Politecnico
di Torino

ScuDo
Scuola di Dottorato ~ Doctoral School
WHAT YOU ARE, TAKES YOU FAR

Doctoral Dissertation
Doctoral Program in Computer and Control Engineering (36^{th}cycle)

Reliability Enhancement in GPU Architectures

By

Juan David Guerrero Balaguera

******

Supervisor(s):
Prof. Matteo Sonza Reorda, Supervisor
Prof. Ernesto Sanchez Sanchez, Co-Supervisor

Doctoral Examination Committee:
Prof. Luigi Carro, Referee, Universidade Federal do Rio Grande do Sul
Prof. Haralampos Stratigopoulos, Referee, Sorbonne Université, CNRS, LIP6
Prof. Antonio Miele, Politecnico di Milano
Prof. Leticia M. Bolzani Poehls, RWTH Aachen University
Prof. Massimo Violante, Politecnico di Torino

Politecnico di Torino
2024

Declaration

I hereby declare that, the contents and organization of this dissertation constitute my own original work and does not compromise in any way the rights of third parties, including those relating to the security of personal data.

Juan David Guerrero Balaguera
2024

* This dissertation is presented in partial fulfillment of the requirements for **Ph.D. degree** in the Graduate School of Politecnico di Torino (ScuDo).

To my mother ***Ana Leonilde Balaguera Cepeda***, and in the eternal memory of my beloved father, **Samuel Guerrero Báez**.

Acknowledgements

First and foremost, I am extremely grateful to my supervisor, Prof. Matteo Sonza Reorda, for his invaluable advice, continuous support, and patience during my PhD journey. His excellent knowledge and plentiful experience have encouraged me throughout my academic research and daily life. I would also like to thank my friend and colleague, Josie Esteban Rodriguez Condia, for his valuable contribution and willingness to discuss new ideas and future research frontiers. I would like to thank Prof. Marco Levorato for his support and collaboration during my stay as a visitor scholar in the Department of Computer Sciences at the University of California Irvine. I am grateful to Prof. Paolo Rech and Fernando Fernandes dos Santos for their invaluable feedback and contribution, which allowed me to achieve successful results. Finally, I want to express my sincere gratitude to my brothers, Francisco Javier and Oscar Andres, who took care of my parents while I was away pursuing my dreams; without their tremendous understanding and support, it would have been impossible to complete this endeavor. Finally, I would like to thank my girlfriend, Audrey, for giving me a different view of the world and inspiring me to continue the academic path.

Abstract

GPUs are important hardware accelerators for modern applications, particularly AI-based ones. They offer a high degree of parallelism, allowing multiple data to be processed simultaneously in a single chip. This is made possible by continuous technology scaling, resulting in higher transistor densities (e.g., 80 billion transistors for NVIDIA Hopper GPUs and 100 billion transistors for Intel GPUs). GPU devices offer great computational performance but can suffer from reliability issues associated with modern semiconductor technologies. As technology scales, several threats, such as accelerated wear-out, premature degradation, and high-temperature conditions, can increase the risk of hardware defects that lead to permanent faults. GPUs are now used in critical applications such as autonomous driving systems, aerospace, and avionics, among others. The reliability of GPUs is paramount, especially when used in such scenarios where they are required to operate correctly for longer lifetimes than typical consumer applications.

The reliability of GPU devices can be improved by implementing functional safety mechanisms (hardware or software) that can detect faults before they produce critical failures. These methods can ultimately reduce the probability of failure to acceptable levels. Unfortunately, hardware-based approaches require the addition of extra hardware structures to the device, which can increase costs, impact performance, or affect power consumption. Alternatively, the software-based self-testing (SBST) strategy is a flexible and noninvasive approach that offers in-field at-speed fault detection capabilities without hardware costs, leveraging the application's idle times to execute test procedures. Recently, several works have successfully demonstrated the feasibility of SBST for the development of Software Test Libraries (STLs), exploiting the inherent parallelism of GPUs for testing purposes and targeting functional units, memory modules, and control units.

Typically, the development of STLs for GPUs usually resorts to assembly languages, only. High-level programming languages (HLLs) for GPUs, such as CUDA C++ or OpenCL, simplify programming and are often the best, and sometimes the only, way to develop and encode applications. However, there are still several challenges and open questions when it comes to using HLLs for the development of STLs.

In this regard, this PhD thesis makes two main contributions. Firstly, it employs high-level (e.g., CUDA C++) or intermediate (e.g., CUDA PTX) programming languages to develop or map SBST strategies that are designed to test specific hardware components in GPUs. Secondly, it devises alternative strategies for compacting test programs, which help to reduce their memory footprint and speed up their test duration when used for in-field testing purposes.

On the other hand, techniques to identify permanent faults that can produce errors during the device's operative life are strongly required, since the typical simulation-based approaches result in prohibitive evaluation time. Additionally, these fault evaluations are crucial for two main reasons: first, they allow the identification of vulnerabilities in GPU's application regarding permanent faults, contributing to the development of effective software-based hardening strategies. Second, they can be used to assess the effectiveness of any software-based fault countermeasure against errors caused by permanent faults in GPUs.

Accordingly, this Ph.D. thesis proposes a method for evaluating the reliability of GPU applications in the presence of Silent Data Errors (SDEs) caused by permanent faults. The proposed method involves multi-level fault evaluations, which provide a better trade-off between accuracy (which is typically higher when we move closer to the hardware) and fast fault evaluations compared to other methods.

In conclusion, the thesis work proposes methods that aim to enhance the reliability of GPUs, taking a step forward from the current state-of-the-art. These methods include strategies for generating and compacting SBST for in-field GPU testing, as well as assessing the impact of permanent faults on GPU workloads. The effectiveness and limitations of these methods were evaluated through experiments on a set of representative benchmarks and compared with alternative solutions currently available.

Contents

List of Figures xi

List of Tables xiii

Nomenclature xv

1 Introduction 1

1.1 Architectural organization of GPUs 4

1.2 Hardware faults and GPU's reliability 9

1.2.1 Reliability enhancement of GPUs 10

1.3 In-field testing of GPUs 12

1.3.1 Software-Based Self-Testing 13

1.3.1.1 Related works 13

1.3.1.2 Main contributions 15

1.3.2 Compaction of STLs for GPU testing 15

1.3.2.1 Related works 16

1.3.2.2 Main contributions 16

1.4 Reliability evaluation of GPUs with respect to permanent faults . . . 17

1.4.1 Related works 19

1.4.2 Main contributions 20

1.5 Thesis contributions summary 20

1.6 Thesis organization 21

2 STLs for GPUs using high-level programming languages 23

2.1 Background 24

2.1.1 Related works 25

2.2 A Method to develop parallel STLs using high-level languages . . . 27

2.2.1 Modular test generation 27

2.2.2 Programming language mapping 28

2.2.3 Test program evaluation 30

2.2.4 Defeating compiler and architectural constraints for testing purposes 31

2.3 HLL's mapping of test algorithms for representative GPU units . . . 33

2.3.1 Functional Units and Register File 33

2.3.2 Embedded Memories 34

2.3.3 Warp Scheduler Memory 35

2.3.4 Divergence Stack Memory 36

2.3.5 Pipeline Registers 37

2.3.6 Decoder Unit 38

2.4 Experimental results 38

2.5 Final remarks 43

3 Compaction of STLs for GPUs 45

3.1 Background 46

3.1.1 Related works 47

3.2 Compaction of STLs for CPUs: main concept 49

3.2.1 Proposed methodology 49

3.2.2 Study cases 54

3.2.3 Experimental results 57

3.3 Compaction of STLs for GPUs 61
3.3.1 Proposed metodology 61
3.3.2 Study case 66
3.3.3 Experimental results 69
3.4 Final remarks 74

4 Modeling and evaluating error effects due to permanent faults on GPUs 75
4.1 Background 77
4.1.1 Related works 78
4.2 Proposed multi-level evaluation methodology: main idea 81
4.3 Evaluation of permanent faults on GPU's SP cores 82
4.3.1 Proposed methodology 82
4.3.1.1 Software/Hardware profiling 83
4.3.1.2 Gate-level fault simulation 83
4.3.1.3 Error identification and classification 85
4.3.1.4 Instruction-level error propagation 85
4.3.1.5 Applications evaluation 88
4.3.2 Experimental results 88
4.3.2.1 Software profiling of SP cores instructions 88
4.3.2.2 Gate-level micro-architecture fault simulations results 89
4.3.2.3 Syndrome tables as instruction-level errors 91
4.3.2.4 Error propagation results on SP cores 92
4.4 Evaluation of permanent faults on GPU's parallel management units 96
4.4.1 Proposed methodology 96
4.4.1.1 Hardware unit profiling 96
4.4.1.2 Gate-level fault simulation 97

4.4.1.3 Error identification and classification 97
4.4.1.4 Instruction-level error propagation 98
4.4.1.5 Applications evaluation 99
4.4.2 Fault characterization results 99
4.4.3 Software-based error propagation results 104
4.4.3.1 Errors implementation/propagation 105
4.4.3.2 Error injection and propagation results 110
4.5 Final remarks 115

5 Conclusions 117

References 123

Appendix A Publication list of the Author 136

A.1 Journal Publications 136
A.2 Conference Proceedings Publications 137

List of Figures

1.1 Global Semiconductor Market (source McKinsey & Company [1]). . 2

1.2 GPU market size worldwide 2022-2032 (source Statista [2]). 3

1.3 A general scheme of the internal organization of a GPU. 4

1.4 Internal components distribution of a Parallel Processing Block (PPB) of an SM. 5

1.5 Thread hierarchy in CUDA programming. 7

1.6 Thread scheduling under the SIMT warp execution model of Volta NVIDIA GPUs. 8

1.7 Product failure rate increases as the technology scales. 10

2.1 A general scheme of the proposed methodology to generate STLs for GPUs using high-level programming languages (adapted from [3]). 26

2.2 An example of mapping the test strategies for the Divergence Stack. 36

3.1 A general scheme of the proposed compaction approach applied to STLs developed for in-field testing of CPUs (adapted from [4]). . . 50

3.2 A general scheme of the proposed compaction approach for functional TPs in GPUs (adapted from [5]). 62

4.1 A general scheme of the proposed multi-level method to evaluate permanent faults in GPUs. 82

4.2 Description of Syndrome Table implementation. 87

4.3 Average results of the error propagation of permanent fault effects on the instructions of a DNN. 93

4.4 A general scheme of the method to characterize fault effects in parallelism management units of GPUs (adapted from [6]). 96

4.5 Fault Activation and Propagation Rate (FAPR) for the identified faults as SW errors in the WSC, fetch, and decoder units. 104

4.6 Description of IRA/IVRA error models (adapted from [6]). 106

4.7 Description of IAT/IAW/IAC error models (adapted from [6]). . . . 107

4.8 Description of IAL error models (adapted from [6]). 108

4.9 Description of IOC/IIO/IMS/IMD/WV error models (adapted from [6]). 109

4.10 Error Propagation Rate results of each error model propagated on 15 applications 112

4.11 Average EPR among the 15 tested applications (adapted from [7]). . 113

List of Tables

2.1 Summary of the mapping strategies from the test algorithm to high-level programming languages and intermediate-level programming Languages (adapted from [3]). 32

2.2 GPU modules features and their STL development approach (adapted from [3]). 39

2.3 Main features of the implemented STLs for regular units and embedded memories of the GPU (adapted from [3]). 40

2.4 Main features of the implemented STLs for the special units of a GPU (adapted from [3]). 41

3.1 Number of faults per module in the RISCY processor (adapted from [4]). 54

3.2 Main features of the considered test programs (adapted from [4]). . 55

3.3 The compaction results in the test programs for the *Execute* unit (adapted from [4]). 57

3.4 The compaction process results of each STL in the *admissible* region (adapted from [4]). 59

3.5 Results showing the compaction impact on the entire STL (adapted from [4]). 60

3.6 Main features of the evaluated parallel Test Programs (PTPs) for the GPU (adapted from [5]). 67

3.7 The compaction results in the test programs for the *Decoder* unit (adapted from [5]). 70

3.8 The compaction results in the test programs for the *Scalar Processors* and *Special Function Units* (adapted from [5]). 71

4.1 Evaluation strategies for reliability assessment concerning hardware faults on digital systems 79

4.2 Fault rate and percentage of input patterns exciting a permanent fault in the FP32 and INT cores (adapted from [7]). 90

4.3 Main features of the error syndromes generated by permanent faults on the evaluated instructions (adapted from [7]). 92

4.4 Execution performance of the implemented method for permanent fault evaluation of DNNs (adapted from [7]). 95

4.5 Tested units area and utilization percentage w.r.t. a FP32 functional unit (adapted from [6]). 100

4.6 Percentage of faults that are uncontrollable, masked, cause hangs or instruction-level errors (adapted from [6]). 101

4.7 Codes used for the software-level error injections (adapted from [6]). 111

Nomenclature

Acronyms / Abbreviations

ABFT Algorithm-Based Fault-Tolerance

AI Artificial Intelligence

ARC Admissible Regions for Compaction

ARF Address Register File

ATPG Automated Test Pattern Generator

BB Basic Block

BIST Built-In Self-Test

CAGR Compound Annual Growth Rate

CFR Constant Failure Rate

CTA Cooperative Thread Array

DfT Design for Testability

DMU Divergence Management Unit

DNN Deep Neural Network

DPU Dot Product Unit

DSM Divergence Stack Memory

DU Decoder Unit

DUE Detected Unrecoverable Error

ECC Erroc Correction Code

EDA Electronic Design Automation

F-sim Fault Simulation

FAPR Fault Activation and Propagation Rate

FC Fault Coverage

FPGA Field Programmable Gate Array

GEMM General Matrix Multiplication

GPC Graphics Processing Cluster

GPRF General Purpose Register File

GPU Graphic Processing Unit

HDL Hardware Description Language

HITPT Hardware Injection Through Program Transformation

HLL High-Level Programming Language

HLSTL High-Level STLs

HLTP High-Level Test Programs

HPC High Performance Computing

IAC Incorrect Active CTA

IAL Incorrect Active Lane

IAT Incorrect Active Thread

IAW Incorrect Active Warp

IIO Incorrect Immediate Operand

IL Intermediate Programming Languages

ILTP Intermediate-Level-Test-Program

IMD Incorrect Memory Destination

IMS Incorrect Memory Source

IOC Incorrect Operation Code

IoT Internet of Things

IPP Incorrect Parallel Parameter

IRA Incorrect Register Addressed

ISA Instruction-Set Architecture

IVOC Invalid Operation Code

IVRA Invalid Register Addressed

LD/ST Load Store Unit

LLL Low-Level Programming Languages

MAC Multiply and Add

mISA machine Instruction Set Architecture

MISR Multiple Input Signature Register

MM Matrix Multiplications

ORNL Oak Ridge National Laboratory

PMU Parallel Management Unit

PPB Parallel Processing Block

PR Pipeline Registers

PRF Predicate Register File

PTP Parallel Test Programs

PULP Parallel Ultra Low Power Platform

RTL	Register Transfer Level
SASS	Shader ASSembly
SBST	Software-Based Self-Test
SDC	Silent Data Corruption
SDE	Silent Data Error
SFU	Special Function Unit
SIMD	Single Instruction Multiple Data
SIMT	Single Instruction Multiple Thread
SM	Streaming Multiprocessor
SoC	System on Chip
SP	Streaming Processor
SpT	Signature per Thread
STL	Software Test Library
TCU	Tensor Core Unit
TP	Test Program
TPats	Test Patterns
TPC	Texture Processing Cluster
vISA	virtual Instruction Set Architecture
WSC	Warp Scheduler
WSm	Warp Scheduler Memory
WV	Work-flow Violation

Chapter 1

Introduction

The modern landscape of applications powered by Artificial Intelligence (AI) is rapidly spreading into multiple domains, such as automotive, healthcare, and robotics, among others. Indeed, the remarkable innovations in AI algorithms and applications started to accelerate the transition of the modern world to the *automation era*. Indeed, the capabilities of modern generative AI models can be applied for generating automated applications such as virtual assistants for customer services [8, 9], multimedia and video production assistance [10, 11], software development automation [12, 13], and applications to robotics and autonomous systems [14, 15].

Recently, the rise of AI has led to a higher demand for advanced electronic devices that can handle the computational requirements of today's systems, such as cloud computing, autonomous systems, and Internet of Things (IoT). As a result, the global semiconductor market is predicted to grow significantly in the upcoming years, with an expected value of one trillion dollars by 2030, as shown in figure 1.1 [1]. It is estimated that approximately 70% of this growth will be driven by three sectors: computation and data storage, automotive electronics, and wireless communications. For example, the usage of supercomputing systems has increased due to the computational burden of cloud computing applications such as AI acceleration, IoT, and banking, among others [16, 17]. Similarly, as the automotive industry moves towards the autonomous age, vehicles need to integrate high-performance system-on-chips (SoCs) to support their onboard functionalities such as infotainment, self-driving assistance, or full autonomous capabilities [18, 19].

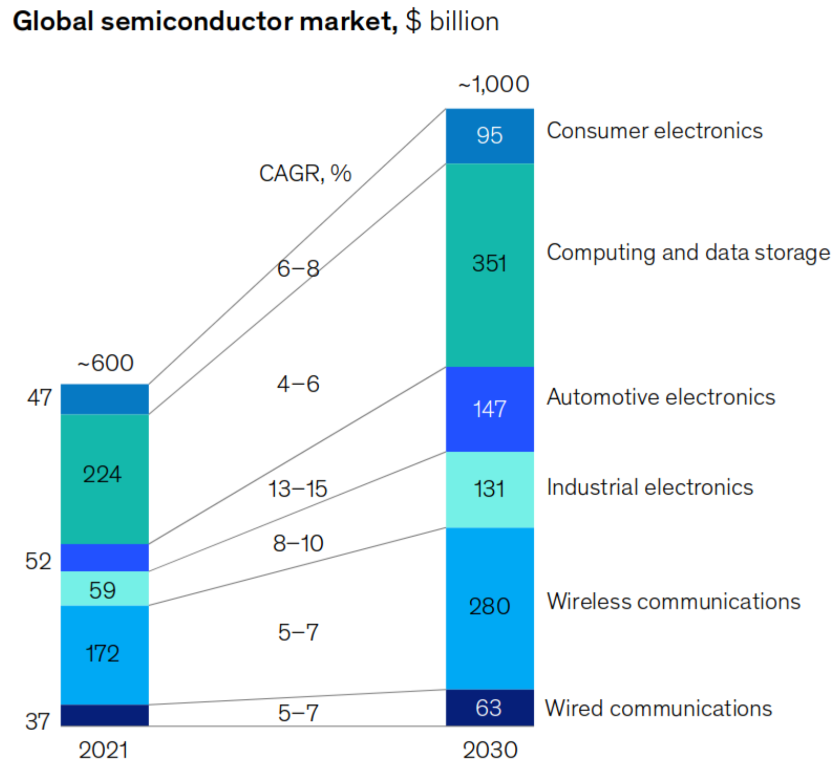


Fig. 1.1 Global Semiconductor Market (source McKinsey & Company [1]).

Undoubtedly, the semiconductor industry has brought innovations in silicon technologies, manufacturing processes, and computer architectures applied to the design of smaller, faster, and more energy-efficient electronic devices. Nonetheless, these technology improvements also introduce challenges regarding the dependability and security of applications that rely on electronic devices. In fact, cutting-edge semiconductor technologies must face crucial aspects such as reliability, security, and safety to guarantee the correct operation of modern applications, especially those considered safety-critical systems, such as autonomous driving systems, robotics, or health care equipment [20].

The advancements in the semiconductor industry have drastically improved the computing performance of modern electronic devices, leading to significant progress in a wide spectrum of application domains such as communication systems, machine learning, and hundreds of other technologies that are shaping our world [20].

Graphic Processing Units (GPUs) have become popular with the rise of AI, and nowadays, they are increasingly adopted in multiple application domains, including

high-performance computing (HPC), autonomous robots, automotive, and aerospace applications. In fact, as presented in figure 1.2, the global GPUs market was valued at 40 billion U.S. dollars in 2022, with forecasts suggesting that by 2032, this is likely to rise to 400 billion U.S. dollars, growing at a compound annual growth rate (CAGR) of 25 percent from 2023 to 2032 [2]. The computational power of GPUs and their flexibility in a wide range of applications has increased their demand in recent years. For example, the Microsoft Azure supercomputing system includes more than 30,000 GPU devices to handle the computational burden required by OpenAI services like ChatGPT [21, 22]. In addition, GPUs have become the "Driving Force" for self-driving vehicles, and they are incorporated in automobiles as specialized SoCs with superior computational capabilities for supporting the processing of huge amounts of data coming from sensors, cameras, and radars [18, 19]. Thus, the use of GPUs in applications that wander off their traditional fields (gaming, multimedia, and consumer market) has suddenly pushed the interest and posed questions about their reliability [23].

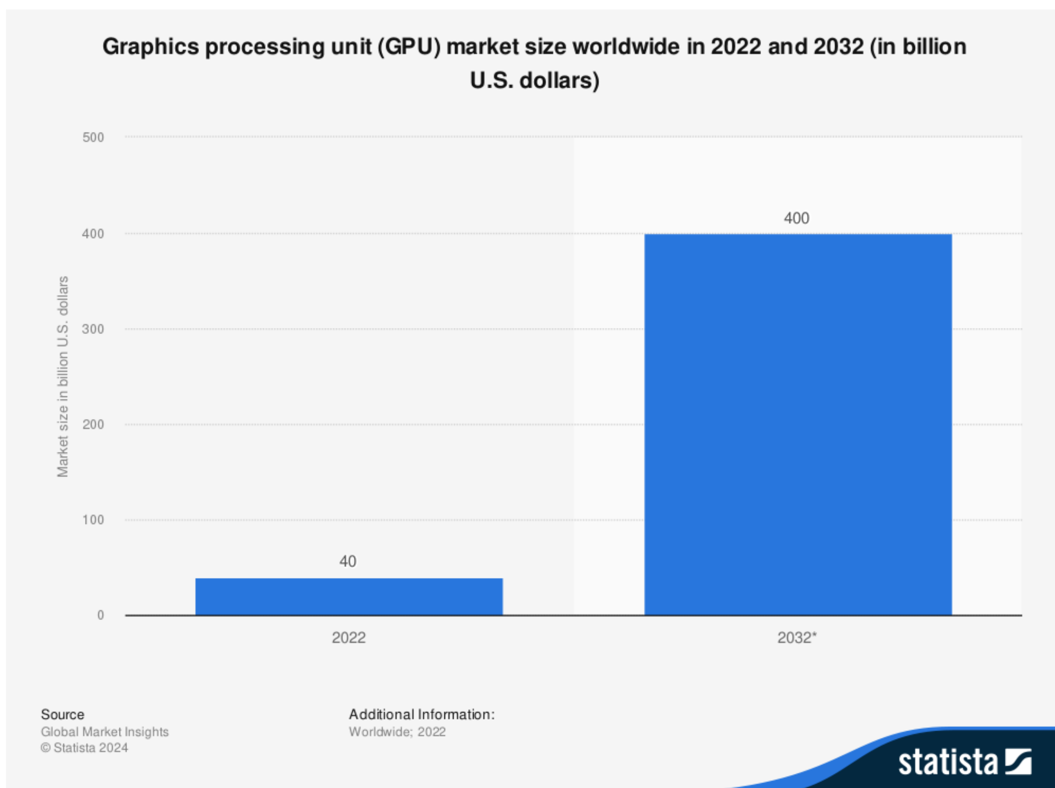


Fig. 1.2 GPU market size worldwide 2022-2032 (source Statista [2]).

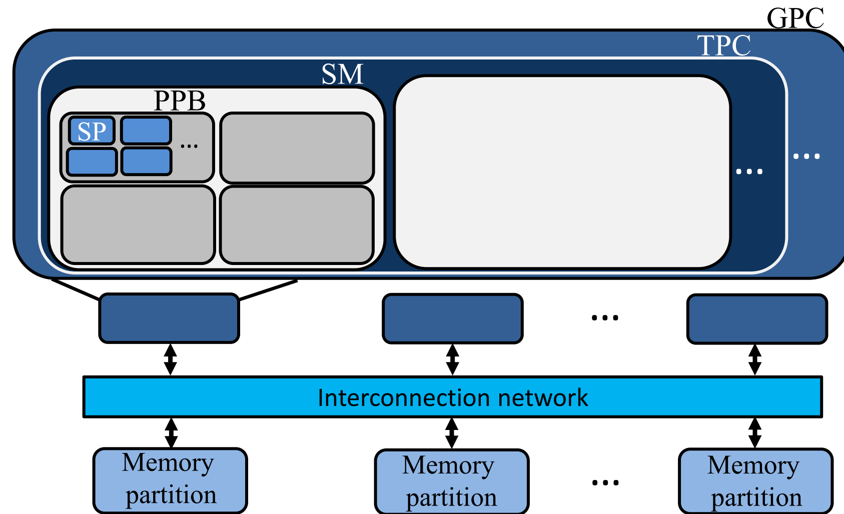


Fig. 1.3 A general scheme of the internal organization of a GPU.

1.1 Architectural organization of GPUs

The reason why GPUs and CPUs have different capabilities is that they were designed with different goals in mind. The CPU's main function is to execute a sequence of operations, also known as a thread, as quickly as possible, and it can execute a few tens of these threads in parallel. On the other hand, the GPU's main function is to excel at executing thousands of threads in parallel, which allows it to achieve greater throughput by amortizing the slower single-thread performance. An application usually contains both parallel and sequential parts, and therefore, systems are designed with a combination of GPUs and CPUs to ensure high overall performance. In scenarios where an application has a significant amount of parallelism, the GPU can be utilized to exploit its massively parallel nature, resulting in better performance compared to the CPU [24].

GPUs are special-purpose processors designed to exploit hardware parallelism and provide high throughput in the execution of applications. Currently, modern GPU designs are mainly composed of a set of homogeneous cores organized in a hierarchical fashion, including *Graphics Processing Clusters* (GPCs), *Texture Processing Clusters* (TPCs), and *Streaming Multiprocessors* (SMs), or *SIMD Engines* [25, 26, 24], as shown in Figure 1.3.

An SM is the main operative core inside modern GPUs, and it implements the Single-Instruction Multiple-Data (SIMD) paradigm or variations, such as Single-

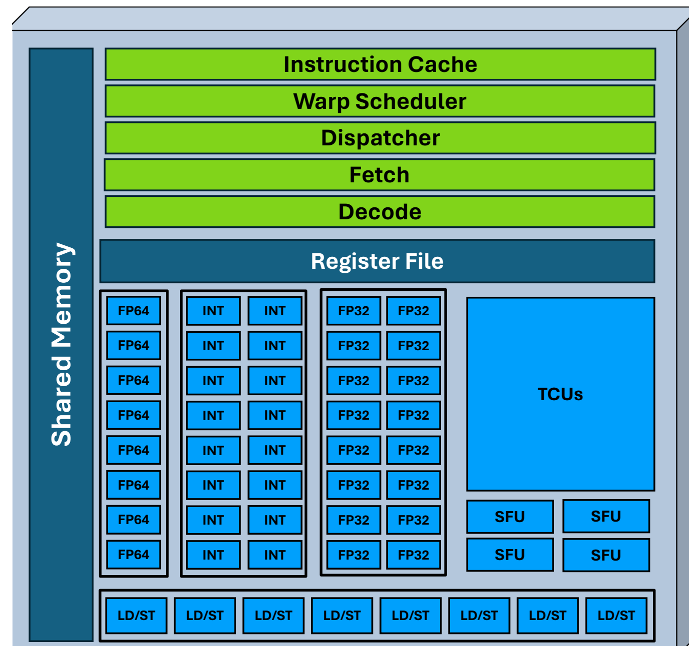


Fig. 1.4 Internal components distribution of a Parallel Processing Block (PPB) of an SM.

Instruction Multiple-Thread (SIMT). The SMs are usually partitioned into 2 to 4 Parallel Processing Blocks (PPBs). Figure 1.4 illustrates the typical organization of a PPB inside an SM. In detail, each PPB handles a set of parallel functional units (known as *Streaming Processors*, or SPs cores), which are used to execute the same operation in parallel for several threads. These SP cores (in the following, we will refer to them as SPs for simplicity) in the NVIDIA terminology are also known as CUDA cores [25, 26, 24].

The number of SPs (from 8 to 32) directly depends on the GPU architecture and the number of parallel threads to be processed simultaneously. The SP-Cores incorporate computational support using different data and formats, such as Floating-Point Units of different widths (FP16/FP32/FP64) and Integer cores (INT32). In addition, every PPB also includes further special computational units, such as the Special Function Units (SFUs) and Tensor Core Units (TCUs), to perform specific operations and support multimedia and artificial intelligence applications [25, 26, 24].

The SFUs are specialized floating point cores that perform fast approximation of transcendental operations, such as $\sin(x)$, $\cos(x)$, $\frac{1}{\sqrt{x}}$, $\log_2(x)$, 2^x, $\frac{1}{x}$, $\sqrt{x}$. SFUs are fundamental cores in the acceleration of scientific and machine learning operations. In fact, the division and module operation in a GPU is supported by the fast approx-

imation of the reciprocal operation $\frac{1}{x}$ implemented by the SFU core. In fact, the acceleration of normalization layers in DNNs is supported by the SFU computation in the GPU [25, 26, 24].

TCUs are specialized hardware arrays of *Dot-Product-Units* (DPUs) to improve performance in the execution of General Matrix Multiplication (GEMM) operations in Machine Learning domains. DPUs comprise *Multiply-and-Add* (MAC) cores that enable GPUs to compute Matrix Multiplication (MM) on a small matrix in one instruction cycle. For example, TCUs can perform a $4 \times 4 \times 4$ MM in the form $D = AB + C$ efficiently [27–29]. Moreover, TCUs are reused to compute large matrix tiles (e.g., $16 \times 16 \times 16$) by accumulating partial results.

The GPU architecture also includes a memory hierarchy mainly used to reduce latency during the kernel execution. The memory resources include a 'General Purpose Register File' (GPRF), a shared memory, a local memory, a constant memory, and an external global/main memory. The Load/Store (LD/ST) units of every PPB guarantee access to the data in the memory hierarchy of the GPU; typically, several of these units allow to have memory access to several execution threads in parallel [25, 26, 24].

Finally, the workload distribution of the SM is ruled by a general scheduler inside each SM that statically distributes the tasks in a group of threads (called *Warps* in NVIDIA terminology and *wavefronts* for AMD GPUs) among the PPB cores. Every PPB includes a Warp Scheduler Controller (WSC), a fetch unit, and an instruction decoder unit to manage parallelism and submit, distribute, and track warps into the available cores [30, 24, 31, 25].

The programming model of a GPU consists of parallel programs, i.e., programs that run on multiple threads. These programs are called *Kernels* when designed or executed on GPUs. A kernel composed of multiple threads is divided into thread blocks that operate separately from one another. Figure 1.5 illustrates the thread hierarchy used by CUDA for NVIDIA GPUs. The threads in the kernel can be arranged into a one-dimensional, two-dimensional, or three-dimensional grid of thread blocks. The number of thread blocks in a grid is usually determined by the size of the data being processed, which generally surpasses the number of SMs in the system. There is a restriction on the number of threads per block because all threads of a block are meant to be located on the same streaming multiprocessor core and, therefore, must share the core's limited memory resources. Presently, GPUs

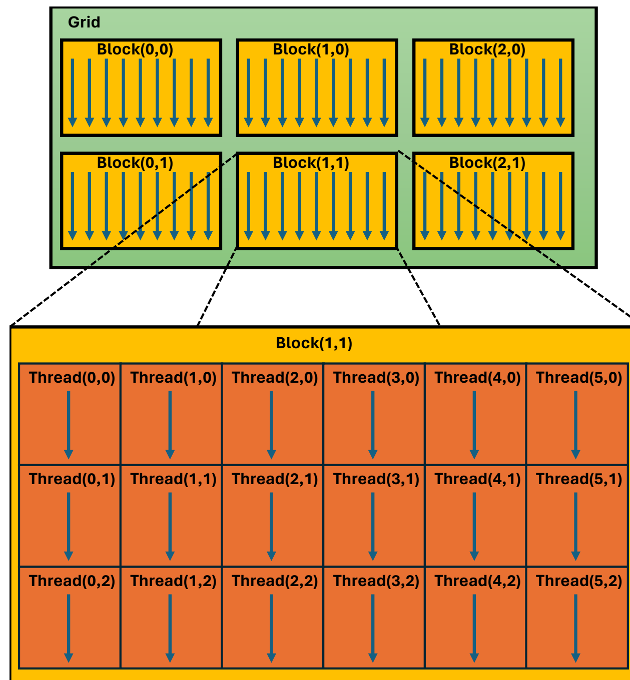


Fig. 1.5 Thread hierarchy in CUDA programming.

allow a thread block to have a maximum of 1,024 threads. Nonetheless, a kernel can be run by multiple thread blocks having the same shape, so that the total number of threads is equivalent to the number of threads per block multiplied by the number of blocks [24].

The thread blocks in kernels, also known as *Cooperative-Thread-Arrays* or (CTAs), are distributed among the SMs using efficient scheduling policies [32, 24]. In general, the scheduler takes all CTAs and issues their execution into waves of CTAs [33]. The size of a wave depends on the number of SMs on a GPU and the kernel's *Theoretical Occupancy* (i.e., number of CTAs concurrently executed per SM). For example, in a hypothetical GPU device composed of four SMs with an assumed occupancy of two CTAs per SM, the wave size corresponds to eight CTAs. Therefore, the full workload execution of a kernel composed of 12 CTAs is two waves; the first one is composed of 8 CTAs distributed among all the SMs in parallel, and the second wave is composed of only four CTAs that will be executed once the resources of an SM are available [24]. This design allows GPUs with more multiprocessors to complete the program faster than GPUs with fewer multiprocessors.

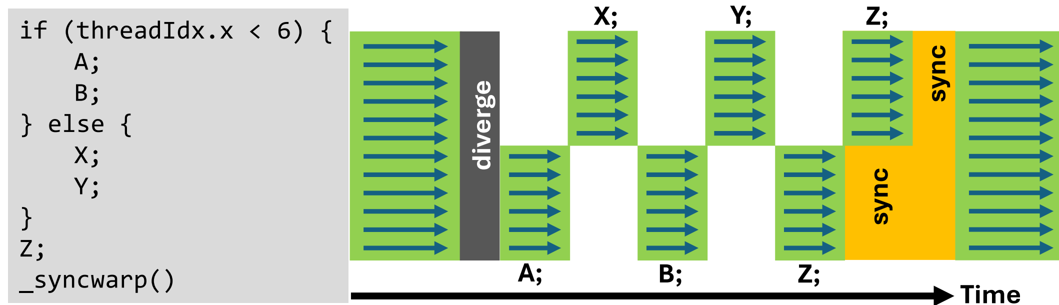


Fig. 1.6 Thread scheduling under the SIMT warp execution model of Volta NVIDIA GPUs.

Finally, each SM executes CTAs following *Single-Instruction Multiple-Threads/Data* (SIMT/SIMD) paradigms by running a group of threads (32 threads or a warp) in parallel [25, 26]. When a CTA is assigned to an SM for execution, it divides them into warps, which are then scheduled by a warp scheduler for execution. The partitioning of a block into warps always follows the same pattern, with each warp consisting of threads having consecutive, increasing thread IDs, and the first warp containing thread 0. The relationship between thread IDs and thread indices in the block is described by the Thread Hierarchy depicted in Figure 1.5.

A warp can achieve full efficiency only when all of its 32 threads follow the same execution path, executing a single common instruction at a time. In case threads within a warp diverge by a data-dependent condition, the warp will execute each branch path taken, while disabling threads that are not on that path. It is important to note that branch divergence occurs only within a warp, while different warps execute independently, regardless of whether they are running common or disjoint code paths. Figure 1.6 illustrates the effect of the thread divergence of the SIMT execution model of a volta GPU architecture. In this example, the "if" statement decides what every thread executes based on their identifiers so that when the thread ID is lower than 6, those threads will execute A, then B, and finally Z. On the other hand, the other threads will execute X, Y, and Z. In the end, all threads need a synchronization or convergence (*sync*) in order to continue the parallel execution of the CTA in the SM [24–26].

1.2 Hardware faults and GPU's reliability

The amount of hardware incorporated in a GPU device in order to support the massive parallelism and computation capabilities also involves the incorporation of high transistor densities on a single chip (e.g., 80B transistors for NVIDIA Hoper GPUs [34, 35] and $>$ 100B transistors for Intel GPUs [36]). Such technological evolution also raises reliability concerns when used in safety-critical applications with long operative conditions. These reliability concerns about GPUs are mainly connected with their lifespan, which does not exceed two years in the consumer electronic market. However, GPUs employed in automotive, aerospace, and military applications are expected to be operational for many years. Additionally, typical operative conditions of HPC-grade GPUs, such as over-stress, high temperature, high frequency of operation, and technology node shrinking, have been shown to accelerate aging [37, 20], which accelerate the rise of faults in the device.

In general, the advancements in semiconductor technologies have enabled the creation of smaller, faster, and more energy-efficient electronic devices, including GPUs. However, the continuous miniaturization of these technologies (e.g., 7nm and below) has also led to reliability concerns as they are more susceptible to faults caused by aging, over-stress, environmental harshness, or potential manufacturing defects. In fact, this technology scaling contributes to make modern devices vulnerable to permanent faults even induced by terrestrial radiation [38]. Several studies have demonstrated that the failure rate grows as technology scales [39, 40, 20], where the constant failure rate (CFR) increases with possible wear-out failures occurring earlier than expected, as depicted in Figure 1.7. This accelerated failure rate of modern silicon technologies, also used to manufacture GPU devices, can seriously reduce the lifespan of devices and affect the reliability of a vast amount of applications nowadays.

