Doctoral Dissertation
Doctoral Program in Computer and Control Engineering (33th cycle)

New Techniques for On-line Testing and Fault Mitigation in GPUs

Josie Esteban Rodriguez Condia

* * * * * *

Supervisor
Prof. Matteo Sonza Reorda

Politecnico di Torino
September, 2021

. . . .
Josie Esteban Rodriguez Condia
Turin, September, 2021

Summary

Currently, Graphics Processing Units (GPUs) are crucial devices able to boost the execution of complex algorithms in the scientific and artificial intelligence domains. Moreover, GPU-based platforms are also relevant components now included in several safety-critical applications (e.g., in the automotive and autonomous machines fields), where reliability and functional safety are essential requirements.

The doctoral research activities were focused on identifying new online techniques for testing and mitigation of faults affecting GPUs.

This work also describes a new microarchitectural GPU model (FlexGripPlus), a supporting tool for detailed reliability evaluation. FlexGripPlus can also be used to support the development of functional test approaches and mitigation solutions. This GPU model is compatible with the CUDA programming environment and is a corrected and extended version of a previous GPU model implementing the G80 architecture by NVIDIA. The GPU model's extensions include support for floating-point operations and for the execution of trigonometric and transcendental operations using Special Function Units.

FlexGripPlus was used to develop, evaluate, and validate functional test techniques based on the software-based self-test (SBST) strategy. More in detail, two main strategies are proposed for GPUs: *i)* the multi-kernel test approach, and *ii)* the modular approach of testing.

On the one hand, the multi-kernel approach exploits the GPU's main operative behavior as a special-purpose accelerator. This approach is based on the parallel execution of test programs intended to detect and propagate the fault effects on any GPU core's available outputs. For this purpose, a set of parallel test programs with different GPU configuration parameters allows the generation of test patterns after dividing a target module into fault groups. This division process allows the coverage of sensitive locations that implicitly remain constant by the effect of the configuration of a parallel program. Moreover, this technique also uses a thread-based method to propagate and identify faults into the GPU's available outputs, simplifying the evaluation of specific modules in the GPU architecture, such as the pipeline registers.

On the other hand, the modular testing approach combines the microarchitectural details of a given module, its functional operation, its major constraints, and a target fault model to design a generic description of a feasible test program. More in detail, these GPU features are combined to generate a scalable high-level abstraction test program, which can later be transformed into the equivalent software routines to test a target module. This modular approach exhibited a good effectiveness when testing internal memories and the divergence stack of the GPU core.

It is worth noting that in both strategies (multi-kernel and modular), the combination of high-level programming and low-level assembly language was adopted whenever it was possible. Moreover, several compilation constraints and limitations were listed when implementing the test programs. Both functional test strategies were evaluated targeting the detection of permanent fault models.

Finally, three hybrid and flexible strategies were proposed to harden the GPU architecture. The proposed hardening solution allows the online fault testing and in-field fault mitigation. All three strategies are configurable using custom instructions, so allowing their adoption as part of the code of an application.

The first strategy is based on a flexible approach for in-field fault detection applied to the execution units of the GPU core. This strategy exploits the high regularity of the execution units and provides hardening with reduced overhead costs. The second strategy allows in-field fault mitigation of any functional unit once a fault is detected. In this approach, several in-field configurable spare units are used to provide repair capabilities in the GPU.

The third strategy aims at detecting and mitigating faults at the same time. This flexible approach allows the activation of one or both reliability features. Moreover, the reliability analysis of the proposed strategy showed a considerable increase in the reliability of the targeted units with a minor effect on the running applications code and minimal effect on performance. Furthermore, the implementation of the strategies was evaluated in several configurations, so determining different hardware overhead figures of the proposed hardware mitigation architectures. In the end, these analyzes provide a advisable configuration to obtain maximum reliability benefits with reduced hardware and performance overheads.

Thanks to the availability of the FlexGripPlus model, this work includes for the first time (to the best our knowledge) quantitative results in terms of microarchitectural reliability evaluations aimed at identifying the fault impact of transient faults on several modules of a GPU core, including the scheduler controller, the pipeline registers, the register file, and the branch unit. All these evaluations were performed using several parallel applications. Furthermore, as described above, this model was used as a validation tool in the quantitative evaluation of several online test solutions and hardening strategies for GPUs.

The main results of the research activities are intended to support the development of fault detection and fault-tolerance mechanisms for GPU devices devoted to safety-critical applications, based on the proposed solutions for online testing and mitigation of faults. Moreover, the experimental results support the evaluation of the different reliability solutions, which are required in the current trends of continuous increment of GPU devices in applications, where the effect of faults is critical.

Finally, as product of the research activities and according to the obtained experimental results, the developed strategies for functional in-field test and in-field mitigation increase the reliability (in up to 50%) and functional-safety (ASIL B) of the GPU, so the combination of the developed strategies can be employed as alternative or complementary fault-tolerance mechanisms for GPUs devoted to applications in the safety-critical domain.

Acknowledgements

- To my Mom and Dad, who encouraged me with tremendous moral support and reminded me to follow my dreams.
- To my sister Melissa and my cousin Sebastian, who supported me with good vibes and fun when were required.
- I would like to express my respect and huge thanks to my mentor and advisor, Prof. Matteo Sonza Reorda, for the vast support and for showing me how simple ideas can turn into reality with discipline and hard work.
- To my friends Aleksa D., Andrea F. and Pratyush K. for the academic and moral support in all steps of this journey.
- To my friends "*ESRs*" from RESCUE-ETN project for the huge support; each meeting was a way to recharge myself with good energy to continue working with more determination.
- To the RESCUE-ETN project, to make this journey possible.
- To all the people that in any way contributed to the development of this work.

  Gracias a todos!

♫
. . .
And I dedicate my roses to no one
. . .
♫
Roses to No One
The Savage Poetry, 2000
Edguy

Contents

List of Tables

List of Figures

List of abbreviations

ABFT Algorithm-Based Fault-Tolerance
aTM Active Threads Mask
ADAS Advanced Driver-Assistance Systems
AFFP(SCH) Addressable Functional Fault Primitive for the scheduler
AI Artificial Intelligence
ARF Address Register File
ASIL Automotive Safety Integrity Level
AVF Architectural Vulnerability Factor
Acc Accumulative
BBK Basic Block Kernel
BISR Built-In Self-Repair
BIST Build-In Self-Test
CFR Component Fault Rate
CFst State Coupling Faults
CMU Convergence Management Unit
CNN Convolutional Neural Network
CORDIC COordinate Rotation DIgital Computer
CP Convergence Point
CUDA Compute Unified Device Architecture
CpT Counter-per-Thread
C Carry
DDWC Dynamic Duplication With Comparison
DETT DETection Trigger
DMR Double Modular Redundancy
DMU Divergence Management Unit
DRDF Deceptive Read destructive fault
DUE Detected Unrecoverable Error
DWC Duplication with Comparison
DYRE DYnamic REconfigurable structure for infield detection and mitigation of faults
Dec Detection
DfT Design for Testability
Dia Diagnosis
ECC Error Correcting Code
EMI Electromagnetic Interference
ETFD Estimate the time for fault detection
EU Execution Unit
FAS Fault Administration Structure
FC Fault Coverage
FFT Fast Fourier Transformation

FIT	Failures In Time
FPGA	Field Programmable Gate Arrays
FP	Fault Primitives
FP32/FP64	Floating-Point Unit
FUF	Functionally Untestable Fault
GEMM	General Matrix Multiplication
GPGPU	General-Purpose Graphics Procesing Unit
GPRS	General-Purpose Registers
GPU	Graphics Procesing Unit
HPC	High Performance Computing
IM	Infant Mortality
INT	Integer Units
IP	Intellectual Property
IRF	Incorrect Read Faults
ISA	instruction set architecture
IST	In-System-Structural-Test
LBIST	Logic Build-In Self-Test
LE	Line Entry
LRU	Least Recently Used
LSU	Load/Store unit
MBIST	Memory Build-In Self-Test
MIMD	Multiple-Instruction Multiple-Data
MITT	MITigation Trigger
MxM	Matrix Multiplication
M	Max
NVM	Non Volatile Memory
OpenCL	Open Computing Language
O	Overflow
PRF	Predicate Register File
PRs	Pipeline Registers
PTX	Parallel Thread Execution pseudo-language
R(x)	Reading
RAM	Random Access Memory
RCP	Reciprocal
RDF	Read destructive fault
RFU	Register Forwarding Unit
RF	Register File
RISC	Reduced-Instruction Set Computer
RSQ	Reciprocal of the Square Root
RTL	Register Transfer Level
R	Random
SASS	Streaming ASSembly language

SBST Software-Based Self-Testing
SBU Single Bit Upset
SC Scheduler Controller
SDC Silent Data Corruption
SEU Single Event Upset
SFU Special Function Unit
SIMD Single-Instruction Multiple-Data
SIMT Single-Instructions Multiple-Thread
SM Streaming Multiprocessor
SOS Sensitizing Operation Sequence
SPC Program Counter of a warp
SPDC Output Data Channel of an Streaming Processor
SPs Streaming/Scalar Processor
SPx Redundant Streaming Processor
SSP Spare Streaming Processor
ST Execution Time
SoCs Systems-On-Chips
SpT Signature-per-Thread
S Sign
TAM Thread Mask
TCU Tensor Core Unit
TFC Testable Fault Coverage
TF Transition fault
TLM Transaction-Level Modeling
TMR Triple Module Redundancy
TM Thread Mask
TP(SCH) Test Pattern for the scheduler
TPB Threads per block
TP Test Pattern
UDR Utilization De-Rating
VHDL Very High Speed Integrated Circuit Hardware Description Language
wPC Warp Program Counter
W(x) Writing
WDF Write destructive fault
XBIST X-state Build-In Self-Test
Z Zero

Chapter 1

Introduction

Currently, new technologies boost our life quality in numerous aspects while increasing the productivity and the efficiency in the society. Simple examples, such as innovative environmental-friendly transportation systems and high-end production methods, reduce the human effort, and optimize appliances and services' production.

In case of automotive systems, new technology trends are focused on introducing new features, which extend the number of on-board embedded systems (and processors) [1]. In the automotive domain, these systems are intended to optimize energy consumption, extend the user's experience by introducing infotainment support, and also provide autonomous and semi-autonomous control mechanisms, including strategies for cruise control and autonomous pilot [2]. Furthermore, in the production case, new tendencies of automation promote cooperative work environments between humans and autonomous robots interacting and increasing the production. Moreover, this automation method aims at reducing the human risk when hazard conditions are involved.

Both cases (automotive and industrial production) are current examples of safety-critical applications where any functional failure of the equipment, machines or devices supporting the application can generate severe consequences, including critical injuries, deaths, significant property damage or extensive environmental damages [3]. For this purpose, the complex electronic devices, which are now integrated into systems, must accomplish strict safety, reliability and security constraints to guarantee the correct operation of the entire system.

1.1 Main Motivation

In the automotive sector, leading companies have invested (and plan to invest) billions in new technologies to implement and extend their usage to several applications involved in the sector, as reported in Figure 1.1. Those applications include the development of various levels of autonomy in vehicles motivated by the benefits in terms of safety and security for the users, latency reduction in the traffic and energy efficiency. However, these technological benefits also imply several technology challenges without clear solutions till the moment.

In principle, mature methods of design and development of safe and secure devices can be employed for devices used in these new applications. However, both sectors (automotive and autonomous machines) now exploit several innovative technologies, such as Artificial Intelligence (AI) and computer vision, which suppose a big advantage for more efficient procedures. However, this trend also implies that modern devices can implement such sophisticated algorithms, so increasing the application's complexity and imposing huge restrictions in terms of real-time operation, available power budget and performance.

In practice, the development of modern safety-critical applications require mainly three elements: *i)* a high performance operation and power efficiency, *ii)* affordable costs, and *iii)* safety and reliability [4]. In many cases, these requirements are faced by most manufacturers and designers using the latest transistor technology and scaling approaches in their products, so reaching the limits of the Moore's law in order to include a high number of transistors in the same device, so obtaining considerable improvements in execution performance, power consumption and practical production costs. However, several studies [5], [6], [7], [8], [9] demonstrated that new devices implemented with these cutting-edge technologies are also prone to be affected by multiple types of faults arising at the early operational stages and, more frequently, during their operative life. These faults can be caused by two main factors: *i)* internal defects derived by the manufacturing processes or component fatigues, and *ii)* environmental or external ones [10].

In the first case, the faults in a device can be the consequence of manufacturing defects that were not identified or detected during the end-of-production testing, so causing unexpected behaviors during its operative life. Furthermore, a device is also prone to suffer of degradation (e.g., electromigration or gate-oxide effects) of its components after long-periods of operation, or even when the device is not used (e.g., idle operational mode) [11], and produce intermittent or permanent faults. In the previous scenarios, the faults are produced by aging or wearout effects [12], [13], [14].

On the other hand, external effects can also affect the operation of a device. In this case, environmental effects change the electrical parameters temporary (or permanently) and produce transient fault effects, so affecting the running application in the device. These fault effects are propagated across the device as soft-errors, which are produced only when the application is running on an affected device.

The exposure of a device to high energy particles (producing radiation effects) or to electromagnetic interference (EMI) increases the sensibility of the device to temporary fault (transient faults), which swap the electronic charge of any or multiple storage components in the device, so switching ON or OFF a transistor used

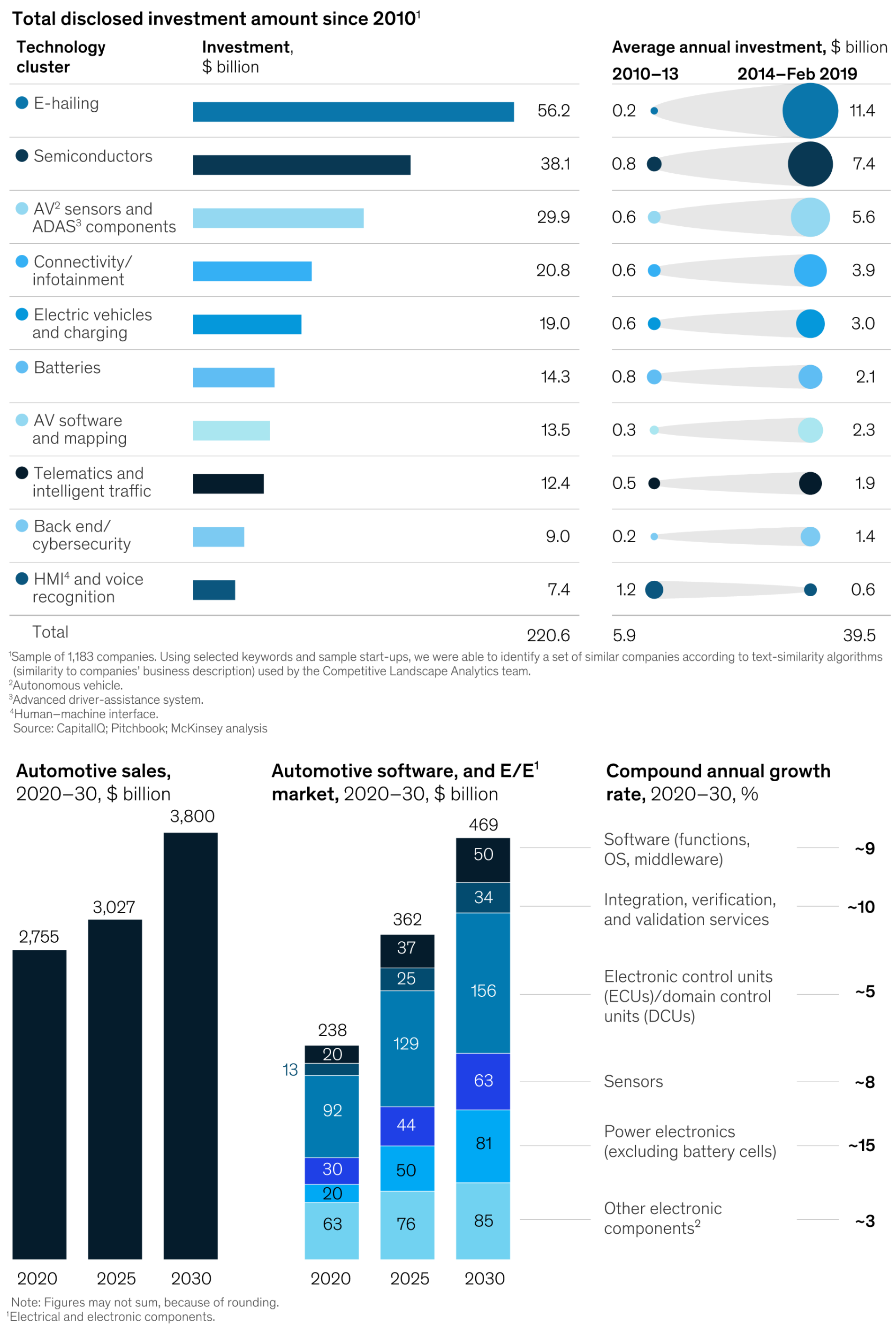


Figure 1.1: Principal investments in the automotive sector (Top), Trends in the market (Bottom) Extracted from McKinsey Analysis.

to store data. When this data is propagated through the circuit, several errors can arise in the application. In the most extreme case, the device can be permanently damaged as effect of those external interventions.

Modern processors and also accelerators, such as Graphics Processing Units (GPUs), which are implemented with the latest integration technologies, can suffer from both types of faults. In fact, the Failures In Time (FIT) rate is larger than it was before in the previous technology scaling generations (see Figure 1.2). Thus, these technology issues originate new reliability challenges for modern devices characterized by a dense number of transistors targeted to operate safety-critical applications. These devices and the target operational environment (safety-critical) demand the development and adaptation of new test and reliability solutions.

More in detail, the trend observed in the scheme of Figure 1.2 suggests that modern integration technologies maintain almost a constant Infant Mortality (IM), but new devices are reaching the end-of-life state before the previous generations (the Component Fault Rate (CFR) period is shorter than in previous integration technologies). Thus, the extension of the operative life in modern technologies requires innovative in-field testing solutions beyond the traditional end-of-line testing, which is performed at the end of the production phase. End of production testing is not affordable anymore in complex devices and becomes a relevant issue in devices targeted for safety-critical applications and expected long-operative lives.

In the safety-critical domain, the reliability becomes imperative, so the compliance with standards of reliability and safety is a must. Most of the reliability constraints and safety development guidelines are included in industrial standards such as the ISO26262 and the IEC61508. These guidelines provide a set of requirements and needs that must be fulfilled in order to maintain optimal levels of functional safety in an application. Moreover, the complexity and the requirements in new applications, currently exploit the usage of (once called!) emerging technologies, such as Field Programmable Gate Arrays (FPGAs) and GPUs.

FPGA technologies are flexible and versatile. However, the development and configuration processes are difficult and complex, so have been historically confined to prototyping and a few critical applications. On the other hand, GPUs are less versatile than FPGAs, but more simple and flexible in programming. In fact, the programming capabilities, performance and internal features of GPUs have evolved to become one of the principal actors used in the development of safety-critical applications involving the implementation of algorithms for image and video processing and more recently AI.

More in detail, currently, GPUs are among the most complex embedded systems available in the market [15] and are now positioned as one of the most popular technologies employed in the development of several safety-critical applications, including automotive and autonomous machines in the industrial sectors. However, these technologies still have several issues in terms of reliability, which are detailed in detail in chapter 2.

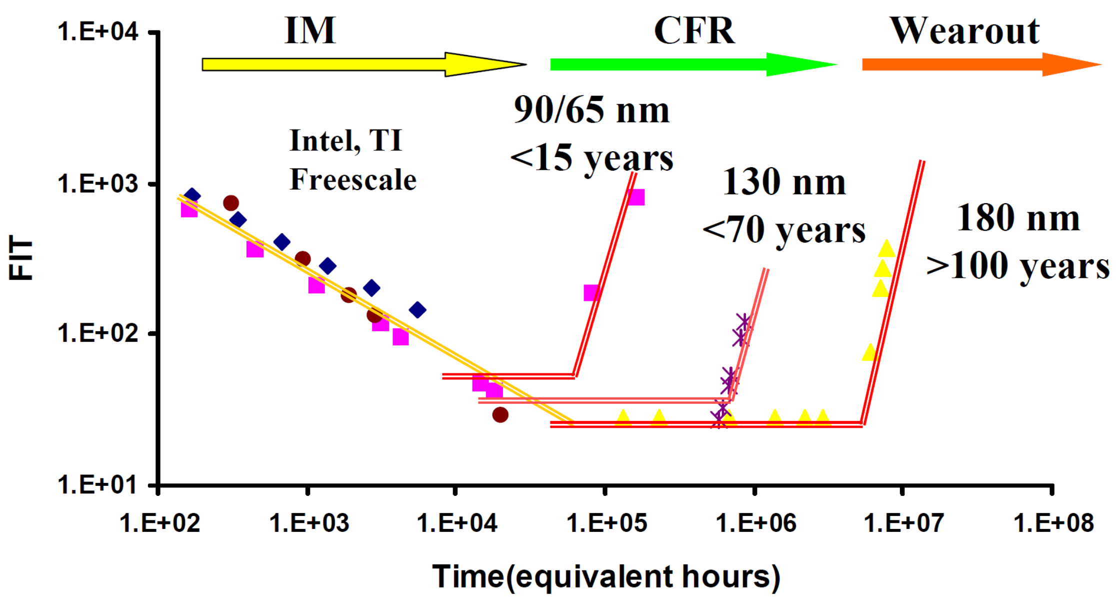


Figure 1.2: A scheme of the trend of FIT rate of transistor integration technologies. Adapted from [16].

Taking into account the reliability issues, technological challenges and the high complexity in GPU devices presented above, the following questions motivate the work on the present thesis:

- What is the individual impact of faults on the internal modules of a GPU during its operative life, and how do the faults affect the execution of applications?
- How to adapt or develop in-field testing techniques based on a functional approach to detect fault affecting internal modules on GPUs?
- What are the rules to consider in the development of in-field testing solutions based on a functional approach for the GPU architecture?
- Is it possible to face reliability challenges by exploring and developing hybrid mitigation architectures for GPUs?, Which are the procedures, and what is the cost of these structures in the GPUs?

These questions served as a guide for the research activities in this work.

1.2 Contribution

This dissertation focuses on three main pillars: *i)* The microarchitectural reliability evaluation focusing on particular modules of the GPU architecture, *ii)*

The proposal of several functional strategies for online test on GPUs, and *iii)* the development of some new hardware structures for in-field fault mitigation in GPUs.

Concerning the microarchitectural reliability evaluation, individual modules of the GPU are evaluated and analyzed for the first time. One particular case in GPU architectures (with respect to the CPU ones) is the presence of task scheduler controllers. This work evaluated the fault sensitivity in the scheduler controller of a GPU. This module is one of the most critical units. Results allow us to support this claim about its fault sensitivity since a considerable percentage (about 25%) of transient faults are propagated, causing fault effects on several of the evaluated applications. Furthermore, other particular modules in the GPU were evaluated, such as the pipeline registers, which are those registers located between consecutive pipeline stages but hidden for the programmer. Other modules were also targeted as part of the sensitivity characterization, including the execution units and the stack memory. However, results also show that the fault sensitivity in these modules is lower than the scheduler controller (in the range of 1% to 15%).

On the other hand, the proposal of several strategies of in-field functional testing and mitigation structures mainly target faults arising during GPUs' operative life.

The proposed functional test methods are based on the Software-Based Self-Test (SBST)[17, 18] strategy that uses the computer programming capabilities of the GPU to design test programs at multiple abstraction levels when possible. It is worth noting that a similar approach based on SBST is frequently used for the in-field test of processor-based systems.

The programming capabilities in a GPU allow the design of SBST programs at a high level. However, restrictions and constraints are fully faced only using a combination of three levels of programming abstraction in the GPU (high-level, pseudo-assembly, and assembly levels).

Three different strategies are proposed for functional tests. The first approach employs a custom approach to combine the operational restrictions of the scheduler controller, in a GPU core, with fault primitives to define the possible test patterns for functional tests. Results show that the proposed solutions can test the module ($\approx$ 100%), hence covering all faults on the targeted fault model.

The second approach of functional testing is based on the operation of multiple test programs to cover different groups of faults. This approach is focused on detect all faults that cannot be observed by the dynamic features of the GPUs, which remain static during the operation of the device. This approach is effective to test several modules inside the GPU and it was validated on the pipeline registers of the GPU core, that handles data and control path information. Furthermore, the results provide main advantages and limitations on developing test programs at high, medium and low abstraction levels.

Finally, the third method is based on a modular approach to develop test programs. This method aim at the exploration of different options in the composition of test programs. More in detail, this approach provides the mechanisms to develop test programs using basic blocks to represent the controllability and observability of faults in a module. Both features (controllability and observability) are combined with the operational parallel features of the GPUs, their main constraints, and a target fault model to develop the generic blocks that, once mapped, describe and implement the test routines for the target unit in the GPU. This approach was validated in several modules of a GPU core, providing adequate coverage of fault and contributing to explore different alternatives in the composition of test programs.

To the best of my knowledge, this is the first time that microarchitectural evaluations and fault injection campaigns on GPUs are performed to measure the effectiveness of functional test solutions based on SBST in GPUs.

The proposed test strategies and mitigation structures are intended to solve testing issues and reliability challenges present in such complex devices.

Other research activities focused on the proposal and evaluation of flexible solutions to mitigate faults in the GPU core. The proposed solutions are optimized combinations of hardware and software structures aiming at the fault detection, fault mitigation, or both in specific modules of the GPU core.

Three flexible solutions are developed and validated using the FlexGripPlus model. the first strategy targets the in-field detection of faults affecting the functional units in the GPU core. The results show that exploiting the high regularity of these functional units, the area overhead of the solution is less than 20%. The second strategy targets the mitigation of faults on the same modules of the GPU core. Finally, the third strategy explore a flexible architecture to provide in-field detection and mitigation of fault on the GPU. Results show a moderate hardware overhead for this solution (<10%). For the second and third strategies, the reliability of the harden module in the GPU reaches more than 40%.

The specific contributions of this research work are the following:

- A complete microarchitectural GPU model (FlexGripPlus), which is an open-source model described in VHDL and based on the G80 architecture of NVIDIA. FlexGripPlus was developed on top of the FlexGrip model [19] and intended to support the analysis of fault effects in GPU cores, perform validation and exploration of software-based techniques, and can be used to implement and evaluate hardware and hybrid mechanisms for fault testing and fault mitigation.

- Several functional test strategies based on SBST approaches targeting the detection of permanent faults in critical units of the GPU architecture. This work introduces for the first time quantitative results about the fault coverage on several modules.

- The proposal of flexible structures for the efficient detection and mitigation of faults, using combinations of hardware structures and software commands.

Several analyzes of reliability and overhead costs are presented for the proposed mechanisms and allows the evaluation of the most suitable trade-offs for implementation.

- The individual microarchitectural reliability evaluation of several data path and control path modules in a GPU device. The evaluation allows us to observe the sensitivity to faults of the targeted modules and to analyze the criticality effect of each module in the operation of a GPU core. Results also support the evaluation of complex applications, such as Convolutional Neural Networks (CNNs) in GPU platforms.

The reminder of this manuscript is structured as follows: Chapter 2 introduces the reliability features of GPUs employed in safety-critical applications and introduces the FlexGripPlus model as a tool to perform reliability research at microarchitectural level.

Chapter 3 describes the main finding of the reliability evaluations performed on several modules of the GPU and also provides the main observations of the reliability evaluation performed on CNN workloads. Then, Chapter 4 describes and reports the main findings of the proposed functional test techniques and solutions for GPU cores employing the SBST strategy.

Chapter 5 describes the flexible strategies developed to optimize the detection and mitigation of faults during the in-field operation of a GPU as combinations of hardware structures and software instructions. Finally, Chapter 6 provides the conclusion of the dissertation and the future works in the topic. The Appendix A describes the microarchitectural details of the FlexGripPlus model and also includes a list of the supported assembly instructions.

Chapter 2

GPUs in safety-critical applications

In the last decade, the GPUs have become the main workhorse in many high-performance and data-intensive applications, mostly present in the High Performance Computing (HPC), industrial, aerospace and automotive sectors [20]. The highly parallel structure of GPU devices combined with their highly flexible programming capabilities perfectly match the requirements of the applications in these sectors, so exploiting the main operational benefits of these products. These modern GPU devices are also known as General-Purpose Graphics Processing Units (GPGPUs).

The GPGPUs are the most flexible and programmable versions of a GPU and are exceptionally effective when used with parallel workloads. In fact, currently these technologies are also solutions in safety-critical applications. Devices including GPUs are considered as one of the most complex Systems-on-Chip (SoCs) ever designed and manufactured [15]. Nowadays, the versatility of GPUs has extended their usage into a large number of domains. Moreover, it continues being the principal device in multimedia and gaming applications.

A GPU oriented to safety-critical tasks can support the development of operations related with the implementation of sensor fusion and deep learning algorithms for AI, such as the NVIDIA Drive [21]. In fact, GPUs can be integrated in the so called Advanced Driver-Assistance Systems (ADAS) in the automotive domain [22] [23] [24]. In this field, the GPUs are integrated in systems devoted to perform several functions, such as Automatic Cruise Control, Pedestrian Recognition and Protection, Forward Collision Warning, Automatic Parking and Automatic Pilot. All these ADAS systems commonly use sensors as inputs, including cameras, radars or lidars, producing a sustained flow of data that must be processed and perform decisions in real-time [25]. GPUs are very well suited for these data-intensive processing tasks and are being increasingly used as feasible solutions by several manufacturers. Moreover, ADAS are also considered as an intermediate step towards semi-autonomous and self-driving cars [26].

Modern safety-critical GPU designs must include strategies to comply with industrial standards, such as the ISO26262 in the automotive domain. These standards require certain conditions to guarantee the correct execution, and a sufficient level of functional-safety and reliability of GPU devices at the end of the production and most importantly, during the operative life of the device. These regulations and standards are relevant for three main factors: *i)* operational constraints of the safety-critical application, *ii)* technology scaling in GPU devices, and *iii)* the architectural features and complexity of GPUs.

Concerning the first point, the proposed fault-tolerance solutions must consider limitations of real-time operation, limited power budget and a high data-intensive processing behavior. Moreover, the possible solutions must evaluate performance limitations in case of in-field testing and mitigation. Finally, the availability of the modules and other resources should be considered during the operation of the safety-critical application.

On the other hand, technology scaling approaches increase performance and reduce size, while maintaining affordable production cost. However, these cutting-edge scales also increase the sensitivity to faults, so new devices might be prone to be affected by faults originated by internal and external factors, such as radiation. More in detail, faults can arise in a GPU after long term operation (caused by *aging* or *wear-out*) or by external factors, including radiation effects, electromagnetic interference and severe variations in temperature and power supply [5], [8], [27], [28], [29]. Thus, fault-tolerance solutions and reliability evaluations become relevant in the safety-critical domain and are important in GPUs devoted to those applications.

Finally, the architectural features and complexity of GPU devices require special fault-tolerance solutions, hence GPUs are dense and complex parallel devices and several singular modules might require special attention. Moreover, the development of ad-hoc solutions would need the combination of suitable strategies from processor-based architectures and new custom approaches of testing and mitigation. Nevertheless, it must be noted that initial designs of these special-purpose processors were developed as dedicated units to support the rendering process conceived for gaming and multimedia workloads, only. Thus, the design and development of GPUs targeted different objective. Moreover, most of these GPU designs included internal modules and functionalities that were not intended for extended verification or targeting industrial certifications. In these cases, the initial GPU devices were designed for a typical commercial operative life, supported by the fast moving trends in technology, and constant changes in the requirements for the gaming and multimedia industries. In fact, the first generations of these devices did not include any fault-tolerant or reliability strategy to manage fault detection or mitigation.

Another relevant feature of GPU devices is the elaborated programming method, which is based on high-level user-friendly programming environments. This method is optimized for performance and fast application development [30], but complicates the development of testing and mitigation solutions. The high-level abstractions of commands, the compiler optimizations and the missing information about the Instructions Set Architecture (ISA) due to proprietary characteristics provoke issues for system and integration companies, which lack of microarchitectural details. In this scenario, modern GPU devices used for safety-critical applications still cannot guarantee the functionality and reliability for long-term operation, so new solutions for the on-line testing and fault mitigation must be developed.

In order to propose reliability and fault-tolerance strategies, a clear description of a GPU is required. For that purpose, this chapter provides the general microarchitectural organization of a GPU and the main relevant modules in its architecture. The microarchitectural organization provides a common terminology used across the following chapters, and also allows the identification of the main operative modules in a GPU. Then, a brief overview of reliability strategies and fault-tolerance methods for GPUs is provided, so allowing to identify actual methods and possible open questions. More in detail, several GPU models and tools are presented. These models are commonly used to explore, implement and validate strategies in reliability research. Finally, the introduction of a new microarchitectural GPU model **FlexGripPlus** is presented. Moreover, the description includes the list of detailed changes and improvements. This model is intended to support the development of fault testing and fault mitigation strategies. For the purpose of this work, the NVIDIA terminology is employed to describe the microarchitectural composition of a GPU.

Section 2.1 introduces the general architectural organization of a GPU and its main relevant internal modules. Then, Section 2.2 provides an introduction to the main reliability challenges in GPU devices and the available reliability solutions. Section 2.3 describes the tools for reliability research. Finally, Section 2.4 introduces and describes the **FlexGripPlus** GPU model as a tool to explore, evaluate and validate methodologies aiming to improve reliability in GPU devices.

2.1 Architectural organization of a GPU

GPUs are specialized parallel processors, which work as special-purpose processors (Hardware Accelerators), in the processing of data-intensive and highly parallel applications. At the beginning (early 90's) these devices were designed and devoted to accelerate 2D/3D rendering and geometry processes for multimedia-based applications [31].

The initial generations of GPUs were modeled and based on the concept of a *Graphics or Rendering Pipeline* [32]. This concept is based on the idea of a single graphics core composed of a set of hardware cores, organized as a pipeline, able to process graphics. More in detail, the graphics core includes a set of fixed-function pipeline stages (Pixel Shading Pipelines), implementing the required operations

(*Application*, *Geometry* and *Rasterization*) to transform 3D coordinates into 2D pixels, which are commonly employed in multimedia operations. The coordination and management of the fixed-function cores is performed through a host using APIs, such as OpenGL or DirectX [33].

The next generations of GPUs started to provide programmable features into the fixed-function cores, so enabling their configuration and extending their capabilities. This programmable trend was exploited in successive generations of GPUs, so evolving from fixed-function pipelines into microcoded processors, configurable processors, programmable processors and finally reaching GPU architectures based on Scalable Parallel Processors [34], which provides the general purpose functionality.

Modern GPU architectures exploit the best trade-off between state-of-the-art approaches in processors' design and traditional parallel architectures. More in detail, these new GPU architectures adopted more flexible processing cores allowing the general-purpose computation as GPGPUs. These General-Purpose GPUs are based on a multi-core design using massive parallel processors to perform highly parallel computations (see Figure 2.1). These fully programmable cores are known as Unified Programmable Shaders, Shader Cores, Single-Instructions Multiple-Thread (SIMT) cores [35] or Streaming Multiprocessor cores (SMs) [34].

In these new architectures, the traditional *Graphic Pipeline* is just a sotfware abstraction and is replaced by one or several SMs able to perform all operations, which were previously performed in the hardware cores of the former GPU generations. The SMs are mainly composed of several scalar cores, also known as Streaming/Scalar Processor cores (or SPs), and can operate sufficient parallel threads in an SM. More in detail, each SM processes the same instruction for several threads which are computed in the available SP cores in the SM.

The execution of a parallel task in a GPU is directly related to the microarchitectural organization and the configuration parameters defined in the task by the programmer. Firstly, the total scalable number of parts of the tasks (*blocks*) is defined. This parameter is internally managed by one or more general controllers in the GPU (*block scheduler*). The main purpose of the block scheduler is the submission and assignment of each block into the available SMs in the system. Then, each SM subdivides the block task into several sets of threads (also called *Warps* or *Wavefronts*) that are operated in parallel on the available SPs. The operation inside the SM follows the Single-Instruction Multiple-Data (SIMD) paradigm [36] or variations, such as the Single-Instruction Multiple-Thread (SIMT), when control-flow divergence among threads is allowed.

On the one hand, a multi-SM GPU architecture provides coarse-grain management of scalable data and task parallelism to execute multiple coarse-grain blocks of the same task, possibly in parallel. Furthermore, in modern GPU architectures, it is also possible the parallel execution of several blocks of two or more tasks. In this case, the blocks are associated to the available SMs, so the GPU follows the

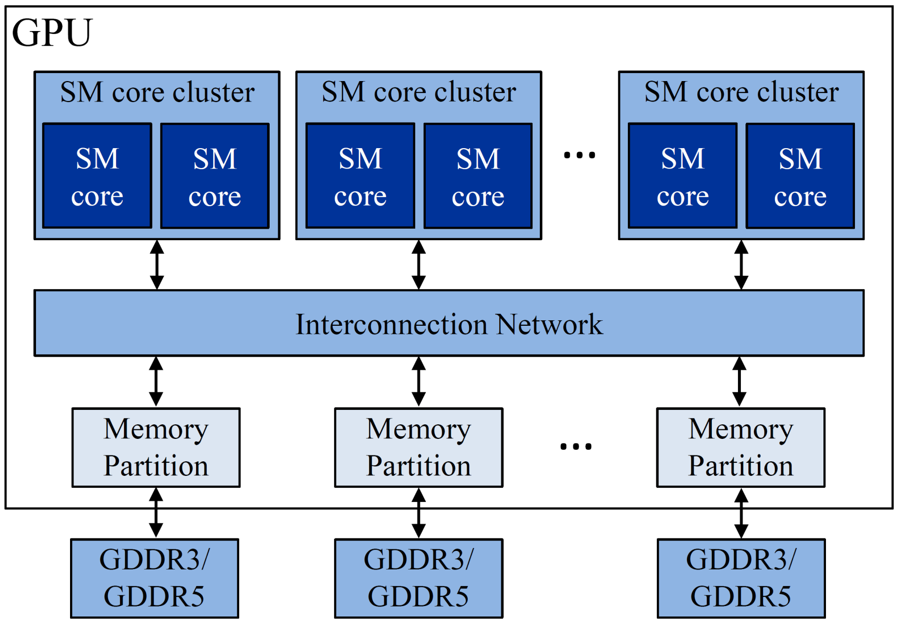


Figure 2.1: A general scheme of a modern GPU architecture.

Multiple-Instruction Multiple-Data (MIMD) paradigm. This behavior is possible due to an extensive interconnection network linking the SMs in the systems and the memory resources in the GPU.

On the other hand, the available SPs in each SM provide fine-grained management of data and thread-level parallelism to execute hundreds or thousands of fine-grained operations (*threads*) in parallel.

It is worth noting that the same essential modules can be located in a GPU device among different manufacturers. These modules include controllers, processing elements or SMs, and the structures for accessing the memory resources. Some designs also include independent accelerators in parallel to the available SMs for specific purposes, such as texture modules, which are mainly devoted to rendering procedures in the multimedia domain. The composition of the processing elements and the dedicated accelerators directly depends on the device's operational targets and potential constraints in terms of power, area or performance.

2.1.1 The Streaming Multiprocessor

The SM is the main operative workhorse module inside a GPU and executes an assigned task in parallel. As introduced above, each SM includes a set of execution units, which are used to perform logic-arithmetical (INT)/floating-point (FP 32/64) operations. More in detail, the SMs are designed using optimized instruction set architectures (ISAs), which are an adaptation of Reduced-Instruction Set

Computer (RISC) instructions into the parallel architecture domain to exploit the SIMD computer taxonomy and other paradigms, such as the SIMT variation by NVIDIA Corporation. Thus, the SMs are special-purpose SIMD (or SIMT) processors (*Engines*) [37]. Internally, the tasks assigned to the SM are divided into multiple groups of threads (*Warps*, *Work-Groups*, *Thread-Groups* or *Wavefronts*) to be executed in the available resources. The size of the groups varies from 32 up to 128 threads across different technologies and architectures. This group size depends on the operative granularity of the structures inside the SM and it is commonly fixed according to the granularity of a given device.

Figure 2.2 shows a general scheme of the architecture of an SM. As observed, the SM is mainly composed of one or multiple scheduler controllers (instruction schedulers) and dispatcher units (1), several execution units (SPs and FPUs) (2), Special Function Units (SFUs) or T-Streams (3), one instruction cache memory (4), associated logic for instruction fetching and decoding (5), one Local Register File (6) and a Shared memory (7), a module for divergence management (8), Load/Store Units to access the memory resources (LSUs) (9) and most recently additional special-purpose accelerators, including Tensor Cores Units (TCUs) (10), targeting the operation of matrices in hardware[38][39]. Furthermore, mostly all SM architectures include vertex accelerators and dedicated texture memory blocks which are commonly used for multimedia applications, but not always exploited in the general-purpose domain. New generations used to combine the texture memory and the Shared memory in order to extend the flexibility of these modules for general purpose applications.

The Register File and the Shared memory are organized as banks, so allowing the parallel access by the processing threads [40], [41]. Moreover, other structures are located inside the SM, including a small local memory, some interconnection networks (Crossbars and Meta-crossbars) and embedded memory structures storing the predicate and the address registers for indirect memory operations. The SM is internally divided in several pipeline stages to take advantage of the instruction level parallelism in the SIMD architecture.

The execution of one task in the SM starts when the block scheduler assign a workload. First, the scheduler controller selects one of the listed warps to operate and dispatches it for execution. Then, one instruction is searched and decoded from cache memory.

After this process, operands and opcodes are selected for each thread in the warp, so reading from any of the memory resources. Then, the warp instruction is issued for parallel processing into the available processing elements (SPs, FPUs, SFUs, etc).

Finally, results are stored in selected memory locations and one new instruction is dispatched. Inside an SM, the processing elements can be organized in parallel starting with groups of 4 up to 128 SP cores [42][43][44]. Each SP is intended to operate one thread of the task on independent data operands. In principle,

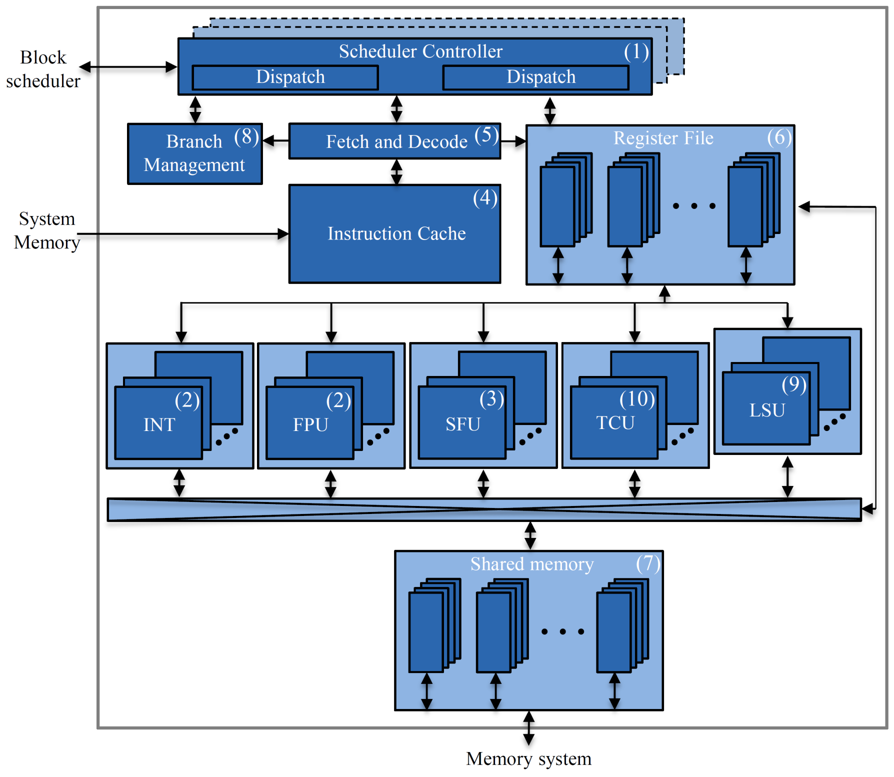


Figure 2.2: A general scheme of the internal organization of a Streaming Multiprocessor in a GPU.

the processing elements and the threads are statically distributed. However, it is also possible to find architectural variations adding crossbar modules, so allowing a dynamic allocation of threads into the available processing elements.

It is worth noting that some manufacturers modify the granularity and composition of the SM, so proposing versions of SMs as combination of two, four or more SMs cores. However, the internal operation remains almost equivalent. In such cases, a new scheduler controller is common for all SMs and it is in charge of starting the execution by submitting the tasks to each SM.

The next subsection describes further details regarding the composition of the most relevant internal modules in the sM.

2.1.2 Execution units

The SPs (also known as *Stream* cores or *CUDA* cores) are the basic scalar units in the SM. These units are employed to perform efficient integer or floating-point operations on one active thread [45], [46].

The microarchitectural implementation of these units is commonly guided by restrictions in terms of power, performance and area [47]. Thus, these units include several optimization strategies to execute operations in a limited amount of clock cycles. More in detail, the overlapping of functionalities and operations in both units (INT and FPU) are the most used approaches in modern designs of these digital arithmetic units [48], [49], [50]. Another approach aims at reducing power consumption by applying circuit gating into specific parts of the arithmetic circuit, while maintaining precision in the operation [51].

The integer units are able to process signed and unsigned arithmetic operations in several formats: reduced precision (8 or 16 bits) and normal precision (32 or 64 bits). Furthermore, the FPUs commonly implement the IEEE 754 floating-point arithmetic standards for 32-bit single-precision and 64-bit double-precision. Moreover, the FPU includes a fused multiply-add (FMA) operation with no loss of precision and maintaining full precision by storing intermediate results, during the product and addition.

Recent versions of SMs also include individual FPUs for single (32 bits) and double-precision (64 bits) operations with independent instructions to access each resource. Other versions of SMs also include scalar programmable units, so allowing inside the SM the concatenation of two single-precision units to perform double-precision operations.

2.1.3 Special Function Units

The SFUs are special hardware co-processors aiming at optimizing operation of trigonometric and transcendent functions, such as *SIN*, *COS*, *SQRT*, *EXP* and *LOG*. These units are often present in SMs, since complex image processing algorithms are typically operated in many GPU applications [52], such as multimedia operations and also in scientific ones. These modules are present in the SM in a lower number than INT and FPU units and are shared among the active threads in the SM.

In principle, the SFUs are designed using two main approaches: *i)* iterative and *ii)* non-iterative. The first approach is devoted to executing iterative algorithms that converge linearly or quadratically to the result, such as the *COordinate Rotation DIgital Computer* (CORDIC), Newton-Raphson and Goldschmidt algorithms, which are more oriented to improve area with acceptable accuracy and precision [53]. On the other hand, non-iterative architectures are based on table-based solutions and polynomial and rational approximations, such as those based on quadratic and bi-quadratic interpolation using enhanced minimax approximations, which are more oriented to optimize power consumption with high accuracy [54][55]. The non-iterative architectures of SFUs are commonly included as part of modern GPU architectures.

2.1.4 Memory hierarchy

GPUs are parallel architectures and are mainly based on big arrays of parallel processing cores. However, each core requires data operands to process. A memory hierarchy in the GPU supports the load and the storage of operands employed by each thread. In these parallel architectures, the massive number of active threads per applications forces the adoption of several levels of memory resources to hide latency when the instructions are executed.

The memory hierarchy of a GPU can be divided mainly in two groups: *"In-chip"* memories, which are located inside the SM core, and *"Out-chip"* memories that are external to the SM and are commonly shared among the SMs. However, it must be noted that this classification is not directly related with the real implementation inside or outside the GPU chip, but provides a hierarchy of the different memories employed for SMs in a GPU.

The *"In-chip"* memories are composed of the Register File, the Predicate Register File, and the Address Register File. All memory resources are distributed among the active threads and the available functional units of the GPU core, so each thread has singular access to data operands. The fastest *"In-chip"* memory resource is the Register File and, as introduced above, is organized in banks and is dynamically distributed during the configuration phase of each parallel program (*kernel*). This configuration defines the number of registers, on each bank, that can be accessed by each thread. Depending on the architecture of the core, the maximum range of registers per thread is defined in the range from 64 to 128 registers.

On the other hand, the Predicate Register File is employed to store the predicate flags per thread as result of logic, arithmetic and setting instructions. These predicate flags are also employed as conditional input in conditional assessment operations. The predicate register file is also organized in banks inside the SM; this module is small in comparison with the register file and depending on the architecture, each thread has access to a variable number from 4 to 12 of predicate registers [56].

The Address Register File is also distributed among the cores during the configuration phase of a parallel program and its main target is the indirect addressing of any out-chip memory, so providing efficient parallelism and maintaining the performance when accessing shared memory resources [57]. Moreover, this unit is intended to avoid race conditions, which are produced when several threads of different warps and blocks require the same memory element, originating multiple repetitive load procedures. The Address Register File addresses the shared, constant and local memories.

Regarding the out-chip memories, these memory resources are employed to share data operands among the warps, blocks, and even SMs in a GPU. The first level of these memories is composed of the Shared memory, which is devoted to store data shared amount warps and blocks in the same SM. The Shared memory has a similar performance of the Register File and is also implemented as a set of banks for parallel access [41][58]. However, it is relatively small in size (16KB-64KB), so it is commonly used when a finite set of data must be processed by several blocks or when minimal latency is required by an application. The most recent implementation of the Shared memory maintains the bank organization, but now it is combined with the first level of cache data memory as a unified module [35].

The Local memory is a small memory employed to store array-type data used in a parallel program. This memory is available for each SM and is mainly used when the program kernel employs any type of consecutive structures, including arrays, vectors or matrices, so if the structure fits in the Local memory, the compiler selects this memory to perform fast operations on consecutive data operands.

The constant memory is an *"Out-chip"* Ram memory storing constant parameters during the execution of a parallel program kernel. This memory is programmed during the configuration phase and it behaves as a read-only memory when executing the program. This memory is shared for all threads in a block and all available SMs can access this memory resource.

Finally, the Global (or Main) memory in a GPU is the slowest memory in the entire memory hierarchy and it is the principal and biggest memory resource of the GPU. This memory commonly employs several ports to serve load and store operations. The size of the Global memory may vary in the range from 2GB to 32GB.

2.1.5 Scheduler Controllers

The schedulers in a GPU are devoted to organize and manage the execution of an assigned task into the available parallel resources. This modules also submit and trace the operations of the parallel tasks.

There are two main types of schedulers inside a GPU. The first type is the general scheduler (also known as the Work Distribution Unit [40] or Block Scheduler controller), which distributes and traces the blocks (big portions of a parallel program) into the available SMs of a device.

The general operation of the scheduler controllers starts after the GPU configuration. The block scheduler distributes the big portions of the parallel program (blocks) across the available SMs in the system. Then, one or more local warp scheduler controllers (on each SM) distribute and group the threads as *Warps* and then submit them to the functional units. Once a warp finishes the execution of one instruction, a new warp and its instruction is loaded and processed. This operation is repeated until the entire parallel program is executed. Then, a new parallel program is loaded and the previous steps are repeated. It is worth noting that most SMs are able to execute up to eight blocks per SM, so the change of warp context switching is mostly managed by the local controllers.

The distribution considers the total number of blocks to operate and the available SMs in the system, so several static distribution policies can be found in these controllers and are used to manage the operation of an application, including Round-robin, Least Recently Used (LRU) [40], and custom policies, including FAIR [59], Thread block-based CAWS [60], and others [61]. Modern versions of these controllers in GPUs include support for concurrent kernel execution and can distribute two or more applications on the available SMs, so behaving as MIMD machines.

The other type of controller is located inside the SM and its known as *Warp Scheduler Controller*. As introduced above, this controller manages the assigned blocks and splits them in *Warps*. Then, these warps are submitted for execution to the available parallel functional units. This scheduler tracks information (number of active threads and actual status of the warp) about the executed warps and uses special purpose memories to store the information per warp. The scheduler controller employs all previous information to perform the context switching during the execution of one or more blocks in the SM. This controller also manages the intra-warp divergence, so controlling the several instruction paths in the threads of a warp.

