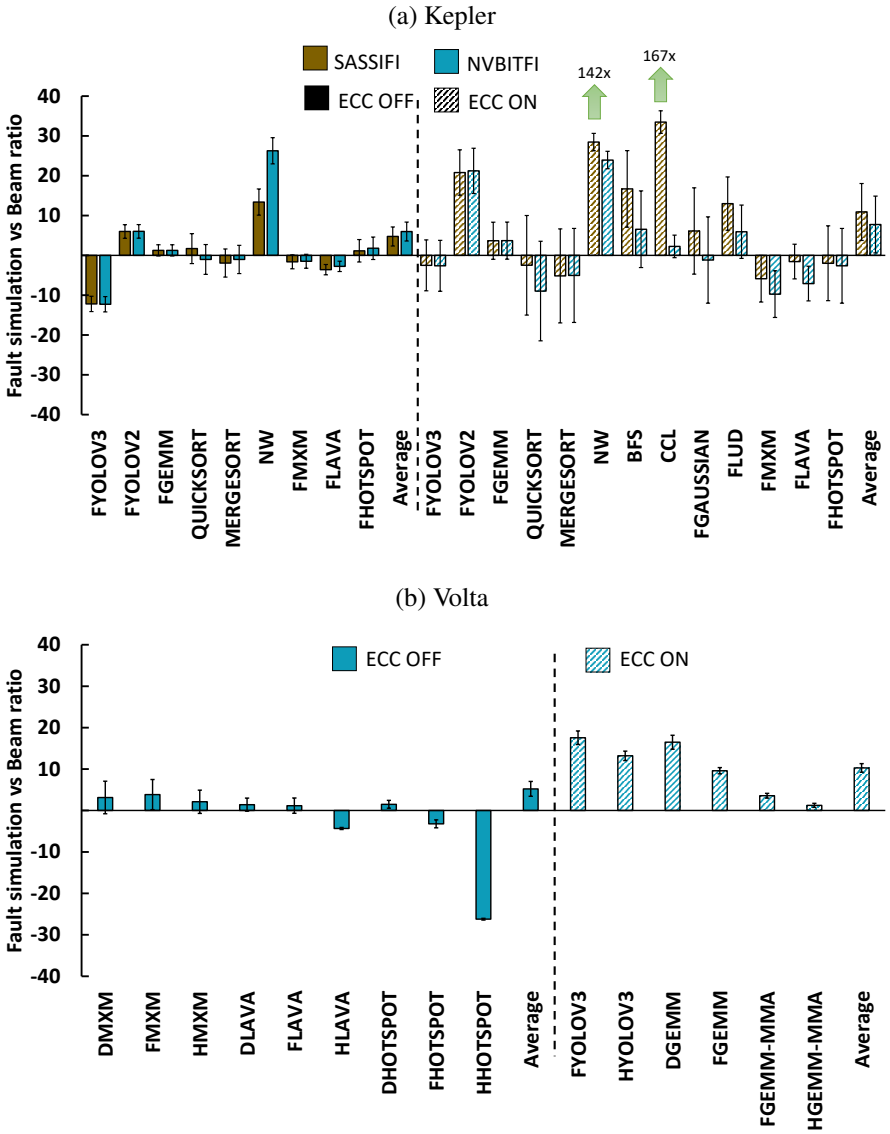
6.4.1 SDC

Figure 6.2 shows the comparison of the codes SDC FIT rate measured with beam experiments and predicted with fault-injection. To ease the visualization of the comparison, for each code, the highest SDC FIT rate between the one measured with beam experiments and the one predicted using fault injection is divided by the lowest SDC FIT rate between the two. Whenever the fault injection SDC FIT rate is higher than the beam one, the value are represented as negative, positive otherwise. For instance, on the Kepler with ECC disabled executing FYOLOv2 beam experiments report a 12$\times$ higher FIT rates than fault injection, for FYOLOv3 fault simulation predicts a FIT rate 7$\times$ bigger than the one experimentally measured.

A promising result highlighted in Figure 6.2 is that despite the simplifications fault simulation introduces, in most cases, the SDC FIT prediction is reasonably close to the SDC FIT measured with the beam. The *absolute average* difference between fault simulation and beam experiments on Kepler is 5$\times$ for SASSIFI and 6$\times$ for NVBITFI, with ECC disabled. When ECC is enabled, on Kepler, the average difference is 11$\times$ for SASSIFI and 8$\times$ for NVBITFI. On Volta, the average is 5$\times$ when ECC is disabled

and 10× when ECC is enabled. As this work compares completely different evaluation strategies, it is reasonable to believe these differences to be extremely promising.

Figure 6.2: Comparison between the SDC FIT rate measured with the beam and predicted with fault injection.



For 25 out of 38 configurations, the fault injection underestimates the SDC FIT rate. One limitation of the model is that not all resources are accessible for fault simulation, preventing from considering all the possible sites for errors. As discussed in Section 6.3, only the most common micro-instructions are contemplated as testing all 20 instructions types is unfeasible. While the considered micro-instructions cover more than 70% of instructions that compose the codes (see Figure 3.1), it is still possible that some errors in the unconsidered micro-instructions generate an error, and this would only count in beam experiments. When ECC is disabled, the prediction model will consider the mem-

ory error rate (Eq. 6.1) that has already been shown to dominate GPUs' FIT rate (Haque; Pande, 2010). On average, when ECC is disabled, fault injection can better predict the beam SDC FIT rate, as the contribution to the FIT rate of the not modeled functional units and instructions is much smaller than the memory contribution.

For some outliers (NW and CCL on Kepler, HHotspot on Volta), the fault-injection based prediction is very different from the beam. The kernels used for NW and CCL are not well suitable for GPUs, and they underuse the available resources and have poor memory access patterns (Table 3.1). These inefficiencies may be the reason to reduce the possibilities of having corruptions in functional units (as they are not stressed) and increase the error rate due to other sources of errors, like threads and memory management. The proposed model does not consider these sources of errors yet, resulting in a poor underestimation for not well-parallelized codes. HHotspot on Volta overestimation (fault injection FIT rate is $27\times$ higher than beam FIT rate) relies on the impossibility to inject faults in half-precision functional units (intrinsic limitation of NVBITFI). The float functional units AVF is also used for the half precision. This simplification is acceptable for most codes (HGEMM and HLava prediction is sufficient) but not for HHostspot, probably because of its intrinsic characteristic of iterating the computation that smooths faults value (Oliveira et al., 2017).

As a final remark on SDC estimation, it would be good to to emphasize that, while the FIT rates of functional units through beam experiments are measured, as shown in (Chatzidimitriou et al., 2019) on ARM CPUs, an estimation of the FIT rate based on micro-architectural models can also be sufficiently precise. The proposed model could then be applied to predict the fault rate of codes executed in future GPUs once the micro-architectural model is available. Despite the model's intrinsic limitations, it can successfully provide a good FIT rate estimation on average, even when ECC is enabled.

6.4.2 DUE

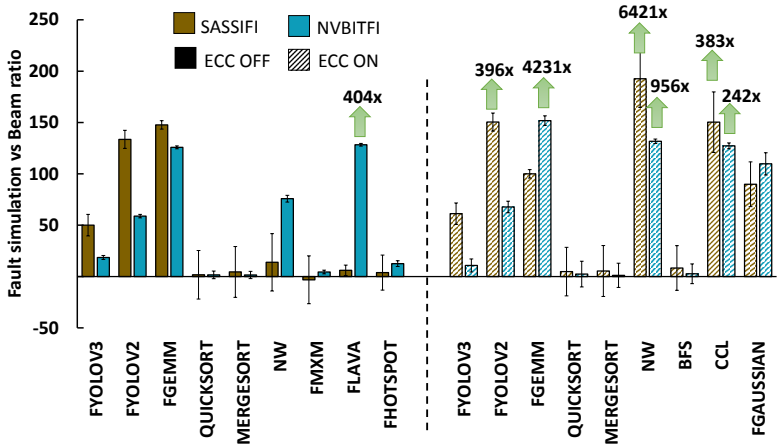
The analysis can be used to derive exciting insights on the origins of DUEs as well. DUEs can be caused by several factors: including interrupts triggered by ECC, a corruption during device-host synchronizations, illegal memory accesses, corruption in the hardware scheduler, changes in the program flow (such as corruption in the instruction cache or the jump destination address), or faults in hardware resources that stuck the device (more details in Section 6.4.2.1). Most of these causes for DUEs are independent

of the arithmetical operation executing and are not modeled by the proposed prediction strategy.

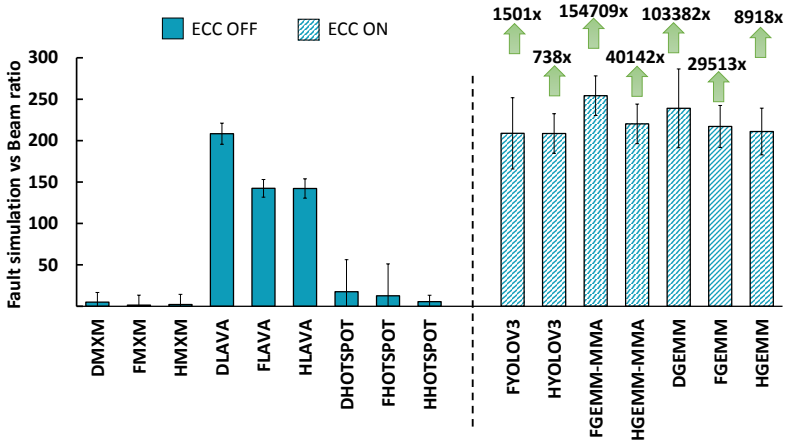
In this work mainly the arithmetical functional units, memories, and Load/Store instructions of GPUs are characterized with beam experiments. In the prediction model, only a subset of the causes for DUEs are included, and a significant underestimation of the code DUE FIT rate is to be expected. Figure 6.3 shows that on the average, the beam DUE FIT rate is 120× higher than the predicted DUE FIT rate for Kepler ECC off, 629× for Kepler ECC on, 60× for Volta ECC off, and 46,700× for Volta ECC on. This high divergence attest that a large portion of DUEs does not come from arithmetic micro-instructions and modeling micro-instructions and memories are not sufficient to predict the GPU DUE rate.

Figure 6.3: Comparison between the DUE FIT rate measured with the beam and predicted with fault injection.

(a) Kepler



(b) Volta



6.4.2.1 DUE source

For a subset of the codes presented on 3.2, the DUE events have been traced and logged using the CUDA Runtime API (NVIDIA, 2021). It is then possible to deeply evaluate the DUEs' causes. Due to beam time limitations, not all codes are assessed.

Below is listed all the observed DUE outcomes under the neutron beam (Adapted from (NVIDIA, 2021)). It is worth noting that only the GPU was exposed to the beam, the CPU and the memories were not exposed. Consequentially, all the errors listed are directly caused by events on GPU, and they are observed at least once in the beam experiments.

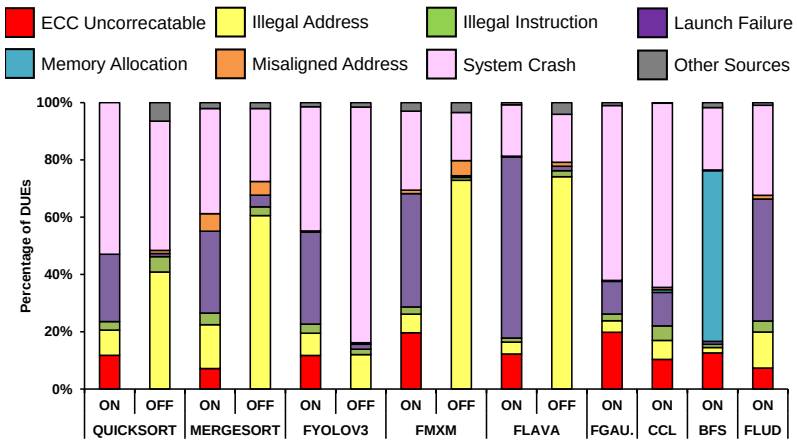
- **Devices Unavailable:** The device became unavailable both when the GPU mode became Exclusive or Prohibited, or the previous kernel execution did not release the used resources, and the device is "falsely" full.
- **Illegal Instruction:** When an illegal instruction happens during the kernel execution, leaving the process in an inconsistent status.
- **Illegal Address or Misaligned Address:** The address used in a Load or Store instruction does not point to a valid memory address or is not aligned.
- **Initialization Error:** The CUDA driver failed to initialize, making it impossible for the API continues.
- **Invalid Device or No Device:** The selected device by the CPU is invalid, or the CUDA driver detects no device.
- **Invalid PC:** One of the kernel program counters got corrupted consequentially became invalid.
- **Invalid Address Space:** A CUDA kernel can operate in different spaces of memories (i.e., global, shared, or local). This error happens when an instruction that belongs to a specific memory space try to operate in a different one.
- **Memory Allocation:** The CUDA API is not able to allocate memory on GPU.
- **System Crash:** It is a DUE triggered by the setup software or hardware watchdog. Then the DUE source could not be traced or is generally unknown.
- **ECC Uncorrectable:** When the Error Correction Code detects more than 1 bit of flip in the memories, it cannot be corrected.
- **Launch Failure:** This error occurs when the CPU tries to launch a kernel and fails. There are many reasons that it could happen, such as dereferencing an invalid device pointer and accessing out of bounds shared memory, etc.

- **Hardware Stack Error:** This error happens due to an error in the call stack during kernel execution, generally due to stack corruption or exceeding the stack size limit.
- **Invalid Value:** This indicates that some parameters passed to the GPU are out of the acceptable range of values. For example, more threads than the GPU supports.

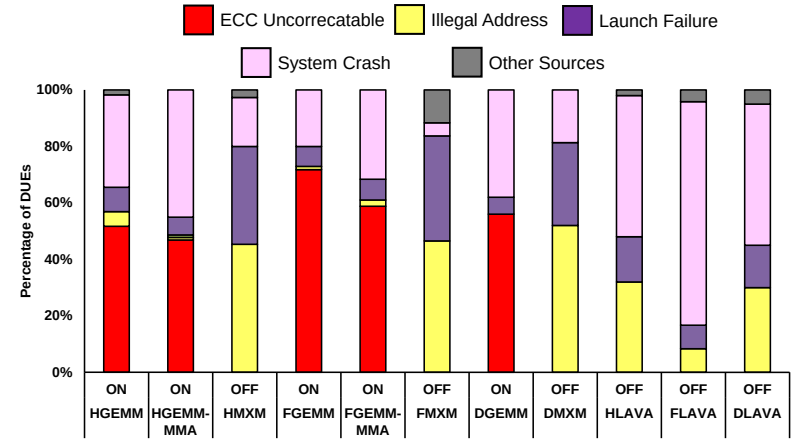
Figure 6.4 shows, in percentage, the sources of DUEs have been identified in the beam experiments for Kepler and Volta. The DUEs that comes from Devices Unavailable, Invalid Value, No Device, Initialization Error, Hardware Stack Error, and Invalid Device have been grouped under the category "Other Sources" as, on average, the combination of these types of events caused less than 3% of DUEs in the experiments.

Figure 6.4: Detailed DUE sources for Kepler and Volta GPUs.

(a) DUE source for Kepler



(b) DUE source for Volta



For Kepler and Volta, the System Crash is the leading cause of DUEs. System Crashes are still the most manageable outcome of transient errors on GPUs. However, tracing System Crashes sources is very hard, as most events are generated by exceptions from the operating system or the hanging kernels inside the GPU killed by the watchdog.

On Kepler and Volta, it is possible to observe an explicit behavior caused by enabling ECC. In fact, when ECC is disabled, the DUEs come mainly from Illegal Address (i.e., trying to access an incorrect memory address), which is an expected phenomenon since all the Register File (RF), shared, and cache memories are unprotected, including the ones that store the memory addresses for Load/Store instructions. A corruption in the memory addresses is likely to violate the memory policy and be translated into an Illegal Address. On the other hand, when the ECC is turned on, protecting the RF, shared, and cache memories, there is a significant reduction of the average of Illegal Address DUEs (less than 8% on Kepler, and less than 2% on Volta) and a consequent increase in probability DUE to be caused by ECC Uncorrectable errors. It is reasonable to believe that the errors that caused IllegalAdreess DUEs when ECC is off migrated to ECC Uncorrectable DUEs when ECC is on.

Section 4.2 have shown that the FIT rate for ECC on DUE is, on average, 30% bigger on Kepler. Thus, it is expected that ECC Uncorrectable errors happened more often than the other types. For instance, on Kepler, Illegal Address happened 1.8× less than ECC Uncorrectable errors, on Volta, Illegal Address happened 37× less than ECC Uncorrectable, when ECC is on.

Volta GPUs have support to Matrix Multiplication to be performed on hardware, though MMA instruction. Figure 6.4b shows an exciting behavior for GEMM that uses MMA instructions when compared to the traditional software Matrix Multiplication, composed of a sequence of Fused Multiply and Add (FMA) instructions. There is a reduction for ECC Uncorrectable DUEs of 10% for HGEMM-MMA and 22% for FGEMM-MMA compared with common HGEMM and FGEMM, respectively.

It is worth remembering that MMA instructions are Single Instruction Multiple Data (SIMD). MMA instructions operate directly in 16x16 matrices, while standard FMA, used for general matrix multiplication, operates only in three registers. As expected, fewer address calculations are necessary, consequentially fewer registers to store the addresses, reducing the ECC Uncorrectable errors on MMA's GEMMs.

Launch Failure is a DUE source directly related to how the kernels manage the resources necessary for computation, such as global, shared, and local memories. The error is caused by accessing invalid memory pointers, out of bounds shared memory and system-related configurations. For instance, Quicksort on Kepler, each thread can also launch other parallel kernels using NVIDIA Dynamic parallelism. So, each thread can generate at least one Launch Failure error on Quicksort.