The relationship between the increased failure rate and the technology scaling can be attributed to changes in manufacturing methods used to improve the performance of modern devices as scaling slows down. This has resulted in the introduction of new materials, transistor architectures, and a shift towards 3D chip design [20]. Unfortunately, all these changes can produce silicon defects corresponding to permanent faults that can appear during the operative life of a GPU device, ultimately leading to serious problems such as system failures [41–43]. A tangible sign of the occurrence of permanent faults in GPUs is given by field tests/reports from

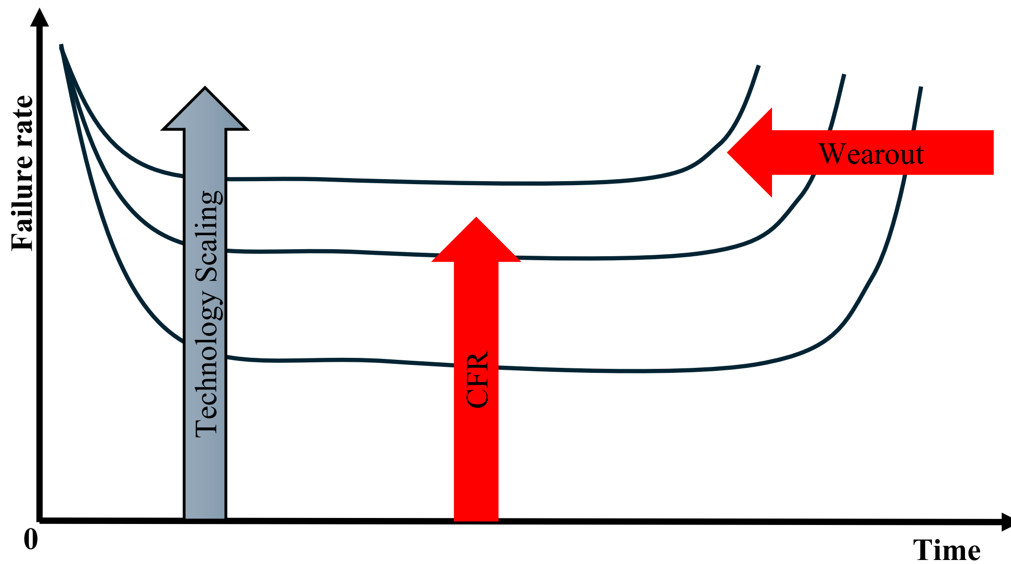


Fig. 1.7 Product failure rate increases as the technology scales. The constant failure rate (CFR) increases with wear-out failures occurring earlier than expected (source NASA [40]).

large-scale server operations from Google [42], Facebook [41], and ORNL [44]. For instance, premature aging on the GPUs of the Titan supercomputing system was found, on average, every 2.8 years, requiring the replacement of 9,500 GPUs [45].

The high sensitivity of GPUs to permanent faults can have an unacceptable impact on critical applications, especially when they arise during the in-field operation. Indeed, Silent Data Errors (SDEs) have become a major reliability concern in cutting-edge semiconductor technologies used in critical applications such as data centers for cloud computing, space, or automotive. In fact, not all permanent defects in a computer system produce a failure of the application, such as a crash or hang, but they may propagate silently, as SDEs, during the application execution and potentially lead to a system failure [41–43, 46]. Hence, functional safety mechanisms must be developed or adopted in order to detect possible faults and prevent catastrophic effects caused by those faults.

1.2.1 Reliability enhancement of GPUs

The development of functional safety mechanisms for GPU devices allows to enhance their reliability against hardware defects. These mechanisms may resort to either

hardware, software, or a combination of both strategies that allow to detect and repair (or counteract) the impact of a fault during the in-field operation of the device, minimizing the effect of the final application, such as preventing catastrophic results.

The hardware-based hardening solutions mandate the introduction of additional hardware structures during the initial design stages of a device. These structures are designed to detect a high number of faults and meet the functional safety standards requirements. Among the most popular functional safety mechanisms in GPUs, we can find the improvement of memory cells design [47], adding error-correcting codes (ECC) [48], hardware redundancy [49, 50] or hardware structures for fault detection [51]. These solutions can detect and possibly correct a very high percentage of faults [52]. However, using such specialized hardware solutions for safety-critical applications may result in a significant increase in cost compared to "normal" devices. This solution may also have an impact on performance, power consumption, and ease of use. Alternatively, adopting in-field testing solutions may allow the detection of faults before they produce critical failures, thus lowering the failure probability to acceptable values. In this regard, Sofware-Based Self-Testing (SBST) provides a suitable solution, since it does not require adding extra hardware to the device, allowing the application of functional testing during the operation of the device

On the other hand, several studies have proposed software-based techniques to ensure the correct operation of systems in the presence of faults. These include software checksums [53], multi-threading redundancy [54], diverse redundancy [55, 56], or Algorithm-Based Fault-Tolerance (ABFT) solutions [57, 58]. While these methods have been effective in mitigating transient faults, it is unknown whether they can effectively prevent permanent faults that can lead to Silent Data Errors during a GPU's lifespan. In fact, modern Graphics Processing Units demand life expectancy extended to many years, exposing the hardware to aging (i.e., permanent faults arising after the end-of-manufacturing test). Hence, techniques to assess permanent faults that can produce SDEs during the device's operative life are strongly required, especially in safety-critical domains. These fault evaluations are crucial for two main reasons: first, the fault evaluation allows the identification of vulnerabilities in GPU's application regarding permanent faults. Consequently, fault evaluations contribute to the development of effective software-based hardening strategies to counteract the impact of such faults during the operative life of the device. On the other hand, fault evaluations allow the assessment of the effectiveness of any fault countermeasure against SDEs caused by permanent faults on GPUs, allowing the

enhancement of the reliability of the whole system using GPU acceleration, (e.g., improving the DNNs resilience in automotive applications).

1.3 In-field testing of GPUs

In-field testing consists of strategies that can identify faults arising during the operational phase before they lead to critical failures, ultimately reducing the probability of failure to acceptable levels.

The in-field testing techniques for fault detection in GPUs can be classified into three main categories. The first category corresponds to Design for Testability (DfT) methods, which focus on including special hardware structures during the design stage. The second category corresponds to hybrid approaches, which involve a combination of hardware structures embedded in the device with reconfigurable capabilities and software routines that can increase fault detection capabilities. The third category corresponds to software-based self-test (SBST) solutions.

DfT methods, such as Logic and Memory *Built-In Self-Test* (BIST), are commonly used to test devices such as GPUs at the end of production [59, 60]. Therefore, their usage for in-field testing can be limited to the "power-on/off mode" of the devices or systems when timing constraints are more relaxed, and the testing can be applied to every hardware structure without affecting the application functionality. Nonetheless, the usage of DfT methods for fault detection while the device (i.e., the GPU) is already in operation can be challenging, since the application may not have time slots to apply the test (i.e., "Mission Continuous mode"), or the available time slots during the application execution are limited or constraint (i.e., "Mission Periodic mode"). Moreover, the protocol to activate the DfT mechanisms and retrieve the test results should be passed by the manufacturing company to the system company. Typically, manufacturing companies are reluctant to do so for confidentiality reasons. In other situations, the DfT requires the introduction of additional hardware structures that may result in useless hardware when the same device is not used in safety-critical applications [61]. Therefore, alternative solutions need to be explored to cover as much of the in-field operation modes of the device as possible.

The hybrid solutions (hardware/software) that involve adding or using available structures (such as performance counters) to increase the fault observability of a

module must be incorporated into the design phases. This is done by modifying the hardware-software interface to provide instruction-based control of the included structures. Jagannadha et al. [62] have proposed an in-system test architecture that combines DfT methods and hybrid structures to detect faults and provide diagnosis features during the in-field operation of system-on-chips (SoCs) and GPUs. Also, Guerrero-Balaguera et al. [63] proposed to take advantage of the GPU's performance counters in combination with software instructions that allow the detection of faults in the Warp Scheduler and divergence stack memory of a GPU. Moreover, in some cases, the cost of implementing DfT solutions may be too high, or the application safety requirements may demand more frequent in-field testing that could potentially use the idle time slots left by the application.

On the other hand, the software-based self-testing strategy is a flexible and noninvasive approach to performing functional in-field tests of processor-based systems [60, 61, 64], and it has been successfully applied to GPU testing as well [65].

1.3.1 Software-Based Self-Testing

The SBST strategy offers in-field at-speed fault detection capabilities without hardware costs, leveraging the idle times of the application to execute test procedures [64]. The SBST strategy consists of developing Test Programs (TPs) by the device developer/manufacturer that can achieve a given Fault Coverage (FC) with respect to selected structural fault models, determined through Fault Simulation (F-sim) [66]. A group of TPs composes a Self-Test Library (STL) [66], which is then integrated by the system company into the application code and activated with the required frequency. This strategy is widely used in the industry, and several semiconductor manufacturers currently develop and offer STLs for their processor-based products [67–70] to support their usage in safety-critical applications. Furthermore, several works successfully demonstrated the feasibility of STL development exploiting the inherent parallelism of GPUs for testing purposes targeting functional units [71, 72], memory modules [73–75], and control units [73, 76].

1.3.1.1 Related works

In parallel architectures, such as GPUs, the SBST strategy can also be adopted to develop Parallel Test Programs (PTPs). Each PTP is built employing the available

Instruction-Set Architecture (ISA) of a target GPU. Each instruction in the PTPs is intended to apply one or more test patterns to one or several target modules in parallel. These instructions compose routines aiming at exciting, propagating, and detecting faults when operating warps in an SM.

In general, the main structure of a PTP for GPU testing comprises three main parts: *i)* thread registers load, *ii)* parallel operation execution, and *iii)* propagation of the result to an observable point.

In principle, these steps are repeated for each thread in the program. However, it is also possible that divergences could be present, so only a portion of the threads execute a given operation. Meanwhile, missing threads can be skipped or performed using different procedures. This divergence behavior is commonly used to excite control modules but may affect the test quality on functional units and regular structures in the GPU. In these parallel architectures, the fault detection of a PTP is commonly performed using exceptions and thread signatures [77] out of the values on any observation point or memory output of the GPU. A comprehensive overview of the main issues (and possible solutions) to be faced when generating STLs in an industrial environment can be found in [78].

Concerning the development of STLs for GPUs, so far, these software-based testing approaches have been developed at the machine level (e.g., using assembly languages also known as, *Low-Level Programming Languages*, or LLLs) [66]. This approach requires significant engineering effort, development time, and deep architecture knowledge of the target device, and it is characterized by limited re-usability. In fact, the ISA architecture among different GPU devices significantly constrains the portability of STLs developed in LLLs among devices of the same vendor.

In contrast, High-Level Programming Languages (HLLs) for GPUs, such as CUDA or OpenCL, simplify programming and are often the best, and sometimes the only, way to develop and encode applications. HLLs also offer programming scalability and flexibility, which can encourage test specialists to use them for the development of STLs for GPU devices [79]. However, there are still several challenges and open questions when it comes to using HLLs for the development of STLs [79].

1.3.1.2 Main contributions

This Ph.D. thesis proposes a methodology for developing STLs for GPUs using high-level languages [80, 3]. The aim of this method is to reduce the inherent complexity in the development flow of STLs that use assembly languages. In addition, this thesis presents the advantages and limitations of developing TPs using different software abstraction levels: HLLs, LLLs, or a combination of both. The method is validated using a microarchitectural open-source GPU model (FlexGripPlus [81]) and two GPU devices (NVIDIA Jetson Nano and Nvidia GeForce GTX 960M). The experimental results show that regular units, such as functional units and register files, in a GPU can be tested using HLL TPs. However, the compiler optimizations, observability constraints, and architectural features in other modules, such as hidden units, require the combination of different abstraction levels (resorting to both LLLs and HLLs) or the explicit development of TPs at the assembly level only. Although this work uses NVIDIA terminology and refers to the NVIDIA GPU architecture, most of the ideas can be generalized to other GPUs.

1.3.2 Compaction of STLs for GPU testing

Although the SBST approach allows for the successful development of test programs and STLs for testing modern devices like GPUs, it may also occupy a significant amount of memory (e.g., program and data) and spend non-negligible execution testing time. In addition, the STLs and test programs can be developed by using high-level programming languages (HLLs) that increase the programmer's productivity and development time, but they might also impact the memory footprint and execution time obtained after the compilation stages [80, 3]. In addition, the in-field operational constraints can impose restrictions related to the maximum size and duration that a given STL can afford when used concurrently with a given operation.

In such situations, the test compaction applied to STLs and test programs is a paramount aspect in guaranteeing the in-field testing constraints. However, the algorithm complexity and the programming abstraction levels (e.g., using HLLs for STL development) make the compression of an STL a challenging task. Although compaction method approaches using evolutionary techniques or instruction reordering strategies, have been proposed, most of them require a high computational effort to analyze and compact a few test programs in STLs.

1.3.2.1 Related works

Several works have proposed methods to compact Test Programs (TPs) for processor-based systems. These methods effectively reduce the size and duration of a given TP while maintaining the same fault coverage [82–84]. In [85], the authors split TPs into subroutines and removed individual instructions after analyzing the fault coverage contribution of each sub-routine. Authors in [86] exploited reordering techniques among different pieces of a TP to maintain the fault coverage and reduce the length of the TP. In both cases, a high computational effort is required to analyze and compact a given TP. In fact, the compaction process is based on the production of compacted TP candidates from the original TP, which are then fault simulated to assess the new fault coverage. However, the required time and computational costs for the compaction of an individual TP are exceptionally high. It is worth noticing that none of the reported techniques in the literature face the compaction of PTPs and STLs for GPUs, and some of them can hardly be extended from CPUs to GPUs.

1.3.2.2 Main contributions

This Ph.D. thesis proposes a method to automatically compact the test programs of a given STL [4, 5]. The method was first developed to reduce the size, and the test duration of STLS developed for CPUs [4]. Then, the method was effectively extended to the compaction of STLs for GPUs [5]. The proposed method combines a multi-level abstraction analysis resorting to logic simulation to extract the microarchitectural operations triggered by the test program and the information about the thread-level activity of each instruction and to fault simulation to know its ability to propagate faults to an observable point. The main advantage of the proposed method is that it requires a single fault simulation to perform the compaction. The effectiveness of the proposed approach was evaluated, resorting to several test programs developed for an open-source GPU model (FlexGripPlus) compatible with NVIDIA GPUs. The results show that the method can compact test programs by up to 98.64% in code size and by up to 98.42% in terms of duration, with minimum effects on the achieved fault coverage.

1.4 Reliability evaluation of GPUs with respect to permanent faults

The reliability evaluation of GPUs concerning *transient* faults has already been evaluated through microarchitectural and low-level fault simulation [87, 81, 88], software-based fault injection [89–93], and beam experiments [94–98]. Since GPUs execute several processes in parallel, it has also been shown that a transient corruption in the warp scheduler or a single error in shared resources of the GPU affects various output elements [95, 53, 97, 99, 98]. Nonetheless, the evaluation of permanent faults that potentially produce SDEs in GPUs is still an open research topic, especially given the reliability concerns exposed by [42, 41, 43], showing that cutting-edge semiconductor technologies accelerate aging and other phenomena that end up into permanent faults causing catastrophic results at the application level. Moreover, the impact of a permanent fault is not always constant and can vary depending on the affected hardware units, the functionality of the faulty unit within the device, and the workload being executed on the system. Consequently, the workloads can produce input patterns that activate permanent faults in a GPU's hardware units, resulting in miscellaneous SDE effects that can spread throughout the application execution. These errors may ultimately affect the final result and lead to undesirable outcomes. Therefore, it is crucial to adopt effective fault evaluations that can accurately assess the impact of these faults on different types of parallel workloads.

Among all the methodologies used for assessing the resilience of applications regarding permanent faults affecting different devices, the *Fault Simulation* approach using RT- or gate-level description is the one that provides more realistic evaluations. In fact, fault-simulation approaches provide accurate evaluation results, since they simulate different fault effects using the target device's hardware at functional [100, 101], RT/gate description [102, 81], or functional levels. Unfortunately, these fault evaluation approaches result in unfeasible solutions to carry out the evaluation of permanent faults in GPUs, given the hardware and software complexity of GPUs and their applications. For example, considering an RT-level GPU model, one fault simulation campaign requires about eight days to evaluate a 32X32 matrix multiplication (MM). Thus, the complete evaluation of a DNN, such as LeNet5, which incorporates hundreds of MM operations plus further max-pooling

and activation operations, would require unacceptable simulation times ($>$ 10,000 days!) [7, 6].

Other fault evaluation approaches resort to *Emulation-Based Fault Injections* that reduce the evaluation time using FPGAs to implement the target circuits, including corruption mechanisms either inserting saboteurs inside the device's RTL model or by corrupting the FPGA bit-stream [103]. Although the evaluations at this level provide accurate evaluations close to reality, this fault evaluation technique requires synthesizable GPU models in Hardware Description Languages (HDLs), costly FPGA devices or clusters of FPGAs, and non-negligible development time.

Recently, other fault evaluation approaches have adopted *Hardware Injection Through Program Transformation (HITPT)*, mainly to study the impact of transient faults on GPU workloads [104]. The HITPT approach is a software-based error injection that mimics the behavior of hardware faults by inserting saboteur routines on the original application at the assembly level, such that the faults are activated during the program execution [104]. This approach offers faster evaluation times compared to simulation-based methods, and its speed is par with or slightly slower than emulation-based strategies. On the other hand, there are limited studies adopting HITP evaluation of permanent faults on GPU's workloads. Guerrero-Balaguera et al. [105–107] have proposed to use of HITPT approaches to assess the resilience of DNN applications concerning the impact of permanent faults on GPU devices. Their works are limited to model and injecting stuck-at-fault models only on the visible hardware structures at the software level (e.g., register files, arithmetic cores, and memory resources). Unfortunately, they do not consider other units, such as parallel control management or more sophisticated error models that can describe in a more realistic way the effects of faults in the circuit (i.e., gate-level description) and the corruption of the executed instructions for a particular GPU workload. In this regard, it is crucial to adopt a more realistic fault evaluation approach that combines the accuracy of the fault simulations with the speed of the instruction-level propagation of errors. Consequently, modeling silent data errors (SDEs) at the software level (i.e., corrupting the instructions of the GPU) that describe the effects of permanent faults is a promising solution that allows for fast and realistic evaluations of a wide range of workloads.

1.4.1 Related works

In the literature, multiple works propose methods to identify and evaluate the impact of hardware faults on different hardware and applications. Some authors use error propagation models [108, 109] (by changing the instruction's parameters) to represent faults at higher levels and speed up the evaluation in complex systems, such as DNN applications [110] and cryptosystems [111].

On the other hand, most of the methods in the literature for GPUs address the error propagation from instructions to the software- and system-levels ([112], [104], [93]), which are commonly performed on special frameworks instrumenting the application's code with instructions to model errors. However, in most cases, there is not a clear link between hardware faults and the modeled errors, so it is also possible to use unrealistic effects or miss the evaluation of some real effects of the lack of correlation.

Other works emphasized the significance of fine-grained, low-level, and cross-layer resilience evaluations[113]. They also point out the limitations of relying solely on software error propagation methods. In fact, these multi-level evaluation methods strike the best balance between evaluation accuracy, which means more realistic fault evaluations, and the time required to perform such evaluations. There is significant research available regarding fault evaluation in CPUs using multi-level or cross-layer approaches combining low-level fault simulation of individual components in the device and then using high-level error propagation at the system of application levels [114, 115, 89, 116]. Similarly, several works have exploited the multi-level fault evaluation approaches to provide fine-grain and accurate characterization of transient faults for GPU structures and propagate errors at instruction [117], and application levels (e.g., DNN workloads) [118]. However, these characterizations are limited to a few units (functional units), or restricted to a limited number of instructions or applications. Also, a complete characterization of the impact of permanent faults in some critical units of GPUs is still missing, and there are no available frameworks that allow a straightforward evaluation of permanent faults in these complex contexts.

1.4.2 Main contributions

This thesis proposes a multi-level methodology that combines the accuracy of gate-level fault simulation with the speed and flexibility of software-level error injection to evaluate the effects of permanent hardware faults affecting a GPU [7, 6]. The proposed method comprises three main phases. First, a selection of GPU workloads is profiled to extract the input patterns generated at the inputs of the target GPU unit to be evaluated. Then, the results of a gate-level fault simulation are used to characterize the effects of permanent faults at the outputs of the targeted core. These fault effects are transformed into instruction-level errors that are later used for instruction-level error propagation on a real GPU device using state-of-the-art binary instrumentation tools such as NVBIT [119]. This dissertation presents two variations of this multi-level fault evaluation methodology. The first version consists of evaluating the impact of permanent faults on integer and floating point GPU's cores while executing Deep Neural Network (DNN) workloads. The method allows for the first time to estimate the percentage of permanent faults leading the DNNs to produce wrong results. The second methodology variation evaluates permanent faults in the GPU's Parallel Management Units (PMUs) (i.e., warp scheduler, instruction fetch, and instruction decoder units) while executing different parallel workloads. It is worth mentioning that the proposed method reduces the computational effort for fault evaluation by several orders of magnitude compared with simulation-based fault evaluation approaches.

1.5 Thesis contributions summary

The specific contributions of the research work summarized in this PhD thesis are the following:

- A method for describing STLs using high-level programming languages (i.e., CUDA C++ or CUDA PTX) to perform functional tests of GPU devices;
- A novel strategy for compaction of STLs using an instruction removal-insertion approach by combining only *ONE* RTL and gate-level simulations;
- A multilevel method for modeling and evaluating instruction-level errors caused by permanent faults on the functional units of GPU devices. This

method uses gate-level fault simulations to generate syndrome tables used at the instruction level of the GPU. This evaluation approach effectively allows the realistic evaluation of complex workloads like DNN, regarding permanent faults;

- A multilevel approach for modeling and evaluating instruction-level errors caused by permanent faults on the Parallel Management Units (PMUs) of GPU devices (i.e., instruction fetch, decoder, and warp schedulers). This approach uses gate-level fault simulation results to generate errors at the instruction level, corrupting thread indexing registers, predicate registers, or instruction replacements, among others;

- A new fault injection framework (called *"NVBitPERfi[1]"*) that allows the modeling of errors at the instruction level produced by permanent faults on several GPU units. NVBitPERfi is an instrumentation tool that inserts and propagates the errors at the assembly level, allowing the assessment of permanent error effects in real GPU' workloads;

- A set of results reporting for the first time the effects of permanent faults in a GPU executing a DNN.

1.6 Thesis organization

The remainder of this thesis is structured as follows:

Chapter 2 describes the methodology and the main findings about the development of STLs for GPUs and the adoption of high-level programming languages

Chapter 3 describes the proposed method and the main finding for the compaction of STLs first on processor-based systems and then on GPU devices

Chapter 4 describes the proposed methodology for modeling Silent Data Errors (SDEs) at the instruction level on GPUs with respect to permanent faults on key cores of the device. This chapter also reports the main findings when evaluating such SDEs on different parallel GPU workloads

[1] https://github.com/divadnauj-GB/nvbitPERfi

Chapter 5 provides the conclusion of the dissertation and highlights some possible future works on the topic.

Chapter 2

STLs for GPUs using high-level programming languages

This chapter introduces a comprehensive methodology for generating STLs (Software Test Libraries) for in-field GPU testing, resorting to high-level programming languages (HLLs). This methodology aims to simplify the development of STLs for GPUs by using flexible programming languages like CUDA C++ or OpenCL. Additionally, the chapter explores the advantages and disadvantages of using Intermediate programming languages (ILs) (e.g., CUDA PTX (Parallel Thread Execution) or AMD IL (Intermediate Language)) to complement the development of HLSTLs (High-Level STLs). Finally, the chapter describes and analyzes the main advantages and constraints connected to the development of test programs by adopting high-level, low-level, and intermediate-level programming languages or a combination of them when required. The main contributions can be summarized as follows:

- The description of a method to develop suitable STLs targeting the detection of permanent faults in selected GPU units resorting to high-level and intermediate programming languages.
- The identification of challenges and constraints when developing test programs and STLs using high-level programming languages for GPU testing.
- The coding guidelines to tame the main limitations when adopting HLLs and ILs to develop TPs and STLs for GPUs.

The methodology described in this chapter was developed and validated using NVIDIA's concepts and tools. Nevertheless, the techniques can be modified and adapted to fit other GPU architectures. It is worth clarifying that part of the work described in this chapter about the generation of STLs using High-Level languages has been previously published by the author of this thesis in [80] and [3]

2.1 Background

The programming flexibility of Graphic Processing Units (GPUs) and their ability to deliver high performance has made them popular in a wide range of applications (e.g., self-driving cars, aerospace, computing & data storage) where reliability is a crucial aspect to guarantee the minimum operational functionality of the system during its operational life, preventing catastrophic results. In fact, reliability and safety are crucial factors in safety-critical domains, particularly when dealing with cutting-edge technology-scaling nodes that might be affected by permanent faults due to premature aging or wear-out during the operational phase [120]. Therefore, it is essential to incorporate effective testing mechanisms that are able to detect faults arising during in-field operations of GPUs. These in-field testing mechanisms are crucial to detect any device malfunction and promptly take action, thus avoiding any catastrophic result during the operational life of the system.

Software-Based Self-Test (SBST) is a flexible and non-invasive testing strategy that offers in-field testing capabilities while demanding zero hardware costs [78]. This testing strategy is based on the development of specialized routines (*Test Programs*, or TPs) specially designed to excite the internal faults and propagate their effects to visible locations for detection purposes. A collection of TPs creates a Self-Test Library (STL) [78]. Currently, several Intellectual Property (IP) core vendors and device developers/manufacturers offer STL solutions for their processor-based products and support their usage in safety-critical applications [66]. These STLs are integrated by the system company in the application code and activated with the required frequency. On the other hand, several works already demonstrated that STLs can also be effectively developed for GPUs, targeting functional units, memory modules, and scheduler controllers [65, 72, 121].

So far, the STLs development has been dependent on assembly languages, or i.e., on the machine Instruction Set Architecture (mISA), considered as low-level

language (LLL)[66]. For GPU devices, developing STLs using the device's assembly languages or their mISA (e.g., Shader ASSembly (SASS) for NVIDIA GPUs) can be a challenging task, requiring extensive engineering effort, development time, and in-depth knowledge of the device's architecture. Additionally, STLs developed using pure assembly-level coding have limited reusability.

Developing STLs using Low-level languages is a complex task for GPUs because of the challenges involved in dealing with implicit parallelism at a fine-grain level. As a result, it is crucial to carefully design, implement, and validate test programs for GPUs, given the limited information available about the ISA. This process can take a long time, even with the help of specialized tools. Additionally, the differences in structure and ISA among GPU products make it difficult to ensure the portability of Low-Level STLs. Conversely, high-level languages (HLLs), such as *CUDA C++* or *OpenCL*, simplify the programming process and are often the best or only option for developing and encoding applications for GPUs. In addition, the GPU's manufacturers also provide programming support using intermediate languages (ILLs) named *virtual Instruction Set Architecture* (vISA), providing most of the control of assembly languages but is highly portable among different devices of the same GPU family. Some examples of ILLs are CUDA Parallel Thread Execution (PTX) from NVIDIA, or the Intermediate Languages (IL) of advanced micro devices (AMD).

In fact, adopting high-level languages instead of low-level languages has been shown to increase a programmer's productivity by three to ten times, depending on the target platform [122, 123]. Furthermore, HLLs can handle the increasing complexity of modern GPUs, making the maintenance process simpler [79]. The scalability and flexibility of HLLs also make them appealing to test specialists for the development of STLs. However, there are still numerous challenges and unanswered questions surrounding the use of either HLLs or ILLs for the development of high-level test programs and high-level self-test libraries.

2.1.1 Related works

In the past, some authors developed STLs for GPUs using high-level programming languages such as CUDA C++, and OpenCL.

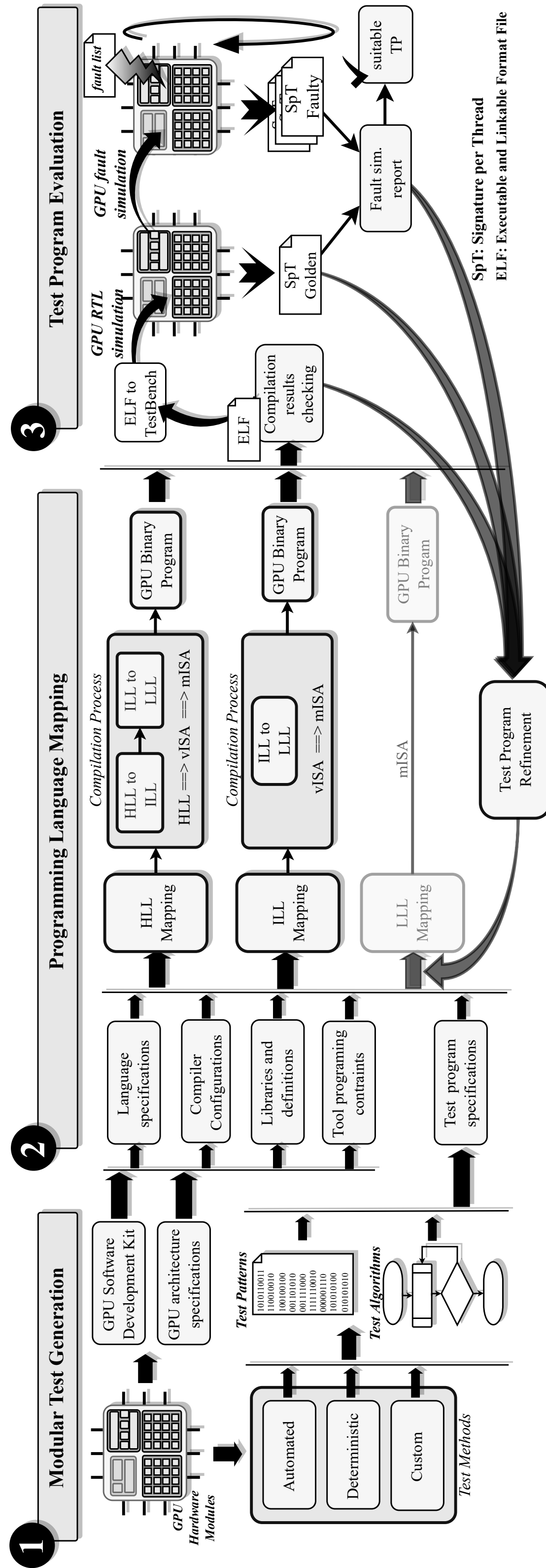


Fig. 2.1 A general scheme of the proposed methodology to generate STLs for GPUs using high-level programming languages (adapted from [3]).

In [72], the authors created a method to detect permanent faults in functional units and the register file of an NVIDIA Fermi GPU by combining CUDA C++ with the virtual assembly vISA (CUDA PTX). To validate this approach, profiling information was used to determine the effects of the proposed Self-Test Libraries on the GPU. However, due to fault observability constraints, it was not possible to quantitatively measure the fault coverage metrics for the proposed STLs.

In [121], researchers utilized various microbenchmarks, which they described in OpenCL, to recognize intermittent faults that were caused by temperature changes or stress in the device. However, the proposed solution was limited to coarse-grain modules (i.e., streaming multiprocessors or SMs) due to the lack of fault coverage metrics and the assumptions made for fault detection. Unfortunately, these works failed to take into account other significant modules inside the GPU, such as schedulers, embedded memories, pipeline registers, and other controllers.

2.2 A Method to develop parallel STLs using high-level languages

The proposed methodology here utilizes the divide-and-conquer technique to break down the structure of a GPU into various modules. Then, individual STLs are created using high-level languages and intermediate-level languages. The flow of the proposed method is illustrated in Figure 2.1, and it comprises three primary steps: *1)* Modular test generation, *2)* Programming language mapping, and *3)* Test program evaluation.

2.2.1 Modular test generation

This phase assesses the following attributes: *i)* the functional features of the unit (such as control, arithmetic, or storage), *ii)* the architecture specifications within the GPU (including data path or control path units), and *iii)* the constraints related to controllability and observability, which means how the program instructions can activate the unit and propagate their results to an observable point. These characteristics are essential for determining the appropriate testing methods, such as automatic, deterministic, or custom, as well as their practicality for implementation

as high-level test programs or intermediate-level test programs. To facilitate testing, each module can be categorized as one of the following three types: *1)* Regular, *2)* GPU's distinctive, and *3)* Hidden structures.

In the first case, the regular modules correspond to replicated units inside the GPU, which perform the same functionality in parallel, such as functional units and register files [74, 71, 3]. These structures, which are typically part of the data path of the device, are also considered visible resources of the GPU. Because of this, any test data can be applied at the unit's input, resulting in high controllability, and the results can be observed after issuing an operation, resulting in high observability. There exist *Automated* and *Deterministic* methods for developing STLs to test these regular units, as described in [78].

The GPU's *distinctive* modules correspond to particular hardware structures specific to GPUs, such as schedulers, divergence controllers, and decoders, which are located both in the data path and control path of the GPUs. It is worth noting that the functional features and complex organization of these particular GPU modules impose testing restrictions such as low controllability or observability. For example, creating suitable input test patterns at the inputs of the scheduler capable of activating and propagating a fault requires complex combinations of instructions and workload configurations that are not trivial to find even by using an Automated Test Pattern Generator (ATPG). The functional constraints of these units demand additional efforts to generate and apply the test data as well as to guarantee the propagation of the fault effects up to an observable point. Consequently, combinations of automated, deterministic, and custom methods are suitable testing solutions for these units [65, 73, 76]

Finally, the third unit type corresponds to hardware structures inside the GPU denoted as *hidden modules*. This category includes hardware structures that are not visible to the programmer, such as embedded controllers and pipeline registers. Due to their inherent complexity, the testing algorithms and methods need to be customized, and elaborated strategies have to be developed [65, 77].

2.2.2 Programming language mapping

When utilizing high-level constructs, mapping the specifications of a given test program into high-level programming languages may be fast and simple. These

constructs, such as nested "for" loops or "if" statements, can effectively describe most of the complex specifications with a reduced engineering effort. However, effectively mapping the testing algorithms for a given GPU hardware structure into HLLs or ILs can lead to challenging tasks. This is because the compilation stages apply optimizations that prioritize the maximum execution performance and security of the code on the target device. Such compiler optimizations can be contrary to the goals of testing, as they limit the control of the programmer. Nonetheless, adequate coding styles and compilation settings can be adopted to avoid or diminish these compiler constraints for testing purposes.

The method proposed in this chapter involves a bottom-up approach that assesses and interprets the test program specifications as one or a series of test routines described in a chosen programming language HLL (e.g., CUDA C++), ILL (e.g., CUDA PTX), or LLL (e.g., Shader ASSembly - SASS). This generation process can require multiple iterations to ensure and verify the testing features of the generated high-level test programs (HLTPs).

In detail, the specifications of a given test program are meticulously transformed into an HLL or ILL while ensuring the following three main considerations: i) Operand allocation, ii) Test algorithm implementation, and iii) Update of the signature per thread.

First, the location of operands is defined according to the test program specifications and can be allocated in the main, shared, or constant memories. It is important to note that some operands can be defined as literals included in the code for HLL mapping, or immediate operands in the case of ILL mapping.

In the second part, the test algorithm is transformed into a sequence of statements according to the selected programming language. For example, "if" statements can be used to induce divergences when using CUDA C++, allowing the testing of the divergence stack memory or the warp scheduler. Similar behaviors can be described using ILLs by implementing a combination of control-flow PTX instructions (i.e., comparison instructions, branch instructions, and predicated instructions). Further details about different mapping strategies of testing algorithms for the most representative GPU cores are described in section 2.3 of this chapter.

Finally, the algorithm updates a signature-per-thread (SpT) after every iteration of the algorithm by implementing a simple counter or a more sophisticated routine,

such as a software-based Multiple Input Signature Register (MISR), as described in [77, 71].

2.2.3 Test program evaluation

This step validates the functionality of a test program or a complete STL (as a binary executable). This validation is divided into three stages: *i)* Compilation results checking, *ii)* Test Program validation, and *iii)* Test program refinement, as depicted in Fig. 2.1. The first stage verifies the compilation results by checking the content of an Executable and Linkable Format (ELF) file. This file contains information about the device's resource usage that allows identifying significant compiler optimizations for a TP or STL, which may lead to removing, compaction, or replacing testing features (i.e., conditional statements generating test patterns (Tpats) or different data allocation).

In addition, this step considers further checks to programming structures and mISA instruction formats (i.e., call to routines, miscellaneous instructions, etc). When the initial checking does not succeed, the TP requires improvements through a refinement process.

The second stage (Test program validation) is divided into two sub-stages: logic simulation and fault simulation. The logic simulation verifies the correct functional execution of a TP using an RTL GPU model. Firstly, the ELF file is transformed into a GPU model-compliant Test-Bench containing the TP's information. Secondly, the GPU model executes the TP and captures the Signature per Thread (SpT) that indicates the fault-free status of the GPU and, in turn, serves to verify the program's correct operation. The fault simulation resorts to a customized simulation environment that takes the GPU's RT-level model and evaluates the Test Programs, targeting one GPU module and injecting stuck-at-faults (SAFs), one at a time. The fault simulation considers a fault as detected when at least one mismatch exists between the fault-free and the faulty SpT.

Finally, a given TP is considered valid if it is compliant with the TP specifications and fulfills minimum fault detection capabilities. Otherwise, the refinement step is used to improve fault coverage by making changes to the algorithm, described functions, or compiler settings.

2.2.4 Defeating compiler and architectural constraints for testing purposes

Developing STLs using HLLs or ILs presents challenges related to the constraints imposed by the compiler and the structural features of certain GPU modules. In order to solve such mapping constraints, several techniques can be applied to reduce or bypass the compiler optimization's effects and preserve testing capabilities in a TP. A first strategy requires the adoption of adequate coding styles to force the compiler to preserve test functionalities. These techniques include the efficient use of *'device intrinsic functions'* from libraries (e.g., *math.h*). For example, High-Level STLs for SFUs cores require intrinsic functions (e.g., *__log2f()* or *__exp2()*) to guarantee the generation of machine mISA instructions (i.e., SASS) addressing the modules and applying the desired test patterns (TPats). Fortunately, in the case of IL instructions for the SFU, these are directly mapped as vISA ones.