The warp scheduler controller also includes similar scheduling dispatching policies as the block scheduler controller, from static policies based on round-robin and custom approaches, up to dynamic and more elaborated ones (i.e., *GIGAThread technology* by NVIDIA) including the management of data-hazards, instruction-type constraints, and thread conflicts [35] [62].

In case of static warp distribution policies, a fixed algorithm is employed on each controller to dispatch workloads and verify the correct execution. On the other hand, dynamic distribution policies are intended to reduce the latency in the execution of warps by duplicating the number of schedulers in the SMs, so the internal distribution of the SPs (INT and FPUs) contributes to optimize the execution of the warps. This duplication allows the execution of half warps (executing two instructions in the same instruction cycle) and splitting the available functional units between the controllers.

Recent versions of the scheduler controller also provide control efficiently. In case of intra-warp divergence (several threads from the same warp have different paths and execute different instructions), these warp scheduler controllers have a fine-grain control in the threads, which is obtained by storing additional information per thread, so allowing the swapping of thread operations of several paths during the warp execution [44].

Finally, recent architectures of warp scheduler controller also simplified the management procedures and replaces those complex algorithms inside the hardware controllers by compiler optimization efforts in software. In this case, strategically placed instructions are used by the warp scheduler controllers to manage the expected latency in basic blocks of instructions of the executed parallel program.[35]

[63].

2.2 GPU devices Reliability

At the beginning, the traditional end-of-manufacturing testing was sufficient solution for initial generations of GPU devices. However, new generations of GPUs are more complex and transistor-dense devices, which are used in new applications of the safety-critical domain, so those traditional techniques are not sufficient any longer to guarantee the correct in-field operation.

Nowadays, modern GPU architectures employ several fault-tolerance strategies to increase reliability and also satisfy the industrial reliability and functional safety standards. These strategies can be mainly divided in three types: *i)* design decomposition, *ii)* hardware techniques, and *iii)* software techniques.

In the first case, the manufacturer and designer companies balance the design and select, for a given product, the specific reliability and testing goal according to the industrial standards [64, 65]. These standards suggest countermeasures to guarantee the correct operation of the device during the production stages and during the in-field execution. Modern manufacturer strategies also used to consider the feedback from costumers in order to improve reliability features [66]. In some reliability and safety designs for automotive, for instance, the manufacturer takes advantage of standards restrictions by decomposing the design (ASIL decomposition of the design) [67].

In this decomposition approach, the systems can be decomposed into several modules, analyzed individually and finally protected [68]. However, complex designs and systems used to include modules from other manufacturers. Generally those modules without detailed structural information, or those that cannot be easily modified (such as GPUs) remain unprotected, but huge effort is used to increase the reliability of the surrounding components, so fulfilling the global reliability restrictions. At the end, the entire design matches the required ASIL level, but internally not all the modules achieve such a level. This method can be used for huge systems involving GPUs, but the local reliability of the GPU remains unvarying.

The second method directly improves the reliability of the GPU by adding modules and hardware structures into the design phases.

The use of fault testing and mitigation strategies, such as Error Correcting Codes (ECCs) [58], [69], can extend the reliability of several components of the memory hierarchy of a GPU, including the Register Files, cache memories and RAM memories. However, this solution is not feasible for execution units and controllers in the system. In those cases, combinations of the state-of-the-art techniques are usually employed. Authors in [21], proposed an in-system-structural-test (IST) solution based on the adaptation of the original GPU design by the refinement of the internal IP cores to increase the fault detectability. Then, a large set of testing structures (scan chains, MBIST, XBIST, LBIST) and additional controllers are included to test and verify the correct operation of the device during the in-field operation. The new structures were included considering power-gating and phase-staggered clocks to reduce the overhead of power budgets. These in-chip mechanisms also provide debug and diagnosis features when faults are detected. It is worth noting that these solutions require huge efforts during development and high costs in terms of area overhead with a moderate cost in performance.

Software mitigation solutions can also be adopted in GPUs. These solutions are based on code adaptations to mitigate fault effects using functional and algorithmic methods [70], [71].

The functional methods are based on traditional redundancy mechanisms and selective hardening solutions [72], such as Duplication with Comparison (DWC), Triple Module Redundancy (TMR) or custom-optimized techniques [73], [74]. On the other hand, the algorithmic methods are based on code modifications applied to specific algorithms to provide hardening and include the development of software-based ECC [75], Algorithm-Based Fault-Tolerance (ABFT) for FFT [76], wavelets [77] and Matrices operations [78], [70], [79, 80]. Nevertheless, these methods are limited to linear algebra operations [81] and the performance degradation and memory over-head can become relevant, so compromising the real-time constraints of some GPU applications. In contrast, there are a few proposed solutions of functional testing on GPUs, which are mainly adaptations of processor-based strategies such as pseudo-random or March algorithms [82]. However, the biggest constraint factor in the development and adoption of these solutions is the limited architectural details and available information of these commercial products, so limiting the possibility to develop methodologies and strategies. The other factor is the missing fault observability to validate techniques using commercial devices.

Further details and an extended description of the fault-tolerance strategies is presented in the chapters 3, 4 and 5.

The brief introduction of the strategies shows that protecting and increasing the reliability features in GPU devices is not a simple task. In fact, most efforts to increase reliability are mainly developed by the design and manufacturing companies which have access to all descriptions and details for that purpose. Furthermore, these solutions cannot be always used and are limited by the availability of those fault-tolerant structures in a device. Moreover, the independent development by system integration companies is not always possible.

The reliability in GPUs is motivated, as introduce above, by the mandatory need of guaranteeing the correct operation of safety-critical applications. More in detail, there are three main methods to evaluate the reliability of GPUs. Two of these can be also employed to explore and validate testing and mitigation strategies. The three methods are: *i)* laboratory experiments, *ii)* software-based experiments, and *iii)* fault simulation experiments.

In the first case, laboratory experiments are performed by exposing real GPU devices to a external source that provokes fault effects in the operation of the device, such as radiation. Then, special applications or benchmarks are employed to evaluate the fault sensitivity of a device. The results are then examined and scrutinized in order to identity, characterize and theorize the main causes of the observed faults and errors in the applications. This reliability evaluation is performed in costly specialized equipment in facilities and laboratories with the available radiation sources and is preferred by the academic community and the industry to evaluate and analyze the effects of faults in new devices and those implementing mitigation strategies. It is worth noting that similar procedures can be applied to other external sources causing fault behaviors in the GPU, such as electromagnetic effects.

In the second case, the software-based experiments are also performed on real devices. In this method, the procedure consists on injecting faults by corrupting the executed instructions of an application, so mimicking the effect of a fault in a module of the device. This method allows the analysis and the evaluation of fault effects on several data-path modules of a GPU and it is a valid and comfortable option to evaluate or explore software-based hardening techniques in GPU devices. However, this method presents difficulties in representing faults (allocation of instructions behaving as faults) on specific modules and specially on those devoted to perform control and management functionalities.

Finally, the third method is based on fault simulation experiments. This method is performed using computational models representing the behavior, functionality, and architectural details of GPUs at several abstraction levels. Furthermore, this method is cheap and is the only method allowing the exploration, evaluation, and validation of testing and mitigation strategies at software, hardware, or optimized combinations of both without implementing the GPU device in silicon. This method is flexible and allows easy adoption when the main target is exploring testing and mitigation strategies and several design tools in the market, including functionalities and utilities, allowing reliability analysis.

This method can also be employed in the reliability evaluation, but special considerations are required in the selection of representative fault models and constraints of the computational GPU model. Moreover, this method demands a high computational power for large analyzes. Nevertheless, its possible that this method can provide accurate and realistic behavior of reliability evaluation applied to individual modules of a GPU.

A deep analysis and evaluation of the previous methods and the main targets of the research work show that a feasible method to explore, evaluate and provide new testing techniques and mitigation solutions for GPUs requires the use of fault simulation and computational GPU models by the high advantages in flexibility and the fine-grain control during the experiments. However, the other methods are not discarded and are considered as supporting tools in the development of the research work.

The next section introduces an overview of the available GPU models. Then, the developed FlexGripPlus GPU model is introduced as a tool to perform reliability analysis, to explore testing techniques and validate mitigation strategies for GPUs.

2.3 GPU models as tools for reliability research

Currently, the exploration, development and research of innovative solutions for testing, reliability evaluation and fault mitigation in GPUs are based on the combination of several strategies.

The development of those procedures requires detailed information regarding structural details and are supported on representative computational models of a device. These models are employed to develop and validate the proposed strategies, including those focused on in-field testing and mitigation of faults. Thus, it is common that in these stages, engineering teams in companies and researchers employ those models to accelerate the design and testing, so evaluating initial solutions before to continue with implementation stages. Moreover, multiple levels of abstraction are also employed for these purposes, combining several models, tools, emulation devices and real devices to analyze and validate the potential solutions.

In the GPU field, the few available models are mainly employed at the academic level to propose and explore fault detection and fault mitigation strategies. A set of works on structural descriptions of GPUs (simulators) are also employed to select the best execution trade-off of an application and also to evaluate execution phenomena on this parallel architecture [83][84].

As introduced before, these models are relevant tools and are important in GPU research topics from parallel programs or parallel architectures. Moreover, the high variety and different architectural abstraction-levels allow the classification of these models as Behavioral, Structural and Microarchitectural. Currently, other hybrid approaches, to validate and verify strategies, include software profilers and instruction-level fault injectors.

The behavioral models are used to perform the operation of a GPU without considering the device's internal organization. These are used in the first steps of the design phase and are rarely available. Moreover, the limited number of architectural features on those models limit their use when reliability and fault testing are the primary goals. Other classifications are discussed in the following sections.

2.3.1 Functional models

These models are developed to mimic the functionality and operation of a GPU. Moreover, several models include a behavioral operation of the modules integrating the organization of a GPU and the interconnections and transactions among their internal modules. Nevertheless, these models used to present limited microarchitectural details, such as the internal operation of execution units. However, the main target of these models is not to include such implementation details. Furthermore, such amount of details extend the simulation time of the model adding irrelevant information for the objectives of this abstraction level.

There are several GPU models at the functional level, most of them described in high-level programming languages and using custom frameworks or also combining commercial tools to perform the simulation. Multi2Sim [85] is a simulation framework devoted to explore and validate heterogeneous hardware designs before they are physically manufactured. This framework includes support for multi-hosts and multi-GPUs, so designers and computer architects can modify and propose new architectures or variations. Moreover, the framework also includes several benchmarks for comparison and performance evaluation purposes.

Another widely-used GPU model is GPGPU-SIM [86], which is a functional and structural simulator of GPUs. This model allows the designer to analyze, at fine-grain level, the execution of a kernel in a selected module. It was originally developed to support the design effort of applications and architectures. This model implements the midrange architectural structure and supports as inputs one unaltered version of a CUDA kernel description. This simulator is the closest to the real devices implementation by the behavior presented during execution. The Barra functional simulator [87] models the NVIDIA Tesla GPU architecture employing the UNISIM framework combining the cycle-based and the transaction-level modeling (TML) approaches. It is based on two layers. The first layer represents the CUDA software stack, meanwhile the second functionally simulates the GPU. The gputejas project [84] is an ava-based parallel GPU simulator employing a fast trace-driven simulation. Its main advantage is the parallel processing of the tasks employed for the simulation effort, which reduces the execution time in up to 17.33 times with respect to a serial operation. However, this is not a purely functional simulator. This model uses the Ocelot framework to execute CUDA applications and then simulates the traces in order to establish timing characteristics. Attila [88] is a software-based simulator of GPUs. This is a detailed cycle-accurate and execution-driven GPU simulator. Moreover, it is highly configurable. However, it only supports the OpenCL programming model. Finally, GEM5-GPU [89] is an extension of the GPGPU-SIM model. This simulator mainly represents an heterogeneous architecture, including a CPU core as host of the system.

Unfortunately, all previous functional models are not intended to perform fine-grain reliability evaluation. In fact, most of them are mainly developed to represent the functionality of a GPU device, so these cannot be used to perform reliability evaluation or evaluate testing strategies at microarchitectural level. For those purposes, custom frameworks for fault injection and evaluation are developed, such as GUFI [90]. In fact, the addition of fault injection frameworks is challenging in some models when considering that changes into critical descriptions of a model may activate undesired internal operations or create incorrect simulation procedures.

2.3.2 Microarchitectural Models

These models include more architectural details that the functional ones and commonly a hardware description language (i.e., Verilog, VHDL) or any synthesizable language (i.e., SystemC, System Verilog) is employed for their description. In this level of abstraction, the model is composed of almost all required modules, specifications and characteristics of a real device. The model is generally described at Register Transfer Level (RTL) or gate level, which is preferable since it provides a detailed representation of the internal structures. However, such models require higher computational effort for simulation or emulation than the high-level ones.

In practice, the number of available GPU models to support reliability analysis is limited. On the one hand, there are GPU models described as a combination of multiple levels of abstraction, such as MIAOW [91], which includes a partial RTL description (mostly execution modules) of the AMD Southern Islands architecture. However, the control modules are described in a high-level programming language and are not synthesizable.

On the other hand, there is a set of fine-grain GPU models available: IPPro [92, 93], FGPU [94], Simty [95], Nyami [96] and FlexGrip [19]. The first four models were designed targeting FPGA platforms using custom GPU architectures and instructions sets. Thus, their custom architectures and implementation target technologies would limit the capability for reliability analysis, especially when commercially available GPU devices are concerned. On the other side, the architecture implemented in the FlexGrip model is closer to commercial devices by NVIDIA. FlexGrip also partially supports the binary compatibility with the NVIDIA G80 instruction set, and it is compatible with the CUDA programming environment. Thus, in principle, the FlexGrip GPU model is clearly a good candidate as tool to implement, design and validate testing, mitigation and reliability evaluation by the direct similitude to commercial devices.

Unfortunately, a deep analysis revealed that the original version of FlexGrip had some restrictions related to technology dependency, instructions format support, and compiler compatibility, limiting the development and study of new applications to support reliability analysis. For this purpose and in order to overcome the limitations, several improvements were performed and in the end, a new version of the GPU model called **FlexGripPlus** was released. The next section describes the main advantages of **FlexGripPlus** and the performed changes.

2.4 From FlexGrip to FlexGripPlus

The FlexGripPlus GPU model [97] [98] is a new version built on top of the FlexGrip model [19]. This version is an extension of the original one and corrects several limitations and bugs observed after a detailed inspection of the hardware modules and the programming capabilities.

This GPU model was developed to support the fine-grain reliability evaluation. In fact, the main advantage of this model is that fault effects can be explained based on conclusions inferred from observed effects on individual modules of the GPU core, since manufacturers do not specify the internal composition and the conventional experiments performed by scientists and researches cannot focus on specific sub-modules. Thus, the development and support of a microarchitectural GPU model, including additional details, is crucial to evaluate reliability and fault sensibility on internal modules. Moreover, FlexGripPlus is a suitable as tool to validate the development of test and mitigation strategies on GPUs with the advantage that hardware overhead, performance and power metrics can be determined.

FlexGrip (the original model) is based on a Streaming multiprocessor (SM) mainly composed of five stages of pipeline (*Fetch*, *Decode*, *Read/Issue*, *Execute*, and *Write-Back*). The model is configurable and the total number of parallel Streaming Processors (SPs) in the *Execute* stage can be selected, before synthesis or simulation, among 8, 16, or 32 parallel cores.

The FlexGrip model also includes a basic memory hierarchy composed of the system memory, the Register File, the global (or main) memory, the shared memory, and the constant memory. It is worth noting that cache memories are not included in the model. However, other minor memories, such as the Address Register File and the Predicate Register File, are included in the model and distributed among the available SP cores. Regarding control units, the model includes two task schedulers (Block Scheduler and Warp Scheduler), both adopting a round-robin algorithm as a distribution policy to manage intra-warp divergences caused by a branch instruction and also allow the control of kernels composed of several blocks in case of a multi-SM configuration. A custom branch module is implemented with a stack memory (denoted as “Divergence Stack”) to handle up to 32 levels of divergence. Please note that there are registers between different stages of the pipeline. One structural comparison between the FlexGrip model and the commercial GPUs (from NVIDIA) indicates that both share the same basic functional structures, including the block and warp scheduler controllers, the Register File, the parallel execution units, and the pipeline stages. Nevertheless, parts of the memory hierarchy in FlexGrip differ from the one in the commercial devices by the missing cache memories. Moreover, FlexGrip has no floating-point modules or special-purpose accelerators. These structural limitations in FlexGrip can affect the adoption of the most recent applications and limit testing and mitigation techniques. However, this model is suitable as a base model to perform analysis of reliability and fault effects on data-path and control-path modules that can still be meaningful with respect to the commercial devices considering the architectural similarities between them.

One of the main advantages of the original FlexGrip is the structural similarity with the G80 architecture of NVIDIA and the compatibility with its commercial tool (CUDA). Moreover, the complete description of the GPU core, including the controllers, in RT-level provides the possibility of analysing and observing fault effects on those critical modules of these parallel architectures.

The development of the new version take into account the original structures and the overall architecture, so instead of changing the complete description of the model, it was completed and extended so providing a better understanding of other structural behaviors.

The major limitations of the original FlexGrip include the dependency on a specific technology (the Xilinx FPGA library), the limited number of supported instructions, and the partial compliance with the CUDA programming environment. The first step was the verification of the correct operation of each initially supported instruction. Then, the newly added instructions have been carefully verified for correctness. To achieve this, a methodology to improve the FlexGrip model and produce the FlexGripPlus version has been used based on the following steps:

- Rigorous structural analysis of the FlexGrip model
- Development of basic test programs to verify each instruction supported by the design
- Simulation and interpretation of results
- Definition of potential corrections
- Application of the revisions and verification with previous programs
- Development of new applications based on typical applications and workloads for validation purposes.

In the beginning, the analysis of the internal structures in FlexGrip revealed that some modules were incomplete, including the distribution controller of the Register File (RF), the Decode stage, and the Execution stage. These incomplete modules can lead to incorrect results under certain circumstances and must be fixed. Then (in step 2), a large number of specific applications were developed to generate all the potential format of the supported instructions and verify the correct execution of each instruction. Each program was developed using the CUDA environment when possible, or directly at the assembly level (SASS).

In this step, the incompatibility issues between the instruction decoding implemented in FlexGrip and the binary code generated by the CUDA environment have been identified and fixed. Furthermore, for each modification, steps 3 to 5 have been repeated to verify the correctness of the new implementation. This methodology should serve as a guideline if the model has to be further improved and new modules have to be added. More in detail, in the process of implementing the FlexGripPlus from the original FlexGrip, the introduced changes can be divided into three groups:

- Technology dependency
- Instruction format support
- Compiler restrictions.

The next subsections provide the actions performed on each change.

2.4.1 Technology dependency

The FlexGrip model was initially designed as a soft-core targeting a Xilinx FPGA, and some internal modules, such as the memories, were automatically generated as IP cores using the Xilinx System Generator tool. The descriptions of these IP cores are not human-friendly to read or analyze and significantly limit the possibility to perform the required gate-level analysis when performing modifications or when validating any proposed mechanism for fault testing or mitigation.

In FlexGripPlus, each module was carefully modified by removing any reference or dependency on specific technology libraries, replacing them with equivalent generic descriptions. Through this process, the names of signals, interconnections, and modules were clarified to simplify the analysis on each signal when performing any reliability evaluation. In the end, about 40% of the modules were modified for this purpose. It is worth noting that after this procedure, the model can now be easily imported into different simulation environments, including *ModelSim*, *QuestaSim*, and *Xcelium Parallel Simulator*. Moreover, the removal of the technology dependency from the model allows mapping the model into other platforms or technologies that were not previously supported, such as ASICs. The FlexGripPlus model has been synthesized using proprietary or independent technology libraries, such as the ASIC 45nm OpenCell [99] or the 15nm OpenCell [100] libraries.

2.4.2 Instruction format support

The original FlexGrip implementation was intended to be compatible with the CUDA programming environment through SASS instructions. However, thanks to a detailed process of simulation and analysis, using a set of test programs to check each instructions, it was possible to observe that some internal modules in control- and data-path units, such as decoding logic, intermediate registers, and block interconnections, were missed or partially implemented.

Hence, we started from the initially supported instructions in FlexGrip, and each instruction was verified against the NVIDIA CUDA programming environment. Unfortunately, NVIDIA has not officially released the SASS op-codes (i.e., the instruction formats and the binary meanings) so, a big effort was performed on decoding most instructions (the op-code and formats) based on reverse engineering, the CUDA binary tools, namely nvcc and cuobjdump, and also supported on some independent projects[1] [63].

Additionally, multiple verification benchmarks were designed targeting specific instructions, so forcing the compiler to produce the target instruction op-code under various formats. The previous process requires the explicit combination of high-level routines and pseudo-assembly commands (PTx). Moreover, some issues (such as the compiler optimizations, the out-of-order mechanism in the compiler that changes or removes instructions) required the explicit addition of numerous output memory locations in the micro-kernels.

All required changes (e.g., descriptions of missing registers, connections, combinational logic, or the correction of incomplete modules) were introduced in the FlexGripPlus model to fully support the set of instructions with all the format variations, including a first version with 28 instructions and 74 formats. Appendix A shows the supported control-flow, arithmetic, logic, data-handling and memory instructions and their formats.

In the end, about 4.8% of the code in FlexGrip was modified for this purpose. Most of those modifications were performed in the Decode and Read stages of the pipeline in the SM.

2.4.3 Compiler restrictions

The primary approach to develop new applications for the FlexGrip model was based on the CUDA programming environment. After the instruction compatibility issues mentioned above were fixed in FlexGripPlus, there was still a gap between all the instructions that can be generated by the CUDA compiler and the instructions supported in FlexGripPlus. Hence, three software tools have been developed to check and preserve the compatibility between the CUDA programming environment and the FlexGripPlus implementation.

Firstly, an assembly language checker tool, named "SASS checker", has been developed to check the binary code generated by the CUDA compiler nvcc against the instructions supported by FlexGripPlus. If any unsupported instruction is generated by nvcc, then the second tool, an assembly code writer tool named "SASS

[1]https://github.com/laanwj/decuda

parser", can be used to convert any unsupported instruction into one or more supported instructions with the equivalent operation. However, in case of failure of such a conversion attempt, the tool reports an error, and the user manually modifies the source code to avoid the generation of unsupported instruction.

Thirdly, a memory configuration tool named "Index corrector" can verify and correct the mismatches in the addressing indices, which are used to address the memories during the execution of an application. The index corrector is required considering that the locations to store the variables in a program are managed and decided by the compiler tool without any significant user intervention. However, in multiple applications, it is required to verify and correct the indices used to address the memories of the system since the CUDA compiler always prioritizes the performance execution over any explicit memory placement. Thus, the compiler chooses the best trade-off between performance and memory operands in an application to store elements in any of the available memory resources, managing the global, shared, local and constant memories. These three tools successfully contribute to reducing the time to develop or adopt new applications for FlexGripPlus. These three tools are also used in step 6, mentioned previously, to design new applications aiming at verifying the operation of FlexGripPlus.

Two main extensions were added to the FlexGripPlus model and in both cases referring to execution units. As introduced before, the original GPU model was only able to perform integer operations, so reducing the possible development of new applications or the development of complex strategies for fault testing or mitigation.

The first extension targets the addition of floating point units (FP32) in single precision format as suggested in the G80 architecture [42]. The module is implemented as a combination of combinational and sequential modules able to process up to three operands. More in detail, the FP32 is composed of four modules: addition, multiplication, fused multiply and addition and conversion. These modules are located in parallel and some additional multiplexers, registers and a small controller selects the operation depending of the incoming instruction.

The FP32 module is devoted to perform the addition (FADD), subtraction (FSUB), multiplication (FMUL), multiplication and addition (FMAD and FFMA), division and conversion (float to integer, integer to float). The *Decode*, *Read/Issue* and *Execute* pipeline stages were modified to include support for the instructions and functional units. The FP32 were also located in parallel to the existing integer cores (SPs) and the flexibility features of the original model were also extended to this module, so in FlexGripPlus it is possible to select among 8, 16 or 32 FP32 units.

The second extension targets the addition of the Special Function Units (SFUs), which are devoted to perform the trigonometric and transcendental operations in the core. The design of this module followed the bi-quadratic approximation approach [54, 55], as suggested in the G80 architecture guidelines [42]. This module can execute the base logarithm (LG2), base 2 exponent (EX2), sin (FSIN), cos

Table 2.1: Comparative analysis of main features in the FlexGrip and FlexGripPlus models

	FlexGrip	**FlexGripPlus**
Instructions	• 28 instructions (partially supported)	• 52 instructions fully supported • 85 formats of instructions verified
Programming environment	• Partially compatible with the CUDA programming environment. • A manual mechanism to compile the CUDA (.cu) file and adapt to the model	• Partially compatible with the CUDA programming environment. • A tool to translate the CUDA (.cu) description into the final binary file used in the model • A tool to develop applications at the assembly level (SASS) • A tool to verify the compatibility of the generated assembly code.
Applications	• Only 5 benchmarks	• More than 20 verified applications
Simulation or Implementation platforms	• Simulation (ISIM, VIVADO simulator) • FPGAs (Xilinx)	• Simulation (ModelSim, Xcelium Parallel Simulator, ZamiaCad) • FPGAs • ASICs
Memory management support	-	• An additional support to manage the address register file employed to access the Shared memory
Support for Floating-Point operations	No	Yes: Floating point in single precision (FP32)
Support for trigonometric and transcendental operations	No	Yes: LOG2, EXP2, SIN, COS, RCP, RSQRT
Documentation and manuals	No	Programmer's manual and Quick Starting Guide

(FCOS), reciprocal of the square root (RSQ) and the reciprocal (RCP) instructions. The main advantage of this module is the overlapping of functions, so all operations are executed using the same sub-modules that are based on memories and a combinational tree, so reducing the hardware overhead and maintaining the precision of each operation. The SFU module, as the FP32, was integrated into the GPU model to exploit the configurable features before simulation or synthesis. Thanks to the development of both extension, it was also possible to develop an alternative SFU module integrating other architectures devoted to mobile applications [101]. The FlexGripPlus model can use any of these two SFU versions.

Table 2.1 shows a comparative analysis of the main features of FlexGrip and FlexGripPlus. Furthermore, it lists the introduced modifications, their effects and main improvements mentioned above. It is worth noting that one additional instruction (ADA) was implemented, which is fully compatible with the CUDA programming environment. The main purpose of the ADA instruction is the management of the address register file modules to address any external memory resource (such as shared or constant memories) to the GPU core.

The FlexGripPlus model is open-source and available for download from two

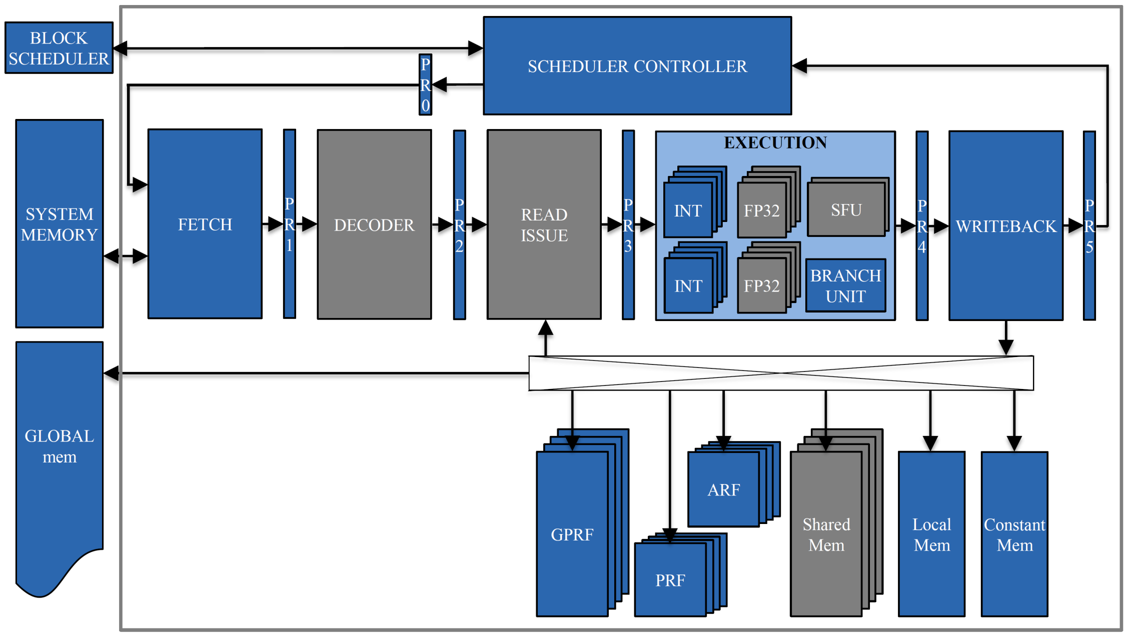


Figure 2.3: A general scheme of the internal organization of one SM core in the FlexGripPlus GPU model. The modules in gray represent the added modules or modified from the FlexGrip model

open repositories [2] [3]. Furthermore, the manuals for programmers [4] (containing all supported instructions and instruction formats) and the user's guide of the model are also available for download.

Fig 2.3 depicts a general scheme of the internal organization of one SM in FlexGripPlus. As can be observed, all modules in gray are those modules and submodules modified, debugged or added into the GPU model.

Besides the improvements made towards increasing the set of supported instructions, another significant improvement in FlexGripPlus is the technology independence. This independence allows the usage of the model at a lower level without any limitation related to the targeted gate library and the simulation tool. More importantly, in this way, it is possible to investigate the fault effects with fault injection techniques targeting specific modules.

Although FlexGripPlus implements the NVIDIA G80 microarchitecture (as inherited from the original FlexGrip model), it includes all principal and critical modules, which are also present in modern GPGPU architectures. Moreover, the

[2]https://github.com/Jerc007/Open-GPGPU-FlexGrip-

[3]https://opencores.org/projects/flexgripplus

[4]https://doi.org/10.5281/zenodo.3819313

compatibility of the model with commercial programming tools allows the use of the same tools as in real application development (with minimal limitations).

2.5 Conclusions

This chapter described the fundamentals of GPU devices used in safety-critical applications and the main challenges and issues in performing testing and mitigation of faults in modern GPU technologies. Moreover, this chapter also introduced the microarchitectural fundamentals of the GPU architecture, their main architectural composition and the main reliability challenges. Then, several tools were introduced, which are commonly used in research for design, verification and validation of new GPU architectures and also to evaluate fault-tolerance solutions in GPUs.

Finally, this chapter also introduced and depicted the development of an open-source microarchitectural GPU model (**FlexGripPlus**), which is a tool to support at fine-grain level of reliability evaluations experiments and also the development, implementation, validation and evaluation of in-field functional testing techniques and mitigation strategies.

Chapter 3

Analysis of transient fault effects in GPUs

Nowadays, GPUs are among the most complex devices available in the market. The high density of transistors and the high complexity in their implementation includes several reliability challenges. Moreover, as introduced in Chapter 1 and 2, modern GPU devices are prone to suffer of transient fault effects, which can be critical during the operative life of a device devoted to safety critical applications.

Since GPUs are moving to markets where reliability is crucial, GPU vendors have worked to design platforms compliant with reliability standards (such as ISO26262). In the meanwhile, the academic community has been carefully studying GPU reliability with different approaches, including software-based and emulation-based fault injection [102, 103, 104, 105, 106], and beam experiments [107, 108].

In several works [10, 109, 110], the authors analyzed these phenomena reaching some relevant conclusions that can be applied from the design up to the system integration levels. They identified sensitive modules in the memory hierarchy of the GPUs, such as the main memory, the shared memory resources, the register file and the caches, that compromised the execution of applications [111, 112]. In [113], the authors shared an introductory evaluation of the register file of a GPU core using a detailed micro architectural evaluation on several representative applications. Authors in [114] analyzed the effect of optimizing the average time to access the memory resources and observed the increasing on the sensitivity by the use of additional resources and some unprotected registers.

Other works explored the reliability and fault sensitivity on the execution cores of GPUs [115]. In [116], the authors explore the effect of hardware and software implementations of Floating-point units in soft-GPUs. The conclusions showed that hardware modules can present fault sensitivity of up to 18 times in comparison with the software-based implementations in the device. Authors in [117, 118] observed that fault-tolerant mechanisms, such as ECC, can contribute to decrease the fault rate, but unprotected areas of the device are still sensitive and are prone to produce critical effects on the device, including crashing effects. Similarly, other modules were analyzed, including the execution units used in modern applications, such as neural networks [119, 120, 121], observing the main effect and also proposing mitigation strategies to reduce the effects. Authors in [122] evaluated and demonstrated the minor fault effect of modifying the distribution policies inside the scheduler controllers of GPU devices. Other works explored the management of the parallelism levels of applications to control the sensitivity of GPU modules [108]. Results showed that dropping the workload activity in GPUs can be adopted as an initial mitigation strategy and reduce the fault effect, but with moderate performance costs. Finally, in [123, 69, 124, 125], the authors evaluated the effect of embedded GPUs to radiation effects and the main improvements when mitigating the effect through several levels of redundancy.

Nevertheless, most of these analyzes consider the complete device or perform reliability evaluations using mainly software-based fault injectors, which in some cases provoke difficulties on targeting specific modules to study particular behaviors. Moreover, these limitations can also interfere in the analysis of fault effects caused by transient faults.

In this chapter, several experimental quantitative results are introduced for individual modules in a microarchitectural description of a GPU (FlexGripPlus). These results are obtained using simulation-based experiments. More in detail, relevant modules of the data path and control path of the GPU architecture are analyzed through several applications aiming to identify critical submodules, which later can be employed for hardening purposes or for building new fault models. It is worth noting that the development of hardening techniques is out of the scope of this chapter.

This chapter is divided into four subsections. The first subsection introduces a simulation environment and the set of benchmarks employed in most experiments. The second and third subsections describe the reliability evaluations performed into the data path and control path modules of the GPU, respectively. Finally, the fourth subsection describes the procedures performed to analyze the effects of microarchitectural faults in GPUs and the effect on CNN-based applications.

3.1 Experimental set up for transient fault injection campaigns

3.1.1 Fault injection environment

A custom fault injector was developed to perform all experiments reported in this chapter and also to validate the proposed solution of chapter 4. This environment mimics the same behaviors and constraints that a GPU may have in the real operation, so the methods to identify and detect fault effects are the same as in real GPU platforms.

This fault injector was designed to perform fault injection campaigns in the FlexGripPlus model supporting a transient fault model. However, it must be noted that the tool is able to introduce two types of faults: permanent faults, based on the Stuck-at fault model, and transient faults, based on the Single Event Upset (or SEU) model.

The injection methodology of the tool is based on the method introduced in [113][126], which employs a set of commands to place a fault in any time of the simulation time and represent a fault effect. Moreover, the environment can reduce the total time of the fault simulation by taking advantage of *i)* parallel capabilities of the modern computers to run multiple simulations simultaneously (e.g., resorting to a multi-thread approach [127]) and *ii)* the *Utilization De-Rating* factor (UDR) of a module to pre-process the fault list and reduce the number of locations to inject faults. The UDR is computed utilizing results from a fault-free simulation. The information is analyzed (i.e., switching activity or correlation) in the target module, and finally, unused locations by an application are removed before simulation [128, 129].

The fault injection environment is implemented in a high-level language (*Python*) and is composed of three main modules, as shown in Fig. 3.1: *1)* a fault controller, *2)* a fault injector, and *3)* a fault checker and classifier. The environment links a set of configuration files with the execution of the behavioral/RTL simulator (*ModelSim* or *QuestaSim* by *Mentor* or *Xcelium Parallel Simulator* by *Cadence*) that works on the GPU model. The functions, in the fault simulator, can handle the execution of multiple fault campaigns on the model.

It is worth noting that it is complex to inject SEU faults in a behavioral/RTL model without timing information details. Thus, the fault simulator injects Single Bit Upsets (SBUs) in memory cells or register signals in order to generate the equivalent SEU effects. For the purpose of the works presented in this chapter, the SEU injection capabilities are used in the fault injection campaigns. It must be noted that all fault campaigns using the transient fault model employ a fault list composed of all locations in a target module, while the fault occurrence time is selected randomly.

A fault simulation campaign starts loading and compiling the GPU model in the *ModelSim* simulator. In this process, the control manager loads the GPU configuration, the application instructions and the initial data memory values. The kernel instructions and the model configuration are provided by the user before the fault campaign starts.

A golden simulation is performed in order to obtain the reference memory results and the performance parameters, which are later used in the classification stage. The fault control manager loads the fault list for the campaign. This fault list is composed of the signal locations where to inject each fault in the model. Depending on the fault model, additional parameters are required, such as the

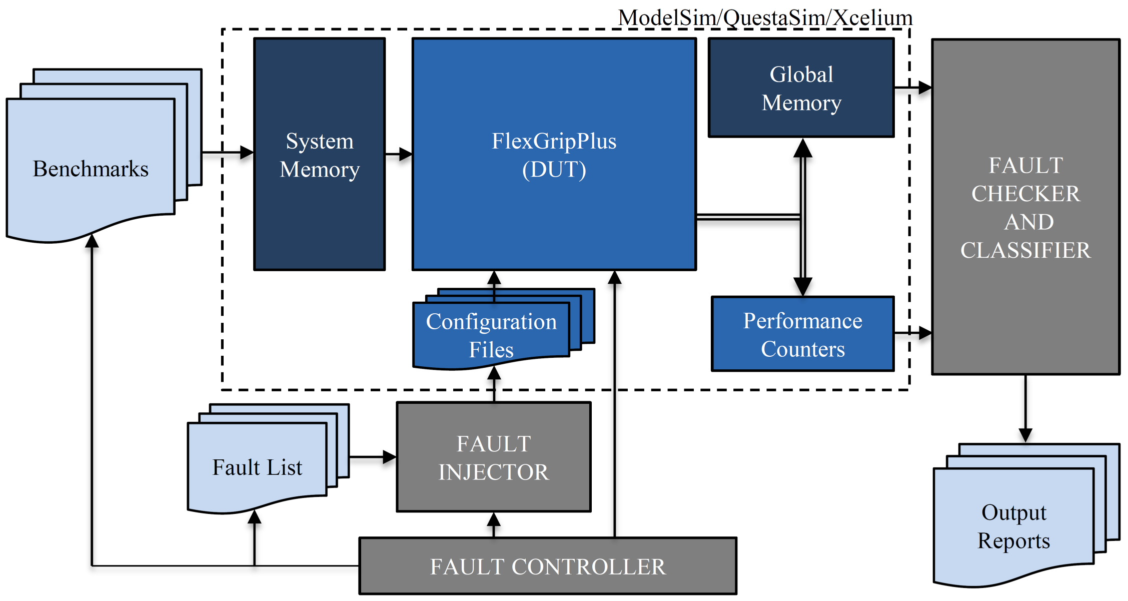


Figure 3.1: A general scheme of the custom environment for the fault injection campaigns.

injection time and the injection period. The fault decoder-generator reads from the SEU fault list (one fault at a time) and generates an equivalent behavioral injection command for the ModelSim environment, using the *force–deposit* syntax. Then, the control manager starts the fault injection campaign. The execution time limit is defined as twice the golden execution time of the program kernel. This value is employed in order to check performance degradation by the fault effect. Once the simulation finishes, the memory results and performance parameters are stored. Finally, the checker and classifier verify the generation of memory results. This classifier catalogues the fault effects in four categories: *Silent Data Corruption* (SDC), *Time-Out* (Performance Degradation), *Hang* (Detected Unrecoverable Error (DUE)) and *Masked* (Silent) when there is not fault effect in the outputs of an application.

A SEU effect is classified as SDC if there is a memory mismatch between the golden and faulty results. A Time-Out happens when the fault simulation time is greater or lower than the golden simulation time. The fault behavior is classified as DUE when the fault simulation is not correctly finished or the GPU model cannot correctly terminate its execution, additionally without results in the global memory. Lastly, a masked classification is used if there are not mismatches in memory results or execution time. It is worth noting that one fault simulation is performed for each considered fault.

Once the fault campaign finished three report files are stored. Those files include the memory results, if generated, the final dictionary file (which is composed of the fault type, the signal location and the final fault classification), and the fault-results file, including a summary of the total number of faults classified grouped by type.

The multi-threaded fault injection methodology is employed in the fault injector, mainly to handle the large number of faults to inject during the campaigns. The total number of SEU faults is commonly divided in three to ten equal-size fault chunks composing a partial fault list. Each partial fault list is assigned to an independent fault simulator with a FlexgripPus model to be processed as an independent simulation. Finally, the final injection results are grouped and analyzed.

3.1.2 Benchmarks

Six applications were selected as benchmark applications to evaluate the behavior of the different modules in FlexGripPlus against SEUs under different workload profiles.

- FFT: The application is based on the Coley-Turkey algorithm [130] and implements the butterfly element using CUDA. Since the original version of FlexGrip does not provide support for division operations. These operations were replaced with a software-based division approach using logarithmic and logical operations.

- Edge detection (Edge): The application is based on the Sobel algorithm applying an image filter of 3x3 to a 2-dimensions input.

- Vector_Add: This application is extracted from the CUDA samples SDK and is directly compiled for FlexGripPlus. This is a embarrassingly parallel application and calculates the sum of two vectors.

- Bitonic-Sort (Sort): This is another original application extracted from the CUDA samples SDK. The application sorts a sequence of consecutive data elements stored in an array. This application includes multiple combinations of data movements between memory and registers, and conditional control-flow instructions, which are data dependent, so generating multiple paths during the execution.

- M3: This application implements a Software-Based Self-Test (SBST) algorithm introduced later in section 4.1.1. M3 targets the generation of patterns to access the memory inside the scheduler controller. Thus, it is composed of multiple control-flow instructions utilizing mainly the control-path modules.

- Matrix Multiplication (MxM): This application is based on the General Matrix Multiplication (GEMM) routine, which is optimized using the square tiling approach. The input matrices are firstly divided into blocks. Then, partial results are obtained by multiplication of corresponding blocks. Finally, partial results are accumulated to get the final results of matrix multiplication. The implementation is limited to 32x32 input matrices.

Table 4.19 reports the main features of the benchmarks employed in the experiments. It is worth noting that each application employs several instructions in both formats (32 and 64 bit-sizes). Furthermore, the applications operate different workloads, so the data memory footprint differs for each of them.

Table 3.1: Main features of the benchmarks used in the experiments.

Target module	Type	Size (Instructions)	Duration (cc)	FC (%)
SP	ATPG	19,604	1,447,620	84.07
	Pseudorandom	55,000	3,434,235	83.99
SFU	ATPG IMM	16,856	1,200,034	90.75
	ATPG Mem	117	212,914	90.75
	Pseudorandom 30K	309	1,546,404	82.55
	Pseudorandom 60K	609	3,074,844	77.67
	Pseudorandom 90K	909	4,603,284	89.11
	Pseudorandom 120K	1,209	6,131,724	88.26

3.2 Analyzing the data path modules in a GPU against transient fault effects

The data path modules in a GPU are those modules that handles all operands for the parallel execution of an applications. This modules are commonly used in all applications.

In the presented section, two data path modules (the Register File (RF) and the Pipeline Registers (PRs)) are the target for the reliability evaluation. The RF is the main memory resource employed in the GPU architecture and almost 90% of the assembly instructions employ the registers in the RF as source of input operands or as destiny of output results for every thread. In contrast, the PRs are hidden structures in the GPU core and there is not a direct management or observability from the GPU programmer, so the behavior of these internal structures requires reliability analyzes, considering that has not fully evaluated in the past. For this purpose, five applications with different code flow and workload are employed for the reliability evaluation of the data path modules in the GPU.

It is worth nothing that other modules, such as the execution units were not target in the first part as several works have documented reliability evaluations for these parallel architectures and the research relevance of individual experiments is low.

3.2.1 Register File

For the RF, 30 fault injection campaigns were performed. The total number of SEU faults to be injected in a fault campaign is determined by establishing the total number of registers used by each application and per thread configuration. Then, the total number of bit fields is multiplied by a constant (i.e., 10) in order to define the faults to inject per location. 34,816 faults were injected for the *FFT*, *MxM*, *Sort*, *M3* and *Edge* applications under all SP configurations resulting in the confidence interval of 99.46%. Meanwhile, for the *Vector_Add* application, 10,240 faults have been injected for 32 SPs configuration, and 8,192 faults for 16 and 8 SPs. It is worth noting that each applications have a different program behavior, so diverse fault impacts are expected for each application.

With the aforementioned fault injection environment, described in section 3.1.1, the fault simulation time was reduced from about 200 hours to less than 25 hours: the adoption of the UDR factor allowed us to reduce by up to 95% the total amount of injected faults. Fig. 3.2 reports the error rate percentage for each application.

When comparing results from different configurations, from some applications, the trend of error rate is not so evident since the error rate is affected by several aspects, for example, the number of registers actually utilized during execution, the duration of each data stayed inside registers, etc. However, some interesting effects can be observed.

The results shows that the *FFT* and *Edge* benchmarks present a similar behavior: by increasing the number of threads per block (64 threads configuration) reduces the total error rate, which is expressed as the Architectural Vulnerability Factor (AVF) is mainly determined by the number of employed registers by configuration. The AVF expresses the probability that a visible error occurs due to a bit-flip in any storage element of a device [131]. In practical terms, AVF is the rate between the number of propagated faults-effects in a device and the total number of injected faults in the device.

A different behavior is shown by the *Vector_Add* application. In this case, raising the number of threads per block (i.e., increasing the parallelism) generates a direct increase in the failure rate provoked by SEUs, see Fig. 3.2. Moreover, *Vector_Add* does not produce any DUE or Timeout as it does not contain control-flow instruction in its description.

Sort and *M3* both have DUE as the majority due to the large percentage of branch instructions that can be affected by SEUs in the registers. In *Sort*, an growth of DUE rate can be observed when the number of threads varies from 32 to 64 as opposite in *FFT* because *Sort* contains a large portion of control-flow instructions whose execution directly depends on operands and addresses stored on the registers. Results show for the *Sort* applications that using more registers increases the probability of fault effects as DUEs when corrupted registers access memory resources or interfere in the execution of control-flow instructions.

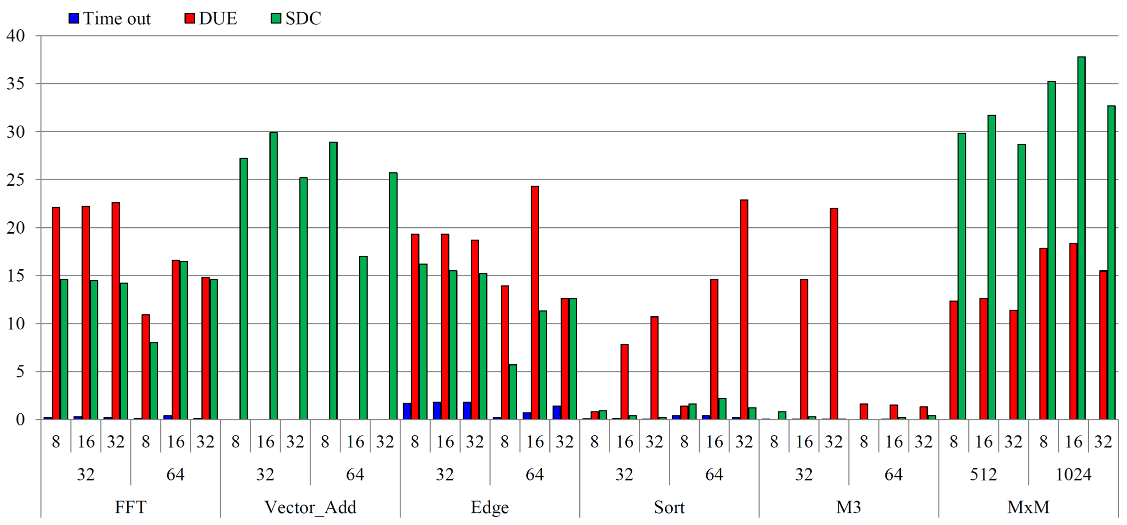


Figure 3.2: Fault rate results in the RF of FlexGripPlus (the horizontal axis, from top to bottom, are #SPs, #Threads and application name).

In *MxM*, the incremental trend of DUEs is not as obvious as in *Sort*. In fact, control-flow instructions in *MxM* do not depend on the input values, though it still depends on the loop variables. Interestingly, these observations are similar to those shown in [113] for applications with a high percentage of control-flow instructions.

Similar trends can be observed in different applications when the number of SPs in the SM increases from 8 to 32, for example, the increment of DUEs in the *Sort* application. However, other applications do not provide such a consistent and visible trend. When the number of SPs changes, the register utilization factors and the organization of threads inside the warps also change. This variation causes different load and store patterns for used registers of an application. Those patterns are dependent on the available number of SPs in the GPU core.

3.2.2 Pipeline Register

The PRs are structures distributed in the GPU core. These hidden registers for the programmer are crucial for the operation of the system. However, a clear fault sensitivity of such a modules have not been described for GPU architectures. For this purpose, FlexGripPlus is employed to evaluate the effect of fault in the PRs. More in detail, the distribution of the PRs in the SM core of FlexGripPlus depends on the number of available SPs, see 3.2. Thus, for the experiments, the three configurations of the SM core where considered.

In total, 144 faults injection campaigns were performed targeting PRs in FlexGripPlus. A total of 30,000 SEUs were injected per configuration. The fault injection results in terms of the averaged fault rate in the entire structure are shown in

Table 3.2: Size of the PRs according to the number of SPs in the SM of FlexGrip-Plus.

Pipeline Registers	Controlpath	Datapath (SPs)		
		8	16	32
Warp to Fetch (W-F)	140	–	–	–
Fetch to Decode (F-D)	237	–	–	–
Decode to Read (D-R)	408	–	–	–
Read to Execute (R-E)	302	1,024	2,048	4,096
Execute to Write (E-W)	251	512	1,024	2,048
Write to Warp (Wr-W)	133	–	–	–
Total	1,471	1,536	3,072	6,144

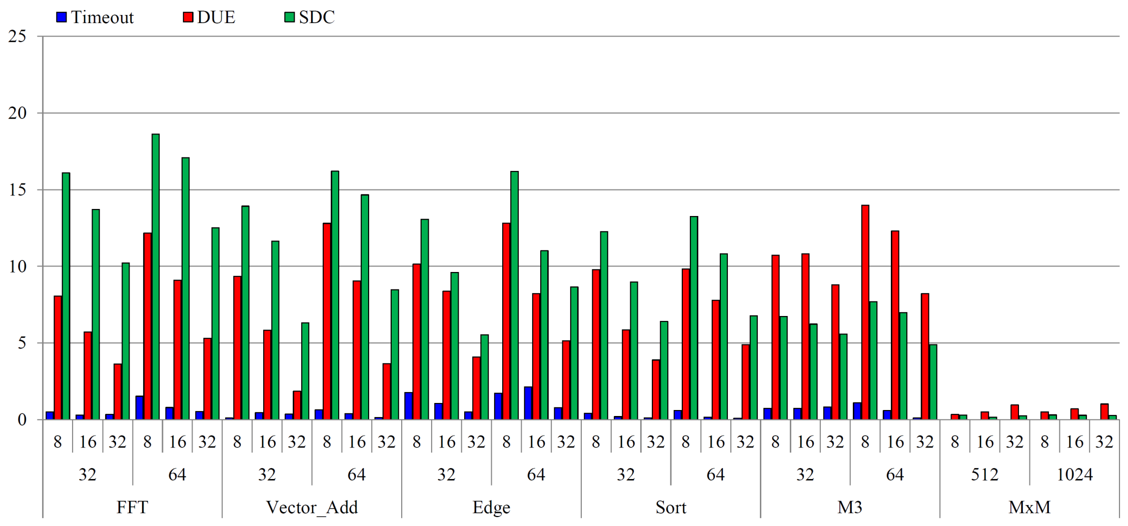


Figure 3.3: Fault rate results in the PRs in FlexGripPlus (the horizontal axis, from top to bottom, are #SPs, #Threads and application name).

Fig. 3.3.