6.5 Case Study: Measuring the Compiler Impact on Reliability with SDC rate estimation

Until now, it has been demonstrated that the SDC estimation can provide a reasonable estimation based on fault simulation and profiling data. However, as shown on the AVF results in Section 4.3 the compiler can have a high impact on the final estimation. It is worth remembering that the software fault injection is done on the SASS code generated by the compiler. Consequentially, it is necessary to study how the compiler can bias the error analysis at the software level. Additionally, it is possible to compare the SDC estimation with other metrics commonly used for error evaluation.

This Section presents a case study on how the compiler impacts the reliability evaluation. Also, the effects of the optimization flags applied at the NVCC Parallel Thread Execution (PTX) compiling phase are analyzed. This Section uses the SDC rate prediction presented before and another metric that only considers software level, the SDC probability (the AVF weighted by the instruction percentage). Multiple NVCC optimizations for the two compiler versions are used (NVCC 10.2 and 11.3), and the impact of each optimization using both the SDC probability (only software) and the SDC error rate estimation are compared.

6.5.1 Optimization flags and compilers

A subset of benchmarks listed in Section 3.2 is selected to be evaluated on Kepler and Volta GPUs. The codes are compiled with two NVIDIA CUDA Compiler (NVCC) versions, NVCC 10.2 and 11.3. The select versions of NVCC are the latest ones that simultaneously support Kepler and Volta architectures. Each major update on NVCC introduces support to newer architectures and compiler optimizations, which can be beneficial or not to reliability.

All the GPU kernels are compiled with multiple NVCC compilation flags to measure the impact of each optimization on the final Source and Assembly (SASS) code. The flags that generate a SASS code that differs from default NVCC optimization (O3) are selected. It is expected that two identical codes will have the exact fault propagation probabilities and the same FIT rate.

The following optimization flags are used in the evaluation:

O0, O1, and O3: The optimization levels available on NVCC for the tested benchmarks.

Although O2 is supported for NVCC, the O2 option generates the same code as O3 for the tested applications. All flags, but O0 and O1, are related to float approximations or register file usage. The approximation flags are tested on top of the default NVCC configuration (03).

FTZ-ON: The *–ftz=true* flag on NVCC flushes *subnormal* numbers (i.e., values smaller than the smallest possible value) to zero. Faults that generate subnormal numbers will probably be masked with FTZ-ON.

FMAD-OFF: The default NVCC optimization is to contract the multiply and accumulate instructions into FMAD, FFMA, or DFMA. To disable the contraction of the multiply and accumulate instructions, it is necessary to pass *–fmad=false* as a parameter. The contracted instruction may generate a different result due to destructive cancellation, so it is expected that the impact of a fault will be different on contracted instructions.

MinRF: Each thread on a CUDA kernel can use up to 255 registers, respecting the limits of the GPU occupation. With the flag *–maxrregcount=N* the maximum number of registers per thread can be set, where N is the limit of registers per thread. For this work, the number of registers per thread is set to the minimum for each architecture, 16 for Kepler, 24 for Volta. Limiting the number of registers per thread can be used to increase the GPU occupation. However, it also increases the register spill to the memory, which can decrease the performance.

PrecSqrt-OFF and PrecDiv-OFF: The flag *–prec-sqrt=false* and *–prec-div=false* allow the NVCC to use a fast approximation for the square root and float divisions, respectively. Float approximations improve performance but may change the impact of the fault in the final result.

FAST-MATH: Passing the flag *–use_fast_math* to NVCC enables all the fast approximations for arithmetic float operations. That is, all the flags *–ftz=true*, *–prec-div=false*, *–prec-sqrt=false*, *–fmad=true* are set in the compilation process.

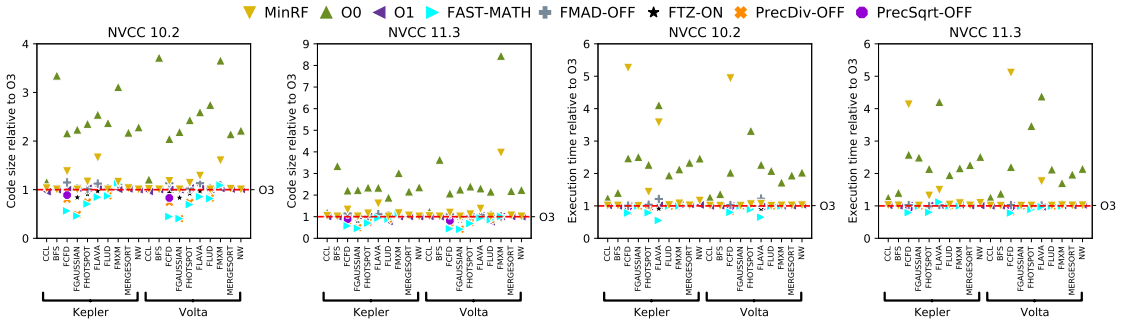
6.5.2 Preliminary analysis: Dynamic instructions profiling

It has been shown that the fault masking is directly related to the basic blocks organizations and instruction dependencies of the code (ANWER et al., 2020). Consequently, a code compiled with the O0 flag can mask more faults than a code compiled with O3, as the unoptimized code (O0) has more dead code and redundant instructions. As a consequence, a fault in a redundant instruction output will be masked (Ashraf et al.,

2017). However, the unoptimized code will take much longer to execute than the optimized one. Thus, it is necessary to consider the code size and the execution time in the reliability analysis of different compiler optimizations.

Almost all the benchmarks listed in Section 3.2 have been built with all the flags and compilers described previously. Figure 6.5 shows the code size and the execution time relative to the default NVCC compilation (O3). The code size is measured by counting the kernels SASS lines compared with the default NVCC configuration. The O3 configuration is marked as a red line in the figure to represent the relative size for each configuration.

Figure 6.5: Code size and execution time relative to default NVCC compilation for versions 10.2 and 11.3



Approximation is one of the most intuitive examples of the trade off between performances and accuracy. Reducing the hardware core iterations to execute a floating-point operation, e.g. using the *fast math* option, is a common type of approximation. While it generally reduces the accuracy of the result, it can reduce the code size and the execution time, as shown in Figure 6.5. From a reliability perspective, approximation improves performance and can reduce the error rate (smaller area and/or fewer operations executed). However, the impact of the fault in the approximate result can be higher as fewer bits are used to represent the output. The probability for the fault to propagate (AVF) can then be higher.

When compiler flags that limit the optimizations are applied (MinRF, O0, and FMAD-OFF), the generated code is larger, and execution time tends to be higher (Figure 6.5). When the code is compiled with limited/no optimizations, the generated SASS is not reorganized for performance or optimized memory accesses. Similarly, when the register file usage is limited, the compiler inserts many register spills instructions to compensate for the reduced registers per thread. The MinRF code size has, on average, 23% more instructions than the only O3 code.

The execution time follows the same trend as code size. The optimizations that perform approximation on the float instructions have a lower execution time than purely

O3 compiled code. In fact, the FAST-MATH flag can reduce 40% of the execution time for some codes. Contrarily, O0 and MinRF have the highest execution time. The unoptimized code (O0) significantly reduces the instruction-level parallelism and reduces the memory accesses performance, increasing the latency of the instructions. Equivalently, for the MinRF version, the register spills necessary for reduced register file usage have a high cost in terms of execution time. The instructions must wait for the operands that are not in the register file.

Even if an unoptimized code has a lower error rate than an optimized one, it may not be beneficial in terms of the amount of work produced before experiencing a failure (i.e., Mean Work Between Failures (MWBF)). As show in Figure 6.5 an incorrect applied optimization can be 5× slower than the optimized one (O3). Section 6.5.5 shows that the optimized versions have higher MWBF than the unoptimized ones.

6.5.3 SDC probability

As discussed in previous works (YIM et al., 2011; FENG et al., 2010; LI; PATTABIRAMAN, 2018; PALAZZI et al., 2019; ANWER et al., 2020), the instruction AVF alone does not represent the probability of a given instruction selected from the code to generate an SDC at the end of the execution. It is also necessary to consider the probability of the instruction to be picked on the fault injection campaign and generate an SDC. As the fault simulation is performed with multiple NVCC optimizations and with two different compiler versions, it is expected that the instruction distribution will be changed. To consider the optimization impact in the fault propagation, the instruction distribution is also considered on the fault propagation analysis, measuring the SDC probability (LI; PATTABIRAMAN, 2018), as show in equation 6.5.

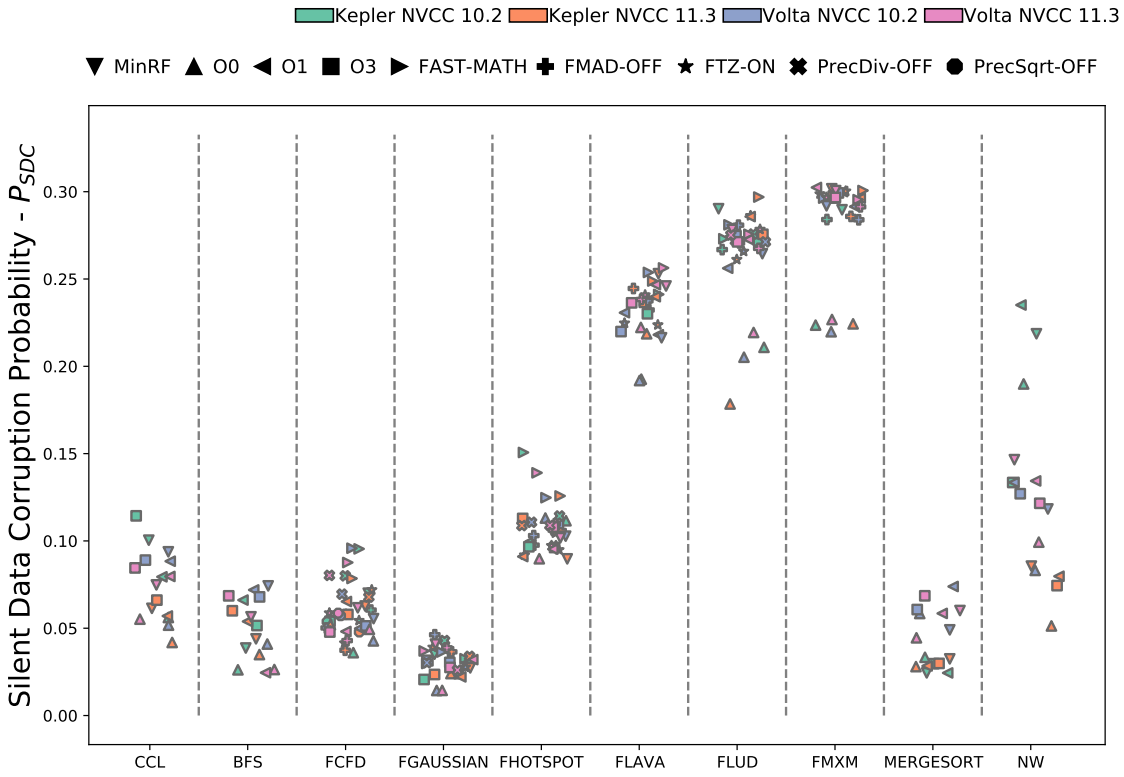
$$P_{SDC} = \sum (AVF_i * \frac{N_i}{N_{total}}) \tag{6.5}$$

Where AVF_i represents an AVF for a given instruction i, and $\frac{N_i}{N_{total}}$ represents the probability of an instruction i in the total instruction count (the percentage of instruction i).

Figure 6.6 shows the SDC probability (P_{SDC}) distribution (vertical axis) for all the evaluated codes (horizontal axis). The results are shown for two GPUs, Kepler and Volta, with two NVCC versions. For each device and compiler, the flags present in Section 6.5.1

are tested. As some codes are mostly composed of NVIDIA closed libraries and can not be optimized, not all codes from Section 3.2 can be compiled with different flags.

Figure 6.6: Silent Data Corruption Probability ($P_{(SDC)}$) distribution. Based on (LI; PATTABIRAMAN, 2018)



For the majority of the benchmarks, the results show only a slight variation in the SDC probability. The *Coefficient of variation* (i.e., how much the values are different from the mean) is on average 15%. Most of the codes, but CCL, BFS, and NW, present a SDC probability for the most optimization flags near the default configuration (O3). This suggests that the error rate should not change much despite a high modification in the executed machine code. This may be counterintuitive and can be caused because SDC probability does not consider a significant part of the reliability equation, i.e., the hardware fault probability. The analysis that will be proposed in the following subsection and the experimental validation proposed in Section 6.5.5 verify this statement.

CCL, BFS, and NW have coefficients of variations of 31%, 35%, and 50%, respectively. CCL, BFS, and NW are benchmarks that have a low AVF compared with the other codes. Additionally, CCL, BFS, and NW are naive and not tuned for performance implementations. Consequently, the benchmarks instruction distribution is very concentrated in load/store and MISC (i.g., sync and NOP) instructions. The reliability evaluation of load, store, and MISC instructions is challenging, and it is hard to determine their sen-

sitivity by software fault injection. Thus, the SDC probability may not reflect a realistic scenario.

The compilation flag that produces the lowest SDC probability is the unoptimized code (O0). The probability for the fault to be propagated (AVF) is lower for O0. The codes built with O0 have useless basic blocks, or the output register is overwritten, increasing the fault masking. On the other hand, the MinRF, FAST-MATH, and FTZ-ON have the highest SDC probabilities. When the number of registers is limited, the criticality of each register increases, as the compiler will continuously optimize to use all registers available. As the number of instructions is reduced for the approximation flags, the fault impact in an approximated instruction is expected to be higher at the software level. Obviously, SDC probability does not consider that a lower number of registers and approximation reduces the probability for the hardware fault to be generated.

Each functional unit has a specific probability of being corrupted. Consequently, if an optimization changes the code instructions distribution, the AVF (evaluated by the SDC probability) and the instruction's contribution to the final error rate will change. It is mandatory to consider both the functional units error rate and the performance of the code to estimate a more accurate error rate. The hardware/software analysis is presented in the following subsection.

6.5.4 SDC rate estimation

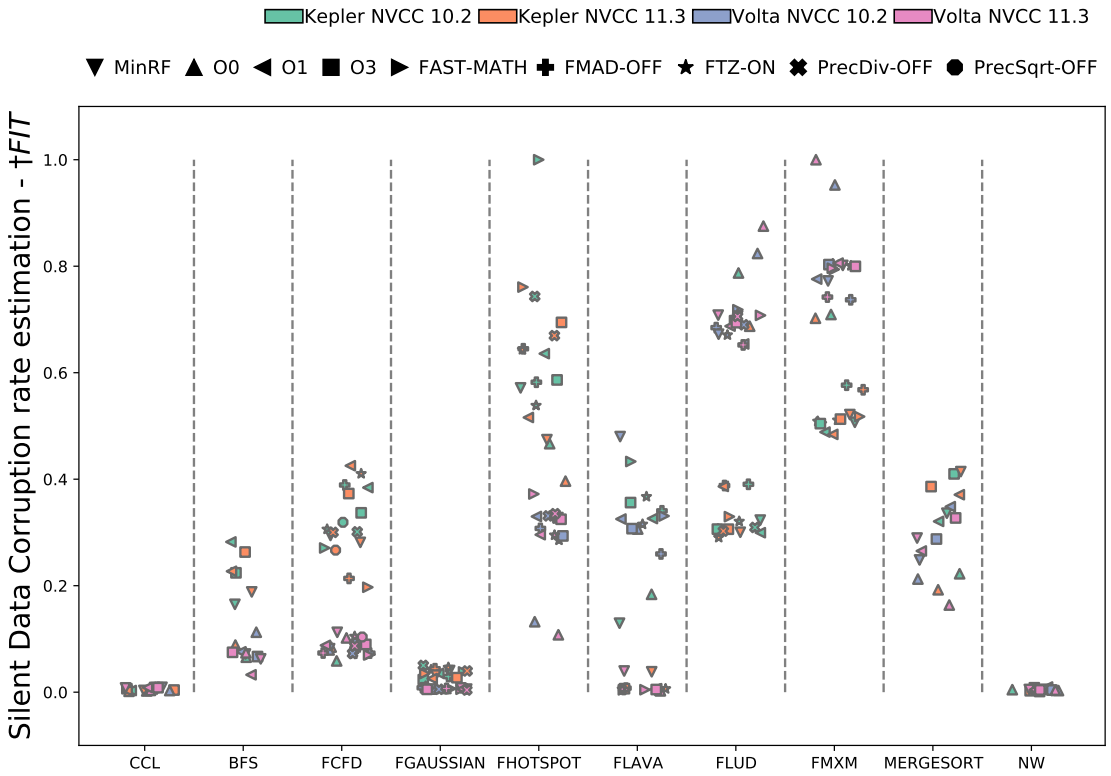
Figure 6.7 shows the SDC rate estimation (using the same approach present on Section 6.1) for all configurations used in the SDC probability experiments. The values in 6.7 are normalized by the highest estimated SDC rate for each board (i.e., FHOTSPOT FAST-MATH 10.2 for Kepler, and FMXM O0 11.3 for Volta). It is worth noting that the SDC probability and SDC rate estimation consider only errors on the functional units and the output registers. The caches and shared memories error rate are not considered for SDC probability or the SDC rate estimation. This scenario would be comparable to the ECC ON on a real device.

The average Coefficient of Variation for the SDC rate estimation is 33.6% for all the configurations present in Figure 6.7. The highest variations come from FLAVA, 98% for Kepler and 103% for Volta. From the experiments, FLAVA is the most computing-intensive code, 75% of the instructions are float arithmetic. FLAVA error rate is directly connected to the functional units' error rate and how they are used. If a flag configuration

or a compiler generates a code that changes the IPC or the instruction distribution, the SDC rate will be directly modified.