Other techniques manipulate the arguments (inputs/outputs of a kernel) to preserve instructions or routines targeting the GPU module. e.g., testing the GPRF requires several individual arguments (from 3 to 127) to force the compiler to allocate and address all possible registers per thread. Then, these arguments are loaded with external patterns. Similarly, testing the Scalar-Processors (SPs) includes additional arguments to generate most mISA formats and preserve TPats. Moreover, reducing the use of local variables and augmenting the input arguments in a kernel help suppress the replacement of instructions and the out-of-order organization of HLTPs. Alternatively, including explicit references to memory (e.g., global or shared) in at least one of the HLL's or ILL's operands prevents the compiler from optimizing the usage of immediate operands. In addition, the manipulation of variables in the shared memory reduces the compression, replacement, and reordering of instructions during compilation.

Other strategies require the creation of data dependencies between consecutive operations/instructions, the usage of the global memory to store partial results, and barrier synchronizations force the compiler to remove instructions to ensure that processed values are available for the next instructions. It is worth noting that these strategies can be used individually or in combination with mapping test specifications such as HLTPs and Intermediate-Level-Test-Programs ILTPs.

Table 2.1 Summary of the mapping strategies from the test algorithm to high-level programming languages and intermediate-level programming Languages (adapted from [3]).

Unit Type	Module	Test Method	Characteristics of the Test Strategy	HLL mapping(Note)	IL mapping(Note)
Regular	FU	A	• Tpats generation (ATPG/Pseudorandom) • Tpats processing as operations and operands • Operations grouping by type (logic, arithmetic) [65]	• The mapping is data-size aware, (8bit, 16bit, or 32bit) • Sequence of (+,-,*,/,%) operations, or math.h functions • Insert intrinsic functions (e.g., SFU operations) • Update an SpT status	• The mapping is data-size aware, (8bit, 16bit, or 32bit) • Explicit operands load from memory to registers • Consecutive vISA instructions (logic, arithmetic) • Insert dedicated SFU vISA instructions • Update an SpT status
	GPRF	A/D	• MARCH algorithm/ Custom method for SAFs as embarrassingly parallel function	• Declaration of local variables as many as HW registers • Consecutive R/W operation on local variables • Update an SpT status	• Declaration of virtual registers, as many as HW registers • Explicit operands load from memory to registers • Consecutive R/W on the virtual registers • Update an SpT status
Specific	PRF	D	• Controlled divergence management based on Tpats (e.g., checkerboard) as an embarrassingly parallel function [65] • Tpats transformation: operand values and relational operators	• Consecutive evaluation of conditional 'if' statements • Update an SpT status on each divergence path	• Explicit operands load from memory to registers • Consecutive control-flow vISA instructions • Update an SpT status on each divergence path
	ARF	D	• MARCH algorithm	Due to functional and language abstraction constraints, it's not possible to map the method at this level.	
	WSm	D	• Warp divergency management and routine placement [65]	• Local and shared variables for enabling/disabling warps • Loop over all warps and threads identifiers • Evaluate nested 'if' statements for warps and threads • Update the SpT on each divergence path • Update local and shared variables (software barrier) • Ending loop	• Explicit operands load from memory to registers • Virtual and private registers for enabling/disabling warps • vISA based loop implementation over warps and threads • Nested control-flow vISA instructions per warp per thread • Update the SpT on each divergence path • Insert barrier vISA instructions • Update virtual and private registers
	DSm	D	• Pyramidal nested divergency management: Consecutive nested divergences for stack addressing [65] • Synchronism management (SyncTrick): exploit synchronism for stack addressing [73]	• Repeat for threads in a warp and stack entries: • Consecutive conditional statements ('if') for operands • Update the SpT on each divergence path • SyncTrick mechanism can't be mapped	• Repeat for threads in a warp and stack entries: • Consecutive control-flow vISA instructions for operands • Update the SpT on each divergence path • SyncTrick mechanism can't be mapped
	DU	A/D	• Pseudorandom approaches for logic-arithmetic and memory-based operations. • Deterministic control-flow operations in combination with miscellaneous operations to build embarrassingly parallel functions [65]	• Various thread indexing implementations • Consecutive operations with different data types • Consecutive data type conversion and casting • R/W operations in shared. main and constant memories • Consecutive conditional 'if' statements • Unconditional branching by nested function invocations • Update the SpT after each operation • Miscellaneous operations cannot be mapped at this level	• Explicit memory address calculations • Load operands from memory to local registers • Consecutive vISA instructions with different data types • Interleave control-flow vISA instructions • Implement nested subroutine calls (unconditional branching) • Update the SpT after each vISA instruction • Include miscellaneous vISA instructions if available
Hidden Units	PR	C	• Multi Kernel approach: particular test methods addresing different hidden elements of the targeted structures [77]	• Several cooperative thread array configuration per function • The function descriptions follow the methods previously described for the other units	

A: Automated; D: Deterministic; C: Custom
FU: Functional Units; GPRF: General-Purpose Reg. File; PRF: Predicate Reg. File; ARF: Address Reg. File; Wsm: Warp Scheduler Memory; DSm: Divergence Stack; DU: Decoder Unit; PR: Pipeline Reg.
Note: Kernel function must have input arguments; Operands allocated in memory (main, shared, constant); Immediate or literal operands are allowed; At least one operand must be a memory reference

2.3 HLL's mapping of test algorithms for representative GPU units

This subsection targets several modules in the GPU and applies the methodology to develop test programs and STLs using high-level programming languages. Table 2.1 introduces the main mapping strategies that can be adopted to implement different test methods into HLLs and ILLs for a given GPU module. The reported information covers diverse modules and test strategies organized by unit type, GPU module, test method, detailed test strategy, HLL mapping strategy, and ILL mapping strategy.

It is worth noting that this work followed state-of-the-art test procedures, originally devised for implementation at the assembly level (SASS) only, and conceived their implementation as STLs using HLLs and ILLs. In the following, we introduce the main procedures that successfully allow the implementation of a test algorithm into a high-level programming language able to test the main building blocks of a GPU, providing insights about their possible limitations and solutions in comparison with direct assembly implementations

2.3.1 Functional Units and Register File

These modules are regular structures in the GPU, and several units are available and used by different threads of an active warp. More in detail, modern GPU architectures allow each thread to address the functional units statically, simplifying the management of the implicit parallelism of a program. Moreover, in these modules, the instructions addressing the functional units employ any user-accessible memory resources, allowing direct application of test patterns as input operands. Similarly, the results are always stored inside the register file, so the fault observability of a fault does not represent an issue.

The combination of the structural organization and the direct controllability and observability allows the adoption of automated methods as described in [78], which resort to using pseudorandom approaches or ATPG tools to generate the input test patterns for a given unit (e.g., Floating Point Units, or Special Function Units). Then, these patterns can be transformed into equivalent test routines. For the HLL mapping procedure (e.g., using CUDA C++), the test programs are developed using three steps: i) test pattern identification, ii) pattern grouping, and iii) fine-grain TP

description. Firstly, the definition of the test patterns uses one of two automatic approaches (pseudorandom or ATPG-based). Then, the pattern grouping consists of each pattern's organization according to the operations to be issued and the operand's size. In this case, there is the option of employing any of the memory resources in the GPU (i.e., global or shared memories) to store the patterns and load them during the execution of the test program. The option of using immediate operands is discarded by the lack of fine-grain control in high-level operations at the CUDA C++ level. Finally, the test program description uses functions to ensure that each pattern is replicated and applied to the functional units, ensuring the test's coherence on these modules.

2.3.2 Embedded Memories

These memories (Predicate Register File or PRF and Address Register File or ARF) are regular structures used by each active thread in a warp. However, the main constraints of both modules are their controllability and observability features. In fact, the PRF can only be addressed indirectly (as the product of other operations). Similarly, the ARF is used to address other memory resources, so TPs for these modules require additional routines to deal with these constraints. The method to test these memories is based on the generation of the indirect conditions to address and excite each part of the modules.

Authors in [74] proposed low-level test programs to detect permanent faults in both modules. In the case of the PRF, the test programs generated a sequence of conditional operations, so individual predicate flags were activated in the PRF, and the effect was observed by conditioning the operation of an SpT on the state of the target predicate. Similarly, the ARF was tested using specialized instructions able to provide any test patterns inside the module. At HLL, the mapping procedure is applied to each module independently. On the one hand, for PRF, an embarrassingly parallel test program is developed, so each thread addresses the assigned set of registers in the PRF in parallel. The test program consists of a sequence of operations forcing the activation of individual predicates on the PRF. Then, after each operation, one conditional statement is used to detect the activation of a target predicate.

Finally, updating and saving a signature-per-thread (SpT) allows for the detection of any fault. On the other hand, there are no methods to map or design a test program

for the ARF using an HLL. Unfortunately, this mapping is not possible because there are no methods or functions to explicitly address the ARF. In fact, CUDA C++ or PTX do not provide a method to state the used memory space or explicitly select the memory address [124], so during the compilation process, the addresses are assigned, thus compromising the application of a given test pattern. For this case, the use of a low-level programming language (e.g., Shader ASSembly or (SASS)) is the only alternative.

2.3.3 Warp Scheduler Memory

The warp scheduler memory is a specific module in a GPU. Thus, the development of TPs for this module requires a combination of deterministic and custom methods to address the operational and observability restrictions. Authors in [76] proposed two strategies to activate and detect faults in the warp scheduler: 1) divergence management and 2) routine placement.

The Divergence management strategy induces divergencies per thread, such as forcing the active and inactive threads field of a given warp, so exciting a target number of locations in the scheduler's memory. on the other hand, the *Routine placement* strategy allocates routines in different address locations seeking to excite additional locations in the warp scheduler memory, such as the warp program counter field. In both cases, an SpT is updated to propagate any fault effect to an observable point.

The divergence management algorithm can directly be mapped to HLL (e.g., CUDA C++) as a consecutive set of conditional statements "if", employing the thread indices as comparison elements according to the snippet code presented in Figure 2.2. The same idea applies to describing this testing algorithm into ILL (e.g., CUDA PTX) but, in this case, explicitly using the predicate instructions for compare and jump as described in Figure 2.2. An alternative feature includes the use of implicit parallelism to describe the divergence management at the warp level so that each warp evaluates different conditions. In this case, software mechanisms such as semaphores, shared variables, and events are used to pause the execution of a given warp and force the operation of a selected one.

Unfortunately, the *explicit placement* strategy can not be implemented at the CUDA C++ level, and there is no method to map this functionality at such level nor

using CUDA PTX. Thus, in this case, the combination of HLL and LLL (e.g., SASS) is required to maintain the fault coverage in the test program.

```
1   __global__ void Divergence_Stack_T(int* SpT, int* vars){
2   ...
3   int Tid = blockDim.x * blockIdx.x + threadIdx.x;
4   ...
5   if (Tid == vars[0]) {
6   ... }                    ⇐ SpT_update(fault-free value);
7   else{
8   ... }                    ⇐ SpT_update(fault value);
9   ...
10  if (Tid == vars[n]) {
11  ... }                    ⇐ SpT_update(fault-free value);
12  else{
13  ... }                    ⇐ SpT_update(fault value);
14  }
```

(a)

```
1   .entry Divergence_Stack_T(.param, .u64 SpT .param .u64 vars){
2   ...                                18  <SpT_update_fault_free_path_ops>
3   cvt.u32.u16 %r1, %tid.x;           19  continue:
4   mov.u16 %rh1, %ctaid.x;            20  ...
5   mov.u16 %rh2, %ntid.x;             21  add.u64 %rd4, %rd4, N;
6   mul.wide.u16 %r2, %rh1, %rh2;      22  ld.global.u32 %r5, [%rd4+0];
7   add.u32 %r3, %r1, %r2;             23  setp.eq.u32 p, %r3, %r5;
8   mul.wide.s32 %rd2, %r3, 4;         24  @p bra SpT_update_FF;
9   ...                                25  <SpT_update_faulty_path_ops>
10  ld.param.u64 %rd3, [vars];         26  bra continue;
11  add.u64 %rd4, %rd3, %rd2;          27  SpT_update_FF:
12  ld.global.u32 %r5, [%rd4+0];       28  <SpT_update_fault_free_path_ops>
13  setp.eq.u32 p, %r3, %r5;           29  continue:
14  @p bra SpT_update_FF;              30  ...
15  <SpT_update_faulty_path_ops>       31  }
16  bra continue;
17  SpT_update_FF:
```

(b)

Fig. 2.2 An example of mapping the test strategies for the Divergence Stack into HLLs and ILLs. (a) CUDA C++ implementation (b) CUDA PTX implementation (adapted from [3]).

2.3.4 Divergence Stack Memory

This specific module in the GPU requires a functional test method based on the management of control-flow operations to address each location in the divergence stack. In [73], the authors proposed two testing strategies by resorting to assembly instructions: *Nesting* and *syncTrick*.

The *Nesting* strategy uses conditional branches, controlled divergences, and nested divergences, allowing the stack pointer to move through the divergence stack memory. On the other hand, the *syncTrick* exploits the functionality of the SSY instruction to force the Divergence Management Unit (DMU) to change the stack

pointer. This method allocates SSY operations in strategically selected locations in the test program to create an input pattern in a new stack line.

The first strategy *(Nesting)* can be partially mapped into the high-level description or intermediate language. In such cases, the procedure requires four steps: *i)* stack addressing, *ii)* active threads management, *iii)* warp management, and *iv)* fault propagation. The *stack addressing* consists of generating divergence or calling sub-functions, so forcing the compiler to address a new line in the stack. Fig. 2.2 shows a CUDA C++ and CUDA PTX snippet sample code of conditional statements that describe the *active thread management* using controlled divergences. *Warp management* consists of locating the routines in selected places of the memory, so patterns generated by the warp program counters are indirectly applied. In this case, compiler constraints force the use of assembly instructions to locate the routines in carefully selected locations. Finally, the *fault propagation* is based on an SpT mechanism that can be directly described at a high level for each thread and performs an arithmetic operation to identify the presence or absence of faults in the module.

Unfortunately, the mapping of the *syncTrick* strategy is not feasible at HLL (not even using CUDA PTX!). The explicit use of some control-flow instructions to optimize the stack addressing is not allowed at a high level.

2.3.5 Pipeline Registers

There exist multiple units in GPUs that are not visible to programmers. However, if any of these units experience a malfunction, it can lead to the failure of an application or the GPU's operation. The pipeline registers are one such group of units that fall under this category. These registers are spread across the GPU and store sensitive information for the operation of the GPU core. The design of test programs for pipeline registers requires a combination of multiple strategies, making it a complex unit to be tested through STLs.

In [77], the authors proposed a multi-kernel approach to address the test of the pipeline registers. One interesting aspect is that the kernels can be created at a high level. In certain cases, the description mixes automatic and deterministic approaches. However, due to the limitations of the HLL (e.g., CUDA C++), it is necessary to provide a detailed description of the test program at the assembly level (e.g., SASS) since some locations in the pipeline registers require the evaluation of most

instructions and formats from the ISA, which is not feasible at high-level. Hence, in order to keep high fault coverage features, this unit requires the combination of high-level and low-level test programs for a complete implementation of the devised test approach.

2.3.6 Decoder Unit

Testing this unit requires executing various instructions and valid formats in the ISA. Hence, the test engineer should have in-depth knowledge of the ISA's specification details. The process can be broken down into multiple stages to test the decode unit [3]. The first stage includes logic, integer, and floating-point instructions using different operational arguments such as immediate, register, or memory. The second stage includes branch, control flow, and miscellaneous instructions. Each group of instructions can have one specific test program designed by automated methods [78].

There are several limitations when it comes to describing and implementing the proposed solution from an HLL perspective. While it is possible to generate most instructions and formats from high-level descriptions, the instructions used in a kernel are reduced and simplified during the compilation process to maximize performance. This may result in limited input test patterns to the units, which in turn affects the maximum fault coverage achievable by the test program. Unfortunately, the compiler does not allow a straightforward mapping of the code at the CUDA C++ level to the assembly level in a fully predictable way. Additionally, the use of explicit instructions at the PTX level does not allow for the implementation of all possible GPU instructions either. As a result, the testing of this unit from HLL can only be carried out using a limited number of logic, integer, floating-point, and branch instructions. In order to keep high fault coverage, the group of instructions composed of miscellaneous, immediate, branches, and memory addressing instructions has to be manually added at the SASS level later.

2.4 Experimental results

The FlexGripPlus GPU model was used to validate the proposed approach, and several *'High-Level-Test-Programs'* HLTPs and *Intermediate-Level-Test-Programs* ILTPs were developed targeting different modules inside this GPU model. The

effectiveness of the developed STLs has been evaluated through fault simulation experiments resorting to commercial EDA tools and considering the stuck-at-fault (SAF) model. These fault simulation campaigns were performed on a workstation with two AMD EPYC 7301 16-core processors running at 2.2GHz and equipped with 128 GB of RAM memory. The FlexGripPlus GPU was configured with one SM, 8 SPs, and 2 SFUs. The GPU model was synthesized using the 15nm NanGate OCL synthesis library [125].

Table 2.2 GPU modules features and their STL development approach (adapted from [3]).

Unit Type	Module	Num of cells (*)	Test Method	STL Mapping		
				CUDA C++	CUDA PTX	SASS
Regular	Streaming Processor (SP)	206,824	A	F	F	F
	Special Function Units (SFU)	90,982	A	F	F	F
	General-Purpose Reg File (GPRF)	524,288	D	F	F	F
	Predicate Reg File (PRF)	16,384	D	P	P	F
	Address Reg File (ARF)	131,072	D	-	-	F
Specific	Warp Scheduler mem (WSm)	5,118	D	P	P	F
	Divergence Stack mem (DSM)	273,600	D	P	P	F
	Decoder Unit (DU)	1,896	A D	P	P	F
Others	Pipeline Regs (PRs)	2,382	C	P	P	F

(*) Combinational and sequential cells using the synthesis library 15nm NanGate OCL
(F) Test algorithm fully mapped into the target programming language
(P) Test algorithm Partially mapped into the target programming language
(-) Test algorithm not mapped into the target programming language
A: Automated; D: Deterministic; C: Custom

The HLTPs and ILTPs were written in CUDA C++ and CUDA PTX, respectively. These test programs were compiled using CUDA toolkit SDK 5.0 with a Compute Capability 1.0 for FlexGripPlus GPU. Additionally, the test programs were implemented on two additional GPUs (NVIDIA Jetson Nano and GeForce GTX 960M) that were employed to evaluate the TP's execution and observe the compilation impact of various coding styles in different environments (CUDA SDK 11.2 and CC 5.3, and CUDA SDK 5.0 and CC 5.1).

Table 2.2 summarizes the most important features of the fundamental hardware modules inside a GPU and the most important characteristics considered for developing STLs for testing such units. More in detail, the table reports the type of unit (i.e., regular, GPU' specific, or others), the specific target module (e.g., Streaming Processor, register files), and the size of the hardware unit in terms of the number of cells. In addition, the table indicates the test method used to develop a given test program and the main aspects of mapping such test program specifications into HLLs, ILLs, or LLLs.

Table 2.3 Main features of the implemented STLs for regular units and embedded memories of the GPU (adapted from [3]).

GPU Module	HLSTLs (CUDA C++)			ILSTLs (CUDA PTX)			LLSTLs (SASS)		
	Duration (cc)	Size (instr)	FC (%)	Duration (cc)	Size (instr)	FC (%)	Duration (cc)	Size (instr)	FC (%)
SP	5,366,208	76,513	86.95	5,922,414	1079	81.94	4,881,855	74,604	87.20
SFU	1,331,200	16,856	94.30	212,914	117	94.30	212,914	117	94.30
GPRF	3,256,058	698	100.00	-	-	-	108,958	82	100.00
ARF	-	-	-	-	-	-	338,240	122	100.00

On the other hand, Table 2.3 reports the implementation details and the Fault Coverage (FC) results of the developed and implemented STLs for regular units, only. The results show that the development of STLs using HLLs and ILs can effectively achieve equivalent fault coverage to the ones developed using LLL description (e.g., using SASS implementations). In fact, the effectiveness of STLs is valid for hardware structures that are fully controllable and observable; in other words, they are hardware structures visible to the programmer (e.g., SPs, SFUs, and GPRF). In the case of the embedded memory ARF, the specific test algorithm reported in section 2.3 cannot be implemented at HLL or ILL, so there is no guarantee of getting acceptable fault coverage when relying on STLs developed in such programming levels.

Moreover, the test duration, in clock cycles, for the high-level STLs (i.e., CUDA C++ STLs) is longer than its equivalent LLSTL (SASS STLs) versions (from 1.1 to 6 times for SPs and SFUs). This cost can be explained by the explicit global memory operations (reading and writing) in the HLTPs to prevent compiler optimizations. Moreover, the test patterns must be replicated inside each block of threads to produce the necessary redundancy in the operations inside the SM, avoiding the scheduling intervention during the test of these units.

Similarly, the duration of Intermediate-Level STLs is longer than the equivalent Low-Level STL developed to test the 'Scalar-Processors' SPs. However, the required number of instructions in Intermediate-Level STLs is significantly reduced since the test program description in CUDA PTX allows more friendly fine-grain management than mISA (i.e., SASS) descriptions, allowing a flexible allocation of operands in memory as well as embedded in the instructions as immediate constants. This memory management also increases the degree of parallelism in CUDA PTX, helping the reduction of test programs. In the case of STLs for SFUs, the description of the Intermediate-Level test program is more straightforward, permitting the generation

of a test program remarkably similar to the equivalent directly developed using Low-Level languages.

Notably, the FC achieved in high-level test programs and intermediate-level test programs for testing the SP cores can sometimes be moderately lower than the FC achieved by the equivalent test programs developed using device assembly directly (e.g., SASS). The main reason behind this slight FC reduction is the SpT computation, which uses logic/arithmetic instructions in the SPs. Although the SpT algorithms encoded in high-level and intermediate-level test programs are functionally equivalent, the compiler produces different SpT versions compared to those at the low level using SASS. Consequently, the FC capabilities of these test programs vary (the patterns produced by the SpT on SPs strongly depend on the mISA instructions used for that purpose).

Table 2.4 Main features of the implemented STLs for the special units of a GPU (adapted from [3]).

GPU	CUDA C++			CUDA PTX			SASS		
Module	Duration (cc)	Size (instr)	FC (%)	Duration (cc)	Size (instr)	FC (%)	Duration (cc)	Size (instr)	FC (%)
WSm	98,480	276	38.20	-	-	-	112,200	392	100.00
DU	2,150,612	12,354	68.75	1,589,678	12,116	73.74	6,125,561	65,653	80.10
DSM	987,526	875,422	35.10	-	-	-	1,030,473	12,524*	98.40
PRF	1,750,023	392	28.00	-	-	-	1,890,106	434	100.00
PRs	649,400	22,292	80.20	-	-	-	1,204,097	27,492	95.10

(*) The use of SASS instructions allowed a significant reduction in the total size of a TP

Table 2.4 reports the main features results of the test programs developed for the distinctive GPU's units (i.e., warp scheduler, divergence stack, and decoder units), predicate registers, and pipeline registers. As the reader can notice, the high-level test programs have limited testing capabilities (about 28% to 68.75% FC) due to the partial mapping of the evaluated test algorithms into HLLs (i.e., CUDA C++) combined with the compilation impact. Indeed, the test of these modules resorts to specific algorithms, which are functionally characterized by low performance. Thus, the compiler modifies the code, following unavoidable optimization philosophies that affect the achieved FC. In fact, the compiler takes the high-level test programs and produces an optimized version of it, maintaining the functional equivalence, but it removes or changes the execution order of operations, so affecting the FC and producing a negative impact on the testing capabilities of the high-level test program. Consequently, to guarantee fault detection capabilities, it is necessary to adopt hybrid approaches, which means combining portions of the test program using high-level

languages with portions added manually using assembly instructions. In fact, these hybrid strategies were applied to the PRFs units and DSM units, allowing the test programs to reach an FC of 100% and 98.41%, respectively.

It is worth noting that such additional assembly instructions in the hybrid test programs are required to inject additional test patterns by addressing a module in specific conditions (i.e., addressing memory locations or addressing different stack lines in the DSM). It is important to mention that for the DSM, the final insertion of assembly instructions also reduced the routine complexity, so compacting the test program. Nonetheless, the test engineer decides when such manual addition of instructions is justified considering the tradeoff between productivity, the effort for the test program generation, and the test coverage improvement.

On the other hand, the Intermediate-Level test programs to test the WSm, DSM, PRF, and PRs produced identical results in terms of duration, size, and FC obtained from the equivalent ones developed using high-level languages. This characteristic behavior is mainly due to the deterministic nature of the test methods used to test such modules, (e.g., based on specific operations, such as conditional statements to induce controlled divergence). However, the Intermediate-Level test program for the DU reaches higher coverage (73.74%) than equivalent ones using high-level languages that reach only 68.75%. In fact, the direct usage of some miscellaneous and control-flow instructions at the CUDA PTX level provides fine-grain control to produce test patterns, which cannot be generated using CUDA C++.

Finally, the test programs implemented using only high-level or intermediate-level languages can test around 50.6% out of the GPU's faults, and represent 9.3% of the size of the STLs. Moreover, hybrid STLs, improved by additional mISA instructions, can test 45% of the faults and occupy 90.6% of the size of the STLs. Additionally, based on our experience, the adoption of High-Level languages for STL's development decreases the development time by about two orders of magnitude (for the functional units), and one order of magnitude (for other modules), resorting to the combination of CUDA C++ and SASS. Actually, this development time reduction is aligned with the statements presented by [122, 123] regarding improving programmer's productivity when adopting High-Level languages for application development, so demonstrating that this productivity improvement is achievable also when developing STL for GPUs as well. On the other hand, the adoption of intermediate-level languages for the development of STLs offers higher flexibility,

reducing the development effort by around 30% with respect to the direct use of assembly language for the development of the same test programs or STLs.

2.5 Final remarks

This chapter introduced a methodology for adopting high-level and intermediate-level programming languages for developing test programs and software test libraries (STLs) applied to the in-field testing of GPUs under the Software-Based Self-Test (SBST) approach. The proposed method leverages the divide-and-conquer strategy to create test programs targeting individual GPU hardware modules, simplifying the development process. This modular approach allows the adaptation or generation of test procedures, especially tailor-made for each of the GPU units. The selected test procedures are then mapped into test programs using high-level languages, paying special attention to suitable coding styles to either accurately describe the test algorithms or prevent compiler optimizations.

The experimental evaluation shows that high-level or intermediate-level programming languages can be effectively used to detect permanent faults in GPUs when test programs are developed to test regular units in the GPU (e.g., functional units or register files), since those hardware structures are visible resources to the programmer, providing high controllability over test patterns and fault observability. On the other hand, the fault detection capabilities are limited when targeting more complex units such as schedulers and controllers. In such cases, the test programs need to be enhanced after the compilation with some specific instructions removed or not included by the compiler.

In general, developing STLs for GPUs using high-level or intermediate-level programming languages is a good solution when the GPU ISA specifications are restricted, or the documentation is not fully available. Nonetheless, there are constraints and challenges in the development of STLs using such programming abstractions as pattern controllability and fault observability. In addition, compiler intervention plays a crucial role in the generation of the final test program and its fault-detection capabilities.

Although STLs provide acceptable fault detection capabilities, combining them with other in-field test strategies is necessary to achieve maximum coverage. For

example, in a given safety-critical system like an autonomous vehicle, the power-on and power-off test can be applied using built-in self-test solutions (e.g., BIST, MBIST, among others). On the other hand, the STLs can be embedded in the system's software to apply periodic tests during the system operation, leveraging the idle states of the GPU to detect any malfunction. In addition, the STLs have the potential to detect different fault models besides stuck-at faults, such as delay and cell-aware faults. Nonetheless, the fault coverage capabilities can be significantly limited, requiring additional test pattern sequences or algorithm improvements that are not covered in the present thesis, opening further research fronts in the future.

Chapter 3

Compaction of STLs for GPUs

This chapter introduces for the first time a time-efficient compaction method, which aims to reduce the size and duration of self-test libraries (STLs) used for testing CPUs or GPUs. This novel test compaction approach was previously published by the author of this thesis in [4] and [5]. This method is based on the knowledge of the structure of most test programs, which consist of a sequence of instructions grouped in blocks, also known as *basic blocks* [126]. The basic block structure is common in test programs that are created by converting test patterns generated by combinational Automatic Test Pattern Generators (ATPGs), evolutionary approaches, or pseudorandom approaches.

This basic block structure allows for easy identification of portions of code that do not detect faults so that we can remove them from the test program, making it more compact and efficient in terms of size and duration. More in detail, the novel compaction approach presented in this chapter employs different abstraction levels (i.e., software, RT level, and gate level) to perform only ONE logic simulation and ONE fault simulation, thus significantly reducing the required time to obtain a compacted version of the test programs and having minimum impact on the FC with respect to the original one.

In both logic and fault simulations, several parameters are collected and extracted to support the compaction of test programs and STLs. Firstly, a logic simulation using the RT-level model of a given device (CPU or GPU) is used to gather detailed tracing information about the executed test program in every clock cycle. Secondly, a fault simulation is performed using the gate-level version of the circuit or target

core. This fault simulation also records the number of faults detected at each clock cycle. The results obtained in both simulations are used to identify those instructions at the software level (i.e., basic blocks) that are unable to stimulate or propagate fault effects in a target module, so they are listed as candidates for elimination.

The main contributions can be summarized as follows:

- Elaboration of the main concept of the proposed compaction methodology for STLs in CPU cores
- Extension of the proposed compaction methodology from the CPU concept to GPU scenarios, considering the parallelism and operational characteristics of such devices.

3.1 Background

Safety-critical applications increasingly employ CPUs and GPUs as the main workhorse to perform complex operations and process large amounts of information (e.g., for artificial intelligence and sensor fusion operations). However, in this domain, effective methods to identify possible permanent faults arising in such devices and to face their effects are crucial goals set by functional safety standards. Software-based self-testing (SBST) is an effective testing approach developed for processors and GPUs through the generation of a set of Test Programs (TPs) known as Self Test Libraries, as described in chapter 1.

When working with STLs, it is important to keep in mind that each Test Program requires a specific amount of time to be executed. However, there may be various application constraints that limit the available execution time. To overcome this challenge, it is ideal to have shorter and faster test programs that can be easily adapted to the available time slots during in-field tests. One way to achieve this is by using compaction methods, which can optimize the test programs and STLs in terms of size (i.e., memory footprint) and duration (i.e., amount of clock cycles). However, the compaction of any test program can be a challenging task, since identifying and reducing portions of test programs that are not capable of detecting faults depends on several aspects, such as the type of hardware (e.g., CPU or GPU), the test program generation method (e.g., custom, ATPG-based, pseudorandom, deterministic), and the required time to make the compaction.

3.1.1 Related works

Several works have proposed methods to compact test programs targeting processor-based systems. These techniques reduce the size and duration of a test program without affecting the Fault Coverage features. Most of these techniques are based on complex evolutionary algorithms, instruction classification, and removal, or a combination of both.

Authors in [127] proposed a compaction method based on an evolutionary approach. This method transforms a TP into several small segments (or *spores*). Each spore program detects a portion of faults from the original program and, once combined, detects the same number of faults as the original program. Then, an evolutionary procedure is performed to get a subset of spores that allow the same fault coverage as the original test program. It is important to mention that the compaction cost of this method is significantly, high since hundreds or thousands of iterations are required in order to obtain acceptable results.

In [85], authors introduced another compaction evolutionary technique to compact test programs developed for CPUs. The authors propose a technique called *genetic programming* that removes redundant instructions in the final test program that do not contribute to any fault detection capabilities. In addition, the authors demonstrate that their *genetic programming* method shows equivalent fault coverage than classical approaches using manual and random test program generation. In addition, they demonstrate that the *genetic programming* technique significantly reduces the size and duration of the generated test programs in comparison with other test program generation strategies.

In [84], authors presented a static compaction technique for test programs developed for CPU testing. This compaction approach considers the availability of a collection of test programs or a Self-Test Library (STL) designed to test several units of a CPU. In this regard, it is possible that some test programs designed for a given hardware structure *A* are also capable of testing or detecting faults on another hardware unit *B*. Under this reasoning, they propose to remove those test programs from the STLs, so that their fault detection capabilities can be covered by other test programs. In this case, only a subset of test programs are considered fundamental for testing the whole CPU, which also reduces the test time and maintains the same

global Fault Coverage. Nonetheless, their approach is not capable of reducing further instructions or portions of the test programs that are not capable of detecting faults.

Authors in [128, 83] presented two compacting methods based on the removal of instructions from a given test program. The first method is called *compaction by instruction removal* (A0). This compaction approach takes the original test program and creates a new one by removing one instruction at a time. For every new test program (A0), a complete fault simulation is performed to observe the effect of the instruction removal in terms of fault coverage. If the instruction removal leads to a fault coverage reduction, then such instruction is reinserted in the program. Otherwise, the instruction is deleted permanently from the test program. This procedure is repeated iteratively for all available instructions in a test program.

The second compaction method proposed by the authors is called *Restoration-based Algorithm* (A1xx). This algorithm initially splits the test program into small blocks of instructions. Then, one block of instructions is removed, and each instruction is individually added to the main program to reach the original fault coverage. If one instruction inside the block does not increase the fault detection capabilities, it is removed permanently from the block of instructions.

Finally, authors in [82] described a compaction mechanism for test programs for MIPS processors. The method is composed of two stages. The first stage identifies redundant instructions using dependency data graph techniques. This graph analysis permits the identification of the data dependency among instructions in order to determine the group of instructions that directly interact in the detection of a fault of a set of faults, preventing the removal of individual instructions that are essential for the test. The second stage reuses the A1xx idea and exploits optimization approaches to reduce the computational cost when executing the compaction algorithm.

Most previous research on the compaction of test programs for in-field test mainly resorts to complex optimization algorithms (e.g., evolutionary algorithms) or iterative approaches that significantly increase the compaction cost. Moreover, these compaction methods usually require a high number of fault simulations proportional to the length of the original test programs (i.e., in terms of the number of instructions), which exacerbates the compaction costs. It is worth noticing that none of the reported techniques in the literature face the compaction of test programs and STLs for GPUs, and some of them can hardly be extended from CPUs to GPUs either.

In order to face the complexity and compaction cost presented by other approaches, this thesis introduces a novel compaction approach that relies on three main aspects: software description, logic simulation, and fault simulation. The logic simulation extracts the microarchitectural status of the circuit by generating a tracing report during the execution of the test program on the target device. Then, ONE fault simulation creates a report of detected faults per test pattern on the target unit. The information from these two simulations allows us to identify the relationship between the fault detection capabilities of each instruction in a test program. This correlation enables the identification of the main candidate's instructions for compaction. Moreover, the compacting approach exploits the microarchitecture-instruction-fault relation to reduce test program size and duration with minimal impact on the final fault coverage.

3.2 Compaction of STLs for CPUs: main concept

3.2.1 Proposed methodology

The proposed approach assumes that a Self-Test Library for a given CPU processor is available and can be divided into test programs. Each test program is composed of a given number of instructions, lasting for a certain amount of clock cycles, and achieving a certain fault coverage with respect to a given fault model. Moreover, those test programs can be split into sets of instructions called *Basic Blocks* or (BBs), each corresponding to a consecutive group of instructions, which are always executed in sequence (no branches or jumps in/out the BB) [126].

More in detail, the structure of a BB in a typical test program comprises instructions to perform three main actions:

1. One or several instructions devoted to loading test data into registers
2. One or several instructions using these loaded values to stimulate some target module
3. One or several instructions dedicated to making the results produced by the target module visible on some observable point.

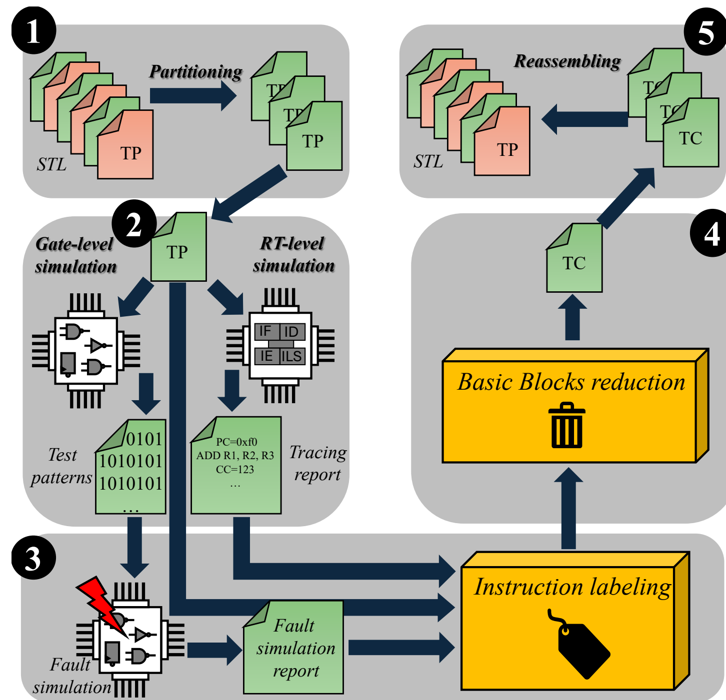


Fig. 3.1 A general scheme of the proposed compaction approach applied to STLs developed for in-field testing of CPUs (adapted from [4]).

The proposed compaction approach introduced in this chapter resorts to BB removal without restoration, combining logic tracing information of the test program with fault simulation results. More in detail, this compaction approach comprises five consecutive stages: ❶ Test program partitioning, ❷ Logic tracing, ❸ Fault detection analysis and instruction labeling, ❹ Program reduction, and ❺ Test program reassembling. Figure 3.1 depicts the steps and the main flow of the proposed compaction approach.

In the first stage, the program partitioning stage, Figure 3.1 ❶, the test programs or STLs are analyzed to select or identify the portions or regions of the original test program that are candidates for the compaction. Such portions of test programs are characterized by including at least one unconditioned BB, i.e., a BB that is executed only once and does not depend on any condition to be executed. Thus, this compaction approach considers those regions in a test program (composed of unconditioned BBs, only) as admissible regions [86], while the other parts are not candidates for compaction and remain unaffected. This partitioning process can be

easily automated. Moreover, the other stages only consider the admissible regions of each test program.