It can be observed from the results that an increment in the number of threads will lead to an increment of error rate, including SDC and DUE. This behavior can be explained as the additional time cost for processing warps of the same block increases the probability of an SEU in PR to be propagated through pipeline. However, this trend also depends on the application type. As can be observed, see *FFT* and *MxM*, each application has its own fault sensitivity to SEUs and some may have considerable fault impact (*FFT*), meanwhile others are less affected (*MxM*) even when more threads are active in the parallel program.

Another clear trend that can be observed is that the error rates decrease when more SPs are available in the SM. This means that the granularity in the GPU core composition is also a relevant parameter when considering the reliability of the GPU. In fact, the original architecture of the G80 architecture (FlexGripPlus) only includes 8 SPs per SM. However, the flexibility in the GPU model allow us to observe architectural changes in the execution units, which are inline with recent GPU designs, so including 32 or more SPs. As can be observed, for each application, the fault rate trend is stable for each SM configuration in the GPU. In contrast, the timeout trend is not consistent across different applications and configurations.

In *M3*, DUEs are the majority for all the configurations since this is a control-flow-oriented application. On the other hand, in *Vector_Add* (the data-oriented application without control-flow instructions), the majority effect lands in SDC. For the other applications, the observed effect is less obvious.

When analyzing the PRs between different stages, the SEU sensitivity fluctuates from 1.2 to 13.5 times (see Figs. 3.4 and 3.5), which indicates the existence of some critical PR locations. These PR locations are architectural locations in the PRs that are most commonly affected by faults and can generate SDCs or DUEs (i.e., the register R_0 on each thread stores the thread identifier for each active thread and a fault affecting this register can provoke critical effects). It turns out the PRs storing the instruction decoding and warp status information are the ones of highest sensitivity against SEU contributing a large proportion of SDC and DUE during fault injection campaigns. In [113], the authors presented similar conclusions when evaluating the PRs of a GPU. Although a direct comparison of the results cannot be performed, both works show that the *E-Wr* PR is among the most SEU sensitive PRs, as indicated in Figs. 3.4 and 3.5. An explanation for the observed trend in Figs. 3.4 and 3.5 is the fact that *E-Wr* is the PR closer to the memory resources after performing an operation, so a SEU affecting one of these registers is more susceptible to be propagated to any output that other SEUs affecting another PR, such as the *Wr-W* or *W-F*, where only the critical affected locations are propagated and compromise the correct operation of the device.

3.3 Analyzing the control path modules in GPUs

The control path modules of a GPU core are small units (in fact, experimental results show that the controller units in the FlexGripPlus model with 32 SP/FP32 cores occupy less than 10% of the total area in the GPU core), which are devoted to manage the operation of an assigned workload during the operation of a parallel program. However, the structural location and the importance in the execution of these modules suggest that a fault can corrupt the operation of one or multiple threads and even stop the execution of a GPU core. In this section, results of reliability evaluation at microarchitectural level are provided for two important modules in the GPU, the warp Scheduler Controller (SC) and the Divergence Management Unit (DMU).

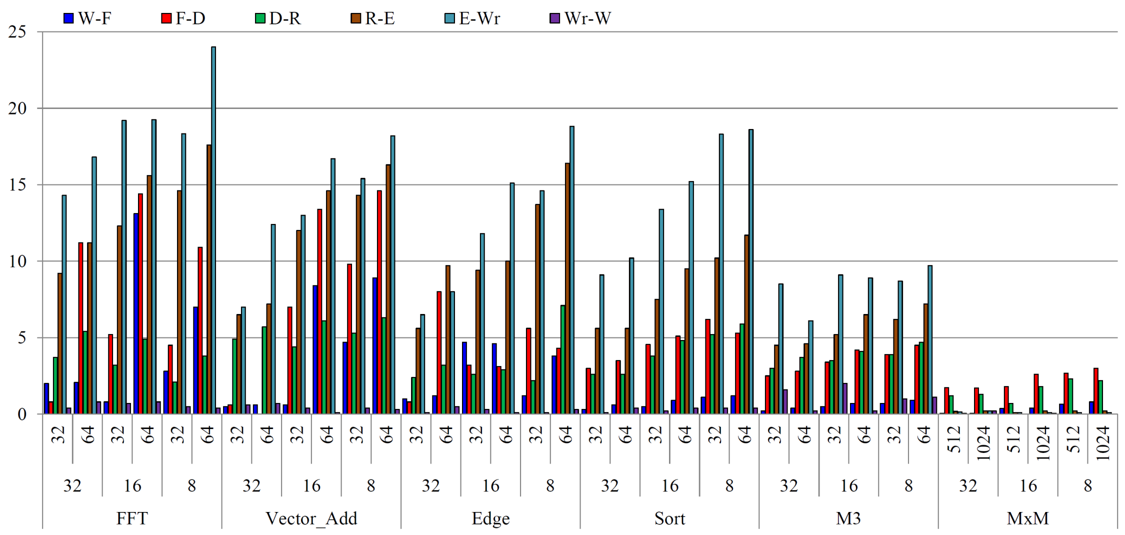


Figure 3.4: SDC rate results per pipeline register in the SM for the evaluated applications (the horizontal axis, from top to bottom, are #threads, #SPs and application name).

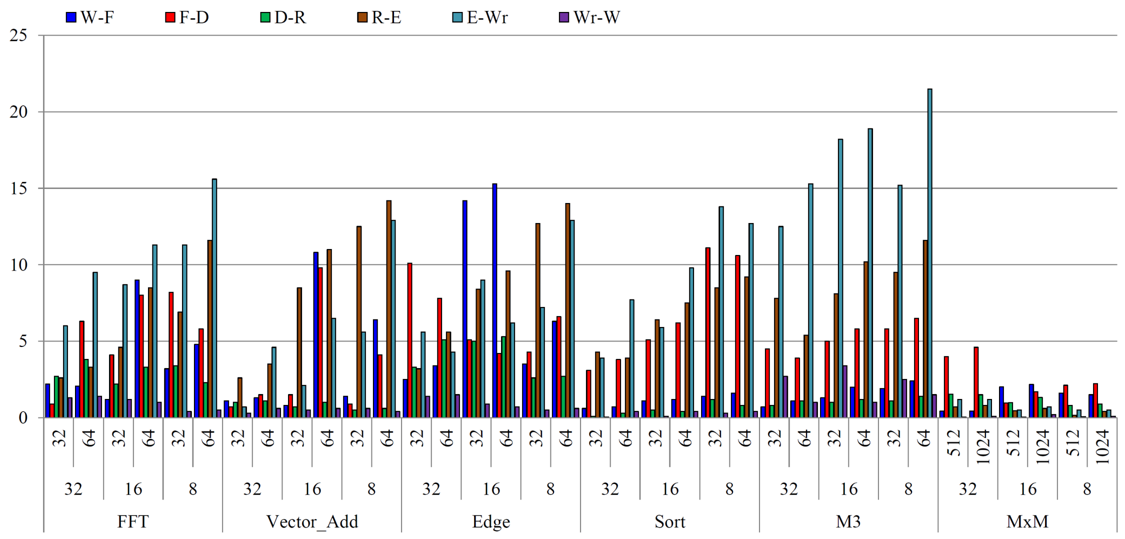


Figure 3.5: DUE rate results per pipeline register in the SM for the evaluated applications (the horizontal axis, from top to bottom, are #threads, #SPs and application name).

3.3.1 Scheduler controller

The SC was divided into two parts for the analyzes: the internal memories and the sequential logic elements. For this module thirty-six transient fault injection campaigns were performed.

In principle, the SC module is critical for the operation of the GPU core. However, a first view to the results contradicts the expected criticality in the analyzed applications. In fact, the fault injection campaign results show low sensitivity against SEUs; though the sequential logic corresponds to 14.3% of the entire elements in the SC, it generates between 85% and 92% of DUEs among the detected faults for all the tested applications.

The unexpectedly low error rate in the internal memories of the SC has two main explanations. Firstly, the low fault impact in the memories seems to be caused by a existing loop between the SC and the pipeline stages in the SM, which contributes to masks the fault effects. More in detail, the execution of an instruction in the SM requires the control and the trace from the SC, which causes that values in the scheduler memories are frequently refreshed after a SEU data corruption is caused. However, in several cases the values are corrected by effect of the incoming values from the pipeline stages, so instead of maintain a corrupted value in the memories, the values from the pipeline stages are stored and mask the corrupted values. It is worth noting that this micro-architectural loop behavior *"as fault-masking"* was observed in the FlexGripPlus model. However, we cannot claim that identical or similar structures are employed in modern GPU designs.

The second main reason is the number of threads configured for the evaluated applications. In most benchmarks, which are configured with a fewer number of threads, the architectural *"fault-masking"* behavior is effective when the workload is relatively small. However, it becomes ineffective when the workload is bigger and the number of threads in the application increases. The main effect of increasing the threads can be observed in the *MxM* application. In this case, the probability of reading values from the scheduler memories (in particular from corrupted locations by SEUs) before any overwritten increases with the workload.

In the operation of the SC, it can be initially expected that one SEU causes a single SDC and corrupt one value in the results. However, as documented in previous works [117][108] it was observed that multiple SDCs could also be presented in the output memory for several applications (i.e., lines, blocks, or random locations with erroneous results).

Table 3.3 reports the percentage of SEUs that causes SDC effects in the scheduler divided into those generating single and multiple effects in the memory. An in-depth analysis shows that the multiple SDCs are caused by SEUs affecting the Warp Program Counter (wPC) and other fields (base addresses in the hardware distribution of a block, which are employed to access the memory resources) of the memory in the scheduler, which are used redundantly by numerous threads. Similarly, the multiple SDCs are caused by logic elements in the scheduler related to the management of the information of the warps.

In general, for all the analyzed applications, errors corrupting the wPC also affected the group of threads and propagated in the execution, so causing multiple SDC. In contrast, faults in the Active Threads Mask (aTM) field produce most

Table 3.3: Distribution of SEU effects in the scheduler causing single and multiple errors in the outputs.

Benchmark	TPB	SPs	SDC (%)					
			Warp Status memory			Logic		
			Single	Multiple	Total	Single	Multiple	Total
FFT	32	8	0.024	0.025	0.049	1.4	1.28	2.68
		16	0.014	0.01	0.024	1.2	1.12	2.32
		32	0.042	0.056	0.098	0.4	0.61	1.01
	64	8	6.53	2.4	8.94	1.1	1.43	2.53
		16	2.19	1.2	3.39	0.7	0.56	1.26
		32	0.12	0.1	0.22	0.64	0.39	1.03
VectorAdd	32	8	1.6	0.54	2.148	1	1.4	2.4
		16	0.3	0.16	0.464	0.9	1.06	1.95
		32	0.06	0.01	0.073	1.10	0.15	1.25
	64	8	6.7	2.43	9.131	1.1	1.59	2.69
		16	3.1	1	4.102	1.2	0.95	2.15
		32	0	0	0	1.01	0.35	1.36
Edge	32	8	0.35	0.18	0.537	0.5	1.14	1.64
		16	0.08	0.06	0.146	0.4	0.51	0.91
		32	0.02	0.03	0.049	0.20	0.51	0.71
	64	8	1.1	1.51	2.612	0.8	1.89	2.69
		16	0.5	1.38	1.88	0.94	1.21	2.15
		32	0.01	0.039	0.049	0.4	0.8	1.2
Sort	32	8	0.002	0.01	0.012	0.03	0.04	0.074
		16	0.002	0.01	0.012	0.05	0.05	0.106
		32	0.01	0.039	0.049	0.10	0.00	0.01
	64	8	0.16	0.31	0.476	0.01	0.025	0.035
		16	0.25	0.33	0.585	0.1	0.27	0.37
		32	0.21	0.36	0.57	0.01	0.005	0.015
M3	32	8	0.41	0.13	0.54	0.3	0.11	0.414
		16	0.28	0.18	0.46	0.12	0.17	0.297
		32	0.14	0.15	0.292	0.23	0.23	0.467
	64	8	0.93	0.6	1.53	0.6	0.21	0.81
		16	1.48	0.7	2.185	0.66	0.24	0.9
		32	1.58	0.65	2.23	0.5	0.49	0.99
MxM	512	8	13.51	20.69	34.2	0.16	0.1	0.26
		16	12.75	18.3	31.05	0.17	0.11	0.28
		32	11.39	13.2	24.59	0.22	0.1	0.32
	1024	8	14.17	25.34	39.51	0.21	0.09	0.3
		16	14.38	23.21	37.59	0.28	0.11	0.39
		32	11.51	21.03	32.54	0.27	0.1	0.37

single SDC effects. In some control-flow-based applications (*EDGE*, *FFT*, *Sort*, and *M3*), a small number of multiple SDC were caused by errors present in the aTM field. Those errors modified the execution of one thread in the execution paths and caused additional operations or the missing of operations, which affected subsequent executions in the program flow.

More in detail, the distribution of single and multiple SDC effects differs on the applications and depends on parameters such as the coding style and internal modules employed that are correlated with the SC module. In the *MxM* application, the high percentage of multiple SDCs is caused by the existent connection between the status information of a warp stored in the SC and the RF and the shared memory modules. However, the trend is not present in other applications. *Vector_Add* and *M3* have limited use of the RF and Shared memory, so the contribution of multiple SDCs by misbehavior in the management of these modules is low.

In other work [132], the authors reported the effect and criticality of SDCs affecting neural network applications in GPUs. In results, the authors also included results for the *MxM* application. A comparison between the results of the *MxM* with those introduced in the present work shows equivalent trends for a small number of threads. Furthermore, from the architectural view point and considering the distribution of the blocks of an application into the available SMs of a GPU, the observed behavior is similar and coherent to the main conclusion observed in the related work. More in detail, in [132], the authors found that the distribution of SDCs caused by multiple errors in the output lay in the range from 45 to 65%, without error margins. In the listed results in Table 3.3, the *MxM* applications have an equivalent tendency for the pool memory with a distribution of SDCs in the range of 54 to 65%.

Unfortunately, the results show that the previous tendency is not followed by the logic part of the SC, and it presents a lower range (27-39%). However, it should be noted that the experiments performed in [132] and the presented in this work have relevant differences in terms of the type on injection performed and the method to inject faults. In [132], the fault injection is performed to all modules of a GPU. In contrast, this work only targets the injection of fault targeting specific parts of the scheduler controller only. In any case, the obtained results show the criticality of the scheduler module and the susceptibility to SEUs of the *MxM* application. On the other hand, the distribution trend of SDCs for other applications is different. In *Vector_Add*, *FFT*, *Edge*, and *M3*, the trend shows a higher percentage of SDCs caused by single output errors than multiple ones, as analyzed previously. Furthermore, additional 24 fault injection campaigns have been performed on *Vector_Add*, and *M3* with different configurations of threads per block, and a fixed number of SPs to 32. The additional experiments are intended to provide remarks regarding the fault masking effect in the SC memories mentioned before.

The two applications were selected mainly by the distinctive execution behaviors in the scheduler. The *Vector_Add* program is fully parallel and includes high data-intensive operations without control-flow or thread divergence operations. In contrast, *M3* is mainly composed of control-flow operations and thread divergence

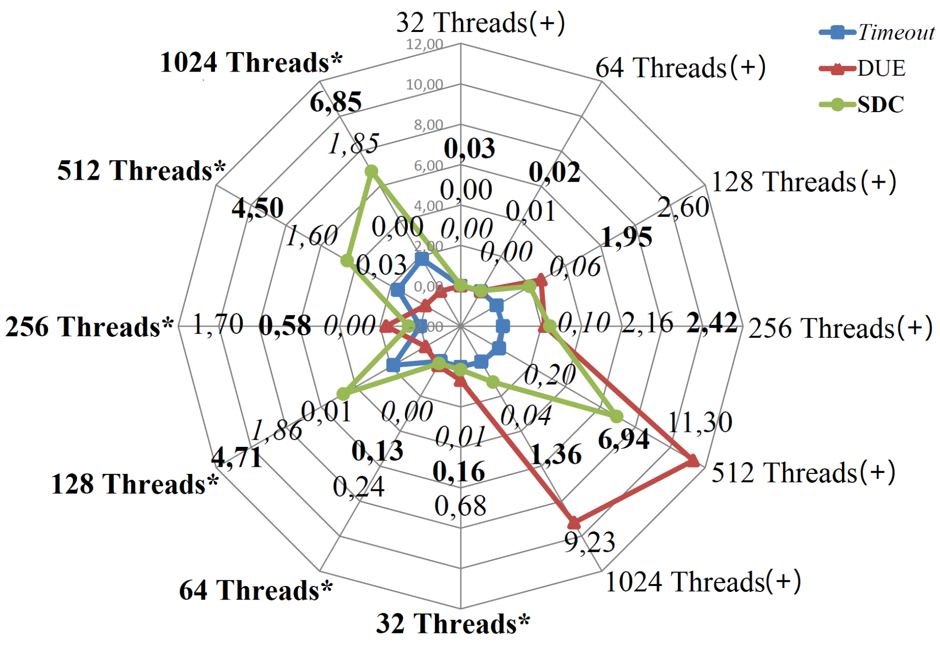


Figure 3.6: AVF results in the memory(+) and the control logic (*) of the scheduler controller of the GPU when executing the VectorAdd application and increasing the number of threads.

routines. Thus, the scheduler is excited with entirely different patterns during the operation of the two applications. The number of threads, for these evaluation, varies in the range from 32 to 1024 for both applications.

From the results (see Fig. 3.6 and 3.7), when the number of threads is configured as 32 or 64 for both applications, the fault masking in scheduler memory is effective to limit the impact of SEUs, though we can still observe a small amount DUE caused by SEUs in the logic part of SC. However, the increment in the workload and parallelism level (more threads per application) causes a rapidly increment in the error rate. In the end, two different distributions of DUE and SDC can be observed when comparing results from *Vector_Add* and *M3* applications.

Thus, an optimized implementation for performance is, as it often happens, not the best solution when reliability is concerned. Further actions to increase reliability should be adopted, such as ECC in the memory and Triple Modular Redundancy (TMR) in the control logic. Finally, depending on the type of application, different solutions (possibly in combination) can have effectiveness for improving system reliability against SEUs. Interestingly, the previous results are coherent, supplement the main findings and are in-line with other works that evaluate the reliability of GPU devices against radiation. More in detail, the reported results show the behavior of several applications under the effect of several modules when affected by faults. A low-level granularity analysis shows that the same trend in terms of

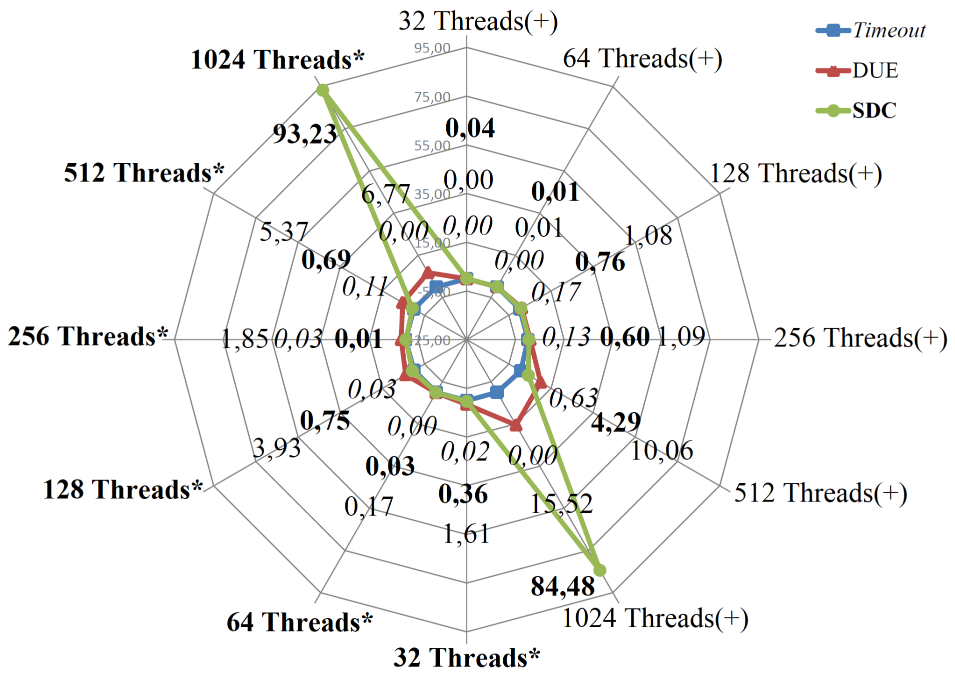


Figure 3.7: AVF results in the memory(+) and the control logic (*) of the scheduler controller of the GPU when executing the M3 application and increasing the number of threads.

reliability when the number of threads is reduced [108].

3.3.2 Divergence stack

Several fault injection campaigns were performed in the Divergence stack module, see Appendix A for additional details of the divergence module. It is worth noting that the fault injection campaigns on this module did not consider the *Vector_Add* application because this parallel kernel does not employ the module. In contrast, the *FFT* and *Edge* benchmarks were evaluated employing 50,688 faults in each fault campaign. The results are presented in Fig. 3.8.

Fig. 3.8 reports the error rate results for the Divergence module. As can be observed, the relatively low sensitivity against SEUs of the module can be explained considering the limited percentage used by the applications. In fact, the main reason for the limited fault effect is the few cases when multiple branches (in the application) activate multiple level entries in the Divergence stack. More in detail, each entry-line is employed for the fraction of a divergence generation, meaning an application timing dependency and different SEU sensitivity per line. This behavior is directly dependent on kernel description, nesting divergence, total number of divergence path instructions, and the number of convergence points.

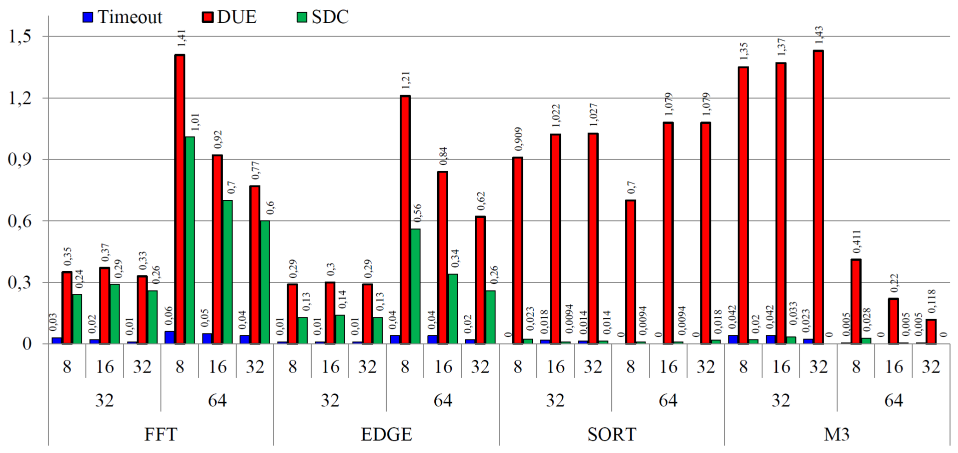


Figure 3.8: Fault injection results for the Divergence module (the horizontal axis, from top to bottom, are #SPs, #Threads and application name).

Results support the previous explanation. Moreover, considering that usage of this unit, for both kernels, is less than two thirds of the total simulation time and each additional pushed line presents less activities, the sensitivity to SEU effects reduces drastically. Nevertheless, the general trend (in the employed parts of the module) shows that a fault affecting the divergence stack is critical and can cause a DUE collapsing the entire operation of the system.

A change in the SP configuration seems to affect the sensibility of faults in the Divergence module. This behavior can be observed in the *FFT*, *Edge*, and *M3* applications with 64 threads. In each case, increasing the number of SPs is inversely proportional to the susceptibility to SEUs and is explained by the reduction in the management operations performed by the scheduler for a large number of SP cores in the SM. However, there is not any direct interaction among the divergence module and the number of SPs, so the observed reduction in the version of 64 threads of the applications is mainly caused by the correlation between reduced management operations in the SC and shorter operation times on the routines executed in a divergence path.

Among the four tested applications, *Sort* includes only one conditional control-flow instruction generating multiple execution paths. However, the divergence in this benchmark is data-dependent, so the generation of a new path depends on the comparison of two operands from memory. This behavior explains that the error rate did not change so much with different configurations in Sort.

The observed behavior in *M3* is also different from others as it intends to generate multiple intra-warp divergences sequentially in the first 32 threads, leading to an intensive switching activity in the SC. However, this application does not generate nesting divergence paths (i.e., it does not use multiple level entries in the Divergence module), so, when the number of threads changes from 32 to 64, the switching activity in scheduler is reduced, while the level of utilization of the Divergence module due to divergence paths is not increased, leading to decreased error rate, as shown in Fig. 3.8.

For *FFT* and *Edge*, similar trends can be observed as error rates increase with the number of threads and decrease with the number of SPs. This behavior is mainly due to the switching activity when the different combinations of active warps and the number of SPs affect the organization of the warp execution.

Regarding the distribution of the DUE and SDC error rates, it depends on the affected location within an entry in the Divergence module. An SEU in the wPC field may create Timeout or DUE (or SDC). Similarly, an SEU affecting the aTM field may generate SDC, by interrupting thread execution (i.e., unfinished computation), or DUE by causing threads to miss the synchronization point. Finally, an SEU in the warp ID field produces Timeout effects. As seen in the comparison between *FFT* and *Edge*, a reduction in the number of threads can help to reduce more than 50% of the SDC error rate, which is coherent with the conclusion introduced in [107][132]. Interestingly, previous analyzes showed that internal modules of the GPU cause different fault sensitivity effects according to the executed workload.

3.4 Analyzing CNN workloads to assess fault impacts in GPU modules

Modern applications require special accelerators, such as GPUs, to increase the performance in several domains, including autonomous systems for the automotive, aerospace, and military markets. The highly parallel architecture of GPUs, in fact, fits the required computational characteristic of most HPC codes and is particularly efficient in executing matrix multiplication, which is the computing core of Convolutional Neural Networks (CNNs) used to detect objects in autonomous machines.

Unfortunately, as explained at the beginning of this chapter, the impact of faults on some characteristic and critical modules of a GPU, such as the SC or the PRs, can hardly be investigated without a detailed microarchitectural analysis, whose complexity and computational requirements make it hard to complete.

Recent works have suspected that a SC corruption can lead to a crash or impact the computation of several parallel threads [133, 117, 134]. As a result, multiple GPU output elements can potentially be corrupted, effectively undermining the reliability of several applications, including CNNs [135]. Unfortunately, as it is not possible to inject faults in software directly on the SC, all previous findings are based only on observations and speculations that still need to be confirmed.

The scope of this subsection is to go a step beyond in the analysis of transient faults in hidden modules of GPU cores, and focus on the impact of faults in the execution of CNNs in GPUs. The FlexGripPlus model [97] is employed to perform the experiments and evaluate the microarchitectural reliability of those modules (SC and PRs) that cannot be evaluated with other methods.

A two-level fault injection [136] approach to GPUs and CNNs is exploited to avoid the prohibitively high time a microarchitectural simulation requires (that would prevent the characterization of complex algorithms such as CNNs). Moreover, the efficiency of the microarchitectural evaluation for the CNNs considered the internal composition and division in a set of independent layers, which execute massive and repetitive convolutions on portions of matrices (*Tiles*).

The main targets for the reliability evaluation and characterization are the scheduler and pipeline registers when the GPU core execute a selected subset of representative CNN tiles. Then, with a software fault injection, the errors are propagated in the corrupted tiles during the CNN execution. This strategy allows to investigate the detailed effects of hardware faults in specific GPU hardware on the execution of convolution in a much shorter time. To the best of my knowledge, this work presents, for the first time, a microarchitectural reliability evaluation performed injecting faults into individual modules of a GPU when operating a CNN workload.

The reminder of the section is organized as follows: Subsection 3.4.1 introduces the proposed two-levels approach describing the microarchitecture and software fault injectors as well as the CNN selected as a case study and Section 3.4.2 presents and discusses the experimental results and provides the main conclusions.

3.4.1 Proposed characterization strategy

The goal is to evaluate the effect of transient faults affecting the SC and PRs during the execution of CNNs in the GPU. While the proposed evaluation strategy can be adapted to any other GPU module, the selected modules are relevant for three main reasons: *i)* They would not otherwise be accessible and, then, could not be characterized using software fault injection, only. *ii)* Previous works have speculated about the criticality of faults in these resources [133, 117, 134] but a detailed analysis about their propagation and effects is still lacking. *iii)* Traditional hardening strategies, such as ECC, can hardly be applied to SC and PRs, due to the high implementation costs for these hidden structures.

The proposed reliability evaluation methodology takes advantage of the computational independence of frames and tiles inside an image to process to significantly reduce the time required to inject faults at the microarchitectural level on GPUs executing CNNs [137].

Each layer in the CNN extracts a specific feature that is then passed as input to the next frame. Frames are executed sequentially, and there is no further interaction between them. Additionally, as multiplying big matrices introduces memory latencies and dependencies that could undermine the performances, in GPUs convolution is divided into smaller parts (*tiles*). Hence, the tile size depends on the

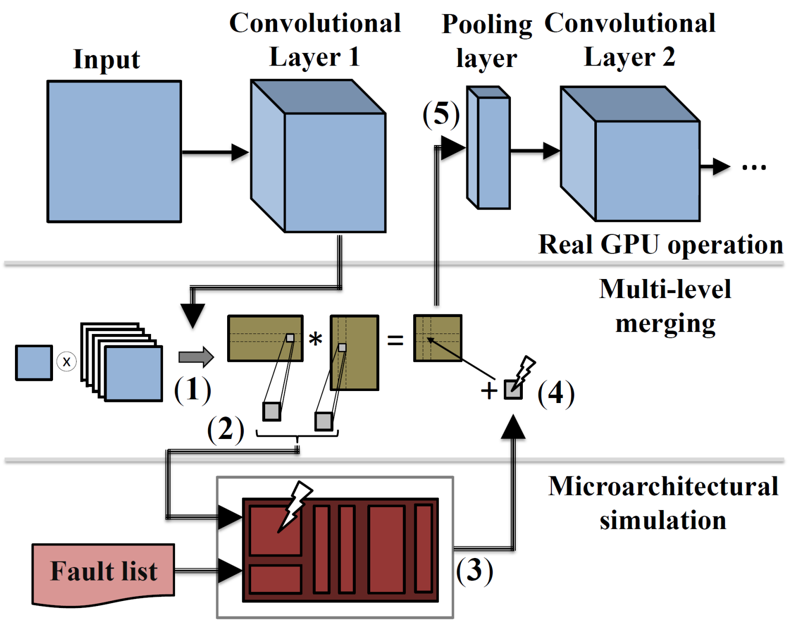


Figure 3.9: A general scheme of the proposed method.

GPU and is tuned to fully use the core resources, guaranteeing that caches and registers are neither saturated nor underutilized. To complete a convolution, tiles are independently processed several times in the GPU cores, with different input values. Finally, tiles are combined to form the layer output.

The strategy is based on first characterize, with microarchitectural fault injection, the effects of faults on a subset of representative tiles. Then, a software-based fault injector employs the previous results to inject the corresponding errors in each tile of the considered layer and propagate each of them to the downstream layers till the CNN output.

The reliability evaluation of a CNN limited to evaluate the Tiles can be justified by the fact that transient faults can only corrupt the operation or data that is being processed when they are activated. A fault in a SM can corrupt only the execution of the tile that is being executed (commonly, CNN workloads (tiles) are assigned to the available SMs in a GPU) but not other tiles nor other layers. A corrupted tile implies an erroneous layer output that will be propagated to the downstream layers. Thus, there is no need to simulate the whole CNN nor a whole layer with the time consuming microarchitectural simulator.

Figure 3.9 shows the operational flow of the proposed strategy. As preliminary tasks, the main features of the CNN under evaluation are extracted, including the number of convolutional layers, the layers' input/output size, and the golden (fault-free) input operands and output feature maps for each layer. Then, the following five steps are applied in sequence.

1. The convolutional layers in the CNN are translated into the equivalent matrix multiplication operations.
2. A subset of tiles from the matrices are selected and used as targets for the fault-simulation environment.
3. The microarchitectural fault injection on a selected tile is performed and the fault effects are propagated to the tile output, generating results reports. These reports are used to identify the fault locations in the output tile.
4. The corrupted tile is merged with the other (fault-free) partial results in the convolution to compose the layer's output, i.e., the feature map resulting from the injection.
5. Finally, the execution at the software level of the CNN propagates the potential errors to the output and the fault impact on the CNN output is classified. The previous process is repeated for each convolutional layer in the CNN.

Fault injection environment

The microarchitectural fault injection campaigns follow the same procedures explained in section 3.1.1, so injecting one SEU per simulation. The *MxM* application is selected for the experiments as the convolution operations performed by a GPU device when processing CNNs. It is worth noting that LeNET CNN is selected to validate and evaluate the proposed method.

The FlexGripPlus model is configured with one SM. A set of representative tiles is selected to be characterized (*Faulty Tiles*) from each convolutional layer. The remaining tiles (*Fault-Free Tiles*) in a layer are taken from the fault-free execution of the CNN. After the injection, the output report is used to merge the output of the faulty tile with the results from the fault-free tiles in order to create the fault effect in the output feature map of the layer.

Reports causing SDCs are used in the merging process while the others are marked as masked or DUE. Finally, the execution time of the fault campaign directly depends on the size of the targeted structure in FlexGripPlus. Similarly, the number of fault campaigns depends on the number of selected faulty tiles in a layer.

Tiles Selection and Characterization

The tile size is tuned depending on the SM computing capabilities. For the experiments and considering the original composition of the GPU model, the optimal tile size is of 8x8 as 8 SPs are available per core. Even if simulating just tiles significantly reduces the complexity of the characterization, the number of tiles that need to be simulated makes the exhaustive simulation unfeasible. A small CNN as LeNET, in fact, has 3,551 8x8 tiles an with a 12 CPUs server, the characterization of each tile takes at least 5h for the SC and 8h for the PRs. Thus, a selection of the most representative tiles is used to reduce the total simulation time. For each layer, a subset of tiles is used for the reliability evaluation and characterization.

Several experiments were performed on subsets of tiles in order to observe patterns that allow a reduced, but still representative, microarchitectural layer evaluation. In the case of convolution, most tiles process very similar data while the tiles at the edge of the matrix have a higher amount of zero elements [138]. Then it is possible to select and characterize three tiles for each layer: (M) **Max tile** (the tile with the highest sum of elements values), (Z) **Zero tile** (the tile with the highest number of zeros), and (R) **Random tile** (a tile selected among the ones that do not have significantly biased values). This selection is also in concordance with other works in the area that showed that the propagation of data path faults in an application on GPUs is data independent if the input data is not biased (i.e., there is not an excessive amount of 0s or all 1s) [139]. Moreover, the filter used in a layer is the same for all tiles, reducing the divergence of input of different tiles.

Multi-level merging

The multi-level merging to integrate FlexGripPlus with the software execution is performed in two steps.

1. Before the microarchitectural fault injection a *"demerging"* process extracts the input operands (i.e., faulty tiles) from the input matrices of the layer under evaluation. The locations of the 8x8 faulty tiles are defined (as explained in the previous subsection) and extracted from the input matrices. Then, the faulty tiles are translated into the format of the GPU model and employed as inputs in the microarchitectural simulation.

2. The partial convolution result of the faulty-tile is merged with the fault-free tiles to create the expected output feature map of the layer in which the fault has been injected. The corrupted feature map is compared with the golden one to identify the faulty locations for further analysis. These reports are also used to feed the software fault injection that propagates the faulty feature map to the next stages of the CNN. Only feature maps that are different from the golden copy are propagated in software while the faults that do not impact the tile and the corrupted tiles that do not impact the layer output (in the merging process the faulty tile can be smoothed by the other fault-free tiles) are marked as masked.

Software-level evaluation

The software environment executes the CNN and includes a fault simulator that performs the fourth stage in Figure 3.9. A Python script performs the execution of the multilevel merging with the real CNN execution, allowing to select the input for each layer in the CNN. For each corrupted tile in each layer, the microarchitectural fault injection report is used to build the corrupted feature map that is sent as the input of the downstream layer. The fault is then propagated till the CNN output. The software environment also tracks the propagation of the fault, comparing the output of all the downstream layers with the fault-free expected output. This information is essential to understand if (and why) a microarchitectural fault is masked during the CNN execution.

When the CNN completes the classification process, the CNN output errors are categorized into two groups: if the error at the CNN output induces a miss-classification it is marked as *critical*, *not critical* otherwise.

Limitations

Unfortunately, to protect strategic technological and economical information, the manufacturers are not releasing any model for their architectures. As explained in Chapter 2, the closest to represent the microarchitectural behavior of the commercial devices is FlexGripPlus. The discrepancy between the FlexGripPlus and the modern GPU architectures, however, does not undermine the proposed idea of decoupling the two fault injection methodologies. The proposed frameworks and the insights it allows to gather can be easily extended to any other GPU model. Moreover, once the fault reaches a visible state for the software, its propagation depends more on the executed code than on the underlying architecture [140]. The propagation of the corrupted tile, then, depends more on the CNN code than on the GPU architecture.

3.4.2 Experimental results

The LeNET [138] CNN is used as case study, which is composed of 2 convolutional layers, 2 fully connected layers (output layers) and 2 pooling layers. For each one of the convolutional and output layers in LeNet, 3 representative tiles (M, R, Z) are selected and used in the fault injection experiments with the FlexGripPlus model. We do not consider pooling layers in the microarchitectural simulation as the simple operation they execute makes them less interesting and less critical for CNNs reliability [135, 141].

As mentioned above, the SC and the PRs are the targets for reliability evaluation. The analysis of these modules can contribute to explain the behaviors of fault affecting these modules and the fault propagation effect when executing CNN workloads.

Microarchitectural fault injection results

Twelve fault injection campaigns are performed on each of the 2 targeted modules (3 tiles for each of the 4 layers) injecting SEUs. All experiments are simulated on a server using an Intel Xeon CPU running at 2.5 GHz, equipped with 12 cores, and 256 GB of RAM.

The fault simulation environment is configured to inject 10 transient faults per each target location in each module. The fault injection time is randomly selected during the execution of the program kernel [128, 129]. In total, 20,000 faults are injected in the scheduler and 50,000 faults in the pipeline registers for each of the 3 faulty tiles and 4 layers in about 288 hours. It is worth noting that the total number of faults injected depends on the module size.

Table 3.4: Microarchitectural Fault Injection Results

		Warp scheduler controller				**Pipeline registers**			
		SDC (%)	**DUE (%)**	**masked (%)**	**AVF (%)**	**SDC (%)**	**DUE (%)**	**masked (%)**	**AVF (%)**
	Z	4.43	7.89	87.68	12.32	0.63	1.12	98.20	1.75
Layer1	***M***	5.23	7.18	87.58	12.41	1.39	1.92	96.60	3.31
	R	5.65	7.83	86.52	13.48	1.34	1.93	96.73	3.27
	Z	4.03	7.78	88.19	11.81	0.28	1.92	97.80	2.20
Layer2	***M***	4.51	8.05	87.44	12.56	1.29	1.92	96.79	3.21
	R	5.65	7.83	86.52	13.48	1.14	1.92	96.94	3.06
	Z	2.85	7.78	89.38	10.63	0.24	1.94	97.83	2.18
Layer3	***M***	5.09	9.15	85.76	14.24	0.88	1.95	97.17	2.83
	R	5.56	7.89	86.54	13.45	0.89	1.94	97.15	2.83
	Z	3.57	7.25	89.18	10.82	0.21	1.92	97.85	2.13
Layer4	***M***	3.85	7.33	88.86	11.18	0.96	1.94	97.10	2.90
	R	3.98	7.20	88.82	11.18	0.75	1.95	97.31	2.70

Table 3.4 reports the Architectural Vulnerability Factor (AVF) [142] for the 3 analyzed tiles on the 4 layers of the CNN.

As it can be observed in the results, a fault in the SC is up to 10x more likely to corrupt the tile than a fault in the PRs, confirming its criticality. Nevertheless, the size of the pipeline module in FlexGripPlus is about 10x bigger than SC. The pipeline registers, then, are more likely to be corrupted by radiation or other sources of transient faults but the generated fault is more critical when affecting the scheduler.

More in detail, for the scheduler, the injections in the Z, M, and R tiles provide very similar AVFs (differences lower than 16.5% in most cases). These results suggest that using just a subset of tiles to characterize the layer reliability to faults in the scheduler can still provide accurate AVFs. Intuitively, the effect of faults in the control logic do not directly depend on the data being processed.

For the pipeline registers, the AVFs of R and M tiles are similar (differences lower than 32.5%) while the Z tile has a lower AVF. Previous works showed that datapath faults propagation does not significantly depend on the input values, unless the input is biased [139]. R and M tiles can be both considered not biased according to the definition of [139] (there is no clear data pattern), while Z is explicitly selected

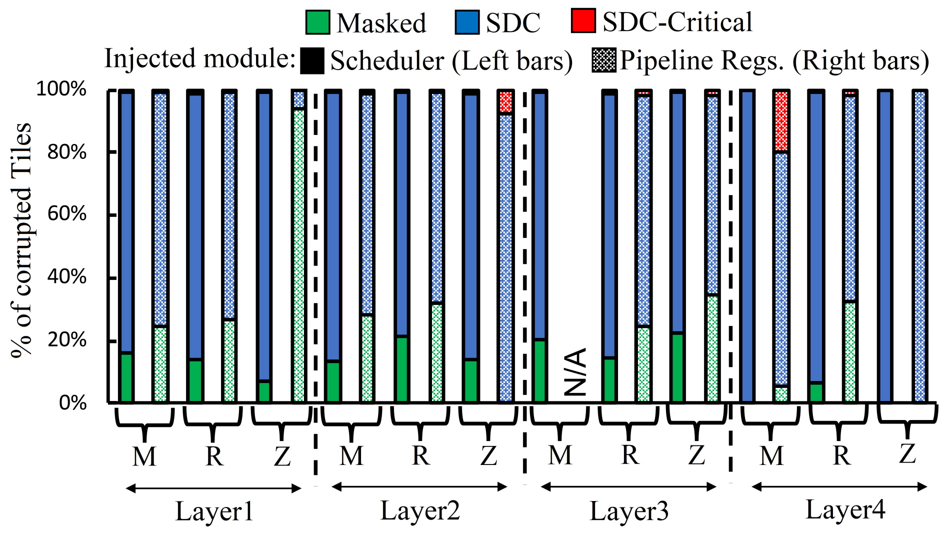


Figure 3.10: Corrupted Max (M), Random (R), and Zero (Z) tiles that propagate to LeNet output generating a critical/not critical SDC or are masked.

as it has an obvious pattern in the input. For data-path faults, then, it might be necessary to characterize more than one tile per layer. However, our data shows that the AVF of Z tiles is much smaller than R and M tiles, eventually making them intrinsically less vulnerable to faults. To have a worst case evaluation of the pipeline registers criticality it is then sufficient to characterize the R or M tiles.

Software-based propagation results

For each subset of tiles (M, R, Z), the propagated faults per tile are combined with the outputs of each layer. Then, in software, the corresponding corrupted layer outputs are operated in the CNN to understand their final impact behavior. The analyzes are focused on the different effects between the subset of tiles and on the differences between injections in different layers.

Figure 3.10 shows the probability for the corrupted tiles to be masked, to generate an error or a critical error (miss-classification) when propagating till the LeNet output. For each layer we have separated the contribution of M, R, Z tiles and distinguish between injections in the scheduler (left bars) and pipeline (right bar).

Interestingly, most corrupted tiles modify the output. However, few corrupted tiles induce a critical error (<2% for the SC and <20% for the PRs). This is to be expected, as the main functionality of LeNET is the classification of only 10 classes (digits from 0 to 9), so, it is hard for one architectural fault to change the output classification probabilities sufficiently to cause miss-classification.

For the first three layers, the propagation behavior is very similar for the three tiles confirming that, for the scheduler evaluation, there is no need to characterize all tiles. Only pipeline registers injections in the Z tile of layer 1 shows a higher masking effect. This is because the small magnitude of the error in the element of the tile is masked during propagation. The masking effect is reduced gradually in the downstream layers. On the other hand, Z tiles in layer 4 always produce an error. The lower masking effect is justified by the fact that layer 4 is the output layer and an injected corrupted tile implies a corrupted layer output. When errors are injected in the last layer, then, there is no masking effect caused by propagation.

One of the reasons for the tiles to generate a critical or not critical errors in the CNN operation can be observed in Figure 3.11. These distributions show, for the injection in the SC (PRs data is similar and not shown), the percentage of corrupted element at each layer output as the corrupted tile (injected in the layer identified by the color of the bars) is propagated through LeNET. Solid lines represent the injections that induce a critical error (miss-classification) and dashed lines the injections that induce a not critical error.

Fig. 3.11 reports only the average result for the three tiles with the error bars highlighting the differences between Z, R, and M. As it can be seen, the injection in all tiles provides very similar propagation patterns. Moreover, the reported results show that a corrupted tile represents a small percentage of the layer output in which it is injected. On the average corrupted tiles injected in layer 1 affect less than 0.1% of the layer elements, in layer 2 less than 0.2%. However, the number of corrupted elements explodes when the fault is propagated to the downstream layer, till affecting most (if not all) the elements in the network output. Interestingly, injections in the first 3 layers induce a critical errors in the output only if, propagating, they affect all the output elements. These layers perform operations (as convolutions) that can easily spread the error. Moreover, if only few elements are corrupted the last (fully connected) layer might still have sufficient data to properly classify the input.

Finally, only injections in the last layer (dark red bars in Figure 3.11) cause a critical error without affecting all the elements (58%, on the average). The value of those elements is eventually sufficiently wrong to modify classification.

Critical faults identification

The combination of microarchitectural and software fault injection allows to identify the faults that are more likely to modify the CNN output eventually modifying classification.

A first major difference between faults in the SC and in the PRs is that the former have a high probability of causing multiple errors in the tile output while the latter impact mostly one single element. Up to 87.3% of the fault in the SC but less than 20% of fault in the PRs corrupt multiple elements in the tile's output. This is because most memory cells in the SC are employed to store information of multiple threads in the GPU core or even about the entire block of threads. One

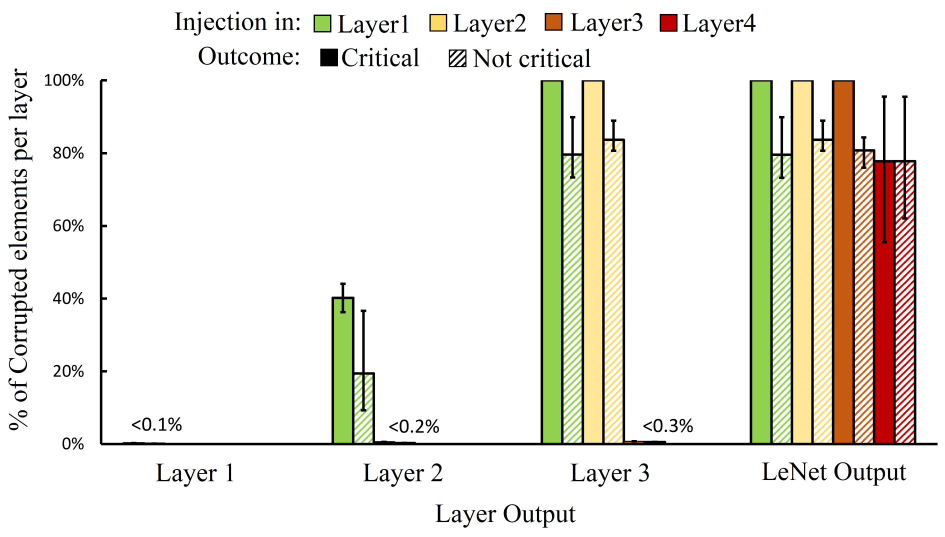
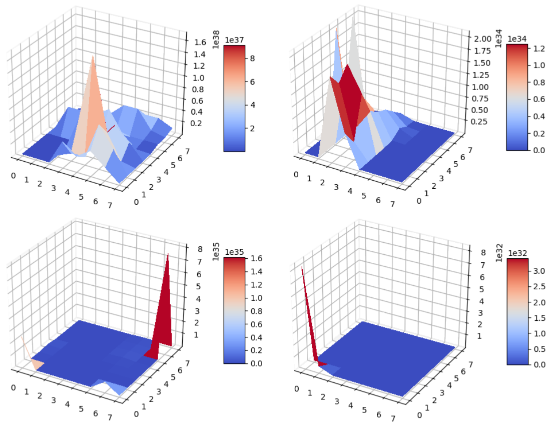


Figure 3.11: Percentage of corrupted elements in a layer output for injections in the layer identified by the bars color. As the fault propagates thought the layers the percentage of corrupted element significantly increases. Solid lines are critical errors, and dashed lines are not critical errors. Error bars report the maximum difference between Z, R, and M tiles.

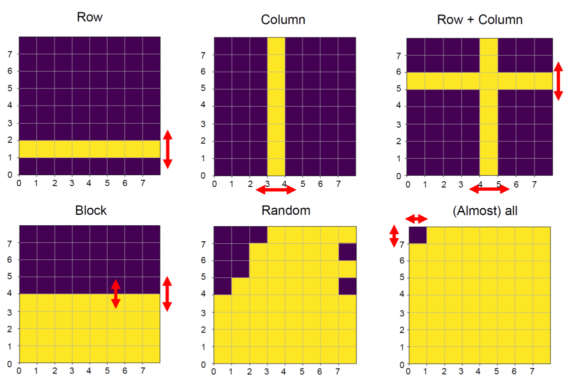
fault in the SC can then affect the execution or the data of multiple threads. In contrast, a fault in the PRs has more probabilities to affect the data-path of just one of the threads being executed in the corrupted core.

Figure 3.12a depicts, for each layer, the multiple error distribution caused by faults in the SC. each graph (in Figure 3.12a) represents the maximum difference between the expected and corrupted value of each element in the R tile (M and Z provided similar results). On the other hand, faults in the PRs produce differences of at most 10^3 without a clear pattern in the spatial distribution and are not depicted. Similarly, Fig 3.12b depicts the spacial distribution (per location in the output tiles) of the multiple corruption patterns observed during the fault injection campaigns. As can be observed, most corrupted locations are part of a group of threads that are being executed and interestingly present clear patterns of corrupted rows, columns or complete regions. These output patterns can be combined with the magnitude results and provide fault model that can be applied at higher abstraction levels when evaluating the reliability of CNNs using GPU platforms.

The faults in the SC can produce a very high difference (up to 10^{38}!) in some specific locations of the tile, that depend on the layer. For layer 1 and 2 the largest difference is identified in the middle locations of the tile, which is expected given the output tile size and the higher probability to propagate the fault effect into multiple threads (columns in the output tile). Similarly, the largest fault effects for layer 3 and 4 are identified in the corners of the output tile, which can be explained by the output size of the tiles and the layer composition, limiting the evaluation of multiple threads, and the low fault effect propagation to multiple threads caused by mostly zero values.



(a) Distribution of the output error magnitude produced by faults in the R tiles of layer 1 (Top-left), layer 2 (Top-right), layer 3 (Bottom-left), and layer 4 (Bottom-right).



(b) Spatial distribution of the multiple corrupted elements patterns. Arrows indicate that neither the position of the observed pattern nor the block size are fixed.

Figure 3.12: Magnitude (left) and spatial (right) distribution of the effect of faults in the scheduler of the GPU when operating tiles of the LeNET CNN.

Additionally, a deep analysis show that faults in the SC are on average more than 5x more likely to cause a DUE than faults in the PRs (Table 3.4). This is justified by the fact that, on GPUs, few control-flow data-dependent operations are executed to avoid dependencies that would reduce performances. Thus, it is unlikely for a data-path fault to generate a DUE while faults in the SC can indeed generate illegal memory requests, the stop of active threads or the activation of unexpected ones, including other misbehaviors that can collapse the execution of the device.

Finally, a feedback analysis is employed to the data from the fault campaigns to determine the microarchitectural locations causing the most severe critical SDCs in the CNN. For both SC and PRs, the critical locations store data employed to manage the execution of the GPU. For the SC, the analysis shows that a group of just 4 memory cells are the cause of 97.3% of the critical SDCs. A fault in these cells affects multiple values in the output and, in 35.14% of cases, even the entire tile output. These critical cells are devoted to store the addresses to access the Shared memory and the Register File per assigned block. The remaining 2.7% of critical SDCs are caused by faults in one register inside the SC. It was also found that only 7 locations in the PRs, when corrupted, generate multiple errors in the tile output. These locations store control signals of the operating instruction and addresses to access the Shared memory.