CCL, BFS, FGAUSSIAN, and NW have the lowest estimated SDC rate (the normalized value is near 10^{-3}). The low SDC estimation for FGAUSSIAN is due to the low AVF related to the instructions that compose the code. Comparatively, the FGAUSSIAN also has the lowest SDC probability. On the other hand, CCL, BFS, and NW have a low SDC estimation due to very low IPCs and GPU occupation. This may present a methodology limitation, as the error rate estimation is normalized on performance metrics. Still, for codes that perform poorly on performance metrics, it is necessary to evaluate other instruction types on the GPU, such as branch, sync, and load/store from multiple levels of cache. That is, the analysis must also evaluate the instructions that reduce the performance of the kernel (i.g, instructions that create stalls in the pipeline) to increase the accuracy of the estimation.

Figure 6.7: SDC rate estimation ($\dagger FIT$) distribution. Based on the methodology presented on 6.1



NVCC 10.2 vs. 11.3: As expected, for the SDC probability, the differences between the compilers are not significant. The SDC probability ratio between NVCC 10.2 and 11.3 is on average 10%. Contrarily, for the SDC rate estimation, the NVCC 10.2 produces an estimation on average 7× higher than NVCC 11.3. In fact, the SDC rate

estimation differences between NVCC 10.2 and 11.3 are not homogeneous as the SDC probability. That is, a subset of the benchmarks has very high differences between the compilers. While the differences between the NVCC 10.2 and 11.3 in the 75% quartile of the experiments are only 25%, they can be $16\times$ higher in the last quartile.

It has been shown that the optimization flags may change the error rate of an application. It is worth noting that the software analysis provides similar SDC probability for most configurations, which can be inaccurate as an estimator for the error rate. The differences between the optimization flags appear only when the hardware error rate is added to the analysis. A subset of configurations evaluated with beam experiments will be analyzed in the next section to help to demonstrate this observation.

6.5.5 Validation through beam experiments

Beam experiments are performed to validate the analysis proposed above and to show that a purely software fault injection is not sufficient to evaluate compiler optimizations effects on the code error rate.

Figure 6.7a shows the **Failure in Time (FIT)** rate for the Matrix Multiplication (MXM) running on a Kepler GPU under a neutron beam. To avoid revealing business-sensitive data, the FIT rate is normalized by the smallest value (i.e., MXM compiled with O0 when ECC is ON). The data is presented with ECC enabled (ECC ON) and disabled (ECC OFF) to evaluate also the impact on the error rate of the register file, caches L1/L2, and shared memories.

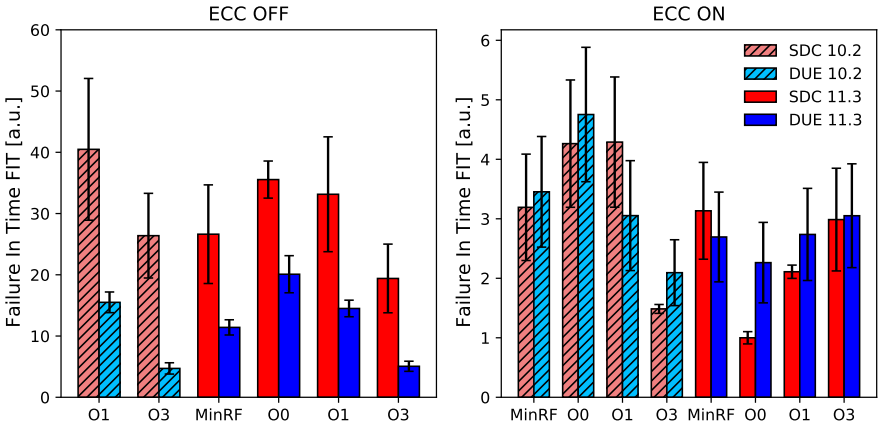
On average, the optimization flags applied to MXM have a Coefficient of Variation between them that is 28.6%. Compared with the variations measured with the two methodologies shown above, the SDC probability has a variation for MXM of 8.7%, and the SDC rate estimation has a variation of 54%.

When the ECC is OFF, the SDC FIT rate is one order of magnitude higher than when ECC is ON. Another interesting aspect is that the SDC FIT rate is always higher than the DUE FIT rate when ECC is OFF while when ECC is ON, the DUE rate is similar to or higher than the SDC rate. This is because a double-bit flip detection triggers an exception, that leads to a crash.

Table 6.1 shows the ratio between the NVCC 10.2 and 11.3 FIT rates. When ECC is OFF, the results show that the SDC rate for NVCC 10.2 is on average 30% higher than NVCC 11.3 SDC rate. For ECC ON, the results show that the SDC and DUE for

Figure 6.8: Error rate for FMXM compiled with different configurations running on a Kepler GPU.

(a) Normalized Failure In Time (FIT) rate for CUDA 10.2 and 11.3



(b) Mean Work Between Failures for CUDA 10.2 and 11.3

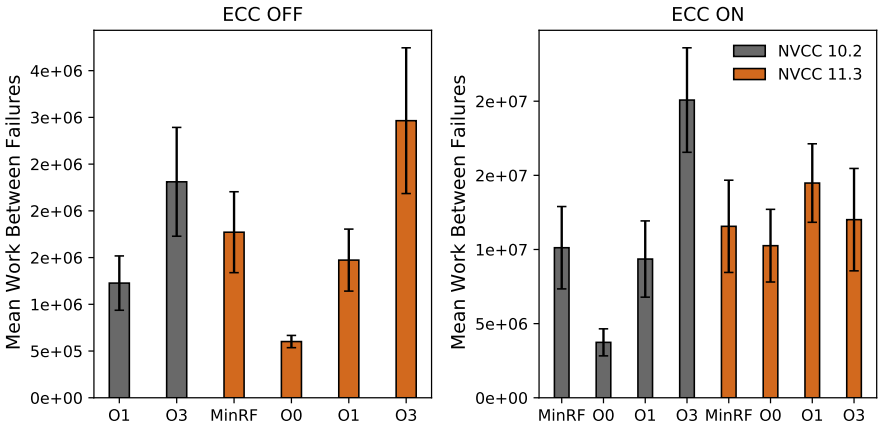


Table 6.1: NVCC Ratio between NVCC 10.2 and 11.3. When the ratio is higher than 1, NVCC 10.2 FIT rate is higher than NVCC 11.3.

		SDC	DUE
ECC OFF	**O1**	1.2	1.1
	O3	1.4	0.9
ECC ON	**MinRF**	1.0	1.3
	O0	4.3	2.1
	O1	2.0	1.1
	O3	0.5	0.7

NVCC 10.2 are on average 95% and 30%, respectively, higher than NVCC 11.3. These results follow the observations presented in Section 6.5.4 and attest that just considering the software fault injection would lead to imprecise estimations. The MXM build with NVCC 10.2 with the O3 uses the GPU resources better (including registers) and has a better performance than O3 NVCC 11.3 version. Consequentially, the FIT rate is lower for NVCC 10.2 with O3.

Table 6.2: Ratio between the estimated SDC FIT rate and the SDC FIT rate obtained from beam experiments.

		ECC OFF	ECC ON
NVCC 10.2	**MinRF**	–	1.3
	O0	–	1.3
	O1	1.1	1.9
	O3	-1.6	-1.6
NVCC 11.3	**MinRF**	-1.1	1.3
	O0	-1.6	-3.3
	O1	-1.3	-1.1
	O3	-2.1	1.2

Table 6.2 shows the ratio between the SDC rate estimation and the SDC rate obtained on beam experiments. The results are following the data presented on Section 6.4. That is, the estimated SDC rate is very similar to the obtained in a realistic environment.

6.5.6 Mean Workload Between Failures

To improve the discussion proposed in the FIT analysis, considering the performance gain brought by GPU code optimizations, it is necessary to use a metric that correlates reliability with performance. The **Mean Work Between Failures (MWBF)** is measured for each tested configuration. The MWBF is defined as the amount of correct data produced by the system before experiencing a failure (SDCs and DUEs) (REIS et al., 2005). The MWBF is calculated by multiplying the number of executions between failures by the workload of the application. A higher MWBF rate means the system can process a bigger workload without experiencing errors. It is possible to determine if a specific optimization or a compiler upgrade generated codes that can produce more useful data before experiencing an error.

Figure 6.7b shows the MWBF for the float matrix multiplication (FMXM) for all the configurations tested on the neutron beam experiments. Despite increasing the error rate, flags that improve the performance also increase the MWBF of the application. In fact, the optimized version can process more data before experiencing a failure. The differences in the FIT rate for the O3 version for NVCC 11.3 reflect in the MWBF. The NVCC 11.3 version uses fewer registers than the 10.2 one. Consequently, the O3 version with the NVCC 11.3 has a higher MWBF when ECC is OFF. However, when the ECC is ON, the O3 for NVCC 11.3 increases the FIT rate by 60% and keeps the performance. The result is that the NVCC 10.2 compiled with O3 when ECC is ON has the highest

MWBF from all configurations. Additionally, the ECC OFF version has a MWBF one order of magnitude lower than the ECC ON. Even with the DUE increase, the ECC ON version is more reliable and delivers more useful data than the ECC OFF.

It has been shown that the minor modifications on the compilation, like a slight increase in the register file usage, can impact the application's reliability. The many test configurations present similar reliability on the analysis based solely on software-level metrics (SDC probability). However, it has been shown unrealistic when the hardware factors are applied to the investigation (SDC rate estimation). The software level analysis is still a reasonable estimator of the code fault propagation. But, it does not consider hardware factors such as instructions latency, GPU occupation, and IPC, which have been demonstrated to impact the final error rate.

6.6 Considerations on FIT estimation

In this chapter, a methodology to estimate the FIT rate for GPUs is presented. Most of the time, the estimation is satisfactory. Even considering the outliers, the SDC prediction average stayed up to 12×. Additionally, in a case study, the FIT rate estimation is compared with another metric that considers only software-level analysis (SDC probability).

Although the FIT estimation can give a satisfactory result, the results showed that the most common error model might not be the best approximation for fault injection to simulate beam acceleration tests. The next chapter will present a new error model based on the RTL fault injection to try to fill the gap between the beam error model and the software error model.

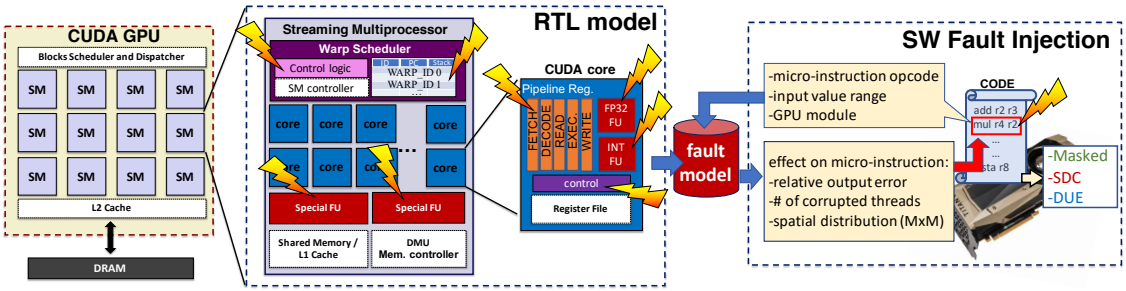
7 IMPROVED FAULT SIMULATION ERROR MODEL

This Chapter presents a new error model for GPUs as one of the significant contributions of this work. The RTL fault simulation results are shown, then the new error model for GPU's fault simulation is proposed.

7.1 Overview of the Idea

The GPUs' proposed reliability evaluation framework is divided into two main steps: RTL fault simulation and software fault simulation (Figure 7.1).

Figure 7.1: Scheme of the proposed two-level fault simulation framework. Using the RTL model the effects that faults in GPU modules (the fault injection is not performed in the modules depicted as white boxes) have on the micro-instructions output are characterized. Based on the micro-instruction, its input, and the module of interest the fault model (syndrome) is injected in software on a real GPU that executes a code.



Using a GPU RTL model (details in Section 3.5.2), faults in the GPU main computing modules are simulated. Pipeline Registers, Warp Scheduler, FP32 and INT functional units, Special Function Units (SFUs), and control signals are considered. Errors in memories (caches and register file) are not considered as it is assumed that GPUs employed in applications with strict reliability requirements feature ECC. Moreover, as a fault in a memory cell(s) affects a software's visible state directly (it translates into a corrupted value with no further operations), its syndrome is already well known (single/double bit-flip) and depends just on the memory technology (BAUMANN, 2005). On the contrary, a fault inside a computing resource during an operation's execution has a not-obvious impact on the output (*syndrome*) (SUBASI et al., 2018), which is intended to characterize.

Rather than executing an application in the RTL model, the effect of faults in a subset of GPU ISA *micro-instructions* is characterized. Micro-instructions are the simplest, atomic, two-input machine operations that form the compiled code. A micro-instruction

is directly translated into hardware signals inside the device. The characterized micro-instructions are based on GPU code profiling shown in Section 6.2. The codes used for RTL fault injection follow the same approach described in Section 6.3. The only difference is that the micro-benchmarks used on the beam tests have 10^8 sequentially operations, while the micro-benchmarks used in RTL fault injection have only one operation. Since there is no time constraint on fault simulation, there is no need to use more than one operation. Additionally, the goal of fault simulation on RTL is to evaluate the functional unit module and not a device as on beam experiments.

The operations evaluated are: Float point (FADD, FMUL, FFMA - Fused Mul and Add), Integer operations (IADD, IMUL, IMAD - Mul and Add), Trascendental functions (SIN, EXP), Load/Store (GLD, GST), Branch (BRA), and Integer set predicate/register (ISET). This work also evaluates a *mini-app (tiled MxM)* to highlight possible scheduler corruption effects that could be hidden in the SASS instructions characterization (details in Section 7.2).

A perfect RTL fault injection would require one to test each micro-instruction with the exact input values it receives when executed in the code being characterized, which is unfeasible. Then the analysis is reduced to three input ranges Small, Medium, and Large. Small range are numbers near to zero (i.e., between 10^{-5} to 10^{-6}), Medium are numbers between 10^1 to 10^3, and Large are numbers with exponent bigger than 10^5. Previous work has shown that software fault injection results for GPUs do not depend on the input value (with unbiased values) (Previlon et al., 2018). Part of this work contribution is to understand if this result still holds for RTL fault injection and how much the fault effect on the instruction output depends on the input value.

With the RTL fault simulation, both the probability for the fault to reach a visible software state (i.e., the Architectural Vulnerability Factor, AVF (MUKHERJEE et al., 2003)) and the fault impact on the instruction output value have been measured. A database of possible fault syndromes has been built based on the micro-instruction opcode, the input range, and the injection site (the corrupted module). To quantify the syndrome, a statistical distribution of the *relative difference* between the expected and the observed corrupted micro-instruction output has been shown. In other words, it is tracked how much in percentage the fault has modified the micro-instruction(s) output. The syndromes, the number of corrupted threads, and the spatial distribution of wrong elements (for tiled-MxM) populate a database used for the software fault injection, available at (SANTOS et al., 2021).

To simulate the RTL fault syndromes in software, the NVBitFI has been updated (details in Section 3.5). The updated version of NVBitFI extracts the most suitable fault syndrome from the RTL fault injection database to apply, considering the opcode and input range. Once the instruction output is corrupted, the code execution continues, and the effect on the output is characterized as SDC, DUE, or Masked. With modified NVBitFI, then, the probability for the faults that reached a visible software state to propagate further till the application output is measured (i.e., the Program Vulnerability Factor, PVF (SRIDHARAN; KAELI, 2009)).

The benefit of the proposed strategy relies on the fact that the detailed and time-consuming RTL evaluation on the SASS instructions is done only once to populate the syndromes database. The software fault injection maintains its efficiency (thus allowing the assessment of complex applications). Still, it provides both extra accuracy, by using the RTL syndromes, and impact, as it is possible to the observed SDCs with their hardware source.

7.1.1 Contributions and Limitations

This work proposes, for the first time for GPUs, to combine the fine-grain evaluation of *RTL fault injection* with the flexibility and efficiency of *software fault injection in real GPUs*. As characterizing realistic codes with RTL fault injection is unfeasible, it is necessary to limit the RTL analysis to common GPU *SASS instructions*, gathering the syndrome induced by faults in the instruction output value, i.e., an accurate fault model for the most common machine operations is produced. As all GPU modules are accessible in the RTL model, it is possible to provide deeper insights into GPU faults source and characterize the effects of the fault on multiple threads. Then, using an updated software framework (NVBitFI), the syndrome from the RTL analysis is injected (rather than a simplistic fault model as all previous works on GPU software fault injection do). The high speed of software fault injection allows us to observe the effect of fault syndromes in the execution of real-world applications. In contrast, the few previous works on GPU RTL fault injection are limited to naive workloads. While the proposed strategy can effectively allow a more detailed and accurate GPU reliability analysis, it is worth noting some intrinsic limitations:

1. RTL is not the lowest possible abstraction layer (see Figure 2.1). RTL fault injection

is chosen because the circuit or gate models are not available for GPUs, and their characterization would, in any case, take too long. As shown in previous work, though, RTL evaluation accuracy is very close to gate level simulation (Kochte et al., 2010).

2. The proposed evaluation shares with any other research work based on open-source models the limitation of being based on mature architectures. FlexGripPlus, the only RTL open-source GPU model currently available, is based on NVIDIA G80 architecture. While it is impossible to guarantee that the observed fault syndrome is representative of cutting-edge GPUs, the G80 is still CUDA compliant. It is based on the same Instruction Set Architecture (ISA) of modern NVIDIA GPUs such as Kepler, Volta, and Turing (except for tensor core and a few other instructions based on updated modules). The hidden structures of a GPU, such as the scheduler and the pipeline registers, are also supposed to be present even in modern architectures. Given the CUDA compatibility, the decision is made based on the only RTL description available, even if it is from a different generation. Also, while probably the FlexGripPlus intrinsic limitation might impact the precision of the evaluation, it does not undermine the impact of the proposed strategy, which is directly adaptable to other GPU RTL models as they become available.
3. The syndrome imposed by a fault could depend on the operation input. Testing all inputs combination is impossible. Then, it was decided to limit the characterization to three input ranges.