The logic tracing stage, Figure 3.1 ❷, aims to collect information during the execution of the test program on the microarchitectural level of the processor in order to later identify the relationship between each instruction inside each BB and its effects in terms of fault detection capabilities. In this stage, each test program is analyzed using one RTL logic simulation and one gate-level (GL) logic simulation.

On the one hand, the RTL logic simulation provides the main execution information, per clock cycle, about the test program with respect to the target device. This simulation produces one tracing report that captures the detailed information of the HW-SW interface. The tracing report contains the following information: *i)* the program counter value, *ii)* the decoded instruction (i.e., mnemonic), and *iii)* the clock cycle. In order to generate this tracing report, one hardware monitor must be included in the RTL model of the device that generates the report and captures the status information in the execution pipeline stage. This instrumentation of the RTL design enables flexibility, such as the results of the executed instructions, which can be observed in the memory bus system or in any other observation point inside the pipeline. Therefore, any malfunction caused by a fault can be identified at any of these points without tracing each pipeline stage.

On the other hand, the GL logic simulation runs each test program and generates the input sequence of logic values per clock cycle, also known as test patterns. In other words, These test patterns are extracted as I/O switching activity for the target processor or component inside the processor and are employed by the fault simulation in the next stage.

The fault detection analysis stage, Figure 3.1 ❸, is divided into two steps: (1) fault simulation and (2) instruction labeling. In the first step, ONE fault simulation is performed on the gate-level description. This fault simulation employs, as inputs, the test patterns generated by the GL logic simulation in the previous stage. Moreover, the fault observation point is restricted to the memory bus system. During the fault simulation, one fault is detected when there is a difference between the fault-free system and the current faulty system at any clock cycle. The fault simulation generates a report containing the list of all test patterns and the number of faults activated and detected by each test pattern. It is worth noting that the proposed

compaction method only requires ONE fault simulation for each test program under analysis.

Algorithm 1 Instruction labeling algorithm (adapted from [4]).

Input: Test program TP, Tracing clock cycle report CC, Tracing program counter report PC, Tracing decoded instruction report DI, Fault sim test patterns report FSR
Output: Labeled test program TPL

```
1: for each clock cycle k in CC do
2:     if FSR(k) detects faults then
3:         if DI(k) matches TP(PC(k)) then
4:             TPLk:= (essential, TP(PC(k))
5:         end if
6:     else
7:         TPLk:= (not essential, TP(PC(k))
8:     end if
9: end for
```

The second step (instruction labeling) correlates the fault simulation report and the collected trace information from the RTL logic simulation. The Algorithm 1 provides deeper insights into this instruction labeling procedure. first, the labeling procedure requires five inputs: *i)* the test program source codes in assembly language (TP), *ii)* the tracing clock cycle report (CC), *iii)* the tracing program counter (PC) on every clock cycle, *iv)* the tracing instruction opcodes or decoded mnemonics (DI) per clock cycle, and *v)* the fault simulation report per test pattern (FSR). Then, the labeling algorithm generates a new test program version (TPLk) where every instruction has one out of two labels *"essential"* or *"not essential"* according to the fault detection capabilities. Finally, in order to identify whether an instruction is *"essential"* or not, the labeling algorithm iterates over every clock cycle k in the CC report and identifies the test patterns $FSR(k)$ associated with the instruction in the same clock cycle k. This instruction matching takes the executed instructions, which are registered in the tracing report on a given clock cycle (k), and crosschecks the number of faults detected by the equivalent test pattern in the fault simulation in the same clock cycle k (i.e., lines 2 to 3). If the fault simulation reports that the test pattern $FSR(k)$ provides fault detection capabilities on such k clock cycle, the associated instruction is then backtracked in the source code and consequently labeled as *“essential.”* Otherwise, the instruction is labeled as *“not essential,”* and it becomes a candidate instruction that needs to be removed from the program since it does not contribute to the detection of any fault.

Algorithm 2 Reduction algorithm applied to test programs used to remove *basic blocks* (adapted from [4]).

Input: Labeled Test program TL with I instructions divided into m consecutive BBs (BB1, BB2, BB3, . . ., BBm);
Output: Compacted test program TC

```
1: for each basic bloc BBx in TL do
2:     for each instruction in BBx do
3:         if label of instruction is essential then
4:             TC: append (BBx)
5:             continue to next basic block
6:         end if
7:     end for
8: end for
```

The fourth stage (Program reduction in Figure 3.1 ❹) analyzes and reduces the labeled test programs by removing *not essential* BBs. The Algorithm 2 describes the reduction algorithm used in the compaction approach presented in this thesis. The reduction procedure requires inputs from the labeled test program obtained from the previous stage, which must be divided into consecutive basic blocks. Then, the reduction algorithm examines instruction by instruction, each BB in a given labeled test program (TL). When at least one instruction inside the BB is marked as *"essential,"*, the entire BB cannot be removed from the test program because that means that all instructions in the BB collectively contribute to detecting faults on the target core. Therefore, such essential BB is added to the final compacted test program (TC). This procedure effectively discards only those BBs in which all instructions are labeled as "not essential."

Finally, the last stage of the compaction strategy (Reassembling in Figure 3.1 ❺) replaces the original version of the test programs using the new one obtained from the compaction procedure. Finally, an additional fault simulation can be used to validate the fault coverage obtained after the compaction of the test programs or the complete STL. This last step permits the assessment of the quality of the compaction in terms of test duration, memory footprint, and fault coverage.

3.2.2 Study cases

This section showcases the case studies of STLs and test programs that were employed to evaluate the effectiveness of the compaction strategy proposed. The first group of STLs and test programs were selected to analyze the efficiency of the proposed compaction approach on the RI5CY processor core of the Parallel Ultra Low Power (PULP) platform. It is noteworthy that the suggested compaction method can be customized to suit any other CPUs.

Table 3.1 Number of faults per module in the RISCY processor (adapted from [4]).

Module in the CPU	Number of faults
Instruction Fetch	24,148
Instruction Decode	50,340
Execute	63,878
LSU	4,442
CS Registers	6,958
Frontend	11,270
Full CPU	161,036

This research work selected a RI5CY processor and its test programs and STLs in order to assess the effectiveness of the proposed compaction strategy. The RI5CY is a pipeline 4-stage RISC-V processor core that belongs to the open-source PULP platform [129]. For the purpose of this work, the processor was synthesized using the 45nm Nangate OpenCell library [130]. Table 3.1 reports the list of the main modules that compose the RI5CY processor with the number of stuck-at faults for each fault.

This work resorts to five STLs specially designed to test stuck-at faults on the RI5CY core processor. The selected STLs were developed using different approaches (i.e., random and deterministic [78]) targeting the main components of the CPU (Execute unit, Registers File, Instruction decode, Instruction fetch, and Load Store Unit). The STLs include test programs whose admissible region for compaction (ARC) reaches 100%. These test programs correspond to those specially designed to test the Execution unit (TEx) of the processor.

Table 3.2 reports the main features of the selected STLs and their TEx. More in detail, the table includes the size, the duration in clock cycles (cc), the fault coverage, and the percentage of the test program cataloged as admissible region as explained in section 3.2.1. From the table, we can notice that variation in the test program

Table 3.2 Main features of the considered test programs (adapted from [4]).

Benchmark	Size (instructions)	Admissible region (%)	Duration (cc)	FC (%)
STL1	13,845	41.75%	126,706	82.97
STL2	64,390	97.04%	374,138	85.04
STL3	91,623	82.87%	977,012	85.04
STL4	218,467	98.16%	601,966	86.59
TE1	5,780	100%	8,589	88.14
TE2	33,034	100%	79,152	91.68
TE3	60,594	100%	84,580	94.93
TE4	206,306	100%	439,954	95.84

construction significantly changes the admissible region for compaction. Thus, the compaction can be applied to 100% of the TEx. Nonetheless, when analyzing full STLs, some of the test programs are described using complex structures or deterministic algorithms that are not necessarily suitable for compaction. For example, 41.75% of the source code of STL1 can be considered a candidate for compaction, significantly limiting the compaction capabilities of the proposed method.

In the following, further details about the main characteristics of every STL are provided.

The STL1 is composed of three main test programs, each test program targeting individual modules in the CPU. The first test program implements the March algorithm specially developed to test the register files and the control and status registers of the processor [131]. The second test program targets the execution units of the processor TE1. This test program comprises instructions that target the individual test of internal units such as the ALU, Dot Product Unit, Multiplier Unit, and Divider Unit. Each instruction in TE1 was generated using an ATPG tool. Then, every ATPG-generated test pattern was transformed into one or a short sequence of instructions that perform three main steps: i) loading the test pattern into registers, ii) issuing the instruction that activates the target unit under test, and iii) propagating the obtained results to an observable point to check if the results correspond with the expected ones. The last test program consists of several deterministic routines specially developed to test processor control units, such as load/store units and decoder fetch, among others.

The STL2 comprises several test routines combined in seven independent test programs. The test programs created to test the register file and the execute unit were developed using procedures similar to those presented in STL1. One March-C

algorithm targets the test of the register files, and ATPG-generated test patterns target the execution units (TE2). Some additional test programs use nested loop-based algorithms to take advantage of the CPU's hardware loop features. Those loops use multiply and divide instructions and contribute to the increment of the fault coverage of the execute unit. Finally, a subset of test programs implements control flow instructions in combination with deterministic algorithms in order to increase the fault coverage in control units of the processor, such as fetch, decoder, and load/store units of the processor.

The STL3 corresponds to a collection of six independent test programs. This STL also resorts to the March-C algorithm to test the register file in the CPU, as introduced in STL1 and STL2. The test programs developed for testing the execution unit resort to pseudorandom instructions. In fact, a special instruction generation tool automatically generates test programs (TE3) composed of sequences of instructions targeting all operations in the execution unit of the processor. More in detail, the TE3 is built using a fixed basic block (BB) structure and size. Each BB contains three parts: i) register initialization using random data, ii) instruction selection (selecting one random instruction taken from the available ISA of the CPU). In this case, branch and control-flow instructions are discarded, and iii) fault propagation, using one store instruction to propagate any fault effect to one of the available observation points. Each instruction's source and destination registers are selected randomly, and each register can only be used once. Additionally, one test program tests faults on the hardware loop core, which incorporates multiplications and division operations. Thus, this test program also contributes to increasing the fault detection in the multiplier and the divider cores, following a similar procedure to the one presented in STL1 and STL2.

Finally, the STL4 is divided into two main independent test programs: one specially designed for the test of the execution unit (TE4) and another one composed of multiple routines designed to test the other modules of the processor. The TE4 follows a similar construction approach as TE3, but in this case, every BB has a random length that varies from 2 to 10 instructions. In STL4, the other test programs use loop-based algorithms to read test data from memory and then apply such test patterns in an interactive process. The test data stored in memory was previously generated using a pseudorandom tool. Finally, STL4 includes some additional routines, which were added manually, to test the load-store unit of the processor.

3.2.3 Experimental results

The proposed compaction approach was implemented as a tool written in Python language. This tool interacts with one commercial logic simulator and one commercial fault injector simulator, composing an environment to analyze and compact the selected STLs. Both simulators (logic and fault injector) can handle the RTL and GL description of the selected hardware models. The RI5CY processor was taken from the PULP platform, using a special testbench to simulate both the RTL and gate-level simulations.

For the experiments, the RI5CY processor was synthesized using the 45nm Nangate OpenCell library [130]. During the fault simulation experiments, around 262,000 faults were evaluated.

The simulation reports employed during the compaction process are generated as text files. The test patterns employed in the fault simulation of the target modules employ the extended Value Change Dump (VCDE) format. The compaction procedures and experiments were performed on a workstation with two AMD EPYC 7301 16-core processors running at 2.2GHz and equipped with 128 GB of RAM memory.

Table 3.3 reports the main results of the compaction approach applied to each test program (TEx), considering only faults in the execute unit of the RI5CY processor. According to the results, the proposed compaction approach can greatly reduce both the size (up to 93.9%) and the duration (up to 95.08%) for the evaluated test programs (TEx).

Table 3.3 The compaction results in the test programs for the *Execute* unit (adapted from [4]).

Test Program	Compaction					
	Size		Duration		Diff FC (%)	Compaction time (min)
	(instr)	(%)	(cc)	(%)		
TE1	3,864	-33.15	5,860	-31.77	-0.07	5.32
TE2	5,806	-82.42	10,915	-86.21	-0.40	7.05
TE3	4,999	-91.75	7,299	-91.37	-0.11	10.52
TE4	12,581	-93.90	21,660	-95.08	-0.06	15.35

In principle, the observed compaction effects are directly related to the description style of the test programs and the capacity of each instruction to propagate any possible fault effect to one of the available observation points. In fact, the style of description defines the granularity of the basic blocks (few to hundreds of instructions). A deep analysis of the test programs revealed that all of them (TEx)

use a few instructions per basic block without control-flow instructions (the basic blocks in TE1 and TE2 are in the range of 6 to 8 instructions. TE3 has a size from 13 to 60, and TE4 has basic blocks with 3 to 6 instructions), so allowing fine grain compaction by evaluating and possibly removing each basic block.

TE1 has the shortest size among the analyzed test programs. Interestingly, the compaction strategy was able to remove 1,916 ineffective instructions. Furthermore, the size of TE2, TE3, and TE4 was reduced by 82%, 91%, and 93%, respectively. Although TE1 and TE2 were created by resorting to an ATPG, the compaction technique demonstrates a good capacity for removing a significant number of unnecessary instructions included during the parsing step that transforms test patterns into instructions.

A deep analysis of the larger test programs (TE2, TE3, and TE4) revealed that the initial set of basic blocks contains the test patterns (or instructions) able to detect a significant amount of faults, so the removal of those basic blocks is not possible, given the characteristics of the compaction method. In contrast, the basic blocks in the middle and at the end of the test programs contain redundant test patterns with less fault detection capabilities. This test pattern redundancy also favors the removal of useless instructions, enabling the test program compaction both in test duration and size, as observed in Table 3.3. Finally, the compaction also contributes to reducing the memory footprint of each test program by an identical percentage.

According to the results, there is a proportional relation between the percentage of reduced size and compacted duration of each test program. In fact, the high percentage of fine-grain basic blocks in all test programs (100.0%) allows compaction with a similar percentage of reduction for both (size and duration). The proposed compaction method can reduce up to 95.05% of the duration for the longest analyzed test program TE4. As explained above, several instructions produce redundant test patterns, which are weak in detecting faults from the execution unit. Then, those "redundant and weak" instructions are removed with minimal effect on the fault coverage. This can be observed in the minimal fault coverage difference (Diff) between the baseline test program and its compacted version reported in Table 3.3).

The results demonstrate that the proposed compaction method provides outstanding compaction capabilities for all of the test programs analyzed. In fact, there was a significant reduction in size and duration. It is worth noting that the test program compaction approach maintains a fault coverage with similar detection capabilities

as the original test program versions. Moreover, the results show that the fault coverage of each test program is reduced by a small percentage (from 0.01% to 0.4%). Although the fault coverage reductions were minimal, it is important to mention that the compaction method eventually can remove basic blocks involved in data dependencies with other consecutive basic blocks. This behavior appears when a basic block requires input data from a previous basic block eliminated from the test program since the compaction approach labeled it as not essential, so the missing basic block does not perform the required operations that are fundamental for the "essential" basic blocks. A deep analysis of the test programs showed that this fault detection degradation due to basic block elimination was less frequent than 1% of all basic blocks.

On the other hand, additional evaluations were performed considering the complete STLs. First, the STLs were analyzed by considering only the admissible region for compaction (ARC) to assess the proposed method's compaction capabilities. More in detail, the admissible region of every STL was extracted and reduced according to the strategy presented in this chapter in total: two test programs for STL1, thirteen for STL2 and STL3, and five for STL4.

Table 3.4 The compaction process results of each STL in the *admissible* region (adapted from [4]).

Benchmark	**Compaction of the admissible region of each STL**					
	Size		**Duration**		**Diff FC (%)**	**Compaction Time**
	(instr)	**(%)**	**(cc)**	**(%)**		**(hours)**
STL1	3,864	-33.15	5,860	-31.77	-0.07	0.28
STL2	33,706	-46.06	182,456	-32.95	+0.08	7.16
STL3	18,939	-75.06	76,745	-82.82	-0.11	8.04
STL4	16,263	-92.42	25,436	-84.95	-0.29	4.18

The results reported in Table 3.4 show that the compaction of the ARC only shows minimal impact on the fault coverage (<0.29%) when considering all sets of test programs per STL. Interestingly, in STL2, the fault coverage slightly increased by 0.08%; this happens due to the data dependencies between a removed basic block and the one that is still essential for testing a given unit. As explained before, the elimination of a basic block creates either an essential or redundant test pattern that, in this case, contributes to detecting additional faults. On the other hand, the compaction approach reduced the size of the admissible region of the STLs in the

range from 33.15% to 92.42%, and the duration was reduced in the range of 31.77% to 84.95%.

According to the results in the admissible region, the percentage of the reduction in size and duration is slightly lower with respect to the reduction of only TEx. This difference is caused by a bigger granularity in the basic blocks in other test programs targeting other units in the processor, e.g., decoder or controller units. In such cases, larger basic blocks with at least one essential instruction must be preserved in the test program; in consequence, the reduction is limited by such granularity in some of the STLS. For example, the STL2 contains several test programs composed of basic blocks that include between 60 to 135 instructions, making it difficult to reduce the number of instructions in a finer grain.

Table 3.5 Results showing the compaction impact on the entire STL (adapted from [4]).

Benchmark	Compaction STLs					
	Size		Duration		Diff FC (%)	Compaction Time
	instr	(%)	(cc)	(%)		(hours)
STL1	11,929	-13.84	123,977	-2.15	-0.07	0.28
STL2	37,513	-41.74	287,967	-23.03	+0.08	7.16
STL3	50,322	-45.08	884,973	-9.42	-0.11	8.04
STL4	24,314	-88.87	178,985	-70.27	-0.29	4.18

Finally, Table 3.5 reports the main compaction features when considering the complete STLs. The results indicate that the STL4 has an outstanding compaction rate in terms of test duration (70.27%) and size (88.87%). In contrast, the compaction method achieves a moderated size reduction for STL1, STL2, and STL3 in the range of 13% to 45%. Unfortunately, the reduction of the test duration achieved for STL1 and STL3 is low (<10%) and moderate for STL2 (23.03%). It is worth noting that these results can be explained due to the composition of the STLs, where less than 50% of STL1 and STL3 can be considered admissible regions for the compaction. This characteristic in the STLs limits the capabilities of the proposed compaction approach to provide higher levels of compaction for those STLs.

It is crucial to underline that the compaction time required by the proposed compaction strategy relies only on one fault simulation campaign per test program. In detail, the compaction procedure applied to the test programs targeting the faults in the CPU's execution units takes 5 to 12 minutes. On the other hand, the compaction time of complete STLs varies due to factors like the number of test programs, the size of every test program, and the required time to perform a complete fault simulation

campaign with all the faults in the CPU. In this work, the experiments showed that the maximum required time for the completion of a complete STL reaches 8.04 hours.

Despite the fact that every test program analyzed in the experiments contained thousands of instructions and the target processor accounts for more than a hundred thousand faults, the proposed compaction approach demonstrated outstanding compaction capabilities by resorting to only one logic and one fault simulation campaign. These characteristics significantly reduce the complexity and effort with respect to those required by other techniques presented in the state-of-the-art to provide similar compaction results. In fact, several works [127, 85, 78, 83, 82, 86] have developed techniques that can reduce the size and the test duration of both STLs and test programs using procedures that require as many fault simulations as the number of instructions in the test programs. In the end, for those methods, the required compaction time is proportional to the number of fault simulations, usually in the order of hundreds or thousands of fault simulations. Thus, the results obtained in the context of CPUs make the proposed compaction approach an excellent candidate for compact, more complex STLs in the GPU domain.

3.3 Compaction of STLs for GPUs

3.3.1 Proposed metodology

The main concept of the compaction strategies applied to STLs for CPUs can also be adapted to STLs and test programs developed for GPU testing. In this case, additional aspects related to the particular characteristics of parallelism and test programs must be considered. In the first place, the proposed compaction method assumes the availability of a Self-Test Library (STL) for a particular module or the entire GPU. Such STLs are composed of Parallel Test Programs (PTPs) generated by different methods, even using high-level programming languages such as CUDA C++ or OpenCL, as introduced previously in chapter 2. It is worth noting that the STLs or test programs for the compaction must be available at the assembly level using the machine ISA (i.e., Shader ASSembly for NVIDIA GPUs). In the end, every PTP is composed of a given number of GPU's instructions and targets a given

fault model with a required execution time (clock cycles or ccs) and a fault coverage per target module in the GPU.

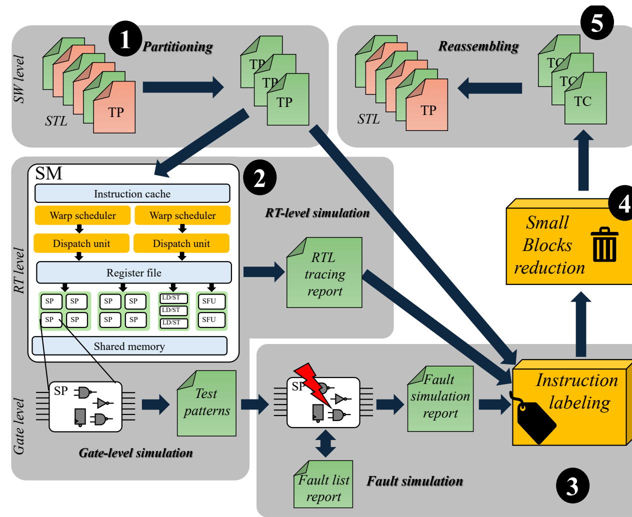


Fig. 3.2 A general scheme of the proposed compaction approach for functional TPs in GPUs (adapted from [5]).

The compaction method applied to STLs for GPUs follows an identical procedure to the one developed for the compaction of STLs for CPUs introduced in the previous subsection. Figure 3.2 illustrates the compaction flow of STL for GPUs, which comprises five main stages: ❶ STL partitioning, ❷ Logic tracing, ❸ Fault detection analysis and instruction labeling, ❹ Test program reduction, and ❺ STL reassembling.

The program partitioning stage (❶ in Figure 3.2) analyzes all test programs in the STLs and selects those ones that are candidates for the compaction. This analysis consists of the identification of the portions of the test programs called Admissible Regions for Compaction (ARCs). The identification of these ARCs follows three steps. The first step defines and finds the Basic Blocks (BBs) of each parallel test program. One BB for a GPU program is a group of instructions or embarrassingly parallel plain sequences of SIMD or SIMT instructions [24] that are always executed in sequence (no in/out jumps or branches allowed). The second step analyzes the control flow graph of the test program and labels BBs as ARC except those BBs involved in parametric loops whose iterative parameter is calculated by any BB inside or outside the loop. Once the ARCs are identified and chosen, the third step of the first stage of the compaction method extracts these regions from the test

programs. In contrast, other regions of the test programs are discarded as candidates for compaction and remain unaffected during the compaction process.

The logic tracing stage (❷ in Figure 3.2) generates two logic simulation reports (one RTL and one GL) containing information about the GPU status during the test program execution, targeting specific cores inside the Streaming Multiprocessor. These reports are crucial during the following stages in order to identify the relationship between each instruction in the BBs and its effects in terms of fault detection per warp.

The RTL logic simulation generates fine-grain information per clock cycle (cc) about the HW-SW interaction of the test programs inside the GPU. One hardware monitor is incorporated inside the SM of the GPU, which does not affect the functional operation of the test program. This monitor captures the instruction opcodes coming from the fetch stage and traces the execution of the instructions in the GPU, generating a report for the hardware module under analysis. The tracing report contains the following information for each clock cycle: the decoded instruction (i.e., Nemonic), the program counter value, the executed instruction per warp, the warp identifier, and the clock cycle value.

On the other hand, the GL logic simulation extracts the sequence of test patterns per clock cycle during the test program's execution. More in detail, those test patterns (binary values) are implicitly generated by each instruction on the specific module under test in the GPU. The sequence of test patterns is extracted by observing the I/O switching activity in the target module under analysis. In the end, one test pattern report is generated and used in the subsequent stage.

The third stage (❸ in Figure 3.2) comprises two steps: i) the fault simulation and ii) the instruction labeling. The first step carry out an optimized GL fault simulation in order to analyze the fault detection effectiveness of each instruction in the target module. The proposed fault simulation strategy isolates the target module under evaluation instead of fault-simulating the complete GPU. This approach significantly reduces the unmanageable fault simulation effort required when dealing with complex designs.

This optimized simulation strategy takes advantage of the fact that test patterns unable to propagate fault effects to the outputs of a module are also unable to propagate these effects to the output of the complete GPU or a selected observation point of a test program (i.e., the memory bus system in a GPU). Thus, the fault observability

resorts to the outputs of the module (module-level fault observability [132]). The optimized fault simulation uses the test patterns report (generated in the previous step) as input. Moreover, one fault is detected when there is a discrepancy in the execution between the fault-free and the faulty versions of the module since the selected observability point allows the trace of each propagated fault per cc.

In addition, a fault-dropping mechanism can increase the compaction rate when more than one test program is available to test the same GPU hardware module. This additional compaction optimization requires a fault list report that can support the compaction process by resorting to the following steps. First, the fault list report initially includes all faults of a target module. Then, after each fault simulation (one per test program), the fault list is updated, and those detected faults are removed from the report, so subsequent fault simulations and test programs applied to the same module of the GPU only target those missing undetected faults. In the end, the fault simulation generates a detailed report (Fault sim report) containing a list of patterns applied to the input's units, the number of activated faults per pattern, and the number of detected faults per pattern.

On the other hand, the *instructions labeling* marks the instructions of a test program according to the observed fault detection capabilities reported during the fault simulation step. This labeling procedure tags each instruction as "*essential*" or "*unessential*" according to the analysis of the Tracing and Fault Sim reports generated in the previous stages.

Algorithm 3 describes the detailed procedure of instruction labeling of the test programs. This instruction labeling uses the source code of the test program, the fault simulation report (FSR), and the RTL tracing report. This algorithm assumes that the parallel test program (PTP) meets the ARC definition and is composed of N instructions. Also, the tracing report must contain per clock cycle (QQ), the warp identifier (W), the decoded instruction (DI), and the program counter (PC).

The labeling procedure iterates over all instructions in the PTP. Initially, every instruction I in the PTP is labeled as *unessential*. Then, the instructions must be labeled according to their fault detection capabilities, considering the parallel execution of a GPU according to the Single Instruction Multiple Tread (SIMT) paradigm as described in chapter 1. The SIMT execution of the GPU issues one instruction of the PTP in several groups of threads called warps. Thus, for every executed j_{th} warp in a TreadBlock ($W_{j_{th}}$), an instruction matching procedure crosschecks the instruction

Algorithm 3 Pseudocode of the instruction labeling algorithm of a test program for GPUs (adapted from [5]).

Input: PTP composed of N instructions, Tracing clock cycle report QQ, Tracing program counter-report PC, tracing decoded instruction report DI, Warp identifier W, Fault sim test patterns report FSR
Output: Labeled parallel test program (LPTP)

```
 1: for each instruction I in PTP do
 2:     LPTPI:= Label I as 'unessential'
 3:     for each warp W_{j_th} in a ThreadBlock, do
 4:         CCs=matches(I, PC, DI, QQ, W_{j_th})
 5:         for each clock cycle k_th in CCs do
 6:             if FSR[QQ_{k_th}] detects faults then
 7:                 LPTPI:= Label I as 'essential'
 8:                 go to next instruction
 9:             end if
10:         end for
11:     end for
12: end for
```

I in the source code with the tracing report per warp $W_{j_{th}}$, matching the program counter PC and the decoded instruction DI. This matching procedure identifies the temporal life (start/end in ccs) of each instruction CCs within the executed warp. Thus, there should be a test pattern correspondence between the k_{th} clock cycle $QQ_{k_{th}}$ and a given test pattern from the fault simulation report $FSR[QQ_{k_{th}}]$. Therefore, the instruction I gets the "*essential*" label when its execution and its associated test pattern $FSR[QQ_{k_{th}}]$ detect faults in at least one of the executed warps. Otherwise, the instruction I keeps the "*unessential*," label, becoming a candidate instruction to be removed in the next stage of the compaction method.

Algorithm 4 Pseudocode of the reduction algorithm to remove SBs from a labeled test program (adapted from [5]).

Input: : Labeled Test program LPTP with N instructions divided into M consecutive BBs, and each BB is segmented in SBs (SB1, SB2, SB3, . . ., SBm)
Output: Compacted Parallel Test Program CPTP

```
1: for each SB in LPTP do
2:     for each instruction I in SB do
3:         if the label of I is 'essential,' then
4:             append SB to CPTP
5:         end if
6:     end for
7: end for
```

The fourth stage test program reduction (❹ in Figure 3.2) processes and reduces the labeled test program obtained from the previous stage. Algorithm 4 illustrates the reduction procedure employed to remove nonessential instructions from the test program. This procedure follows the same philosophy adopted from the test program reduction developed for CPUs. Firstly, the labeled test programs (LPTP) are segmented into basic blocks as defined in the first stage. Each basis block is then further divided into Small Blocks (SBs), each composed of a sequence of instructions that load test operands into the registers, execute a given operation, and propagate the result to an observable point. Then, each SB is analyzed instruction by instruction. When all instructions inside one SB are labeled as "*unessential*", the SB is removed from the test program since there are no instructions that contribute to increasing the fault coverage of the test. On the other hand, the SBs containing at least one "*essential*" instruction stay untouched for the final Compacted test program (CPTP). It is worth noting that removing an SB may also imply the additional removal and relocation of associated input data from the main memory, which depends on the parallel kernel parameters and the location of the SB within the PTP.

Finally, the reassembling step (❺ in Figure 3.2) replaces the original test programs with the STLs using the final compacted test program version. In this stage, a final fault simulation evaluates the quality of the compaction, considering aspects like the fault coverage, the test duration in clock cycles, and the memory footprint. Indeed, comparing the fault coverage of the test programs before and after the compaction facilitates the decision of whether the compacted version of the original test program fulfills the minimum testing capabilities. If the fault coverage is significantly reduced (e.g., by more than 5%), there are two paths that can be adopted: discard the compacted program and keep the original one, or apply a further compaction procedure targeting smaller parts of the test program.

3.3.2 Study case

The proposed compaction approach was assessed and verified using one available STL for GPUs [65]. The selected STLs are designed to detect faults on several units of the FlexGripPlus GPU model. The STL comprises several parallel test programs targeting diverse units inside the GPU, such as control units, memory modules, and functional units. In the STL, the test programs devoted to testing the Decoder Unit (DU) and the parallel functional units occupy around 90.69% (157,113 instructions

out of 173,241) of the program size and 75.70% (12 million out of 16 million ccs) of the test duration in the whole STL. Thus, any compaction in those test programs represents a noteworthy reduction in the size and duration of the overall STL.

Additionally, the programming structure employed to develop those test programs fits the Admissible Region for Compaction (ARC) definition explained in section 3.2.1. It is worth noticing that 47.60% of faults of the overall GPU belong to the DU and the parallel functional units (or SPs). Then, the test programs targeting these units in the GPU are good candidates to apply the compaction methodology devised in this work because they contain the most considerable size and duration of the whole STL and target a significant number of faults of the GPU. The other set of test programs is excluded from the compaction since they have been developed carefully to test control units, and any instruction removal breaks the devised test algorithm. For example, the algorithms used to test the warp scheduler or the divergency stack memory resort to a sequence of "if" or control flow statements specially designed to induce divergencies during the GPU execution (i.e., syncTrick [73]). Thus, any instruction removal in the algorithm will corrupt the proper execution of the test program or affect the fault detection capabilities.

Table 3.6 Main features of the evaluated parallel Test Programs (PTPs) for the GPU (adapted from [5]).

Target Module	PTP	Size (instructions)	ARC (%)	Duration (ccs)	FC (%)
Decoder Unit	IMM	32,736	100.0	2,229,225	71.13
	MEM	32,581	100.0	3,186,236	76.59
	CNTRL	336	90.0	710,100	71.18
	IMM+MEM+CNTRL	65,653	99.0	6,125,561	80.15
SP	TPGEN	19,604	100.0	1,447,620	84.07
	RAND	55,000	100.0	3,434,235	83.99
	TPGEN+RAND	74,604	100.0	4,881,855	87.22
SFU	SFU_IMM	16,856	100.0	1,200,034	90.75

In this regard, three different sets of test programs are selected as candidates for applying the compaction of STLs in GPU testing contexts. Table 3.6 reports the main features of every test program considered for the compaction of STLs for GPUs. The first column reports the type of unit that the test programs are targeting. The second column reports the name of every test program. The third column reports the size of the test program in terms of the number of instructions. The fourth column provides information about the percentage of the test program considered as ARC. It must be noted that more than 90% of the test programs are suitable for compaction. Finally, the last two columns report the test duration of every test program in clock cycles

and the fault coverage. In the following, a detailed description of every test program is provided.

The first set of test programs comprises three main test programs specially designed to detect faults in the Decoder Unit. These test programs are called IMM, MEM, and CNTRL. The IMM test program targets the execution of all instruction formats using at least one immediate operand. This test program also includes the register-based instructions for the GPU. Similarly, the MEM test program is composed of instructions that perform memory accesses (global memory and shared memory). Finally, The CNTRL test program uses immediate-based instructions, memory-addressing instructions, and register-based instructions to generate special conditions to be used by the control flow instructions. The IMM and MEM test programs are configured for parallel operation as one block and 32 threads per block. On the other hand, the CNTRL test program is configured as one block and 1024 threads per block. The test program generation resorts to a pseudorandom approach using all instruction formats of the supported assembly language (Streaming ASSembler language or SASS) of the FlexGripPlus GPU. The individual execution of IMM, MEM, and CNTRL allows the detection of 71.73%, 76.59%, and 71.18% of the faults in the DU unit, respectively. When them all (IMM+MEM+CNTRL) are combined, the fault coverage reaches 80.15%.

The second set of test programs targets the Scalar Processors of the GPU. The first test program (called TPGEN) resorts to ATPG-generated patterns that are later implemented in a sequence of operations to activate every functional unit of the SP. More in detail, a parser tool converts the ATPG test patterns into valid instructions for the GPU. It is worth noting that some test patterns are converted partially since not all of them can be mapped into valid instructions or operations inside the GPU. TPGEN loads from memory to the registers of each of the test operands per thread, and then a SIMT instruction issues the target operation in the SPs. Immediately, a routine updates a signature per thread (SpT). At the end of the test program execution, the SpT is stored in memory, where a final checking process allows the identification of any malfunction when comparing the obtained signature with the baseline. As some of the ATPG-generated patterns couldn't be implemented, a second test program (called RAND) increased the detection capabilities of any fault on the SP core. RAND is a pseudorandom-generated test program that combines all computational instructions, and data types to activate all the functional units in the SPs. This test program also included a SpT mechanism similar to the one developed for TPGEN.

The fault coverage of individual execution of TPGEN and RAND guarantees a fault coverage of around 84% that, in combination with both test programs, the fault coverage increases by 87%.

Finally, the last test program (called SFU_IMM) provides fault detection capabilities for the Special Function Units of the GPU device. This test program employs an ATPG tool for the test pattern generation. Then, a parser tool converts those test patterns into GPU instructions, as described in TPGEN. The kernel configuration of each test program is one block and 32 threads per block. The fault coverage capabilities of this test program reach 90.75%, and around 4% of faults are functionally untestable due to the impossibility of converting some test patterns into valid instructions to activate such faults.

3.3.3 Experimental results

The proposed compaction approach was implemented as a tool written in Python language. This tool interacts with one commercial logic simulator and one commercial fault injector simulator, composing an environment to analyze and compact the selected STLs. Both simulators (logic and fault injector) can handle the RTL and GL description of the selected hardware models. The FlexGripPlus GPU was configured with one SM, 8 SP cores, and 2 SFUs. The GPU simulation model is compatible with the CUDA toolkit SDK 5.0 with a Compute Capability 1.0, allowing the programming of the GPU device at different levels: CUDA C++, CUDA PTX, and SASS.

For the experiments, the analyzed modules in the FlexGripPlus GPU (i.e., Decoder Unit, Scalar Processors, and Special Function Units) were synthesized using the 15nm Nangate OpenCell library [125]. During the fault simulation experiments, around 12,834, 191,616, and 180,540 faults were injected into the decode unit, the scalar processors, and special function units, respectively.

The simulation reports employed during the compaction process are generated as text files. The test patterns employed in the fault simulation of the target modules employ the extended Value Change Dump (VCDE) format. The compaction procedures and experiments were performed on a workstation with two AMD EPYC 7301 16-core processors running at 2.2GHz and equipped with 128 GB of RAM memory.

Table 3.7 The compaction results in the test programs for the *Decoder* unit (adapted from [5]).

PTP	Compaction					
	Size		Duration		Diff FC (%)	Compaction time (hours)
	(instr)	(%)	(ccs)	(%)		
IMM	884	-97.30	92,423	-95.85	+0.06	2.28
MEM	442	-98.64	50,144	-98.42	-1.79	2.62
CNTRL	89	-73.51	447,689	-36.95	-0.00	0.91
IMM+MEM+CNTRL	1,415	-97.84	590,256	-90.36	-0.05	5.81

Tables 3.7 and 3.8 report the results after applying the proposed compaction approach to the selected parallel test programs for the GPU. In both tables, the second column reports the final size for the compacted version of each test program. Moreover, the third column reports the compaction percentage obtained for each compacted test program in comparison with their original length. Similarly, the fourth and fifth columns show the test duration results after the compaction of the test programs. Finally, the sixth column provides the difference in the fault coverage between the fault detection of the baseline test program and the compacted one. The last column reports the required time to perform the compaction.

The results reported in Table 3.7 show that the compaction approach can considerably reduce the size of the test programs by up to 90.36% and their duration by up to 97.84% for the evaluated set of test programs targeting the Decoder Units.