3.5 Conclusions

In this section several reliability evaluations about transient faults were described. The carried out fault injection campaigns provided detailed results about the sensibility to SEUs of individual modules of the GPU under different encoding styles (e.g., varying the number of threads or the parallelism). The evaluation was performed employing representative applications with diverse workloads to sensitize each module with different patterns. In some cases, it was possible to determine correlations with previous works. However, the existence of some inconsistency across the different applications and GPU configurations prompts for further investigation for evaluating specific modules in GPU devices against SEUs.

More in detail, as a major contribution of this chapter, a detailed microarchitectural evaluation is performed for the analyzed modules with respect to SEUs. The experiments were performed resorting to an extended GPU model and to some realistic applications.

In the Register File, a program kernel divided in several blocks contributes to reduce the effects of SEUs and promotes the increment of error masking, since it is possible that several blocks submitted to the same GPU core occupy and use the same register set for the operation of independent belonging to those different blocks. This behavior can be observed when the maximum number of blocks is submitted to the SM and a new block is ready for dispatching to the same SM (in different time periods), so one of the already submitted blocks and the new block share the same register set in the register file, as it happens in FlexGripPlus. Results also suggest that the distribution of threads per block may play a major role as, a zero hardware cost, mitigation strategy for SEU effects for applications with a high usage of data-path units, such as the Registers Files. On the other side, this distribution seems not to impact in a significant manner on the targeted control units.

A general conclusion of the analyzed modules in the GPU is that an increment in the number of threads per block seems to generate a higher SEU sensitivity on all data-path modules. This can be explained by the additional time to process all threads in a program and the higher occupancy of resources in the GPU. Nevertheless, the final percentage of rising sensitivity effect directly depends on the behavior of the parallel program and the instructions employed in its implementation. In contrast, in some control path modules of the GPU, the sensitivity to fault is affected (positively or negatively) when changing the number of threads.

The specific characteristics of each application may further change the above behaviors. Previous results gathered at the GPU level could not catch these aspects, which must be taken into account when optimizing an application code for performance, reliability, or in conjunction.

Finally, in section 3.4, the microarchitectural evaluation of reliability in critical modules of the GPU (i.e., the scheduler controller and the pipeline registers) allows the adaptation of the concept of multi-level fault injection that was employed to evaluate the microarchitectural effects of faults in GPU operating CNN workloads.

The proposed strategy was used to perform the experiments and reduced the total fault simulation latency into the evaluated workloads (convolutions) in the CNN. In the experiments, only a subset of convolution tiles was considered. From the performed experiments, it was shown that their propagation is very similar, so it is still possible to have a precise reliability evaluation for CNNs with the evaluation of a subset of tiles.

At the end, thanks to the adopted combination of microarchitectural and software fault injection, it was possible for the first time to quantitatively assess the effects of faults in GPUs critical modules during the execution of a CNN.

Chapter 4

Functional testing on GPUs: Software-based solutions for in-field fault detection

This chapter describes the main contributions in the field of functional testing of GPUs using the Software-Based Self-Testing (SBST) strategy. The proposed methods mainly target modules that are critical in the execution of a GPU. For this purpose, several custom techniques were explored targeting a critical module in the GPU core (the SC) for several fault models. Then, multi-kernel test strategies are proposed targeting the functional testing of modules that are dynamically reconfigured and hidden for the programmer. Finally, a modular approach to develop functional test programs is proposed. This modular approach is based on a higher abstraction level and exploits the main operational features of a module under test to propose several test solutions. This approach is validated on several embedded memories of a GPU.

It is worth noting that the in-field constraints and the operational restrictions of the target modules were considered in all proposed strategies. Moreover, compiler restrictions and other programming limitations of GPU architectures were also discussed in this chapter. Real GPU devices and the FlexGripPlus GPU model were employed in the experiments and in the validation of the proposed functional test techniques.

At the end of the chapter, classical test methods (such as deterministic and pseudorandom) are adapted and used to test the regular structures in GPUs, such as the functional units. Finally, a general overview of the fault coverage of all test strategies is provided, allowing us to observe the main benefits of the SBST approach in GPUs. This analysis also depicts the uncovered parts in the GPU by the proposed test strategies.

The first section targets the development of functional test solutions for the scheduler controller in the GPU. The second section describes the methodology to develop test programs aiming at the detection of faults in the pipeline registers using a multi-kernel approach, so providing a mechanism to test structures in the GPU that are dynamically configured for each application. The third section introduces a modular approach to efficiently develop test programs using a high abstraction level, so aiming at the selection and exploration of different test routines for a target structure under test. The fourth section describes classical test strategies that can be adapted from CPUs to GPUs and used for test of functional units and regular structures in GPUs. This section also provides a general overview of the benefits of SBST strategies applied on GPUs in terms of fault coverage and functional safety.

4.1 Functional test of the warp scheduler controller in GPUs

This section introduces functional test strategies to test the SC in GPU cores. As introduced previously, see chapter 2, the SC is a critical module in the operation of a GPU and any fault can affect the operation of a running application or even collapse the execution of the entire GPU.

In general, several works targeted the detection and identification of faults on GPU platforms. However, most approaches explored traditional application-based techniques, including the duplication and comparison of the complete program, memories or both [69, 143]. Other approaches proposed optimized programs by using a fine-grain solution aiming at identifying and detecting fault through duplicated warps [144]. In principle, this solution is effective to evaluate any data path module. Similarly, authors in [145], proposed a framework to exploit consecutive warps and possible divergences in a program to duplicate or triplicate the operations and perform detection and mitigation of faults in any data-path module. In [146], the authors proposed a methodology based on the duplication of instructions to perform the fault detection. However, in all these software-based detection strategies, the performance cost is a relevant restriction. For that purpose, in [146], the authors also proposed an ISA extension to optimize and reduce in up to 30% the comparison costs.

Authors in [147] explored the idea of employing idle SMs, commonly available during the execution of control-flow instructions, to detect through comparison faults affecting any SM. Other works explored the idea of manage the error resilience of internal modules, such as the Register file, employing compiler algorithms to provide fault mitigation on GPUs [148].

More in detail, several authors [124] [149], see also Chapter 3, already investigated the sensitivity features of complete GPUs to radiation effects, which are a major cause for transient faults. These works are mainly based on exposing real GPUs to accelerated radiation fluxes, and then observing the resulting effects. In this way, it is possible to estimate both the probability of a single fault to arise, and the fault effect caused to the application when a fault propagates through the system. Those conclusions are used to build application-based mitigation strategies, such as those introduced above. Furthermore, the conclusions can also be used to propose innovative hardware testing strategies.

Moving to permanent faults, achieving the required safety targets clearly mandates the adoption of special techniques to minimize the chances that possible faults created by the manufacturing process or by other mechanisms (e.g., aging) escape the different test procedures applied at the device, board and system level. Moreover, given the very high safety targets required, the advanced semiconductor technology used to manufacture current GPU devices, and the relatively short lifetime of these technologies in safety-critical applications, including *ADAS* systems, it is mandatory to develop efficient techniques to detect permanent faults (in-field test) before they cause critical failures.

In all cases, a test must be performed, which should be able to detect a very high percentage of those faults that may cause critical failures. In case of permanent faults, the test must be relatively fast, and must be performed taking into account the environment where the target device works, with minimum impact on the rest of the system. Due to these constraints, the SBST strategy has been widely adopted for in-field test [150], due to the flexible and non-invasive features of SBST. Moreover, several guidelines are provided by research works in the area [151]. The development of SBST test procedures can sometimes be done by abstracting detailed structural information of a module, when the function and architecture of the target module is known (e.g., for caches, or branch prediction units). In other cases, the development is based on these information for both developing the SBST code, and for assessing the achieved fault coverage.

In the case of GPUs, the development of effective SBST test procedures can be split in two parts:

- When targeting the computational cores of the GPU, one can use the techniques developed for traditional processor-based systems [151]. The work in [82] is an example of what can be done in this direction.
- More efforts are required to target those modules, which are specific of the GPU architecture, such as parallel controllers for execution and memory resources.

Authors in [82] proposed several techniques to test GPU devices employing special test programs exploiting well-known strategies from processor-based systems, such as pseudo-random algorithms to test the execution units and adaptations of march algorithms for internal memories. However, these mechanisms are not effective for all internal modules of a GPU. As an example, testing the scheduler existing in any GPU, as well as the interface between the scheduler and the computational cores, requires specific algorithms able to excite the different faults and to make them observable.

In other works [152] [153] [122] data and control modules are tested and analyzed by the presence of faults. These experiments included reliability characterization by statistical fault injection methods and checking the effect of software errors in the system. In [154] it is presented a methodology to mitigate hardware failures in GPUs. Moreover, a rescheduling strategy is introduced to ensure execution under hardware malfunction.

Similarly, the use of functional testing routines as self-test programs is increasingly common and now widely supported also by several semiconductor manufacturers and IP providers in the automotive field, such as STMicroelectronics [155], Infineon [156], Microchip [157], ARM [158] and Renesas [159].

The main focus of this section lies on the development of functional test techniques of the scheduler inside the cores of a GPU (also called warp unit). It is worth noting that there is a huge deficit of functional testing solutions targeting the scheduler controllers in the GPU architecture. Moreover, this scheduler is probably the most critical unit within a GPU core since it manages the execution of the parallel tasks and their distribution among the available cores, considering the memory resources in the device.

The implementation of this module is not known in detail for commercial GPU devices, thus FlexGripPlus is employed as a representative GPU to validate the proposed testing solution. We first evaluated the stuck-at fault coverage that can be reached on some specific parts of the scheduler and on the scheduler/core interface by simple programs running on the GPU (identifying also those faults, that cannot produce any failure, given a specific scenario), and then proposed some techniques to improve such a figure.

This section is divided into two parts. Subsections 4.1.1 and 4.1.2 describe a functional testing technique targeting the scheduler controller. On the other hand, Subsections 4.1.3, 4.1.4, 4.1.5, 4.1.6, 4.1.7, 4.1.8, 4.1.9, and 4.1.10 describes the development and validation of an on-line testing technique for the memory in the scheduler of the scheduler controller of the GPU.

4.1.1 A method for functional testing for the Scheduler Controller

The evaluation performed consists of a preliminary permanent fault coverage analysis of a simple benchmark application using the whole set of GPU assembly instructions. Secondly, three incremental methods to enhance the fault testing capabilities of any application implemented on a GPU were developed. The three methods are accurately described in this subsection showing the interdependence between the application execution flow (e.g., divergence and convergence paths), the thread mapping and the shared memory allocation.

Please note that for the purpose of this strategy, a fault can be labeled as detected at the end of the execution of a given piece of SBST code when one or both of the following conditions hold (called detection *by memory content* and *by performance degradation*, respectively).

In the first situation, the results, produced in memory by the fault-free and each faulty system, are compared. A fault is detected when the comparison produces a mismatch. In the second, the performance evaluation is based on two checks. Firstly, we check if the kernel execution was correctly completed and the simulator generated the results in memory. If the fault caused the system to hang or the kernel execution did not finish, the fault is labeled as detected. In the case of execution completed, we check and compare the kernel execution time (e.g., the number of clock cycles) with the fault-free one. In case of time mismatch, the fault is labeled as detected by performance degradation. In these evaluation it is possible that a fault could be detected by evaluation of the performance and memory mismatch. However, the checker priorities a fault detected by wrong results in memory instead of performance degradation.

Basic program behavior

We first selected a simple benchmark *(VectorAdd)* to evaluate the effects of permanent faults in the warp unit (i.e., in the warp pool memory and in the interconnections with the SM). This sample program performs an add operation between two vectors and generates a result vector. The benchmark is composed of 18 SASS assembly instructions. This simple program corresponds to a one dimension data intensive and embarrassingly parallel application. Despite its simplicity, it suitably stimulates the warp unit. The application is configured to use one grid and 256 threads per block. This implies that the size of the input vectors is limited to 256 operands.

Under the previous configuration, the number of warp pool line entries used is 8 out of the 32 available lines. Hence, we evaluated the effects of the program execution on the testable fields, i.e., the mask field and the program counter in the used warp pool line entries and in the interconnections interface.

A fault affecting the controls signals or the warp base parameters between the warp unit and the SM could stop or hang the system. As a consequence, the faulty SM hardware uses other system configuration parameters than the correct ones and the application never finishes or generates a valid result. A wrong base address for either the shared memory or the file registers could overlap operands from different threads and thus compromise the application execution. Furthermore, a fault in the actual thread state connections can generate performance issues, such as unexpected latency, or a mismatch in the final results in memory.

Permanent faults in the warp pool memory may also generate several issues. In the actual mask field, faults can produce two possible scenarios. In the first scenario, some threads preserve a permanent active state and never finish the kernel execution. In the second scenario, some threads preserve a permanent inactive state and are not allowed to execute instructions and write results in the global memory. This implies that some results will not be written to memory. In this case, a memory mismatch is produced at end of the program execution.

A fault in the execution program counter field can generate a hang in the thread execution and access to invalid locations in the program memory. Finally, a fault in other warp pool fields, which correspond to memory and register file base addresses configuration can cause the kernel to crash. However, some of these faults are observable but not controllable. These permanent faults belong to block scheduler configuration settings and a GPU functional testing approach cannot access to block settings, so those faults could be observed by the proposed observation mechanism but it is not possible to propagate and observe the effect of other faults.

Enhanced versions

Taking into account the previous analysis, three approaches were proposed to enhance an existing program (such as the one introduced in the previous subsection) to increase its ability to test the faults affecting the fields of a warp pool line entry and the connections between the warp unit and the SM.

These approaches heavily rely on the thread ID, or index, which uniquely identifies a thread during its execution which, in turn, depends on the kernel dimension and application complexity. Moreover, the proposed approach defines the distribution of blocks and threads in a grid and could be used by multiple threads to access multiple data locations (Multiple-Threads Multiple-Data or MTMD). This ID parameter is also used as a base address to load or store operands from different memory locations.

The proposed SBST approaches employ the thread ID to identify the number of a thread executed in a SP and to change the thread execution order. The change in execution generates divergence paths and the expected stimuli to check the actual mask field. The reader is invited to refer to Fig. 4.1 for a pseudocode with the different methods.

Method M1

This method uses the thread ID during the execution of an application without compromising the code integrity, so it is possible to add the test mechanism at the end or at the beginning of the application code. M1 is based on introducing a set of comparisons between the thread ID of each thread, and a set of constant values.

Firstly, the application is executed. Secondly, the thread ID and the constant value j are compared, and depending on the comparison, two possible divergence paths are generated in the SM. Each path is composed of different instructions, so

```
j ← 0                                          ▶ Clear constant
…                                              ▶ Normal app. Execution
Sig_per_thread[] ← 0                           ▶ Initialize signature       (M3)
for i ϵ {set of ThreadId in SM} do             ▶ Evaluate for every ThreadID
    if i == j then                             ▶ If ThreadID Matches
        Divergence_path_GroupA();              ▶ Divergence path Group A
        Thread_Store_in_memory();              ▶ Memory results store     (M2)
        Sig_per_thread[i] ← Sig_per_thread[i]+1 ▶ Set signature             (M3)
        Sig_store_in_memory();                 ▶ Store  signature (M3)
    else
        Divergence_path_GroupB ();             ▶ Divergence path Group B
    j ←j+1                                     ▶ Change constant value
```

Figure 4.1: Pseudocode for method M1 (white), M2 (white and light gray), and M3 (white and dark gray).

it is possible to detect a fault by comparing the execution time of the paths. In fact, each path is carefully described for this purpose. a fault is detected once a change in the expected thread execution time is detected. Then, the divergence is repeated to consider the missing fields of the warp pool lines. The total number of such repeated evaluations depends directly on the number of warps-per-block and threads-per-warp configured by the programmer according to the application.

A permanent fault in the mask field generates wrong divergence paths, leading to performance variation which can be used for fault detection. This technique only targets the permanent faults affecting the ID field of the warp pool line. However, some comparison routines can be suitably placed in the program memory in order to also detect faults affecting the bits belonging to the warp program counter.

Method M2

In this method, the divergence paths, generated by the comparison, are divided into two groups. The first thread group execute one or several store instructions in global memory to extend the performance variation. Moreover, these instructions allow the possibility of checking the global memory as mechanism for fault detection. On the other hand, the second group bypass the store instruction and executes NOP instructions, so reducing the latency of this path.

In the presence of a permanent fault in the mask field, unauthorized and permanently active threads will store the final memory results more than once. This will not affect the final result of the application in global memory but will cause a performance variation due to extra global memory accesses. On the other hand, a permanently inactive thread will never write to global memory, leading to some missing values in the final results of the global memory.

The same approach employed in M1 method is used to check the warp program counter fields in the warp pool line. Additional branches are also inserted to detect faults affecting the warp program counter. With this approach some interface interconnection faults can also be detected.

Method M3

The methods M1 and M2 depend on performance variation triggered by divergence paths in the test program to detect the permanent faults. However, in practical GPUs, these methods would require performance counters for detection, which are not necessarily implemented, fully available or easily accessible. To solve this issue, a variation of M2 is presented. In this approach, a mechanism of Signature-Per-Thread (SpT) is introduced. The SpT is intended to aim the propagation of any fault effect on one available memory output. In this approach, each thread of an applications uses special locations in memory to represent the actual state of a test program and propagate any detected fault effect.

Firstly, The SpT of each thread are initialized. Then, one exclusive memory location in global memory is assigned to each thread, so each particular memory location is devoted to store a SpT during the operation of the test program. One of the two divergence paths (i.e., convergence path) is equipped to perform the following procedures on the SpT: loading, updating, and storing in memory, as in M2.

The procedure of updating of an SpT can behave as a Multiple-Input Signature Register (MISR). Nevertheless, in M3 a compacted version of the SpT reduces the number of required instructions by updating the SpT as an incremental counter. The SpT increments each time the associated thread writes into memory (the expected value for every thread is one). A higher value implies that the thread has written more than once in memory. A signature with value zero implies that the thread never accessed the global memory. At the end, the detection of permanent faults is performed by checking the signatures after retrieving the information from the global memory in the final results.

4.1.2 Experimental Analysis

The FlexGripPlus simulation setup

In order to assess the effectiveness of the proposed techniques we performed some simulation-based fault injection campaigns using the FlexGripPlus model. The same tool introduced in Chapter 3 was adapted for the experiments. For the purpose of this work, the fault injection tool uses the stuck-at fault model to generate the equivalent commands for the simulation framework handling the GPU model.

Besides the check of the final results in global memory, a performance evaluation is also employed by comparing the actual and golden kernel execution time. A fault is labeled as detected by *“performance degradation (Time-Out)”* when the execution time, in the fault simulation, is longer or shorter than the expected value.

For fault injection purposes, a fault list is employed to feed the fault injector module in the tool. This fault list includes the fault model type and the location to inject. At the end of the fault injection campaign, the list of faults injected, labeled according to the two checks mentioned above, is gathered for coverage analysis as presented later in this section.

Implementation

For implementation purposes, the Instruction Set Architecture (ISA) of FlexGripPlus (SASS SM 1.0) was employed. The three SBST methods are applied to the original application code.

Method M1

In FlexGripPlus, the configuration stage of the SMs distribute the available registers in the Register File among the threads to be executed by the application. Then, the first register for each thread (R_0) is loaded with the thread ID (X, Y and Z). This register R_0 is used for each thread to access the memory resources. In this case, the Thread ID(X) was employed as part of the test program to address specific threads and manage the test execution. In this case, a movement operation from R_0 to an available and free register is included.

The total number of constants to compare depends on the blocks and threads per block configuration. For the present application, eight comparison routines are required, according to the total number of SPs available in the SM (8) for FlexGripPlus. The implementation of the M1 method code is composed of 91 assembly instructions, including the original application.

The routines employed are aimed to perform the following operations:

1. Selection of a convergence address point. An instruction memory address is defined as a convergence point for thread divergence.

2. Comparison between the thread ID register and a constant value. Each constant value is loaded using immediate instructions or through movements from shared or constant memories). Those threads with different thread ID values enter in a waiting state before reaching the convergence point.

3. Conditional branch execution. This generates the divergence paths, which indirectly generates the test patterns to be employed when evaluating a line of the scheduler memory.

```
...                    ► Application code
GLD Rx, R0.L           ► Move of threadIdx.x (stored in Register R0)
MVI Ry, Z              ► Move constant parameter per SP (from 0 to (Z-1))
...                    ► Application code
                       --------------- M1 code ---------------
AND Rx, Ry             ► Comparison between constant value and threadIdx.x

SSY Dir_1              ► Convergence point definition
BRANCH Dir_2           ► Conditional evaluation
NOP                    ► Divergence Path
NOP
Dir_2:GST M[Ra],Rb     ► Convergence Path, Storage thread results
Dir_1:  NOP.S          ► Warp branch stack release (Convergence point)
--- Repeat Z-1 times according to the number of threads per block.
--- End of M1 code.
```

Figure 4.2: General pseudocode for a routine implementing Method M1. Z is the number of threads per block or threads defined for the application.

4. Execution of divergence paths. Threads which match their Thread ID value with the constant value execute additional instructions. All threads converge again at the convergence point (determined in step 1) and continue the parallel execution.

   These operations are repeated for every constant value predefined in the test code, as illustrated in Fig. 4.2.

The number of conditional branch routines is directly related with the maximum number of threads per block and the number of threads defined in the application. These routines are placed at the end of the application code.

Method M2

This method exploits the same approach used in M1 and based on the thread ID index. However, the M1 test program is modified and updated by replacing the final global memory store instruction with a series of conditional routines followed with the global memory storage. In the proposed solution, 90 assembly instructions are employed.

In this approach, the first three operations are equal to the method described above. Nevertheless, the step 4 is replaced with several conditional branches, so generating the test patterns for the program counter fields in the memory and the interconnections of the SC. Each conditional branch targets strategical locations in the memory so changing the test pattern each time a new address is reached. One global memory storage is performed after each control-flow operation (by changing the memory address). In this case, a fault can be identified if the final results is never stored in the global memory by the fault effect. It is worth nothing that the combination of conditional branches and the execution of the store instructions generate additional latency to the test program. The operations are repeated as many times as M1.

Method M3

In this approach, a signature (SpT) is included as part of the test program. This signature is individual for each execution thread and requires additional space in the global memory. In this case, an input parameter to the parallel kernel is employed as base address for the final storage of the signature. The following operations are carried out replacing the final storage operation:

1. SpT initialization. For every active thread, the signature in the global memory is initialized with a value zero.
2. Selection of a convergence point.
3. Comparison between thread ID and a constant value, generating divergence paths of thread execution.
4. Threads whose thread ID matches the constant value perform directly a final memory store operation, load the thread signature, increase once, and store back into global memory.
5. Other threads (divergent ones) execute additional instructions and one or more unconditional branch instructions to reach the convergence point.

In FlexGripPlus, the code is implemented using assembly language with 152 instructions. However, the proposed approaches can also be mapped and developed at a higher level (e.g., using CUDA) with the use of *switch* and consecutive *if* statement and the *ThreadIdx.x* parameter. An example of routine implementation of the method M2 at high level is presented in Fig 4.3.

When implementing the test code in CUDA, in-line PTX operations are required to allow a finer control of the branch instructions. However, the most control-flow instructions are complex to manage from the PTX level. Thus, instrumentation with SASS instructions is still needed (after PTX-SASS translation) to guarantee the intended operation of the test program. It must be noted that the main target of the high level compilers of GPUs are devoted to optimize performance operation and compact the code length, so are not fully suitable for testing purposes.

Experimental Results

Two fault simulation campaigns have been performed on the GPU model using the fault list of the warp pool memory lines (2,048 elements) and the interface

```
...                          ► Normal application code
switch(threadIdx.x)          ► Comparison of threadIdx.x
{
  case Z:                    ► Thread execution for threadIdx.x with Z value
   if (true)                 ► Branches to produce test patterns for the program
     if (true)               counter field
       ...
       Thread_final_Store(); ► Store of results in global memory
       break;
  ...                        ► Comparison with other Z-1 value
}                            ► End of M2 code
```

Figure 4.3: Pseudo-CUDA-code for method M2. Z is the number of threads per block or thread in the application.

connection (478 elements). FlexGripPlus has been configured with one grid, 256 blocks and 32 threads per warp. Table 4.1 presents the results of the fault simulation campaign for the warp pool memory lines. The original application (VectorAdd) requires 142.005µs. of simulation time. Algorithms M1, M2 and M3 require 142.005 µs, 415.215 µs and 1.122 ms respectively.

Table 4.1: Fault detection results in warp pool memory.

Application Code	VectorAdd	M1	M2	M3
Total faults		2,048		
Testable faults		984		
Detected faults	624	728	984	984
Hang	440	613	616	616
Memory mismatch	184	115	112	368
Performance degradation	0	0	256	0
Testable Fault coverage (%)	63.4	74.0	100.0	100.0
Fault coverage (%)	30.5	35.5	48.0	48.0

During the fault simulation experiments on the warp pool memory, it was possible to identify several cells in the structure that once affected by faults do not cause any effect on the outputs of the GPU cor, so those fault effects were masked. Moreover, it was possible to observe that these cells could not be accessed by any available instruction in the ISA of the FlexGripPlus. A detailed analysis revealed that some permanent faults are functionally untestable (faults that cannot be activated by instructions, but also those do not affect the main functionality of the device), such as the bits in the higher part of the warp ID and some bits composing the base address to access memory resources, including the shared memory and general-purpose registers. In fact, these fields remain constant or are defined as constant values during a parallel kernel execution and are inaccessible. Taking into account those restrictions, the total number of untestable faults per entry line is equal to 156 permanent faults. Furthermore, 23 faults are observable but not controllable. These faults correspond to the base addresses of the general purposes registers file and shared memory space of every thread, so for the application with eight warps entry lines, the total amount of untestable faults is 1,064.

As reported in Table 4.1, method M1 increases the number of detected faults comparing to the original application (VectorAdd), while method M2 is capable to detect all the testable permanent faults in the warp pool memory. The 256 permanent faults detected by memory mismatch in M2 are all related to the actual mask field for the eight lines used by application. It means that all permanent faults in the actual field are detected. The same numbers of permanent faults in program counter, detected by method M1, are detected also by M2.

The method M3 aims at a different purpose. This approach allows fault detection in the actual mask field employing a different observation mechanism, i.e. checking the final results in the global memory only. Hence, it does not require the use of additional or complex hardware for performance or timing measurement during test.

A new analysis of the interconnection signals shows that a total of 201 connections are not relevant for kernel execution and thread control in the GPU and are classified as untestable faults for the proposed methods. In Table 4.2, the results of the fault injection campaign regarding the interconnections between the warp unit and the SM is reported. M1 increases the fault detection coverage by 7.94% as it is able to detect those faults affecting the bits carrying the thread state information between the warp unit and the branch unit to control the branch execution. M2 further increases the fault coverage to 85.92% instead of 100%, as connections regarding shared memory size, general-purpose registers size or number of warps do not generate misbehavior in FlexGripPlus operation.

Table 4.2: Fault detection results in the interconnections of the scheduler controller of the GPU.

Application Code	**VectorAdd**	**M1**	**M2**	**M3**
Total Fault	478			
Testable Faults	277			
Detected Faults	155	177	238	236
Hang	105	157	154	161
Memory Mismatch	50	20	20	75
Performance degradation	0	0	64	0
Testable Fault Coverage (%)	56.0	63.9	85.9	85.2
Fault Coverage (%)	32.4	37.0	49.8	49.4

Faults detected by performance degradation correspond to the signals between the warp unit and the Execution/Control stage in SM which are in charge of preserving execution coherency between the two modules.

The method M3, without checking Performance degradation, is still able to achieve almost the same fault coverage by only checking the final results in global memory. Fault coverage is increased by 29.25% comparing to the original application, which means that 81 additional permanent faults are detected: 64 of them belong to the actual thread state and the other faults belong to the program counter of the parallel threads.

In M2 some permanent faults are detected with different observation mechanisms. Some faults affecting the thread program counter and base address of the general-purpose register file previously detected by memory mismatch become detected by Hang or Performance observation. Faults related to bus control signals, memory and general-purpose register file base addresses are detected by Performance degradation in M2.

Finally, M1 is able to detect some faults in the warp pool memory and in the interconnections; however, the percentage of fault coverage is low. On the other hand, M2 achieves higher fault coverage by introducing store instruction to access global memory to increase performance variation among different divergence paths. Meanwhile, M3 achieves similar high fault coverage by only checking the final results in global memory, taking advantage of a signature variable for each thread.

4.1.3 An on-line testing technique for the scheduler memory of a GPU

This section describes a technique to generate functional self-test programs targeting the detection of faults in the memory of the SC of a GPU. The targeted faults are permanent under the stuck-at and coupling fault model between pairs of the cells of the memory. The proposed technique translates Fault Primitives (FPs), which represent the effect of faults in a memory cell, into self-test functions and programs composed of a sequence of operations to excite the fault in the memory and to propagate its effects to a visible location, thus detecting its presence.

The proposed strategy focuses on the memory in the SC because it represents a crucial module for the device operation. A micro-architectural analysis of the SC shows that most of its area is devoted to storing information in a status memory. This information is continuously updated after each instruction cycle in the GPU. Furthermore, this status memory is present in each Streaming Multiprocessor (SM) of a GPU. The experimental results to validate the proposed method have been gathered resorting to a commercial profiler (NVIDIA Visual Profiler and the Nsight Debugger), using the NVIDIA-GEFORCE GTX GPU platform and a structural fault simulator. The CUDA programming environment was used to implement most of the proposed test procedures.

In principle, the traditional functional test strategies implement March algorithms (composed of March elements) for memory testing. One March element is a sequence of reading and writing operations performed on all the memory words in a specified order. Each sequence generates specific pattern values to be written and evaluate those read from memory. In order to designs these sequences, FPs are used. These FPs represent the faults affecting a memory cell, and these are employed to design the test patterns as self-test procedures. Moreover, the collection of self-test procedures (composing libraries) can be activated when required (e.g., at the power-on, power-off, or periodically), generating test patterns and exciting a target module. Moreover, these routines check the test results and trigger an observability mechanism, such as an interrupt flag or a software exception, when a fault is detected. Thus, it would be possible to adapt these strategies and propose unconventional methods to evaluate critical modules, such as the memory in the SC, which is specific to GPU architectures.

The proposed method is based on the definition of a set of FPs describing static and permanent faults in memory. These FPs are customized to the scheduler and translated considering the operational restrictions of the memory in the scheduler. Then, the FPs are used to extract the corresponding test patterns (TPs), i.e., the sequence of reading and writing operations. These TPs maps into high-level self-test routines or functions for the GPU, thus generating the test programs. Finally, the same mapping and translation process is performed from March elements into self-test routines, thus providing the same fault detection coverage of the original March elements. In the end, this method can translate any element of a March algorithm targeting the status memory of the scheduler into a self-test procedure. A general scheme describing the proposed approach is shown in Fig. 4.4.

The main contributions of the proposed method can be summarized as:

- The identification of the FPs for all faults (including single and multiple coupling faults) in the status memory of the warp scheduler controller of a GPU.

- A method to translate, map and adapt each FP (and associated test patterns) or March element into self-test routines and high-level functions targeting the detection of all permanent static faults in the memory cells of the status memory in the warp scheduler of a GPU.

- A software mechanism to avoid the operation of the dispatcher units in the SM during the execution of a program kernel in the GPU.

- A method to employ the dispatcher units in an efficient manner by using a parallel approach to test the memory in the warp scheduler controller.

- A method to design self-test routines by only using a high-level abstraction language by means of the CUDA programming environment.

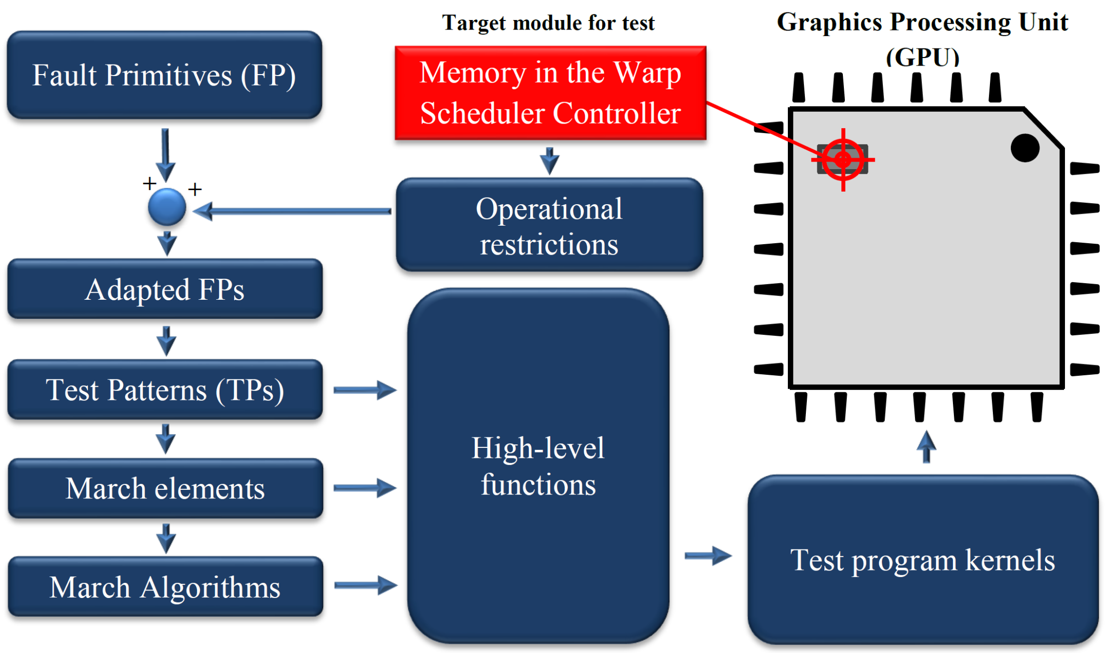


Figure 4.4: General scheme of the proposed approach for mapping TPs targeting FPs into test kernels.

The organization of this section is the following. Subsection 4.1.4 overviews the effects of permanent faults in the memory of the scheduler for generic applications. subsection 4.1.5 introduces the fault primitives (FPs) for a generic memory and describes the main features of the memory and its operational restrictions. Section 4.1.8 defines the methods to generate test patterns for the target memory employing software-based approaches. Section 4.1.9 describes the test pattern generation targeting each memory field and presents a test case algorithm, detailing the general implementation of TPs using high-level functions in the CUDA environment. Section 4.1.10 reports on the validation we performed to assess the effectiveness of the proposed techniques and introduces an alternative implementation method for performance optimization.

4.1.4 Effects of permanent faults in the memory of the Scheduler controller

A fault located in the memory of the scheduler can generate misbehaviors which may seriously compromise the correct operation of the device. Some previous works reported that faults affecting this module may have a critical impact on the system execution and can cause wrong memory results or even system hanging [108]. Nevertheless, those works only targeted transient fault effects.

In order to extend this conclusion and evaluate the effects of permanent faults, fault simulation experiments on a GPU model (FlexGripPlus) were performed using four representative applications (*Vector_add*, *FFT*, *Edge* and *MxM*). Using this model it is possible to inject a permanent fault in each module location and observe the fault effects during the execution of the selected application. Each memory location includes the Actual Thread Mask (TAM) and the Warp instruction Program Counter (WPC) fields. Some other fields are also included in the memory and are related to the configuration parameters coming from the host or external schedulers.

It is worth noting that due to the configuration of the applications, all the above applications employ integer operands, only.

The fault injection campaign was performed employing the RTL level description of FlexGripPlus. The same fault injection environment employed in section 4.1.1 was used to target the warp memory in the scheduler of one SM. Thus, the fault injector can place permanent faults corresponding to stuck-at faults affecting single bits of the SC memory.

The fault simulator uses a multi-threading fault injection approach and each fault list was divided into two pieces. Eight fault campaigns were performed injecting 4,096 randomly sampled permanent faults per campaign. According to [160] this allows us to reach a 2% error margin on the estimated metrics with a 99% confidence level. A summary about fault effects is presented in Table 4.3, where the rightmost column reports the cumulative percentage of the faults in the SDC, Hang and Timeout categories. As it can be observed from the results, a permanent fault can generate different effects depending on the features of the application and used GPU resources.

In general, a permanent fault affecting the WPC field generates mainly hanging conditions. On the other hand, faults affecting the TAM field generate most of the SDC conditions.

A high percentage of faults (56.5% to 85.9%) affecting the scheduler memory produces a failure, proving the criticality of this unit in the operation of the GPU. Moreover, these faults may generate unacceptable conditions for complex applications and some of these effects can be unacceptable for safety-critical applications. Other kinds of faults (e.g., Coupling Faults) affecting the scheduler memory do produce similar results.

4.1.5 Fault primitives for the memory in the scheduler

The FPs represent memory failures and are defined as the combination of a sequence of memory operations and the observations, including deviations from the expected value. This sequence of memory operations are employed to sensitize a condition in the memory cell and may also be employed to verify and to detect any possible failure in a memory cell.

An FP is composed of one or a sequence of memory operations, which can be

Table 4.3: Effects of permanent faults affecting the memory in the scheduler controller.

Benchmark	**Faults (%)**				
	SDC	**Hang**	**Timeout**	**Silent**	**Cumulative Total**
VectorAdd	21.87	35.93	0.0	42.18	57.81
FFT	34.37	51.56	0.0	14.07	85.93
Edge	9.37	62.50	7.03	21.10	78.90
MatrixMul	1.90	54.68	0.0	43.46	56.54

writing or reading, and the observed effect on the cell. In reading operations, it is common to include the logic output value as a third element. The FPs may target multiple sets of functional faults that can affect a single memory cell and also couples of interfering cells in a memory.

The FPs have been used in the past to define memory test techniques, i.e., March algorithms, able to generate an appropriate sequence of patterns (reading and writing operations), thus testing a memory. The complexity of these algorithms grows proportionally to the size of the memory. Additional details regarding a complete theoretical description of memory functional fault models may be found in [161]. For the purpose of this work, we target the procedures required to translate FPs and March algorithms into functional self-test programs able to detect faults in the scheduler memory of GPUs.

Single Cell Static faults

Considering the set of FPs presented in [161], Table 4.4 reports the full set of static FPs for single memory cells. The term "static" refers to the fact that they represent faults sensitized by a single memory operation. Each row includes the Addressable Functional Fault Primitive for the scheduler, denoted as AFFP(SCH). As it can be noted, the AFFP(SCH) also contains the initialization steps required to sensitize the fault.

Table 4.4: Fault primitives for a single memory cell.

Fault type	**Fault Model**	**FP**	**AFFP(SCH)**
DRDF	Deceptive RDF	$< \bar{A}r_{\bar{A}}/A/\bar{A} >$	$< \bar{A}, (W_{\bar{A}}, r_{\bar{A}}, r_{\bar{A}}, A/\bar{A} >$
		$< Ar_A/\bar{A}/A >$	$< A, (W_A, r_A, r_A), \bar{A}/A >$
TF	Transition fault	$< \bar{A}W_A/\bar{A}/- >$	$< \bar{A}, (W_{\bar{A}}, r_{\bar{A}}, W_{\bar{A}}, r_{\bar{A}}, \bar{A}/A >$
		$< AW_{\bar{A}}/A/- >$	$< A, (W_A, r_A, W_{\bar{A}}, r_{\bar{A}}, A/\bar{A} >$
WDF	Write destructive fault	$< \bar{A}W_{\bar{A}}/A/- >$	$< \bar{A}, (W_{\bar{A}}, r_{\bar{A}}, W_{\bar{A}}, r_{\bar{A}}, A/\bar{A} >$
		$< AW_A/\bar{A}/- >$	$< A, (W_A, r_A, W_A, r_A, \bar{A}/A >$
RDF	Read destructive fault	$< \bar{A}r_{\bar{A}}/A/A >$	$< \bar{A}, (W_{\bar{A}}, r_{\bar{A}}, r_{\bar{A}}, A/\bar{A} >$
		$< Ar_A/\bar{A}/\bar{A} >$	$< A, (W_A, r_A, r_A, \bar{A}/A >$

In Table 4.4, "A" represents a logic test stimulus written in the target bit-field or cell of a LE. Similarly, "$\bar{A}$" is the complementary pattern. Each AFFP is organized as follows:

$$AFFP =< Initial\_condition, (Stimuli), Fault\_value, Fault\_free\_value > \tag{4.1}$$

For example, considering the DRDF, the FP is expressed as: $FP :< \bar{A} > r_{\bar{A}}/A/\bar{A}$, with ($\bar{A} > r_{\bar{A}}$) as the initial sequence of operations in the cell (the initial state and one reading operation), the second element (A) describes the effect of the fault in the cell, and the last element ($\bar{A}$) represents the logic output value of the cell. Similarly, the associated AFFP(SCH) is defined as: $AFFP(SCH) :< \bar{A}, (W_{\bar{A}}, r_{\bar{A}}, r_{\bar{A}}), A/\bar{A} >$. This expression employs the same initial logic state ($\bar{A}$). The second and third memory operations in the sequence ($W_{\bar{A}}, r_{\bar{A}}$) are added and employed to initialize the cell. The reading is included as one of the operational restrictions presented in the target memory. The fifth element(A) represents the effect of the fault in the cell, and the last parameter ($\bar{A}$) is the fault-free logic output value, which is employed during the test pattern generation process.

Coupling Cell Permanent faults

The coupling FPs are associated with the interaction and effect between two independent cells, an aggressor (a) cell, and a victim (v) cell. These cells can belong to the same Line Entry (LE) or not. The considered FPs are shown in Table 4.5. X, Y, and Z represent logic values.

In the proposed approach, the State Faults (SF) FP $< \bar{A}/A/- >< A/\bar{A}/- >$, the State Coupling Faults (CFst) FP $< \bar{A}; \bar{V}/V/- >< \bar{V}/V/- >< \bar{A}; V/\bar{V}/- >< A; V/\bar{V}/- >< A; \bar{V}/V/- >$, and the Incorrect Read Faults (IRF) FP $< \bar{A} >< A >< r_{\bar{A}}/\bar{A}/A >< r_A/A/\bar{A} >$ are not considered, mainly because of the absence of a clear mechanism to cause the initial conditions or to detect these faults in the scheduler memory.

The AFFP(SCH)s in both cases, single and coupling, present some similarities in the associated Sensitizing Operation Sequences (SOSs). Thus, it is feasible to collapse identical patterns. The single-cell AFFPs of RDF and DRDF share the same sensibility patterns. Similarly, some coupling faults (CFrd, CFir, and CFdrd) were grouped using the same SOS, since the only difference among them is the number of consecutive reading operations. Therefore, the SOSs with the lowest number of reading operations were neglected and the CFdrd SOS is employed to sensitize those coupling faults. In the end, the number of patterns was reduced to 30.

At this point, the AFFP(SCH) is partially complete and must be adapted considering the operational restrictions valid for the SC. The next sub-section describes the operational restrictions of the target memory.

Table 4.5: Static Fault Primitives for coupling cells in memory.

Fault type	**Fault model**	**FP**	**AFFP(SCH)**
CFtr	Transition coupling fault	$< X^a0, W_1^v/0^v/- >$	$< \bar{X}^a0^v, (W_{\bar{X}}^a, r_{\bar{X}}^a, W_0^v, r_0^v, r_{\bar{X}}^v, r_0^v), 0^v, X^v >$
			$< X^a0^v, (W_X^a, r_X^a, W_0^v, r_0^v, r_X^v, r_1^v), 0^v, X^v >$
		$< X^a1, W_0^v/1^v/- >$	$< \bar{X}^a1^v, (W_{\bar{X}}^a, r_{\bar{X}}^a, W_1^v, r_1^v, r_{\bar{X}}^v, r_0^v), 1^v, X^v >$
			$< X^a1^v, (W_X^a, r_X^a, W_1^v, r_1^v, r_{\bar{X}}^v, r_1^v), 1^v, \bar{X}^v >$
CFds	Disturb coupling faul	$< X^aZ^v, W_y^a/\bar{Z}^v/- >$	$< \bar{X}^a\bar{Z}^v, (W_{\bar{X}}^a, r_{\bar{X}}^a, W_{\bar{Z}}^v, r_{\bar{Z}}^v, W_{\bar{X}}^a, r_{\bar{X}}^a, r_{\bar{Z}}^v), Z^v, \bar{Z}^v >$
			$< \bar{X}^aZ^v, (W_{\bar{X}}^a, r_{\bar{X}}^a, W_Z^v, r_Z^v, W_{\bar{X}}^a, r_{\bar{X}}^a, r_Z^v), \bar{Z}^v, Z^v >$
			$< X^a\bar{Z}^v, (W_X^a, r_X^a, W_{\bar{Z}}^v, r_{\bar{Z}}^v, W_{\bar{X}}^a, r_{\bar{X}}^a, r_Z^v), Z^v, \bar{Z}^v >$
			$< X^aZ^v, (W_X^a, r_X^a, W_Z^v, r_Z^v, W_{\bar{X}}^a, r_{\bar{X}}^a, r_Z^v), \bar{Z}^v, Z^v >$
			$< X^a\bar{Z}^v, (W_X^a, r_X^a, W_{\bar{Z}}^v, r_{\bar{Z}}^v, W_X^a, r_X^a, r_{\bar{Z}}^v), Z^v, \bar{Z}^v >$
			$< X^aZ^v, (W_X^a, r_X^a, W_Z^v, r_Z^v, W_X^a, r_X^a, r_Z^v), \bar{Z}^v, Z^v >$
		$< X^aY^v, r_X^a/\bar{Y}^v/- >$	$< \bar{X}^a\bar{Y}^v, (W_{\bar{X}}^a, r_{\bar{X}}^a, W_{\bar{Y}}^v, r_{\bar{Y}}^v, r_{\bar{X}}^a, r_{\bar{Y}}^v), Y^v, \bar{Y}^v >$
			$< \bar{X}^aY^v, (W_{\bar{X}}^a, r_{\bar{X}}^a, W_Y^v, r_Y^v, r_{\bar{X}}^a, r_Y^v), \bar{Y}^v, Y^v >$
			$< X^a\bar{Y}^v, (W_X^a, r_X^a, W_{\bar{Y}}^v, r_{\bar{Y}}^v, r_X^a, r_{\bar{Y}}^v), Y^v, \bar{Y}^v >$
			$< X^aY^v, (W_X^a, r_X^a, W_Y^v, r_Y^v, r_X^a, r_Y^v), \bar{Y}^v, Y^v >$
CFwd	Write destructive coupling fault	$< X^aY^v, W_X^v/\bar{Y}^v/- >$	$< \bar{X}^a\bar{Y}^v, (W_{\bar{X}}^a, r_{\bar{X}}^a, W_{\bar{Y}}^v, r_{\bar{Y}}^v, W_{\bar{Y}}^v, r_{\bar{Y}}^v), Y^v, \bar{Y}^v >$
			$< X^a\bar{Y}^v, (W_X^a, r_X^a, W_{\bar{Y}}^v, r_{\bar{Y}}^v, W_{\bar{Y}}^v, r_{\bar{Y}}^v), Y^v, \bar{Y}^v >$
			$< \bar{X}^aY^v, (W_{\bar{X}}^a, r_{\bar{X}}^a, W_Y^v, r_Y^v, W_Y^v, r_Y^v), \bar{Y}^v, X^v >$
			$< X^aY^v, (W_X^a, r_X^a, W_Y^v, r_Y^v, W_Y^v, r_Y^v), \bar{Y}^v, X^v >$
CFrd	Read destructive coupling fault	$< X^aY^v, r_X^v/\bar{Y}^v/\bar{Y}^v >$	$< \bar{X}^a\bar{Y}^v, (W_{\bar{X}}^a, r_{\bar{X}}^a, W_{\bar{Y}}^v, r_{\bar{Y}}^v, r_{\bar{Y}}^v, r_{\bar{Y}}^v), Y^v, \bar{Y}^v >$
			$< X^a\bar{Y}^v, (W_X^a, r_X^a, W_{\bar{Y}}^v, r_{\bar{Y}}^v, r_{\bar{Y}}^v, r_{\bar{Y}}^v), Y^v, \bar{Y}^v >$
			$< \bar{X}^aY^v, (W_X^a, r_X^a, W_Y^v, r_Y^v, r_Y^v, r_Y^v), \bar{Y}^v, X^v >$
			$< X^aY^v, (W_X^a, r_X^a, W_Y^v, r_Y^v, r_Y^v, r_Y^v), \bar{Y}^v, X^v >$
CFir	Incorrect read coupling fault	$< X^aY^v, r_X^v/Y^v/\bar{Y}^v >$	$< \bar{X}^a\bar{Y}^v, (W_{\bar{X}}^a, r_{\bar{X}}^a, W_{\bar{Y}}^v, r_{\bar{Y}}^v, r_{\bar{Y}}^v), Y^v, \bar{Y}^v >$
			$< X^a\bar{Y}^v, (W_X^a, r_X^a, W_{\bar{Y}}^v, r_{\bar{Y}}^v, r_{\bar{Y}}^v), Y^v, \bar{Y}^v >$
CFdrd	Deceptive read destructive CF	$< X^aY^v, r_X^v/\bar{Y}^v/Y^v >$	$< \bar{X}^aY^v, (W_{\bar{X}}^a, r_{\bar{X}}^a, W_Y^v, r_Y^v, r_Y^v), \bar{Y}^v, Y^v >$
			$< X^aY^v, (W_X^a, r_X^a, W_Y^v, r_Y^v, r_Y^v), \bar{Y}^v, Y^v >$

4.1.6 Operational restrictions of the memory in the scheduler

As said previously, this memory has some operational constraints. These are considered in the FP adaptation process and are crucial to generate the test patterns. The memory in the scheduler cannot undergo any possible operation or sequence of operations. In particular, the following operational restrictions are present (for each of them, we first describe the restriction, then summarize the effects, and finally describe how to take it into account):

1. By default, the initial state of all threads in a warp is the active state. It

means that the TAM field starts with a “1” value in all bits. This condition is triggered during the GPU configuration phase.

Restriction effect: This condition reduces the available number of test patterns to be applied and also forces the addition of an initial condition during test pattern injection, so reducing the performance of the test.

Potential solution: The initial conditions cannot be avoided. Nevertheless, it is possible to split the target test fields during a test procedure and apply multiple test patterns replacing the effect of the missing one. Multiple injections procedures are required for this purpose.

2. In the TAM field, when a warp ends the instruction execution, the scheduler updates the content of the associate LE including an implicit reading procedure. In contrast, writing procedures can be forced each time an instruction starts its execution or when the application flow changes the number of active threads in a warp.

   Restriction effect: This condition complicates the injection of consecutive specific writing and reading operations in the LEs. The implicit reading operation may increase the complexity of the injection of some specific operations in this field.

   Potential solution: In this memory, the operational restriction cannot be avoided. However, this restriction does not affect the testing procedures of static coupling faults.

3. Stimulus applied into the TAM field can generate thread divergence causing a program-flow division into two paths (the *Taken* and the *Not-Taken*). These paths are managed by the SC and executed in a serial fashion.

   Restriction effect: This condition adds some undesirable writing and reading operations and latency. Moreover, initialization procedures may be required between injections of consecutive patterns.

   Potential solution: This restriction cannot be avoided. Nevertheless, the additional path *Not-Taken* can be neglected for the purpose of the coupling faults test and considered as a transition of the initialization phase for the next pattern injection.

4. The execution of a divergence path (*Taken* or *Not-Taken*) cannot be stopped when a thread group is operating in the SM resources. The use of conventional functions has no effect in stopping these threads.

   Restriction effect: This condition involves the development of additional mechanisms to manage the operations of stop and restart of warps located in the LEs. Moreover, this behavior imposes restrictions in test patterns selected for the TAM field. Finally, multiple writing and reading procedures are required to test the field considering that the imposed restrictions limit the injection pattern and at least one active state bit should be maintained during evaluation.