7.2 RTL fault injection results

As described in Figure 7.1 the first step to the proposed idea is to perform RTL fault simulation. This section describes the process and discusses the results of the micro-instructions AVF. The fault simulation is performed in 6 GPU modules characterizing, for most modules, 12 micro-instructions with different input sets (3 ranges and 4 values per range for arithmetic operations). In total, 144 RTL fault-injection campaigns are performed and, for each campaign, more than 12,000 faults are injected. That is, data is presented from more than $1{,}72 \times 10^6$ fault injections. This guarantees a statistical margin error lower than 3%. RTL fault injection experiments were performed at *Politecnico di Torino* in a research collaboration between both universities.

Figure 7.2 shows, for simulations in Functional Units (FP32, INT, SFU), Warp Scheduler, and Pipeline Registers, the AVF of each micro-instruction. Injections in functional units for GLD, GST, BRA, and ISET have not been considered as the FUs are idle when executing those micro-instructions. In Figure 7.2 it is possible to distinguish between SDCs affecting single or multiple threads. The AVF does not significantly depend on the input range (the AVF difference between S, M, and L inputs is always lower than 5%), in accordance with (Previlon et al., 2018). In Figure 7.2 only the average AVF measured with the three input ranges has been shown. Section 7.2.1 shows that the fault syndrome does depend on the input range.

Figure 7.2: AVF of the injections at RTL level on the functional units (FP32, INT, SFU), the scheduler, and pipeline registers for the different micro-instructions. The average AVF measured with the S, M, L input ranges is plot.

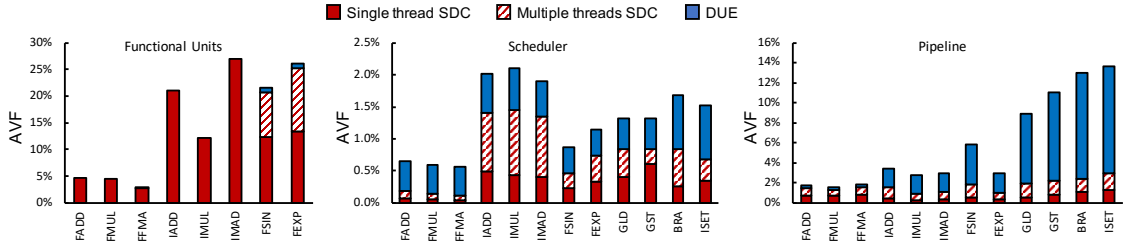


Figure 7.2 shows that faults in the scheduler are less likely to impact the computation than faults in the functional units or pipeline (the y-axes are on different scales). It is reasonable to recall that in the micro-benchmarks, threads do not interact with each other, reducing the scheduling strain. Moreover, the functional units corruptions are much more likely to generate SDCs than DUEs, while DUEs mainly dominate the outcome of injections in the pipeline. The observed behaviors are further investigated.

More than 60% of the SDCs caused by scheduler corruptions affect more than one thread for the INT and FP32 micro-benchmarks, while injections in the functional units cause multiple threads corruption only FSIN and FEXP. This is because the GPU has a dedicated ADD, MUL, and MAD unit for each thread while the few (two) available special function units (SFUs) need to be shared among different threads (see Figure 7.1). A more in-depth analysis of the multiple SDCs sources revealed that faults cause the multiple corrupted threads observed with functional units corruptions in FSIN and FEXP are in the control units of the SFUs. Interestingly, also pipeline injections cause multiple threads corruption. Investigating the causes for those multiple threads, it has been found that, while most of the pipeline registers (≈84%) store operands for each parallel core, there is also a tiny portion of registers (≈16%) devoted to controlling signals. The

corruption of these latter registers caused the observed multiple threads SDCs.

On average, the number of corrupted parallel threads per warp is 1 for INT and FP32 functional units, 8 for the SFUs, 28 for the scheduler, and 18 for the pipeline. These averages show that the parallel operation's modules in the GPU, such as the scheduler and the pipeline, are more prone to corrupt a high number of multiple threads in a warp than others. A fault in the control structures and signals of the pipeline and, mainly, of the scheduler (which manages the warp operation) affects multiple threads. The lower number of threads corrupted in the pipeline is related to the number of available FUs and active threads at a given time (8 in our case). As some signals are not updated until a new warp is dispatched, their corruption affects, on average, two of the four groups of 8 threads in a warp (32 threads).

The high DUE AVF for the pipeline (0.3% for floating-point, 2% for integer, 4% for special units, 10% for control operations) is caused by corruptions of pipeline control registers, despite being few (≈16%), are confirmed to be highly critical. The DUE AVF is exacerbated for special function units because of the additional control signals required to share the few available SFUs and control flow operations (GLD, GST, BRA, ISET), a fault in the data path registers can corrupt the control flow.

The scheduler DUE AVF is almost constant (between 0.5% and 0.6%) for all micro-instructions but BRA and ISET, for which the DUE AVF is about 0.8%. Tracking the fault propagation, scheduler DUEs are caused by faults affecting structures in the controller devoted to storing the state of the warp or memory addresses. In contrast, the scheduler SDCs are mainly caused by faults affecting warp state bits, disabling active or enabling inactive threads.

As observed in Figure 7.2, the functional units AVF for the floating-point instructions (FADD, FMUL, FMAD) is much smaller than for the integer instructions (IADD, IMUL, IMAD). This is caused by the higher complexity and area of the floating point units, which are more than 3x larger than the integer units. A larger area increases the number of injection sites, thus reducing the probability of selecting a critical computation resource.

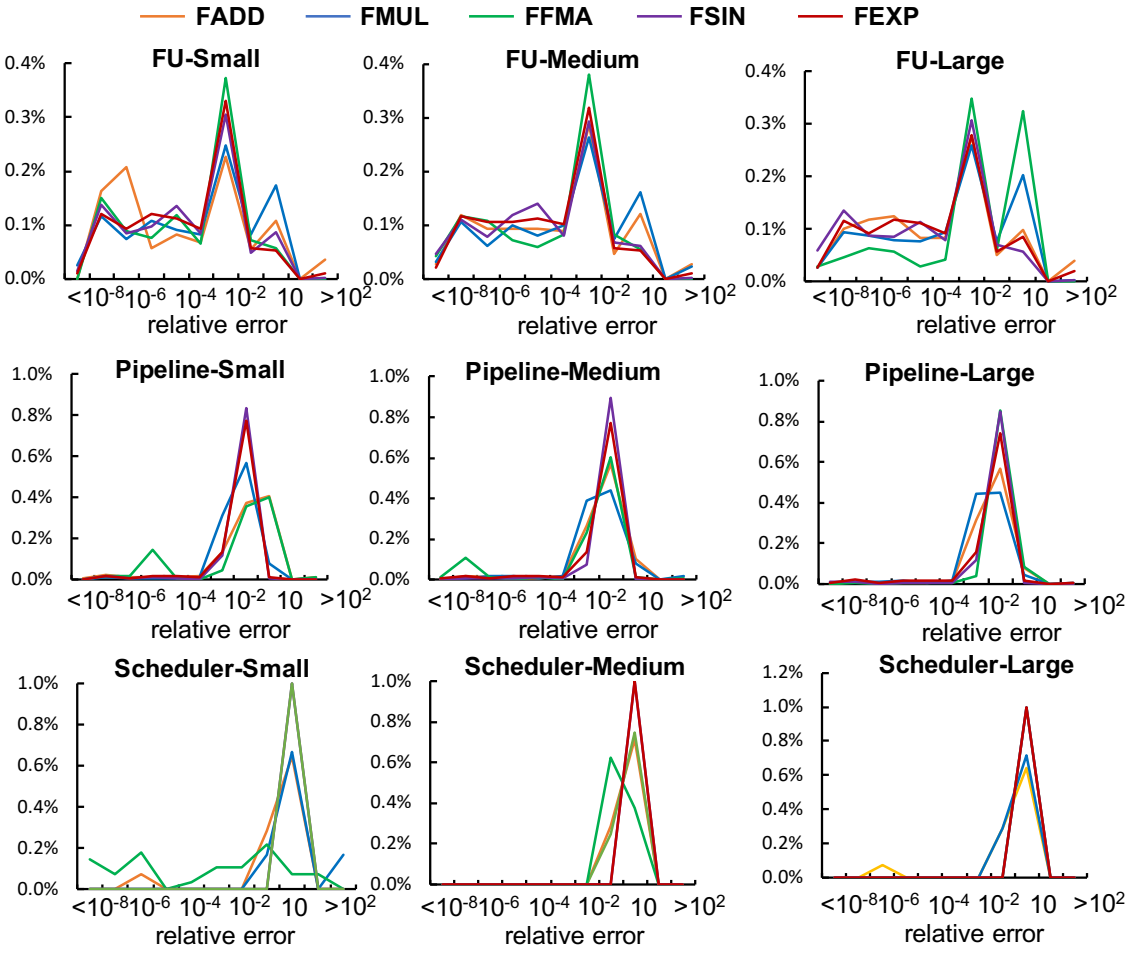
7.2.1 Fault Syndrome

For each SDC observed at the micro-instructions output a *detailed report* a detailed report has been kept (described in Section 7.2), that includes also the corrupted

output value. The fault syndrome is then characterized by measuring the *relative error* induced by the fault (i.e., the ratio between the observed wrong output and the expected output).

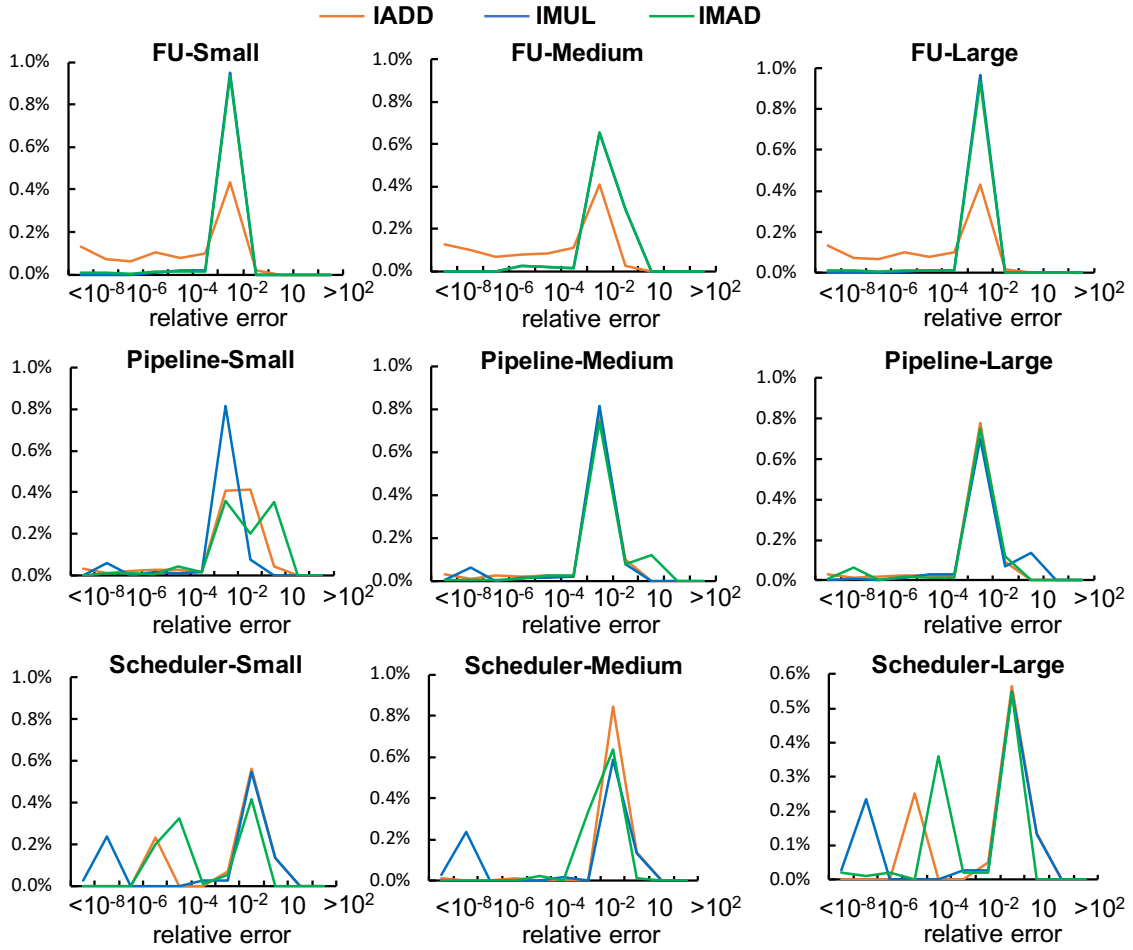
The relative error syndrome analysis highlights an exciting trend. Figures 7.3 and 7.4 show the distribution of relative errors for the tested micro-benchmarks. In the y-axes is plotted the percentage of observed SDCs that modify the micro-instruction output value from less than 10^{-8} to over 10^{2}. That is, 0.2% of SDCs observed on FADD executed with the Small input range modify the output value by $10^{-6}\times$ and 0.3% of SDCs observed on FFMA executed with the Large input range modify the output value by $10\times$.

Figure 7.3: Distribution of the fault syndrome (relative error) from the RTL fault injection in the Functional Units (top), Pipeline (middle), and Scheduler (bottom) for the floating point instructions executed with S, M, L inputs.



As shown in Figures 7.3 for floating-point and Figure 7.4 for integer instructions, the relative difference between the observed corrupted values and the expected value does not follow a Gaussian distribution. There is a clear peak for all instructions that depend on the input range and the injection site. In some configurations (FFMA and FMUL FU

Figure 7.4: Distribution of the fault syndrome (relative error) from the RTL fault injection in the Functional Units (top), Pipeline (middle), and Scheduler (bottom) for the integer instructions executed with S, M, L inputs.



injections with L input), two peaks are present. It is also interesting that the relative difference distribution is narrow compared to the floating point or integer representation range. Only a few cases (less than 0.05%) have been observed a syndrome with a relative error higher than 10^2 (i.e., the corrupted output value is 100x bigger/smaller than the expected one). This observation attests that injecting a random number of bit-flips in the instruction output might not be realistic. The results show that there is a limited difference between corrupted and correct values. Interestingly, the median of the syndrome values between S/M/L varies by just $\sim$1% in all cases but MUL and FMA, for which the median changes by up to 30% (bigger input range has higher median). Only for MUL and FMA, then, is expected a syndrome dependence on the input.

Once the opcode, the input range, and the injection site have been determined, to inject the syndrome in software, it is necessary to select a relative error, statistically taken from the data presented in Figures 7.3 and 7.4 and available at (SANTOS et al.,

2021). The syndrome (as relative error) does not follow a Gaussian distribution (almost all distributions have a p-value smaller than 0.05 on the Shapiro-Wilk test), but instead follows a *power law* distribution (CLAUSET; SHALIZI; NEWMAN, 2009), in which few events are predominant. A Pseudo-Random Number Generator (PRNG) function is created, based on the mathematical formalization in (CLAUSET; SHALIZI; NEWMAN, 2009), that generates the syndrome to be injected as follows:

$$relative\_error = P^{-1}(1-r) = x_{min}(1-r)^{-1/(\alpha-1)} \tag{7.1}$$

Where $0 \leq r < 1$ is a uniformly distributed random value, α is the scaling factor from the data, and x_{min} is the values lower bound. Both parameters are extracted from the data on figures 7.4 and 7.3 based on the methods described in (CLAUSET; SHALIZI; NEWMAN, 2009).

7.2.2 Tiled MxM errors distribution

Scheduler corruptions (and multiple threads corruptions in general) may have specific effects on the execution of codes in which threads interact with each other that may not be detected with the micro-benchmarks that have been designed. As a specific and extremely important case study, this work also characterizes with RTL fault injection a **tile-based matrix multiplication (t-MxM)** mini-app. The observation dictates the choice of the mini-app that more than 70% of operations inside a CNN is MxM related (REDMON; FARHADI, 2018). To avoid memory latencies and, thus, improve matrix multiplication efficiency, large matrix multiplications are split into *tiles* (smaller MxM). The tile size is set to maximize performances without saturating caches and registers. In the proposed framework, the optimal tile size is 8x8. Each tile is assigned to a Streaming Multiprocessor and, then, all tiles are combined to form the output of MxM. In a CNN, the MxM output forms the layer output (feature map).

To select the input for t-MxM, FLENET and FYOLOV3 have been executed with the MNIST (LeCun et al., 1989) and VOC2012 (EVERINGHAM et al., 2012) datasets and observed that most tiles involved in the convolution process have similar values. In contrast, the tiles at the edge of the feature map have a higher amount of zero operands because of padding (REDMON; FARHADI, 2018; LeCun et al., 1989). Then three inputs for the tiles have been characterized with the RTL fault injection: (Max) Max tile (the tile

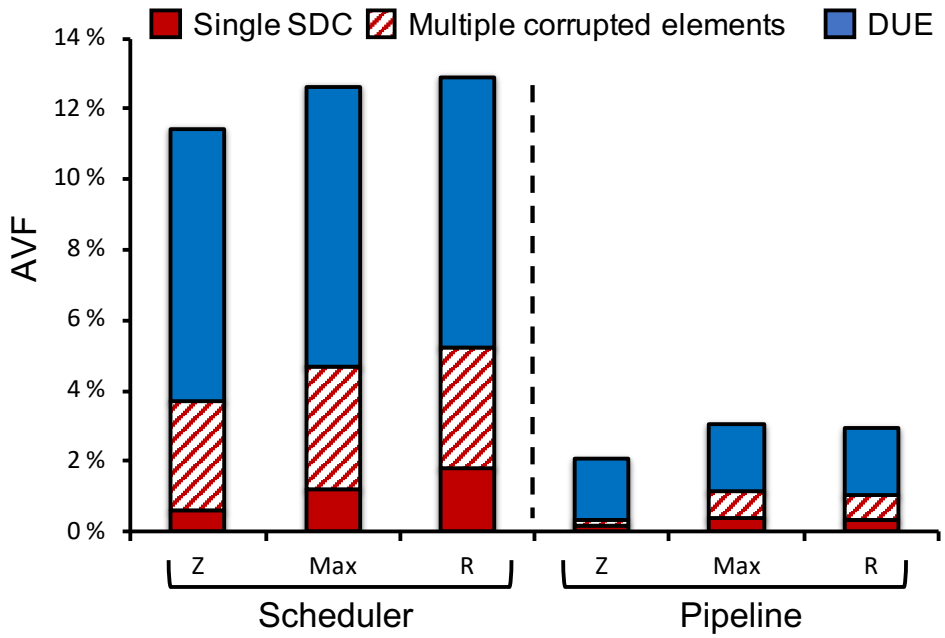
with the highest sum of elements values), (Z) Zero tile (the tile with the highest number of zeros), and (R) Random tile (a tile selected among the ones without significantly biased values). Four different values per tile type (Max, Z, R) are used.