An analysis of each test program of the Decoder Units shows that IMM, MEM, and CNTRL are composed of a regular structure of Basic Blocks, composed of a few instructions (15 to 18). Interestingly, the IMM and MEM test programs have similar size and test duration features; nonetheless, MEM got around 2% higher compaction rate than IMM. This behavior can be explained by the fault-dropping mechanism that removes the faults detected by the first set of instructions or test programs. In this regard, the compaction order of the test programs rules the compaction of the test programs, so the first set of test programs will detect a significant amount of faults. This also means that multiple basic blocks in the first test program are fundamental for fault detection. Thus, subsequent test programs only have to keep the basic blocks that are able to detect faults that the previous test programs cannot. In other words, those faults already detected by IMM are dropped from the fault list, such that during the compaction of MEM, only new faults are considered. Therefore, the proposed compaction strategy keeps only those essential instructions that can produce suitable patterns and yield additional fault detection in the decode unit.

In contrast, the compaction results for the CNTRL test program were slightly lower than for IMM and MEM. This moderate compaction difference among test programs is mainly caused by the limited number of instructions (around 300) available in the original test program, combined with an inadmissible region for compaction that contains conditional and control-flow instructions that cannot be reduced.

Regarding the fault coverage results, the compacted test programs targeting the DU (IMM+MEM+CNTRL) have a negligible impact on their capabilities to detect faults with a small reduction of 0.05% with respect to the fault coverage of the original test programs. It is worth noting that the fault coverage sometimes can be improved. For example, the compacted version of the IMM test program showed an increment of +0.06% of fault coverage. Those fault coverage results respond to the effects of removing a sequence of Small Blocks (SBs), which, due to the data dependencies broken, may produce still favorable test sequences, leaving a fault coverage increment or an adverse sequence that can decrease the fault coverage capabilities of the compacted test program.

Table 3.8 The compaction results in the test programs for the *Scalar Processors* and *Special Function Units* (adapted from [5]).

PTP	**Compaction**					
	Size		**Duration**		**Diff FC (%)**	**Compaction time (hours)**
	(instr)	**(%)**	**(ccs)**	**(%)**		
TPGEN	4,742	-75.81	452,401	-68.75	-1.31	0.28
RAND	1,215	-97.79	112,030	-96.74	-17.07	1.12
TPGEN+RAND	5,957	-92.02	564,431	-88.44	-3.13	1.40
SFU IMM	9,910	-41.20	662,524	-44.79	0.0	0.31

The results in Table 3.8 show the main features of the compacted test programs developed for testing the SP cores of the GPU and the SFU of the GPU. The results show that the compacted test programs exhibit outstanding reduction sizes of up to 97.79%. likewise, the test duration of those compacted test programs achieved up to 96.74%.

It is important to underline that the proposed compaction method successfully reduced (the size and duration) of test programs generated by parsing test patterns ATP-generated into instructions. Indeed, the compacted version of TPGEN achieves 75.81% less amount of instructions and 68.75% less test duration than the original

version of TPGEN. Similarly, the compacted version of SFU_IMM exhibited a 41.20% reduction in a number of instructions, which also implied a test duration reduction of 44.79% regarding the original version. These compaction results in the case of the SFU_IMM were obtained by applying the test patterns in reverse order during the fault simulation of stage 3 in the proposed compaction approach. The overall compaction result considering the whole STL for the GPU provides 80.71% size reduction and 64.43% test duration reduction rates.

A stand-alone analysis of the compacted version of the RAND test program indicates that the fault coverage drops by around 17.07%. The reason for this figure is due to the fault dropping performed during the previous compaction of the TPGEN test program. This means that several instructions in the RAND test program detect some faults that TPGEN also detects; therefore, these instructions are redundant and can be removed during the compaction of the RAND test program. When evaluating the complete set of compacted test programs for the SP cores of the GPU (TPGEN + RAND), the results indicate that the fault coverage drops by only 3.13%. This fault coverage difference is caused by changes in the computation of the signature-per-thread (SpT) due to the removal of some basic blocks. In fact, the SpT is also computed by the SP cores, which apply an MISR-like algorithm and take each test operation's result. This SpT procedure contributes to detecting additional faults in the SPs that the main test patterns were not able to detect during the generation of the test program. Therefore, due to the compaction, the result value of one missing SB changes the SpT calculation for subsequent SBs, restricting the ability to detect the faults that were previously detected. Despite these circumstances, it is worth noting that the compaction strategy is still able to achieve significant fault detection capabilities for the test programs for the SP cores after the compaction. Furthermore, the missing detected faults can be covered by an additional refinement resorting to the generation of new Random-based test programs that, once compacted, can only contain the fundamental instructions able to increase the fault coverage.

In contrast, the results show that the compacted version of the SFU_IMM test program does not show fault coverage dropping. In this case, the SpT does not have any impact on the fault detection capabilities of the test program since the SpT is calculated on the SP cores side, so the test patterns applied to the SFU are not affected after removing some small blocks from the test program. Clearly, this happens because the SFU unit only performs transcendent floating-point operations. Consequently, removing a non-essential small block from the SFU_IMM does not

affect the fault detection capabilities of the other small blocks since there is no data dependence among small blocks. It is worth noting that these excellent compaction results respond to the regular structure of the SFU_IMM test programs, where the test patterns are embedded in the code as immediate values, facilitating the removal of blocks of code with no impact on the test functionalities

The compaction time required to transform the original test programs into reduced versions varies from some minutes to hours, depending on the size of the test programs and the simulation complexities. In the case of the test programs that test the Decoder Unit, the compaction CNTRL, IMM, and MEM required 0.91, 2.18, and 2.62 hours, respectively. In contrast, the required time to apply the compaction method to test programs that test the SP cores and the Special Function Units TPGEN, RAND, and SFU_IMM reached only 1.71 hours in total. This demonstrates the effectiveness of the test compaction method that quickly takes an input test program, generating a compacted version with quite similar fault detection capabilities to the original test program. Also, the results demonstrate that the method effectively can be used to perform test compaction of STLs developed for complex systems like GPUs.

Finally, it is important to underline that the proposed compaction method only requires a fraction of the time to analyze and process the compaction on test programs of an STL for GPUs in comparison with state-of-the-art proposals. In the experiments performed, the test programs included thousands of instructions with durations in the order of millions of clock cycles. Moreover, the evaluated GPU's modules are significantly large, containing more than a hundred thousand faults per hardware unit, which is a similar amount of faults to the complete RI5CY core. These complexity characteristics of GPUs and their test programs were successfully addressed by the proposed test compaction approach, making it suitable for both CPUs and GPUs. As presented in the previous subsection, several works in the literature [82–85] addressed the compaction for test programs in CPUs contexts only, using techniques that require as many fault simulations as the number of instructions in the test programs. None of such works provides compaction methods or results resorting to test compaction of a test program in the GPU's contexts. This work, for the first time, proposes an effective test compaction method able to significantly reduce the size of the test programs and the test duration of test programs and STLs developed for CPUs and GPUs. Moreover, the results demonstrated that the compaction method

requires from some minutes to a couple of hours for the compaction of complete STLs.

3.4 Final remarks

This chapter introduced a time-efficient compaction method, which aims to reduce the size and duration of self-test libraries (STLs) used for in-field testing of CPUs and GPUs. The devised method resorts to an instruction removal approach by selecting a group of instructions (basic blocks) that do not contribute to fault detection in the test program. More in detail, the compaction approach uses a multi-level strategy that analyses the software and hardware interaction at the instruction level to identify essential instructions based on their fault detection capabilities. It must be noticed that the compaction strategy performs only one logic and fault simulation per test program under compaction, speeding the compaction procedure several orders of magnitude in comparison with state-of-the-art approaches that resort to heuristic or evolutionary approaches.

The experiments demonstrate that the method applied to several test programs and STLs developed for CPUs and GPUs achieves an outstanding compaction ratio of about 90% in terms of size and test duration. These results indicate that the proposed method is a suitable solution for performing test compression of STLs and test programs developed using pseudo-random approaches or, even better, for improving those test programs developed using high-level programming languages.

Chapter 4

Modeling and evaluating error effects due to permanent faults on GPUs

As the technology scales, modern semiconductor technologies become more prone to faults. Such hardware defects eventually induce errors at the software level that silently propagate throughout the application execution and potentially cause system failures. These silent data errors can affect GPUs as many other hardware accelerators, treating the reliability of the overall application [43]. In this regard, it is crucial to assess the reliability of any GPU application regarding errors induced by permanent faults in the underlying hardware. These fault evaluations are crucial to designing effective software-based hardening solutions at the application level. Also, it is crucial to count on tools that allow us to assess the effectiveness of any software-based hardening solution against permanent faults affecting GPU devices.

This chapter introduces a combined fault injection strategy that enables a detailed and efficient evaluation of the effects of permanent faults in GPUs executing parallel workloads. The proposed evaluation strategy evaluates the effects of permanent faults in the functional units (i.e., SP cores of the GPU) and in the Parallel Management Units (PMUs) (i.e., Warp Scheduler (WSC), Instruction Fetch, and Instruction Decoder). To track the effect of a permanent fault, the proposed method combines the fault injection in two different abstraction levels (hardware and software). It is worth clarifying that this proposed evaluation method was previously published by the author of this thesis in [7, 6].

For the purpose of the experiments done to assess the effectiveness and limitations of the proposed approach, the low-level microarchitectural hardware effect of a permanent fault is determined using an open-source model of the hardware of an NVIDIA GPU (FlexGripPlus [81]). The impact of the permanent faults on the execution of each machine instruction is evaluated and classified, identifying software error models to propagate in a real GPU. Then, the observed effects targeting specific hardware units are used to generate instruction-level errors.

At the software level, the errors are injected and propagated in a real GPU device. This approach reduces the prohibitively high execution times of low-level microarchitectural hardware simulations for complex algorithms (e.g., DNN workloads) by several orders of magnitude while still allowing an accurate (i.e., realistic evaluations) and detailed permanent fault analysis.

It is important to mention that the software-level error propagation resorts to a dedicated software-based error injection framework (NVBitPERfi), specially designed to handle the error instrumentation of the GPU's kernels automatically. The framework is able to apply the observed SDEs from the low-level fault evaluations and to properly corrupt particular instructions in multiple threads and/or warps. The proposed frameworks, the gate-level analyses, the software-level reports, and the new NVBitPERfi tool are available in a public repository [133].

The main contributions of the work reported in this chapter are:

- A method to identify the effects of permanent faults in terms of errors at the instruction level for GPU's SP cores and PMUs;
- The development of syndrome tables as error models of the permanent fault effects on the SP cores considering the input activation and propagation operands;
- The formalization of 13 categories of instruction errors *(error models)* based on the effects of the permanent faults in the GPU's warp scheduler controller, instruction fetch, and instruction decoder unit;
- A new fault injection framework (*NVBitPERfi*) built on top of NVBit, to map the error models in software, instrument the code, and evaluate the permanent error effects in real GPU' workloads;

- A detailed evaluation of the effects of permanent faults in the SP cores and their impact on the operation of DNN workloads;
- An accurate understanding of why and how permanent faults in the GPU parallelism management units affect the execution of different workloads.

4.1 Background

GPUs have been adopted in applications (such as automotive, robotics, aerospace, and health care) in which the device life expectancy is in the order of 5 to 10 years. This life expectation is much longer than the typical 1-2 years for GPUs used in consumer applications and poses novel challenges in the GPU reliability evaluation. In fact, typical operative conditions of GPUs, such as over-stress, high temperature, high frequency of operation, and technology node shrinking, are shown to accelerate aging [37]. These technological phenomena can lead to *permanent* hardware faults in the GPUs, which may induce Silent Data Errors (SDEs) at the instruction level and potentially can produce unacceptable critical effects when the GPU is used in safety-critical domains or high-performance computing applications.

In fact, several companies like Google and Meta have raised concerns about the increasing presence of SDE effects during the operation of high-performance computing applications [42, 41, 43]. These SDEs significantly affect cutting-edge semiconductor devices like AI accelerators or GPU devices, which are crucial for a wide range of application domains nowadays. Therefore, it is crucial to assess the reliability of parallel workloads concerning SDEs produced by hardware defects.

It is worth noting that most of the available research on GPU's reliability targets *transient* faults and their effects as software errors [94, 89, 95–97, 90, 98, 91, 93, 89, 104, 72, 98], leaving *permanent* faults largely unexplored.

Unfortunately, there are very limited research results available to estimate how frequently a permanent fault affecting a GPU that runs a parallel workload may produce a critical failure (e.g., a failure that produces a misclassification of a DNN). The probability of a (permanent) fault producing a critical failure depends not only on the architecture of the underlying GPU but also on the workload behavior that can activate the fault and propagate its effects to the final result. In this regard, it is crucial to adopt effective fault evaluations that enable an effective assessment of the

impact of those faults on different kinds of parallel workloads. Furthermore, such evaluations contribute to the assessment of the effectiveness of any software-based fault tolerance solution against permanent faults and their SDEs effects.

An exhaustive gate-level fault simulation is impractical for this purpose due to the unacceptably high computational requirements. In fact, the parallel nature of GPU architecture, the huge number of possible faults due to the number of gates (millions of them for a GPU core), and the complexity of the software workload implementation (e.g., the evaluation of a small DNN like LeNet5 can take more than 10,000 hours using RTL fault simulations) makes the permanent fault effect evaluation extremely challenging. Therefore, a novel, accurate, and efficient approach is required. In this regard, this chapter illustrates a multi-level strategy that combines the speed of high-level software fault injections with the detailed analysis supported by gate-level fault simulation.

4.1.1 Related works

Several works performed extensive analyses of possible sources of permanent faults in processor-based systems [141, 142]. Other studies wisely focused on identifying error models at higher abstraction levels to simplify the analysis [134]. Table 4.1 reports different fault evaluation strategies reported in the literature striving for the reliability evaluation of hardware faults on different digital systems. These methodologies wander from fine-grain approaches, using fault simulations or emulation platforms, to high-level abstraction approaches using software-level corruptions.

In [113], the authors emphasize the importance of fine-grain low-level and cross-layer resilience evaluations, highlighting the weakness of purely software error propagation approaches. In this regard, the adoption of multi-level evaluation approaches provides the best trade-off between the accuracy (i.e., more realistic fault evaluations) of the evaluation and the required time to perform such evaluations. Typically, these multi-level approaches combine microarchitectural simulation and physical emulation [135], or software fault injection [92]. The latter approach has been successfully used in CPUs.

In [115], the authors exploit context switching between RT-level and Gate-level abstractions to combine high-level fault simulation speed and accuracy in CPUs. Similarly, Other work [102] proposed a hybrid fault injector approach integrating

Table 4.1 Evaluation strategies for reliability assessment concerning hardware faults on digital systems

Fault evaluation method	Fault model P	Fault model T	Fault model O	Evaluation level	Device	Target Units	Benchmarks
Fsim vs SWFI [113]		x		• Architectural F-sim • IR simulation	CPU	ARM Mem./Regs	MiBench suite apps
SWFI [134]		x		• IR FI		CPU Mem./ALU/Regs(*)	Various benchmarks
Hybrid FI [135]	x	x	x	• RTL F-sim • FPGA FI		DP32 core	Bubble Sort Inverse Matrix
SWFI [92]		x		• Assembly-level FI • IR FI		Assembly vs. IR	SPEC CPU 2006 suite SPLASH-2 suite
Hybrid FI [115]	x	x		• Gate-level F-sim • RTL sim.		VLIW core	WCET suite Powerstone suite
Multi-Level FI [102]	x			• RTL F-sim • SW management		RISC core	6-layer CNN CIFAR10
Emulation FI [136]		x		• FPGA FI		RISCV Regs	Sum of vectors CCSDS-123 Coremark benchmark
Cross-layer evaluation Error emulation [137]	x	x		• Gate-level delay modeling • FPGA FI		OR1200 core	SPEC benchmarks
Co-simulation [114]		x		• SW simulation • RTL F-sim		OpenSPARC / flip-flops	SPLASH-2 PARSEC-2.1 Phoenix MapReduce
Emulation FI Beam experiments [138]		x		• FPGA FI • Radiation tests		LEON2 / (SRAM FPGA Mem.)	Tripe DES alg.
Multi-Level FI [139]			x	• Gate-level F-sim • RTL simulation	AES core	Internal components	-
SWFI [89, 116]		x		• Assembly-level FI	GPU	FUs	Various (***)
Physical Stress [121]			x	• Temperature stress		-	Rodinia SHOCK PARBOIL
Application-Level FI Beam experiments [140]	x	x		• App-level FI • Beam experiments		-	Custom DNN Obj. Detect.
SWFI [104]	x(-)	x		• Assembly-level FI		FUs/Regs	SpecACCEL
SWFI [106]	x(-)			• Assembly-level FI		Regs	4 CNNs
Low level error modeling SW error propagation [117]		x		• Gate-level F-sim • RTL simulation • SW error propagation		SM (**)	Rodinia
Low Level error modeling SW error propagation [118]		x		• Gate-level F-sim • RTL simulation • SW error propagation		WSC/ Pipeline Regs	4 layer CNN
This Thesis	x			• Gate-level F-sim • RTL simulation • SW error propagation	GPU	SM(**)	Rodinia CNNs

FU: Functional Units, IR: Intermediate Representation, P: Permanent, T: Transient, O: Others.
(-) Pin-level like fault injections.
(*) Instructions associated with the target hardware unit.
(**) Functional and control units inside the GPU's SM.
(***) GPU benchmarks from PARBOIL, CUDA SDK, Rodinia, Totem, and other suites.

software application and RT-level abstractions boosted with pipeline-type stages to accelerate the evaluation of DNNs in CPUs. However, these techniques can hardly used in dense and complex GPU workloads such as DNN applications.

In CPUs, the approaches to evaluate the effect of permanent faults consider architectural simulation, software fault injection [136], and microarchitectural simulation and emulation. Other studies wisely focused on identifying error models at higher abstraction levels to simplify the analysis [134]. These methods are applicable only for small and medium designs and could require the combination of different strategies when dealing with large CPU designs [143].

Some works [89, 116] proposed multi-level approaches combining high-level architectural simulation and error propagation in real GPUs with the purpose of evaluating transient fault effects [117]. Nevertheless, the adoption of this multi-level philosophy for the evaluation of permanent faults has not been fully explored in GPUs. It is worth noting that, even in CPUs, the permanent fault evaluation can be so complex that most studies limit the evaluation to memories [138].

Only a few preliminary studies evaluate the incidence of permanent fault effects in GPUs. In [121], the authors investigate the effect of permanent faults in GPUs by increasing the temperature and accelerating the aging process. Other works propose to evaluate GPU memory permanent faults by corrupting at the software level the weights of DNN applications [140].

Some permanent fault injectors have also been proposed. In [104], the authors proposed the NVBitFI framework. NVBitFI is a software-based fault injection framework that is able to assess the effects of transient faults on GPU workloads. NVBitFI also includes a basic error modeling of permanent faults for functional units on GPUs by injecting single bit-flips on every executed instance of a target instruction during the GPU workload execution. In addition, authors in [106] proposed a customized software-based error injector based on NVBitFI to evaluate the effect of permanent faults on the register files and functional units for evaluating the reliability of DNNs workloads. Unfortunately, in all the available permanent fault injectors, the proposed error models are limited only to stuck-at or bit-flip models applied to the register files of the GPU, without considering more realistic fault effect injections.

This thesis presents a two-level fault injection concept to address the evaluation of permanent fault effects in GPUs. This evaluation methodology takes inspiration from other works in the field [114, 118, 139, 117, 137, 115] that evaluate CPUs systems

or transient faults evaluations only. The proposed approach selects the software instructions mapped on a specific hardware unit and identifies the inputs that activate the fault. Then, the permanent fault effects are modeled as errors at the instruction level that are later injected and propagated in software using a real GPU. To the best of our knowledge, this is the first work proposing a multi-level framework to evaluate permanent fault effects in GPUs, considering the fault behavior in low-level circuit descriptions, propagated as errors at the software level.

4.2 Proposed multi-level evaluation methodology: main idea

The proposed multi-level fault injection approach allows us to overcome the limitation of completely low-level fault simulations using RT or gate-level fault injection while executing GPU workloads. The *low-level* evaluation exploits the accuracy of gate-level simulation to classify fault effects in terms of instruction errors. The *high-level* part employs a time-efficient software-based fault injector on real GPUs to assess the effect of permanent faults on complete applications. More in detail, the method first identifies, through software profiling, the machine instructions (and their inputs/output) that compose the kernels of the target GPU workload (e.g., the implementation of a DNN). Then, this information is used to perform gate-level micro-architectural simulations (exclusively on the target unit of the GPU performing a targeted instruction) with the purpose of evaluating the impact of permanent faults and their propagation to the output of the operation. This simulation injects a permanent fault (stuck-at-0/1) in each site of a target hardware, applying the patterns generated by the instruction previously obtained from software/hardware profiling. This procedure allows the identification of all inputs that activate a fault and the effect on the outputs of an instruction. Then, the results of the fault simulation can be used to evaluate the propagation effects induced by the injected faults in order to generate instruction-level errors that can be used later to asses complete GPU workloads. Finally, the method propagates, at the software level, the observed errors during the execution of the code in a real GPU in order to mimic the propagation of permanent faults and to asses their impact at the application level.

More in detail, the proposed method comprises five steps: ❶ software/hardware profiling, ❷ gate-level fault injection, ❸ error identification and classification, ❹ code instrumentation and instruction-level error propagation, and ❺ application evaluation and failure classification.

The following sections provide further details on applying the proposed evaluation methodology to different GPU hardware components under different scenarios. The section 4.3 provides a detailed description of the evaluation methodology used to assess the effects of permanent faults in the SP cores of the GPU while executing DNN workloads. In addition, section 4.4 presents the detailed steps to model SDEs produced by permanent faults on the Parallel Management units of the GPU (i.e., Warp Scheduler Unit, Instruction Fetch Unit, and Instruction Decoder Unit).

4.3 Evaluation of permanent faults on GPU's SP cores

4.3.1 Proposed methodology

This section describes the details of every stage of the proposed methodology to evaluate the effects of permanent faults affecting the functional units (i.e., SP cores) of a GPU when executing parallel applications, especially for DNN workload evaluation. Figure 4.1 illustrates the main stages of the proposed fault-error-failure evaluation strategy.

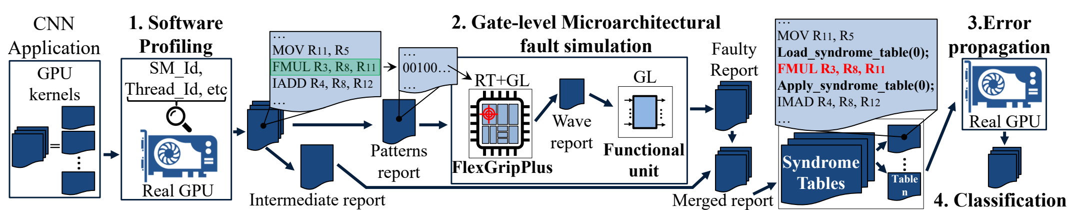


Fig. 4.1 A general scheme of the proposed multi-level method to evaluate permanent faults in GPUs. The application profiling identifies the instructions (and their inputs) that are mapped to a target hardware module. The gate-level simulation performs the permanent fault injection, identifying the inputs that activate each fault and reporting its effects on the output. Then, a software fault injection propagates fault error effects for all the instructions executed on the target unit (adapted from [7]).

4.3.1.1 Software/Hardware profiling

This step aims to collect and trace all the executed instructions from the kernels of the GPU workload (e.g., a DNN application). The main target is to use the collected information to identify the relation between the executed instructions and the SP cores of the GPU (e.g., FADD and FFMA in the Floating Point Unit (FPU), and IADD in the INT core). Moreover, the software profiling provides fine-grain and detailed information concerning the execution of each instruction. This information includes the distribution of each instruction among the GPU resources. These resources are described in terms of software parameters (*Blocks*, *Warps*, and *Threads*) and hardware parameters (*Streaming multiprocessor cores* or SMs, and CUDA cores or *Lanes*), so providing virtual *(software)* and physical *(hardware)* identification of each executed instruction in the system. Both sets of parameters (software and hardware) are later used to propagate any fault effect in the software-level fault propagation.

The software profiling step provides a complete report (*Golden profile report*) of the application executed on a real device and includes all executed instructions in the GPU. The report incorporates (for each instruction), the input operands (from the register file or memories), the software configuration parameters (*thread_ID*, *warp_ID*, and *Block_ID*), and the hardware core used (*lane_ID*, and *SM_ID*).

A data processing stage takes the Golden profile report and extracts a sub-set of executed instructions (e.g., IADD, or FFMA) as main candidates for evaluation in the gate-level fault injection campaigns. This sampled profile report (containing only the sub-set of instructions for evaluation) is divided into two reports: **1)** the *Patterns Report* and **2)** the *Intermediate Report*, as depicted in Figure 4.1. The *Patterns Report* contains only the input operands and the mnemonic of the instruction for the gate-level microarchitectural fault simulation. On the other hand, the *Intermediate Report* holds complementary information for each instruction (e.g., the *lane_ID*, and *SM_ID*), which is later used in the software-level fault propagation step to localize targets for error injection. It is worth noting that redundant input patterns from the same instruction are not included in the patterns report.

4.3.1.2 Gate-level fault simulation

The main target of the gate-level fault simulation step is the accurate study of hardware fault effects on the output of an instruction (e.g., IADD or FFMA). For

this purpose, rather than simulating a fault in a given instruction in the complete micro-architecture of a GPU, the proposed method only focuses on the gate-level microarchitectural fault simulation of the specific functional unit. Since these units are part of the data path and are directly connected to the memory hierarchy in the GPU (register file and memories), the specific gate-level fault simulation propagates any permanent fault impact from the functional units exactly as a complete micro-architectural simulation. In other words, emulating the whole GPU would just increase the simulation time while not improving accuracy.

The targeted instruction is fault-freely executed at the microarchitectural level. For this purpose, the RT-level description of the GPU model is crafted to include the gate-level description of the target GPU's SP core. This mixed description of the GPU model (RT and gate-level) provides the accuracy of the gate-level execution inside the functional unit under evaluation with a reduced simulation time, removing the details of the other units unused in the evaluation. In this step, the input operands from the patterns report are used to obtain the golden outputs of the instruction, which are then stored as a complete report ("*wave report*") and later used in the fault simulation campaigns. More in detail, the *wave report* works as a hardware unit profiling report, which contains the switching activity information of the input/output ports to the evaluated SP core. Moreover, this report is used to carry out the procedures for comparing and classifying the fault effects.

A set of gate-level fault injection campaigns is performed exclusively using the gate-level description of the GPU's SP cores, applying to the inputs the obtained *wave report* from the previous step. These campaigns provide fine-grain control of all possible faults in the unit, providing exhaustive or focused fault injection configurations (i.e., subsets of faults in flip-flops or ports).

The fault injection campaign starts with the placement of one permanent hardware fault (stuck-at) inside the GPU's SP core while computing the target operation. In addition, one group of input operands from the wave report is applied in sequence for simulation (i.e., the values of R_1 and R_2 in IADD R_3, R_1, R_2). Then, the obtained outputs are stored for later analysis. Finally, the fault simulation restarts with the placement of a new permanent fault and the injection of all groups of input operands. This procedure is repeated for all faults inside the SP core under evaluation.

4.3.1.3 Error identification and classification

A post-processing step identifies the propagated fault effects from the collected results by comparing the collected results of the fault simulation against the golden ones in the *wave report*. In case of a mismatch, the result is stored as propagated faulty results indicating an error. Otherwise, the results, free of fault effects, are discarded.

When the fault injection is completed, the identified faulty results (those containing any fault effect from a permanent fault) are merged with the information stored in the intermediate report. This *merged report* (see Figure 4.1) is useful to identify at the software level the locations and the instruction related to the propagation of a permanent hardware fault, since this report contains the identification of the permanent fault in the unit, input operands, the golden output result, the faulty result, the mnemonic of the instruction and the parallel configuration parameters.

As the first step, the merged report is divided according to the identified hardware faults during the gate-level simulations. Thus, the software-based fault injection propagates (each time) the equivalent micro-architectural fault effects in a given instruction (as an instruction error) affected by a hardware fault. The propagation (of instruction errors as an equivalent fault effect) is based on *Fault Syndromes*, which are built from the bit-wise comparison between the golden and faulty values from the gate-level fault simulation. Each *Fault Syndrome* contains the output effect of a hardware fault affecting an instruction, so the error effects are propagated across the application during the software-based fault injection in the next stage.

4.3.1.4 Instruction-level error propagation

In this step, the collected hardware fault effects are propagated as instruction errors in the software application. For this purpose, the code of the application is instrumented (off-line) with flexible functions to inject and propagate the effects of permanent hardware faults across the GPU application as errors in the instructions (fault syndrome). This error propagation was implemented in a customized binary instrumentation tool (*NVBitPERfi* [133]) built over the NVBit framework [119] to propagate the instruction-level error through the application software. To implement the syndrome error propagation at the ISA level, *NVBitPERfi* adopts the Hardware-Injection through Program Transformation (HITPT) technique [104].

Algorithm 5 outlines the proposed method that performs the instrumentation process of the GPU's source code in order to insert the corruption routines necessary to propagate error effects at the instruction level through the GPU application.

In order to mimic the permanent effect of a fault on a given functional unit of the GPU (i.e., SP core operation). The fault injection tool instruments all instances of the target instruction (e.g., FADD) propagating errors on a specific functional unit in the GPU (i.e., a combination of *SM*, *PPB* and *Lane*) resorting to *'Syndrome Tables'*, which are a collection of fault syndromes. Each input pattern applied to the evaluated functional unit can excite one permanent fault differently; thus, one fault may produce multiple error syndromes during the execution of the application.

Algorithm 5 Propagation algorithm of instruction error effects (adapted from [7]).

Input: Syndrome table for fault F_i; Faulty location in the GPU (*SM*, *PPB*, and *Lane*); Target opcode instruction OP_t
Output: Fault classification for F_i (DUE, SDC, Masked)

1: load syndrome table for F_i
2: **for** each kernel K_i in the GPU's workload **do**
3:     **for** each instruction I_j in K_i **do**
4:         Inspection(I_j)
5:         **if** I_j matches the target OP_t **then**
6:             Insert instrumentation function before I_j ▷ Load Syndrome
7:             Instruction I_j
8:             Insert instrumentation function after I_j ▷ Apply Syndrome
9:         **end if**
10:     **end for**
11:     Just-In-Time compilation
12:     Launch the GPU execution of the instrumented kernel
13: **end for**
14: Assign failure classification category to F_i

Thus, one syndromes' table is available per injected fault at the gate level during the execution of the GPU's application in order to support the code instrumentation. The selection and propagation of a syndrome (from the Syndrome Table) follows three main steps during the run-time application on the GPU: *i)* retrieval of the fault syndrome (*Load_syndrome*), *ii)* execution of the instruction, *iii)* propagation of the error syndrome (*Apply_syndrome*), as shown in Algorithm 5. First, in the retrieval step, a matching procedure searches the specific instruction and identifies, from the syndrome table, the feasible syndrome representing the fault effect. Two parameters (inputs and instruction type) are used to search inside the syndrome table. Then,

during the propagation step, the output value of the instruction is processed with the fault syndrome to produce the error effect. Finally, the error is injected by replacing the original output value with the affected one, and the application resumes.

It is worth noting that propagating permanent fault effects as instruction errors is a challenging task, since the effect of a permanent fault must remain active across the application, but only for those instructions executed on an affected hardware unit (i.e., the combination of a given *SM*, *PPB*, and *Lane*). For this purpose, the syndrome tables are used as a highly flexible mechanism to inject error effects from the gate-level results, i.e., acting as a database mechanism. These tables contain all possible fault effects from an instruction, so it is possible to use the two conditional functions (*load_syndrome* and *apply_syndrome*) to inject and evaluate fault effects (on the GPU's structures) every time the target instruction is identified, as illustrated in Figure 4.2. Moreover, the mechanism of tables of syndromes can be used for any number of instructions, and it is independent of the application.

```
1       ...
2       /** Load Syndrome pattern*/
3       M^1 ⇐ SyndromeTable[Rsx,Rsy,Rsz][Lane,W_x]
4       /**Target SASS instruction*/
5       IMAD Rd, Rsx, Rsy, Rsz
6       /** Apply Syndrome Pattern*/
7       Rd[Lane,W_x] ← Rd[Lane,W_x] ⊕ M^1
8       ...
```

Fig. 4.2 Description of Syndrome Table implementation.

The first function, called *Load Syndrome Pattern*, retrieves the content of the operand values present in the registers $[Rsx, Rsy, Rsz]$ before the instruction is executed. The retrieved values are then used to look up the syndrome table for the corruption mask $SyndromeTable[Rsx, Rsy, Rsz]$. If the syndrome table does not contain the operand values, it means that such values do not activate at the gate level of the fault; therefore, there is nothing to propagate at the application level. The obtained corruption mask is then stored in memory (global or shared), denoted as M. The second function, called *Apply Syndrome Pattern*, retrieves the corruption masks stored previously in M and applies an XOR operation $\oplus$ between the result of the executed instruction and the corruption mask. It is important to mention that this error injection procedure only affects operation in runtime-specific threads and warps

[1] M indicates a global memory location used for temporary data storage.

executed by the GPU only in the target *SM*, *PPB*, and *Lane*; denoted in Figure 4.2 as $[Lane, W_x]$.

4.3.1.5 Applications evaluation

The final step of the proposed methodology is the identification of the effects of the propagated errors in the application output. The output reports from the software-based propagation experiments are analyzed in search of critical effects in the final application.

For this purpose, the output results are classified as follows:

- **Masked:** The injected error does not produce any effect on the application's outputs;
- **Detected Unrecoverable Error (DUE):** The error propagation hangs or crashes the GPU operation, stopping the application's proper execution. Therefore, the application does not produce any valid output data;
- **Silent Data Corruption (SDC):** Error effects propagate at the application's output, inducing at least one difference with respect to the fault-free version of the application.

4.3.2 Experimental results

A DNN model was used as a study case to validate the proposed permanent fault evaluation methodology on the SP cores operations of a GPU. The selected DNN model corresponds to the six-layer LeNet5 DNN [144] built over the Darknet framework [145]. Thus, the selected DNN passes through a profiling tool on a real GPU device in order to extract the information about the executed instructions and the data operations executed by every instruction in the SP cores.

4.3.2.1 Software profiling of SP cores instructions

A preliminary software profiling of the LeNet DNN provided the total instruction count revealing that more than 65.9% of the executed instructions (11,536,325 out

of 17,505,804) use either the Floating-point unit (FP32) or the Integer core (INT). In particular, 45.88% of the total instructions use the Floating-point unit, and 20.01% use the Integer core (INT). The remaining 34.1% instructions are memory movement, control flow, and miscellaneous instruction. Then, a data processing of the profiling report extracts those instructions (65.9% of the total) that employ the FP32s (FADD, FMUL, and FFMA) and INT cores (IADD3, IMAD). For each of these instructions, an additional report contained all the input values used as operands during their execution on the GPU, as described in Section 4.3.1.1.

4.3.2.2 Gate-level micro-architecture fault simulations results

In order to handle the huge volumes of data during the fault simulation campaigns. The extracted data obtained from the profiling is further segmented per GPU kernel that the DNN executes on the GPU. Thus, a fault simulation campaign was performed for each kernel that contained at least one of the selected instructions and data operands. 141 fault campaigns out of 160 possible gate-level fault injections on the FlexGripPlus GPU model were performed on the 32 kernels of the LeNet5. For each kernel, the evaluations focused on the FPU and INT units and evaluated the instructions listed in the profiling report (FADD, FMUL, FFMA, IADD3, and IMAD). Each fault simulation campaign was also divided into up to 25 parts to speed up, through multi-threading, the fault simulations. These experiments were conducted on a server powered by an Intel Xeon CPU running at 2.5 GHz, equipped with 12 cores and 256 GB of RAM.

The fault simulations considered permanent stuck-at faults in all sites (gate-level cells and flip-flops) of the gate-level model of the FP32 and INT cores (22,044 faults in total). In addition, every fault simulation employed an average of 1,485,126 input vectors per instruction on the SP cores. That is, more than 1.54×10^{10} effects of permanent faults per input vectors are evaluated in the FP32 and INT cores per instruction.

The evaluation of the fault simulations was performed per each atomic instruction, in other words, per SP core operation (FADD, FFMA, IADD, etc.). Table 4.2 reports the SDC fault rate. This table also reports the percentage of input patterns that were able to activate and propagate the effects of permanent faults (on the primary outputs of a unit) for each evaluated operation in the SP cores of the GPU.

Table 4.2 Fault rate and percentage of input patterns exciting a permanent fault in the FP32 and INT cores (adapted from [7]).

Instruction	SDCs fault rate (%)	Input patterns exciting permanent faults (%)
FADD	9.84	7.24
FFMA	18.72	20.0
FMUL	13.82	10.42
IADD3	4.76	27.1
IMAD	8.46	10.8

This experimental evaluation shows that each evaluated instruction is affected differently by permanent hardware faults. A small percentage of permanent faults (from 9.8% to 18.7%) is propagated to the primary outputs of the FP32 unit and corrupts the result of the floating-point (FP) instructions. A surprisingly lower percentage of permanent faults in the INT core (about 4.7% to 8.4%) were activated and propagated across the unit.

The percentage of input patterns (instruction's operands from the DNN profiling) activating at least one permanent fault is small for the INT core (IADD with 27.1%, and IMAD with 10.8%). The fault rate in both instructions (IADD and IMAD) shows that input patterns recurrently activate a limited group of faults inside the core. Thus, some internal structures of the units are not activated by the input patterns. Furthermore, the percentage of patterns activating faults inside the FP32 core (FADD with 7.24%, FMUL with 10.42%, and FFMA with 20.0%) implies that each set of patterns per instruction excites different regions of the Floating-Point (FP) core. These relatively low fault rates (see Table 4.2) depend on the input patterns from the application and the operational capabilities of a functional unit.

The input patterns include two special subsets: *i)* those identical, and *ii)* the partially identical. The identical patterns share the same values of input operands and are easily removed for the evaluation. However, the second group (partially ones) includes partially shared operands (i.e., only one or two operands are identical). However, these cannot be discarded since the missing input patterns are different. Thus, this group of patterns is prone to activate an exclusive set of faults inside a unit. Finally, both hardware operations of the SP cores (FP32 and INT) are used to operate several instructions, so that all faults cannot be excited by a limited number of instructions.