Potential solution: A method to stop the warp execution lies in resorting to some thread synchronization methods (i.e., __syncthreads()) and semaphore variables to identify the warp state. Nevertheless, at least one thread must remain in the active state in order to retain the warp in hanging condition with the possibility to restart it again. On the other hand, a LE with the TAM field filled with inactive threads denotes a finished warp and cannot be launched again for this program kernel.

5. The GPU has two warp dispatcher units. These units dispatch the available warps into the SM resources based on performance and optimization features, by means of complex data-hazards mechanisms and elaborated dispatching policies. Thus, the execution order of a group of warps may be complex to predict. Moreover, clear structural descriptions and internal operational details are not provided by the device manufacturer [162] [34].

   Restriction effect: This behavior may compromise the execution of test patterns in consecutive LEs and adds latency among the injection of patterns into the memory LEs.

   Potential solution: software-based mechanisms must be employed to skip the operation of these modules in order to obtain the expected interaction behavior among consecutive LEs during test injection. This mechanism can be based on the addition of semaphore variables and internal loops to maintain the state condition of a target warp and the associated LE.

4.1.7 Adapting FPs to test the memory in the scheduler

We can identify a test pattern for each one of the considered FPs. This pattern is directly derived from the associated AFFP. The following example shows the steps required to generate and adapt the test patterns from the associated FP. Consider the FP:

$$FP :< X^a Z^v, W_y{}^a/\bar{Z}^v/- > \tag{4.2}$$

In this example, this FP describes one disturb coupling fault between two cells (a) and (v) with an initial logic state X and Z, respectively. A writing process in the aggressor (a) cell generates the logic toggle of the victim (v) cell. The associated AFFP is based on:

$$FP :< X^a Z^v, W_x{}^a/\bar{Z}^v/Z^v > \tag{4.3}$$

From the AFFP it is possible to derive an initial test pattern (TP). The TP adds the condition to check the state of the (v) cell and is defined as:

$$TP :< X^a Z^v, (W_x{}^a, r_z{}^v) > \tag{4.4}$$

The first two terms in 4.4 represent the initial logic state in (a) and (v) cells. The consecutive terms describe the patterns to evaluate the fault. Thus, the TP starts a writing process in the (a) cell, to generate the fault condition, and then, a reading procedure is performed on the (v) cell to verify the fault effect in the affected cell. This TP can be employed in general-purpose memories with normal writing and reading procedures. However, the special purpose memory, in the SC, includes a set of constraints presented in the previous section; thus, in order to include the operational restrictions and initialization conditions into the AFFP and TP, additional procedures are added as part of the pattern for test purposes. The adapted versions of the AFFP and TP for the SC memory are presented in equations 4.5 and 4.6.

The additional memory procedures in 4.5 and 4.6 were selected as follows: the first four terms in parenthesis, in AFFP(SCH) and TP(SCH), describe the initial logic states for both cells. The second and fourth terms are the unavoidable reading operations on each cell of the SC memory by the operational restrictions. The fifth term represents the sensitive operation of the FP. The sixth term is the implicit reading procedure due to the previous operation. Finally, the seventh parameter, in TP(SCH) is the additional operation to observe the failure in the memory cell.

$$AFFP(SCH) :< X^a Z^v, (W_Z{}^a, r_X{}^a, W_Z{}^v, r_Z{}^v, W_{\bar{X}}{}^a, r_{\bar{X}}{}^a), \bar{Z}^v, Z^a > \tag{4.5}$$

$$TP(SCH) :< X^a Z^v, (W_Z{}^a, r_X{}^a, W_Z{}^v, r_Z{}^v, W_{\bar{X}}{}^a, r_{\bar{X}}{}^a) > \tag{4.6}$$

It is worth noting that during the scheduler operation it is possible to perform reading operations in the memory for the TAM and WPC fields. Considering the TAM fields the only condition is that each bit field maintains the same logic value. On the other hand, the WPC changes by executing a new instruction, thus a reading operation can be partially performed. These toggling bit fields cannot be included in the previous pattern. This means that the previous pattern should be applied more than once to fully test the WPC field. As explained below, the TP(SCH) can be directly derived from the associated AFFP(SCH), since the same writing and reading operations are performed. The complete list of AFFP(SCH)s is presented in Tables 4.4 and 4.5.

4.1.8 Methods to Generate SBST Patterns for the Scheduler Memory

the scheduler memory is a special purpose memory and includes some restrictions in terms of writing and reading operations. Moreover, the accessing method to each LEs cannot follow the conventional procedures used in processors to access data memories. Thus, alternative methods should be proposed to inject test patterns based on writing and reading operations.

The proposed methods are focused for detecting permanent faults and static coupling faults in the WPC and the TAM fields of the memory in the scheduler.

This subsection presents the software methods employed to perform reading and writing operations in the SC memory for the TAM and WPC fields.

The memory behavior when performing reading and writing operations

The in-field controller operation imposes restrictions when perform writing ($W_{(x)}$) and reading ($R_{(x)}$) sequences in the memory. These restrictions depend on the target field in the LE.

The methods to perform $W_{(x)}$ procedures into the WPC and the TAM fields are similar among them and are based on executing control-flow instructions.

Conditional control-flow instructions can be used to write in the TAM field. However, their execution may generate thread divergence creating two paths (the *Taken* and the *not-Taken*), which are managed by the warp scheduler and are executed independently in a serial fashion until a convergence point is reached. Then, the threads are executed again in parallel (see Fig. 4.5). Thus, the execution of one conditional control-flow instruction generates a process where all the instructions in the first thread path (*Taken*) are executed, generating a $W_{(x)}$, followed by the execution of all instructions in the second path (*Not-Taken*), forcing an inverse writing $W_{(\bar{x})}$ operation on the same bit field. New architectures include deeper granularity and are able to process independently divergence threads by storing more parameters into the memory [44], thus partially reducing the latency of divergence thread paths.

The time execution of each writing procedure on each path directly depends on the total number of instructions in the path. For the TAM field, the reading sequence is implicitly added after the execution of each instruction. Fig. 4.6 represents the effect of a conditional control-flow instruction which modifies the state of one-bit of the TAM field in terms of $W_{(x)}$ and $R_{(x)}$ operations. The divergence instruction is able to generate writing operations of both logic values (1 and 0) on the same bit field consecutively.

The access to the WPC field in a LE is based on the sequential execution of instructions. Nevertheless, this execution is a naïve method and some high-part fields are complex to stimulate. In a GPU, each warp employs the same

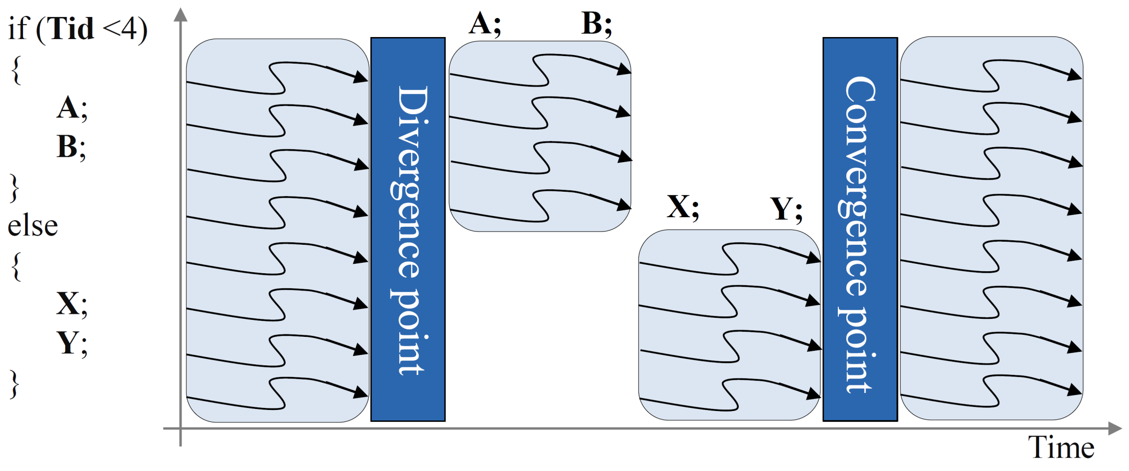


Figure 4.5: A general scheme of the classical intra-warp divergence management for the NIVIDIA's Pascal and previous GPU architectures.

program counter to fetch and execute an instruction. Unconditional control-flow instructions to specific locations can be used to generate the test patterns in this field. Taking into account that each warp executed by the SM resources employs a shared program counter, it is required to add mechanisms to stimulate these fields. Nevertheless, the same behavior of the TAM field is also presented in this field and $R_{(x)}$ operations are generated after a $W_{(x)}$ sequence.

In order to face the TAM fields, we consider the divergence management mechanism implemented in NVIDIA Pascal and previous architectures.

4.1.9 Test Pattern Generation

In order to generate TPs for the target memory, each restriction is analyzed and considered in the mapping process of FPs into software functions.

Regarding operational restrictions, the first and the second ones cannot be avoided and test patterns targeting the TAM fields must face the starting condition of all threads active (all bits equal to 1). Some FPs require initialization conditions of bits in logic 0, thus additional patterns must be generated and applied before starting the test sequences for the target FPs.

The second restriction describes the impossibility of performing a $W_{(x)}$ without one consecutive implicit $R_{(x)}$ operation at the end of each instruction cycle. The FPs definition, design, and implementation TPs must include those consecutive $W_{(x)}$ and $R_{(x)}$ procedures.

Regarding the third listed constraint, it describes the impossibility to stop a divergence path when started. A technique to control and reduce the effect is based on the selection of a limited number of operations and control-flow functions presented on each path. A set of input patterns (see Table 4.6) may be employed to

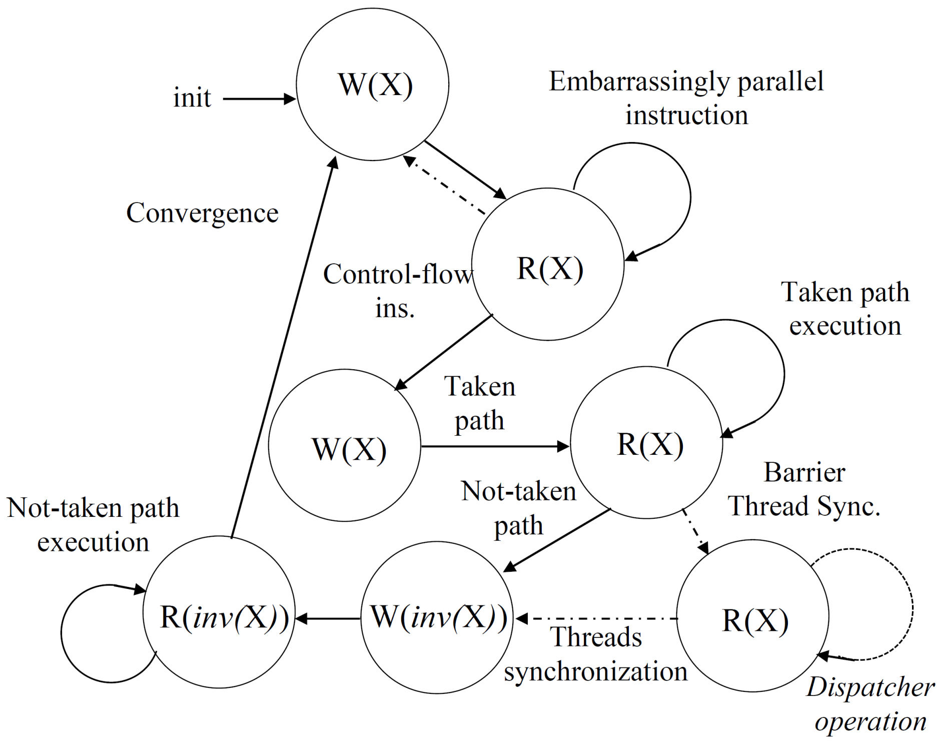


Figure 4.6: A general scheme of the write and read operations generated by a divergence path function in a bit of the TAM field. The nesting divergence case is not considered in the scheme.

divide the self-test program into chunks. These test patterns are carefully selected to increase the fault detection inside and among LEs.

Table 4.6: Selected test patterns for coupling fault detection in a LE.

Test Pattern	**Description**
1111...0000 / 0000....1111	First-Half X, Second-Half $\bar{X}$
00001111.. / 11110000...	First four bits X, Second four $\bar{X}$
0011...0011 / 1100...1100..	First two bits X, Second four $\bar{X}$
1010...1010... / 0101...0101...	Alternated X and $\bar{X}$
1111111111111111111111...	All in ones

The third restriction arises during the execution of a not-taken path. Initially, the scheduler submits threads executing the divergence taken-path. Then, the scheduler inverts the state of the threads in a warp activating those that were inactive and inactivating those that executed the taken path. In this way, an inverse $W_{(x)}$ operation, in the TAM field, starts the not-taken path execution. The not-taken path can be temporary skipped or delayed by the addition of nested divergence paths. These nesting conditions introduce additional operations in a procedure. Nevertheless, these do not affect the test pattern generation or the adaptation of a March operation into high-level functions.

Concerning the fourth constraint, the previous test patterns must remain at least one thread (bit) in the active state (logic 1). This condition explains the missing pattern (all in 0s) in Table 4.6.

The fifth restriction (warp control issues by dispatcher units) is faced by using combinations of thread synchronization functions, i.e. thread barrier instructions, semaphore mechanisms (local variables), and control-flow loops. These elements are placed in strategic locations in a function in order to stop/skip the operation of the dispatchers and to control the warp submission into the SM resources. The loops are placed in order to retain the state of LEs during a stopping condition.

This technique is able to hang temporarily the operation of an active warp and to dispatch a desired one. It is worth noting that, some undesired consecutive $R_{(x)}$ operations may be executed as a product of the skipping method for the dispatchers. Thus, adding latency in the program execution. On the other hand, these $R_{(x)}$ operations in a LE do not produce consequences in the test pattern generation and the March operation or algorithm adaptation, as previously described.

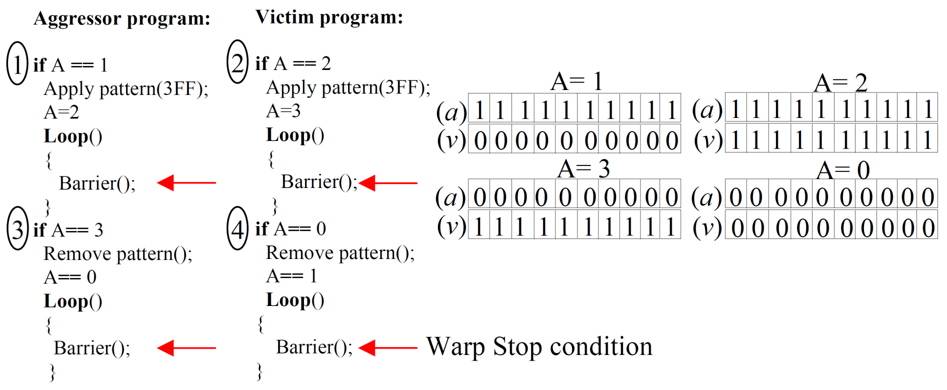
During the scheduler operation, this unit manages the content of the WPC and the TAM parameters employing $W_{(x)}$ operations, which are the product of control-flow instructions (conditional and unconditional). Considering that in most GPU technologies, the divergence paths are executed in different operation cycles in a serial manner, the proposed technique uses the taken-path as the principal path for test pattern generation. Thus, the inverse $W_{(x)}$ operations can be neglected and it is not used in most of the cases.

The next subsection proposes and details a method to skip the dispatcher operation employing the mechanisms introduced below. The consecutive sections present the sequence of steps to produce $W_{(x)}$ and $R_{(x)}$ procedures into the WPC and the TAM parameters of a LE.

A method to skip the operation of the dispatcher unit in the SM

The basic operation of the dispatcher units is the management and warp submission to the SM resources. The operation of this unit depends on multiple parameters in order to increase the performance. Those parameters include the application coding style, data operands sources, internal hazards, and availability. Moreover, it is also considered the instruction conflict and latency, and internal warp distribution policies; thus in most of the cases warps are not dispatched following a sequential approach with a direct incidence in the LEs. This operation restricts the application of test patterns on (a) and (v) cells, thus its operation should be controlled or avoided.

The proposed method employs software approaches to start and stop the warp



Aggressor program:

```
(1) if A == 1
      Apply pattern(3FF);
      A=2
      Loop()
      {
        Barrier();
      }
(3) if A== 3
      Remove pattern();
      A== 0
      Loop()
      {
        Barrier();
      }
```

Victim program:

```
(2) if A == 2
      Apply pattern(3FF);
      A=3
      Loop()
      {
        Barrier();
      }
(4) if A== 0
      Remove pattern();
      A== 1
      Loop()
      {
        Barrier();
      }
```

Figure 4.7: Example of the method used to skip the operations of the dispatcher unit employing semaphores and synchronization functions between consecutive aggressor and victim cells.

execution and keep the belonging LE state. It is worth noting that the main idea is to stop temporarily the execution of an active warp instead of fully terminate the warp operation and inject patterns in a sequential fashion in order to keep the coherency of the TP injection. Moreover, a set of variables which behave as semaphores and thread synchronization checkers are employed for this purpose. It should be clarified that the addition of these mechanisms requires communication among threads. Moreover, the same kernel operations are executed by the (a) and (v) cells.

Fig.4.7 represents a basic example of the mechanism employed to skip the operation of the dispatcher. In this example, a kernel program is executed concurrently by warps corresponding to consecutive LEs behaving as (a) and (v) cells. One external comparison parameters (A) is loaded into the kernel. Moreover, (A) is shared between both warps. As shown in the example, the injection pattern and the starting cell can be selected by changing the (A) value in the semaphore.

The process starts with a warp selection through (A) value comparison. In this example, (A) starts in 1 and the warp corresponding to the (a) cell is selected to start its execution. Then, a pattern is applied (3FFh), the semaphore variable is updated with (A=2) and the barrier instruction stops the warp.

The dispatcher picks any other available warp and dispatches it to the system. However, if the submitted warp is not the expected one, a loop mechanism stops it again. It should be clarified that a previously stopped warp continues its execution after the last executed instruction before the stopping condition. Employing this mechanism, the target (v) warp, with an A=2 condition, can start the execution in an ordered manner.

Once (v) is dispatched, the semaphore condition is true and the pattern is injected on the cell followed by a new update in the semaphore value and a barrier function. This last instruction restarts the dispatcher operation and picks again any available warp.

In final steps, the dispatcher submits again the warp corresponding to (a) and enters in a path to remove the pattern. This pattern can be the Not-Taken path or an additional nesting path for a new pattern injection. Moreover, the semaphore is again updated. Finally, the dispatcher submits (v) and removes the pattern. In the end, the cells are ready to start a new pattern injection. The loop requires an additional local variable to calculate the number of loop execution times. It is worth noting that, we did not consider the Not-taken path to include operations related to the injection of new patterns.

Test patterns for the TAM field

The scheme in Fig. 4.6 describes the sequence of steps and stimulus instructions to generate patterns in one LE. The following steps are derived from this scheme and perform $W_{(x)}$ procedures in the target parameter (TAM).

This method presents effectiveness in evaluating some coupling faults between two cells belonging to different LEs (CFir, CFdrd, and CFrd).

Sequence of steps for pattern generation in a single cell:

1. Execute an embarrassingly parallel function (F) (Initial condition).
2. Execute a divergence generator function ($W_{(x)}$ in the target bit(s) of TAM).
3. Execute the taken path ($R_{(x)}$ in the full TAM).
4. Execute the not-taken path (Inverse $W_{(x)}$ and $R_{(x)}$ procedures on selected bit(s) field).
5. Convergence point execution (CP) (parallel execution of instructions and implicit $R_{(x)}$ procedures).

The previous steps form the basic $W_{(x)}$ procedure into a LE. On the other hand, the proposed method may neglect the Not-Taken path for test patterns generation. Thus, step 4 can be ignored, for the purpose of test pattern generation, and is mainly employed as a connection to start a new March operation.

In the Taken Path, of a thread group, additional functions are added to stop and hang the warp execution. Then, the warp scheduler selects an available warp and the dispatcher launches it in the SM resources. The CFdrd, the CFrd, and the CFir coupling faults can be suitably tested by means of both divergence paths, due to the number of $W_{(x)}$ and $R_{(x)}$ operations involved. Nevertheless, some additional steps must be added to test a bigger number of coupling fault sets.

Sequence of steps for pattern generation in multiple cell:

The detection of coupling faults, among cells of different LEs, requires more steps including synchronization operations, nesting divergence functions, and warp selection routines to assure the correct pattern generation and evaluation of each fault set. The following steps are based on the scheme presented in Fig. 4.6, and the interaction between two cells. Experimental observations were used to determine the interaction among cells. The following list describes the suggested steps to test this type of coupling faults:

1. Execute an embarrassingly parallel function (F).
2. Select a target warp or LE ((a) cell).
3. Execute the first divergence function ($W_{(x)}$ operation in the target bit(s) field of (a) cell).
4. Execute the taken path of divergence ($R_{(x)}$ the full TAM field for (a) cell).
5. Execute a barrier function in the (a) cell path, thus stopping the warp execution and launching a new warp.
6. Select a new target warp ((v) cell).
7. Execute a second divergence function ($W_{(x)}$ in the target bit(s) of a (v) cell).
8. Execute the taken path for the second divergence ($R_{(x)}$ operations of the full TAM field in the (v) cell).
9. Execute a barrier function in the (v) cell path launching a new warp.
10. Execute of the not-taken path for the (a) cell (inverted $W_{(x)}$ operation in the target bit(s) of the TAM field, followed by $R_{(x)}$ operations).
11. Execute a barrier function in the (a) cell path.
12. Execute of the not-taken path for the (v) cell (inverted W(x) procedures in the target bit(s) of the TAM field, followed by R(x) operations).
13. Execute a barrier function in the (v) cell path.
14. Convergence point execution (CP).

The step range, from 9 up to 13, originally is part of the neglected operations required to start a new test patterns sequence. Nevertheless, these steps potentially can be employed to test additional coupling fault conditions in the TAM parameter (CFtr, CFds).

In the first divergence, an external pattern is applied to the target (a) cell. This pattern divides the warp by selecting the active threads during the evaluation of a (v) cell. This division is performed by a divergence which splits the warp into two equal groups of consecutive threads (group 1: 0 to 15 and group 2: 16 to 31). Nevertheless, the functions must be evaluated independently, in both groups, to cover all coupling faults.

In this method, the $R_{(x)}$ operations are integrated with the $W_{(x)}$ ones. In the simplest case, $R_{(x)}$ operations can be performed by the execution of non-control-flow instructions during the execution of a warp. In the TAM field, one $R_{(x)}$ operation is presented when the number of active threads, in a warp, is the same after executing one instruction. Similarly, for the WPC field, the execution of instructions is able to maintain the value in most fields, but for the WPC parameter, some bits should be neglected during the fault evaluation.

As introduced previously, $R_{(x)}$ operations are presented after each instruction cycle in both targeted fields. Moreover, these cannot be avoided. The scheduler checks the LE continuously at the starting and ending points of each instruction cycle to preserve the SM coherency during the warp operation. Other halted and stopped LEs are read when those turn into the active state by the SC management.

Step 8 describes the addition of a nested divergence function. A nested divergence function is effective to retain a warp in an active state and maintain at least one active thread in the TAM field, thus limiting the effect of the fourth constraint in the memory. Moreover, this is suitable to detect coupling faults of the groups (CFir and CFrd). The divergence function can be located after steps 4 or 10 considering an (a) cell. Similarly, this function can also be placed after step 8 for a (v) cell.

A third nested divergence function is required to detect the faults in the CFwd set. This sensitizes some missing conditions and guarantees the required $W_{(x)}$ operations in the memory. The third nested divergence function operates selecting only the thread 0 (first) or the thread 31 (last) for each path. This divergence should be applied in both cases to also test coupling faults on those fields. Some barrier instructions are added on each path for synchronization purposes. In the sequence of steps, the 14th step is replaced and the 4th, 5th, 10th and 11th are added to generate a new operation on the (a) cell. Thus, the third nested divergence function is executed during the second nested divergence, in the not-taken path.

This sequence can be used to evaluate any interaction among the (a) and (v) cells. Finally, barrier functions and shared variables may be placed on the steps to control the displacements across the LEs in any memory direction (*dropping* or *incrementing*).

Test patterns in WPC

The proposed technique considers those GPU architectures with a shared WPC parameter in all threads of a warp. In this strategy, the test program design and execution must target the highest possible parallelism, thus the approach avoids thread divergence to maintain the parallel execution of the threads in a warp. For this purpose, a set of routines are located in specific and strategic addresses of the system memory. The target memory addresses, for each routine, are based on the test patterns introduced in Table 4.6.

Each routine is mainly composed of embarrassingly parallel operations and thread barriers. The main function calls each routine using unconditional control-flow instructions. Inside the routines, a set of barrier functions halts the operation of a warp and starts the execution on a new one. The same method used to select the warps, in the TAM parameter evaluation, is also used for testing the WPC parameter.

Some WPC bit fields, in the highest part of the memory, are difficult to evaluate by the complexity to locate the test routines. This issue is solved by the inclusion of additional GPU functions and program kernels in the memory. These functions and programs are placed in memory during the compilation. Finally, the method is flexible and the implementation may consider division into pieces in order to evaluate specific WPC fields through independent kernels in short execution times.

Sequence of steps for pattern generation in single cells

The following sequence of steps describes the procedures of writing and reading into the WPC parameter to evaluate a single cell.

1. Execution of an embarrassingly parallel instruction (F) ($R_{(x)}$ procedures).
2. Execution of an unconditional control-flow routine (calling a function and $W_{(x)}$ and $R_{(x)}$ procedures).
3. Return from the routine, and then compare the signature. Start a new call to another routine ($W_{(x)}$ and $R_{(x)}$ procedures).
4. (Restart and repeat steps 2 and 3 when required).

As previously discussed, $R_{(x)}$ operations are integrated into the W(x) operations in the target memory. For each memory parameter (TAM or WPC), the processing thread computes a signature (d_signature). This parameter is included as an observation mechanism to evaluate and detect any fault in the LE. A mismatch in the final signature represents the presence of a fault in the memory cell. This signature is evaluated at the end or in the middle of each test program routine.

Sequence of steps for pattern generation in multiple cells

As previously introduced for the TAM parameter, the test pattern generation of coupling faults among LEs requires more elaborated steps. The (a) and (v) target cells are carefully chosen by the warp selector mechanism which may generate divergence, in case of intra-warps cells, or not when the cells belong to different LEs. The following sequence of steps is able to generate test conditions and patterns.

1. Execution of an embarrassingly parallel instruction (F).
2. Execution of unconditional control-flow operations calling and selecting an (a) cell ($W_{(x)}$ operation in the (a) cell).
3. Execution of the routine and a barrier instruction in the path of the (a) cell launching a warp containing the (v) cell ($R_{(x)}$ operation in the target (a) cell of a LE).
4. Selection of the target (v) cell and execution of unconditional control-flow operations ($W_{(x)}$ operation in the (v) cell).
5. Execution of the routine and a barrier instruction in the path of the (v) cell, dispatch of a new warp containing (a) cell.
6. Return to the main program path and the start of the new operation of embarrassingly parallel instructions ($W_{(x)}$ operation in the returning stage, $R_{(x)}$ in the parallel execution).

This sequence of steps allows the test pattern generation and evaluation of the groups, CFtr and CFds, of coupling faults. Nevertheless, other steps must be added to evaluate missing coupling fault groups, such as CFir. In this case, the new steps are located after the 5th step and these include the execution of additional functions in the routine, thus generating implicit $R_{(x)}$ operations in the LE. New barrier instructions are also placed in order to generate elaborated stimulus in both cells. The generation of test patterns for CFwd coupling faults needs a new $W_{(x)}$ operation in the (a) cell by means of a new routine, which is included after the 6th step.

March algorithms adaptation

March algorithms are well-known methods to detect coupling faults in memories. Thus, those methods can be employed to generalize the proposed technique to detect permanent and static coupling faults in the memory.

Those algorithms are composed on a set of writing and reading operations (or March operations) to be performed in the memory. The injection direction in the operation is considered as an additional parameter to consider in the implementation phase.

It is not possible to inject consecutive March operations in the scheduler memory by its operational restrictions. Nevertheless, this restriction can be partially avoided by applying March operation patterns in a segmented fashion. In this method, a March operation is firstly performed followed by a set of initialization operations. Then, the second March operation is injected and, finally, new initialization operations are included for other March operations. Fig. 4.8 shows an adaptation example employing the segmented approach.

Example of March operations	Adapted approach of March operations
$\updownarrow (W_{(0)})$	$\Downarrow (W_{(0)}, R_{(0)})^*$, $\Uparrow (W_{(1)})^*$, $\Uparrow (R_{(1)})^*$, $\updownarrow (W_{(0)})$
$\Uparrow (R_{(0)}, W_{(1)})$	$\Downarrow (W_{(0)}, R_{(0)})^*$, $\Uparrow (R_{(0)})^*$, $\Uparrow (R_{(0)}, W_{(1)})$
$\Downarrow (W_{(0)})$	$\Downarrow (W_{(1)}, R_{(1)})^*$, $\Uparrow (R_{(1)})^*$, $\Downarrow (W_{(0)})$
$\Uparrow (R_{(0)})$	$\Downarrow (W_{(0)}, R_{(0)})^*$, $\Uparrow (R_{(0)})$
$\Downarrow (W_{(1)})$	$\Downarrow (W_{(1)}, R_{(1)})^*$, $\Uparrow (R_{(1)})^*$, $\Downarrow (W_{(1)})$
$\Uparrow (R_{(1)})$	$\Downarrow (W_{(1)}, R_{(1)})^*$, $\Uparrow (R_{(1)})$

Figure 4.8: A normal sequence of March operations and the adapted approach for the scheduler memory, (*) Example of initialization operations.

The application of segmented patterns into the target memory locations does not reduce the effectiveness of the proposed method and the adaptation of a March operation.

This initiation sequence is composed of additional $W_{(x)}$ and $R_{(x)}$ operations required to access the target memory locations or to avoid the normal scheduler execution. In some cases, these initialization operations include March operations in other LEs. Thus, March operations are described independently as a set of program kernels.

The division of a test kernel in chunks, or multiple test programs, allows the fault evaluation during idle intervals of the in-field operation of the device. The division must also consider the observability mechanism and the Host interaction. For this purpose, the test signatures are stores in free locations of the global memory. Thus, these can be reused in multiple test kernels. Similarly, the Host may locate and trigger the test programs aside from the application kernels.

As a proof of concept, we employed two March algorithms (MATS+ and MATS++) to demonstrate the features of the proposed technique and generated a set of test programs for the TAM and WPC fields for each LE in the memory.

MATS+ AND MATS++ algorithms

Each algorithm is adapted following the proposed technique. Moreover, it is described in multiple test kernels; thus, its operation is performed by applying multiple test parts. Table 4.7 shows the operations in the MATS+ algorithm (the reader may refer to [163] for details regarding March test notation). It can be noted that initialization steps and implicit $R_{(x)}$ operations are included in the adaptation of each operation. The initialization steps are needed to generate a specific test pattern in the target LE (or warp) and to avoid the execution of the dispatcher units. This method is used for each LE in the memory.

The adaptation to high-level functions is based on independent kernels (Basic Block Kernel or BBK) describing a March operation. These BBKs uses the signature location in memory and an external test pattern as input parameters. The

Table 4.7: MATS+ and (*) MATS++ operations as a sequence of CUDA kernel executions.

Original March operations (MATS+)	**Adapted March Operations**	**Equivalent CUDA BBK kernel**
M1: $\Updownarrow$ (W(0))	Init. Steps:(W(X), R(X)); M1:$\Updownarrow$ (W(0), R(0))	Test_kernel_dropping <TOTAL_BLOCKS, TOTAL_THREADS> (TOTAL_THREADS, vector_params[0], d_signature);
M2: $\Uparrow$ (R(0), W(1))	Init. Steps: (W(X), R(X)); M2:$\Uparrow$ (W(0), R(0), W(1), R(1))	Test_kernel_dropping_x <TOTAL_BLOCKS, TOTAL_THREADS> (TOTAL_THREADS, vector_params[1], d_signature);
M3: $\Downarrow$ (R(1), W(0), R(0)*)	Init. Steps: (W(X), R(X)); M3:$\Downarrow$ (R(1), W(0), R(0))	Test_kernel_incrementing <TOTAL_BLOCKS, TOTAL_THREADS> (TOTAL_THREADS, vector_params[0], d_signature);

external pattern is only present for the evaluation of the TAM field in the LE.

Table 4.7 also presents the adaptation of the algorithm into a set of independent test kernels. In this example, it is can be observed that the external pattern is different in each case. The same pattern is applied to the first and third kernels. On the other hand, the second program needs the opposed value to generate the desired pattern. The kernel sequence (dropping, dropping_x, incrementing) must use the patterns in Table 4.5 to evaluate all target coupling faults. In the end, this program kernel sequence is applied eight times.

Algorithm implementation in the CUDA environment

In Fig.4.9, the scheme presents the interaction between (a) and (v) using the method described above. Although the diagram depicts a parallel program execution in (a) and (v), the execution of the algorithm is fully serialized for each SM. In fact, on each SM core, only one SC and its internal memory exist. However, this scheme shows the complexity in the test program description in order to ensure that (a) and (v) cells interact in the expected fashion.

The method employed to inject patterns in the cells uses a selection mechanism to identify and classify the LEs (warps) as even or odd (as (a)-(v), or (v)-(a)). Moreover, additional conditions in the kernel description were added to check the edge conditions of (a) in the memory borders (i.e., LE number 0 or 31 and target (v) number -1 or 32, respectively). In these cases, the (a) cell is maintained as inactive while the other cells perform the injection sequence. The mechanism employed for thread communication is based on the method to skip the dispatcher units operation (see section 4.1.9).

The implementation of the program kernels also considered the direction of pattern injection. Independent kernels were designed, as presented in Table 4.7, to perform the pattern injection in both directions.

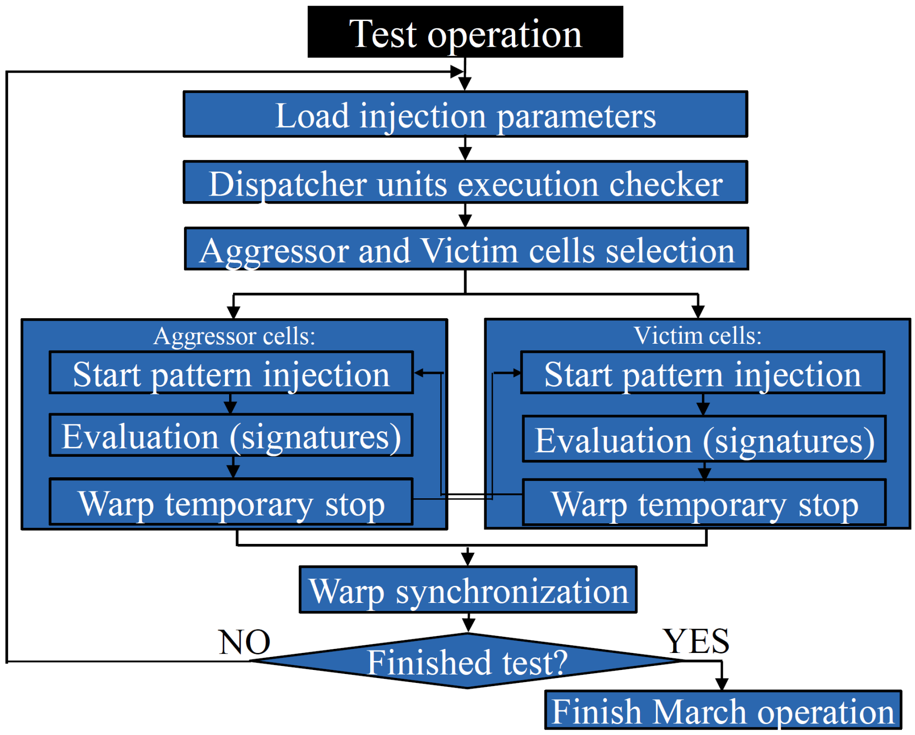


Figure 4.9: A general flow diagram of the test algorithm for the TAM and WPC fields in the scheduler memory.

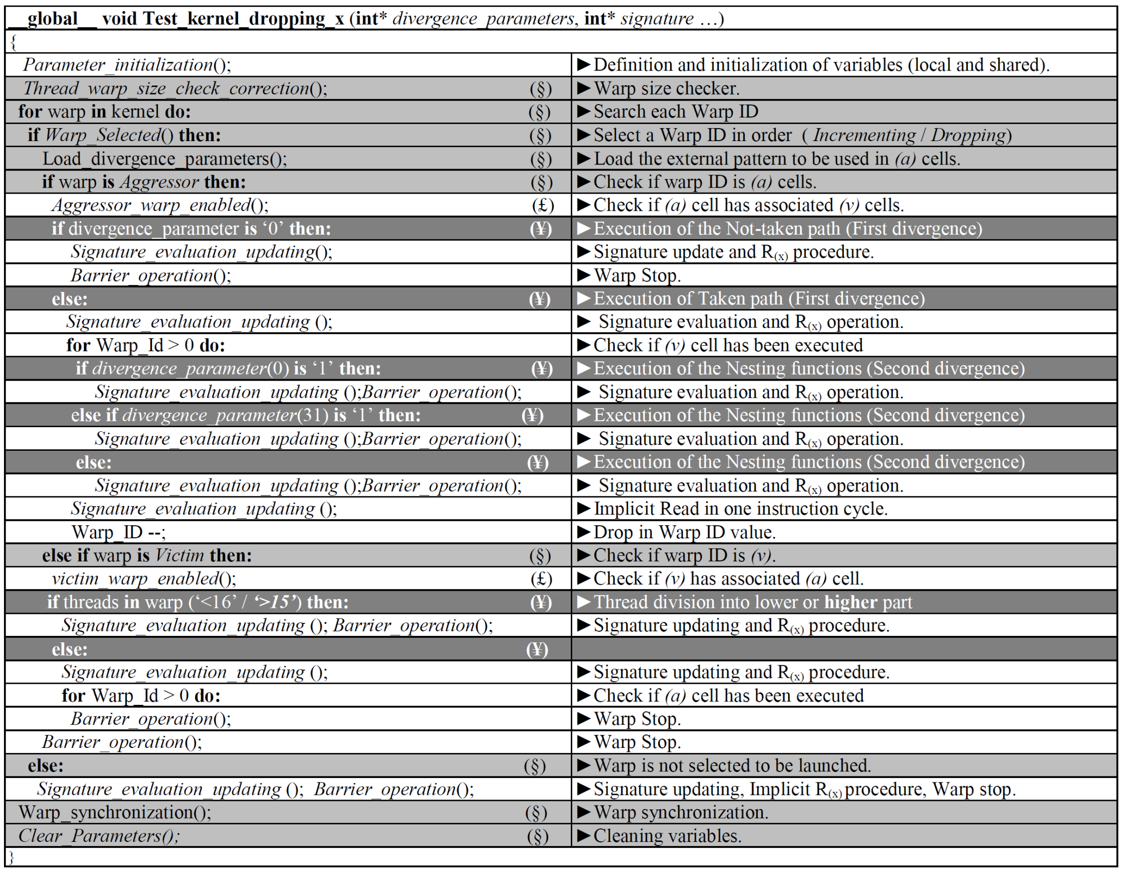
The pseudocode reported in Fig. 4.10 describes a general CUDA implementation of one of the test programs aimed to detect coupling faults. This kernel is executed concurrently by an (a) and a (v) warps (or LEs).

The warp selection mechanism divides both cells and also controls the execution of the program in a sequential fashion.

In Fig. 4.10 we detailed the implementation of the warp selection and detection process. The main difference among them is the total number of conditions required to generate the start of detection of the target LE.

It should be noted that, in both cases, the returning steps after and before warp synchronization are neglected to inject any test pattern into the cells. In the TAM case, these intervals are generated during the Not-Taken path evaluation. In the WPC algorithm, these intervals belong to the returning procedures from a called routine.

The warp selector mechanism generates independent paths for the (a) and (v) cells. Internally, the path may use divergence operations stimulating the parameter under evaluation. In (a), a second divergence path is employed to add the required initialization condition ($W_{(0)}$) for evaluation of those coupling faults which need this initial state.



```
__global__ void Test_kernel_dropping_x (int* divergence_parameters, int* signature …)
{
  Parameter_initialization();                                        ► Definition and initialization of variables (local and shared).
  Thread_warp_size_check_correction();                          (§)  ► Warp size checker.
 for warp in kernel do:                                         (§)  ► Search each Warp ID
  if Warp_Selected() then:                                      (§)  ► Select a Warp ID in order ( Incrementing / Dropping)
    Load_divergence_parameters();                               (§)  ► Load the external pattern to be used in (a) cells.
    if warp is Aggressor then:                                  (§)  ► Check if warp ID is (a) cells.
      Aggressor_warp_enabled();                                 (£)  ► Check if (a) cell has associated (v) cells.
      if divergence_parameter is '0' then:                      (¥)  ► Execution of the Not-taken path (First divergence)
        Signature_evaluation_updating();                             ► Signature update and R(x) procedure.
        Barrier_operation();                                         ► Warp Stop.
      else:                                                     (¥)  ► Execution of Taken path (First divergence)
        Signature_evaluation_updating ();                            ►  Signature evaluation and R(x) operation.
        for Warp_Id > 0 do:                                          ► Check if (v) cell has been executed
          if divergence_parameter(0) is '1' then:               (¥)  ► Execution of the Nesting functions (Second divergence)
            Signature_evaluation_updating ();Barrier_operation();    ►  Signature evaluation and R(x) operation.
          else if divergence_parameter(31) is '1' then:         (¥)  ► Execution of the Nesting functions (Second divergence)
            Signature_evaluation_updating ();Barrier_operation();    ►  Signature evaluation and R(x) operation.
          else:                                                 (¥)  ► Execution of the Nesting functions (Second divergence)
            Signature_evaluation_updating ();Barrier_operation();    ►  Signature evaluation and R(x) operation.
          Signature_evaluation_updating ();                          ► Implicit Read in one instruction cycle.
          Warp_ID --;                                                ► Drop in Warp ID value.
    else if warp is Victim then:                                (§)  ► Check if warp ID is (v).
      victim_warp_enabled();                                    (£)  ► Check if (v) has associated (a) cell.
      if threads in warp ('<16' / '>15') then:                  (¥)  ► Thread division into lower or higher part
        Signature_evaluation_updating (); Barrier_operation();       ► Signature updating and R(x) procedure.
      else:                                                     (¥)
        Signature_evaluation_updating ();                            ► Signature updating and R(x) procedure.
        for Warp_Id > 0 do:                                          ► Check if (a) cell has been executed
          Barrier_operation();                                       ► Warp Stop.
    Barrier_operation();                                             ► Warp Stop.
  else:                                                         (§)  ► Warp is not selected to be launched.
    Signature_evaluation_updating (); Barrier_operation();           ► Signature updating, Implicit R(x) procedure, Warp stop.
 Warp_synchronization();                                        (§)  ► Warp synchronization.
 Clear_Parameters();                                            (§)  ► Cleaning variables.
}
```

Figure 4.10: A pseudocode (CUDA) describing the program kernel implementation to detect coupling faults. (¥) Functions to generate divergence paths in the (a) and (v) cells in the LEs. (§) functions to skip dispatchers. (£) optional functions to evaluate edge conditions in LEs.

The pseudocode presents the worst-case scenario for test pattern injection, thus the kernel description can be simplified in order to avoid the second divergence for some static coupling and permanent faults.

Adapting March operations to test the scheduler memory

Each March operation can be represented as a set of $W_{(x)}$ and $R_{(x)}$ operations in a selected direction. The implementation of a March operation, considering the FPs for the SC memory included the direction parameter as an additional factor in the adaptation.

Adapting the March writing and reading procedures

Each operation ($W_{(x)}$ and $R_{(x)}$) is translated into equivalent software functions on the target fields (TAM and WPC). Table 4.8 presents an example of the adaptation of a TP for coupling fault detection composed of multiple $W_{(x)}$ and $R_{(x)}$ operations.

Table 4.8: Example of a TP sequence for coupling fault evaluation.

Target TP operations TPx = $\{\mathbf{w}_x^a, r_x^a, w_y^v, r_y^v, {r_y}^v, {r_y}^v\}$		
March Operation	**The equivalent operation for TAM fields**	**The equivalent operation for WPC fields**
w_x^a	Divergence generation in (a)	Subroutine call in (a) cell
r_x^a	Execution of multiple instructions in the taken path of (a)	Execution of consecutive instructions in (a) cell
w_y^v	Divergence generation in (v)	Subroutine call in (v) cell
r_y^v	Execution of multiple instructions in the taken path of (v)	Execution of consecutive instructions in (v) cell
r_y^v	Execution of multiple instructions in the taken path of (v)	Execution of consecutive instructions in (v) cell

It is worth noting that this example does not present the additional steps after the pattern injection and the Not-taken path operations for (a) and (v) warps. Thus, those steps are considered as initialization operations for the next pattern.

The final step in order to adopt a March operation is the adaptation of the direction parameter in the kernel. The next subsection presents the method to add and select this parameter in the design.

Adapting the March direction operations

The application of a March operation has an associated injection direction ($\Updownarrow$, $\Uparrow$, $\Downarrow$). This parameter is related to the FP to be tested and TP to be injected.

The CUDA implementation of the TPs included, on each kernel, an external parameter describing the injection direction (*Incrementing* and *Dropping*) of each operation.

Moreover, this organization is static. Thus, the warps with lower or higher IDs will be the first selected and have priority access to the SM resources. It is worth noting that warp selection does not generate any kind of divergence in the SM. Nevertheless, the warp selection process and direction management are based on the previously described method to skip the dispatcher unit from section 4.1.9.

One selection loop is added into the test programs to manage the warp selection considering the direction of the March operation. This selection loop is initialized with a base, an offset, and a limit. The base and the limit define the direction to be applied. This starts with the same conditions for each warp in the test kernel; thus during the kernel execution, the loop selects the warps in the range of the base and the offset values. Then, the base and offset are updated until the limit is reached. This selection mechanism has a higher priority for the warps in the range of selection. However, this method has low priority in the border warps and increases the latency in their execution. Moreover, the dispatching process may add additional conditions in order to select even or odd warps and generate TPs for multiple coupling faults.

Inside the loop, a base and offset are employed to select the external parameters and to identify the warps that can be dispatched into the SM.

Multiple shared memory locations are employed to control and manage the coherency of pattern injection between consecutive (a) and (v) warps. This local synchronization mechanism is local for the interaction between consecutive cells belonging to different LEs in the memory.

4.1.10 Experimental Results

Performance results

Several experiments were performed to validate the proposed methods. We employed an NVIDIA© GeForce GTX 960M GPU with 32 threads per warp and 1.176 GHz of clock rate. The evaluation and validation of the test programs are performed by employing the NVIDIA© Nsight™ 5.6 tool and the NVIDIA© Visual profiler. These tools are employed to determine the resources overhead and performance parameters. Moreover, these also verify the correct operation of the implemented test kernels in the GPU.

Table 4.9 reports the performance results of the implemented test programs following the proposed technique for different LE sizes in the scheduler memory. The second column in the table also reports the required idle time intervals to perform an individual test sequence of the whole procedure.

The BBKs were originally designed and implemented to be executed in one SM of the GPU with a configuration of one block per grid.

Table 4.9: Performance parameters of the implemented test programs to evaluate different LE sizes. The symbol (+) identifies the cost for active kernel functions, only.

		LE parameter					
		TAM			**WPC**		
LE size		32	16	8	32	16	8
BBK execution (uS)		778.98	359.30	168.30	573.50	187.79	91.52
Total execution (mS)		18.69	8.62	4.04	13.86	4.51	2.19
Number of instructions	***BBK incrementing kernel (KB)***	276.25	78.28	8.43	186.48	50.36	14.46
	BBK dropping kernel (KB)	276.61	78.18	8.40	436.13	73.49	15.58
	per pattern (KB)	829.11	234.74	25.26	809.08	174.21	44.49
	Total (MB)	6.63	1.88	0.202	6.473	1.397	0.356
GPGPU memory overhead	***System memory (KB)***	1.84	1.84	1.84	2.42+	2.42+	2.42+
	Shared memory (B)	260.0	132.0	68.0	4.0	4.0	4.0

The implementation of the proposed method used an equal number of local registers and this number is independent on the LE size. The TAM kernels required 28 registers and the WPC kernels required 37.

The number of instructions to evaluate a parameter (3 test kernels) in the LEs of the memory is relatively high (809 and 809KB, for WPC and TAM parameters respectively). This behavior is explained by the selection of the programming environment (CUDA) to implement the functions and the additional mechanism employed to manage and avoid the dispatcher units. Moreover, the total number of instructions is also proportional to the number of LEs tested.

As shown in Table 4.9, some milliseconds are required to execute all program kernels. Moreover, the kernels present a low cost in terms of the system memory overhead to perform the test of one pattern for 32 LEs (2.4KB in WPC parameter and 1.84KB in the TAM field). Nevertheless, the test program for the WPC parameter is also composed of some inactive kernels in memory to place the routines in the target memory locations to be tested. This test procedure requires the entire system memory space. Hence, the evaluation of the WPC parameter during device operation should be limited to the Power-On/Power-Off intervals.

Regarding the share memory resources (Table 4.9), the TAM kernels require a low number of memory locations. For this kernel, this parameter is directly dependent on the total number of line entries considered and the number of shared thread variables employed as semaphores used to avoid the dispatcher units execution. On the other hand, the WPC kernels employ a constant number of shared memory locations, as this kernel does not require semaphore variables among the threads.

Six test kernel functions are needed to control the warp execution and to stop the operation of the dispatchers. For this purpose, the conditional functions (divergence functions) are employed during the evaluation of coupling faults between cells. These functions include additional instructions in the test programs implementation. In contrast, the instruction size of the BBK is relatively low ($\approx$ 280KB) and is executed in less than 780. In the TAM test programs, the number of shared memory locations has a linear dependency with the evaluated LEs. In contrast, the WPC test programs employ a constant number of shared variables independently of the SC memory size. This can be explained considering that the techniques for testing the WPC parameter are simpler than those employed to evaluate the TAM field including the warp selection mechanism to stop the operation of the dispatchers.

A detailed analysis of the TAM test programs with the NVIDIA Visual Profiler shows that the concurrency operation of these test kernels, under multiple LE size, is 0%. The concurrency operation is an indicator of the level of parallelism in the program. This zero percentage in concurrency is mainly generated in the program by the complexity to follow the required steps and generate the patterns to evaluate the TAM parameter in a LE and the TAM. Additional analyses with the Visual Profiler showed that the implementations of the test programs spend almost 60%

of the execution time in thread synchronization operations or in halting conditions. In the WPC test kernels, the halting and the synchronization functions affect in a lower manner the execution time, which is near to 50%. In both cases, these operations are needed in order to stop or avoid the execution of the dispatcher modules during the test operation.

Fault detection results

In order to evaluate the effectiveness of the proposed technique and to check the FC, we employed as a verification tool a memory fault simulator. This simulator reads an input file with the generated memory procedures (writing and reading) during the program kernel execution. A detailed description and additional information about the memory fault simulator can be found in [164].

To the best of our knowledge, comparative functional test techniques targeting the same memory structure in the scheduler of a GPU were not proposed in the past, thus limiting the possibilities of a direct comparison. Nevertheless, we selected three representative benchmarks (*Vector_add*, *MxM*, and *Edge*) to evaluate, compare and show the efficacy of the proposed functional test mechanism. Each benchmark was configured with a workload equivalent to the one adopted for the functional test.

In order to generate a compatible input file for the memory simulator, each test program and benchmark was instrumented with debugging functions able to trace the functions, the operations or the instructions executed at each phase. Table 4.10 presents the obtained results for the proposed functional test mechanisms and the selected benchmarks expressed in terms of FC.

According to the results, the proposed technique is able to test effectively the static single and coupling faults in the SC memory. Although the original test programs design was not targeted to consider some groups of faults, such as the state coupling faults (CFst), the state faults (SF), and inversion coupling faults (CFid), the results in the fault simulation show that these groups of faults were also evaluated by the test program kernels in an indirect manner. The additional nesting divergence functions, for both cells (a and v), directly caused the detection of faults in the CFid group. Moreover, the time delays during the test injection provoke the required initial conditions in those cells that are not directly evaluated, thus, indirectly evaluating the faults of the CFst and SF groups.