A dedicated procedure o corrupt the output of tiled-MxM is included inside The CNNs. The fault simulator picks a random tile during the execution of a random CNN layer and modifies its output elements according to the syndrome (relative error and spatial distribution) defined with the RTL fault injection.

To better study the impact of multiple threads corruptions in codes with threads interactions, with the RTL fault injector, the mini-app t-MxM is characterized. Three input types (Max, Zero, and Random tiles) are tested, averaging the results obtained with four values per input type. For t-MxM, faults are injected in the scheduler and pipeline registers but not in the functional units. Faults in this latter module, as shown in Figure 7.2, would not cause multiple threads corruptions in t-MxM (there is no transcendental operation). The effects of the single thread SDC would be the same as those observed injecting the FU syndrome in software without requiring a costly RTL simulation.

Figure 7.5 shows the average AVF for DUEs, single and multiple corrupted elements in the t-MxM output for injections in the scheduler and pipeline. It is worth recalling that *one fault is injected per execution*, the multiple corrupted elements are caused by the single fault propagation.

Figure 7.5: AVF of the scheduler (left) and pipeline (right) for DUEs, single and multiple thread SDCs for the Max, Zero, and Random t-MxM.



A significant difference from the micro-benchmarks AVF in Figure 7.2 is that, for t-MxM, the scheduler AVF is higher than the pipeline one. As mentioned, while the micro-benchmarks are very simple and do not implement threads interactions, t-MxM also includes several instructions for computing memory addresses and threads indices. The higher strain on the scheduler and the higher portion of time spent in scheduling operation increases the AVF (for both SDCs and DUEs). On the contrary, the pipeline AVF is higher in the micro-benchmarks because, when a fault appears at the first instruction output, it is marked as SDCs, without further chances to be masked (there is no other computation). In t-MxM, an instruction's wrong output can be masked in a downstream operation (for instance, a multiplication by zero). This statement is supported by our pipeline data in Figure 7.5, which shows a much lower SDC AVF for the Z tile.

An additional exciting result from Figure 7.5 is that the portion of SDCs that affect multiple elements is very high (at least 70% of scheduler induced and 50% of pipeline induced SDCs). It is possible to further study the multiple errors at the t-MxM output by identifying the geometrical distribution of the corrupted elements. Table 7.1 shows the different spatial multiple corrupted elements distribution patterns that are observed injecting faults in the scheduler and pipeline registers. The corrupted elements are distributed in a row, a column, a row and column, a block of elements (varying in size), randomly, and all (or almost all) elements corrupted. Table 7.1 lists the percentage of occurrences of the different patterns (single SDCs are not listed). As the distribution of the observed patterns is very similar in the three inputs tested (M, Z, R tiles), the average distributions are listed.

Table 7.1: Distribution of the multiple patterns (single corrupted elements are not listed) observed with t-MxM.

	row	**column**	**row+column**	**block**	**random**	**all**
scheduler	0.96%	0.07%	0.45%	5.77%	0.69%	54.6%
pipeline	45.4%	1.36%	1.04%	7.29%	0.42%	4.17%

Interestingly, pipeline injection mainly produces corrupted rows, while scheduler injection is more likely to affect the whole output matrix. Having an entire column corrupted is very unlikely for both injection sites. This is because the t-MxM calculation is row-major and, as mentioned, the distribution of these error patterns depends not only on the way the GPU hardware reacts to the faults but also on the software propagation. While this multiple elements distribution is not generic, the choice of t-MxM extended evaluation is dictated by its importance in CNN's execution. As shown in (Santos et al., 2019; Ibrahim et al., 2020), the observed multiple errors patterns (but not single element

corruptions) can indeed induce misdetections in CNNs.

7.3 HPC Applications Evaluation

The modified NVBiTFI version selects the most suitable fault syndrome to apply depending on the opcode, the input, and the module assumed to be the cause of the fault. A cocktail of fault syndromes are injected following the power-law statistical distribution described in Section 7.3 and shown in Figures 7.3 and 7.4.

A subset of applications from Section 3.2 are selected to be evaluated that are representative of different HPC computational classes: FMXM, FLUD, Quicksort, FLAVA, FGAUSSIAN, and FHOTSPOT. Also, two CNNs for classification and object detection (FLENET and FYOLOV3) are considered. Each code is likely to stimulate specific GPU modules, according to the distribution of opcodes depicted in Figure 3.1. Hence, results obtained with the selected benchmarks could be representative also of similar applications.

To compare with the traditional single bit-flip evaluation, only single-thread SDC are injected using the fault syndrome proposed. For CNNs, the RTL fault-injection on the execution of t-MxM are also included to evaluate better the effects of scheduler faults and multiple threads corruptions in the detection and classification of objects.

Figure 7.6: SDC Program Vulnerability Factor for HPC codes.

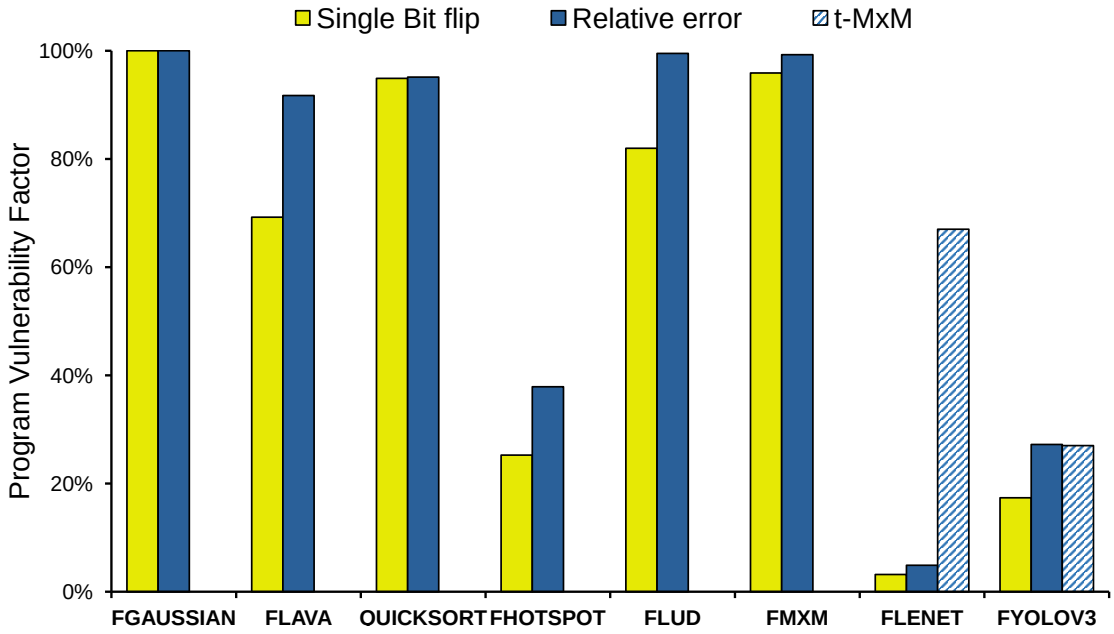


Figure 7.6 shows the SDC Program Vulnerability Factor (PVF) for the HPC codes.

PVF is the probability of the faults injected in the software visible states to generate an SDC at the end of execution. In other words, when injecting in software, it is assumed that the fault injected in RTL has corrupted the instruction output. For the data in Figure 7.6, the fault injection is performed only in the 12 opcodes characterized with RTL fault injection (that represent more than 70% of all executed opcodes, as shown in Figure 3.1).

RTL faults that generate DUEs (shown in Figure 7.2) are not considered in software fault injection, as they simply hang the application. As the injection is performed only in the output of the instructions and is intended to evaluate the manifestation at the program's output for the two error models, the DUEs are not considered in this analysis.

Two error models are considered: single bit-flip (randomly injected in the 32 bits values) and the fault syndrome (injected using the power-law distribution) from RTL injection. For all the codes presented in Figure 7.6, the fault syndrome model generates a higher or equal PVF compared with the traditional single bit-flip error model. Interestingly, it is possible to observe that the single bit-flip injection would underestimate the application's reliability of up to 30% for Lava and 48% for Hotspot, respectively. For other codes (FGAUSSIAN and QUICKSORT), the two fault models provide very similar results, as the PVF of the considered instructions is, by itself, extremely high (close to 1).

For CNNs, if it is considered an SDC, as it is done in Figure 7.6, a mismatch in the application output, the single bit-flip injection underestimates the PVF of 33% for FLENET and 50% for FYOLOV3. The higher reliability to the transient fault of CNNs compared to HPC codes should not surprise, as it has already been observed and studied on Section 4.1 and other works (Ibrahim et al., 2020).

For FLENET and FYOLOV3, the PVF is also measured when injecting the corrupted t-MxM, as presented earlier in this section. On FLENET, the SDC PVF when t-MxM fault model is injected is much higher than the other two fault models (12x higher), while for FYOLOV3, it is similar to the relative error PVF. This different behavior is because FLENET has a minimal number of network parameters per layer (12,000, on average). Thus, the corruption of a tile consists of the corruption of a considerable number of parameters. On the contrary, FYOLOV3 layers are very big (100,000 parameters, on average), and even fully corrupted 8x8 tile represent a small percentage of the matrix.

It is possible to further analyze the impact of faults in CNNs by distinguishing between tolerable SDCs and critical SDCs, i.e., those that corrupt the output sufficiently to cause a network misclassification/misdetection. The results show that t-MxM injection produces an unacceptable amount of critical errors. For FLENET, the number of errors

that change the classification entirely is 20% and, for FYOLOV3, it is 15%. It is worth noting that, in FLENET, none of the injected single bit-flips nor RTL single thread syndrome produce misclassifications nor misdetections. A realistic and accurate fault model that also considers faults in GPU critical resources as the scheduler is necessary not to risk underestimating the effect of transient faults in CNNs.

By investigating further the RTL fault propagation, it has been found that the control structures (inside the scheduler, the pipeline, and the SFU) are the primary sources of errors that corrupt multiple threads, affecting a warp or even generating the geometrical patterns of errors shown in Table 7.1. As seen with the software fault injection, despite the limited size of these structures in a GPU core and the relatively low AVF, these critical modules might produce severe consequences for an application, especially CNNs. An efficient and effective hardening solution for GPU should target these modules.

Finally, it must be highlighted that injecting at RTL level *one single fault* in just one of the applications listed in Figure 7.6 would take more than 10 hours, using our 12 CPUs server. As a total of 48,000 faults are injected, it would take 4.8×10^5 hours to produce all data in Figure 7.6. That is more than 54 *years*. Despite the limitations listed in Section 7.1.1 and the introduction of some simplifications on the input range, our two-level framework allows an analysis that would otherwise be impossible.

8 CONCLUSIONS

This chapter summarizes the main conclusions of this work and presents some possible directions for the open questions raised during the research process.

8.1 Summary of contributions

At the beginning of this work, four items have been proposed to be discussed. In the following, a summary of the main contributions for each item is presented.

1. *Evaluate the reliability of High Performance Computing (HPC) applications and Convolutional Neural Networks (CNNs):* This work covered the error rate of several types of HPC applications from different domains and also the error rate and criticality of CNNs in Chapter 4. This thesis presented more data on radiation experiments, and fault simulation for NVIDIA GPUs than any work in the literature had shown until now.
2. *Propose a new hardening for current GPU architectures:* In Chapter 5 very efficient and feasible fault tolerance solutions were presented. For CNNs the *GEMM ABFT* and *Reliable Max-pooling* were introduced as a solution to the critical errors on object detection. A more generically idea was implemented for the HPC benchmarks, the RP-DWC. Both techniques can be efficiently achieved at the software level with the available GPUs or with specific hardware.
3. *Combine beam experiments and fault simulation:* This work presented a novel approach for error rate estimation for NVIDIA GPUs that considers the GPU's particular aspects. Based on the Error Probability concept, Chapter 6 provided a satisfactory methodology to estimate the SDC rate of an application. In contrast with other reliability evaluation methodologies, SDC rate estimation is able to consider the software and hardware characteristics. Consequently, SDC rate estimation could produce more insights on the reliability than a sole software-level analysis.
4. *The conception of a new error model for GPUs:* Lastly, based on all knowledge built in the previous chapters, a new error model was presented in Chapter 7. The novelty error model is based on a two-level fault injection that combines RTL and software simulations. The proposed approach is able to incorporate the accuracy of the RTL level fault propagation and the speed of software-based fault injection.

The speedup for the new method can reduce the simulation time from months to hours.

8.2 Future work

This thesis has presented a discussion on some of the main open problems for GPU reliability. Still, some issues are highly complex and need to be more analyzed. For instance, one of the issues with the FIT estimation method presented in Chapter 6 is the benchmarks that have unsatisfactory performance on GPUs (low IPC and low occupancy) due to the algorithm characteristics, such as memory access patterns, low functional units utilization, etc. For those codes, a deeper study must be performed. Adding the error rate of a higher number of instructions to the model can be a path to solve this issue. Also, the model can be improved by combining the data from fault simulation in lower levels than software, such as RTL and Micro-architectural. With a detailed analysis from RTL and Micro-architectural fault injections, it may be possible to find the bottleneck for the codes with a bad FIT estimation.

Also, it is necessary to investigate how the compiler impacts the reliability analysis. As demonstrated in Chapter 6, when the compiler modifies the final machine code, it is expected that the error rate will also be changed. Thus, it is necessary to evaluate more the compiler optimizations and code generation impact on the code's reliability. Additionally, it is mandatory to investigate if higher performance leads to better or worse reliability since the final goal of GPU users is to achieve as much performance per watt as possible.

In Chapter 5 very efficient software approaches were presented for CNNs. However, as the deep learning algorithms are increasing in complexity and size, it is necessary to reduce even more the overhead added from hardening techniques applied to CNNs. Consequently, it is essential to start designing hardened CNNs from the training process. That is, the model training can be tuned to be hardened against radiation-induced errors in the execution. A hardened CNN model would tolerate critical errors without the overhead from the fault tolerance technique applied to the inference process.

8.3 Conclusions

This work has presented a deep reliability analysis of a broad domain of applications from HPC to machine learning applications. This thesis provided a database for two GPU architectures, Kepler and Volta, and presented trends for each application reliability running on them. The efficiency of Error Correction Code (ECC) has also been measured for both GPU architectures.

For the CNNs particular case, three CNNs were evaluated (YOLO, Faster R-CNN, ResNet) on different NVIDIA GPUs. The CNNs were exposed to atmospheric-like neutron beams and fault simulation. The results have shown that for CNNs, Crashes are more frequent than SDCs, and the higher number of operations executed in Faster R-CNN and ResNet makes them more prone to corruption than YOLO. It has been demonstrated that GPUs microarchitecture can propagate a single fault to affect several output elements, and this behavior significantly impacts CNN reliability. The critical and tolerable errors have been distinguished in the execution of object detection frameworks. Unfortunately, ECC is insufficient to ensure high reliability in CNNs, as it does not reduce the number of critical errors. As a contribution, this thesis proposed alternative protection techniques, such as ABFT, that can be much more effective in reducing the critical error rate. Additionally, fault propagation was studied for YOLO. Faults tend to spread during convolution, suggesting that faults should be promptly detected to ensure high reliability. When analyzing how faults propagate through GPUs when executing a CNN pipeline, this thesis has proposed the design of maxpool layers to detect radiation-induced errors at runtime.

The reliability of mixed precision applications on NVIDIA GPUs was also evaluated in this research. It has been demonstrated that the use of mixed-precision data or operation significantly affects the device error rate and the code behavior in the presence of transient faults. The experimental data highlights that reducing precision is beneficial both in terms of reduced error rate and improved performance. Additionally, the impact of faults in output correctness has been studied. On GPUs, the higher the precision, the lower the fault impact. In general, GPU double-precision executions always benefit from a lower impact of faults in the output correctness. Based on the data generated for the mixed-precision evaluation, a new hardening strategy was presented for GPUs, Reduced Precision Duplication With Comparison (RP-DWC). RP-DWC is a strategy to detect errors duplicating the original instructions with reduced-precision instructions. The technique is particularly promising for (but not limited to) mixed-precision architectures, as

novel GPUs that have dedicated hardware cores to execute different precision operations. RP-DWC uses available hardware that would otherwise be idle to improve the reliability of codes with reduced overhead. Among the limitations of RP-DWC, the most challenging one is the reduced error detection capability that comes from the different intrinsic results of lower precision executions. However, a significant amount of errors can still be detected, and, more importantly, the undetected errors are the ones that have a more negligible impact on the output correctness. As a result, undetected errors may still be tolerated by various applications.