A deeper analysis of the output results shows that about 90% of the fault effects induced the corruption of just one bit (single-bit flip) in the output results of the

FADD, FFMA, and FMUL instructions. In this case, the exponent bits of the result showed a higher probability of fault propagation (>25%) in comparison to those in the mantissa. Interestingly, the lower bits in the mantissa are also prone to propagating the effect of a permanent fault on the primary outputs of the unit. This is probably happening since most DNN's operands are in the range from -1.0 to 1.0. For the INT core, one bit in the output was mainly affected in most of the cases (75.88%). Interestingly, there is no clear tendency for the most commonly affected locations in the outputs. In most of the cases, any of the affected bit sites of the output is located among the least significant 23 bits of the result. This suggests that the permanent fault effects in GPUs are not trivial and should be accurately studied. These gate-level results can be employed to model error effects into higher abstraction levels, allowing a more accurate fault injection and error propagation in the applications running on GPUs.

4.3.2.3 Syndrome tables as instruction-level errors

The gate-level fault simulation campaigns are used to build a set of syndromes' tables (for each hardware fault and each evaluated instruction). These syndromes are the permanent fault effects that need to be propagated by the software-level instructions of the DNN kernels. In the experiments, all DNN kernels were evaluated on the software injection framework using the syndromes' tables.

The table of syndromes includes only the set of error syndromes generated by a specific hardware fault. Therefore, error propagation at the application level implements at the instruction level the equivalent effects of a hardware fault in the GPU's SPcore. Since each hardware fault produces different errors during the execution of the same instruction in the function of the input operand values, each table is composed of a different number of error syndromes. This error propagation approach allows evaluation of the individual impact of each hardware fault on the application execution and contributes to identifying the hardware faults that are most likely to modify the application results.

Table 4.3 reports the number of propagated faults and the number of syndromes generated per operation at the gate-level SP core. Interestingly, the number of propagated faults affecting the INT instructions is about twice the number of the FP ones. From the microarchitectural results in Table 4.2, it is important to notice that a percentage of hardware faults caused identical error effects in the outputs

Table 4.3 Main features of the error syndromes generated by permanent faults on the evaluated instructions (adapted from [7]).

Instruction	**Propagated gate-level faults**	**Error syndromes (size of syndrome tables)**	
		Min	**Max**
IMAD	1,563	4	913,752
IADD3	880	4	599,355
FADD	352	1	86,583
FMUL	494	23	15,741
FFMA	637	1	143,591

of the instructions independently of the applied input operands (0.36% for FADD, 71.65% for FMUL, 31.39% for FFMA, 45.56% for IADD, and 50.41% in IMAD). Furthermore, these identical output effects caused a few error syndromes (from 1 to 23 in Table 4.3). However, other subsets of hardware faults, produced up to 599,355 syndromes. This variation in the number of error syndromes indicates that permanent faults in functional units (INT or FP32) can produce either identical error effects or specific errors (syndromes) depending on the input operands of instruction in the evaluated DNN workload.

4.3.2.4 Error propagation results on SP cores

The software-based injection experiments have been performed on a workstation with an Intel i9-10900 CPU with 10 CPU cores, 32 GB of RAM, and one NVIDIA Ampere 3070ti GPU.

For the software error propagation experiments at the software level, the NVBit-PERfi was configured to target three different CUDA cores (SP cores) inside two representative execution cores in the GPU (SM_0 and SM_{37}). The software level evaluation was limited to these two SM cores since the GPU architecture is highly homogeneous, and an exhaustive evaluation of the error propagation in each CUDA core (4,864 cores in 38 SMs for Ampere) would be unfeasible due to the long simulation time to propagate error syndromes, especially for those faults producing a considerable amount of error syndromes. The two targeted SMs (0 and 37) were selected after several profiling trials on the DNN and the Ampere GPU. These trials show that the GPU's SM usage is unbalanced. In particular, SM_0 executes more threads than other SMs, whereas SM_{37} executes a minimal set of threads. The unbalanced use of the SMs relies on two reasons: block occupancy and block distribution,

which affect the dynamic dispatching policy in the schedulers and cause unbalanced behavior. From the SM_0 two SP cores were selected (SP_7 and SP_8 inside PPB_1), and the SP_8 inside the PPB_3 from the SM_{37}. The error propagation experiments were performed on every characterized instruction: IMAD, IADD3, FADD, FFMA, and FMUL.

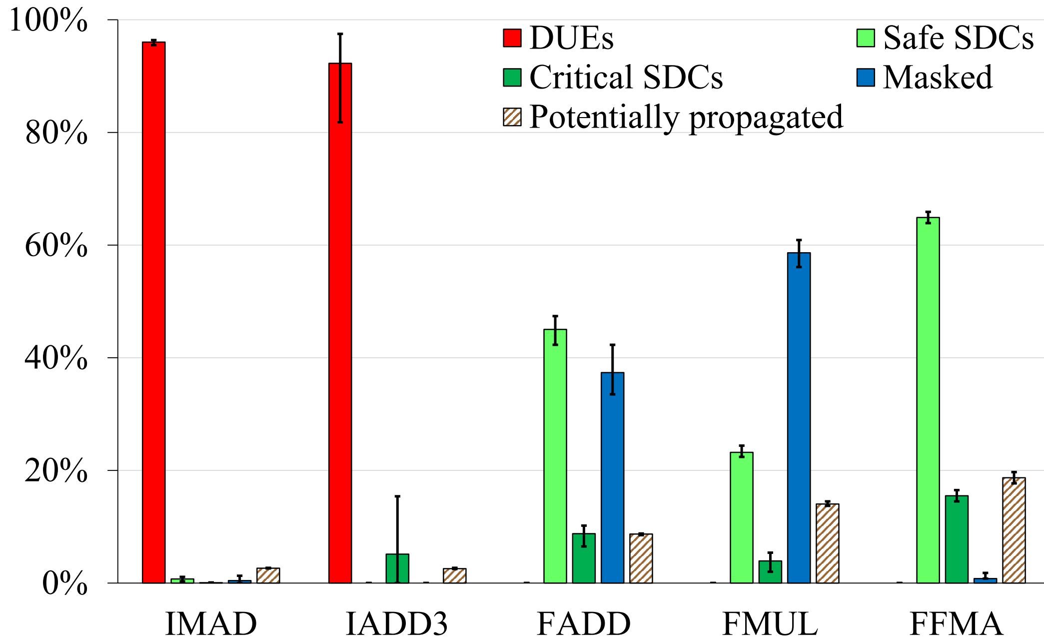


Fig. 4.3 Average results of the error propagation of permanent fault effects on the instructions of a DNN. Error bars show the maximum and the minimum changes in the error classification among the evaluated CUDA cores (caused by the operation of the scheduling controllers) (adapted from [7]).

Figure 4.3 reports the average error rate results for the evaluated instructions of the DNN in the selected SP cores according to the proposed classification 4.3.1.5. It is important to mention that Silent Data Corruption (SDCs) results can be further classified into types: *i)* safe SDCs and *ii)* critical SDCs. In the first case (*safe SDC*), the propagation effect of the injected errors causes a mismatch in the output results of the DNN. However, the error effect is not enough to produce an error in the classification outcome. In the second case, a *critical SDC* is identified when the propagation of the fault effects produces changes in the classification by the DNN.

The error bars depicted in Figure 4.3 show the maximum and minimum values obtained in the evaluated cores. Some faults are classified as *potentially propagated*, which differs from the Masked ones since, during the run-time error propagation,

the input operands do not create the right conditions to generate syndromes and propagate error effects. However, those faults in other cores of the GPU could potentially be activated and propagated through the DNN computation. The model of thread execution in the GPU elucidates this behavior since all parallel cores (CUDA cores in the SMs) work together, performing calculations related to different threads. Therefore, the thread execution model also interacts with the error propagation by distributing different threads (instructions and input operands) among the CUDA cores of an SM, which are mainly defined by a scheduling policy, so potentially missing the injection and propagation of error effects on a specific CUDA core. In any case, those *potentially propagated* errors could generate SDCs or DUE effects in the DNN when evaluated in other cores.

The results show that INT instructions are highly vulnerable to permanent fault effects and mainly collapse the operation of the application in the GPU (from 81.8% to 97.5% of faults lead to a DUE). This high sensitivity should not cause a surprise since INT instructions are mainly used in the DNN to calculate thread identifiers and memory addresses. Thus, errors in these instructions are likely to produce failures, such as memory misalignment or illegal access, which then halt the execution of the kernel in the GPU.

Most propagated faults on the FP instructions (82.4% in FADD, 81.8% in FMUL, and 65.7% in FFMA) caused minimal changes (Safe SDC or Masked faults) in the output of the DNN. However, some errors (from 2.0% to 15.5%) can jeopardize the application and change the output classification of the DNN. Since the FP instructions mainly process inputs, weights, and bias values from a DNN, most errors directly affect the DNN's classification. These experimental results demonstrate that permanent faults in an FPU are alone the main cause of Silent Data Errors (SDEs) that can produce important error effects in the classification of a DNN (more than 15%), so these faults can be more critical in comparison with other error effects, such as those produced by single bit-flip transient faults [117].

Interestingly, the dynamic dispatching policy in the scheduling controller seems to affect the error propagation results for the three evaluated CUDA cores in two SMs in the GPU (see error bars in Figure 4.3). These results indicate that the scheduling policy in the controllers can affect (or benefit) the propagation of permanent hardware effects from the SP cores to the evaluated application. The results in the most used (and the most affected by faults) core in the GPU (SM_0) provided a higher percentage

of error effects (SDCs and DUEs), in comparison to the less used cores (SM_{37}), which takes advantage of the dispatching policy and propagates a lower percentage of errors.

Table 4.4 Execution performance of the implemented method for permanent fault evaluation of DNNs (adapted from [7]).

Step	**Time (h)**
Profiling	$(0.5-2)$
Target Selection	$(2.7x10^{-3}-4)$
Gate-level Microarchitectural Simulation	$(1.5-1,180.0)$
Software-based error propagation	$(2.1-207.4)$
Classification	$(0.1-0.5)$

Finally, Table 4.4 reports the performance of the method. Two factors determine the time costs in the gate-level fault simulations (up to 1,180 hours): *i)* the number of faults (determined by the gate-level structure of the unit and the used technology library), and *ii)* the number of input patterns (directly connected with the application and the number of processed instructions). Furthermore, the software error propagation time (about 207 hours) depends on: *i)* the number of syndromes per table (generated during the gate-level evaluation and dependent on the number of input patterns), and *ii)* the number of syndrome's tables (strictly related to the number of faults activated by a given input pattern in a unit during the gate-level evaluation). Since the software error propagation step requires time to search for errors in the tables, the larger the number of syndromes per table, the longer its simulation time. Thus, the proposed method can be easily adjusted (e.g., by modifying the number of patterns per fault and the size of the syndrome table) to trade-off accuracy with computational effort, allowing for even more complex DNNs to be scaled.

Despite the long simulation times, our multi-level method of permanent fault evaluation can reduce by several orders of magnitude in comparison to fully microarchitectural approaches the time required to evaluate dense applications, such as DNNs. Our method maintains the same accuracy of the microarchitectural fault analysis at the instruction level by propagating error effects at the output of each operation.

4.4 Evaluation of permanent faults on GPU's parallel management units

4.4.1 Proposed methodology

This section describes the details of the method adopted to evaluate the impact of permanent faults on the Warp Scheduler controller, instruction fetch, and instruction decoder units in GPUs. Figure 4.4 illustrates the proposed evaluation flow.

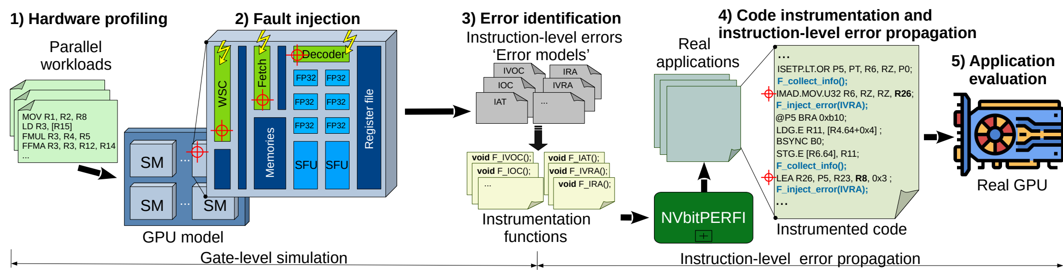


Fig. 4.4 A general scheme of the method to characterize fault effects in parallelism management units of GPUs (adapted from [6]).

4.4.1.1 Hardware unit profiling

In this step, the unit profiling resorts to the characterization of each instruction from several representative parallel workloads. In particular, every dynamic instruction is executed on the GPU, collecting the accurate golden (fault-free) operation from the targeted hardware units (*WSC*, *fetch*, and *decoder*). This stage uses at the gate-level the unit under analysis, while the rest of the GPU is simulated at the Register Transfer Level (RTL). The GPU mixed implementation (gate level for the units of interest, RTL for the rest) allows to collect and trace per-cycle information on the tested unit at the gate level and keep the interaction with the other units at RTL. This step provides the golden copy of all the unit signals, including the input patterns.

To do so, a *profiling hardware mechanism* tool was developed in order to instrument the GPU model and profile the hardware unit utilization. For each instruction, the profiling tool collects structural and operational information from the unit, including *(i)* the status of primary inputs and outputs, *(ii)* the timing information for

the instruction, *(iii)* the instruction's type, and *(iv)* the time intervals (start and end) of each performed operation.

4.4.1.2 Gate-level fault simulation

This stage characterizes permanent *stuck-at* faults on each possible fault site in a targeted unit. Exhaustive gate-level fault injection campaigns are essential to determine if a fault is activated or propagated and how it possibly manifests as an error at the output of the evaluated component. The simulation complexity would explode the case of considering all possible stimuli combinations for each fault characterization and instruction. Therefore, as mentioned in the previous stage, a selected group of instructions with different variations are extracted from representative GPU workloads in order to generate the necessary stimuli *(patterns)* produced by every individual instruction.

The effect of a permanent fault can manifest at any point in time, depending on the executed instructions (*stimuli*). Thus, it is necessary to exhaustively evaluate the individual execution of every instruction and the activation and propagation of each individual fault. To track the propagation of any possible effect on the outputs of the unit, the fault simulation results are compared with the fault-free outputs obtained in the profiling stage. Thus, all the observed effects obtained from the fault simulation are collected in combination with the hardware profiling information that later serves to identify instruction-level corruption effects per every fault. It is worth noting that the effects produced by faults in the PMUs of the GPU cannot be modeled as syndrome tables as in the SP cores case. Instead, every effect can be categorized as masking, hanging, and latent (inactive) effects, considering the effect of the fault on the GPU execution.

4.4.1.3 Error identification and classification

In this step, deep evaluation analysis of the fault simulation reports in combination with the hardware profiling allows us to identify hardware fault effects in terms of visible instruction errors (e.g., change of operand or incorrect addressing to memory), making it feasible their modeling at the software level. This stage generates a list of possible instruction errors caused by the injected permanent faults. The error models from the low-level fault injections are classified according to the

corrupted functionality and effects on the software's visible state of every instruction. For example, a fault may corrupt the opcode field in the fetch stage, changing the mnemonic of the instruction from IMAD to FMAD. Thus, the error models represent the mapping of hardware faults as instruction-level errors. Given the parallel architecture of GPUs, a fault might corrupt the instruction execution in one or multiple threads, and in one or multiple warps.

4.4.1.4 Instruction-level error propagation

Once the permanent hardware fault effect has been characterized as a visible software error effect, the next stage consists of propagating such errors through representative applications using fast software-based error injection in real GPUs. This error implementation requires a software *error function* for each permanent error model obtained in the error identification and classification experiments (step 3). These error functions are inserted in the application's SASS using instrumentation code to implement the permanent error effect during the application's execution, mimicking in software the equivalent effect of a permanent fault in the GPU unit. This instrumentation procedure extends the capabilities of the *NVBitPERfi* tool that performs an instruction-level instrumentation of the GPU's source code in order to inject the error effects produced during the fault simulation campaigns at the gate-level descriptions. The instrumentation process follows the same philosophy exposed by the Algorihtm 5 in the section 4.3.1.4. Moreover, NVBitPERfi mimics the behavior of a permanent error during the application's execution considering: *i)* the error model specifications, *ii)* the GPU architecture details, and *iii)* the instrumentation mechanisms offered by NVBit [119].

The incorporation of the PMU errors in NVBitPERfi requires handling two main challenges. (1) The corruption of a target hardware unit can impact one or multiple threads in one or multiple warps. (2) Since this work considers permanent faults, each instruction mapped to the corrupted hardware unit must be corrupted every time it is executed. Thus, we need to identify all the instructions activating the fault. It is crucial to have detailed hardware error specifications to identify how many threads/warps need to be affected by the permanent hardware fault. An error, for instance, may disable/enable one or multiple threads by interchanging their execution with a set of threads from the same warp or different warps (e.g., $< Thread_0, Warp_0 >$

issues the $< Thread_{17}, Warp_8 >$, and this produces the skipping execution of the $< Thread_0\ Warp_0 >$).

For the identification of instructions mapped to the corrupted hardware, NVBit-PERfi considers the GPU's architectural details that denote the parallelism specifications, such as the maximum number of resident warps/threads per Streaming Multiprocessor (SM) and the number of sub-partitions that every SM contains. Using these architectural functionalities, NVBitPERfi defines an error descriptor that links the physical defects of hardware units under analysis and the portions of the parallel application where the error will take effect. Consequently, the PMU error implementation in NVBitPERfi considers the following fields: *i)* the SM identifier number, *ii)* the sub-partition identifier (PPB), *iii)* the set of warps associated with the sub-partition, *iv)* the target threads inside the selected warps, and *v)* additional parameters related to the specific error model, such as targeted operands, opcodes, error bit-masks, etc.

4.4.1.5 Applications evaluation

Once the permanent fault effect has been characterized as software error models and the procedures to corrupt thread(s)/warps(s) in a real GPU have been implemented, we can effectively evaluate the impact of permanent faults on real workloads by using the *NVBitPERfi* framework implementing the HITPT technique. This methodology reduces the simulation times by several orders of magnitude compared to the classical logic simulation approach. For example, an entire fault injection campaign for all the error models using our methodology for the GEMM code can be performed in less than 24h, while using only low-level fault injections, the same campaign would take 60,000 hours (i.e., more than 6 *years*). Thus, the evaluation of any GPU workload regarding PMU instruction error models can be easily performed using the NVBitPERfi tool, and the effects of such error outcomes are characterized in one of the three classes (i.e., Masked, SDC, or DUE), as described previously in section 4.3.1.5.

4.4.2 Fault characterization results

This section presents the results of the gate-level permanent fault injection experiments and error characterization performed for the SP cores and parallelism manage-

ment units of GPUs using the RTL description of the freely available FlexGripPlus GPU model. FlexGripPlus was configured with one PPB per SM cluster and 32 SP cores per PPB. The gate-level implementations of the evaluated GPU units are obtained using a 15nms Open Cell Library [125].

Table 4.5 Tested units area and utilization percentage w.r.t. a FP32 functional unit (adapted from [6]).

Unit	Area (nm^2)	FP32 core (%)	Utilization (%)
WSC	11,854.4	114.3	100.0
Decoder	760.8	7.3	100.0
Fetch	708.2	6.8	100.0
FP32 unit	10,367.8	100.0	~(10.0 - 40.0)

Table 4.5 reports the percentage of area occupied by each PPU unit, compared to one FP32 functional unit core, and their utilization percentage, taken from profiling several workloads (described below). Despite the relatively low area of the fetch and decoder units, these units are of paramount importance in the execution of instructions since they are continuously stimulated by every instruction (while the FP32 unit is stressed, on average, only by 10% to 40% of instructions), thus accelerating aging. Despite the relatively small area of the evaluated units, their continuous operation and their failure criticality motivate their evaluation against permanent faults.

The low-level evaluation starts with the golden unit hardware profiling of the *WSC*, *fetch*, and *decoder*. In this case, we identify the signals of interest and the golden (fault-free) unit outputs. We use *all* the dynamic instructions (more than 25,200 in the real code) from 14 representative parallel workloads from Rodinia and NVIDIA SDK benchmarks (*Sort*, *Vector_Add*, *FFT*, *Tiled Matrix Multiplication*, *Naïve Matrix Multiplication*, *Reduction*, *Gray_Filter*, *Sobel*, *Scalar Vector Multiply*, *Nn*, *Scan_3D*, *Transpose*, *Euler_3D*, and *Back Propagation*).

Then, the fault evaluation resorts to 42 localized fault injection campaigns (one per benchmark for each of the three units) on an industrial-grade logic simulator (*ZOIX* by *Synopsis*) to evaluate the execution of every individual dynamic instruction from the workloads (i.e., the equivalent *exciting pattern* activating a unit) and identify the fault propagation effects. This procedure evaluates 708,808 permanent faults (i.e., the whole stuck-at-fault list) from the WSC (426,092), fetch (130,480), and decoder (152,236) units, respectively. The hardware profiling and the fault injection campaigns are performed on a server machine, which includes 12 Intel Xeon CPUs

running at 2.5 GHz and with 256 GB of RAM. It is worth noting that extensive multi-threading schemes (from 10 up to 40 parallel processes) are used to speed up the fault evaluation campaigns.

Table 4.6 Percentage of faults that are uncontrollable, masked, cause hangs or instruction-level errors (adapted from [6]).

Unit	Total	Uncontrol-lable	HW Masked	HW Hang	SW errors
WSC	29,850	35.9%	30.0%	3.6%	30.5%
Fetch	9,320	26.9%	24.5%	1.2%	47.4%
Decoder	10,874	26.0%	22.2%	2.5%	49.3%

Table 4.6 first reports the total number of considered stuck-at faults for each unit and classifies faults in the following categories:

- *Uncontrollable* faults (125,808), i.e., those permanent faults that are never activated or propagated by any input stimuli.
- *Hardware Masked* faults, i.e., faults that are activated by the input stimuli but whose effect never reaches the unit outputs (30.0% in the WSC, 24.5% in the fetch, and 22.2% in the decoder) in any of the executed instructions. These faults are thus innocuous and can be discarded from our analysis.
- Permanent faults that cause a *hardware hang*, so the GPU stops responding, or the unit's ports are corrupted, e.g., high-impedance. Only 1.2% to 3.5% of the faults caused a hang. A detailed analysis shows that most hang sources handle control signals (e.g., state machine control signals) or synchronization signals among the units (e.g., pipeline).
- *Software errors:* faults that reach one or more unit's outputs and can corrupt the software. These faults are highly likely, being 30.5% of injections for the *WSC*, 47.39% for the *fetch*, and 49.29% for the *decoder* unit. These faults corrupt the unit's outputs handling or selecting instruction's parameters, such as the memory type or the thread(s)/warps(s) status.

To further categorize the faults in the last category, i.e., those that produce instruction-level errors on any instruction from the real code, it is necessary to analyze the hardware profiles, the fault injection campaign results, and the structural

information of the GPU. From such analysis, it was possible to identify four main error groups (*i)* operation, *ii)* control-flow, *iii)* parallel management, and *iv)* resource management errors), which are further divided into 13 types of errors affecting any software instruction, as follows.

- **Operation errors**
  - **Incorrect Operation Code Error (IOC):** The operational code of an instruction is modified and still valid, but the executed instruction type (or its parameters) is different.
  - **Invalid Operation Code Error (IVOC):** The opcode of the instruction is modified and not valid.
  - **Incorrect Register Addressed Error (IRA):** An incorrect (yet valid) register is addressed, affecting the instruction.
  - **Invalid Register Addressed Error (IVRA):** an incorrect and not valid register is addressed (i.e., a register outside the limit of registers per thread).
  - **Incorrect Immediate Operand Error (IIO):** the immediate operand is corrupted.
- **Control-flow errors**
  - **Work-flow Violation Error (WV):** The workflow of an instruction is modified by corrupting the predicate conditions.
- **Parallel management errors**
  - **Incorrect Parallel Parameter Error (IPP):** incorrect addressing of resources shared among the warp, such as the shared memory and register files regions.
  - **Incorrect Active Thread Error (IAT):** unauthorized enable or disable of threads in a warp.
  - **Incorrect Active Warp Error (IAW):** incorrect detention, assignation, or unauthorized submission of a warp.
  - **Incorrect Active CTA Error (IAC):** incorrect detention, assignation, or unauthorized submission of a CTA (cooperative thread array) in the GPU core.

- **Resource management errors**
  - **Incorrect Active Lane Error (IAL):** unauthorized enable or disable lanes in a GPU core.
  - **Incorrect Memory Source Error (IMS):** incorrect assignation of a memory resource for operand loading.
  - **Incorrect Memory Destination Error (IMD):** incorrect assignation of a memory resource for result's storing.

It is important to mention that the error classification proposed in this thesis differentiates errors that cause an *incorrect* operand from those that cause an *invalid* operation or action. While both types of errors modify the same instruction field (e.g., both IOC and IVOC modify the opcode), the former is likely to induce a data error since a (wrong) instruction is executed or a (wrong) memory value is read/written, while the latter blocks the execution. It is worth noting that most errors affect the thread management units and the parallelism in the GPU. Thus, most error types (IOC, IVOC, IRA, IVRA, IPP, IAW) affect all threads in a warp, while others (IIO, WV, IAT, IAC, IMS, and IMD) mainly corrupt one or a few threads per warp. The information about multiple threads/warps corruption is used in Section 4.4.3.1 to map the effects into instruction-level errors.

Figure 4.5 shows the Fault Activation and Propagation Rate (FAPR), i.e., the probability for a hardware permanent fault to be activated and to propagate to a software visible state. The figure discriminates the FAPR of permanent faults injected in each of the three units to cause one or more of the identified error types. The most common instruction error models are IVRA, IMS, and IMD in the decoder unit, IVOC in the fetch unit, and IOC for all units. On the contrary, some instruction error classes are highly unlikely to occur (e.g., from 0.48% of IAC in WSC to 7.53% of IAW in the fetch unit). The low percentage of IAC errors (i.e., wrong block scheduling) is explained by noticing that the considered WSC, the fetch, and the decoder units handle finer-grain parallel management (operation of threads and warps), instead of coarse-grain (CTAs) parallel management. Interestingly, faults in the decoder unit cause a wider spectrum of possible instruction effects (11 out of 13 error categories). This is due to the fact that the decoder directly interacts with the machine code of the instructions.

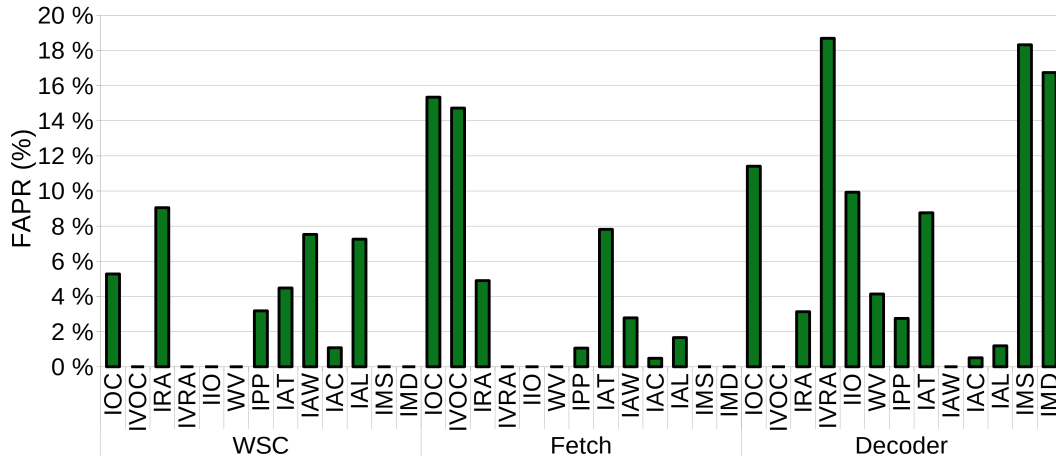


Fig. 4.5 Fault Activation and Propagation Rate (FAPR) for the identified faults as SW errors in the WSC, fetch, and decoder units. Faults are grouped by error types (adapted from [6]).

Additionally, the results allow the identification of some single permanent faults causing more than one error type. This is unsurprising since the permanent faults can be activated differently based on the applied stimuli from different executed instructions. The results show that (a) the same permanent fault may produce different types of software errors (from 1.28% to 14.9% for the WSC, about 1.98% in the fetch, and less than 0.25% in the decoder unit, depending on the executed instruction), and (b) the same permanent fault may simultaneously produce two or more types of software errors during the operation of a single instruction (less than 18.4% of faults). The intermediate reports, including the instruction opcode and input stimuli that activated the permanent fault, are used to correlate the error model to be injected at the software level (Section 4.4.3.1) with the instruction being executed. This information allows the accurate propagation of the hardware fault effects in software and enables an understanding of the probability for a permanent fault to be activated in a realistic application.

4.4.3 Software-based error propagation results

This section describes the implementation details of NVBitPERfi developed to analyze the error propagation at the software level. This section also presents and discusses the main results using several realistic GPU workloads.

4.4.3.1 Errors implementation/propagation

The permanent error models derived from the fine-grain circuit-level analysis are implemented in NVBitPERfi to mimic in detail each error according to their specifications. The proposed implementation follows a similar strategy proposed on the NVBitFI framework. This procedure was described in the section 4.3.1.4 by means of the Algorithm 5. It is worth mentioning that the error models are implemented by resorting to special corruption routines inserted in the instruction-level source code of the evaluated application. In detail, those error routines are inserted in the assembly source code of the GPU kernels during the instrumentation stage [119]. Then, the error is injected, propagated, and evaluated (at speed) once the *faulty kernel* is issued on the device.

It must be noted that some error models require only one instrumentation function right before or after the targeted assembly instruction in the application program. Nonetheless, other error models are more complex when modeled at this level since they require modifying an operand before the actual instruction execution and then restoring its content after the execution. In the end, these models are implemented with two instrumentation functions, plus a global memory storage mechanism to keep temporary data that allows communication between both functions during the runtime error propagation.

The software-level implementation details for each error model are described below. The descriptions are grouped based on their technical similarities and highlight their peculiarities.

IRA and IVRA

Incorrect/Invalid Register Addressed Error models select a wrong register address in one of the operands fields for all instructions issued by the GPU. IRA selects an address that points to a valid wrong register (i.e., within the maximum number of registers per thread). IVRA selects registers outside of these boundaries as one of the operands. The implementation of IRA and IVRA uses two different approaches; one is used when the corrupted register address represents the source operand, and the other when the destination address is the one corrupted. The error descriptor for IRA and IVRA includes the parameters introduced in the section 4.4.1.4 (instruction, thread(s), warp(s) affected) plus additional parameters: *bitErrMask*, and *errOperloc*. The *bitErrMask* is the bit level mask that modifies the target operand register number,

and *errOperLoc* is the operand position inside the instruction (0 means destination operand Rd, and 1, 2, or 3 one of the source operands $R_{<sx,sy,sz>}$).

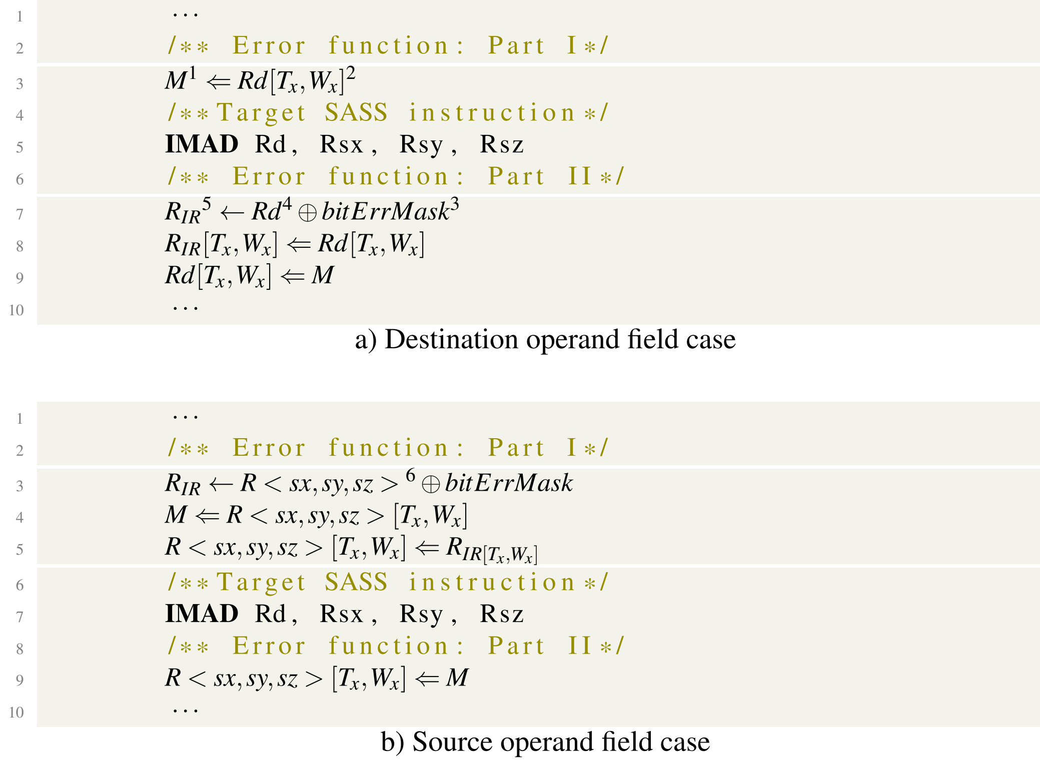


Fig. 4.6 Description of IRA/IVRA error models (adapted from [6]).

Fig. 4.6 shows the implementation of the two operation modes of IRA/IVRA. The first mode refers to the error that targets the destination operands, thus the error function stores the content of the destination register Rd into M before the instruction is executed. Then, after launching the target instruction, the second instrumentation function copies into the target error register (R_{IR}) the result of the operation stored in Rd; then, the Rd content is restored.

In the case of an error affecting the source operands, a function (issued before the instruction's execution) uses a memory location M to store the content of the source

---

[2] The $[T_x, W_x]$ indicates the set of threads T_x on selected warps W_x where the error takes effect.
[3] $bitErrMask$ denotes the bits mask used to induce the index error to overwrite value.
[4] Rd corresponds to the destination register.
[5] R_{IR} represents the incorrect or invalid register to be accessed obtained from applying the $bitErrMask$ field to the original register number.
[6] $R < sx, sy, sz >$ denotes one of the source registers R_sx, R_sy, or R_sz.

register $R_{<sx,sy,sz>}$ before performing any data modifications. Then, the targeted register operand $R_{<sx,sy,sz>}$ takes the content of the error-accessed register I_{IR}. A second function, executed after the execution of the target instruction, restores the original source register $R_{<sx,sy,sz>}$ content.

IAT, IAW, and IAC

Incorrect Active Thread/Warp/CTA error models disable/enable or wrongly assign threads, warps, or CTA. To implement this behavior at the software level, we disable the execution of a set of threads on the selected warp(s) by replacing their identifiers with different (wrong) ones, pointing threads to the same or different warps. For example, for disabling thread0 in warp0, the index associated with the thread changes to the index of another thread (e.g., thread8 in warp0). Thus, the register that contains indexes for all threads will not contain the index of the disabled thread, producing the error effect during the execution by skipping the execution of thread0 in warp0.

```
1       ...
2       /** Target SASS instruction */
3       S2R Rd, SpecialRegisterID<x,y,z> [7]
4       /** Error function */
5       Rd[Tx,Wx] ⇐ Rd[Tx,Wx]^2 ⊕ bitErrMask[Tx,Wx]
6       ...
```

Fig. 4.7 Description of IAT/IAW/IAC error models (adapted from [6]).

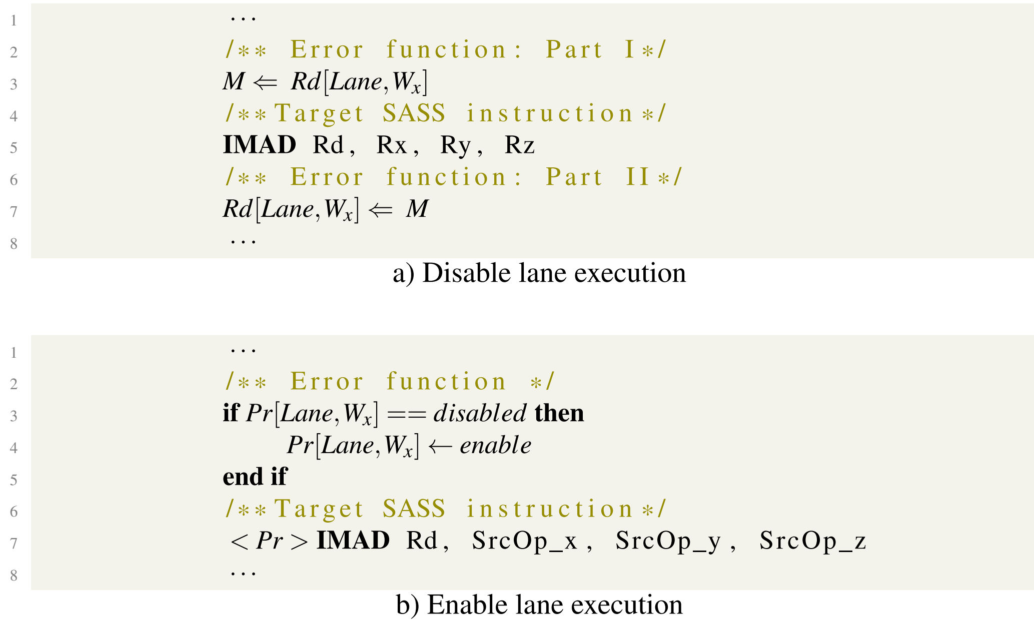
Figure 4.7 presents the modeling concept of IAT, IAW, and IAC. This procedure is applied to the desired number of threads on selected target warp(s) $[T_x, W_x]$ issued on a specific SM sub-partition. It implements one instrumentation function after the instructions that copy the content of a special register $SpecialRegisterID_{<x,y,z>}$ into a destination register Rd. In the case of IAT or IAW, the instrumentation function affects only the instructions that take the content of SR_TID for one of the x, y, or z dimensions of the parallel thread indexing of the application. The IAT (thread) error model keeps at least one thread active in the warp for its execution, whereas the IAW (warp) error model forces all indexes inside a warp to change, producing a full substitution of a particular warp for another. For IAC (CTA) error, the instrumentation function modifies the destination register Rd of the instructions

[7] $SpecialRegisterID_{<x,y,z>}$ refers to the special register SR_TID or SR_CTAID in any of the dimensions x, y or z

reading the *SR_CTAID* special register of one of the thread's three dimensions x, y, or z indexing registers. In this case, when the index of the block changes, the obtained effect leads to incorrect block thread execution.