During the first attempts to determine FC results from the fault simulations using the proposed method, we obtained some moderate percentages (96%) for some specific patterns in the memory. After a detailed check of these conditions, we concluded that the initial stage of some LEs, targeting the TAM fields, remains in an active state, thus avoiding the evaluation of some FPs with a $W_{(0)}$ as starting logic state. These restrictions were finally removed and corrected through the addition of the second divergence path in order to consider the conditions of initialization.

A comparison of results among the implemented test programs and the selected

Table 4.10: Fault coverage of the MATS++ test kernels for a SC memory with 32 LEs. (*) FPs that were not initially considered in the proposed technique.

Fault primitive	**MATS+ Algorithm FC (%)**		**VectorAdd FC (%)**		**MxM FC (%)**		**Edge FC (%)**	
	TAM	**WPC**	**TAM**	**WPC**	**TAM**	**WPC**	**TAM**	**WPC**
TF_1	100.0	100.0	100.0	34.4	100.0	37.5	1.6	35.5
TF_0			0.0	84.4	0.0	87.5	100.0	89.5
RDF_1			100.0	84.4	0.0	38.5	100.0	40.5
RDF_0			0.0	34.4	100.0	89.1	1.6	90.5
DRDF_1			100.0	0.0	0.0	0.0	0.0	0.0
DRDF_0			0.0	0.0	100.0	0.0	0.0	0.0
WDF_1			0.0	34.4	0.0	37.5	100.0	35.5
WDF_0			0.0	84.4	0.0	87.5	1.6	85.5
CFds_X, CFir_X, CFtr_X	100.0	100.0	-	-	-	-	-	-
CFrd_X, CFwd_X, CFdrd_X	100.0	100.0	-	-	-	-	-	-
CFst_X	100.0 (*)	100.0 (*)	-	-	-	-	-	-
SF_X	100.0 (*)	100.0 (*)	-	-	-	-	-	-
CFid_X	100.0 (*)	100.0 (*)	-	-	-	-	-	-

benchmarks allow us to affirm that an application by itself is not suitable to generate a considerable number of functional test patterns in the SC memory. The previous behavior is clearly represented in the low percentage of FC of each single cell FP group. *Vector_Add* and *MxM* applications operated fully parallel and did not produce all the required logic state changes for the TAM field. The parallel behavior explains the zero percentage presented in the FC in some FPs groups. A similar situation is presented in the low percentage of the TAM field in the Edge application, this behavior is generated by some divergence paths produced during the evaluation of the convolution algorithm. Nevertheless, the generated divergence is not enough to increase the percentage of FC in most single cell groups of FPs.

Concerning FC results in the WPC field, these directly depend on the total number of instructions and the number of internal loops described in the program kernel. The longest benchmark (*Edge*), which also includes up to 25 loops, denotes a higher FC in comparison with the other applications with lower number of instructions and with none (*Vector_Add*) or a few (*MxM*) loops in their description. A direct comparison of FC results between the proposed functional test mechanism and the representative applications shows the need and relevance of the proposed functional test solution in terms of detecting faults in a large number of groups of

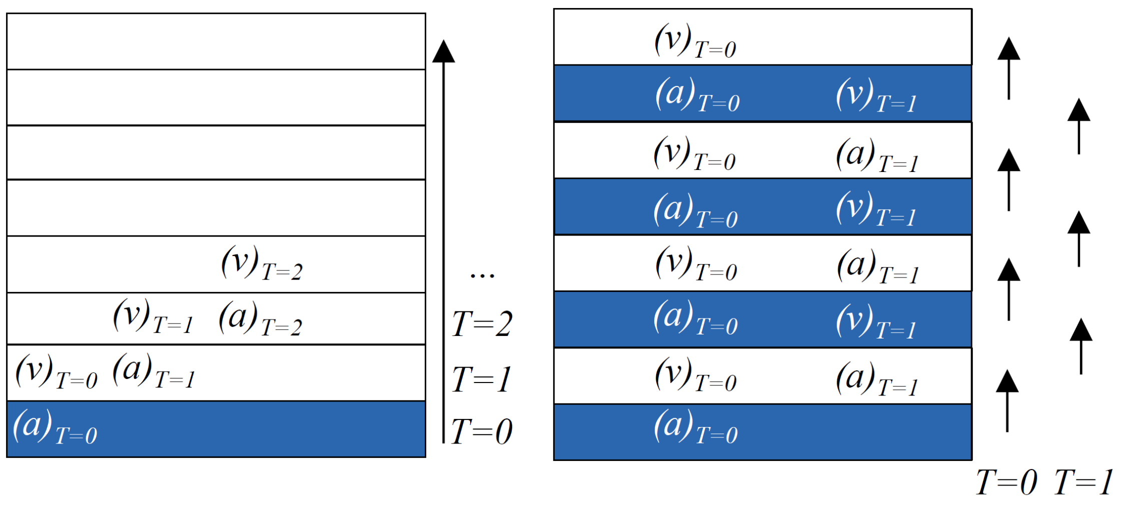


Figure 4.11: Proposed methods to apply a March operation including the direction parameter. (Left) Sequential method using software techniques to skip the dispatcher unit operation. (Right) Using a parallel arbitrary injection in pairs of LEs.

FPs.

The FC results in the benchmark for the coupling fault groups are not presented in Table 4.10. However, it is possible to state that the same trend towards a low FC for single cells FPs is also valid for the multiple cells groups

Kernel performance optimization

The implemented kernels were designed to apply the March operations in a sequential fashion following the direction of a March operation. However, the software management of the kernels (by using semaphore variables, loops, and synchronization functions) is effective in generating test patterns in the memory, but it also reduces the execution performance. Thus, the total kernel execution time can be measured in terms of hundreds of microseconds in the experiments. Nevertheless, as detailed previously, it was noted that a large part of the latency in execution time is caused by the operation of the synchronization loops.

In order to reduce the time execution in the kernels, we propose a set of optimizations in the kernel description by modifying the method employed to apply the March operation. This method applies the operation in an arbitrary manner on groups of two independent consecutive LEs (Warps), one as (a) and the other as (v), following the direction parameter of the original BBK. Thus, the injection of a W(x) or R(x) operation only depends on the correlated pairs of LEs (see Fig. 4.11).

This method takes advantage of the dispatcher units operation and uses them to manage the pattern injection on consecutive LEs. Employing this approach a high percentage of the latency presented in the previous sequential method can be removed. Nevertheless, it must be considered that synchronization loops cannot be fully removed. However, those can be optimized in order to reduce the final latency.

The initial loop employed for sequential warp selection was removed and replaced by the simple loop mechanism which also selects LE as (a) or (v). Then, on each path, this solution includes optional synchronization loops for each condition. One additional loop and break-loop condition are placed at the end of the kernel execution after the pattern is applied and the LEs are ready to finish.

The main purpose of the loop and the break-loop conditions is the reduction of inefficient synchronization loops and the associated latency for a large number of warps. In this strategy, the synchronization is only evaluated between the aggressor and victim warps, instead of adopting a global synchronization with all warps, as in the other strategy.

The break-loop condition was designed to be executed at the end, when a warp terminates its execution and intra-warp divergence is not active, thus simplifying the management of non-consecutive warps.

The same semaphore mechanisms are employed to keep the order in the execution. Nevertheless, the total amount of semaphores is incremented up to 32, which is equal to the number of LEs in the scheduler memory.

This optimized solution should be applied twice, for the (a)-(v) and (v)-(a) cells configurations respectively, in order to cover all conditions in the memory.

A performance comparison between a sequential version of a test kernel and the optimized version was performed. Table 4.11 presents the performance parameters for two BBKs (increasing and dropping).

Table 4.11: Performance parameter for the original and optimized version of a test kernel. (*) The resource overhead is calculated per kernel.

BBK	Time execution (us)	Registers	Shared memory (bytes)	Instruction size (bytes)
Dropping (Original)	208,701	37	4	1,152
Dropping (Optimized)	47,868	37(*)	256(*)	2,944
Improved	160,833	0	-252	-1,792
Increasing (Original)	176,750	37	4	1,728
Increasing (Optimized)	35,801	37(*)	256(*)	2,944
Improved	140,949	0	-252	-1,216

The effectiveness of the proposed optimizations in the kernel description can be noted by comparing the execution time of the BBKs (original and optimized).

For the Dropping BBK version, the execution time is reduced by more than 77%. Similarly, the Increasing BBK version presents a reduction of more than 79%. The total number of registers employed by each kernel remains the same. In contrast, each optimized kernel version employs up to 64 times more shared memory locations and almost the double of instructions in the kernel description. Although this optimization required more resources in term of shared and system memory locations, these can be partially neglected when comparing with the high percentage of execution performance gain.

In Table 4.11 , it should be considered that the selected BBKs do not include a second nesting divergence for pattern injection. This additional nesting would require one additional synchronization loop, thus each path would include one such loop, as it is presented in the original implementation. Nevertheless, the improvement process is the same as described above.

It is well-known for parallel programs that the usage of synchronization loops reduces the execution performance and could generate conflicts in the intra-warp execution. Thus, for the performance view point, these methods should be avoided. Nevertheless, the added mechanisms were carefully designed in order to operate in specific regions when the warp does not diverge. Moreover, the loop break condition is limited to convergence warp states: in this way the coherence of the parallel program is not affected nor compromised.

4.2 Multi-kernel approach of functional test on GPUs

As described in Chapter 2, GPUs are mainly composed of groups of SMs. Each SM includes multiple levels of pipeline stages to improve application performance. A large register is placed between each pair of adjacent stages to temporarily store information about the multiple instructions executed on each stage. These registers are crucial for the SM operation and stores temporary information about the data and control signals employed in the different stages of the SM, so it is expected that a fault affecting these structures could generate critical and unexpected behaviors, from an erroneous result (if the affected location is a data-path register field) until a system crash (when a control-path register field is affected). Hence, faults in these structures could often be unacceptable for safety-critical applications. In particular, faults in control-path fields of Pipeline Registers (PRs) are complex to detect during the device operation. Moreover, systematic solutions for in-field test of faults in these GPU structures are still missing.

Some recent works [90][102][153] proposed tools to inject soft-errors at the low-level code in real o micro-architectural models of GPUs, in particular, to support the analysis of transient faults effects. However, the injection locations are limited to some data-path units and it is not possible to determine a Fault Coverage

(FC) for permanent faults resorting to these solutions. In other works, the authors employ RT-level models and real GPUs to propose test techniques targeting permanent faults in some modules, such as register files (RF) [82], memories [165] and controllers [166]. Similarly, other works [167][108] analyzed the effect of transient faults on data-path and found a relation among the fault effects, the employed instructions and the module usage. In [168], the authors proposed hardening techniques to mitigate the effect of transient faults in the PRs calculating the effects of faults affecting each register for some selected applications. However, to the best of our knowledge, there are no works proposing methods addressing permanent fault detection in PRs of GPUs and reporting experimental data on their effectiveness.

In this section, a software-based technique is proposed to explore the feasibility of such technique when detecting permanent faults in the PRs of the SMs of a GPU. More in detail, this work targets the control-path fields in the pipeline registers considering that these structures are present in GPU technologies and include control and management information. This information is crucial for SM operation and instruction execution. Moreover, these are not easy testable and are located across PRs in the SM. The data-path fields are neglected since it is known that the test of data-path structures is more dependent on low-level implementation details and can be successfully achieved using techniques such as the one proposed in [169].

A multiple-kernel approach is used to target different sub-sections of the control-path fields in the PRs, showing that traditional in-field functional techniques developed for CPUs can be applied in this case, provided that a careful combination of high- and low-level programming structures are adopted. Experiments were performed employing fault simulation on FlexGripPlus.

The experimental results show the effectiveness and limitations of the approach. As far as we know, this is the first work presenting an experimental evaluation (i.e., assessing the achieved FC) of SBST approaches to detect permanent faults in the PRs of a GPU.

The section is organized as follows: subsection 4.2.1 briefly introduces the function of the PRs in the architecture of a GPU core. Subsection 4.2.2 introduces the proposed SBST techniques to detect permanent faults in the pipeline registers. Subsection 4.2.3 reports the experimental results to validate the proposed approach.

4.2.1 PRs in GPU architectures

The PRs in a GPU are mainly used to store operands and control signals for the execution of a warp instruction. The control signals are employed to manage information related to the warp instruction status. The Warp-Fetch *(W-F)* registers are composed of control fields related to the actual instruction, the warp status and execution state on the SM, including the Warp program counter (WPC), the initial and active thread mask (TAM or AThM), parameters for shared memory and general purpose registers size configuration. On the other hand, the PR between the Fetch-Decode *(F-D)* stage includes the same information of the previous stage, adding the warp instruction operational code. The Decode-Read *(D-R)* PR stores the format of the specific instructions fields to activate some operational modes or sub-modules in the next stage. The Read-Execution *(R-E)* PR additionally includes the Temporary Registers (TRs), which handle operands and predicate conditions for each SP in the execution stage. The Execution-Writeback *(E-Wr)* PR also contains some TRs storing the results. Table 4.12 summarizes the basic information about the control-path fields of PRs.

Table 4.12: General information about PRs in FlexGripPlus (Control-Path only).

Regs	Warp Instructions	Warp Status	Instruction Opcode	Instruction Formats	Bits per Reg
F-D	X	X			237
D-R	X	X	X	X	391
R-E	X	X		X	302
E-Wr	X	X		X	251
Wr-W	X	X			133
W-F	X	X			140

4.2.2 Proposed approach

The proposed technique is based on a bottom-up approach to target the test of the PRs in the GPU. This approach is focused on designing multiple parallel programs (kernels) to detect faults in different groups of sub-registers, so each designed kernel focuses on some specific parts of the register fields. In the end, the cumulative FC achieved by all the kernels is assessed. Each kernel is written through a high-level compiler (CUDA) when possible. Moreover, we added some assembly instructions (SASS) if required. As previously introduced in Chapter 2, the GPU assembly language (SASS) is not fully known as it has not been released by NVIDIA.

Proposed functional test methods

PRs are divided into two main groups and multiple sub-sets for the purpose of developing multiple functional test methods, one for each target. Fig. 4.12 represents the pipeline fields division. The proposed test methods are explained in the following subsections.

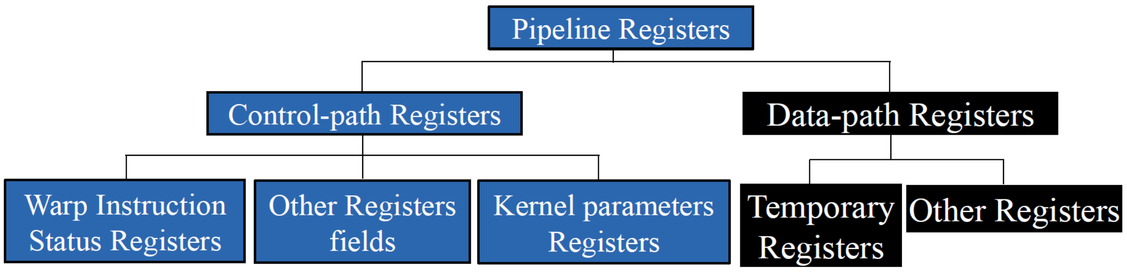


Figure 4.12: Composition of the PRs in the GPU core.

Algorithms to test the warp instruction status registers

As shown in Table 4.12, each PR stores warp instruction and status information across the SM. It means that, when targeting one sub-set of registers in one PR, it is enough to generate the detection on other PRs. The test method for the WPC field and AThM are:

WPC technique

This method (PC_T) employs a main program which calls a set of sub-routines, strategically placed in memory, in order to generate test patterns in the WPC registers of each PR. Each sub-routine is composed of a Signature-per-Thread (SpT) and a Counter-per-Thread (CpT). These two elements increase the observability of the target registers in the memory of the test program and also stop the execution if a permanent fault affects one of these fields.

The CpT verifies if a fault generates loop conditions and a hanging effect in the system. Kernel termination instructions (RET) are placed in memory locations between two subroutines to solve this issue. These instructions stop the kernel execution when a control-flow instruction does not reach a target memory location due to a fault turning hanging conditions into fault detections. Each subroutine checks the CpT value. If this value corresponds to the expected one, the SpT is loaded, updated and stored. Then, a new subroutine is launched. Otherwise, the kernel is stopped. Fig 4.13 shows the operations performed by a subroutine.

In the PC_T implementation, it was necessary to replace the CALL and RETURN instructions, not supported by FlexGripPlus, with unconditional and control-flow instructions (BRA). Thus, the test program consists of multiple unconditional jumps to and from subroutines. The lower bits in WPC registers were not explicitly tested with subroutines considering that instructions in master program implicitly generate patterns for them.

The program kernel is configured with 32 Threads per block and one Block in the grid. The execution of one warp checks the state of the WPC fields, considering that WPC is shared for all threads in a warp. The program kernel was designed to skip thread divergence and avoid the incidence of other modules during execution.

```
RET                                  (*)  ▶ Added before starting the routine
Init:  CpT_S ← Expected_param             ▶ Load expected CpT in subroutine
       load_CpT(i);                       ▶ Load CpT from global memory
       If CpT[i] == CpT_S then            ▶ Compare CpT and CpT_S
         CpT[i]  ← CpT[i]  + 1            ▶ Update CpT
         Store_CpT(i);                    ▶ Store CpT in Global memory
         load_SpT(i);                     ▶ Load SpT from global memory
         SpT[i] ← SpT[i] + S_Param        ▶ Update SpT
         Store_SpT(i);                    ▶ Store the SpT in Global memory
       else                               ▶
         RET                         (*)  ▶ Finish kernel Execution
```

Figure 4.13: Pseudo-code of the routine targeting the WPC fields. (*) Assembly instructions manually included. The subroutines where explicitly located in specific locations of the system memory.

AThM technique

This technique is an adaptation of the M3 algorithm, introduced in section 4.1.1, see [166]. In this case, the same warp status information (presented in the SC memory and PRs) is employed to propagate any fault effect and detect the error. Thus, is possible to adapt this method targeting AThM fields. The method generates thread divergence operations to supply test patterns targeting the AThM fields. Two control-flow instructions are employed to start and finish the divergence in the model. The first instruction, a conditional control–flow type, is activated through logical comparisons between the Thread Identifier and a set of constant values. These values are all the potential Thread IDs in a warp (0-31). The divergence generates two potential paths (*Taken* and *not-taken*). In the first one (*Taken*), an SpT is updated and stored in memory. The other path (*not-taken*) is not employed and an unconditional control-flow instruction returns to the convergence point for a new comparison. In the end, 32 comparisons are performed in a sequential fashion and each bit register is tested. The SpT is employed as observability mechanism of a fault in memory. Fig 4 shows the general procedure to test AThM fields.

We proposed two solutions by changing one logic operation in order to select between detection and diagnosis test. In the first case, a fast fault detection test is designed with logical comparison through an AND operator. On the other hand, XOR operators are used in the diagnosis test version.

The diagnosis test (WS_T_D) is able to identify an individual permanent fault in the AThM field. On the other hand, a detection test requires only two comparisons. For this purpose, two variations where proposed. In the first (WS_T_V1), the divergence is performed on even and odd thread groups (16 threads per time).

```
Load_ThreadId();                          ▶ Load the Thread.Id
Load_SpT($i$);                            ▶ Load signature
for $i \in$ {set of ThreadId in a warp} do  ▶ Evaluate for every Thread.Id
    j ← Params[k];                        ▶ Load the comparison parameter
    if $i$ == j then                      ▶ Comparison of Thread.Id and Constant
        Load_SpT($i$);                    ▶ Load SpT from global memory
        SpT [$i$] ← SpT [$i$] + 1         ▶ Set signature
        Store_SpT($i$);                   ▶ Store SpT to global memory
    else
        NOP                       (*)     ▶ Not used path
    k ←k+1                                ▶ Change constant value (Convergence Point)
```

Figure 4.14: Pseudo-code of the test program for the ATHM fields. (*) Assembly instructions included.

Initially, an even constant is loaded and the comparison generates divergence on even threads. Those threads actualize the SpT and return to convergence. Then, an odd constant is loaded and the previous process is repeated with the odd threads. Finally, a comparison of SpTs in memory is performed to detect a permanent fault. The (WS_T_V2) test version employs only one comparison to generate the divergence (an even constant parameter). Nevertheless, in both paths (even and odd) the SpTs are updated. All test programs are configured with 32 Threads per block and one block taking into account that the execution of one warp in the SM is enough to test the fields in the pipeline registers.

Method to test the kernel parameters fields

- *The GPRS size*

  These fields define the number of registers to be employed for each thread during kernel execution and are programmed during the device configuration stage. Thus, one kernel is not able to generate the required test patterns. The proposed method is based on designing three program kernels forcing the compiler to use an expected number of registers and generate the patterns.

  Test kernels GPR_T_3R, GPR_T_12R and GPR_T_63R were designed using 3 (0x03), 12 (0x0C) and 63 (0x3F) registers, respectively, which are also the patterns for the target field. The pattern selection followed the guidelines of the CUDA Tool-kit manual. The GPUs with computer capability 1.0 is able to handle up to 64 registers per thread. Over this limit, the compiler generates optimizations or data transfer to other memories.

  GPR_T_3R program executes one logical and one arithmetical operation. This is configured with 1024 Threads per block and one Blocks per grid in order to use a complete SM. GPR_T_12R kernel executes a set of addition operations on global memory locations. This program is configured with 64 threads per block and one blocks per grid. Considering that RF placement policy assigns the registers of each thread in a consecutive way, it is possible to detect faults with this configuration. Finally, GPR_T_63 kernel computes an accumulative addition using each register as part of the result avoiding the optimization by the CUDA compiler. The kernel is configured with 256 threads per block and 1 blocks per grid. Each test program includes an SpT. Kernel termination and SpTs are used as observability mechanism for fault detection.

GPR_T_3R is employed to test permanent faults in “1” on the higher part of the register field. Similarly, GPR_T_12R detects those in the higher part and the lower part of the field. GPR_T_63R is employed to test permanent faults in “0”, considering that a fault would overlap other thread registers, thus, corrupting the result. GPR_T_3R kernel was also able to generate patterns to test other control-path fields by employing a large number of threads in the kernel.

- *GPRS and shared memory base fields*

  These fields are also programmed during the device configuration stage. Nevertheless, the execution of multiple blocks or a large set of threads per block, in the same SM, is able to generate test patterns on these fields. The proposed approach is a combination of both approaches.

  GPR_T_3R kernel is reused to test the low part of the target fields, considering that it uses the maximum number of threads per block and a low number of registers. On the other hand, the high part requires the assignation of distributed memory addresses across the RF. For this purpose, we designed one kernel (B_T) employing 16 registers per thread and configured with 8 block per grid. In this way, the 9 bits in the GPRS base field can be tested. The test kernels execute a set of arithmetical operations in order to employ the selected number of registers; finally, the SpT is stored in global memory.

- *Other register fields*

  This kernel targets the missing register fields in the control-path fields. Most of them are presented in D-R register and are composed of the instructions op-codes, predicate registers flags, immediate operands values, and logical and arithmetic selectors. It was considered to employ a pseudo-random kernel employing most instructions, thus generating most test patterns. Nevertheless, this solution is feasible only when the ISA of the GPU is well known and it is directly generated at the assembly level. Nevertheless, CUDA employs multiple compiler optimizations removing instructions or modifying those that do not contribute to a thread result in memory. This restriction minimizes the effectiveness of this method.

As a solution, this kernel employs most representative instruction op-code to increase missing FC through selective operations. Then kernel (PSR_T) is designed to generate the highest number of potential variations on some selected target fields. Those fields are: *i)* operand order and sign, *ii)* immediate operand parameters, *iii)* predicate flags and *iv)* op-code, which represent a high percentage of the missing faults.

Targets (*i* and *ii*) require the generation of multiple arithmetic operations. On the other hand, the pattern for (*iii*) requires the explicit comparison of parameters. In order to avoid the compilation optimizations and force it to generate the expected pattern in those fields, the comparisons are made based on memory locations. An initial approach considered seven comparisons (!=, <, ==, <=, >, < | >, >=) to generate the patterns considering an unknown op-code. In the optimized version, it was required only two comparisons.

The op-code generation was carried out employing memory and kernel parameters movement combined with shift operations. Those instructions were analyzed following the CUDA compilation and analyzing the assembly code of multiple arithmetics and movement operations. The kernel is configured with 32 Threads per block and one block per grid, since SM shares those fields during the execution of a warp.

- *Compiler restrictions in kernel implementation*

  In some of the proposed methods, problems and restrictions were faced during kernel implementation. Those restrictions are caused by the CUDA environment, which employs advanced algorithms for resource reduction and performance improvement in the application.

  In the PC_T program, the RET instructions were manually added in free memory locations to terminate the program execution, since; CUDA compiler removes all instructions without any direct relation with the kernel execution. Besides, each subroutine was placed in the target location. Similarly, in WS_T_x kernels, the implementation required the explicit comparison of each thread identifier with the constant parameter independently in order to generate the expected divergence.

  In the description, the GPR_T_xR kernels avoid arrays and matrices definitions. In these kernels, the register declaration is replaced with an independent declaration of each variable. A consecutive register declaration, such as an array would be interpreted by the compiler as local memory locations for the kernel, so limiting the target of the test kernel. Moreover, the command to increase the registers usage per thread was required in order to guarantee the total of registers employed in the test kernel. Finally, every register, in a thread, should be part of a memory store operation in order to avoid optimizations and reduce the number of registers employed.

4.2.3 Experimental results

The fault simulation environment is the same introduced in Chapter 3, Section 3.1.1. For the experiments, the environment was configured targeting permanent faults and injected one fault per simulation. the input fault list was divided into ten parts.

In the experiments, we injected 2,382 faults in the control-path fields of the PRs. For the purpose of this paper, we only considered the RT-level model of the GPU, hence, the analysis is limited to the stuck-at faults on the inputs and outputs of the Flip-Flops composing each pipeline register. FlexGripPlus was configured with one SM and 32 SP-cores in the SM during the fault injection campaigns. Fault simulation campaigns required about 6 hours to be completed. The experiments were performed on a workstation with an Intel Xeon CPU running at 2.5 GHz, equipped with 12 cores, and 256 GB of RAM.

We performed injection campaigns on four representative benchmarks (*Vector_Add*, *MxM*, *FFT*, and *Edge*) to compare the FC of applications with the one provided by the proposed solutions.

Table 4.13 shows the characteristics of the developed SBST kernels and the four applications. It shows that most of the kernels are composed of a low number of instructions and have a short execution time. Tables 3 and 4 show the achieved FC in the targeted structures. The total FC does not consider Functionally Untestable Faults (FUFs) in the system, i.e., faults that cannot be tested resorting to a functional approach. For the identification of FUFs we adopted a method derived from the one presented in [170]; unfortunately, the identification of all FUFs in a complex circuit goes beyond the state of the art techniques. For the considered applications, Table 4.14 presents the average result of multiple simulations employing various data input sets. Results show that benchmarks provide a relatively moderate FC (32-57%). Moreover, a high percentage of fault effects are detected through hanging conditions in the system, stopping the operative state of the device.

VectorAdd and *MxM* applications employ mainly data-path structures including data-path fields in the PRs of GPUs. However, the execution is affected by the incidence of a fault in the control-path fields. In the first application, faults are distributed among a system hanging condition and an SDC in results. In contrast, most faults in *MxM* generate hanging conditions, due to the execution of flow-control and conditional instructions. Similarly, *FFT* and *Edge* applications, which are composed of multiple control-flow instructions, are more prone to fault effects belonging to the system hanging category.

Every proposed SBST kernel achieves a low FC (lower than 40%). Nevertheless, as explained below, the multi-kernel test approach is composed of multiple kernels,

Table 4.13: Characteristics of the implemented test kernels and applications. (*) using the CPT.

SBST kernels or Benchmark	Execution time (Clock Cycles)	Memory size (Bytes)
WS_T_D	16,449	128
WS_T_V1	2,175	128
WS_T_V2	1,913	128
TR_T	2,273	384
GPR_T_3R	23,586	8,192
GPR_T_12R	103,930	400
GPR_T_63R	283,714	1,500
PC_T	31,570	128 / 256(*)
B_T	178,750	9,256
PSR_T	7,313	2,304
VectorAdd	28,565	768
MatrixMul	201,365	768
Edge Detection	688,305	2,048
FFT	584,265	512

Table 4.14: FC of selected applications in control-path fields of PRs

Kernel	SDC (%)	Hanging (%)	Timeout (%)	Total FC (%)	Testable FC (%)
VectorAdd	18.10	20.82	0.62	32.37	39.54
MatrixMul	9.74	42.67	0.92	43.66	53.33
Edge Detection	19.89	49.44	1.03	57.60	70.36
FFT	21.89	42.36	0.67	53.15	64.92

designed to target different pipeline registers fields, and executed independently. The FC in the control path is obtained as the cumulative number of faults detected by all the test kernels. The joint testable FC of those kernels reaches a relatively high percentage (80% in control-path). Moreover, the multi-kernel SBST approach is able to detect up to 38.31% of the permanent faults employing only memory results, a traditional mechanism for in-field test. On the other hand, the benchmarks are only able to detect 21% of the faults with the same detection mechanism, showing the effectiveness of the SBST approach.

The *Edge* kernel can detect 70% of the permanent faults in the control-path field of the PRs. Nevertheless, 49.4% of it is through hanging detection. On the other hand, the proposed kernels reduced in up to 26% the hanging conditions and translating them into memory errors.

The multi-kernel approach also guarantees that the in-field test can be performed employing chunks (multiple kernels) with short execution time. In Table 4.15, we reported both the total and the Testable FC%. The total FC has been computed excluding FUFs. Multiple FUFs can be found in the control-path of the GPGPU. These include faults affecting the two lowest bits of each WPC pipeline register, the initial active thread mask, and some other fields that are present in the design but did not affect the benchmarks or the SBST kernels execution. In the end, 456 faults in the control-path are labeled as FUFs.

Table 4.15: FC of the proposed SBST approach

Kernel	SDC (%)	Hanging (%)	Timeout (%)	Total FC (%)	Testable FC (%)
WS_T_D	4.61	25.23	16.67	38.08	43.51
WS_T_V1	4.77	23.33	13.85	34.34	41.95
WS_T_V2	4.82	23.64	13.95	34.72	42.41
GPR_T_3R	14.51	21.85	0.82	30.35	37.07
GPR_T_12R	16.77	21.49	0.51	31.74	38.77
GPR_T_63R	20.10	22.49	0.56	35.10	42.87
B_T	9.13	22.51	1.23	26.91	32.87
PC_T	21.69	17.59	0.41	38.37	39.69
PSR_T	19.74	23.54	4.46	39.08	47.74
Overall	38.31	23.44	18.51	65.70	80.26

Table 4.16 reports the FC in the control-path fields for each PR using the proposed multi-kernel approach. The proposed method seems to be globally effective for fault detection on most of the fields in the pipeline registers.

Table 4.16: FC in the individual pipeline registers (PRs)

Pipeline Register	SDC (%)	Hanging (%)	Timeout (%)	Total FC (%)	Testable FC (%)
F-D	51.24	16.17	9.20	64.98	76.62
D-R	27.89	8.45	6.06	38.49	42.39
R-E	36.21	25.0	19.82	46.69	81.03
E-Wr	33.96	14.71	18.45	50.0	67.11
Wr-W	45.36	28.35	16.49	65.79	90.21
W-F	50.48	25.48	15.38	68.21	91.83
Overall	38.31	23.44	18.51	65.70	80.26

The relatively low FC in some PRs (D-R, and E-Wr) is mainly caused by restrictions stemming from the adoption of a high-level kernel description and implementation using the CUDA compilation tool. This tool sometimes removes or changes the execution order, instruction type, and operand placement in the device depending on the program description, the compiler configuration, and the optimizations options. This behavior is intended to increase the execution performance in the device. Nevertheless, from the reliability viewpoint, this abstraction level introduces restrictions for test pattern generation in fields, such as instruction opcodes, special operand types, physical memory addresses and fields that depend on external configuration units such as the block scheduler. Most previous registers fields are found in the D-R and E-Wr PRs. A naïve solution to improve the FC is the addition of assembly instructions, as we did in some of the proposed methods and increasing by up to 10% the obtained FC. However, this solution is feasible only when the SASS specifications are complete.

It is worth noting that, the proposed approaches target the fault detection employing the SDC mechanisms (i.e., looking at the memory content). This effect can be observed in all PRs results. Moreover, the proposed methods were partially effective in detecting faults in the targeted data-path fields by checking memory errors without affecting the system operation.

4.3 Modular testing of GPUs

This section describes a modular approach to develop functional testing solutions based on the non-invasive Software-Based Self-Test (SBST) strategy. The proposed strategy is based on a scalar and modular mechanism to develop test programs based on schematic organizations of functions, so allowing the exploration of different test solutions using software functions. The FlexGripPlus model was employed to evaluate experimentally the proposed strategies, targeting the embedded memories in the GPU. Results show that the proposed strategies are effective to test the target structures and detect from 98% up to 100% of permanent stuck-at faults.

Several works demonstrated that SBST solutions [17] could be successfully integrated into safety-critical applications, such as the automotive ones [171]. Most previous works on GPUs proposed SBST strategies targeting some data-path modules [172], including the execution units [82] [154], the register file [71], the pipeline registers [173] and some embedded memories [174]. Moreover, other solutions targeted critical modules in the control-path (i.e., the warp scheduler [175], their internal memories [166][176], and parts of the convergence management unit [177]). Nevertheless, to the best of my knowledge (and as observed in sections 4.1 and 4.2) most of the proposed strategies were designed after relevant programming efforts and analyses, as custom solutions for each specific module using the specific micro-architectural details. Thus, it reduces the possibility of any portability and generalization. It means that practical strategies to provide convenient procedures in developing SBST mechanisms are still missing in parallel architectures, including GPUs.

In this section, some critical memory modules present in the FlexGripPlus model are employed to validate the proposed modular strategy. In fact, the modular approach exploits a key feature of most SBST strategies in which test program corresponds to a combination of several routines, which are linked together and integrated into the test program. Thus, each routine's intended functionality can be seen as a 'modular' and independent block. This abstraction level (routines as blocks) can be used to explore alternative descriptions and observe the advantages and limitations of diverse topologies for a given target. Moreover, this method allows to port test routines between different targets, simplifying the development of functional test programs [177, 174].

In this section, a detailed description of the proposed modular strategy to generate test programs is provided. Moreover, the exploration of different test-program topologies for a given module is depicted and discussed. Then, several different test routines (in a test program) are implemented for the same module under test, considering operational constraints. Finally, the validation is performed through experiments using the small embedded memories in the GPU core.

This section is organized as follows. Subsection 4.3.1 introduces a basic overview of the target modules in the context of the GPU architecture. Subsection 4.3.2 describes the modular SBST strategies to test permanent faults. Subsections 4.3.3, 4.3.4, and 4.3.5 describe the procedures to develop test programs in the stack memory, the Predicate register file, and in the address register and vector register files, respectively. Then, subsection 4.3.6 reports the main constraints and limitations during the developmentof the test programs. Finally, subsection 4.3.7 reports the experimental results.

4.3.1 Embedded memories in the GPU core

Inside the SM core, several embedded memories are used to indirectly access to memory resources, to store the predicate flags, after the execution of conditional instructions, and to store information for divergence management. These embedded memories are limited in size, in some cases lie inside controllers, making hard and expensive to add fault detection or mitigation structures, such as ECC or BIST.

Stack Memory

This special-purpose embedded memory is located inside the convergence management unit, see Appendix A for additional details. This unit stores the starting (divergence point) and ending (convergence point) addresses when a conditional assessment instruction is executed by a warp. More in detail, the memory contains a set of 32 Line Entries (LEs). The number of LEs is directly related with the number of threads in a warp and the maximum number of nested divergences per warp. A divergence point can be defined as the address, in a parallel program, where two paths (*Taken* and *No-Taken*) are produced by effect of a conditional operation, so causing intra-warp divergence (threads in a warp execute different paths with different instructions). Furthermore, a convergence point is the location in the parallel program where the intra-warp divergence ends, so the threads in a warp execute one path again.

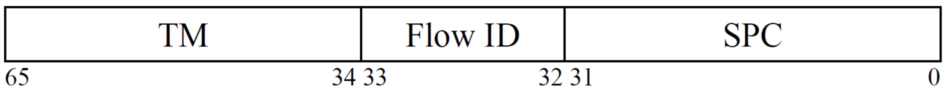


Figure 4.15: Organization of the LEs in the stack memory of FlexGripPlus.

Each LE in the stack is composed of three fields, (see Fig. 4.15). These fields are the 'thread mask' (TM), the flow ID, and the 'program counter of a warp' (SPC). The TM stores the status of the active threads in a warp and an active logic state represents the number of active threads executing a path (Taken or No-Taken). The flow ID represents the execution state of the intra-warp divergence. This field can be "01" (for a branch condition) or "00" (for a convergence point or embarrassingly parallel condition). The SPC can store the starting address of the paths or the convergence point address after both paths are executed.

The CMU employs two LEs to manage the intra-warp divergence. The first LE stores the convergence point (also known as synchronization point) and the number of active threads at the moment of starting the divergence. The second LE stores the starting address for the No-taken path and the threads to execute this path. It is worth noting that the CMU uses a new set of LEs to store the status once nesting divergence is produced.

Predicate Register File (PRF)

This module stores the predicate flags after the execution of conditional assessments, by each thread, in a warp. These conditional assessments are the product of logic-arithmetical operations or explicit setting operations. When the GPU model is configured with 8 SPs, 2,048 one-bit size locations are assigned per active thread. These locations are divided in groups of 4-bits registers (C0, C1, C2 and C3) and distributed among the available threads. Each predicate register Cx stores the logical state of the zero (Z), the sign (S), the carry (C), and the overflow (O) flags for each thread. The flags remain constant in the subsequent clock cycles until the execution of a new instruction affects their state. Furthermore, these predicate flags are also used as conditions for the executions of instructions, so these are commonly read before the execution if required. Recent implementations of the PRF provide support for up to 8 predicate registers per thread.

Address Register File (ARF)

This module is a structure of registers devoted to perform indirect indexing for external memories to the SM, including the shared and constant memories. These additional registers are mainly used in case of performance optimization for the several threads in a program and are mainly focused on the efficient access of memory sectors organized as arrays or matrices. Furthermore, the ARF reduces latency of accessing frequently used data by a kernel.

Each one of the eight SPs has an associated ARF module composed of 512 registers of 32 bit-size holding up to 128 threads. Each ARF module is distributed among the threads, so four registers (A0, A1, A2, and A3) can be employed per thread.

Vector Register File (VRF)

This is a massive structure composed of 16KB general-purpose registers of 32 bit-size and located inside of an SM. This structure is the fastest element in the memory hierarchy of the SM and is one of the most critical units in the operation of a thread, since most instructions store or load operands from this structure. The VRF is divided among the eight cores and it is distributed among the threads in a program during the configuration phase.

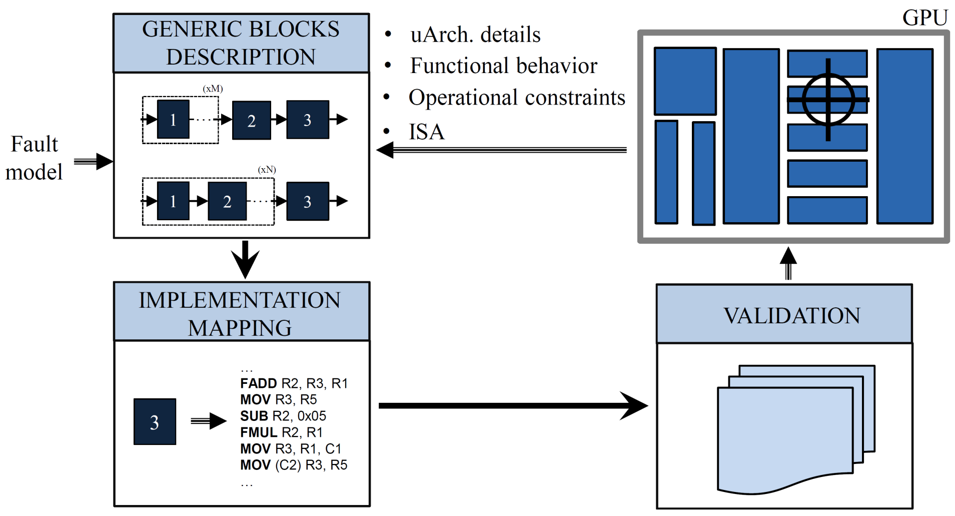
Since recent GPU architectures protect the VRF against fault effects through ECC structures, this module is not considered as the main target for the development of SBST programs. However, we employ this module to validate and also explore different options of implementing test programs.

4.3.2 A modular approach of functional testing

The modular approach for testing is a generic strategy to develop functional test programs taking into account the microarchitectural composition of a target module, the interaction with the parallel architecture of the GPU, its functional operation, its constraints, and the fault model. This modular approach is based on the development of a group of generic procedures, which are represented as a set of interconnected blocks, that once translated, compose a test program.

The approach for modular testing considers three steps: *i)* Generic blocks description, *ii)* Implementation or mapping, and *iii)* validation, see Figure 4.16.

In the beginning, the organization of the test program is initially defined as a set of generic high-level blocks, which are then divided into a group of interconnected procedures to generate the intended test functionality. This modular abstraction



```
...
FADD R2, R3, R1
MOV R3, R5
SUB R2, 0x05
FMUL R2, R1
MOV R3, R1, C1
MOV (C2) R3, R5
...
```

Figure 4.16: A general scheme of the proposed modular approach to develop functional test programs.

provides flexibility that can be used to explore and address different approaches of functional test in any module.

The Generic blocks description is a strategy to represent the behavior of the interconnected procedures aiming the test a given target module. This representation considers the operation of the module and its interaction with the system, the operational constraints, and the features of the target fault model.

In this stage, the most relevant functionalities of each hardware module are employed to define a sequence of generic procedures (blocks) that, once combined, allows the functional test of the module. Each procedure is intended to aim one of this three functionalities: 1) *Fault Controllability*, 2) *Fault Observability* or 3) *Program Monitoring*. *Fault Controllability* procedures are directly related to the ability to inject test patterns through the available instructions and structural resources. *Fault Observability* procedures propagate the effect of a fault in the module into one of the available outputs of the GPU, such as control signals, buses, or external memories.

In principle, a *Fault Controllability* procedure injects test patterns in the target module. However, the feasibility of applying those patterns must be evaluated for each target module. It is worth noting that SBST is a non-invasive test strategy, so it is possible that some modules have not controllability support (i.e., instructions able to activate faults in the module) to apply a test pattern. On the other hand, *Fault Observability* procedures describe the feasible methods to propagate the fault effect to any visible output. This feature becomes important in modules that are operated by different threads. In a parallel architecture, such as the GPU, the observability of faults might implicitly include parallel fault propagation. The micro-architectural features of a module provide the composition and contribute to identifying the *Fault Controllability* and *Fault Observability* procedures for a module. Finally, *Program Monitoring* procedures introduce optional management, tracing, or check-pointing in the test program. They can be used to increase the observability of faults or other purposes, such as the test program's division into parts. In this stage, a general analysis of the module's observability and controllability allows the definition of the procedures to integrate the functional test program.

The operational constraints and the target fault model provide the relevant limitations regarding the controllability or the fault's observability. At this level, the constraints are used to propose alternative procedures aiming at the management or even removing these limitations. The features of the fault model are used as complementary information to verify that each procedure and the combination of them allow the test of faults.

For illustration purposes, the method is supported in a scheme describing the procedures (software functions) of the modular test program, see Figure 4.17. This scheme is composed of *blocks* (1, 2 and 3), representing modular procedures, and a set of *interconnections* (Arrows), indicating the serial sequence of operations during the test. Finally, dotted modules in a scheme represent loop functions as the repetition of one or several blocks. It is worth noting that in parallel architectures, and depending on the structural location of the target module, several threads might execute the same test program in parallel.

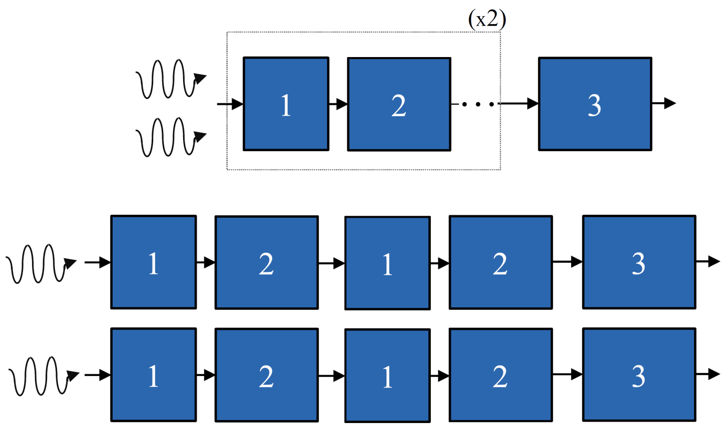


Figure 4.17: An example of the modular organization of a test program. General organization and test program description (Top), Equivalent organization per thread (Bottom).

The flexibility of the modular approach allows the exploration of test programs by composing different block interconnections and different routines, so potentially allowing test designers to explore different benefits or constraints in the development of a test program.

The second step (Implementation or mapping) builds a test program by translating the blocks in the modular scheme into equivalent software routines. In fact, the modular schemes only consider the main functional features of a test program and include all microarchitectural constraints. Thus, the implementation stage is based on the translation or mapping of each procedure (block or function) into the equivalent software routines using the available instructions of the GPU's microarchitecture. In this step, the interconnections and internal loop are also considered and included in the mapping process.

The identification of the specific set of instructions allows the operation of any intended functionality from the modular organization, so aiming the test of the targeted modules. More in detail, the implementation of the modular test program requires the use of an incremental approach. Initially, each block (procedure) is analyzed and translated individually. Then, a preliminary evaluation is performed to verify the functional test operation. This process is repeated for each procedure in the modular scheme. Then, the main interconnections among the blocks are mapped and checked. Finally, the internal loop is automated to provide portability according to the number of resources per thread and warps in the test program.

Each procedure (block) can be composed of a number of instructions ranging from simple instruction up to complex program procedures. Similarly, the internal loop requires the addition of several instructions at the beginning or at the end of the routines, which are commonly included to manage the control flow and the sequence of the program. In the end, the blocks of all proposed SBST strategies are described as a combination of a high-level programming language, such as CUDA (when possible), and instructions at the assembly-level (SASS for the used GPU). The main advantage is that minimal changes in a block (procedure) are required to change the functionality of a test program (i.e., the test can be focused on fault detection or diagnosis).

Finally, in the third step (validation), the self-test routines are verified using fault injection campaigns into the target modules. The output reports can be employed to improve the quality of a test program.

The following subsections describe the development of the functional blocks and schemes used to develop SBST programs for the divergence stack memory, PRF, ARF, and the VRF in a GPU core.

4.3.3 Stack Memory

As introduced above, the stack memory is a particular module in a parallel architecture. This module is part of the control-flow management, so one warp may access this memory to push or pop information.

Controllability

The controllability (and the injection of test patterns) of this module is achieved by forcing the execution of controlled divergences for each thread in a warp. The generation of divergences forces the stack to store the information of the number of active threads in the TM field and the starting instruction address in the SPC field, so both fields are excited each time a divergence is produced. When more than one divergence is produce, two possible effects in the stack are observed. In the first case, a serial divergence only access the same LE in the stack and changes the values of TM and SPC. On the other hand, a nesting divergence changes the target LE and both values (TM and SPC) are stored in a new addressed LE.

The detection of permanent faults in the stack is reduced to generate and perform a sequence of divergence paths as a method to excite the TM field of each LE in the Stack memory.

Using the previous information, we propose two possible methods to control the address pointer of the LEs and inject test patterns in both fields (TM and SPC) of the LEs in the stack.

The first method (*Nesting*), see Fig. 4.18 (Top), generates test patterns by using a sequence of recursive intra-warp divergence routines, so nesting functions cause the movement of the address in the stack pointer into a deeper LE. The divergence is produced by successive conditional assessments between the thread identified of each thread in a warp and constant values, so generating an ordered number of comparisons (following a specific path, blue path in Fig. 4.18) and producing the required test pattern in the TM field of each LE.

On each comparison, one or a group of threads is disabled, so defining a pattern to be stored into the deeper LE and generating two execution paths. This method is useful in managing the addressing of the LEs and injecting patterns into the TM field. The routines on each path (Taken and No-taken) expose the presence of a permanent fault in the TM. The previous process is repeated for half the number of threads in a warp, hence two LEs are required during nesting divergence management. Once the Taken routine finishes, the DMU submits the No-taken path routine when a fault is present.

A fault-free divergence stack always executes the routine in the Taken path, which generates new divergence paths and forces the test of other levels of LEs. Moreover, once a divergence is generated, two LEs store the synchronization point and the address to start the not-taken path (which can be used as test patterns). Thus, a fault can be detected when retrieving the stored values, or when the number of threads executing a path is different from the expected one, so making the fault effects visible in the outputs.

The Nesting strategy can inject test patterns on the even LEs of the stack memory. However, the odd ones are missing. The generation of test patterns for these fields requires the explicit addition of one synchronization function (SSY) before start the comparisons causing the divergence. The effect of SSY is the movement of the address pointer to the next or deeper LE in the stack memory. Then, the same previous procedure can be applied again, so testing the odd LEs.

On the other hand, the main issue of this strategy is the procedure to manage disabled threads. When a thread is disabled, this cannot be turned active again until the divergence paths are executed, and a convergence point is reached. Thus, it is not possible to test or detect a permanent fault in a deeper LE location. This restriction implies that the comparisons should be performed multiple times, targeting different threads in the TM field. We anticipate that this strategy may suffer from considerable code length and excessive execution times.

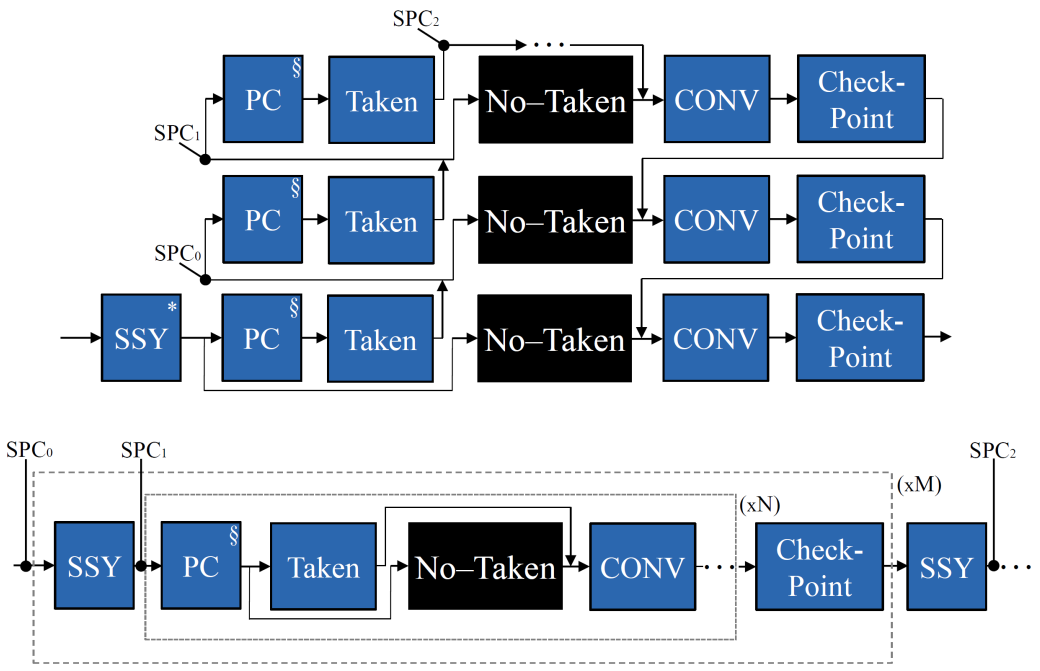


Figure 4.18: A general scheme of the proposed modular SBST strategies Nesting (Top) and Sync-Trick (Bottom) to test the stack memory. (*) Optional function to test the odd LEs. (§) Optional functions to distribute the test functions in the system memory.