In order to evaluate if the most used error model for software fault simulation (i.e., single bit flip) is comparable with realistic experiments, the FIT rates obtained from beam experiments and fault simulation estimation for 15 codes were compared, using Kepler and Volta based NVIDIA GPUs and two fault injection frameworks (SASSIFI and NVBitFI). When considering the GPU parallelism management (GPU occupancy and IPC), fault simulation provides SDC FIT rates comparable to the beam test results. The result holds for two GPUs for a broad set of codes. The source of the SDC FIT rates of codes executed on GPUs was investigated. The data from the beam and fault simulation data are combined to understand if the FIT rate is due to the high resource usage, the high criticality of resources (AVF), or a combination of the two. Finally, the main GPUs' functional units sensitivity are evaluated, including mixed-precision and tensor cores. Unfortunately, fault simulation alone is not enough to assess the probability of DUEs, as faults in inaccessible resources probably are the leading cause of these events.

Finally, a new error model was conceived based on the knowledge and data from all the experiments, hardening strategies, and error rate estimation. The concept of multi-level fault injection to GPUs is used. And, thanks to the combination of RTL and software fault injection, the time required to have a detailed and accurate analysis of faults propagation from the hardware source to the application output is reduced by orders of magnitude. The RTL accuracy of the proposed framework identifies the most critical GPU resources for both SDCs and DUEs and identifies a set of possible fault effects (syndromes) in the instructions output. The efficiency of the modified NVBitFI allows the propagation of these effects in real-world applications. The proposed faults syndrome database presented in this work is made publicly available to provide a more accurate fault model than the naive single bit-flip to evaluate the reliability of applications and validate hardening solutions. Moreover, the framework's flexibility grants the possibility of future updates, both in terms of updated RTL model or extended instructions evaluation.

REFERENCES

ALIPPI, C.; PIURI, V.; SAMI, M. Sensitivity to errors in artificial neural networks: a behavioral approach. **IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications**, v. 42, n. 6, p. 358–361, Jun 1995. ISSN 1057-7122.

ALOM, M. Z. et al. A state-of-the-art survey on deep learning theory and architectures. **Electronics**, MDPI AG, v. 8, n. 3, p. 292, Mar 2019. ISSN 2079-9292. Available from Internet: <http://dx.doi.org/10.3390/electronics8030292>.

ANGELOVA, A. et al. Real-time pedestrian detection with deep network cascades. In: **Proceedings of BMVC 2015**. [S.l.: s.n.], 2015.

ANWER, A. R. et al. Gpu-trident: Efficient modeling of error propagation in gpu programs. In: **Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis**. [S.l.]: IEEE Press, 2020. (SC '20). ISBN 9781728199986.

ARAFA, Y. et al. Low overhead instruction latency characterization for NVIDIA gpgpus. In: **2019 IEEE High Performance Extreme Computing Conference, HPEC 2019, Waltham, MA, USA, September 24-26, 2019**. IEEE, 2019. p. 1–8. Available from Internet: <https://doi.org/10.1109/HPEC.2019.8916466>.

Ashraf, R. A. et al. Exploring the effect of compiler optimizations on the reliability of hpc applications. In: **2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)**. [S.l.: s.n.], 2017. p. 1274–1283.

BAGGIO, J. et al. Analysis of proton/neutron SEU sensitivity of commercial SRAMs-application to the terrestrial environment test method. **IEEE Transactions on Nuclear Science**, Institute of Electrical and Electronics Engineers (IEEE), v. 51, n. 6, p. 3420–3426, dec. 2004. Available from Internet: <https://doi.org/10.1109/tns.2004.839135>.

BAUMANN, R. Soft errors in advanced computer systems. **IEEE Design Test of Computers**, v. 22, n. 3, p. 258–266, May 2005. ISSN 0740-7475.

BETTOLA, S.; PIURI, V. High performance fault-tolerant digital neural networks. **IEEE Transactions on Computers**, v. 47, n. 3, p. 357–363, Mar 1998. ISSN 0018-9340.

BIYIKLI, E.; YANG, Q.; TO, A. C. Multiresolution molecular mechanics: Dynamics. **Computer Methods in Applied Mechanics and Engineering**, v. 274, p. 42–55, 2014. ISSN 0045-7825. Available from Internet: <http://www.sciencedirect.com/science/article/pii/S0045782514000589>.

BOLT, G. **Fault Tolerant Multi-Layer Perceptron Networks**. [S.l.], 1992.

BRAUN, C.; HALDER, S.; WUNDERLICH, H. J. A-abft: Autonomous algorithm-based fault tolerance for matrix multiplications on graphics processing units. In: **Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks**. Washington, DC, USA: IEEE Computer Society, 2014. (DSN '14), p. 443–454. ISBN 978-1-4799-2233-8. Available from Internet: <http://dx.doi.org/10.1109/DSN.2014.48>.

BREUER, M. A. Multi-media applications and imprecise computation. In: IEEE. **Digital System Design, 2005. Proceedings. 8th Euromicro Conference on**. [S.l.], 2005. p. 2–7.

BUCHNER, S. et al. Comparison of error rates in combinational and sequential logic. **Nuclear Science, IEEE Transactions on**, IEEE Press, v. 44, n. 6, p. 2209–2216, 1997.

CANDEA, G.; FOX, A. Recursive restartability: Turning the reboot sledgehammer into a scalpel. In: **Proceedings of the Eighth Workshop on Hot Topics in Operating Systems**. Washington, DC, USA: IEEE Computer Society, 2001. (HOTOS '01), p. 125–. Available from Internet: <http://dl.acm.org/citation.cfm?id=874075.876408>.

CAZZANIGA, C.; FROST, C. D. Progress of the scientific commissioning of a fast neutron beamline for chip irradiation. **Journal of Physics: Conference Series**, IOP Publishing, v. 1021, p. 012037, may 2018. Available from Internet: <https://doi.org/10.1088\%2F1742-6596\%2F1021\%2F1\%2F012037>.

CHAKRABARTI, G. et al. CUDA: Compiling and optimizing for a GPU platform. **Procedia Computer Science**, v. 9, p. 1910–1919, 12 2012.

Chatzidimitriou, A. et al. Demystifying soft error assessment strategies on ARM CPUs: Microarchitectural fault injection vs. neutron beam experiments. In: **2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)**. [S.l.: s.n.], 2019. p. 26–38. ISSN 1530-0889.

CHE, S. et al. Rodinia: A benchmark suite for heterogeneous computing. In: **Proceedings of the IEEE Int. Symposium on Workload Characterization (IISWC)**. [S.l.: s.n.], 2009. p. 44–54.

Chen, H. et al. Using low cost erasure and error correction schemes to improve reliability of commodity dram systems. **IEEE Transactions on Computers**, v. 65, n. 12, p. 3766–3779, 2016.

Chen, J.; Li, S.; Chen, Z. Gpu-abft: Optimizing algorithm-based fault tolerance for heterogeneous systems with gpus. In: **2016 IEEE International Conference on Networking, Architecture and Storage (NAS)**. [S.l.: s.n.], 2016. p. 1–2.

CHETLUR, S. et al. cudnn: Efficient primitives for deep learning. **CoRR**, abs/1410.0759, 2014. Available from Internet: <http://arxiv.org/abs/1410.0759>.

Cho, H. et al. Understanding soft errors in uncore components. In: **2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)**. [S.l.: s.n.], 2015. p. 1–6.

CIREGAN, D.; MEIER, U.; SCHMIDHUBER, J. Multi-column deep neural networks for image classification. In: **2012 IEEE Conference on Computer Vision and Pattern Recognition**. [S.l.: s.n.], 2012. p. 3642–3649. ISSN 1063-6919.

CIRESAN, D. C. et al. Deep neural networks segment neuronal membranes in electron microscopy images. In: **Proceedings of the 25th International Conference on Neural Information Processing Systems**. USA: Curran Associates Inc., 2012. (NIPS'12), p. 2843–2851. Available from Internet: <http://dl.acm.org/citation.cfm?id=2999325.2999452>.

CLARK, M. et al. Solving lattice qcd systems of equations using mixed precision solvers on gpus. **Computer Physics Communications**, v. 181, n. 9, p. 1517–1528, 2010. ISSN 0010-4655. Available from Internet: <http://www.sciencedirect.com/science/article/pii/S0010465510001426>.

CLAUSET, A.; SHALIZI, C. R.; NEWMAN, M. E. J. Power-law distributions in empirical data. **SIAM Review**, v. 51, n. 4, p. 661–703, 2009. Available from Internet: <https://doi.org/10.1137/070710111>.

COLLOBERT, R.; KAVUKCUOGLU, K.; FARABET, C. **Torch7: A Matlab-like Environment for Machine Learning**. 2011.

CONDIA, J. E. R. et al. Flexgripplus: An improved gpgpu model to support reliability analysis. **Microelectronics Reliability**, v. 109, p. 113660, 2020. ISSN 0026-2714. Available from Internet: <https://www.sciencedirect.com/science/article/pii/S0026271419307978>.

CONSTANTINESCU, C. Impact of deep submicron technology on dependability of vlsi circuits. In: **Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on**. [S.l.: s.n.], 2002. p. 205–209.

COURBARIAUX, M.; BENGIO, Y.; DAVID, J. Low precision arithmetic for deep learning. **CoRR**, abs/1412.7024, 2014. Available from Internet: <http://arxiv.org/abs/1412.7024>.

DISTANTE, F.; PIURI, V. Anna-use of pads for artificial neural network analysis: the defect and fault tolerance issue. In: **[1991] Proceedings, Advanced Computer Technology, Reliable Systems and Applications**. [S.l.: s.n.], 1991. p. 537–540.

DISTANTE, F. et al. Mapping neural nets onto a massively parallel architecture: a defect-tolerance solution. **Proceedings of the IEEE**, v. 79, n. 4, p. 444–460, Apr 1991. ISSN 0018-9219.

DOLLÁR, P. et al. Pedestrian detection: An evaluation of the state of the art. **PAMI**, v. 34, 2012.

Du, B. et al. On the evaluation of seu effects in gpgpus. In: **2019 IEEE Latin American Test Symposium (LATS)**. [S.l.: s.n.], 2019. p. 1–6.

Ejlali, A. et al. A hybrid fault injection approach based on simulation and emulation co-operation. In: **2003 International Conference on Dependable Systems and Networks, 2003. Proceedings.** [S.l.: s.n.], 2003. p. 479–488.

EVERINGHAM, M. et al. **The PASCAL Visual Object Classes Challenge 2012 (VOC2012) Results**. 2012. Http://www.pascal-network.org/challenges/VOC/voc2012/workshop/index.html.

FANG, B. et al. Gpu-qin: A methodology for evaluating the error resilience of gpgpu applications. In: **Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on**. [S.l.: s.n.], 2014. p. 221–230.

FAWCETT, T. An introduction to roc analysis. **Pattern recognition letters**, Elsevier, v. 27, n. 8, p. 861–874, 2006.

FENG, S. et al. Shoestring: Probabilistic soft error reliability on the cheap. **SIGPLAN Not.**, Association for Computing Machinery, New York, NY, USA, v. 45, n. 3, p. 385–396, mar. 2010. ISSN 0362-1340. Available from Internet: <https://doi.org/10.1145/1735971.1736063>.

Fernandes dos Santos, F. et al. Reliability evaluation of mixed-precision architectures. In: **2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)**. [S.l.: s.n.], 2019. p. 238–249. ISSN 2378-203X.

FRATIN, V. et al. Code-dependent and architecture-dependent reliability behaviors. In: **2018 48th Annual IEEE/IFIP Int. Conference on Dependable Systems and Networks (DSN)**. [S.l.: s.n.], 2018. p. 13–26.

GAWANDE, N. A. et al. Scaling deep learning workloads: Nvidia dgx-1/pascal and intel knights landing. **Future Generation Computer Systems**, 2018. ISSN 0167-739X. Available from Internet: <http://www.sciencedirect.com/science/article/pii/S0167739X17318599>.

GKIOXARI, G.; MALIK, J.; JOHNSON, J. Mesh R-CNN. **CoRR**, abs/1906.02739, 2019. Available from Internet: <http://arxiv.org/abs/1906.02739>.

GODDEKE, D.; STRZODKA, R.; TUREK, S. Performance and accuracy of hardware-oriented native, emulated and mixed-precision solvers in fem simulations. **Int. Journal of Parallel, Emergent and Distributed Systems**, Taylor and Francis, v. 22, n. 4, p. 221–256, 2007. Available from Internet: <https://doi.org/10.1080/17445760601122076>.

GOMEZ, L. A. B. et al. GPGPUs: How to Combine High Computational Power with High Reliability. In: **2014 Design Automation and Test in Europe Conference and Exhibition**. Dresden, Germany: [s.n.], 2014.

Goncalves de Oliveira, D. A. G. et al. Evaluation and mitigation of radiation-induced soft errors in graphics processing units. **IEEE Transactions on Computers**, v. 65, n. 3, p. 791–804, 2016.

GRAND, S. L.; GöTZ, A. W.; WALKER, R. C. Spfp: Speed without compromise - a mixed precision model for gpu accelerated molecular dynamics simulations. **Computer Physics Communications**, v. 184, n. 2, p. 374–380, 2013. ISSN 0010-4655. Available from Internet: <http://www.sciencedirect.com/science/article/pii/S0010465512003098>.

GUPTA, S. et al. Deep learning with limited numerical precision. In: **Proceedings of the 32Nd Int. Conference on Machine Learning - Volume 37**. JMLR.org, 2015. (ICML'15), p. 1737–1746. Available from Internet: <http://dl.acm.org/citation.cfm?id=3045118.3045303>.

Haque, I. S.; Pande, V. S. Hard data on soft errors: A large-scale assessment of real-world error rates in gpgpu. In: **2010 10th CCGRID**. [S.l.: s.n.], 2010. p. 691–696.

HARI, S. K. S.; ADVE, S. V.; NAEIMI, H. Low-cost program-level detectors for reducing silent data corruptions. In: **IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)**. [S.l.: s.n.], 2012. p. 1–12.

HARI, S. K. S. et al. **Estimating Silent Data Corruption Rates Using a Two-Level Model**. 2020.

HARI, S. K. S. et al. Making convolutions resilient via algorithm-based error detection techniques. **IEEE Transactions on Dependable and Secure Computing**, p. 1–1, 2021.

Hari, S. K. S. et al. SASSIFI: An architecture-level fault injection tool for GPU application resilience evaluation. In: **2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)**. [S.l.: s.n.], 2017. p. 249–258.

HARISH, P.; NARAYANAN, P. J. Accelerating large graph algorithms on the gpu using cuda. In: ALURU, S. et al. (Ed.). **High Performance Computing – HiPC 2007**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007. p. 197–208. ISBN 978-3-540-77220-0.

HE, K. et al. Identity mappings in deep residual networks. **CoRR**, abs/1603.05027, 2016. Available from Internet: <http://arxiv.org/abs/1603.05027>.

HO, N. et al. Efficient floating point precision tuning for approximate computing. In: **2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)**. [S.l.: s.n.], 2017. p. 63–68. ISSN 2153-697X.

HO, N.; WONG, W. Exploiting half precision arithmetic in nvidia gpus. In: **2017 IEEE High Performance Extreme Computing Conference (HPEC)**. [S.l.: s.n.], 2017. p. 1–7.

HUANG, K.-H.; ABRAHAM, J. Algorithm-based fault tolerance for matrix operations. **Computers, IEEE Transactions on**, C-33, n. 6, p. 518–528, June 1984. ISSN 0018-9340.

HWANG, K.; SUNG, W. Fixed-point feedforward deep neural network design using weights +1, 0, and -1. In: **2014 IEEE Workshop on Signal Processing Systems (SiPS)**. [S.l.: s.n.], 2014. p. 1–6. ISSN 2162-3562.

Ibrahim, Y. et al. Soft error resilience of deep residual networks for object recognition. **IEEE Access**, v. 8, p. 19490–19503, 2020.

IEEE. Ieee standard for floating-point arithmetic. **IEEE Std 754-2008**, p. 1–70, Aug 2008.

INTEL. **Intel C++ Compiler 17.0 Developer Guide and Reference**. [S.l.], 2016. Available from Internet: <https://software.intel.com/en-us/node/683943>.

ISO. **ISO 26262-9:2011 Preview Road Vehicles Functional Safety**. 2011. Https://www.iso.org/standard/51365.html.

JEDEC. **Measurement and Reporting of Alpha Particle and Terrestrial Cosmic Ray-Induced Soft Errors in Semiconductor Devices**. [S.l.], 2006.

JHA, S. et al. **Kayotee: A Fault Injection-based System to Assess the Safety and Reliability of Autonomous Vehicles to Faults and Errors**. 2019.

JIA, Y. et al. Caffe: Convolutional architecture for fast feature embedding. In: **Proceedings of the 22Nd ACM International Conference on Multimedia**. New York, NY, USA: ACM, 2014. (MM '14), p. 675–678. ISBN 978-1-4503-3063-3. Available from Internet: <http://doi.acm.org/10.1145/2647868.2654889>.

JIA, Z. et al. Dissecting the NVIDIA volta GPU architecture via microbenchmarking. **CoRR**, abs/1804.06826, 2018. Available from Internet: <http://arxiv.org/abs/1804.06826>.

JOUPPI, N. P. et al. In-datacenter performance analysis of a tensor processing unit. In: **Proceedings of the 44th Annual International Symposium on Computer Architecture**. New York, NY, USA: ACM, 2017. (ISCA '17), p. 1–12. ISBN 978-1-4503-4892-8. Available from Internet: <http://doi.acm.org/10.1145/3079856.3080246>.

KALIORAKIS, M. et al. Differential fault injection on microarchitectural simulators. In: **2015 IEEE Int. Symposium on Workload Characterization**. [S.l.: s.n.], 2015. p. 172–182.

Kastensmidt, F. L. et al. On the optimal design of triple modular redundancy logic for sram-based fpgas. In: **Design, Automation and Test in Europe**. [S.l.: s.n.], 2005. p. 1290–1295 Vol. 2.