IAL

The software-level implementation of Incorrect Active Lane error requires two different approaches. The first one, the unauthorized inactive lane (Figure. 4.8.a), ignores the result of all instructions executed on a specific functional unit in one or several lanes (e.g., Integer or floating point cores). This functionality can be achieved by replacing the result of such instructions with the content of the destination register captured before executing the instructions.



```
1  ...
2  /** Error function: Part I*/
3  M ⇐ Rd[Lane,W_x]
4  /**Target SASS instruction*/
5  IMAD Rd, Rx, Ry, Rz
6  /** Error function: Part II*/
7  Rd[Lane,W_x] ⇐ M
8  ...
```

a) Disable lane execution

```
1  ...
2  /** Error function */
3  if Pr[Lane,W_x] == disabled then
4      Pr[Lane,W_x] ← enable
5  end if
6  /**Target SASS instruction*/
7  <Pr> IMAD Rd, SrcOp_x, SrcOp_y, SrcOp_z
8  ...
```

b) Enable lane execution

Fig. 4.8 Description of IAL error models (adapted from [6]).

The second approach (Figure. 4.8.b) forces the execution of all predicated instructions associated with the Integer or Floating Point Lane where the error is injected. An instrumentation function is inserted before the target instruction to check the predicate register status. Hence, if the predicate register *disables* an instruction's execution, then the function changes its status to *enabled*, forcing the execution of an instruction that was not supposed to be executed.

IIO, IMS, IMD, WV, and IOC

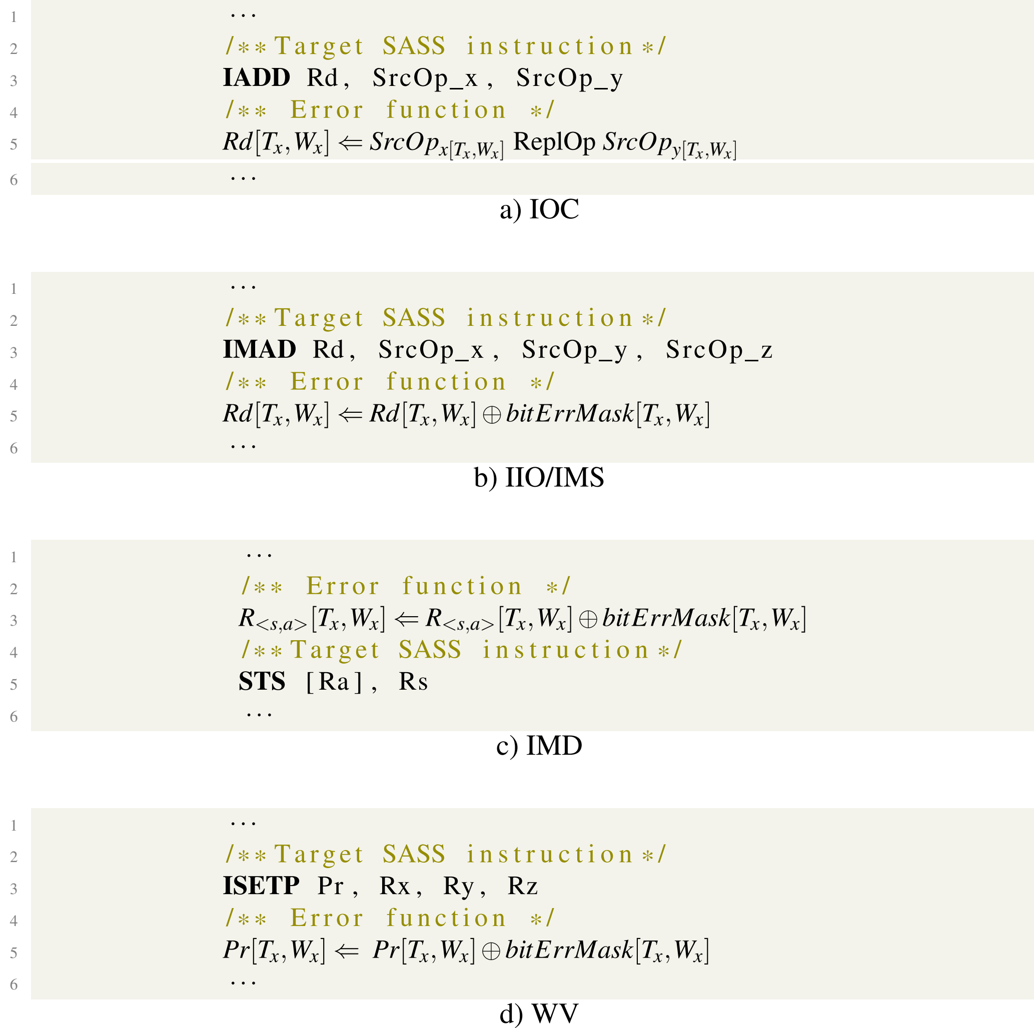


Fig. 4.9 Description of IOC/IIO/IMS/IMD/WV error models (adapted from [6]).

All these errors modify a field in the executed instruction(s) ISA. These errors can be implemented by modifying the destination register of a selected group of instructions either with a random value or with a different operation, using the same instruction operands (see Figure. 4.9). The instruction's group subject of the instrumentation and/or error injection is determined by the error type. Incorrect Immediate Operand (IIO) applies an error mask in the destination register for all instructions containing at least one reference to immediate operands. Incorrect Memory Source (IMS) inserts an error mask in all instructions containing at least one operand reference to constant or shared memory. Work-flow Violation (WV) selects and inserts an error mask in all instructions that write to a selected predicate

register, affecting the application's control flow. Incorrect Memory Destination (IMD) targets all the instructions with shared memory as a destination reference by inserting an error bitErrMask either into the data register to be stored or in the register that addresses the shared memory. Finally, Incorrect Operation Code (IOC) targets all instructions issued by the integer or floating point cores by taking the input operands and replacing them with any other operation.

IPP and IVOC

Incorrect Parallel Parameter (IPP) error has several ways of affecting the GPU operation, but most of them lay into two main categories *i)* the wrong resource addressing hardware resources (i.e., registers or shared memory modeled by IRA, IVRA, IMS, and IMD), and *ii)* by generating an incorrect threat execution modeled by IAT or IAW. On the contrary, an Invalid Operation Code (IVOC) represents an invalid opcode operation that generates an invalid instruction exception at the software level, leading to a Device Unrecoverable Error (DUE) in all cases where the error is injected.

4.4.3.2 Error injection and propagation results

This section presents the results of injecting and propagating instruction-level errors using NVBitPERfi on several real GPU workloads. 15 realistic workloads were selected (listed in Table 4.7) to evaluate the error models for the PMU implemented on NVBitPERfi. In order to demonstrate that the proposed evaluation methodology can be applied to any application, the selected workloads correspond to various domains, including Deep Learning, Linear algebra, N-body simulation, and Graphs.

The software-based injection experiments have been performed on a workstation with an Intel i9-10900 CPU with 10 Cores, 32 GB of RAM, and one NVIDIA Ampere 3070ti GPU.

This section presents the evaluation results of error injection and propagation on 15 real applications. The experiments were performed by injecting 1,000 errors per application per error model. In addition, the NVBitPERfi was configured to target the error injections on one sub-partition (PPB_0) of SM0. Overall, the complete evaluation resorted to more than 165,000 errors that took 300 hours of real GPU simulation for all the applications.

Table 4.7 Codes used for the software-level error injections (adapted from [6]).

	Data type	Domain	Suite
vectoradd	FP32	Linear algebra	CUDA SDK
lava	FP32	N-body	Rodinia
mxm	FP32	Linear algebra	CUDA SDK
gemm	FP32	Linear algebra	CUDA SDK
hotspot	FP32	Structured Grid	Rodinia
gaussian	FP32	Linear algebra	Rodinia
bfs	INT32	Graphs	Rodinia
lud	FP32	Linear algebra	Rodinia
accl	INT32	Graphs	NUPAR
nw	INT32	Dyn. Programming	Rodinia
cfd	FP32	Unstructured Grid	Rodinia
quicksort	INT32	Sorting	CUDA SDK
mergesort	INT32	Sorting	CUDA SDK
lenet	FP32	Deep Learning	Darknet
yolov3	FP32	Deep Learning	Darknet

Figure 4.10 reports, for the 15 applications, the *Error Propagation Rate* (EPR), i.e., the probability for an error (produced by a fault that was activated and corrupted one or more of the unit outputs) to propagate to the software output. The figure shows the EPR for SDC, DUEs, and Masked outcomes. It must be noticed that the results in the figure are grouped per error model, as discussed in Section 4.4.1.3. Figure 4.10 reports the results for 11 error models grouped by the four main error groups (i.e., *Operation Errors*, *Control-flow Errors*, *Parallel Management Errors*, and *Resource Management Errors*).

The error models IPP and IVOC are not reported since IPP can be implemented by any of the other error representations (IRA, IVRA, IAT, IAW, IMS, or IMD), and IVOC always generates DUEs at the low-level injections.

An interesting result from Figure 4.10 is the very high EPR for all error models and applications (the average EPR is 84.2%). The applications that are either compute-intensive (i.e., yolov3, lava, or LeNet) or the ones that use many kernels during the execution (i.e., bfs, mergesort, and quicksort) present, for most of the error models, an EPR equal or close to 100%. It is worth noting that permanent faults, by definition, are less likely to be masked compared to transient faults, as the resources are permanently damaged.

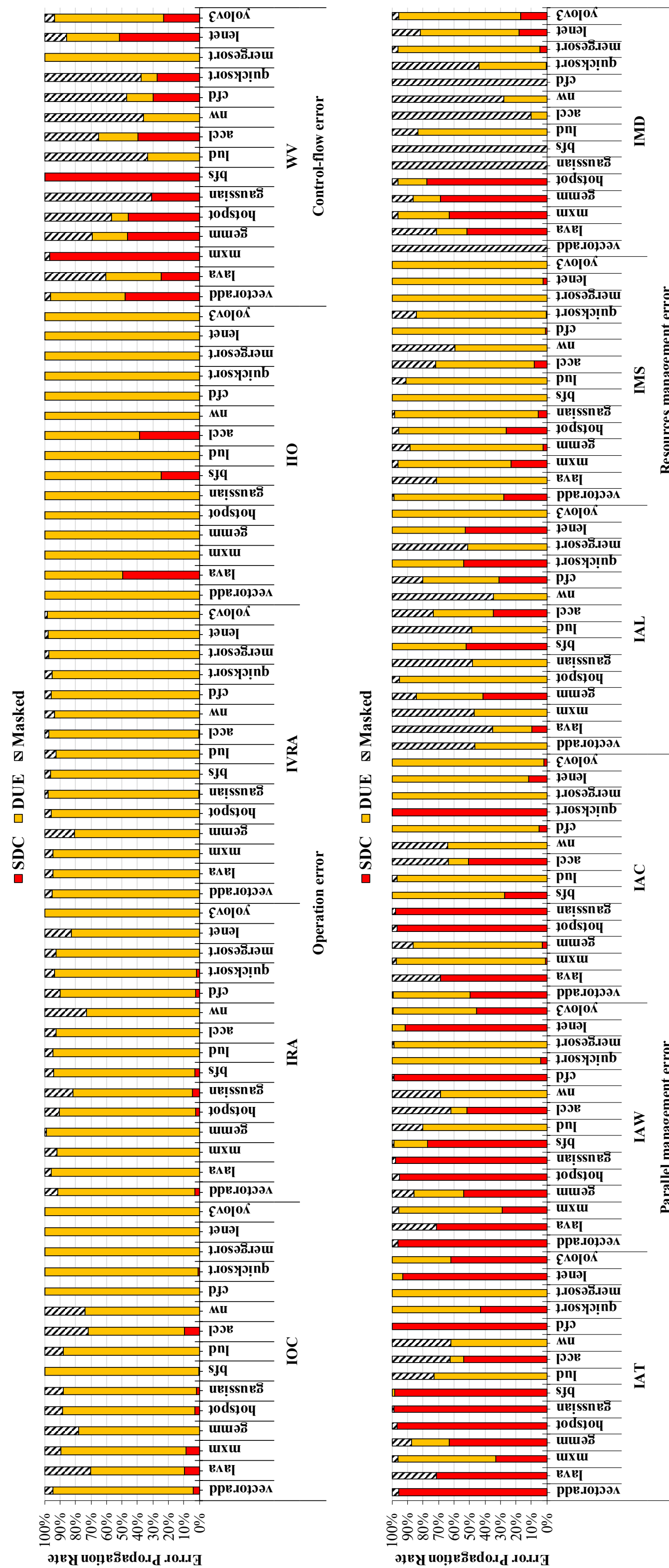


Fig. 4.10 Error Propagation Rate results of each error model propagated on 15 applications. IPP and IVOC are not shown since IPP is similar to other models, and IVOC induces only DUEs (adapted from [6]).

Moreover, the results indicate that the code's characteristics can significantly impact the EPR. This is particularly evident in two error models, WV (work-flow) and IMD (incorrect memory destination). For the WV error model, codes with many control flow blocks or thread indexing limitations that, once modified, can impact a significant amount of data (i.e., vectoradd, mxm, gemm, hotspot, bfs, and gaussian) show a high SDC EPR. Additionally, applications that can impact the memory addressing or block synchronization (i.e., lud, nw, and mergesort) show a high DUE EPR. The EPR changes similarly for the IMD error model. For many codes, the error model IMD has no impact on the execution (i.e., vectoradd, gaussian, bfs, and cfd). The IMD error model affects instructions that operate on shared memories by changing the register, which is the source or destination of an instruction that loads or stores on shared memory. Consequently, codes that do not use shared memories will have 100% of the injected faults masked.

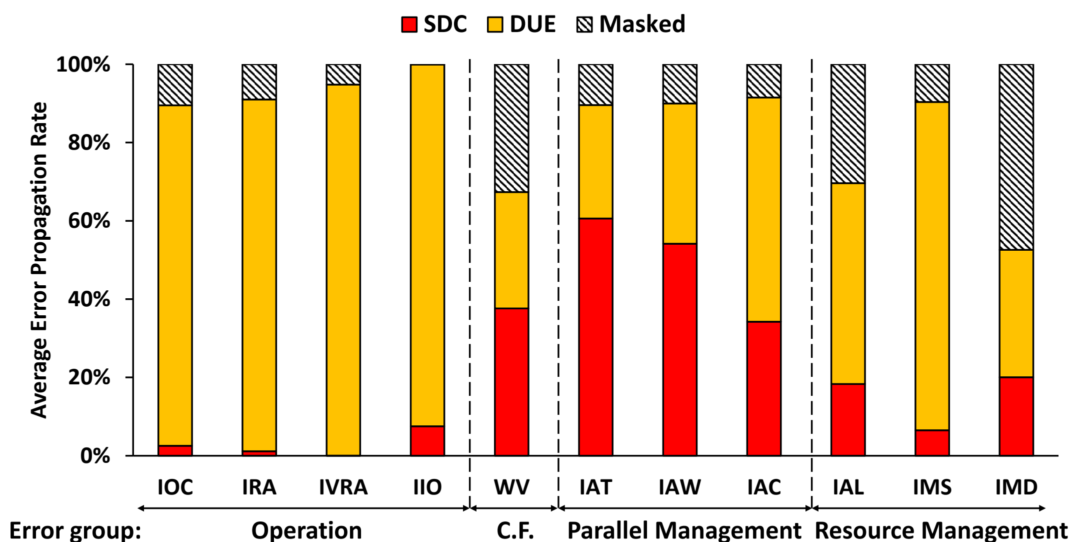


Fig. 4.11 Average EPR among the 15 tested applications (adapted from [7]).

Figure 4.11 summarizes the main findings for the 11 evaluated error models by showing the *Average* EPR between all evaluated applications. Interestingly, the group of **Operation Errors** shows a predominance of DUEs for all error models. On average, the percentage of IOC, IRA, IVRA, and IIO injections that generate a DUE is 87%, 90%, 95%, and 92%, respectively. The Operation Errors, as discussed in Section 4.4.1.4, have a particular characteristic of modifying the behavior of all or many instructions in one or all threads within a warp or multiple warps. When many instructions are modified due to a permanent fault, the expected outcome is to

have at least one thread, or many threads, performing illegal instructions, accessing incorrect memory accesses, or operating with registers outside the thread register bounds, which leads to a DUE. In fact, the percentage of incorrect memory addresses and illegal instructions generated by IOC, IRA, IVRA, and IIO error models are, on average, 99.05%, 99.76%, 100%, and 98.29% of the total DUEs.

On the contrary, most of the error models that belong to the **Control-flow** and **Parallel Management** groups (WV, IAT, and IAW) have a high SDC EPR which is, on average, 38%, 61%, and 54%, respectively. The combination of the error model and the executing code significantly changes these injections' outcomes. For example, when single or multiple threads are disabled due to the IAT error model, the output that would be expected from that thread will not be produced, generating an SDC. This is the case of codes like vectoradd, gaussian, cfd, and bfs, where the IAT error model enables/disables threads on the execution, and the code is able to finish (i.e., due to low interdependencies of the threads) but generates, most of the time, SDCs. Similar behavior is observed for IAW and WV error models, but in these cases, there is a slightly higher incidence of DUEs than for IAT. In fact, these error models affect multiple threads or warps simultaneously, which can lead to the corruption of multiple output elements.

The errors from the **Parallel Management** group mostly induce SDCs in the applications. The only error model with an average DUE EPR higher than the SDC EPR is IAC (SDC EPR is 34% and DUE EPR is 57%). This happens because IAC is an error model that causes an incorrect execution (i.e., detention, assignation, or unauthorized submission) of an entire CTA (thread block) in the kernel execution, increasing the probability of DUEs. As with the other error models from the Parallel Management group, the EPR will change according to how the code uses the GPU resources. For instance, when the IAC error model is injected, applications such as lava, hotspot, gaussian, accl, and quicksort have most of the injections leading to SDC, i.e., the SDC EPR is 69%, 97%, 98%, 51%, and 99% respectively. This happens because those applications schedule many independent parallel CTAs, then an incorrectly assigned block may still finish and produce an incorrect output.

For the error models from the **Resource Management Errors** group, the EPR shows a strong dependence between the injection outcome and the code, where, for example, in the case of IMD, the use of shared memory can determine if the error will be masked or not. Similar behavior also can be observed for the IAL and IMS

error models. However, as IAL and IMS affect resources used for all the codes (by disabling GPU lanes or causing an incorrect assignation of a memory resource for the result's storing), we can see that both error models impact all codes, increasing their average EPR.

Finally, the required time to perform the evaluations is significantly faster than simulation-based fault injection approaches and more realistic than error injections at the application levels (e.g., corrupting parameters in DNNs). In fact, the proposed evaluation methodology required only 20.5 hours for profiling, 178.1 hours for low-level characterization, 4.2 hours for error analysis, and $\approx$ 300 hours of software-level error propagation for all workloads and targeted units (502.8 h in total), so speeding up the simulation of more than four orders of magnitude in comparison with gate-level fault simulations. In addition, the flexibility of the proposed method and the developed simulation framework NVBitPERfi, allows its adaptation/extension for the evaluation of other units and the development of additional fault models. The low-level micro-architecture characterization just requires the adaptation of the hardware profiling tool, according to the specifications of the new target unit or fault model, to identify the stimuli.

4.5 Final remarks

This chapter introduced a multi-level fault evaluation approach to evaluate the impact produced by permanent faults on GPU applications. The proposed methodology resorts to modular evaluation targeting low-level fault simulations on a GPU hardware unit. These evaluations allow the characterization of the fault effects considering representative profile information of realistic workloads using software and hardware profiling information. The fault simulation results are then transformed into instruction-level errors describing the propagation effect of permanent faults on the evaluated unit. Later, the modeled errors are propagated at the software level during the execution of a realistic GPU workload to assess the application's resilience regarding such errors.

This fault evaluation approach significantly speeds up the evaluation of permanent faults in GPUs by up to four orders of magnitude compared with pure fault simulation strategies. Moreover, this fault assessment strategy provides the best balance between the fault evaluation time and the accuracy of the evaluation (i.e., closer to reality).

The multi-level fault evaluation can be scaled to a wide variety of complex application domains, including deep learning, linear algebra, N-body simulation, and graphs. It is worth noting that this methodology allows for the first time to obtain realistic results regarding the resilience of complex GPU workloads (e.g., DNNs) regarding permanent faults in GPUs rather than state-of-the-art approaches that perform fault evaluation using hardware-agnostic approaches (e.g., corrupting DNN parameters) [7, 106, 105, 107].

It is important to mention that the DNNs chosen for this thesis, such as LeNet5 and Yolov3, are relatively small. However, the proposed methodology can be easily adapted to assess larger DNN models. Initially, application profiling and gate-level fault simulations are carried out to generate syndrome masks for functional units or instruction errors in the case of parallel management units. Subsequently, the low-level characterization is utilized at the software level as needed by the reliability engineer, who can reuse them for similar workloads without resorting to low-level evaluations again.

The syndrome tables can be reused as they contain fault effects related to input operands, which are commonly found in multiple DNN workloads. For instance, floating point operations typically fall within the range of [0,1], given the typical characteristics of DNNs. Additionally, from our experimental assessments, we noticed that some syndrome tables contain corruption masks that affect the least significant bits of target operations. Removing these "benign corruption masks" significantly reduces the required time for software-level error propagation, thus speeding up the evaluations

On the other hand, the instruction-level errors, that represent fault effects in the GPU's PMU, do not significantly impact the execution of applications. This means that different applications, including larger DNN models, can be evaluated without a lot of extra work. It is important to note that NVBitPERfi offers a user-friendly framework that allows users to evaluate other applications by making minor modifications to the setup. However, performing low-level evaluations still requires some engineering effort, as it depends on the available RTL GPU model and knowledge of industrial tools for circuit-level simulations.

Chapter 5

Conclusions

This thesis addresses the reliability enhancement of GPU devices using software-based approaches from two different perspectives. First, this work tackles the effective generation and compaction of Software Test Libraries developed under the SBST concept applied to in-field test scenarios. It is important to mention that the main contributions of this aspect include *i)* the adoption of high-level (e.g., CUDA C++) or intermediate (e.g., CUDA PTX) programming languages to develop or map SBST strategies devised for testing specific hardware components in GPUs, and *ii)* the devising of alternative test program compaction strategies to reduce their memory footprint and their test duration when adopted in the in-field test.

On the other hand, this work proposes multi-level fault evaluations for effective reliability evaluations of GPU applications concerning Silent Data Errors caused by permanent faults. The proposed method provides the best trade-off between accuracy (i.e., closer to reality) and fast fault evaluations compared to other evaluation methods. In addition, these evaluations are crucial for devising effective software-based hardening solutions for GPU applications. Likewise, the proposed multi-level fault evaluation allows the assessment of software-based hardening solutions regarding their capabilities to minimize the impact of permanent faults on different GPU-based applications (e.g., DNN workloads in safety-critical systems).

In chapter 2, we introduced a method to develop STLs aimed at testing GPUs during in-field operations using High-level or Intermediate-level programming languages. The proposed method employs a divide-and-conquer approach to target individual modules in a GPU and apply test patterns or test algorithms that can

later be mapped into high-level functions. The methodology was applied by using state-of-the-art test procedures designed to test the *FlexGripPlus* GPU model for the experiments. The experimental evaluation demonstrates that HLLs or ILL programming levels can be used to detect permanent faults effectively when test programs are described in the regular structures of the GPU (e.g., functional units). In fact, these hardware structures are also visible resources to the programs providing high test-pattern controllability and fault observability. On the other hand, using HLLs for the development of test programs for more complex modules (i.e., scheduler units, divergence management units, and embedded memories) exhibits limited fault detection capabilities (from 28% to 68% of fault coverage). In fact, the test algorithms or procedures can not be fully implemented in HLLs due to language restrictions or compilation optimizations. Nonetheless, we found that the usage of ILLs provides better fault detection capabilities for these GPU cores. In addition, when combining HLL and assembly languages (e.g., SASS), the fault coverage results are equivalent to the test programs developed only using assembly languages. On the other hand, using HLLs for describing STLs for fault detection on regular units of the GPU significantly increases the program size and the test duration with respect to assembly level only, whereas using ILLs can produce equivalent program sizes than the assembly ones but slightly increasing the test duration.

In general, developing STLs for GPUs using HLLs or ILLs is a good solution when the GPU ISA specifications are restricted, or the documentation is not fully available. Nonetheless, there are constraints and challenges in the development of STLs using such programming abstractions as pattern controllability and fault observability. In addition, compiler intervention plays a crucial role in the generation of the final test program. More specifically, compiler features must be faced using strict coding styles or combining several abstraction levels to develop effective STLs. Finally, the adoption of HLLs when developing STLs for GPUs also plays an important role in finding the best trade-off between the development time, the fault coverage, the program size, and the test duration of the STL.

In chapter 3, we presented a novel approach to perform test program compaction, reducing the size and duration of STLs and functional test programs described using the SBST strategy. The proposed strategy was successfully applied to test programs developed to compact STLs and test programs developed for CPUs and GPU cores. The compaction approach addresses test programs with a regular structure of consecutive instructions or basic blocks. The proposed method greatly

reduces the required compaction time by exploiting only one fault simulation per test program. The compaction method uses different levels of abstraction by combining one RTL logic simulation and one GL fault simulation to extract, trace, and label the portions of a test program that can be removed. Then, after identifying the essential instructions per basic block, these non-essential instructions are removed from a given test program. The proposed approach was validated by using five different STLs developed to test stuck-at faults on a state-of-the-art pipelined microprocessor (*RI5CY*) and several Parallel Test programs developed to test stuck-at faults on an open-source GPU model (*FlexGripPlus*).

In the context of STLs for CPUs, the results showed that the method could reduce the size by up to 93.9% and the duration by up to 95.08% of the test programs developed to test the execution units, only. On the other hand, the method achieved up to 88.87% reduction in size and 70.27%, respectively, for STLs developed to test the entire CPU. It must be noticed that the compaction method has a negligible impact on the fault coverage (<0.4%), while we also observed cases where the fault coverage slightly increased.

On the other hand, when targeting the test programs developed for the GPU, the compaction strategy reaches a high compaction ratio: up to 98.64% in terms of size and up to 98.42% in terms of duration when compacting PTPs with regular structures, excluding those regions with parametric loops. The compaction method showed a minimal impact on the achieved fault coverage for the evaluated PTPs. The compaction of the selected PTPs implies 80.71% size and 64.43% duration compaction rates for the complete STL analyzed.

It must be highlighted that the main advantage of the proposed compaction method is the limited computational time required in comparison with heuristic or evolutionary compaction strategies. This outstanding improvement is the result of the reduced number of logic and fault simulations required to perform the compaction in comparison with the state-of-the-art methods. In detail, the proposed compaction approach only uses one RTL logic simulation to trace the behavior of a target test program and one GL fault simulation to identify helpful instructions for detecting and propagating faults.

Finally, in chapter 4, we introduced a multi-level approach that combines accurate gate-level simulations with efficient software-based error propagation to assess the resilience of GPU workloads regarding permanent faults. The purpose of this

method is to generate instruction-level errors that depict the effects of permanent faults on individual components or hardware structures of the GPU through fault simulations. These instruction-level errors are then propagated during the runtime of the application by corrupting the source code of the application through the insertion of corruption routines at the assembly level. The methodology has been adapted to assess the impact of permanent faults on the compute cores of the GPU (i.e., SP-Cores or CUDA cores), then a variation of the method has been used to study the parallel management units of the GPU (i.e., Warp Scheduler, Instruction Fetch and Instruction Decoder units). Regarding the instruction error modeling, we propose the adoption of syndrome tables based on the fault simulation of the functional units, and regarding the PPU units, we have identified from gate-level fault simulation four main groups of errors: *(i)* Operation, *ii)* Control-flow, *iii)* Parallel management, and *iv)* Resource management errors), corresponding to 13 instruction error categories. The error propagation at the software level resorts to an error injector (NVBitPERfi) developed to implement and propagate the instruction-level error effects obtained from the gate-level fault simulations. For the error propagation analysis, several realistic workloads were used by performing error injection through NVBitPERfi. The evaluation of permanent faults affecting the functional units was evaluated using a DNN workload, whereas similar evaluations on the PMUs were conducted in 15 real parallel applications in different domains, including machine learning applications.

This methodology allows for the first time to obtain realistic results regarding the resilience of DNNs regarding permanent faults on the functional units in GPUs rather than the hardware-agnostic evaluations available in the state-of-the-art. Indeed, the results indicate that faults at the hardware level corrupt a fraction of the evaluated data, inducing different syndrome errors at the instruction outputs (7.24% for FADD, 10.42% for FMUL, 20.0% for FFMA, 27.1% for IADD, and 10.8% for IMAD). When propagating the syndrome error at the application level, the results unveil that permanent faults in functional units are highly critical for DNN workloads. Such permanent faults can affect the execution of a DNN, degrading the accuracy by 15.5% when targeting floating point hardware, whereas faults on the integer hardware collapse the device's operation (i.e., crash or hang) in 97.5% of the cases.

On the other hand, the experimental evaluations regarding permanent faults on the GPU's PMU show that the permanent fault effect depends on the corrupted unit and the executed instruction. Faults in the fetch unit mainly (66.80% of the cases) lead to

Operation errors, faults in the Decoder unit lead to *operation* (44.32%) and *resource management* (38.35%) errors and faults in the scheduler lead to *parallel management* errors (54.87%). The software-level propagation of the observed error categories shows that parallel management errors (mainly generated within the Warp scheduler) generate a high amount of silent data corruption (20% to 60%), whereas faults in the Fetch and Decoder units mainly lead to DUEs ($>$ 90% and 70%, respectively). However, in the case of the Decoder, there is a non-negligible 20% of SDCs that jeopardizes the application execution.

Lastly, the fault evaluation strategy dramatically reduces the time complexity required for the evaluation of permanent faults in GPU, allowing an accurate error characterization at the gate level and a practical propagation of errors at the application level. A similar evaluation using only low-level hardware descriptions would be simply unfeasible. In fact, the simulation of a complete GPU at gate level takes, in our server, $\approx$14.5 hours for characterizing *one* permanent fault in *one* application. If we scale the simulation time for all workloads (15 applications) and all fault locations (50,044) we tested, we would reach a theoretical simulation time of around $14.5 \times 50,044 \times 15$ hours, that is 10.8×10^6 hours: $\approx$ 1,242 years! In contrast, our approach required only 20.5 hours for profiling, 178.1 hours for low-level characterization, 4.2 hours for error analysis, and $\approx$ 300 hours of software-level error propagation for all workloads and targeted units (502.8 h in total), so speeding up the simulation by more than four orders of magnitude.

In conclusion, software-based approaches can be effectively adopted to enhance the reliability of complex computational accelerators like GPUs. This thesis shows that test programs and software test libraries can be effectively developed for testing regular structures by adopting high-level or intermediate programming languages. Nonetheless, it is crucial to adopt adequate coding styles to prevent compiler optimizations and to include intrinsic functions that enable the activation of specific hardware units during the test. On the other hand, the generation of test programs, either using high-level programming languages or assembly levels, might contain instructions that do not contribute to the fault detection of faults during the execution of the test program. In that case, the compaction of test programs can be effectively accomplished by using only one RTL and one gate-level fault simulation to significantly reduce the size of test programs developed either for CPUs or GPU devices. Alternatively, the adoption of multi-level fault evaluations contributes to the development of effective software-based hardening solutions to counteract the impact

of permanent faults on GPU's workloads. Moreover, such evaluations offer the best tradeoff between the accuracy and the evaluation time required by the evaluations.

Future directions

The work presented in this thesis paves the way for the development of new investigations or extensions that can be applied to other hardware devices or accelerators. Specifically, regarding the Software-Based Self-Testing resorting to high-level languages (chapter 2), there are still several directions regarding their exploration and application in the development of test programs to test other fault models, such as transition delay fault models or cell-aware fault models, either for CPU or GPU devices. In addition, the mapping of test strategies into high-level programming languages has limitations regarding the programming language specifications as well as the compiler intervention. Therefore, it is crucial to develop guidelines that mitigate the impact of the compilation process on the effectiveness of the fault coverage of a given STL.

Regarding the compaction of test programs (chapter 3), it is crucial to extend the capabilities to the method to compact more complex programming structures used to develop test programs such as conditional or loop statements. Among the possible extensions to face such contains, it is worth exploring the preprocessing stages of the test programs, such as loop unrolling and data dependency analysis, in order to provide more flexibility and compaction capabilities to the proposed compaction approach.

Finally, when it comes to reliability evaluation techniques regarding permanent faults (chapter 4), the future directions drive toward the development of additional instruction-level errors to be included in the NVBitPERfi error evaluation tool. For example, an instruction-level error can be generated that describes the effects of permanent faults on the Tensor Core Units or the Special Function Units in GPUs in order to assess machine learning applications. Additionally, the proposed evaluation approach can be used to assess different hardening techniques developed at the application level to increase the robustness of DNN models.

References

[1] Ondrej Burkacky, Matteo Mancini, Mark Patel, Giulietta Poltronieri, and Taylor Roundtree. Exploring new regions: The greenfield opportunity in semiconductors. https://www.mckinsey.com/industries/semiconductors/our-insights/exploring-new-regions-the-greenfield-opportunity-in-semiconductors, 2024. [Online; accessed 01-April-2024].

[2] Statista. Graphics processing unit (GPU) market size worldwide in 2022 and 2032 . https://www.statista.com/statistics/1166028/gpu-market-size-worldwide/, 2023. [Online; accessed 01-April-2024].

[3] Juan-David Guerrero-Balaguera, Josie E. Rodriguez Condia, and Matteo Sonza Reorda. Stls for gpus: Using high-level language approaches. *IEEE Design & Test*, 40(4):51–60, 2023.

[4] Juan-David Guerrero-Balaguera, Josie E. Rodriguez Condia, and Matteo Sonza Reorda. A novel compaction approach for sbst test programs. In *2021 IEEE 30th Asian Test Symposium (ATS)*, pages 67–72, 2021.

[5] Juan-David Guerrero-Balaguera, Josie E. Rodriguez Condia, and Matteo Sonza Reorda. A compaction method for stls for gpu in-field test. In *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 454–459, 2022.

[6] Juan-David Guerrero-Balaguera, Josie Esteban Rodriguez Condia, Fernando Fernandes Dos Santos, Matteo Sonza Reorda, and Paolo Rech. Understanding the effects of permanent faults in gpu's parallelism management and control units. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '23, New York, NY, USA, 2023. Association for Computing Machinery.

[7] Josie E. Rodriguez Condia, Juan-David Guerrero-Balaguera, Fernando F. Dos Santos, Matteo Sonza Reorda, and Paolo Rech. A multi-level approach to evaluate the impact of gpu permanent faults on cnn's reliability. In *2022 IEEE International Test Conference (ITC)*, pages 278–287, 2022.

[8] Martin Adam, Michael Wessel, and Alexander Benlian. Ai-based chatbots in customer service and their effects on user compliance. *Electronic Markets*, 31(2):427–445, 2021.

[9] IBM. watsonx: The AI and data platform that's built for business. https://www.ibm.com/watsonx, 2024. [Online; accessed 01-April-2024].

[10] Xu Chen and Di Wu. Automatic generation of multimedia teaching materials based on generative ai: Taking tang poetry as an example. *IEEE Transactions on Learning Technologies*, pages 1–14, 2024.

[11] Suhib Y Bdoor and Mohammad Habes. Use chat gpt in media content production digital newsrooms perspective. In *Artificial Intelligence in Education: The Power and Dangers of ChatGPT in the Classroom*, pages 545–561. Springer, 2024.

[12] Samuel Greengard. Ai rewrites coding. *Commun. ACM*, 66(4):12–14, mar 2023.

[13] Brett A. Becker, Paul Denny, James Finnie-Ansley, Andrew Luxton-Reilly, James Prather, and Eddie Antonio Santos. Programming is hard - or at least it used to be: Educational opportunities and challenges of ai code generation. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, SIGCSE 2023, page 500–506, New York, NY, USA, 2023. Association for Computing Machinery.

[14] Naoki Wake, Atsushi Kanehira, Kazuhiro Sasabuchi, Jun Takamatsu, and Katsushi Ikeuchi. Gpt-4v(ision) for robotics: Multimodal task planning from human demonstration, 2023.

[15] Naoki Wake, Atsushi Kanehira, Kazuhiro Sasabuchi, Jun Takamatsu, and Katsushi Ikeuchi. Chatgpt empowered long-step robot control in various environments: A case application. *IEEE Access*, 11:95060–95078, 2023.

[16] Daniel Reed, Dennis Gannon, and Jack Dongarra. Reinventing high performance computing: Challenges and opportunities, 2022.

[17] Emma Roth. Microsoft spent hundreds of millions of dollars on a ChatGPT supercomputer. https://www.theverge.com/2023/3/13/23637675/microsoft-chatgpt-bing-millions-dollars-supercomputer-openai, 2023. [Online; accessed 01-April-2024].

[18] NVIDIA Corporation. NVIDIA DRIVE End-to-End Platform for Software-Defined Vehicles. https://www.nvidia.com/en-us/self-driving-cars/, 2024. [Online; accessed 13-March-2024].

[19] Animesh Patel. The Driving Force: GPUs' Crucial Role in Autonomous Vehicles. https://medium.com/@apcmpe09/the-driving-force-gpus-crucial-role-in-autonomous-vehicles-bdc0a303e21e, 2023. [Online; accessed 13-March-2024].