The second method (called *Sync-Trick*), see Fig. 4.18 (Bottom), exploits the functionality of synchronization functions (SSY) to deceive the CMU when testing the stack memory. This method allocates SSY functions in strategically selected locations in the test program to generate the movement in the stack pointer.

More in detail, one SSY is explicitly located before each sequence of controlled divergence functions to test the TM of a LE. Hence, this function forces the controller to allocate a new level of LE in the memory without the need to generate an intra-warp divergence explicitly. The advantage of this method is that each LE can be addressed without the need of disabling specific threads to create nesting addressing of the memory. Thus, this strategy replaces the generation of nesting divergence by the management of the stack pointer and the execution of sequential controlled divergences.

The sequences of intra-warp divergence operations, generating the Taken and No-taken paths, inject the test patterns into the target LE. This process can be repeated N times (number of threads in a warp) to use different active threads and memory addresses as test patterns. Then, a new SSY addresses a deeper LE and the test procedure is restarted. It is worth noting that this mechanism is effective to move across one direction and reach deeper LEs in the memory. However, the returning phase (to a previous LE) requires the achievement of the convergence point address, which is initially stored in the stack by the execution of the SSY instruction.

Observability

The fault effect propagation is achieved using the Signature per Thread (SpT) strategy [173, 166, 178], introduced above. Each SpT is updated, taking advantage of both paths (Taken and No-taken) produced during an intra-warp divergence. Thus, the same mechanism used to test faults is used to increase the observability of the structure under test. Each SpT computes and accumulates intermediate results for each verified LE. The SpTs are finally grouped and stored in global memory for later analyses.

Test Program Organization

The interconnections and the main architecture of each proposed test approach are defined knowing the stack memory's observability and controllability methods.

Figure 4.18 presents the basic schemes of the modular composition of the Nesting and Sync-Trick test mechanisms.

In the first case (Nesting), the test generation is based on nesting divergences, so the general test strategy starts with selecting the target LE in the stack (optional use of the SSY function). Then, one divergence (two paths) divides the number of active threads, followed by the execution of a taken routine. This routine is in charge of update the SPT in the active threads. Then, a new divergence is produced (two new paths). Similarly, the same procedure restarts again, and a new taken routine is operated. The previous procedure continues until all LEs in the stack are addressed. Finally, the No-Taken paths are operated before reach the routine of convergence (CONV).

In the second approach (Sync-Trick), the operation starts selecting a target LE in the stack (using the SSY function). Then, the divergence is generated, and the two paths are created. The taken (function) path updates the signature per thread, and, finally, the Not-Taken path is executed, and both paths reach the convergence function (CONV). The previous procedure is repeated as the number of threads in a warp (N). Then, a new target LE in the stack is addressed, and the procedure is restarted again until the total number of LEs in the stack (M) is tested.

In both schemes, the address pointers SPC0, SPC1, and SPC2 represent each block function's effect on the stack address pointer and the values stored in the SPC field of the stack memory.

As depicted in both schemes, the PC and Check-point procedures are also included in the test strategy. These complementary functions are introduced to increase observability or to allow the division into parts when possible.

A control-flow routine (PC) can be included before or in one of the divergence paths to test the high bits in the SPC field. In fact, a detailed overview of the SPC field revealed that this field is partially tested. This issue is mainly caused by the short length of the test program for both strategies. In order to complete the test of the SPC field, the test routines are redistributed across the system memory, so generating the missing test patterns and the PC routine is used to address those test routines distributed in the memory.

The check-point routines are included to verify the testing of the SPC field of the LEs in the stack. These routines are located after the convergence point. In this way, any permanent fault in the SPC is detected when the convergence point or the starting address of the No-Taken path are incorrectly read from the LEs by the effect of any permanent fault. A fault in the SPC field generates an unexpected addressing in the system memory. The permanent fault is detected by mismatches in time execution and through the signatures stored in the global memory.

The check-point routines verify, through a check-point signature, the correct flow execution of a program. Moreover, this function compares an expected check signature value with the actual accumulated value during the test program's execution. When the comparison matches, the accumulated signature is updated, otherwise the test program finishes propagating in memory the error in the SPC field of the evaluated LE. The same strategy can be applied to any of the two controllability methods (Nesting or Sync-Trick).

The use of these additional functions (Check-Point and PC) is optional, considering that these strategies are costly in memory overhead for an in-field execution. It is worth noting that the proposed technique takes into account the operational restrictions to develop the test programs using the Stuck-at fault model. Other fault models would require the adaptation of the Sync-trick mechanism. However, it would be hard or impossible to follow the Nesting strategy. The convergence function (CONV) synchronizes both paths' operation and restart (from that point) the embarrassingly parallel operation of all threads in a warp.

Implementation

The synch-trick strategy cannot be directly described in CUDA, and explicit assembly level descriptions are required. In contrast, the Nesting mechanism can be directly mapped into the CUDA without modifications. The implemented code for both test methods is composed of the following functions: *i)* Initialization function, *ii)* synchronization function (SSY), *iii)* flow control function (PC), *iv)* intra-warp divergence function and SpT update functions (Taken and No-Taken), and *v)* check-point function (Check-point).

Each function is described independently and can be attached depending on the target of a test program. The initialization function defines and initializes the registers for each thread. Moreover, this function initializes the addresses to store the SpTs and check-point signatures. The functionality of other functions was introduced in the previous section.

Two main operations can be employed to manage the addressing of LEs in the stack memory. Initially, the convergence function is implemented using one synchronization instruction (SSY), which affects the stack pointer in the memory, and moves it to the next LE. When the program reaches the convergence point, the pointer returns to the previously addressed LE. During the execution, the first LE is used only for storing purposes. In contrast, the second LE is employed during the management of the divergence, and control-flow instructions can affect this LE with writing or reading operations. Thus, when the operation of the first path ends, the information in the second LE is used to start the not-taken path until the convergence point is reached. The CONV function, which is interpreted as the return from an addressed LE to the previous one, is described using exit control-flow instructions, such as (NOP.S).

The PC functions are relocations in the memory of the intra-warp divergence routines. These PC functions require of some instructions (in the format of 32 and 64 bits) located before each relocated function. These instructions avoid hanging conditions by permanent faults in the SPC field. In this way, when the program counter is affected by a fault, and it jumps to any unexpected memory location, it is always possible to retake control of the program and finish the execution of the GPU. Nevertheless, it is expected degradation in performance by the effect of the permanent fault. On the other hand, the intra-warp divergence routines are generated by successive comparisons between the Thread.id values, of a warp, with a constant value. The constant value is loaded using immediate instructions. The check-point signatures are predefined before execution and also loaded through immediate instructions. Then, the two paths are executed.

The functions in both paths (Taken and No-taken) update the SpT, which is firstly loaded from memory and then increased as a counter according to the path. A similar procedure is applied in the check-point routines that update check-point signatures to verify the step-by-step execution of the program, so avoiding infinite loops or unexpected branches by faults in the SPC field.

The modular description of both SBST strategies allows the exploration of multiple options for the programs. In the Nesting method, the modular approach guides the addition of functions, such as the nesting divergence, and also provides support to add or to remove optional functions targeting the SPC field (PC and Check-point). In contrast, the modularity presents considerable advantages for the Sync-Trick method. The code description of this method is scalable and modular, so it is possible to append or remove block functions in the description of the program, targeting the individual test of LEs in the stack memory. This modularity gives us the possibility to address any or a group of LEs and to generate an independent test program. The division of the test contributes to reducing the execution time of the test program during the in-field operation of a GPU.

The Sync-Trick method can employ two approaches to evaluate LEs in memory. The first approach (Accumulative or Acc) aims the test of a consecutive group of LEs and accumulates the signatures in memory. This approach must always start from the first LE and can finish at any of the other 31 LEs in the stack.

On the other hand, the second approach (Individual or Ind) targets the testing of an individual LE and then the retrieving of signature results to the host. This approach only focuses on one of the LEs in the memory and is intended to have a reduced execution time. The performance cost (execution time (ST)) of both approaches (Acc and Ind) can be calculated using the equations 4.7 and 4.8.

$$ST(Acc) = Ts \cdot n + Ch \cdot n + SSY \cdot (n-1) \tag{4.7}$$

$$ST(Ind) = Ts + Ch + SSY \cdot (n-1) \tag{4.8}$$

where n represents the target LE in the stack memory. SSY, Ts, and Ch represent the execution time of the synchronization, test pattern injection, and check-point functions, respectively. The initialization function was not included considering that it is constant for both cases, and it is negligible in terms of duration.

From equations 4.7 and 4.8, it is clear that the cost of the Accumulative version (Acc) is higher than for the Ind version. The cost is mainly caused by the different approaches in each case. In the Acc version, the program is intended to test the number of selected LEs sequentially. In contrast, the Ind approach targets the test on one LE, so the test patterns and check-point functions are used once. The number of synchronization functions depends on the target level of LE in the stack memory.

On the other hand, the performance cost of the Nesting method is described by the expression in equation 4.9.

$$Ns = N \cdot Ch \cdot \sum_{i=0}^{m}(SSY + Ts) \tag{4.9}$$

where N represents the total number of threads in a warp, and m is the target LE to be tested. CH, SSY, and Ts have the same meaning than in equations 4.7 and 4.8. As introduced previously, the target LE could be even or odd. Thus, the starting value of i in the summation could be 0 or 1.

4.3.4 Predicate Register File

This module is a parallel structure in the GPU and it is addressed in parallel by the active threads during the execution of a program, so the maximum number of threads per core are required to perform the test of the complete module.

Controllability

The PRF stores homogeneous information, so each active thread in a program have direct access to this memory, Thus, only one procedure is required to inject test patterns. The test of each register is based on the generation of load procedures (Ld), see Fig. 4.19. Initially, Ld targets one predicate register per thread (Cx) and assigns a value by using two possible methods: *i)* conditional assessments(PRF_T) or *ii)* direct assignments(PRF_T_R2C).

In the first case, a sequence of conditional assessments causes the activation of each predicate flag, injecting a test pattern, and propagates its effect for evaluation. On the other hand, in the direct assignment, the movement operation changes the content of the predicate register. It is worth noting that in both cases, one flag was targeted to clearly identify a fault.

Observability

A function (PROP) performs conditional evaluations to identify and classify a fault. This function propagates the effect of the target predicate register. The conditional evaluations produce two paths (Taken and No-taken). Both paths are used to update an SpT to identify if a fault was present in a given flag of a predicate register. As depicted in Fig. 4.19, the blue path describes the fault-free case of the test. When there are no detected faults, the test program remains convergent for all threads in a warp. In contrast, when a fault is detected, an intra-warp divergence is produced as effect of the fault and the SpT are updated indicating a detected fault. It is worth nothing that four serial procedures of conditional evaluations are required to test each register.

Test Program Organization

Two different approaches of modular test can be proposed for this module. In the first case (PRF_T), see Fig. 4.19 (Top), the organization of the test program is fully sequencial, so the Ld procedure injects a test pattern into a target register in the PRF. Then, the Prop routine propagates any fault and evaluates the previously register in search of faults. The taken and not-taken routines are used to update the SpT and propagate any fault effect into the available outputs. The Taken routine is intended to be operated when a faults are not present in the evaluated register of the PRF. Once, both paths reaches convergence, the previous sequence is repeated as the number of exclusive predicate registers per thread (M), the total number of flags (N) and the number of test patterns to inject per register (T).

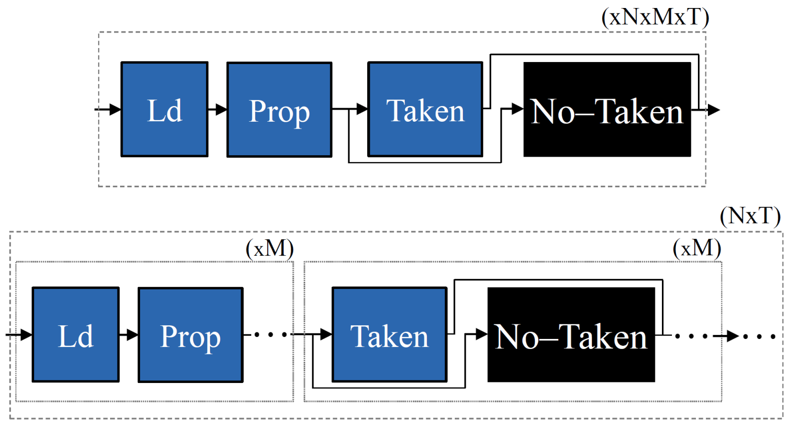


Figure 4.19: General schemes of the modular approaches to test the PRF.

In the second case (PRF_T_R2C), see Fig. 4.19 (Bottom), the organization of the test program varies and is divided as a sequence of individual operations. First, The injection is performed using the Ld procedure and propagated with the Prop routine. Then, it is repeated M times, as the number of registers, so injecting the same test pattern to each register.

The evaluation is performed using the one divergence and the Taken procedures. Again, these procedures are repeated as the number of registers per thread, so evaluating all previously addressed flags. Finally, the complete procedure of injection, propagation and evaluation is repeated as the number of flags per register and the number of test patterns to inject. It is worth noting that the main fault model target of the proposed strategy is stuck-at faults. However, it is also possible to adapt the second case to evaluate other fault models in the PRF.

Implementation

Three versions of the Ld function can be devised; one based on logic-arithmetic instructions (IOP.AND, IOP.XOR) and a second version using setting instructions (ISETP) to modify a flag in the predicate register. Furthermore, it is also possible to modify a register using a direct assignment (R2C). Thus, the main functionality (inject a test pattern) can be obtained through several descriptions. The block functions in the SBST strategy for the PRF are parametrically developed, so it is possible to easily replace one function (such as Ld) by another in the program. It is worth noting that the first two cases require several instructions before the comparison or setting.

The Prop function also supports different implementation methods. One method is based on a sequence of conditional operations, which are executed once a specific flag is active. The other method is through intra-warp divergence, so explicitly producing two execution paths corresponding to the faulty (No-Taken) and fault-free (Taken) cases. In both versions, the parametric description allows the selection of a predicate register and a target flag, so simplifying the procedures for the generation of the test program.

The routines on each path follow a similar description with respect to those developed for the stack memory, so in principle, these functions are imported into these SBST programs.

Equations 4.10 and 4.11 represent the performance cost of both strategies for testing the PRF. As you can observe, both equations are equivalent and the performance of both strategies remains the same. The main difference between the two approaches is the order of executing the test on each register of the PRF module.

$$ST = ((Ld + Prop) \cdot M + T \cdot M) \cdot N \cdot T \tag{4.10}$$

$$ST = (Ld + Prop + T) \cdot N \cdot M \cdot T \tag{4.11}$$

The Ld implementation presents the same cost for the IOP (arithmetic and logical) and the ISETP (setting) descriptions of the functions. However, for the R2C alternative, the total description and memory footprint is reduced to a total of 36 instructions.

4.3.5 Address Register File and Vector Register File

These modules are parallel in the operation of a program in the GPU and can be accessed in parallel by the active threads.

Controllability

The ARF and the VRF modules stores homogeneous information on each register, so there are not internal field divisions. This feature allows the use of only one procedure (Ld) to perform the test pattern injection. The main idea of the Ld procedure is to perform direct assignation of test patterns on each register of both modules (General Purpose Registers Rx in the VRF and address registers Ax in the ARF). The addressing routine of the registers is mainly sequential. However, the direction and the limits are defined according to the number of threads in a program (T). It is worth noting that all active threads can access the ARF or VRF during the execution of the instructions.

Observability

The propagation of a fault is performed using a function (Comp). This procedure performs conditional assessments and compare the value in any register with several predefined masks. These masks are used to identify any fault in the evaluated registers and are the base for the comparison.

There are two possible selections of the mask: *i)* Fine-grain and *ii)* Coarse-grain. A fine-grain mask allows identifying the location of the fault affecting a register. However, several detection procedures are required. On the other hand, a coarse-grain mask allows the rapid detection of a fault, but it is not possible to identify its location.

After each comparison, a divergence is produced. This divergence is employed as mechanism to evaluate the propagation of a fault and also to update the SpT. The gray (see Fig. 4.20) path shows the embarrassingly parallel operations, when the test approach is used and there are not fault in the module. In contrast, the Not-taken path in black is used when a fault is detected.

Test Program Organization

The organization of the modular test programs can be defined in two methods and can be applied for both modules (ARF and VRF). It must be considered that the internal content of each routine is adapted according to the target module. In the representative schemes, we considered a test program configured with a defined number of register per thread in a warp (N), a defined number of warps (M), a predefined number of parallel threads (P) and a fixed number of test patterns to inject (T).

In the first case, see Fig. 4.20(Top), the procedure is performed targeting a sequential test detection. First, the Ld procedure injects test patterns in one register. Then, the Prop procedure propagates the faults effect (if present) and a comparison is performed using the Comp procedure. This comparison starts the divergence and the Taken path is evaluated to update the signature for each active thread. Finally, the previous procedure is repeated as the number of registers per thread in a warp, number of warps in the test program and the number of test patterns to inject.

The second approach, see Fig. 4.20 (Bottom), is intended to divide the test program into small parts, so the sequential procedure of test is replaced by a two independent stages that combined provide the same test functionality of the previously explained approach. These two independent stages are: general test patterns injection (1) and general evaluation (2). In (1), a complete sequence of one test pattern injection is performed injecting in all registers assigned to any thread. Then, in

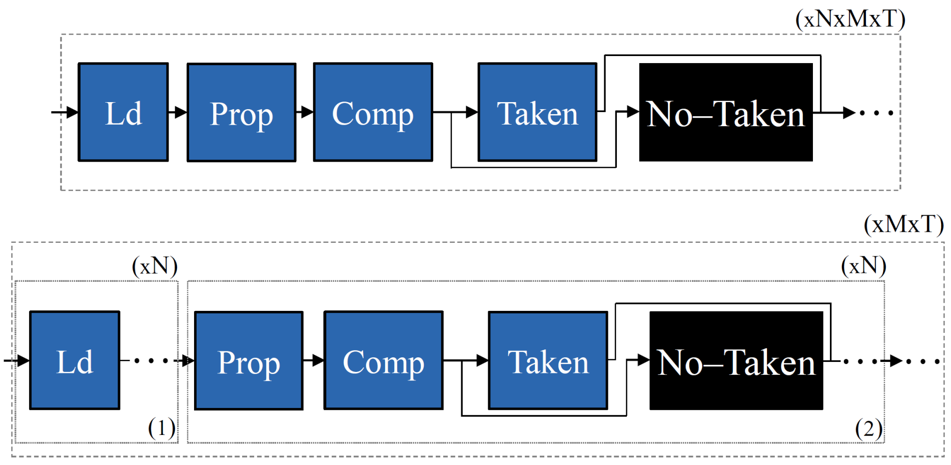


Figure 4.20: General schemes of the modular approaches to test the ARF and VRF.

the stage (2), all propagation and comparison operations are performed to identify faults in the registers. Finally, both stages (1 and 2) are repeated as the number of warp in the program and the number of test patterns to inject.

Implementation

In the case of the ARF and the VRF memories, both approaches employ similar methods to implement the Ld function. This is based on the direct assign instructions R2A and MOV, respectively. However, the ARF structure can use an equivalent instruction to perform the assignation using another address register as source A2A.

The Comp function compares and enables a flag when mismatches are found. In both cases this function is implemented using similar mechanisms through constant values (from immediate instructions or loaded from memory). Finally, the same intra-warp divergence functions are imported and adopted for these modules. It is worth noting that the development of the blocks allows different targets of the SBST program. On the one hand, it is possible to develop the functions to only perform fault detection. Furthermore, the functions can be replaced with special versions, which provide fault diagnosis features (VRF_T_dia), so allowing the identification of the location causing the fault effect. A detailed evaluation shows that both versions of the R2A and A2A routines to implement the Ld function have the same cost in terms of execution and resource overhead.

Equation 4.12 describes the performance costs for both approaches, which are equivalent from the performance point of view. Nevertheless, the test can be performed in parts (ld + comp) when the modules are free and the application can be stopped for a long interval. Otherwise, the complete evaluation of each register and the splitting into parts can be employed when short interval times can be employed.

It is worth noting that the performance of each approach is affected when faults are detected on each module by the execution of the missing path in order to update the detection signatures.

$$\begin{aligned} ST &= ((Ld + Pr + Co + Ts) \cdot N \cdot M \cdot P \\ &= ((Ld) \cdot N + (Pr + Co + Ts) \cdot N) \cdot M \cdot P \end{aligned} \tag{4.12}$$

4.3.6 Limitations and constraints

Although the proposed modular approach can be applied to any hardware module in a GPU, several programming constraints must be consider in the mapping or implementation step.

In the implementation of the SBST techniques, we explored different possible description styles for each program using high-level and middle-level abstraction languages for programming the GPU (CUDA and PTX, respectively). However, we observed some constraints related to the implementation of the SBST technique in CUDA or PTX, due to the fact that particular combinations of instructions are required to excite any of the target modules. Furthermore, the generation of specific instructions at the two levels in some cases was not possible (e.g., the SSY instruction cannot be used in CUDA or PTX levels). Thus, we adopted the assembly language (SASS) to describe most SBST techniques that cannot be directly described at other abstraction levels.

It should be noted that compilers in the programming environment of GPUs have as main target the performance optimization of a program, so removing or reorganizing the intended test program and causing a mismatch in the intended behavior of the SBST programs. The solution to overcome this issue is based on a combination of different levels of description, when possible. This solution is affordable only when the Instruction Set Architecture (ISA) of a GPU device is well known, which unfortunately is not always the case.

4.3.7 Experimental results

The RTL FlexGripPlus model was used in the experiments. Initially, the performance parameters of the implemented test programs are determined. Then, fault injection campaign results are reported showing the effectiveness of the proposed modular approach on the target modules.

Performance of the Test Programs

Table 4.17 reports the results concerning the performance parameters for all implemented test programs. It is worth noting that results reported in Table 4.17 were obtained by simulations performed resorting to the *ModelSim* environment.

Table 4.17: Performance parameters of the SBST programs using the two approaches to detect permanent faults in the LEs

Approach	**Module**	**Instructions**	**Execution time [Clock cycles]**	**System memory overhead [Bytes]**
Sync-Trick Ind	LE #1 Stack	403	33,449	1,612
	LE #2 Stack	404	34,211	1,616
	LE #10 Stack	412	34,589	1,648
Sync-Trick Acc	LEs 1 – 2 Stack	794	66,637	3,176
	LEs 1-10 Stack	3,922	326,423	15,688
	All Stack	12,524	1,030,473	50,096
Nesting	LE 1 Stack	683	37,986	2,732
	LEs 1-2 Stack	1,323	83,569	5,292
	LEs 1-10 Stack	6,443	528,086	25,772
	All Stack	19,883	2,567,209	79,532
PRF_T	PRF	434	1,890,106	1,736
PRF_T_R2C	PRF	398	1,795,596	1,592
ARF_T	ARF	122	338,240	488
VRF_T_det	VRF	82	108,958	368
VRF_T_dia	VRF	350	1,503,254	2,800

The reported results show the performance parameters for the two possible test methods of the stack memory (Nesting and Sync-Trick methods under the accumulative and individual approaches). All versions present an overhead in the global memory of 64 locations (256 bytes) devoted to saving the SpTs and the Check-point signatures.

Regarding the performance results of both versions, it can be noted that the Sync-Trick (Ind) approach maintains an average performance cost to test any LE in the stack memory. The only difference among these programs is the number of SSY instructions included to address a selected LE. Similarly, the Sync-Trick (Acc) version can test a group of LEs consecutively. However, it requires additional execution time and cannot be stopped once the test program starts.

On the other hand, Table 4.17 also reports the required execution time to test the first and the second LEs in the stack using the Sync-Trick Ind (rows 2 and 3, column 4) and Sync-Trick Acc (row 5, column 4) approaches. The Individual approach requires 76 additional clock cycles to test the LEs, but it has the advantage of being able to test each LE independently. In contrast, the Accumulative method must check both LEs consecutively. Thus, the Ind approach can be adapted for in-field operation by the limited number of clock cycles required during the execution.

The performance parameters show that for the Nesting approach, there is a proportional relation between the number of instructions and the number of LEs to test. Similarly, the relationship between the execution time and the number of LEs to test presents an increasing exponential ratio. In the end, the Nesting method requires more than twice the execution time to test the entire stack than Sync-Trick using the Acc approach. The execution time could be the relevant parameter to take into account when targeting the in-field operation.

As observed in Table 4.17, the implemented test programs, targeting the PRF and using two different *Fault controllability* procedures (PRF_T and PRT_T_R2C) require different execution times. Moreover, the number of instructions in both test programs is different. However, the intended test functionality of both versions is the same, so showing that the modular approach can produce different test programs for the same module and allowing the exploration of different test solutions.

Regarding the implemented test programs for the VRF, The fault diagnosis (VRF_T_dia) version requires up to 13 times the execution time of the test program targeting fault detection (VRF_T_det), only. The interesting of both test programs is that the modular program is the same, but the implemented functions produce the difference in time execution and intended functionality.

Fault injection results

The fault injection environment follows the methodology described in [173], and we injected permanent faults using the Stuck-at Fault model. On each target module, fault simulation campaigns were performed injecting faults in every location of each module, meaning 4,224, 32,768, 262,144 and 262,144 permanent faults in the stack memory, PRF, ARF and VRF, respectively. These fault simulation campaigns were performed using both representative benchmarks and the test programs implementing the proposed SBST strategies. Moreover, the SBST programs targeting the stack memory were evaluated with and without the optional PC functions.

The representative benchmarks have been carefully selected to compare the detection capabilities they can achieve with the ones provided by the proposed SBST programs. Descriptions and details regarding the chosen benchmarks can be found in Chapter 3. For the sake of completeness and comparison, the different versions of the SBST strategy are reported in Table 4.18. The results are reported based on the output effect of the faults as: Faults corrupting the output results, or 'Corrupted Output Data' (Data)), faults corrupting the complete execution of the system, or 'Hang', and fault affecting the performance of a given benchmark, or 'Timeout'.

The last column of Table 4.18 reports the testable FC (TFC) of the benchmarks and the proposed SBST strategy. The TFC is defined as the ratio between the number of detected faults and the number of injected faults after removing the untestable faults. The untestable faults are those faults that due to structural or functional issues cannot be tested and cannot produce any failure. A detailed analysis of the stack memory revealed that a total of 192 faults are untestable. These are related to the lowest bits of the SPC field of each LE, which does not affect the execution of an instruction. Thus, these faults were removed when computing the TFC.

The Sync-Trick strategy provides a moderate FC for both cases (Ind and Acc). Moreover, the FC increases when adding the PC functions and the relocation of the test functions in the memory. These comprehensive approaches (Ind+SP and Acc+SP) obtain a high percentage of FC for the target structure.

An in-depth analysis of the results shows that the Individual approach allows detecting 100% of the faults in the TM of all LEs by looking at the results of the test procedure "Data" type faults. In contrast, the Acc version causes a small percentage (0.75%) of faults produced in the TM field and visible because they hang or crash the GPU. This behavior can be explained considering that in the Ind approach, each LE is evaluated individually, and so all detections can be labeled as Data. On the other hand, for the Acc method, a permanent fault in one LE affects the synchronization point, thus corrupting the convergence point and causing the Hang condition. More in detail, a Stuck-at-0 fault is a sensitive case during the run of the test program. A fault affecting one LE when used as synchronization causes the Hang condition.

The Nesting SBST program has a slightly lower FC than Sync-Trick with an increment of more than twice the percentage of faults causing hanging and timeout. This fault effect is equivalent to the effect shown by in the Acc version of Sync-Trick. In this case, the Nesting method generates intra-warp divergence to move the stack pointer among the LEs (in the stack memory), testing all LEs even when a fault is detected, so other LEs are also tested. The continuous evaluation generates issues when a fault affects the LE used for synchronization purposes (when testing the even LEs). Thus, the test program may confuse the convergence point and produce the Hang or Timeout condition. According to results, the Nesting strategy seems to be more susceptible to Hang and Timeout effects than the Sync-Trick using the Acc approach.

In both approaches, the addition of the relocation in memory and the SPC functions increase the testable coverage in the stack memory. However, as explained previously, these optional functions can be employed when it is possible to use the entire system memory to relocate the test functions in specific memory locations, or the application code allows this adaptation. Similarly, both SBST approaches can detect a considerable percentage of the permanent faults in the stack memory. However, a direct comparison involving the performance parameters from Table 4.17 shows that the Nesting approach consumes more than twice the execution time and 37% of additional instructions. In conclusion, the Sync-Trick strategy seems to be a feasible candidate for in-field operations. Moreover, the Ind strategy can be divided into parts and adapted with the application code.

Regarding the SBST program for the PRF module, the two implemented versions obtain the 100% of fault coverage. The main advantage of both versions is the fault propagation to the global memory, so enabling the detection as Data. In fact, all faults detected are identified using this classification. Thus, the main difference between SBST approaches is the performance and overhead cost, which are mainly caused by the internal description of one modular function (Ld). The previous fault coverage results allow us to validate the exploration of different methods to implement different modular functions. In both cases, the replacement of a modular function does not affect the final fault coverage. A similar situation is observed in the SBST programs for VRF. In both approaches (Detection (Dec) and Diagnosis (Dia)), the programs reach a full fault coverage (100%). However, the performance degradation rises up to 13.8 times when employing the diagnosis version of the test program. Nevertheless, this SBST version can be affordable with mild system time constraints, such as during the Switch-on of the system.

Table 4.18: FC results for the representative benchmarks and the proposed SBST strategies

SBST strategy or benchmark		**Module**	**(%)**				
			Data	**Hang**	**Timeout**	**FC**	**TFC**
MxM		Stack	0.00	0.38	-	0.38	0.40
		PRF	0.00	0.38	-	0.38	0.38
		ARF	25.07	0.0	-	25.07	25.07
		VRF	18.26	8.24	-	26.5	26.5
Sort		Stack	0.15	0.04	-	0.19	0.19
		PRF	0.16	0.04	-	0.20	0.20
		ARF	0.00	0.00	-	0.00	0.00
		VRF	0.18	0.07	-	0.25	0.25
FFT		Stack	0.14	0.19	-	0.33	0.35
		PRF	0.15	0.19	-	0.34	0.34
		ARF	0.0	0.0	-	0.00	0.00
		VRF	0.19	0.21	-	0.4	0.4
Edge		Stack	0.15	0.28	-	0.43	0.47
		PRF	0.00	7.05	-	7.05	7.05
		ARF	0.00	0.00	-	0.00	0.00
		VRF	12.25	5.6	-	17.85	17.85
Sync-Trick	***Ind***	Stack	65.64	2.08	1.01	68.75	72.02
	Acc		64.89	2.84	1.01	68.75	72.02
	Ind + PC		83.00	8.49	2.44	93.93	98.41
	Acc + PC		82.24	9.25	2.44	93.93	98.41
Nesting		Stack	54.12	11.81	1.23	67.16	70.04
	+ PC		76.94	13.16	2.81	92.91	97.34
PRF_T		PRF	100.0	-	-	100.0	100.0
	R2C		100.0	-	-	100.0	100.0
VRF_T	***Det***	VRF	100.0	-	-	100.0	100.0
	Dia		100.0	-	-	100.0	100.0
ARF_T		ARF	100.0	-	-	100.0	100.0

Although Table 4.18 reports one SBST program for the PRF, the two versions were evaluated changing each time the Ld function. The two versions achieved the same fault coverage (100%) and the Ld functions, in both versions, have the same performance cost, so both solutions can be used identically.

A comparison of the FC obtained by the proposed SBST strategies and the representative benchmarks shows that the FC using these specialized programs is higher for the targeted modules than the FC obtained with typical applications. Thus, the FC capabilities of a representative benchmark is mostly lower. This behavior can be explained considering that most applications only use parts of the modules (e.g., only the first levels of stack memory to handle the divergence or certain registers per thread in the ARF or VRF modules) to operate the instructions of each application.

The matrix multiplication application generates one level of divergence. Thus, other levels inside the Stack memory are not employed, and the fault effect in not detected or propagated into the application. On the other hand, the VRF and the ARF modules are excited in almost 25% and 20%, which helps to explain the fault coverage obtained for both modules (26.5% and 25.07%, respectively). In contrast, the Sort application can generate intra-warp divergence, depending on the input data operands, but it remains limited to the first LE in the stack memory. However, the percentage of detection (0.33% and 0.19%) is negligible in comparison with the proposed test strategies. A similar behavior is observed for the ARF, PRF and VRF when executing this application.

The FFT benchmark produces two levels of intra-warp divergence, so using up to four LEs during the operation. This behavior slightly increases the percentage of faults detected. Nevertheless, the achieved percentage remains small. Finally, the Edge application causes two levels of intra-warp divergence and can detect some faults as Data and hangs. However, the total coverage of all representative kernels is minimal.

The previous scenario supports the idea that executing applications and checking their results (as it is often done when using a functional test approach) is definitely not enough to verify the functionality of crucial hardware modules in the GPU. Thus, special test programs, as those proposed in this work using the modular approach, are required to guarantee the correct operation of a module inside a device used in a safety-critical application.

The main advantage of the proposed method lies in its modularity and scalability. Scalability allows the configuration and the selection of the number of LEs to be tested in the SBST programs for the stack memory. Moreover, the test programs for the ARF, PRF and VRF can also be reduced to target only specific registers in the target modules. Finally, the scalability of their structure allows splitting the overall program in several parts, as presented for the Sync-Trick SBST program.

As introduced previously, the implementation of the test programs required the combination of high-level descriptions (about 15% of the total code in all SBST strategies), and the addition of assembly functions (about 85%). For both proposed SBST strategies targeting the memory stack, the synchronization functions (SSY) were implemented in assembly language. In this way, we could also avoid that the compiler removes or changes important parts of the test code. Similarly, in the PRF test program, the Comp modular function used specific procedures that required assembly language support. Thus, these parts and others are written at the assembly level. These limitations show that the development of test programs for these complex structures in GPUs requires access to the assembly formats to provide feasible and efficient solutions. The implementation effort could be reduced by the design of an automatic tool to include the subroutines at the assembly or binary level. Moreover, such a tool could also be employed to develop modular approaches, targeting other modules, such as functional units in the GPU.

Although the proposed SBST strategies targeted the test of unprotected memories in a GPU model with the G80 micro-architecture, we still claim that the proposed methodology can be adapted and used for the most recent GPU architectures, such as Maxwell and Pascal that include similar structures. Moreover, other parallel architectures can also use the proposed method.

4.4 Complementary test strategies and general overview of the SBST strategy on GPUs

This section describes the adoption of one classical method of functional testing from processor-based systems into GPUs. Then, a general overview of the main benefits of SBST strategies applied to GPUs is presented. In this evaluation, the overall fault coverage and functional safety analysis is performed for the complete GPU depicting the main advantages and limitations of the SBST strategy as functional testing solution for GPUs.

4.4.1 Adapting automated approaches of functional test on GPUs

In the past, functional test strategies were proposed and applied in processor-based systems (CPUs). These can be divided into two main approaches: deterministic and automated [82][150].

On the one hand, the deterministic approach exploits the functions and the architectural composition of each target module to apply well-defined algorithms, such as March algorithms in internal memories (as presented in section 4.1.1) [175]. However, as also explained in section 4.1.1, huge effort is required to adapt deterministic strategies by the additional operational constraints and operational restrictions.

On the other hand, the automated approaches of functional testing comprise the use of pseudorandom-based and ATPG-based methods to test a module in a device. The first method (Pseudorandom) focuses on the most suitable set of instructions, which are randomly combined and repeated with pseudo-random operand values. The selected instructions aims at activating any propagating any existent fault on a given module. commonly, evolutionary algorithms are used to optimize the number of instructions in a test procedure. On the other hand, ATPG-based methods resort to Automatic Test Pattern Generation (ATPG) tools to analyze and extract test patterns for a given target module. Then, these patterns are translated into equivalent valid instructions, so composing one or more test programs. It is worth noting that it is possible that some test patterns cannot be translated into proper instructions and must be ignored (possibly resulting in safe faults).

In case of parallel architectures, such a GPUs, both methods (pseudorandom and ATPG-based) are effective when applied to regular structures of a GPU, such as the functional units (i.e., SPs and SFUs), so taking advantage of the multi-thread parallelism in the GPU and using the active parallel threads to address and perform the test procedures on each module in a parallel manner [179]. The parallel adoption is required in order to exploit the parallel features of a GPU. Moreover, additional mechanisms to propagate the fault effects and to manage the input test patterns in memory are required.

In the first case, a parallel mechanism of signatures, such as the Signature per Thread (SpT) (introduced in Section 4.1) is used to indicate the host or any external controller the presence of faults in any of the functional units in the GPU. On the other hand, the organization of the input test patterns in the memory directly depends on the effective number of test patterns, the number of parallel cores and the expected performance in the test procedure. A simple organization of the test patterns can be based on immediate operations, so the memory hierarchy of the GPU is not employed. However, it is expected long execution times. Another organization method is based on storing the input test patterns in the main memory and force the parallel threads to address the same memory locations, so applying in parallel the same patterns into each active functional unit. It is worth noting that the static organization of the regular structures in the GPU (i.e., a group of threads always access a given functional unit) and the fixed distribution policies in the schedulers allow the development of embarrassingly parallel test programs and the use of the generated test patterns by the pseudorandom and ATPG-based approaches.

In order to determine the feasible adoption of the automatic approach, two functional units (the SPs and the SFUs) in the GPU are target with the purpose of develop test programs and validate the approach using the pseudorandom and the ATPG-based methods.

In the development of the test programs, the pseudorandom approach is employed considering only the available instructions addressing each target module. On the other hand, the ATPG-based approach employed all valid instructions obtained after adopting the test patterns into equivalent instructions for the GPU. In case of the pseudorandom approach targeting the SFU, four different data sizes are selected (30,000, 60,000, 90,000, and 120,000) in order to observe the fault coverage effect. Similarly, the ATPG-based approach is adopted considering immediate (IMM) operations and storing the test patterns as operands in the main memory (Mem).

Each test program is fault simulated employing the gate level representation of each functional unit. The gate level version of both functional units was obtained employing the 15nm technology library [100]. Then, a custom fault injector is employed to inject one permanent fault (Stuck-at) per simulation in the fault campaigns. In the experiments, the GPU model was configured with 8 SP cores and 2 SFUs per SM. Table 4.19 reports the designed test programs, their main features and the fault coverage obtained after each fault campaign. It is worth noting that FC results are calculated considering all available functional units in the GPU.

Table 4.19: Main features and fault coverage of several test programs for the functional units (SP and SFU) in the GPU.

Target module	**Type**	**Size (Instructions)**	**Duration (cc)**	**FC (%)**
SP	ATPG	19,604	1,447,620	84.07
	Pseudorandom	55,000	3,434,235	83.99
SFU	ATPG IMM	16,856	1,200,034	90.75
	ATPG Mem	117	212,914	90.75
	Pseudorandom 30K	309	1,546,404	82.55
	Pseudorandom 60K	609	3,074,844	77.67
	Pseudorandom 90K	909	4,603,284	89.11
	Pseudorandom 120K	1,209	6,131,724	88.26

As observed in the FC results, the adoption of automatic approaches for the development of parallel test programs is feasible and provide good results in GPUs. More in detail, results in both target modules, the use of the ATPG-based and pseudorandom approaches are able to coverage a high percentage of permanent faults in the functional units. In both modules, the ATPG-based solutions provide a higher FC with a lower number of instructions per test program which is highly desirable. Moreover, in case of the SFU, the management of the input test patterns as data operands provided a high compaction, so reducing the size and duration of the test program in up to 99.3% and 82.2% respectively, but maintaining the same FC. On the other hand, the use of different data sizes on the pseudorandom test programs provides a moderate increasing effect in the fault coverage (<10%). However, the number of instructions per test program and the test duration are also incremented proportionally.

The missing percentage of FC when using the ATPG-based approach is mainly caused by the unfeasible adoption of input test patterns as valid instructions in the GPU. More in detail, about 10% of the input test patterns were discarded in both modules by the lack of available instructions activating the functional units.

Although both automatic methods are able to detect most faults in the targeted modules, the ATPG-based test programs are more compact that the pseudorandom version and required less time to be executed, so these approach can be considered as a feasible solution if ATPG tools can be used on the target modules. In contrast, the pseudorandom approach is useful only when microarchitectural details are not available on the target modules.

4.4.2 Overall evaluation of the SBST strategy in GPUs

This subsection provides a general overview of the proposed SBST strategies on the complete GPU. It is worth noting that results do not report the fault coverage on the memory controllers or external memories in the memory hierarchy of the FlexGripPlus model, since the memory hierarchy contains several restrictions (missing cache memories and associated controllers) and it is not fully representative of real GPU devices and specific test programs or algorithms were not developed for such modules.

Each module in the RTL version of the FlexGripPlus model was evaluated with all developed test programs using the proposed SBST strategies presented in this chapter. Each fault campaign injected permanent faults on each module when executing an individual test program. In the experiments, the detection of a fault is considered only when it causes an SDC or timeout effect in the outputs for each each test program. At the end, the FC is determined considering the accumulative number of faults detected for all test programs.

Table 4.20 reports the FC results concerning each module in the GPU. As can be observed, the effectiveness of the SBST strategy in some modules is moderate, even when applying all test programs (see schedulers). On the other hand, there are several effective test programs (70% to 100% of detected faults).

Table 4.20: A summary of the FC obtained for the different modules of the FlexGripPlus GPU model using the proposed SBST approaches. All test programs were used to determine the FC of the module.

Module	Detected faults	FC (%)	Reference manuscript
SMP controller	1,156	57.8	[173, 174, 177, 166, 179]
Warp Unit	8,416	58.5	[173, 174, 177, 166, 179]
Pipeline Fetch	538	88.8	[173, 174, 177, 166, 179]
Pipeline Decode	837	70.4	[173, 174, 177, 166, 179]
Pipeline Execute	91,429	84.5	[179]
Address Register File	32,768	100.0	[174]
Vector Register File	131,072	100.0	[174]
Predicate Register File	32,768	100.0	[174]
Divergence Stack Memory	4,139	98.0	[177]
Overall GPU	303,295	92.6	all

At the end, the overall percentage of detected faults in the GPU core reaches 92.6%, which supports the argument that functional testing techniques based on SBST are feasible in-field testing solutions for GPU devices and can be considered as a complementary technique for complex hardware accelerators, as it was proved in the past for CPUs.

4.4.3 Functional safety evaluation of the SBST strategy in GPUs

In this subsection, the functional safety of the GPU is determined considering only the SBST strategies and the fault coverage results as safety mechanism. In fact, the calculation of the FC is an indication of the design safety based on the efficiency of a given Safety Mechanism. However, it is not sufficient to assure compliance with Functional Safety standards (i.e., ISO26262). For such a purpose, it is required the reduction in the probability of system failures rate (also known as Failure in Time or FIT).

The Single Point Faults Metric (SPFM), which represents the potential of the permanent faults to violate safety-related functionalities, is defined by ISO26262 as evidence of Safety Integrity [65]. The SPFM considers the total FIT rate (λ), and the contribution of the fault classes:

- Single-Point Faults (λSPF): not covered by safety mechanisms
- Residual faults (λR): undetected by safety mechanisms

The SPFM can be calculated according to the equation:

$$SPFM = 1 - \frac{\sum(\lambda SPF + \lambda R)}{\sum \lambda} \tag{4.13}$$

The primary methodology for determining the Safety Metrics parameters is the Failure Modes Effects and Diagnostic Analysis (FMEDA). The FMEDA correlates IC components (Gates, Flops, and Memory cells) to Failure Modes (FMs). Then, by computing the λ of individual IC components, the FC, and the Safe faults, we can determine the total λ of each FM.

The FMEDA starts with the definition of the FMs and the mapping of design components. For FlexGripPlus, we considered 9 main parts (see Table 4.20) and 28 internal subparts,by dividing the composition of each module in the GPU. Each subpart was analyzed to determine function-specific FMs, so determining the potential effect of a fault in the functional operation of the module and the GPU.

After mapping each FM to the appropriate design component(s), we can evaluate the percentage of Safe faults per module and the FC. The safe faults are determined after analysing the structural activation and structural propagation of faults on each module in the GPU. More in detail, the structural activation forces the primary inputs in the GPU and evaluates the propagation of any input value in the internal nodes. Similarly, the structural propagation evaluates the outputs of the GPU and correlates the internal connections in order to find all possible nodes in the modules which can be propagated into the available outputs. In the end, those internal locations in the modules that are not related with the activation or propagation analysis are considered as safe faults for the GPU, since there are not feasible mechanisms to activate (by using inputs) or propagate (outputs) the effect of a fault. The results showed that 11.05% of the faults in the modules of the GPU can be considered as safe faults. It must be noted that we considered the gate level representation of the FlexGripPlus model during the safe fault analysis. At the end, The FlexGripPlus FMEDA comprises 92 Failure Modes mapped to 2,751,088 Gates, 1,507,085 Flops, and 784,224 Memory cells.

The analysis is performed employing a commercial tool in the FlexGripPlus model, considering the 15nm FinFET-based Open Cell Library and its internal technology and physical parameters, resulted in a total λ of 10.08 FIT; from these, the implemented safety strategies (SBST test programs) provides the following results:

- Detected by the test programs: 9.17 FIT
- Undetected (λR): 0.57 FIT
- Safe faults (λS): 0.33 FIT.

Finally, reducing λR by increasing λS and FC, directly impacts the SPFM. The proposed Safety technique based on only STLs for FlexGripPlus resulted in an SPFM of 94.27%, allowing ASIL B assessment without hardware modifications and without any other safety mechanism.

The previous evaluation and the results supports the idea that functional test strategies based on the SBST approach can be employed in GPU devices as a complementary technique to provide safety mechanisms into devices oriented to safety-critical applications. The ASIL B assessment proves that GPU devices can use STLs as an alternative approach to guarantee functional-safety features into these complex parallel accelerators with zero hardware cost and without changes in the original microarchitectural design of the GPU core. However, it must be noted that most test programs were developed at assembly level, so avoiding compiler optimizations.

4.5 Conclusions

This chapter described several SBST strategies targeting the development of in-field test procedures for critical modules in the GPU architecture and in some cases hidden to the programmer, such as the scheduler controller, pipeline registers and embedded memories. In the end, the overall evaluation is performed on the complete GPU showing that SBST approaches can be employed as a complementary strategy to support the in-field testing procedures in GPUs.

Regarding the SBST methods proposed for the scheduler controller, three incremental solutions were proposed and evaluated to detect faults inside the warp scheduler in the GPU. Moreover, the methods to adapt and use them in any application were also provided. The key idea behind the methods is their capability to generate divergence paths, during the thread execution, and use the performance variation among the threads and/or final results in global memory to detect permanent faults. Moreover, the strategy employs the signature per thread mechanisms, so each variation on the thread execution can be monitored and it is used to propagate the fault effect to any observation point. More in detail, results indicate that both method M2 and M3, see section 4.1.1, are promising SBST methods able to achieve high fault coverage. Especially, the M3 method requires only to check the final results in global memory after test program execution, which is a typical mechanism used in SBST techniques for processors.

A complementary work was developed and validated using real NVIDIA GPU platforms. In this case, a functional technique was proposed targeting the development of Self-test programs aiming to perform the on-line test of static and coupling faults in the scheduler memory of a GPU. This technique was developed and implemented considering the available micro-architectural information of a GPU and a high-level programming environment (CUDA). In the proposed method, the fault primitives for such a model were described and adapted considering the operational constraints of the target module. The results on some representative test cases showed that the proposed approach is effective and can test all the identified fault primitives, so validating and ensuring complete coverage of all static and coupling faults in the memory of the scheduler structure. It is worth noting that the validation procedure was performed using real GPU devices. Finally, optimizations to effectively implement the same kernel to describe a generic March operation targeting the memory within the scheduler are also presented.

Section 4.2.2 described a multi-kernel approach to develop self-test routines to test a hidden module for the programmers (the pipeline registers). The complexity of the parallel execution and the configurable capabilities of an SM core (each time a kernel is executed) were the key elements employed to develop the proposed test routines. Besides, the reported results assessed for the first time the Fault Coverage of SBST programs on the pipeline registers of a GPU core. A multi-kernel test approach composed of multiple SBST programs was proposed to test the control-path fields in the pipeline registers. Resorting to an RT-level fault simulation environment, we could compute the related FC, which amounts to more than 80%. The implementation of the test programs was based on a combination of high-level and low-level programming abstractions, developed through CUDA and manually included assembly (SASS) instructions. This work revealed and detailed multiple imposed compiler restrictions and constraints during test kernel implementation at a high-level. Each proposed SBST targeted different portions of the control-path fields in the pipeline register. An overlap of the proposed solutions can detect a considerable percentage of faults employing only memory results comparisons as the detection mechanism. Although the experiments target the pipeline registers, we claim that the same strategy can be applied to other modules in GPU devices.

Section 4.3.2 introduced a modular and scalable method to design functional programs for testing modules of GPU cores. The method was validated resorting to the small embedded memories in GPU cores, such as the address and predicate register files and the divergence stack. For this purpose, each target embedded memory was analyzed and based on controllability, observability and composition features. A set of parametric functions (called *blocks*) were developed and then combined to test each target structure in the GPU. Results show that the modular solution allows the exploration of the advantages and limitations of the implementation of different routines employed in a test program. Moreover, this technique also allows the split of a test program into several parts, while still achieving the same FC, so allowing to adjust the test program to potential requirements of in-field operations.

Finally, the adoption of automatic methods (pseudorandom-based and ATPG-based) in the GPU domain showed that it is possible to employ functional techniques from CPUs into GPUs. According to the results in the parallel functional units of the GPU, the percentage of detected faults by the designed test programs is high (about 85 to 90%), so these techniques can be employed as a complementary technique when testing GPU cores.

The overall evaluation of the SBST technique supports the idea that STLs are feasible alternatives for the in-field test of GPUs given their high fault coverage, low intrusiveness, and zero hardware cost. It must be noted that most test programs were designed exploiting the assembly language, so skipping the compiler optimizations. Similarly, the functional-safety analysis provided a high coverage of faults in the design. According to the results, the SBST strategy can be used as a complementary safety mechanism in devices aimed at safety-critical applications.

Chapter 5

Flexible hybrid mechanisms for in-field fault detection and mitigation in GPUs

This chapter describes three flexible hybrid mechanisms aiming at detecting and mitigating faults during the in-field operation of a GPU core. Each mechanism is composed of a structure added into the design of the GPU core that can be configured and enabled through custom instructions, added into the ISA of the GPU. The design, implementation and experimental validation were performed resorting to the FlexGripPlus GPU model.

Section 5.1 describes a flexible mechanism targeting the in-field fault detection of permanent faults by performing parallel operations and comparing the results searching for errors. On the other hand, sections 5.2 and 5.3 introduce hybrid structures to mitigate faults, and detect and mitigate faults, respectively. The proposed structure in section 5.2 exploits the adition of spare modules to replace and mitigate fault effects on the GPU. Finally, the proposed solution in section 5.3 describes a flexible structure integrating both functionalities (fault detection and mitigation).

In each section, the proposed structure, the implementation and main results, including resources overhead and reliability estimations, are presented.

5.1 DDWC: A Dynamic Hardware Mechanism for In-field Fault Detection in GPUs

This section describes a Dynamic Duplication With Comparison (DDWC) mechanism intended to harden the operational units (SPs) located in the SMs of a GPU. The proposed mechanism targets the detection of permanent faults that may arise inside the SPs. The main idea of the proposed strategy consists in the insertion of one additional functional unit in the GPU core, to compute redundantly the same operations of one selected SP. After each operation, the results are compared and possible failures are detected. A custom reconfiguration instruction allows the dynamic selection of the target SP to be monitored [180].

This work is motivated by the fact that new reliability challenges in modern devices may compromise the correct operation of an application and the system during the operative life, thus showing reduced long-term reliability. Moreover, due to the high density and complexity of GPUs, this work explores alternative solutions to face these reliability issues.