Kochte, M. A. et al. Efficient simulation of structural faults for the reliability evaluation at system-level. In: **2010 19th IEEE Asian Test Symposium**. [S.l.: s.n.], 2010. p. 3–8.

KRIZHEVSKY, A. One weird trick for parallelizing convolutional neural networks. **CoRR**, abs/1404.5997, 2014. Available from Internet: <http://arxiv.org/abs/1404.5997>.

KUDVA, P. et al. Low-cost concurrent error detection for floating-point unit (FPU) controllers. **IEEE Transactions on Computers**, IEEE Computer Society, Los Alamitos, CA, USA, v. 62, n. 07, p. 1376–1388, jul 2013. ISSN 1557-9956.

KULAKOV, A.; ZWOLINSKI, M.; REEVE, J. S. Fault tolerance in distributed neural computing. **CoRR**, abs/1509.09199, 2015. Available from Internet: <http://arxiv.org/abs/1509.09199>.

LAM, M. O. et al. Automatically adapting programs for mixed-precision floating-point computation. In: **Proceedings of the 27th International ACM Conference on International Conference on Supercomputing**. New York, NY, USA: ACM, 2013. (ICS '13), p. 369–378. ISBN 978-1-4503-2130-3. Available from Internet: <http://doi.acm.org/10.1145/2464996.2465018>.

LeCun, Y. et al. Backpropagation applied to handwritten zip code recognition. **Neural Computation**, v. 1, n. 4, p. 541–551, Dec 1989. ISSN 0899-7667.

LECUN, Y. et al. Gradient-based learning applied to document recognition. **Proceedings of the IEEE**, v. 86, n. 11, p. 2278–2324, Nov 1998. ISSN 0018-9219.

LI, G. et al. 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**. New York, NY, USA: Association for Computing Machinery, 2017. (SC '17). ISBN 9781450351140. Available from Internet: <https://doi.org/10.1145/3126908.3126964>.

LI, G.; PATTABIRAMAN, K. Modeling input-dependent error propagation in programs. In: **2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)**. [S.l.: s.n.], 2018. p. 279–290.

LI, G. et al. Understanding error propagation in gpgpu applications. In: **SC16: Int. Conference for High Performance Computing, Networking, Storage and Analysis**. [S.l.: s.n.], 2016. p. 240–251.

Lindholm, E. et al. Nvidia tesla: A unified graphics and computing architecture. **IEEE Micro**, v. 28, n. 2, p. 39–55, March 2008. ISSN 1937-4143.

LOMONT, C. Introduction to intel advanced vector extensions. **Intel White Paper**, p. 1–21, 2011.

LU, Q. et al. Llfi: An intermediate code-level fault injection tool for hardware faults. In: **2015 IEEE Int. Conference on Software Quality, Reliability and Security**. [S.l.: s.n.], 2015. p. 11–16.

LUO, T. et al. Dadiannao: A neural network supercomputer. **IEEE Transactions on Computers**, v. 66, n. 1, p. 73–88, Jan 2017. ISSN 0018-9340.

Lyons, R. E.; Vanderkulk, W. The use of triple-modular redundancy to improve computer reliability. **IBM Journal of Research and Development**, v. 6, n. 2, p. 200–209, 1962.

MAHATME, N. et al. Comparison of Combinational and Sequential Error Rates for a Deep Submicron Process. **Nuclear Science, IEEE Transactions on**, IEEE Press, v. 58, n. 6, p. 2719–2725, 2011.

MAHMOUD, A. et al. Optimizing software-directed instruction replication for GPU error detection. In: **Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis**. Piscataway, NJ, USA: IEEE Press, 2018. (SC '18), p. 67:1–67:12. Available from Internet: <http://dl.acm.org/citation.cfm?id=3291656.3291746>.

Maniatakos, M. et al. Exponent monitoring for low-cost concurrent error detection in FPU control logic. In: **29th VLSI Test Symposium**. [S.l.: s.n.], 2011. p. 235–240. ISSN 1093-0167.

Mao, H. et al. Towards real-time object detection on embedded systems. **IEEE Transactions on Emerging Topics in Computing**, v. 6, n. 3, p. 417–431, July 2018. ISSN 2168-6750.

MATHIEU, M.; HENAFF, M.; LECUN, Y. Fast training of convolutional networks through ffts. **CoRR**, abs/1312.5851, 2013. Available from Internet: <http://arxiv.org/abs/1312.5851>.

MICIKEVICIUS, P. et al. Mixed precision training. **CoRR**, abs/1710.03740, 2017. Available from Internet: <http://arxiv.org/abs/1710.03740>.

MINHAS, U. I.; BAYLISS, S.; CONSTANTINIDES, G. A. Gpu vs fpga: A comparative analysis for non-standard precision. In: GOEHRINGER, D. et al. (Ed.). **Reconfigurable Computing: Architectures, Tools, and Applications**. Cham: Springer Int. Publishing, 2014. p. 298–305. ISBN 978-3-319-05960-0.

MUKHERJEE, S. S. et al. A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor. In: **Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture**. Washington, DC, USA: IEEE Computer Society, 2003. p. 29–. ISBN 0-7695-2043-X.

NETI, C.; SCHNEIDER, M. H.; YOUNG, E. D. Maximally fault tolerant neural networks. **IEEE Transactions on Neural Networks**, v. 3, n. 1, p. 14–23, Jan 1992. ISSN 1045-9227.

NGUYEN, A. et al. 3.5dd blocking optimization for stencil computations on modern cpus and gpus. In: **Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis**. Washington, DC, USA: IEEE Computer Society, 2010. (SC '10), p. 1–13. ISBN 978-1-4244-7559-9. Available from Internet: <http://dx.doi.org/10.1109/SC.2010.2>.

NGUYEN, H. T. et al. Chip-level soft error estimation method. **IEEE Transactions on Device and Materials Reliability**, v. 5, n. 3, p. 365–381, Sept 2005. ISSN 1530-4388.

NIMARA, S. et al. Multi-level simulated fault injection for data dependent reliability analysis of rtl circuit descriptions. **Advances in Electrical and Computer Engineering**, v. 16, p. 93–98, 02 2016.

NOH, J. et al. Study of neutron soft error rate (ser) sensitivity: Investigation of upset mechanisms by comparative simulation of finfet and planar mosfet srams. **Nuclear Science, IEEE Transactions on**, v. 62, n. 4, p. 1642–1649, Aug 2015. ISSN 0018-9499.

NVIDIA. **CUDA toolkit documentation**. [S.l.], 2015. Available from Internet: <http://docs.nvidia.com/cuda/cuda-samples/#axzz4JVHU2IKO>.

NVIDIA. Whitepaper gpu-based deep learning inference : A performance and power analysis. In: . [S.l.: s.n.], 2015.

NVIDIA. **NVIDIA TESLA V100 GPU ARCHITECTURE - Whitepaper**. [S.l.], 2017.

NVIDIA. **Floating Point and IEEE 754 Compliance for NVIDIA GPUs**. 2018. Available from Internet: <https://docs.nvidia.com/cuda/floating-point/index.html>.

NVIDIA. **NVIDIA Announces World's First Functionally Safe AI Self-Driving Platform**. 2018. "https://nvidianews.nvidia.com/news/nvidia-announces-worlds-first-functionally-safe-ai-self-driving-platform".

NVIDIA. **CUDA Runtime API**. 2021. Available from Internet: <https://docs.nvidia.com/cuda/cuda-runtime-api/index.html>.

OLIVEIRA, D. et al. Carol-fi: An efficient fault-injection tool for vulnerability evaluation of modern hpc parallel accelerators. In: **Proceedings of the Computing Frontiers Conference**. New York, NY, USA: ACM, 2017. (CF'17), p. 295–298. ISBN 978-1-4503-4487-6. Available from Internet: <http://doi.acm.org/10.1145/3075564.3075598>.

OLIVEIRA, D. et al. Experimental and analytical study of xeon phi reliability. In: **Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis**. New York, NY, USA: ACM, 2017. (SC '17), p. 28:1–28:12. ISBN 978-1-4503-5114-0. Available from Internet: <http://doi.acm.org/10.1145/3126908.3126960>.

OLIVEIRA, D. et al. Radiation-Induced Error Criticality in Modern HPC Parallel Accelerators. In: ACM. **Proceedings of 21st IEEE Symp. on High Performance Computer Architecture (HPCA)**. [S.l.], 2017.

OLIVEIRA, D. et al. Modern GPUs radiation sensitivity evaluation and mitigation through duplication with comparison. **Nuclear Science, IEEE Transactions on**, v. 61, n. 6, p. 3115–3122, Dec 2014. ISSN 0018-9499.

Oliveira, D. A. G. et al. Radiation-induced error criticality in modern hpc parallel accelerators. In: **2017 IEEE HPCA**. [S.l.: s.n.], 2017. p. 577–588.

PALAZZI, L. et al. A tale of two injectors: End-to-end comparison of ir-level and assembly-level fault injection. In: **2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)**. [S.l.: s.n.], 2019. p. 151–162.

Palazzi, L. et al. A tale of two injectors: End-to-end comparison of ir-level and assembly-level fault injection. In: **2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)**. [S.l.: s.n.], 2019. p. 151–162.

PATTABIRAMAN, K. et al. Dynamic derivation of application-specific error detectors and their implementation in hardware. In: **2006 Sixth European Dependable Computing Conference**. [S.l.: s.n.], 2006. p. 97–108.

PHATAK, D. S.; KOREN, I. Complete and partial fault tolerance of feedforward neural nets. **IEEE Transactions on Neural Networks**, v. 6, n. 2, p. 446–456, Mar 1995. ISSN 1045-9227.

PILLA, L. L. et al. Software-based hardening strategies for neutron sensitive fft algorithms on gpus. **IEEE Transactions on Nuclear Science**, v. 61, n. 4, p. 1874–1880, Aug 2014. ISSN 0018-9499.

PIURI, V. Analysis of fault tolerance in artificial neural networks. **J. Parallel Distrib. Comput.**, Academic Press, Inc., Orlando, FL, USA, v. 61, n. 1, p. 18–48, jan. 2001. ISSN 0743-7315. Available from Internet: <http://dx.doi.org/10.1006/jpdc.2000.1663>.

Previlon, F. G. et al. Evaluating the impact of execution parameters on program vulnerability in GPU applications. In: **2018 Design, Automation Test in Europe Conference Exhibition (DATE)**. [S.l.: s.n.], 2018. p. 809–814.

PROTZEL, P. W.; PALUMBO, D. L.; ARRAS, M. K. Performance and fault-tolerance of neural networks for optimization. **IEEE Transactions on Neural Networks**, v. 4, n. 4, p. 600–614, Jul 1993. ISSN 1045-9227.

PUENTE, J. de la et al. Mimetic seismic wave modeling including topography on deformed staggered grids. **GEOPHYSICS**, v. 79, n. 3, p. T125–T141, 2014. Available from Internet: <http://dx.doi.org/10.1190/geo2013-0371.1>.

QUINN, H. et al. Radiation-induced multi-bit upsets in sram-based fpgas. **IEEE Transactions on Nuclear Science**, v. 52, n. 6, p. 2455–2461, 2005.

RAINA, R.; MADHAVAN, A.; NG, A. Y. Large-scale deep unsupervised learning using graphics processors. In: **Proceedings of the 26th Annual International Conference on Machine Learning**. New York, NY, USA: ACM, 2009. (ICML '09), p. 873–880. ISBN 978-1-60558-516-1. Available from Internet: <http://doi.acm.org/10.1145/1553374.1553486>.

RECH, P. et al. An Efficient and Experimentally Tuned Software-Based Hardening Strategy for Matrix Multiplication on GPUs. **Nuclear Science, IEEE Transactions on**, v. 60, n. 4, p. 2797–2804, 2013. ISSN 0018-9499.

RECH, P. et al. Measuring the Radiation Reliability of SRAM Structures in GPUS Designed for HPC. In: **IEEE 10th Workshop on Silicon Errors in Logic - System Effects (SELSE)**. [S.l.: s.n.], 2014.

RECH, P. et al. Threads distribution effects on graphics processing units neutron sensitivity. **Nuclear Science, IEEE Transactions on**, v. 60, n. 6, p. 4220–4225, Dec 2013. ISSN 0018-9499.

RECH, P. et al. Impact of GPUs Parallelism Management on Safety-Critical and HPC Applications Reliability. In: **IEEE Int. Conference on Dependable Systems and Networks (DSN 2014)**. Atlanta, USA: [s.n.], 2014.

REDMON, J. et al. You only look once: Unified, real-time object detection. **CoRR**, abs/1506.02640, 2015. Available from Internet: <http://arxiv.org/abs/1506.02640>.

REDMON, J.; FARHADI, A. Yolo9000: Better, faster, stronger. **arXiv preprint arXiv:1612.08242**, 2016.

REDMON, J.; FARHADI, A. Yolov3: An incremental improvement. **CoRR**, abs/1804.02767, 2018. Available from Internet: <http://arxiv.org/abs/1804.02767>.

REIS, G. et al. Design and evaluation of hybrid fault-detection systems. In: **32nd International Symposium on Computer Architecture (ISCA'05)**. [S.l.: s.n.], 2005. p. 148–159.

REN, S. et al. Faster R-CNN: Towards real-time object detection with region proposal networks. In: **Advances in Neural Information Processing Systems (NIPS)**. [S.l.: s.n.], 2015.

RIBEIRO, D. et al. A real-time pedestrian detector using deep learning for human-aware navigation. **CoRR**, abs/1607.04441, 2016. Available from Internet: <http://arxiv.org/abs/1607.04441>.

RIESENHUBER, M.; POGGIO., T. Hierarchical models of object recognition in cortex. Nature Neuroscience, 1999. Available from Internet: <https://www.ncbi.nlm.nih.gov/pubmed/10526343>.

RODRIGUES, G. et al. Approximate TMR based on successive approximation and loop perforation in microprocessors. **Microelectronics Reliability**, v. 100-101, p. 113385, 2019. ISSN 0026-2714. Available from Internet: <http://www.sciencedirect.com/science/article/pii/S0026271419304706>.

Saeng-Hwan Kim et al. A low power and highly reliable 400mbps mobile ddr sdram with on-chip distributed ecc. In: **2007 IEEE Asian Solid-State Circuits Conference**. [S.l.: s.n.], 2007. p. 34–37.

Santos, F. F. d. et al. Analyzing and increasing the reliability of convolutional neural networks on GPUs. **IEEE Transactions on Reliability**, v. 68, n. 2, p. 663–677, 2019.

SANTOS, F. F. dos et al. **DSN 2021 Data repository**. 2021. <https://github.com/UFRGS-CAROL/dsn_2021_data>.

SANTOS, F. F. dos et al. Reliability evaluation of mixed-precision architectures. In: **2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)**. [S.l.: s.n.], 2019. p. 238–249.

SANTOS, F. F. dos et al. **Radiation Benchmarks Repository**. [S.l.]: GitHub, 2014. <https://github.com/UFRGS-CAROL/radiation-benchmarks>.

SANTOS, F. F. dos et al. **Transaction on Reliability Repository**. [S.l.]: GitHub, 2018. <https://github.com/UFRGS-CAROL/transaction_on_reliability_2018>.

SANTOS, F. F. dos et al. **IPDPS Repository**. [S.l.]: GitHub, 2021. <https://github.com/UFRGS-CAROL/ipdps2021>.

SARTOR, A. L.; BECKER, P. H.; BECK, A. C. A fast and accurate hybrid fault injection platform for transient and permanent faults. **Des. Autom. Embedded Syst.**, Kluwer Academic Publishers, USA, v. 23, n. 1–2, p. 3–19, jun. 2019. ISSN 0929-5585. Available from Internet: <https://doi.org/10.1007/s10617-018-9217-0>.

SCHERER, D.; MÜLLER, A.; BEHNKE, S. Evaluation of pooling operations in convolutional architectures for object recognition", booktitle="artificial neural networks – icann 2010: 20th international conference, thessaloniki, greece, september 15-18, 2010, proceedings, part iii. In: ____. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010. p. 92–101. ISBN 978-3-642-15825-4. Available from Internet: <http://dx.doi.org/10.1007/978-3-642-15825-4_10>.

Seetharam, K. et al. Applying reduced precision arithmetic to detect errors in floating point multiplication. In: **2013 IEEE 19th Pacific Rim International Symposium on Dependable Computing**. [S.l.: s.n.], 2013. p. 232–235. ISSN null.

SEIFERT, N.; ZHU, X.; MASSENGILL, L. W. Impact of scaling on soft-error rates in commercial microprocessors. **Nuclear Science, IEEE Transactions on**, IEEE, v. 49, n. 6, p. 3100–3106, 2002.

SEQUIN, C. H.; CLAY, R. D. Fault tolerance in artificial neural networks. In: **1990 IJCNN International Joint Conference on Neural Networks**. [S.l.: s.n.], 1990. p. 703–708 vol.1.

Serrano-Cases, A. et al. Empirical mathematical model of microprocessor sensitivity and early prediction to proton and neutron radiation-induced soft errors. **IEEE Transactions on Nuclear Science**, v. 67, n. 7, p. 1511–1520, 2020.

SRIDHARAN, V. et al. Memory errors in modern systems: The good, the bad, and the ugly. **SIGARCH Comput. Archit. News**, Association for Computing Machinery, New York, NY, USA, v. 43, n. 1, p. 297–310, mar. 2015. ISSN 0163-5964. Available from Internet: <https://doi.org/10.1145/2786763.2694348>.

SRIDHARAN, V.; KAELI, D. R. The effect of input data on program vulnerability. In: **Proceedings of the 2009 Workshop on Silicon Errors in Logic and System Effects**. [S.l.: s.n.], 2009. (SELSE '09).

SRIDHARAN, V.; KAELI, D. R. Using hardware vulnerability factors to enhance avf analysis. In: **Proceedings of the 37th Annual International Symposium on Computer Architecture**. New York, NY, USA: ACM, 2010. (ISCA '10), p. 461–472. ISBN 978-1-4503-0053-7. Available from Internet: <http://doi.acm.org/10.1145/1815961.1816023>.