[20] Ian Hill, Parvez Chanawala, Rohit Singh, S. Arash Sheikholeslam, and André Ivanov. Cmos reliability from past to future: A survey of requirements, trends, and prediction methods. *IEEE Transactions on Device and Materials Reliability*, 22(1):1–18, 2022.

[21] P.K Tseng. TrendForce Says with Cloud Companies Initiating AI Arms Race, GPU Demand from ChatGPT Could Reach 30,000 Chips as It Readies for Commercialization. https://www.trendforce.com/presscenter/news/20230301-11584.html, 2023. [Online; accessed 01-April-2024].

[22] Zhiye Liu. ChatGPT Will Command More Than 30,000 Nvidia GPUs: Report. https://www.tomshardware.com/news/chatgpt-nvidia-30000-gpus, 2023. [Online; accessed 01-April-2024].

[23] Sergi Alcaide et al. Safety-related challenges and opportunities for gpus in the automotive domain. *IEEE Micro*, 38(6):46–55, 2018.

[24] C. Nvidia. Programming guide: Cuda toolkit documentation, 2022. Available online on 01/12/2022.

[25] Jack Choquette, Olivier Giroux, and Denis Foley. Volta: Performance and programmability. *IEEE Micro*, 38(2):42–52, 2018.

[26] John Burgess. Rtx on—the nvidia turing gpu. *IEEE Micro*, 40(2):36–44, 2020.

[27] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S. Vetter. Nvidia tensor core programmability, performance & precision. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 522–531, 2018.

[28] M. Raihan, N. Goli, and T. M. Aamodt. Modeling deep learning accelerator enabled gpus. In *IEEE Int. Symp. on Performance Analysis of Systems and Software (ISPASS)*, pages 79–92, mar 2019.

[29] Brent Ralph Boswell, Ming Y Siu, Jack H Choquette, Jonah M Alben, and Stuart Oberman. Generalized acceleration of matrix multiply accumulate operations, July 2 2019. U.S. Patent 10,338,919.

[30] J. R. Nickolls et al. Systems and methods for voting among parallel threads, 2005. US Patent No. 8,200,947B1.

[31] A. S. TIRUMALA et al. Techniques for comprehensively synchronizing execution threads, 2017. US Patent No. 10,977,037B2.

[32] lu wang, Xia Zhao, David Kaeli, Zhiying Wang, and Lieven Eeckhout. Intra-cluster coalescing and distributed-block scheduling to reduce gpu noc pressure. *IEE Trans. Comput*, 68(7), 2019.

[33] Julien Demouth. CUDA Pro Tip: Minimize the Tail Effect | NVIDIA Technical Blog — developer.nvidia.com. https://developer.nvidia.com/blog/cuda-pro-tip-minimize-the-tail-effect/. [Accessed 01-11-2023].

[34] NVIDIA Corporation. Nvidia h100 tensor core gpu architecture. https://www.nvidia.com/en-us/data-center/technologies/hopper-architecture/. [Accessed 01-04-2024].

[35] NVIDIA Corporation. Nvidia h100 cnx. https://www.techpowerup.com/gpu-specs/docs/nvidia-gh100-architecture.pdf. [Accessed 01-04-2024].

[36] Wilfred Gomes, Altug Koker, Pat Stover, Doug Ingerly, Scott Siers, Srikrishnan Venkataraman, Chris Pelto, Tejas Shah, Amreesh Rao, Frank O'Mahony, Eric Karl, Lance Cheney, Iqbal Rajwani, Hemant Jain, Ryan Cortez, Arun Chandrasekhar, Basavaraj Kanthi, and Raja Koduri. Ponte vecchio: A multi-tile 3d stacked processor for exascale computing. In *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, volume 65, pages 42–44, 2022.

[37] IEEE. The international roadmap for devices and systems: 2022. In *Institute of Electrical and Electronics Engineers (IEEE)*, 2022.

[38] Jin-Woo Han et al. Single event hard error due to terrestrial radiation. In *2021 IEEE International Reliability Physics Symposium (IRPS)*, pages 1–6, 2021.

[39] IEEE. The international roadmap for devices and systems: 2022. In *Institute of Electrical and Electronics Engineers (IEEE)*, 2022.

[40] Mark White. Scaled cmos technology reliability users guide. Technical report, NASA, 2010. https://nepp.nasa.gov/files/18209/09_102_5_JPL_White_Scaled%20CMOS%20Technology%20Reliability%20Users%20Guide%201_10.pdf Online; accessed 01-April-2024].

[41] Harish Dattatraya Dixit, Sneha Pendharkar, Matt Beadon, Chris Mason, Tejasvi Chakravarthy, Bharath Muthiah, and Sriram Sankar. Silent data corruptions at scale. *CoRR*, abs/2102.11245, 2021.

[42] David F. Bacon. Detection and prevention of silent data corruption in an exabyte-scale database system. In *The 18th IEEE Workshop on Silicon Errors in Logic – System Effects*, 2022.

[43] Rich Bonderson. Training in turmoil: Silent data corruption in systems at scale. https://research.google/pubs/training-in-turmoil-silent-data-corruption-in-systems-at-scale/, 2021. [Online; accessed 01-April-2024].

[44] David Fiala, Frank Mueller, Christian Engelmann, Rolf Riesen, Kurt Ferreira, and Ron Brightwell. Detection and correction of silent data corruption for large-scale high-performance computing. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2012.

[45] George Ostrouchov, Don Maxwell, Rizwan A. Ashraf, Christian Engelmann, Mallikarjun Shankar, and James H. Rogers. Gpu lifetimes on titan supercomputer: Survival analysis and reliability. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2020.

[46] Adit Singh, Sreejit Chakravarty, George Papadimitriou, and Dimitris Gizopoulos. Silent data errors: Sources, detection, and modeling. In *2023 IEEE 41st VLSI Test Symposium (VTS)*, pages 1–12, 2023.

[47] Paolo Rech 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)*, 2014.

[48] Oscar Ferraz et al. A survey on high-throughput non-binary ldpc decoders: Asic, fpga, and gpu architectures. *IEEE Communications Surveys & Tutorials*, 24(1):524–556, 2022.

[49] Jeremy W. Sheaffer, David P. Luebke, and Kevin Skadron. A hardware redundancy and recovery mechanism for reliable scientific computation on graphics processors. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, GH '07, page 55–64, Goslar, DEU, 2007. Eurographics Association.

[50] Josie E. Rodriguez Condia, Pierpaolo Narducci, M. Sonza Reorda, and L. Sterpone. A dynamic hardware redundancy mechanism for the in-field fault detection in cores of gpgpus. In *2020 23rd International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*, pages 1–6, 2020.

[51] Datla Jagannadha et al. Special session: In-system-test (ist) architecture for nvidia drive-agx platforms. In *2019 IEEE 37th VLSI Test Symposium (VTS)*, pages 1–8, 2019.

[52] Josie E. Rodriguez Condia, Felipe A. Da Silva, S. Hamdioui, C. Sauer, and M. Sonza Reorda. Untestable faults identification in gpgpus for safety-critical applications. In *2019 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pages 570–573, 2019.

[53] Siva Kumar Sastry Hari et al. Making convolutions resilient via algorithm-based error detection techniques. *IEEE Transactions on Dependable and Secure Computing*, 19(4):2546–2558, 2022.

[54] Jack Wadden et al. Real-world design and evaluation of compiler-managed gpu redundant multithreading. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 73–84, 2014.

[55] Sergi Alcaide, Leonidas Kosmidis, Carles Hernandez, and Jaume Abella. Achieving diverse redundancy for gpu kernels. *IEEE Transactions on Emerging Topics in Computing*, 10(2):618–634, 2022.

[56] Nikolaos Andriotis, Alejandro Serrano-Cases, Sergi Alcaide, Jaume Abella, Francisco J. Cazorla, Yang Peng, Andrea Baldovin, Michael Paulitsch, and Vladimir Tsymbal. A software-only approach to enable diverse redundancy on intel gpus for safety-related kernels. In *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing*, SAC '23, page 451–460, New York, NY, USA, 2023. Association for Computing Machinery.

[57] Jieyang Chen, Sihuan Li, and Zizhong Chen. Gpu-abft: Optimizing algorithm-based fault tolerance for heterogeneous systems with gpus. In *2016 IEEE International Conference on Networking, Architecture and Storage (NAS)*, pages 1–2, 2016.

[58] Jack Kosaian and K. V. Rashmi. Arithmetic-intensity-guided fault tolerance for neural network inference on gpus. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21. ACM, November 2021.

[59] Yehonatan Abotbol, Shahar Dror, Grigor Tshagharyan, Gurgen Harutyunyan, and Yervant Zorian. In-field test solution for enhancing safety in automotive applications. *Microelectronics Reliability*, 137:114774, 2022.

[60] P. Bernardi, M. Restifo, E. Sanchez, and M. Sonza Reorda. On the in-field test of embedded memories. In *2017 IEEE 23rd International Symposium on On-Line Testing and Robust System Design (IOLTS)*, pages 67–70, 2017.

[61] Andreas Riefert, Riccardo Cantoro, Matthias Sauer, Matteo Sonza Reorda, and Bernd Becker. A flexible framework for the automatic generation of sbst programs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24(10):3055–3066, 2016.

[62] Pavan Kumar Datla Jagannadha, Mahmut Yilmaz, Milind Sonawane, Sailendra Chadalavada, Shantanu Sarangi, Bonita Bhaskaran, Shashank Bajpai, Venkat Abilash Reddy, Jayesh Pandey, and Sam Jiang. Special session: In-system-test (ist) architecture for nvidia drive-agx platforms. In *2019 IEEE 37th VLSI Test Symposium (VTS)*, pages 1–8, 2019.

[63] Juan-David Guerrero-Balaguera, Josie E. Rodriguez Condia, and Matteo Sonza Reorda. Using hardware performance counters to support infield gpu testing. In *2021 28th IEEE International Conference on Electronics, Circuits, and Systems (ICECS)*, pages 1–4, 2021.

[64] Mihalis Psarakis, Dimitris Gizopoulos, Ernesto Sanchez, and Matteo Sonza Reorda. Microprocessor software-based self-testing. *IEEE Design & Test of Computers*, 27(3):4–19, 2010.

[65] Josie E. Rodriguez Condia, Felipe Augusto da Silva, Ahmet Çağrı Bağbaga, Juan-David Guerrero-Balaguera, Said Hamdioui, Christian Sauer, and Matteo Sonza Reorda. Using stls for effective in-field test of gpus. *IEEE Design & Test*, 40(2):109–117, 2023.

[66] Priyanka Viswanathan et al. State of the art software test libraries (stl) and asil b: Truths, myths, and guidance. Technical report, ARM-Technologies, 2022. Accessed: Sept. 20, 2022.

[67] Infineon Technologies. Microcontroller self-test libraries. https://www.hitex.com/tools-components/software-components/selftest-libraries-safety-libs/. [Online; accessed 26-February-2024].

[68] Microchip Technology Inc. DS52076A: 16-bit CPU Self-Test Library User's Guide. https://www.microchip.com/content/dam/mchp/softwarelibrary/16-bit-cpu-self-test-library/52076a.pdf, 2012. [Online; accessed 26-February-2024].

[69] Microchip Technology Inc. CPU Self-test Library for dsPIC33 DSCs and PIC24 MCUs. https://www.microchip.com/en-us/software-library/dspic33-pic24-cpu-self-test-library. [Online; accessed 26-February-2024].

[70] ARM Technologies. Arm Software Test Libraries . https://www.arm.com/products/development-tools/embedded-and-software/software-test-libraries. [Online; accessed 26-February-2024].

[71] Juan-David Guerrero-Balaguera, Josie E. Rodriguez Condia, and Matteo Sonza Reorda. On the functional test of special function units in gpus. In *2021 24th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*, pages 81–86, 2021.

[72] Stefano Di Carlo, Giulio Gambardella, Marco Indaco, Ippazio Martella, Paolo Prinetto, Daniele Rolfo, and Pascal Trotta. A software-based self test of cuda fermi gpus. In *2013 18th IEEE European Test Symposium (ETS)*, pages 1–6, 2013.

[73] Josie E. Rodriguez Condia and M. Sonza Reorda. Testing the divergence stack memory on gpgpus: A modular in-field test strategy. In *2020 IFIP/IEEE 28th International Conference on Very Large Scale Integration (VLSI-SOC)*, pages 153–158, 2020.

[74] Josie E. Rodriguez Condia and Matteo Sonza Reorda. On the testing of special memories in gpgpus. In *2020 IEEE 26th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, pages 1–6, 2020.

[75] Stefano Di Carlo, Josie E. Rodriguez Condia, and Matteo Sonza Reorda. An on-line testing technique for the scheduler memory of a gpgpu. *IEEE Access*, 8:16893–16912, 2020.

[76] B. Du, Josie E. Rodriguez Condia, M. Sonza Reorda, and L. Sterpone. About the functional test of the gpgpu scheduler. In *2018 IEEE 24th International Symposium on On-Line Testing And Robust System Design (IOLTS)*, pages 85–90, 2018.

[77] Josie Rodriguez Condia and Matteo Sonza Reorda. Testing permanent faults in pipeline registers of gpgpus: A multi-kernel approach. In *2019 IEEE 25th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, pages 97–102, 2019.

[78] Paolo Bernardi, Riccardo Cantoro, Sergio De Luca, Ernesto Sánchez, and Alessandro Sansonetti. Development flow for on-line core self-test of automotive microcontrollers. *IEEE Transactions on Computers*, 65(3):744–754, 2016.

[79] Heather Quinn. *Microprocessor Testing*, pages 83–125. Springer, Velazco, R., McMorrow, D., Estela, J. (eds) Radiation Effects on Integrated Circuits and Systems for Space Applications, 2019.

[80] Juan-David Guerrero-Balaguera, Josie E. Rodriguez Condia, and Matteo Sonza Reorda. A new method to generate software test libraries for in-field gpu testing resorting to high-level languages. In *2022 IEEE 40th VLSI Test Symposium (VTS)*, pages 1–7, 2022.

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

[82] M S Vasudevan, Santosh Biswas, and Aryabartta Sahu. Automated low-cost sbst optimization techniques for processor testing. In *2021 34th International Conference on VLSI Design and 2021 20th International Conference on Embedded Systems (VLSID)*, pages 299–304, 2021.

[83] Marco Gaudesi, Irith Pomeranz, Matteo Sonza Reorda, and Giovanni Squillero. New techniques to reduce the execution time of functional test programs. *IEEE Transactions on Computers*, 66(7):1268–1273, 2017.

[84] A. Touati, A. Bosio, P. Girard, A. Virazel, P. Bernardi, and M. Sonza Reorda. An effective approach for functional test programs compaction. In *2016 IEEE 19th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*, pages 1–6, 2016.

[85] R. Cantoro, M. Gaudesi, E. Sanchez, P. Schiavone, and G. Squillero. An evolutionary approach for test program compaction. In *2015 16th Latin-American Test Symposium (LATS)*, pages 1–6, 2015.

[86] R. Cantoro, E. Cetrulo, E. Sanchez, M. Sonza Reorda, and A. Voza. Automated test program reordering for efficient sbst. In *2017 32nd Conference on Design of Circuits and Integrated Systems (DCIS)*, pages 1–6, 2017.

[87] Muhammed Al Kadi et al. Fgpu: An simt-architecture for fpgas. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA ’16, page 254–263, 2016.

[88] Dimitris Sartzetakis et al. gpufi-4: A microarchitecture-level framework for assessing the cross-layer resilience of nvidia gpus. In *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 35–45, 2022.

[89] B. Fang 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*, pages 221–230, March 2014.

[90] B. Nie et al. Fault site pruning for practical reliability analysis of GPGPU applications. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 749–761, 2018.

[91] Alessandro Vallero, Dimitris Gizopoulos, and Stefano Di Carlo. Sifi: Amd southern islands gpu microarchitectural level fault injector. In *2017 IEEE 23rd International Symposium on On-Line Testing and Robust System Design (IOLTS)*, pages 138–144, 2017.

[92] J. Wei 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*, pages 375–382, 2014.

[93] Lishan Yang, Bin Nie, Adwait Jog, and Evgenia Smirni. Practical resilience analysis of gpgpu applications in the presence of single- and multi-bit faults. *IEEE Transactions on Computers*, 70(1):30–44, 2021.

[94] F. F. d. Santos et al. Analyzing and increasing the reliability of convolutional neural networks on GPUs. *IEEE Transactions on Reliability*, 68(2):663–677, 2019.

[95] D. A. G. Goncalves de Oliveira et al. Evaluation and mitigation of radiation-induced soft errors in graphics processing units. *IEEE Transactions on Computers*, 65(3):791–804, 2016.

[96] Kojiro Ito et al. Analyzing due errors on gpus with neutron irradiation test and fault injection to control flow. *IEEE Transactions on Nuclear Science*, 68(8):1668–1674, 2021.

[97] G. Leon et al. Evaluating the soft error sensitivity of a gpu-based soc for matrix multiplication. *Microelectronics Reliability*, 114:113856, 2020. 31st European Symposium on Reliability of Electron Devices, Failure Physics and Analysis, ESREF 2020.

[98] Michael B. Sullivan et al. *Characterizing And Mitigating Soft Errors in GPU DRAM*, pages 641–653. Association for Computing Machinery, New York, NY, USA, 2021.

[99] P. Rech, L.L. Pilla, P.O.A. Navaux, and L. Carro. Impact of gpus parallelism management on safety-critical and hpc applications reliability. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 455–466, 2014.

[100] Sotiris Tselonis et al. The functional and performance tolerance of gpus to permanent faults in registers. In *2013 IEEE 19th International On-Line Testing Symposium (IOLTS)*, pages 236–239, 2013.

[101] Dimitris Sartzetakis et al. gpufi-4: A microarchitecture-level framework for assessing the cross-layer resilience of nvidia gpus. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2022)*, pages 236–239, 2022.

[102] Annachiara Ruospo et al. A pipelined multi-level fault injector for deep neural networks. In *2020 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pages 1–6, 2020.

[103] Corrado De Sio, Sarah Azimi, and Luca Sterpone. Firenn: Neural networks reliability evaluation on hybrid platforms. *IEEE Transactions on Emerging Topics in Computing*, 10(2):549–563, 2022.

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

[105] Juan-David Guerrero-Balaguera, Luigi Galasso, Robert Limas Sierra, and Matteo Sonza Reorda. Reliability assessment of neural networks in gpus: A framework for permanent faults injections. In *2022 IEEE 31st International Symposium on Industrial Electronics (ISIE)*, pages 959–962, 2022.

[106] Juan-David Guerrero-Balaguera, Luigi Galasso, Robert Limas Sierra, Ernesto Sanchez, and Matteo Sonza Reorda. Evaluating the impact of permanent faults in a gpu running a deep neural network. In *2022 IEEE International Test Conference in Asia (ITC-Asia)*, pages 96–101, 2022.

[107] Juan-David Guerrero-Balaguera, Robert Limas Sierra, and Matteo Sonza Reorda. Effective fault simulation of gpu's permanent faults for reliability estimation of cnns. In *2022 IEEE 28th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, pages 1–6, 2022.

[108] J. Irwin, D. Page, and N.P. Smart. Instruction stream mutation for non-deterministic processors. In *Proceedings IEEE International Conference on Application- Specific Systems, Architectures, and Processors*, pages 286–295, 2002.

[109] J. Duraes and H. Madeira. Characterization of operating systems behavior in the presence of faulty drivers through software fault emulation. In *2002 Pacific Rim International Symposium on Dependable Computing, 2002. Proceedings.*, pages 201–209, 2002.

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

[111] Troya Çağ&il Köylü, Cezar Rodolfo Wedig Reinbrecht, Said Hamdioui, and Mottaqiallah Taouil. Rnn-based detection of fault attacks on rsa. In *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, 2020.

[112] Siva Kumar Sastry Hari, Timothy Tsai, Mark Stephenson, Stephen W. Keckler, and Joel Emer. Sassifi: An architecture-level fault injection tool for gpu application resilience evaluation. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 249–258, 2017.

[113] George Papadimitriou and Dimitris Gizopoulos. Demystifying the system vulnerability stack: Transient fault effects across the layers. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 902–915, 2021.

[114] Hyungmin Cho et al. Understanding soft errors in uncore components. In *Proceedings of the 52nd Annual Design Automation Conference*, DAC '15, 2015.

[115] Anderson L. Sartor et al. A fast and accurate hybrid fault injection platform for transient and permanent faults. *Design Automation for Embedded Systems*, 23(1–2), 2019.

[116] Bo Fang et al. A systematic methodology for evaluating the error resilience of gpgpu applications. *IEEE Transactions on Parallel and Distributed Systems*, 27(12):3397–3411, 2016.

[117] Fernando F. dos Santos, Josie E. Rodriguez Condia, Luigi Carro, Matteo Sonza Reorda, and Paolo Rech. 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)*, pages 292–304, 2021.

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

[119] Oreste Villa et al. Nvbit: A dynamic binary instrumentation framework for nvidia gpus. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 372–383, New York, NY, USA, 2019. Association for Computing Machinery.

[120] Said Hamdioui, Dimitris Gizopoulos, Groeseneken Guido, Michael Nicolaidis, Arnaud Grasset, and Philippe Bonnot. Reliability challenges of real-time systems in forthcoming technology nodes. In *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 129–134, 2013.

[121] David Defour and Eric Petit. Gpuburn: A system to test and mitigate gpu hardware failures. In *2013 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pages 263–270, 2013.

[122] L. Hochstein et al. Parallel programmer productivity: A case study of novice parallel programmers. In *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, pages 35–35, 2005.

[123] Matina Maria Trompouki and Leonidas Kosmidis. Brook auto: High-level certification-friendly programming for gpu-powered automotive systems. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6, 2018.

[124] Gautam Chakrabarti, Vinod Grover, Bastiaan Aarts, Xiangyun Kong, Manjunath Kudlur, Yuan Lin, Jaydeep Marathe, Mike Murphy, and Jian-Zhong Wang. Cuda: Compiling and optimizing for a gpu platform. *Procedia Computer Science*, 9:1910–1919, 2012. Proceedings of the International Conference on Computational Science, ICCS 2012.

[125] Mayler Martins, Jody Maick Matos, Renato P. Ribas, André Reis, Guilherme Schlinker, Lucio Rech, and Jens Michelsen. Open cell library in 15nm freepdk technology. In *Proceedings of the 2015 Symposium on International Symposium on Physical Design*, ISPD '15, page 171–178, New York, NY, USA, 2015. Association for Computing Machinery.

[126] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[127] E. Sanchez, M. Schillaci, and G. Squillero. Enhanced test program compaction using genetic programming. In *2006 IEEE International Conference on Evolutionary Computation*, pages 865–870, 2006.

[128] M. Gaudesi, M. Sonza Reorda, and I. Pomeranz. On test program compaction. In *2015 20th IEEE European Test Symposium (ETS)*, pages 1–6, 2015.

[129] Pulp Team. PULP Platform: Open hardware, the way it should be! . https://www.pulp-platform.org/, 2013. [Online; accessed 26-February-2024].

[130] J. Knudsen. Nangate 45nm open cell library . CDNLive, EMEA, 2008.

[131] Davide Sabena, Matteo Sonza Reorda, and Luca Sterpone. A new sbst algorithm for testing the register file of vliw processors. In *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 412–417, 2012.

[132] J. Perez Acle, R. Cantoro, E. Sanchez, M. Sonza Reorda, and G. Squillero. Observability solutions for in-field functional test of processor-based systems: A survey and quantitative test case evaluation. *Microprocessors and Microsystems*, 47:392–403, 2016.

[133] Juan-David Guerrero-Balaguera, Josie E. Rodriguez C., Fernando F. do Santos, Matteo Sonza Reorda, and Paolo Rech. Nvbitperfi. https://github.com/divadnauj-GB/nvbitPERfi, June 2023.

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

[135] A. Ejlali et al. A hybrid fault injection approach based on simulation and emulation co-operation. In *International Conference on Dependable Systems and Networks*, pages 479–488, 2003.

[136] Douglas Almeida Santos et al. Reliability analysis of a fault-tolerant risc-v system-on-chip. *Microelectronics Reliability*, 125:114346, 2021.

[137] Raghuraman Balasubramanian and Karthikeyan Sankaralingam. Understanding the impact of gate-level physical reliability effects on whole program execution. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 60–71, 2014.

[138] Raoul Velazco et al. Combining results of accelerated radiation tests and fault injections to predict the error rate of an application implemented in sram-based fpgas. *IEEE Transactions on Nuclear Science*, 57(6):3500–3505, 2010.

[139] Sergiu Nimara, A. Amaricai, Oana Boncalo, and M. Popa. Multi-level simulated fault injection for data dependent reliability analysis of rtl circuit descriptions. *Advances in Electrical and Computer Engineering*, 16:93–98, 02 2016.

[140] Atieh Lotfi et al. Resiliency of automotive object detection networks on gpu architectures. In *2019 IEEE International Test Conference (ITC)*, pages 1–9, 2019.

[141] Hussam Amrouch and Jorg Henkel. Reliability degradation in the scope of aging — from physical to system level. In *2015 10th International Design & Test Symposium (IDT)*, pages 9–12, 2015.

[142] Jörg Henkel et al. Reliable on-chip systems in the nano-era: Lessons learnt and future trends. In *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–10, 2013.

[143] Denis Dutey et al. Prevention and detection methods of systematic failures in the implementation of soc safety mechanisms not covered by regular functional tests. In *2021 24th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, pages 87–92, 2021.

[144] Yann LeCun et al. Lenet-5, convolutional neural networks. *URL: http://yann.lecun.com/exdb/lenet*, 20(5), 2015.

[145] Joseph Redmon. Darknet: Open source neural networks in c. http://pjreddie.com/darknet/, 2013–2016.

Appendix A

Publication list of the Author

The following list enumerates the research and dissemination products developed during the PhD research activities.

A.1 Journal Publications

2023

1. J. E. Rodriguez Condia, F. A. da Silva, A. Ç. Bağbaga, **J. -D. Guerrero-Balaguera**, S. Hamdioui, C. Sauer, M. Sonza Reorda, "Using STLs for Effective In-Field Test of GPUs," in IEEE Design & Test, vol. 40, no. 2, pp. 109-117, April 2023, doi: 10.1109/MDAT.2022.3188573.

2. F. Fernandez dos Santos, A. Kritikakou, J. E. Rodriguez Condia, **J. -D. Guerrero-Balaguera**, M. Sonza Reorda, O. Sentieys, P. Rech , "Characterizing a Neutron-Induced Fault Model for Deep Neural Networks," in IEEE Transactions on Nuclear Science, vol. 70, no. 4, pp. 370-380, April 2023, doi: 10.1109/TNS.2022.3224538.

3. **J. -D. Guerrero-Balaguera**, J. E. Rodriguez Condia and M. Sonza Reorda, "STLs for GPUs: Using High-Level Language Approaches," in IEEE Design & Test, vol. 40, no. 4, pp. 51-60, Aug. 2023, doi: 10.1109/MDAT.2023.3267601.

2024

1. R. Limas Sierra, **J.-D Guerrero-Balaguera**, J. E. Rodriguez Condia, M. Sonza Reorda, "Exploring Hardware Fault Impacts on Different Real Number Representations of the Structural Resilience of TCUs in GPUs." Electronics 2024, 13, 578. https://doi.org/10.3390/electronics13030578.

2. J. E. Rodriguez Condia, **J. -D. Guerrero-Balaguera**, E. J. Patiño Núñez, R. Limas Sierra, M. Sonza Reorda, "Investigating and Reducing the Architectural Impact of Transient Faults in Special Function Units for GPUs," in Journal of Electronic Testing (2024), pp. 1-14. https://doi.org/10.1007/s10836-024-06107-9.

3. **J. -D. Guerrero-Balaguera**, J. E. Rodriguez Condia, M. Sonza Reorda, "Effective Fault Injection Techniques for Permanent Faults in GPUs running DNNs," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, **Submitted, Under Review**.

A.2 Conference Proceedings Publications

2021

1. **J. -D. Guerrero-Balaguera**, J. E. Rodriguez Condia and M. Sonza Reorda, "On the Functional Test of Special Function Units in GPUs," 2021 24th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS), Vienna, Austria, 2021, pp. 81-86, doi: 10.1109/DDECS52668.2021.9417025. **Best paper Award** in the field of testing.

2. **J. -D. Guerrero-Balaguera**, J. E. Rodriguez Condia and M. Sonza Reorda, "A Novel Compaction Approach for SBST Test Programs," 2021 IEEE 30th Asian Test Symposium (ATS), Matsuyama, Ehime, Japan, 2021, pp. 67-72, doi: 10.1109/ATS52891.2021.00024.

3. **J. -D. Guerrero-Balaguera**, J. E. Rodriguez Condia and M. Sonza Reorda, "Using Hardware Performance Counters to support infield GPU Testing," 2021 28th IEEE International Conference on Electronics, Circuits, and Systems

(ICECS), Dubai, United Arab Emirates, 2021, pp. 1-4, doi: 10.1109/ICECS53 924.2021.9665511.

2022

1. **J. -D. Guerrero-Balaguera**, J. E. Rodriguez Condia and M. Sonza Reorda, "A Compaction Method for STLs for GPU in-field test," 2022 Design, Automation & Test in Europe Conference & Exhibition (DATE), Antwerp, Belgium, 2022, pp. 454-459, doi: 10.23919/DATE54114.2022.9774597.

2. **J. -D. Guerrero-Balaguera**, J. E. Rodriguez Condia and M. Sonza Reorda, "A New Method to Generate Software Test Libraries for In-Field GPU Testing Resorting to High-Level Languages," 2022 IEEE 40th VLSI Test Symposium (VTS), San Diego, CA, USA, 2022, pp. 1-7, doi: 10.1109/VTS52500.2021.9794 225.

3. **J. -D. Guerrero-Balaguera**, L. Galasso, R. L. Sierra and M. Sonza Reorda, "Reliability Assessment of Neural Networks in GPUs: A Framework For Permanent Faults Injections," 2022 IEEE 31st International Symposium on Industrial Electronics (ISIE), Anchorage, AK, USA, 2022, pp. 959-962, doi: 10.1109/ISIE51582.2022.9831549.

4. **J. -D. Guerrero-Balaguera**, L. Galasso, R. L. Sierra, E. Sanchez, and M. Sonza Reorda, "Evaluating the impact of Permanent Faults in a GPU running a Deep Neural Network," 2022 IEEE International Test Conference in Asia (ITC-Asia), Taipei, Taiwan, 2022, pp. 96-101, doi: 10.1109/ITCAsia55616.2022.00027.

5. **J. -D. Guerrero-Balaguera**, R. L. Sierra and M. Sonza Reorda, "Effective fault simulation of GPU's permanent faults for reliability estimation of CNNs," 2022 IEEE 28th International Symposium on On-Line Testing and Robust System Design (IOLTS), Torino, Italy, 2022, pp. 1-6, doi: 10.1109/IOLTS56730.2022.9897823.

6. J. E. Rodriguez Condia, **J. -D. Guerrero-Balaguera**, F. Fernandez Dos Santos, M. Sonza Reorda and P. Rech, "A Multi-level Approach to Evaluate the

Impact of GPU Permanent Faults on CNN's Reliability," 2022 IEEE International Test Conference (ITC), Anaheim, CA, USA, 2022, pp. 278-287, doi: 10.1109/ITC50671.2022.00036.

7. **J. -D. Guerrero-Balaguera**, J. E. Rodriguez Condia and M. Sonza Reorda, "Neural Network's Reliability to Permanent Faults: Analyzing the Impact of Performance Optimizations in GPUs," 2022 29th IEEE International Conference on Electronics, Circuits and Systems (ICECS), Glasgow, United Kingdom, 2022, pp. 1-4, doi: 10.1109/ICECS202256217.2022.9971036.

8. F. Angione, D. Appello, J. Aribido, J. Athavale, N. Bellarmino, P. Bernardi, R. Cantoro, C. De Sio, T. Foscale, G. Gavarini, **J. Guerrero**, M. Huch, G. Iaria, T. Kilian, R. Mariani, R. Martone, A. Ruospo, E. Sanchez, U. Schlichtmann, G. Squillero, M. Sonza Reorda, L. Sterpone, V. Tancorre, R. Ugioli, "Test, Reliability and Functional Safety Trends for Automotive System-on-Chip," 2022 IEEE European Test Symposium (ETS), Barcelona, Spain, 2022, pp. 1-10, doi: 10.1109/ETS54262.2022.9810388.

2023

1. J. E. Rodriguez Condia, **J. -D. Guerrero-Balaguera**, E. J. Patiño Núñez, R. Limas and M. Sonza Reorda, "Analyzing the Architectural Impact of Transient Fault Effects in SFUs of GPUs," 2023 IEEE 24th Latin American Test Symposium (LATS), Veracruz, Mexico, 2023, pp. 1-6, doi: 10.1109/LATS5812 5.2023.10154504.

2. A. Ruospo, G. Gavarini, C. de Sio, **J. Guerrero**, L. Sterpone, M. Sonza Reorda, E. Sanchez, R. Mariani, J. Aribido, J. Athavale, "Assessing Convolutional Neural Networks Reliability through Statistical Fault Injections," 2023 Design, Automation & Test in Europe Conference & Exhibition (DATE), Antwerp, Belgium, 2023, pp. 1-6, doi: 10.23919/DATE56975.2023.10136998.

3. R. L. Sierra, **J. -D. Guerrero-Balaguera**, J. E. Rodriguez Condia and M. Sonza Reorda, "A Reliability-aware Environment for Design Exploration for GPU Devices," 2023 26th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS), Tallinn, Estonia, 2023, pp. 169-174, doi: 10.1109/DDECS57882.2023.10139643.

4. J. E. Rodriguez Condia, **J. -D. Guerrero-Balaguera**, E. J. Patiño Núñez, R. Limas and M. Sonza Reorda, "Evaluating the Prevalence of SFUs in the Reliability of GPUs," 2023 IEEE European Test Symposium (ETS), Venezia, Italy, 2023, pp. 1-6, doi: 10.1109/ETS56758.2023.10174110.

5. **J. -D. Guerrero-Balaguera**, I. A. Harshbarger, J. E. Rodriguez Condia, M. Levorato and M. Sonza Reorda, "Reliability Estimation of Split DNN Models for Distributed Computing in IoT Systems," 2023 IEEE 32nd International Symposium on Industrial Electronics (ISIE), Helsinki, Finland, 2023, pp. 1-4, doi: 10.1109/ISIE51358.2023.10227928.

6. R. L. Sierra, **J. -D. Guerrero-Balaguera**, J. E. Rodriguez Condia and M. Sonza Reorda, "Analyzing the Impact of Different Real Number Formats on the Structural Reliability of TCUs in GPUs," 2023 IFIP/IEEE 31st International Conference on Very Large Scale Integration (VLSI-SoC), Dubai, United Arab Emirates, 2023, pp. 1-6, doi: 10.1109/VLSI-SoC57769.2023.10321881.

7. R. L. Sierra, **J. -D. Guerrero-Balaguera**, J. E. Rodriguez Condia and M. Sonza Reorda, "Optimizing the Analysis and Evaluation of Logic Simulation Workloads in HPC Systems," 2023 IEEE 17th International Conference on Application of Information and Communication Technologies (AICT), Baku, Azerbaijan, 2023, pp. 1-6, doi: 10.1109/AICT59525.2023.10313156.

8. **J. -D Guerrero-Balaguera**, J. E. Rodriguez Condia, F. Fernandes Dos Santos, M. Sonza Reorda, and P. Rech. 2023. "Understanding the Effects of Permanent Faults in GPU's Parallelism Management and Control Units." In Proceedings of the International Conference for High-Performance Computing, Networking, Storage and Analysis (SC' 23). Association for Computing Machinery, New York, NY, USA, Article 46, 1–14. https://doi.org/10.1145/3581784.3607086

2024

1. R. Limas Sierra, **J. -D Guerrero-Balaguera**, F. Pessia, J. E. Rodriguez Condia, M. Sonza Reorda, "Analyzing the Impact of Scheduling Policies on the Reliability of GPUs Running CNN Operations," 42nd IEEE VLSI Test Symposium (VTS2024), Tempe, AZ, USA, 2024.

2. **J. -D Guerrero-Balaguera**, J. E. Rodriguez Condia, M. Levorato, M. Sonza Reorda, "Evaluating the Reliability of Supervised Compression for Split Computing," 42nd IEEE VLSI Test Symposium (VTS2024), Tempe, AZ, USA, 2024.

3. G. Esposito, **J. -D Guerrero-Balaguera**, J. E. Rodriguez Condia, M. Levorato, M. Sonza Reorda, "Assessing the Reliability of Different Split Computing Neural Network Applications," 25th IEEE Latin American Test Symposium (LATS2024), Maceió, Brazil, 2024.

4. L. A. Mesa, **J. -D Guerrero-Balaguera**, E. D. Castañeda, E. Sanchez, W. -J. Perez-Holguin, "An Integrated Environment for the Reliability Assessment of CNNs Accelerators Implemented in FPGAs," 25th IEEE Latin American Test Symposium (LATS2024), Maceió, Brazil, 2024.

5. J. E. Rodriguez Condia, **J. -D Guerrero-Balaguera**, R. Limas Sierra, M. Sonza Reorda, "Analyzing the Structural and Operational Impact of Faults in Floating-Point and Posit Arithmetic Cores for CNN Operations," 29th IEEE European Test Symposium (ETS), The Hague, Netherlands, 2024.

6. F. Pessia, **J. -D Guerrero-Balaguera**, R. Limas Sierra, J. E. Rodriguez Condia, M. Levorato, M. Sonza Reorda, "Effective Application-level Error Modeling of Permanent Faults on AI Accelerators," 30th IEEE International Symposium on On-Line Testing and Robust System Design (IOLTS2024).

7. G. Esposito, **J. -D Guerrero-Balaguera**, J. E. Rodriguez Condia, M. Levorato, M. Sonza Reorda, "Enhancing the Reliability of Split Computing Deep Neural Networks," 30th IEEE International Symposium on On-Line Testing and Robust System Design (IOLTS2024).