STRZODKA, R.; GODDEKE, D. Pipelined mixed precision algorithms on fpgas for fast and accurate pde solvers from low precision components. In: **2006 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines**. [S.l.: s.n.], 2006. p. 259–270.

SUBASI, O. et al. Characterizing the impact of soft errors affecting floating-point alus using rtl-ievel fault injection. In: **Proceedings of the 47th International Conference on Parallel Processing**. New York, NY, USA: Association for Computing Machinery, 2018. (ICPP 2018). ISBN 9781450365109. Available from Internet: <https://doi.org/10.1145/3225058.3225089>.

SULLIVAN, M. B. **Low-cost duplication for separable error detection in computer arithmetic**. 51–69 p. Thesis (PhD) — The University of Texas at Austin, 2015.

SULLIVAN, M. B. et al. Characterizing and mitigating soft errors in gpu dram. In: **MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture**. New York, NY, USA: Association for Computing Machinery, 2021. (MICRO '21), p. 641–653. ISBN 9781450385572. Available from Internet: <https://doi.org/10.1145/3466752.3480111>.

SZAFARYN, L. G. et al. **Experiences with Achieving Portability across Heterogeneous Architectures**. 2010.

TCHERNEV, E. B.; MULVANEY, R. G.; PHATAK, D. S. Investigating the fault tolerance of neural networks. **Neural Comput.**, MIT Press, Cambridge, MA, USA, v. 17, n. 7, p. 1646–1664, jul. 2005. ISSN 0899-7667. Available from Internet: <http://dx.doi.org/10.1162/0899766053723096>.

TIWARI, D. et al. Understanding GPU Errors on Large-scale HPC Systems and the Implications for System Design and Operation. In: ACM. **Proceedings of 21st IEEE Symp. on High Performance Computer Architecture (HPCA)**. [S.l.], 2015.

TONG, J. Y. F.; NAGLE, D.; RUTENBAR, R. A. Reducing power by optimizing the necessary precision/range of floating-point arithmetic. **IEEE Transactions on Very Large Scale Integration (VLSI) Systems**, v. 8, n. 3, p. 273–286, June 2000. ISSN 1063-8210.

Traiola, M. et al. Predicting the impact of functional approximation: from component- to application-level. In: **IOLTS**. [S.l.: s.n.], 2018. p. 61–64. ISSN 1942-9401.

TSAI, T. et al. Nvbitfi: Dynamic fault injection for gpus. In: **2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)**. [S.l.: s.n.], 2021. p. 284–291.

TSELONIS, S.; GIZOPOULOS, D. Gufi: A framework for gpus reliability assessment. In: **2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)**. [S.l.: s.n.], 2016. p. 90–100.

Ubal, R. et al. Multi2sim: A simulation framework for cpu-gpu computing. In: **2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)**. [S.l.: s.n.], 2012. p. 335–344.

UKIDAVE, Y. et al. Nupar: A benchmark suite for modern gpu architectures. In: **Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering**. New York, NY, USA: Association for Computing Machinery, 2015. (ICPE '15), p. 253–264. ISBN 9781450332484. Available from Internet: <https://doi.org/10.1145/2668930.2688046>.

Vallero, A.; Gizopoulos, D.; Di Carlo, S. Sifi: Amd southern islands gpu microarchitectural level fault injector. In: **2017 IEEE 23rd International Symposium on On-Line Testing and Robust System Design (IOLTS)**. [S.l.: s.n.], 2017. p. 138–144.

VENKATESH, G.; NURVITADHI, E.; MARR, D. Accelerating deep convolutional networks using low-precision and sparsity. In: **2017 IEEE Int. Conference on Acoustics, Speech and Signal Processing (ICASSP)**. [S.l.: s.n.], 2017. p. 2861–2865. ISSN 2379-190X.

WADDEN, J. et al. Real-world design and evaluation of compiler-managed gpu redundant multithreading. In: **Proceeding of the 41st Annual International Symposium on Computer Architecuture**. [S.l.]: IEEE Press, 2014. (ISCA '14), p. 73–84. ISBN 9781479943944.

WADDEN, J. et al. Real-world design and evaluation of compiler-managed GPU redundant multithreading. In: **2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)**. [S.l.: s.n.], 2014. p. 73–84. ISSN 1063-6897.

Wei, J. et al. 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**. [S.l.: s.n.], 2014. p. 375–382.

WUNDERLICH, H. J.; BRAUN, C.; HALDER, S. Efficacy and efficiency of algorithm-based fault-tolerance on gpus. In: **On-Line Testing Symposium (IOLTS), 2013 IEEE 19th Int.** [S.l.: s.n.], 2013. p. 240–243.

YANG, L. et al. Enabling software resilience in gpgpu applications via partial thread protection. In: **2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)**. Los Alamitos, CA, USA: IEEE Computer Society, 2021. p. 1248–1259. ISSN 1558-1225. Available from Internet: <https://doi.ieeecomputersociety.org/10.1109/ICSE43902.2021.00114>.

YANG, L. et al. Practical resilience analysis of gpgpu applications in the presence of single-and multi-bit faults. **IEEE Transactions on Computers**, v. 70, n. 1, p. 30–44, 2021.

Yazdanbakhsh, A. et al. Axbench: A multiplatform benchmark suite for approximate computing. **IEEE Design Test**, v. 34, n. 2, p. 60–68, 2017.

YIM, K. S. et al. Hauberk: Lightweight silent data corruption error detector for gpgpu. In: **2011 IEEE International Parallel Distributed Processing Symposium**. [S.l.: s.n.], 2011. p. 287–300.

ZHANG, Y.; NATHAN, R.; SORIN, D. J. Reduced precision checking to detect errors in floating point arithmetic. **CoRR**, abs/1510.01145, 2015. Available from Internet: <http://arxiv.org/abs/1510.01145>.

ZIEGLER, J. F.; PUCHNER, H. **SER–history, Trends and Challenges: A Guide for Designing with Memory ICs**. [S.l.]: Cypress, 2010.

APPENDIX A — RESUMO EXPANDIDO

Graphics Processing Units (GPUs), que anteriormente eram usadas como hardware restrito a aplicações voltadas ao entretenimento e renderização gráfica, onde confiabilidade não é um requisito fundamental, estão sendo utilizadas em *High Performance Computing* (HPC) e aplicações críticas como *Convolutional Neural Networks* (CNNs) para carros autoguiados e exploração espacial. Tal mudança no mercado das GPUs foi causada pelo aumento da capacidade de processamento e eficiência, melhorias nos frameworks de programação e ferramentas de validação, e também uma maior preocupação com a confiabilidade das GPUs. Neste cenário, esta pesquisa tem como objetivo entender o processo de geração, propagação, e o impacto de falhas causadas nas GPUs. O conhecimento adquirido ao longo deste trabalho foi usado para propor técnicas de detecção e correção de erros para GPUs.

Com o objetivo de investigar a confiabilidade das GPUs, este trabalho compara e combina os resultados de um número considerável de experimentos de radiação, injeção de falhas, e de análises arquitetural e algorítmica. Ao total oito experimentos de radiação foram realizados para esta tese, totalizando mais de 2,000 horas de experimentos, que equivalem a mais de 13,000,000 anos de radiação em um ambiente com fluxo equivalente ao nível do mar. Este trabalho também é o primeiro a experimentalmente validar o *Failure In Time (FIT)* das CNNs. Entretanto, experimentos de radiação, quando realizados isoladamente, não são suficientes para prover informações sobre a propagação das falhas, sendo assim este trabalho também estuda sobre propagação de falhas nas GPUs através de injeção de falhas nos níveis arquiteturais (usando simulações em *Register Transfer Level - RTL*) e de software (usando as ferramentas SASSIFI e NVBITFI). A combinação de experimentos de radiação e injeção de falhas tem o objetivo de responder duas questões fundamentais na avaliação da confiabilidade de GPUs: Como é possível extrair resultados representativos e um estimar o FIT de uma forma realística usando somente injeção de falhas, e também, validar se os tipos de modelo de falhas mais utilizados atualmente, bit flip único ou duplo, são ou não modelos acurados para simulação de falhas no nível de software para GPUs. Os resultados mostram que, para a maioria dos casos, a taxa de *Silent Data Corruption* (SDC) extraída da injeção de falhas é suficientemente próxima (diferenças menores que $5\times$) da taxa de SDCs obtida em experimentos de radiação. Este trabalho também propõe um novo modelo de falhas baseado nos erros relativos extraídos de injeções de falhas usando RTL em oposição aos comumente usados, bit flip único e

duplo.

Usando uma análise experimental, arquitetural, e algorítmica, este trabalho propõe novas soluções de tolerância a falhas para HPC e aplicações críticas. Para isso, usa-se o fato de que nem todas as falhas manifestadas na saída de uma aplicação são críticas, por exemplo, um veículo autônomo não é afetado por uma falha que modifica um objeto que não seria detectado mesmo sem a presença de falhas, devido à baixa probabilidade de detecção que alguns objetos possuem, não sendo incluídos na detecção final. O mesmo ocorre se o objeto detectado tem uma alta probabilidade de detecção e tem a forma alterada pela falha, porém mesmo assim, tem um formato próximo ao esperado em um cenário sem falhas. Fenômeno similar também acontece para aplicações HPC, onde uma serie de aplicações aceitam pequenas variações na saída como parcialmente "corretas". Sendo assim, este trabalho usa as validações de confiabilidade previamente feitas em diferentes camadas (software e hardware) para identificar os recursos ou partes do código que são mais críticos para a execução de uma aplicação, e então proteger somente o que é necessário. Outro fato que baseia a proposição de novas técnicas, é que as técnicas existentes de tolerância a falhas para GPUs não são suficientes para garantir uma grande confiabilidade para aplicações críticas. Aliado a isso, diversos recursos existentes nas GPUs atuais são por muitas vezes não utilizados, e podem ser empregados para replicação. Os experimentos feitos para esta tese mostram que *Single Error Correction Double Error Detection ECC* consegue mascarar falhas nas estruturas de memória das GPUs, porém os erros críticos (que tem um impacto grande na saída) são gerados por falhas em outros recursos não protegidos como, unidades funcionais, escalonadores, filas e buffers. Consequentemente, este trabalho aproveita-se das estruturas paralelas existentes nas GPUs para implementar tolerância a falhas de forma eficiente e de propósito geral que sejam capazes de detectar e corrigir erros críticos.

Baseado nos estudos feitos neste trabalho, duas soluções principais de tolerância a falhas foram propostas: A primeira, uma solução específica para CNNs foi desenvolvida em software capaz de detectar e corrigir erros, onde foi usado uma técnica já conhecida chamada de *Algorithm-Based Fault Tolerance* (ABFT) para multiplicação de matrizes, que nos experimentos realizados nesta tese mostrou-se responsável pela maioria dos erros críticos das CNNs. Quando integrada as CNNs, ABFT consegue corrigir mais de 60% dos erros que modificam a saída e que por consequência modificam o comportamento de um carro autônomo. Também, a partir de experimentos de injeção de falhas foi observado que as camadas *maxpool* são responsáveis por propagarem a maioria das falhas críticas

em uma CNN, sendo assim, uma camada modificada da *maxpool* foi proposta para incluir detecção de erros, a *smartpool*, que consegue detectar 98% dos SDCs. A segunda solução baseia-se no fato que as GPUs modernas possuem hardware dedicado para executar operações em precisões diferentes, e quando um recurso específico para uma certa precisão é utilizado, os recursos para outras precisões ficam ociosos. Sabendo disso, este trabalho propõe a *Reduced Precision Duplication With Comparison* (RP-DWC), onde o principal objetivo é reduzir a sobrecarga da redundância, executando a cópia em uma precisão reduzida. Os resultados mostraram que o RP-DWC teve uma excelente taxa de detecção de erros, podendo chegar a 86%, com sobrecargas mínimas de até 0.1% de aumento de tempo de execução e para alguns casos 24% de aumento de consumo de energia. Para validar as técnicas propostas neste trabalho foram usados experimentos de radiação, com o intuito de garantir que os resultados obtidos são de fato realísticos.

Dessa forma, neste trabalho foi proposta uma metodologia para investigar a confiabilidade das GPUs e dispositivos paralelos no geral. Foi mostrado como integrar experimentos de radiação e injeção de falhas, e também que com um entendimento detalhado da confiabilidade dos dispositivos e algoritmos é possível desenvolver soluções de tolerância a falhas eficientes e efetivas. Este trabalho, então, pode servir de base para análises futuras de GPUs ou outro dispositivo paralelo, uma vez que além dos resultados, todos os dados e aplicações utilizadas estão públicos em repositórios online, com o objetivo de facilitar futuras validações e acesso de terceiros.

APPENDIX B — PUBLICATIONS

Below are listed the papers published as a result of this thesis. The first set of articles are the ones published by the author of this work as the **first author**.

dos Santos FF, Rech P. *Analyzing the criticality of transient faults-induced SDCS on GPU applications*. In **Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems 2017** Nov 12 (pp. 1-7).

dos Santos FF, Draghetti L, Weigel L, Carro L, Navaux P, Rech P. *Evaluation and mitigation of soft-errors in neural network-based object detection in three GPU architectures*. In **2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)** 2017 Jun 26 (pp. 169-176). IEEE.

dos Santos FF, Pimenta PF, Lunardi C, Draghetti L, Carro L, Kaeli D, Rech P. *Analyzing and increasing the reliability of convolutional neural networks on GPUs*. **IEEE Transactions on Reliability**. 2018 Nov 15;68(2):663-77.

dos Santos FF, Carro L, Rech P. *Kernel and layer vulnerability factor to evaluate object detection reliability in GPUs*. **IET Computers & Digital Techniques**. 2019 May;13(3):178-86.

dos Santos FF, Lunardi C, Oliveira D, Libano F, Rech P. *Reliability evaluation of mixed-precision architectures*. In **2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)** 2019 Feb 16 (pp. 238-249). IEEE.

dos Santos FF, Navaux P, Carro L, Rech P. *Impact of reduced precision in the reliability of deep neural networks for object detection*. In **2019 IEEE European Test Symposium (ETS)**s 2019 May 27 (pp. 1-6). IEEE.

dos Santos FF, Brandalero M, Basso PM, Hubner M, Carro L, Rech P. *Reduced-Precision DWC for Mixed-Precision GPUs*. In **2020 IEEE 26th International Symposium on On-Line Testing and Robust System Design (IOLTS)** 2020 Jul 13 (pp. 1-6). IEEE.

dos Santos FF, Hari SK, Basso PM, Carro L, Rech P. *Demystifying gpu reliability: comparing and combining beam experiments, fault simulation, and profiling*. In **2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)** 2021 May 17 (pp. 289-298). IEEE.

dos Santos FF, Condia JE, Carro L, Reorda MS, Rech P. *Revealing GPUs Vulnerabilities by Combining Register-Transfer and Software-Level Fault Injection*. In **2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)** 2021 Jun 21 (pp. 292-304). IEEE.

dos Santos FF, Brandalero, M., Sullivan, M., Junior, R. L. R., Basso, P. M., Hubner, M., Carro L., Rech P. (2021). *Reduced precision DWC: an efficient hardening strategy for mixed-precision architectures*. **IEEE Transactions on Computers**. 2021 Feb 15 (pp. 1-1) IEEE.

dos Santos FF, Malde S, Cazzaniga C, Frost C, Carro L, Rech P. *Experimental Findings on the Sources of Detected Unrecoverable Errors in GPUs*. **arXiv preprint arXiv:2108.00554**. 2021 Aug 1.

The following articles were published as **colaborator**:

De Oliveira DA, Pilla LL, Hanzich M, Fratin V, Fernandes F, Lunardi C, Cela JM, Navaux PO, Carro L, Rech P. *Radiation-induced error criticality in modern HPC parallel accelerators*. In **2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)** 2017 Feb 4 (pp. 577-588). IEEE.

Fratin V, Oliveira D, Lunardi C, Santos F, Rodrigues G, Rech P. *Code-dependent and architecture-dependent reliability behaviors*. In **2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)** 2018 Jun 25 (pp. 13-26). IEEE.

Basso PM, dos Santos FF, Rech P. *Impact of tensor cores and mixed precision on the reliability of matrix multiplication in GPUs*. **IEEE Transactions on Nuclear Science**. 2020 Mar 2;67(7):1560-5.

Oliveira D, Blanchard S, DeBardeleben N, dos Santos FF, Davila GP, Navaux P, Wender S, Cazzaniga C, Frost C, Baumann R, Rech P. *An Overview of the Risk Posed by Thermal Neutrons to the Reliability of Computing Devices*. In **2020 50th Annual IEEE-IFIP International Conference on Dependable Systems and Networks-Supplemental** Volume (DSN-S) 2020 Jun 29 (pp. 92-97). IEEE.

Oliveira D, Blanchard S, DeBardeleben N, Fernandes dos Santos F, Piscoya Dávila G, Navaux P, Favalli A, Schappert O, Wender S, Cazzaniga C, Frost C. *Thermal neutrons: a possible threat for supercomputer reliability*. **The Journal of Supercomputing**. 2021 Feb;77:1612-34.

Condia JE, Rech P, dos Santos FF, Carrot L, Reorda MS. *Protecting GPU's Microarchitectural Vulnerabilities via Effective Selective Hardening*. In **2021 IEEE 27th International Symposium on On-Line Testing and Robust System Design (IOLTS)** 2021 Jun 28 (pp. 1-7). IEEE.

Condia JE, dos Santos FF, Reorda MS, Rech P. *Combining Architectural Simulation and Software Fault Injection for a Fast and Accurate CNNs Reliability Evaluation on GPUs*. In **2021 IEEE 39th VLSI Test Symposium (VTS)** 2021 Apr 25 (pp. 1-7). IEEE.