Currently, design and reliability teams face reliability challenges in GPUs by adding special structures, such as ECCs, to reduce the sensitivity to faults of some structures, such as memories resources (i.e., register file and cache memories) and communication interfaces. Nevertheless, other internal modules, such as the execution cores and the task controllers, cannot be easily protected with an acceptable design and production cost. Thus, these internal modules represent a challenge when dealing with their protection [181]. Moreover, traditional solutions to extend the reliability of control and execution units (EUs), such as dual-core lock-step and design diversity [182], increase the hardware overhead and the complexity of the device exponentially. Moreover, these solutions can also reduce the performance of the system by the included reliability structures.

In processor-based systems, there are several solutions to increase the in-field reliability, consisting in the adoption of classical strategies of Duplication with Comparison (DWC), Double and Triple Modular Redundancy (DMR, TMR). However, these strategies are mainly neglected (for in-field test) in GPU products due to economic and technical reasons. In fact, the architecture of the GPU implicitly includes (coarse-grain!) duplicated or triplicated structures, but with other purposes. In the GPU the duplication or multiplication of structures is fully devoted to exploit the parallel and performance operation of the device. Nevertheless, these classical solutions may be used in applications where safety is a major constraint. In these safety-critical fields, the additional cost in terms of design and production can be reasonably accepted due to the reliability benefits. To the best of our knowledge, there are no hardware solutions for in-field operation used in real GPU cores based on the previous techniques.

In the literature, several related works have been proposed in the past, exploiting DWC techniques as an error detection strategy for GPU devices. These works can be classified into two categories: *i)* software and *ii)* hardware. In both cases, the main target is to provide the detection or correction of faults. However, hardware approaches may be complex to develop and implement, although they are more efficient than software approaches.

On the one hand, software DWC mechanisms take advantage of time redundancy by repeatedly executing instructions [167] [168] [183], functions [69], or application tasks [69] [71] [146] on a GPU device. At the end, results are compared to detect faults. These mechanisms introduce zero hardware overhead but cause significant performance degradation, additional switching activity, and also a moderate overhead in the memory and similar resources. Furthermore, in [184], the authors proposed an interesting automatic multithreading environment to modify the parallel program for GPUs and duplicate the number of operations. This solution is efficient in the use of both spatial and time redundancy, but the performance degradation of the modified program directly depends on the behavior of the application.

On the other hand, hardware solutions exploit spatial redundancy and consist of adding dedicated hardware structures in the design to increase the reliability by allocating operations into redundant and independent modules. Results are then compared to detect possible faults. The main advantage of these mechanisms is a reduced performance degradation. However, the hardware overhead directly depends on the redundant modules to be added and modified. In [185], a mechanism is used to duplicate the internal cores in one SM of a GPU. Special structures are added in the input and output modules to process the data operands and the interconnections. Nevertheless, the mechanism was only evaluated resorting to a structural simulator; hence, the hardware overhead and the resulting performance degradation were not quantitatively assessed. In [144], the warped-Dual Modular Redundancy was proposed, which uses the free execution cores, inside the GPU core or SM, to provide a redundant mechanism and detect errors. The inactive threads are configured using a Register Forwarding Unit (RFU), which can be configured either as EUs or as comparators. The RFU uses the original structures to monitor and verify the cores using timing redundancy. Although this method was evaluated with a simulator, results showed that the associated hardware overhead might be significant, considering the RFU existing in each core. Moreover, this method can only be activated when intra-warp divergence conditions are present in the application, so reducing the scope of this solution.

In [186], several execution units are included in the design of a GPU and are used to replace by reconfiguration and fuse the faulty units after performing the end-of-manufacturing testing. Finally, In [187], the authors propose and addition of heterogeneous cores to detect faults in a multi-core processor when executing out-of-order operations.

5.1.1 Main idea of the DDWC mechanism

The proposed dynamic redundancy mechanism (DDWC) targets the detection of permanent faults and also those faults potentially causing mismatching by effect of a fault. The proposed mechanism combines the flexibility provided by the software to identify a target module, combined with the low-performance degradation and spatial redundancy obtained by the hardware structures, so providing an effective fault detection mechanism. DDWC is intended for the in-field test by slightly modifying the application code by just adding, at specific locations of the code, instructions to control the DDWC structure; so apart from the additional instructions, the DDWC mechanism is completely transparent to the programmer. This solution is intended to introduce small or null performance overhead and minimal modifications into the existing structures.

More in detail, DDWC is an adaptation of the classical DWC structure. This strategy uses a sphere of redundancy (see Fig. 5.1) replicating one or more modules to increase the fault detection capabilities in a system. The DDWC structure is mainly composed of an input selector module, a redundant or spare module (a copy of the target functional unit or SP core), an output selector module, and a comparator (to check the coherence of the results). Finally, one controller manages the input and output selectors to provide the data and feed the redundant and comparator modules through the selection performed by a controller. This selection is managed and activated by a custom instructions *(DDWC_i)*, so the configuration of the DDWC structure can be programmed and controlled from the application and programmer views. This strategy allows the enabling and disabling of DDWC when the GPU is in operation. In fact, the controller can be dynamically programmed through the *DDWC_i* instruction, which identifies the module to be monitored and activates the input data channel, using the input and output selector modules in the sphere of redundancy, see Fig. 5.1. The input and output selector modules are composed of crossbar or meta-crossbar structures. Some additional decoders and registers are also included in the DDWC structure. The flexibility of the added instructions allows the designer to choose a suitable trade-off between fault detection latency and performance overhead by selecting the frequency of execution of the DDWC_i instruction.

Once the custom instructions activates the DDWC mechanism, the controller selects one input data channel. Each data channel is statically associated to each target module, so it is possible to use the selector modules (input and output) to coordinate the assignation of the same data operands into the spare module. Then, all input operations are redundantly executed by both: the target module and the redundant one. This comparison and fault detection structure is well-known and contributes to reducing the performance latency for fault detection. The comparison of both obtained results (the target module and the redundant one) is the based to identify a fault in the system. This direct comparison also includes the evaluation of the output flags after each operation performed in the functional unit. A fault is detected if there is at least one mismatch in the results. The comparator generates an output error signal triggering the logic state that indicates the fault detection.

It should be noted that the sphere of redundancy can be applied at multiple levels of granularity in a GPU architecture. However, the DDWC structure could be unfeasible when applied to big modules, which might represent a considerable hardware overhead for a complex system, such as pipeline blocks in a processor or a GPU. Moreover, the non-regularity of some structures could limit the adaptation of the DDWC to keep low performance and hardware costs. It is worth noting that the DDWC structure focuses on the fault detection mechanism, so error handling and mitigation strategies are out of the scope of this section and will be discussed in section 5.3.

Initially, several modules of the GPU architecture were explored as candidates to implement the DDWC mechanism.

Firstly, the entire Execute pipeline, in the SM core of FlexGripPlus, was considered as priority target module for the sphere of redundancy in DDWC. However, as commented before, the non-regularity of its internal structures (the Execute module includes SPs, FP32s, SFUs and some controllers) were a strong factor to avoid that possibility by the limited adaptation and use of the main idea of the DDWC mechanism, so introducing unacceptable hardware overhead in the GPU core.

Other feasible candidates are the sub-module structures of the functional units in the GPU core. In principle, the adaptation of the DDWC mechanism requires the duplication of each sub-module, so basically duplicating hardware overhead for each functional unit considered. Finally, the functional units were evaluated. In this case, the high regularity of the functional units (specifically SPs) in the execute pipeline allows the adoption of the DDWC mechanism with a limited hardware overhead when sharing the same spare module, so it is possible to duplicate the SPs and perform the comparison on selected time intervals. This combination (regularity of the SPs and the reconfiguration mechanism) contributes to reducing the total overhead costs in the GPU core. In the end, it was proposed the addition of only one SP core, and a newly introduced custom instruction allows selecting a target SP with the comparison structure. Fig 5.1 summarizes a general scheme of the mechanism using SP cores as the main modules in the sphere of redundancy.

The input switch collects all the data channels (SPCs) and activates one of the data paths coming from the previous pipeline stage and feeds the redundant SP core. In contrast, the output switch collects all output data channels (SPDCs) for the active SPs, selects one among them, and feeds the comparator module. A switch controller is included to manage the channel selection in both switches (input and output). The signals decoded by the DDWC_i instruction are employed to reconfigure both switching structures.

The redundant SP is placed in parallel to the existing SP modules, and the inputs are directly connected to the input switching selector. Similarly, the outputs of the redundant SP are connected to one of the inputs of the comparator module.

The comparator module is a bit-wise comparator of the output results (SPDCs) coming from a target SP and the redundant SP. An output strobe flag is included as observability mechanism to inform the Host, or an exception handler in the GPU, about potential faults found in one of the SPs. A mismatch in the results indicates a fault in the SP, hence triggering the output flag.

Some additional combinational modules, such as the interconnections and the

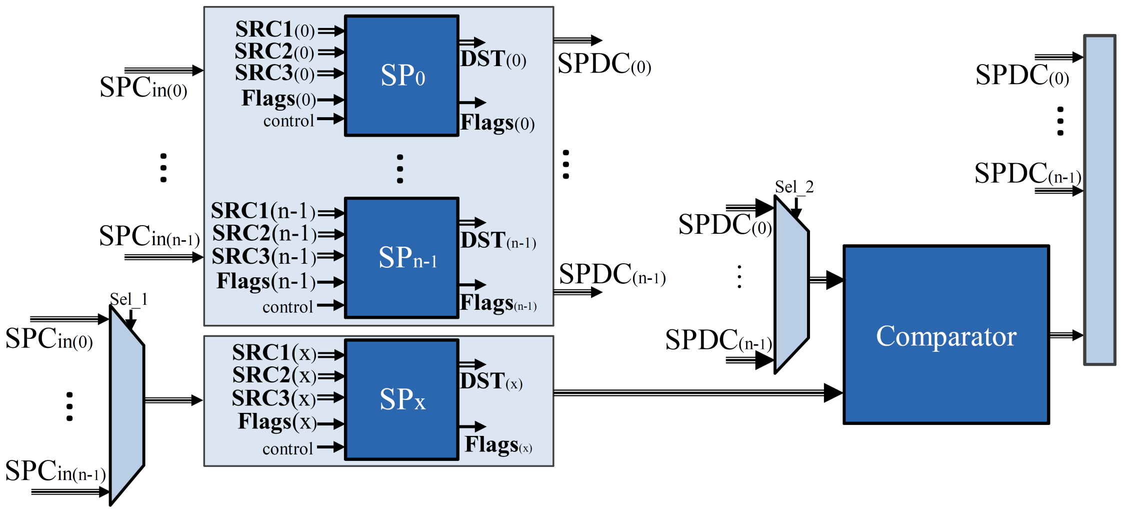


Figure 5.1: A general scheme of the proposed DDWC structure for in-field fault detection implemented into the FlexGripPlus model

decoding hardware, are included in the DDWC mechanism. Similarly, a configuration register holds the latest configuration in the DDWC structure after using the DDWC_i instruction. It is worth noting that in order to maintain the original structural configuration of the GPU core, the DDWC mechanism starts disabled (disable mode) after a reset or switch-off event and can be activated only through the DDWC_i instruction.

The explicit selection of a target SP core to perform the comparison with the DDWC_i instruction may affect the optimal fault detection performance of all SP cores in the sphere of redundancy. Moreover, the input operations (extracted from the instruction in the application) and the frequency of activation of the custom instruction can also limit the number of patterns applied to each core and the final capacity to identify errors. Thus, the operation of the DDWC mechanism requires a balance between the selection and activation of a target core and the insertion of the DDWC_i instruction in the application code to obtain the best fault detection rate performance.

It is worth noting that other modules in the GPU, such as memories and control units, were not targeted and are out of the scope of the proposed DDWC solution. The use of the DDWC mechanism in these modules may introduce hardware overheads greater than 100%, considering the duplication of modules and the required additional structures, such as registers, comparators, and interconnections.

5.1.2 Implementation and experimental evaluation

The FlexGripPlus model was used to implement and evaluate the fault detection capabilities of the DDWC mechanism experimentally. Three pipelines modules

(*Decode*, *Read*, and *Execute/Control-flow*) were modified to include the DDWC structures. It is worth noting that the selection of the SP cores, as sphere of redundancy, to implement the DDWC mechanism contributes to reduce the number of deep modifications in the original memory hierarchy and the warp scheduling of the GPU.

In the *Decode* stage, the DDWC_i instruction was implemented by carefully adding some combinational logic. The Instruction Set Architecture (ISA) in FlexGripPlus (SASS) was analyzed, and one available operation code was selected for its description. The format of the DDWC_i instruction includes 5 bits to select the SP module to be duplicated, which is directly related with the data channel feeding the module, 2 bits to enable or disable the DDWC module, and 6 bits stating the instruction type.

In the *Read* stage, a bypass structure was included. This bypass is composed of registers and keeps the pipeline coherence during the execution of the instruction by managing the operands and configuration signals in the DDWC structure to be used in the *Execute/Control-flow* stage.

The *Execute/Control-flow* stage was modified, adding a copy of an SP core (SPx), the input and output multiplexers (implementing the input and output selector switches), and a logic comparator, see Fig. 5.1. The two multiplexers are located in the inputs and outputs of the redundant SP core (SPx) to select the data channel to be duplicated.

The comparator is a fully combinational module, built with XOR gates. The output multiplexer selects and feeds one of the inputs of the comparator with results coming from SP cores. The other input is constantly connected to the outputs of the redundant core (SPx). The result of the comparator enables an output flag and it is propagated to the next stage. Then, this flag indicates the Host or a error handling module in the GPU that one error was detected. This flag is intended to activate an interruption/error-handling routine in the host and indicate the presence of a fault in the SP core. After a fault is detected, the controller also propagates one configuration code to the Host. This code can be used to locate a faulty SP core in the SM or in one available SM in a GPU by the Host.

One additional local controller, one decoder, and some registers were also included to configure both multiplexers. As previously mentioned, this configuration is based on the decoding signals from the DDWC_i instruction.

The input an output data-path interconnections (data channels) of each SP core were duplicated to feed the multiplexer units. On the other hand, the control-path fields of all SP cores in the GPU core were not duplicated, but shared when considering that The FlexGripPlus architecture employs the same instructions to be processed into the available SP cores, so all available cores remain with the same configuration during the execution of one instruction.

Under the inactive mode of the DDWC mechanism, both multiplexers are unconnected from the redundant SPx core and the comparator, so avoiding unnecessary switching activity in both modules. In this way, the dynamic power is reduced during inactivity periods.

Regarding application code modifications, the code requires a minor modification to use the DDWC mechanism for in-field detection. According to the intended functionality, it is suggested to include the DDWC_i instruction at the beginning of the application code to reduce performance degradation. Similarly, the DDWC_i instruction can be placed in strategic locations in the code (i.e., before any logic-arithmetic instruction) or in a periodical manner (after a given number of instructions). However, in both cases, the performance degradation for the application must be considered. In any case, during the in-field execution, the output flag signal in the comparator is directly assigned as a source of fault identification. The implementation of the DDWC_i instructions supports two formats, allowing the selection of the target SP core to be duplicated from an immediate source of from a register source.

It is worth noting that the DDWC mechanism is intended to be used during the in-field operation of the GPU. Once DDWC is active, the redundant core should swap among the SPs of the SM. For this purpose, the program code is modified by adding some DDWC_i instructions. Depending on the application and on the selected frequency for the swap procedure, the locations where the DDWC_i instructions are inserted can be suitably selected. A higher rate reduces the fault detection latency but increases the performance overhead stemming from the DDWC_i instruction addition. The method is flexible since it allows adopting the solution which best fits the target system specifications.

5.1.3 Experimental Results

The experiments were performed on the SM core using three SP configurations (8, 16 and 32 SP cores). This hardware flexibility allow the analysis and evaluation of the proposed DDWC mechanism in the SM core with multiple dimensions.

An extensive group of simulations were performed to verify the correct operation of the proposed structure.

Figure 5.2 depicts the changes in the code to activate and swap the DDWC structure during the execution. The Fig. 5.2(left) shows the case of activating the DDWC mechanism before to start the execution of a part of code, using a static selection. On the other hand, the example of Fig.5.2(Right) shows a modified code using a dynamic selection of channels using a register as source.

A preliminary observation shows that any application in a GPU core, integrated with the DDWC mechanism, uses the DDWC structure four times and twice per SP core when the GPU core is configured with 8 and 16 SP cores configuration, respectively. This behavior is explained considering that the thread management

```
...
DDWC CH0
MOV.U16 R0H, g [0x1].U16;
I2I.U32.U16 R1, R0L;
IMAD.U16 R0, g [0x6].U16, R0H, R1;
SHL R2, R0, 0x2;
IADD32 R0, g [0x4], R2;
IADD32 R3, g [0x5], R2;
GLD.U32 R1, global14[R0];
GLD.U32 R0, global14[R3];
IADD32 R1, R1, R0;
IADD32 R0, g [0x6], R2;
GST.U32 global14[R0], R1;
...
```

```
...
LC:
MOV R1, R124;
SHL R2, R2, 0x1;
...
SSY 0xe8;
BRA C0.NE, 0xe8;
SHL R4, R4, 0x2;
IADD R4, g [0x4], R4;
IADD32I R6, R4, 0x4;
GLD.U32 R6, global14[R6];
ISET.S32.C0 o[0x7f], R5, R6, LE;
GST.U32 global14[R4] (C0.EQU), R6;
IADD R4 (C0.EQU), R4, c[0x1][0x0];
...
I2I.S32.S32 R1, -R1;
DDWC R1;
BRA C0.NE, LC;
...
```

Figure 5.2: Examples of code modifications when using the DDWC mechanism. The static selection (left) maintains configuration according to the constant channel CH0. The dynamic selection (right) employs a general purpose register to address any channel, so allowing the online configuration of the DDWC mechanism.

in a SM core when the number of SP cores is lower than the number of threads in a warp, so the threads are divided in groups and submitted to the available SPs (and the spare SP in DDWC). Thus, for an 8 SPs configuration, four groups of threads share the same SP (and the spare SP if DDWC is active). Similarly, for a 16 SPs configuration, the controller in the SM dispatch two threads to each SP serially. It means that for every configuration of the DDWC structure, the comparisons, and possible fault detection are performed four times and twice with instructions belonging to the same warp for the 8 and 16 SP cores configurations. In contrast, a configuration of 32 SP cores in the SM has the equal number of threads in a warp, so the instruction per warp is only executed ones on a SP (and the spare SP core for comparison).

The experimental performance results were gathered using a gate-level netlist version of the FlexGripPlus model integrating the DDWC mechanism. The Synopsis toolchain *(Design Vision)* was used to estimate the hardware overhead and performance degradation. The NAND-Open-cell library for 15nm [100] and 45nm [99] were used for synthesis purposes.

Both GPU models (the original one and the one integrated with DDWC) were compared for all configurations of SP cores. Results in terms of size of cells are reported for the modified modules during the integration and also for the entire design in Table 5.1. The reported results do not considered the hardware cost of the memory elements in the SM of the GPU core.

The reported results show that hardware overhead of the DDWC mechanism is relatively low. In the one hand, in the *Decode* pipeline stage of the GPU core, the total overhead of the implementation of the controlling instruction represents only 3%. Moreover, the hardware cost seems to be insignificant (≈0.1%) for the *Read* pipeline stage. In contrast, in the *Execute/Control-flow* pipeline module, the overhead is inversely proportional to the number of SPs in the SM. Hence, a lower number of SPs introduce a higher overhead cost. However, this overhead seems to be moderate (≈3.5-10%). On the other hand, the analysis of the overhead cost in the entire design shows that it is lower than 3% for all SPs configurations. These results support the initial goal of limiting the impact in the hardware overhead by the DDWC mechanism.

The above results support the claim that the proposed DDWC mechanism represents a practical solution as a fault detection strategy without including a critical impact in the area of the GPU.

Table 5.1: Hardware and performance overhead of the DDWC mechanism for multiple configurations in the GPU core using the 45nm technology library.

Modules	**SP cores**	**Number of Cells**		**Area overhead (%)**	**Time delay in the critical path (pS)**		**Performance degradation (%)**
		FlexGripPlus	**FlexGripPlus + DDWC**		**FlexGripPlus**	**FlexGripPlus + DDWC**	
Decode	8/16/32	1,229	1,266	3.04	1.72	1.77	1.16
Read	8/16/32	142,397	142,545	0.10	3.65	3.65	0.0
Execute	8	60,309	65,959	9.37	6.51	7.0	7.52
	16	113,293	118,739	4.81	6.69	7.68	14.79
	32	219,261	226,822	3.45	7.52	8.54	13.56
All	8	229,515	235,964	2.81	11.88	12.42	4.54
	16	280,132	286,360	2.22	12.06	13.10	8.62
	32	386,100	394,516	2.18	12.89	13.96	8.30

Performance overhead

The performance overhead was evaluated at the module and design levels. Initially, the critical path delay was determined for each modified module and the whole design. It is worth noting that in both cases, the synthesis and analysis were performed without adding constraints or optimizations in the synthesis tool. A frequency of 100MHz was selected for the synthesis. The results are presented in Table 5.1.

By looking at the results, it can be noted that the additional structures in the *Decode* module added a small percentage of performance degradation (1.16%). In the *Read* module, a bypass register was added. Nevertheless, this does not introduce any overhead. This behavior can be explained by observing that the bypass register was concatenated with existing structures in the module, thus guaranteeing the same functionality of the original structures. In contrast, the performance degradation in the *Execute/Control-flow* module seems to be directly affected by the number of SP cores. For a low number of SP cores, the critical delay path is increased by 7.52%. The configuration of 16 SP cores seems to present the maximum percentage of performance overhead with 14.79%. In contrast, the overhead drops for the 32 SP cores. Although the total delay overhead is higher for 32 SPs, the delay overhead is lower. This behavior can be explained considering that added SPs are placed parallel in the design, thus adding a low timing overhead. However, the most representative timing effects are due to the added modules in the path. These modules are the input and output switches and the comparator.

Power consumption overhead

The consumption overhead for the DDWC strategy was determined after analyzing the initial description of the GPU core and the GPU containing the DDWC structure. Each affected module of the GPU core to implement the DDWC (*Decode*, *Read*, and *Execute*) was analyzed in terms of the individual effect of the added structures. The analyzes were performed on the synthesized version of both GPUs (the initial and holding DDWC).

In results, The power overhead produced by the DDWC in the *Decode* and *Read* modules is negligible ($< 0.001\%$). In contrast, the overhead in the *Execute* module directly depends on the available SP cores, since the power consumption increases according to the available SP cores in the design (73.6% and 222.45% of power consumption incremented for 16 and 32 SP cores, respectively, concerning the 8 SP cores version of the module). Interestingly, the added spare cores to perform the comparison remain as cold spare modules when inactive, so the additional power consumption is only present in the active mode of the DDWC structure. Once the DDWC is in active state, the increment in the power consumption raises to 18.68% for a GPU with only 8 SP cores. This power overhead is reduced to 5.03% employing a GPU with 32 SP cores. Finally, analyzing the complete GPU, the power overhead is increased by up to 2.65% for the minimum number of SP cores (8). Meanwhile, the power costs are reduced up to 1.1% in a GPU with 32 SP cores.

Fault Detection Capabilities

The fault injection environment is based on the *ModelSim* framework, and the injection methodology used is similar to the introduced in [167] [173]. Further details regarding the descriptions and configurations of the used benchmarks can be found in Chapter 3.1.2. For the experiments, the output flag from the comparator was included as an observability mechanism to detect faults in the simulation environment. It is worth noting that in both versions of the GPU (original and the one including DDWC) the main outputs of the GPU (memory and control signals)

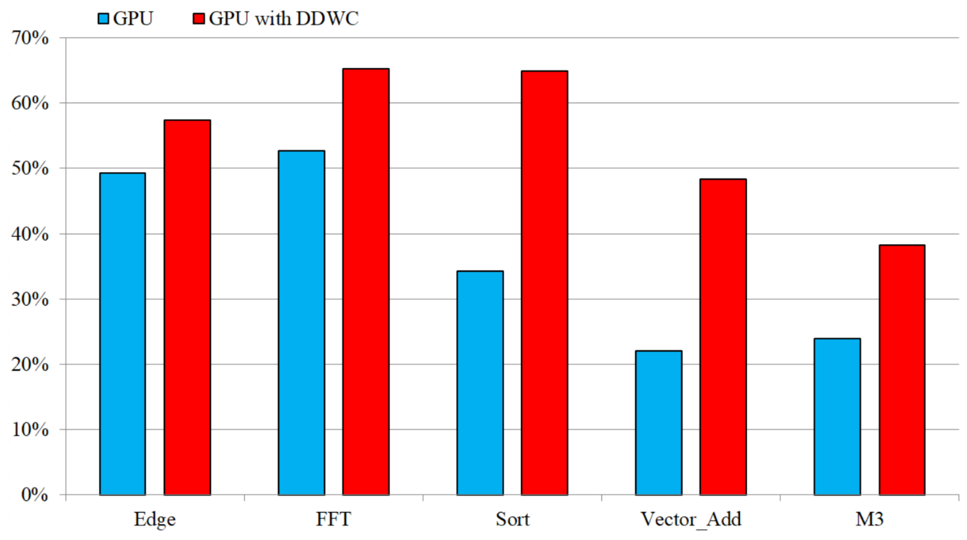


Figure 5.3: Fault coverage results for several applications when enabling the DDWC structure in a GPU.

were used as observation points to detect fault effects during the fault campaigns.

A set of benchmarks with different workloads was employed to validate and evaluate the detection capabilities of the DDWC strategy experimentally. The fault injection campaigns were performed in the FlexGripPlus with and without the DDWC mechanism and considering single stuck-at faults at the RT level. On each fault campaign, the GPU model was configured with 8 SPs and 32 threads per block. Each benchmark was adapted to include the DDWC_i instruction activating the DDWC mechanism. Gathered results were combined to mimic the long-term operation of the GPU with the DDWC mechanism rotating among the SPs.

Figure 5.3 reports the Fault coverage results for five applications evaluated in the original GPU and one implementing and enabling the DDWC mechanism. In the experiments, 8 SP cores are configured in the SM and one SP core was permanently active to perform comparisons.

The results showed that the increment in the detection of fault, reflected in the fault coverage of the analyzed applications, varies in the range from 8% to 30%. However, results support the idea of the increment in fault detection capabilities when adding the DDWC mechanism in the GPU core. The final Fault Coverage that can be achieved on each SP resorting to the proposed method strongly depends on the application. In fact, the observed behavior for each application directly depends on the number of instructions using the SP core during the execution of the program. Moreover, it should be considered that the DDWC strategy requires the explicit selection of a target SP core to perform fault detection. Thus, a balance between the frequency of the SP switching and the application features is required to obtain optimal in-field fault detection.

Estimation of the fault detection time

The proposed DDWC mechanism increases the fault detection capabilities concerning faults in the target structures. However, as the DDWC structure is intended to select the SP to be monitored during in-field execution dynamically, the overall fault detection capabilities of the DDWC strategy depends on several parameters, such as the switching frequency and detection time, and the detection capabilities of the operations (or test patterns). A test pattern is one or a sequence of values applied to the inputs of the target SP to excite a fault and propagate the error to the outputs.

Considering a fault-free DDWC structure and an SM composed of n SPs, the SM in principle cannot detect permanent faults in the SPs. Hence, fault detection capabilities are zero. Using the DDWC mechanism, the fault detection capabilities in the system increase. This increase can be estimated by resorting to the relation between the fault observability and the required time to detect a fault. The fault observability (Ob)[188] in one SP can be defined as:

$$Ob_{SP(p))} = \left[\frac{N_p}{N_{np}}\right] \tag{5.1}$$

where N_p and N_{np} are the number of input patterns that propagate and do not propagate a fault effect to the output, respectively, and $P = N_p + N_{np}$ is the number of patterns, assuming that the average observability remains constant across the time. In the DDWC mechanism, the patterns are mainly generated by the execution of instructions and data operands.

The time for detecting a fault (tt) is composed of a set of time intervals needed to perform the fault detection in an SP. If a fault arises in the system at time $t = 0$, the fault is excited by a pattern after a time t_1. Then, it is propagated to the output after a time t_2, and finally, it is detected after a time t_3. Thus, it is possible to estimate the time for fault detection ($ETFD$) in a continuous fault detection structure case as:

$$ETFD_{DWC(t,p)} = \frac{[N_p + N_{np}] \cdot tt}{N_p}; tt = t_1 + t_2 + t_3 \tag{5.2}$$

In Equation 5.2, short times are desirable to perform fault detection. However, the configuration of the DDWC strategy directly depends on the time interval employed to switch among SP cores t_4 and the required time to execute the configuration instruction t_5. Thus, it is possible to express the time for fault detection (ETFD) in the SPs using the DDWC mechanisms as:

$$ETFD_{GPU(tt,p)} = \frac{[N_p + N_{np}] \cdot [tt + \sum_{i=1}^{n} (t_4 + t_5)]}{N_p} \tag{5.3}$$

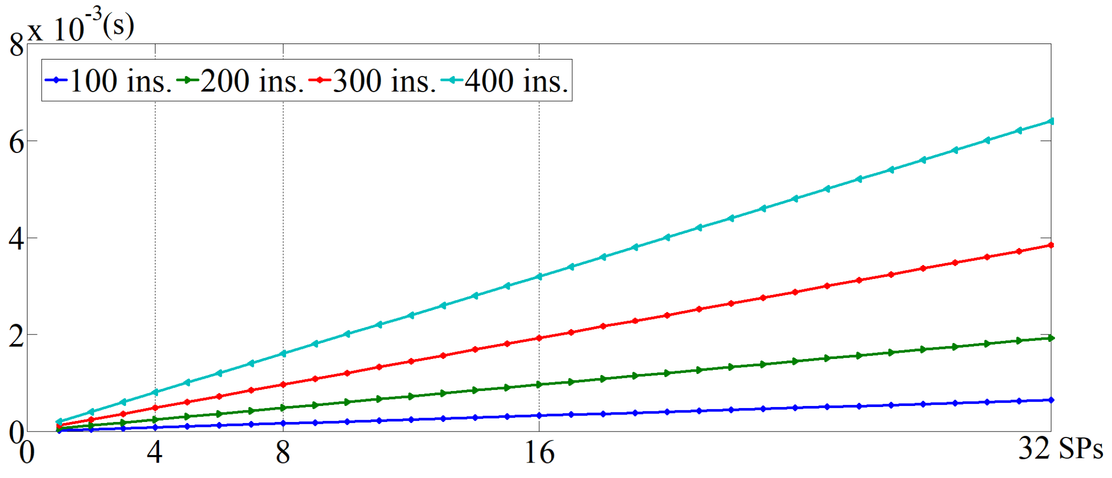


Figure 5.4: The maximum value of ETFD for multiple frequencies of the DDWC_i instruction (100, 200, 300, and 400 instructions) in the application code and under multiple SPs configurations in the SM.

where **n** is the number of SPs in the SM, and time t_5 is proportional to the number of instructions between two sequential configuration instructions.

Figure 5.4 represents the worst-case scenario for fault detection in an SP, assuming that the switching among the available SPs is performed every 100 instructions in the program code. The expression is calculated using a clock period of 10 ns, an average time for instruction execution of 20 clock cycles, and times for pattern injection, fault propagation and fault detection at a fixed rate of 3 clock cycles. It is assumed that the fault can be propagated to the output by one of the test patterns.

Equation 5.3 can be used to find an optimal trade-off between the switching frequency for the DDWC mechanism among the SPs and the performance degradation in the application by the insertion of DDWC_i instructions.

Comparison with other fault detection strategies

A comparison of the proposed mechanism with classical fault detection methods, such as lock-step and Build-In Self-Test (BIST), shows that the lock-step structure requires the duplication of each structure in the design. Thus, the hardware overhead overcomes 100%, considering the additional structures. On the other hand, a BIST mechanism can be an effective solution for the end-of-production test. Nevertheless, the same approach could be challenging to use with the SPs during the operational phase of a GPU. These difficulties are based on the required execution time to perform fault detection. Moreover, the required structures for implementing such a solution may be equal to or greater than the DDWC strategy in terms of hardware overhead.

In general, testing solutions based on SBST mechanisms add zero hardware cost to the system. However, the latency required to perform the fault detection might compromise the performance of an application. In contrast, the DDWC mechanism faces those latency issues by reducing the fault detection time as a negligible effect in the execution of instructions, so the performance degradation is not included as part of the instruction's cycle. Moreover, a big advantage is that the DDWC mechanism can be combined with SBST techniques to increase the fault observability.

The previous analysis proves that the DDWC mechanism is an alternative solution for in-field fault detection, equivalent to lock-step and BIST, but DDWC has lower cost in terms of hardware overhead when implemented in the SP cores of GPUs. Moreover, the proposed mechanism may also coexist with other structures to detect, mitigate, or repair faults in the SP cores.

5.2 BISR: a dynamic reconfiguration mechanism to increase the reliability of GPUs

This subsection introduces a dynamic reconfiguration mechanism aiming at the mitigation of faults in a GPU core. For this purpose, the Built-In Self-Repair mechanism was adapted and employed to protect the execution units in a GPU.

In principle, many commercial GPUs include fault mitigation mechanisms for memories based on ECCs. In most industrial applications, these ECCs are enough to provide the required reliability. In the worst case, when the GPU stops working, it may be replaced. A different scenario exists in the automotive industry. For this kind of functional-safety applications, the effects of a fault may cause unacceptable operational failures. Thus, GPUs may require complementary fault mitigation solutions to be applied during real-time operation.

As introduced before, traditional solutions to increase the reliability of digital design are based on hardware, software, and hybrid approaches. The hardware mechanisms are considered as feasible solutions in applications with strong requirements in terms of functional safety and reliability, such as the automotive one. In this case, an additional cost can be justified by the improved features and capabilities. Hardware solutions include Duplication with Comparison (DWC), Double and Triple Modular Redundancy (DMR, TMR), ECC and the hardening of selective logic gates. The adoption of these solutions requires a careful evaluation of the involved area and power budget overhead. Moreover, some of these techniques are mainly intended to mitigate the effects of transient faults in a system. In contrast, mitigation of permanent faults requires strategies, such as Built-In Self-Repair (BISR), replacing a faulty block with a spare one.

In BISR, the granularity of the block depends on the target module, and the complexity and criticality of the device [189]. In the past, BISR has been successfully applied in the memory blocks of processor-based systems by adding spare rows, columns, and additional controller structures to correct faults during the production phase and also during in-field execution [190][191]. Other works [192][193][194] targeted data-path units, such as the register file, and some internal components of the execution units (EUs) [195]. Similarly, some works proposed reconfiguration solutions targeting computational blocks in GPUs [196] or other modules in the GPU, such as the memories [197], and functional units [198], or combinations of both aligning the system to the specific workload requirements [199][200]. In [186], the author proposes selective hardware redundancy to correct defective blocks during the production stage. This method is intended to increase the production yield during manufacturing.

Other works [167][201][202] introduced mitigation strategies only based on software mechanisms. These solutions are effective in detecting most faults and tolerating a high percentage of them. Moreover, the added area overhead is zero. Nevertheless, their cost in terms of performance degradation and memory overhead may be relevant due to these solutions are implemented by instrumenting the application code with custom functions. Considering the previous works, the combination of software mechanisms and additional hardware modules in a hybrid structure to achieve the same results may be attractive.

This subsection introduces a BISR strategy mainly aiming at addressing permanent faults (or any other type of fault that can be observed) during the in-field operation in the SP cores (Stream/Scalar Processors) inside the SM of a GPU. This BISR strategy leverages on the high regularity of the SPs in the GPU architecture. We leverage the techniques recently described in chapter 4, which allow the detection of permanent faults in the SP cores of a GPU resorting to software self-test procedures. Similarly, we employed suggestions and guidelines proposed in [82].

The basic idea behind the BISR mechanism lies in introducing a given number of Spare SP (SSP) cores, which may substitute any faulty SP as soon as a permanent fault affecting it is detected. In our proposal, the reconfiguration can be activated via software with an additional instruction (*Config_SPs*), which has been purposely introduced in the GPU Instruction Set. Moreover, this instruction is compatible with the original programming language of the GPU. Apart from the execution of *Config_SPs* instruction, the mechanism is completely transparent to the programmer. The method only allows tolerating faults affecting SP cores, which correspond to a significant fraction of the total SM area.

The BISR mechanism is intended to add as minimal as possible structural changes into the existing hardware of the GPU, and thus on its performance. Finally, the proposed solution does not require any change in the application code. The proposed BISR mechanism (together with the related test) can be activated during power-on or at reset when timing constraints for fault detection and hardware reconfiguration are not so relevant.

The proposed BISR solution has been implemented and evaluated resorting to an extended version of the FlexGripPlus model. Extensive experimental results showing its cost and effectiveness have been gathered referring to that model.

Although the usage of spare units is a well-known solution in dependable architectures, to the best of our knowledge this is the first work proposing its adoption at the SP level in a GPU core, and exploring in a comprehensive way the costs/benefits of its integration in a hardware model representing a real GPU.

The next subsection describes the proposed fault mitigation mechanism, as well as a summary of the software test mechanisms which can be used to detect faults in the SP cores and how they can be integrated in the proposed solution. Then, the implementation of the proposed BISR mechanism is described, using the FlexGripPlus model. Finally, the experimental results, analyzes and main conclusions are reported.

5.2.1 BISR mechanism in GPU cores

Given the complexity of a GPU, different mitigation methods should be used addressing the different composing parts. This subsection proposes a fault mitigation strategy targeting the SPs in the *Execute/Control-flow* stage of a GPU. This method aims at increasing the reliability of this stage by disabling an SP once it has been labeled as faulty due to a permanent fault, and substituting it with a Spare SP core (SSP). The solution is based on a hybrid approach [203].

The hybrid approach combines some mechanism to detect permanent faults in the SPs, based for example, on Design for Testability (DfT) or SBST test programs. For instance, in [173] and in Chapter 4, it was showed that suitable test programs can detect a high percentage of permanent faults within a single module. Once a faulty SP has been identified, a re-configuration process is launched. This process executes an ad-hoc instruction, which replaces the faulty SP by a spare one. For the purpose of this mechanism, the fault detection and localization phases were not targeted, but it is focus on the hardware changes to be introduced to support the reconfiguration phase (see Fig. 5.5). It is worth noting that in this work, it was not considered fault administration structures (FAS) to be activated after a device shut-down to recover the previous configuration state in the device. These FAS could be composed of flash memories and controllers to store the state of SPs and SSPs in the device.

Fault mitigation architecture

The BISR architecture is based on adding a given number of SSP modules in parallel to the existing SPs. These SSPs are cold standby modules, thus reducing the power consumption during inactivity. Two switching modules are added to control the data-path signals in the existing SP and SSPs. The switching units

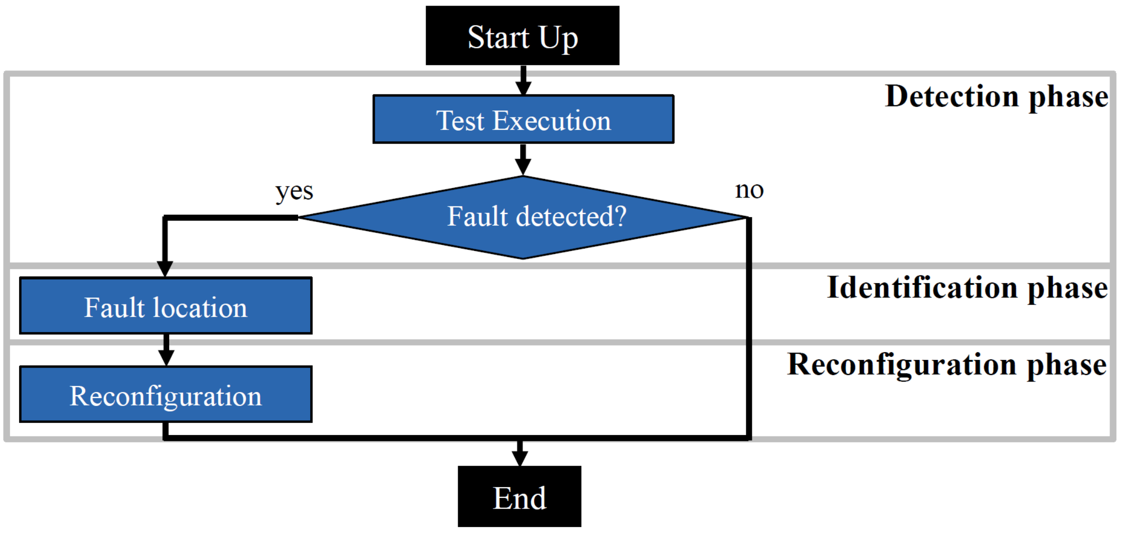


Figure 5.5: A general scheme of the detection, identification, and configuration approach for fault mitigation.

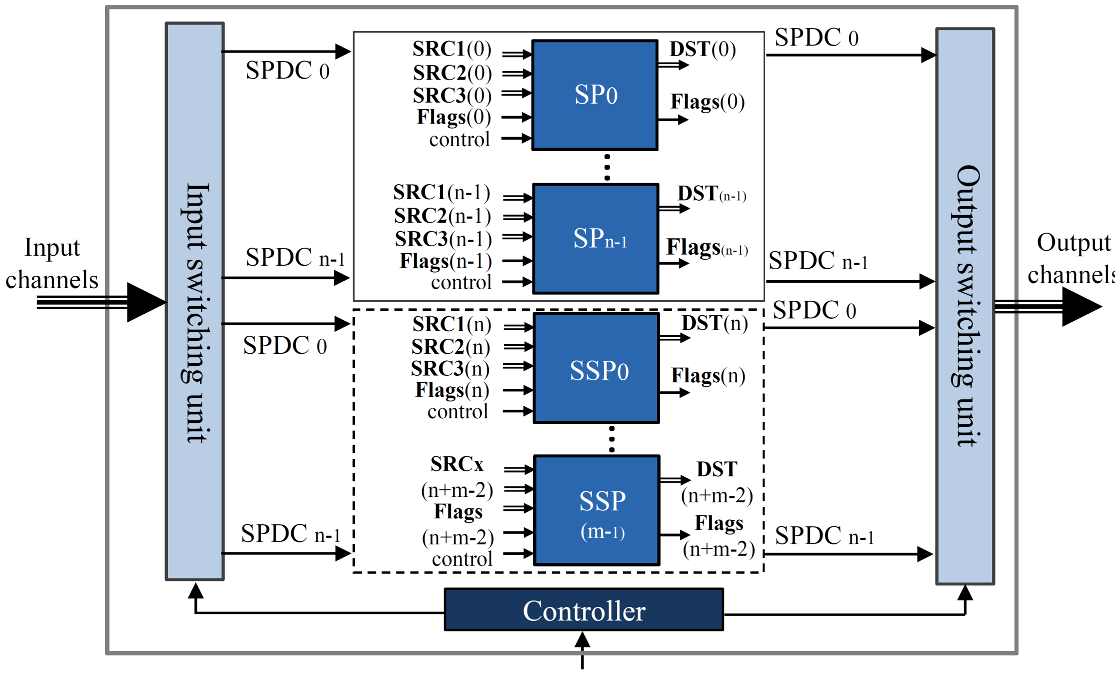


Figure 5.6: A general scheme of the adopted circular switching method for SP core configuration.

are placed in the Execution/Control-flow stage of the GPU following the Circular Switching Scheme, see Fig. 5.6. This Circular Switching Scheme is based on managing the access to data and control channels of the modules to possibly repair (SPs). The two switching units manage the access to the channels for each module (SP), so enabling or disabling the operation of any selected module in the device. These switching units are mainly composed of meta-crossbar structures. A controller manages the switching units and enables or disables the operation of SPs and SSPs in the system. Fig. 5.6 shows the general scheme of the configuration structure composed of **n** SP cores and **m** SSPs added and connected to the SM. It is worth noting that the number of input and output signals on each SP is different. Thus, the size of the input and output switches may differ. Nevertheless, the same mechanism is employed to manage each SP core.

In order to minimize the impact of the changes on the existing architecture, we preserved the original memory hierarchy and WSC modules. Thus, each input Thread Data Channel (ThDC) is kept at the input of the execution stage.

The input switching unit includes, as outputs, the additional SP-data channels (SPDCs) and also connects them to the SPs and SSPs. In this way, each input ThDC is connected with one SPDC in the configurable scheme. The output SPDCs, coming from the SPs and SSPs, are connected with the output switch unit and the original output ThDC of the execute pipeline stage.

The output switch reduces the total number of data-path channels in order to keep the same pipeline interconnections. In the adapted mechanism the input switch behaves as a data channel de-multiplexer. Similarly, the output switch acts as a data channel multiplexer.

The placement of the two switching modules at the input and output of the SP cores contributes to maintaining the original memory hierarchy for each thread execution without relevant changes in the design. This is achieved considering that each ThDC does not include information related to the direct association of a ThDC to a specific SP core to perform operations. Thus, each thread process uses the original registers and memory locations, even when a spare core is active. Hence, the proposed solution is entirely transparent to the software, which must not be modified in any way.

The solution we followed does not impact the WSC existing in the SM. The WSC traces the execution and the state of each thread in a warp, but it does not include information related to the SP core allocation, thus remaining without changes.

The switch controller manages the configuration of both switching units using the same input control signals. The custom *Config_SPs* instruction generates the input control signals and activates the re-configuration of the SPs.

More in detail, the instruction selects one SP and one SSP core and forces the GPU to substitute the former with the latter for all the following activities. In the current version, the selected configuration is not saved anywhere: hence a test and possible re-configuration should be performed at each power-on or reset. The introduction of a small Non Volatile Memory could allow storing the configuration.

The format of the instruction is selected avoiding any overlapping with the original instruction set of the GPU. The instruction format is divided into two parts. The first part of the instruction picks and enables one of the available SSPs in the system. The second part manages the switching units by selecting the correct input and output data channels for each SP and SSP core.

Fault detection, fault identification, and reconfiguration

The BISR mechanism is intended to operate during the Power-on (or reset) conditions, so it is assumed that the test and possible reconfiguration steps are both performed during this initial configuration stage of the device. Moreover, the fault detection and location phases, as well as the reconfiguration one, can be executed without any strict time and memory constraints.

For the sake of completeness, we summarize here how the fault detection and location phases could be implemented. More details about this solution can be found in Chapter 4. However, other solutions (e.g., based on DfT) could be used as well.

At the power-on, the BISR structure is inactive. Thus the SP cores are initially connected with each ThDC and the SSP cores remain in cold standby mode. Then, a set of test patterns is applied to the SPs. These patterns are based on the execution of well-defined operations to test in parallel each SP in the SM.

The strategy employed is based on targeting all sub-modules in the SP cores and forcing them to execute suitable test patterns using instructions. The partial thread results are stored in the global memory for later analysis. As each thread executes the same instructions on different SP cores, a Signature-per-Thread (SpT) mechanism is used. Every SpT is compared with a set of previously stored results, and depending on the comparison each SP core can be labeled as faulty. This method allows for quick identification of the faulty SP.

Assuming that **n** SPs and **m** SSPs are available, the test is then repeated after reconfiguring the GPU so that **m** SP cores are substituted with **m** available SSP cores. At the end of this phase, a full map of the faulty and fault-free cores is available. Based on this map, if at least **n** cores are available, the GPU can be reconfigured accordingly and can continue working correctly.

For the sake of simplicity and to avoid reconfiguring the code, It is assumed that the minimum number of cumulative fault-free SPs and SSPs in the SM is **n**. This value is compared each time to verify when the SM cannot operate anymore.

The above procedure assumes that the system does not include any Non Volatile Memory (NVM). Hence, at each power-on, a complete test is required to build the map of faulty/fault-free cores. If an NVM is available, this map can be stored there and used to reconfigure the GPU accordingly at each power-on. The map is then updated with a given frequency, depending on the reliability targets and scenario parameters.

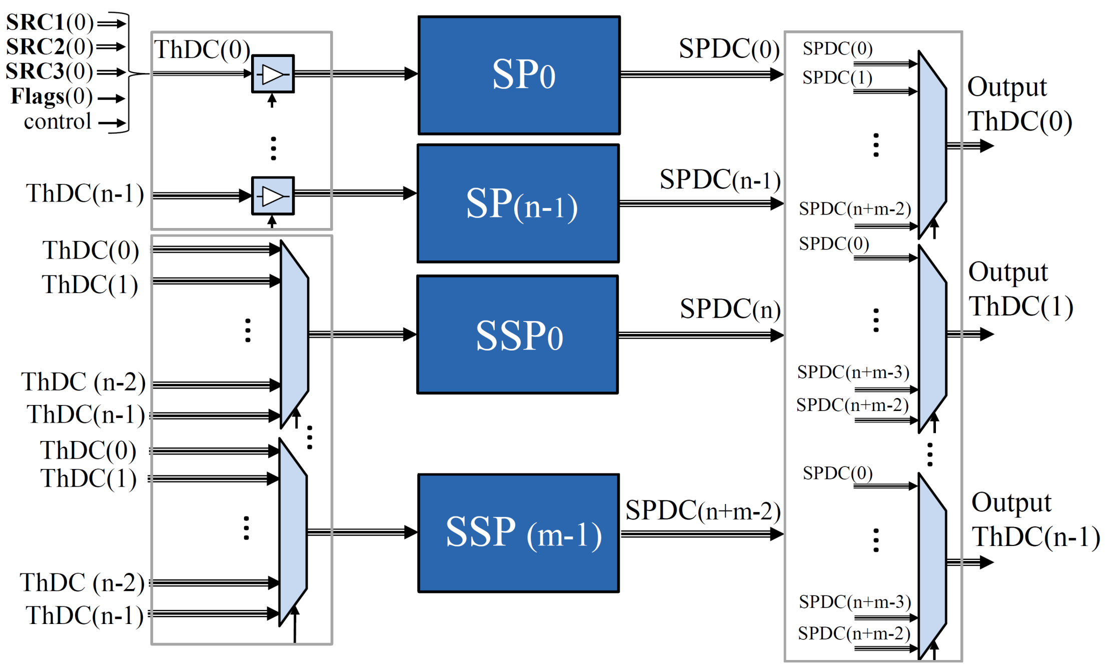


Figure 5.7: A general scheme of the implemented structure in the FlexGripPlus model.

Implementation

The FlexGripPlus GPU model was used to experimentally evaluate the performance of the proposed BISR fault mitigation mechanism. This mechanism is implemented in the *Execute/Control-flow* pipeline stage of the GPU. Nevertheless, additional changes were made in the Decode and Read stages by the introduction of the configuration instruction. The Decode stage was modified by adding new combinational logic to decode the added instruction.

The *Read* stage includes a bypass register to keep the pipeline coherence during instruction execution and also to store the configuration information for the SP cores. This information is then decoded and employed in the *Execute/Control-flow* stage. In the *Execute/Control-flow* stage, the description of the switching structures is the same for the input and output switches: both switching units were designed using the same basic modules, i.e., multiplexer blocks and bypass register structures (see Fig. 5.7). Moreover, an automatic generation mechanism is used to interconnect the SPs and busses with the input and output of the multiplexers in the same stage. Additional decoding, concatenating and de-concatenating blocks control the activation of the module and reduce the number of multiplexers in the system. In this way, a big multiplexer module is attached to each additional SSP. Similarly, the output switch uses multiplexers to interconnect with the output ThDCs. The ThDCs are selectable depending on the number of SPs in FlexGripPlus (8, 16 or

32).

Both switches (input and output) are activated under the same control condition, thus reusing the controller structure. This switching controller includes additional registers, to store the configuration information, and decoders to reduce the total number of control bits. This controller information is used to design the instruction op-code composed of eleven active bits fields. Moreover, these registers maintain the inactive SSPs and SPs in a cold standby mode, thus avoiding any unnecessary switching activity and reducing the power consumption.

The management of the information flow (ThDCs and SPDCs), allows the usage of the same registers and memory locations employed by the SPs and the replacing SSPs, when active. In this way, the memory is virtualized from the mitigation modules. Instead of a restriction, this condition was exploited to add the BISR infrastructures in the GPU by employing the same memory hierarchy modules. The control-path lines on each SP core are not considered as inputs in the switches due to these interconnections are shared on all SP cores and can be directly assigned on the SSPs.

The capabilities and flexibility of the FlexGripPlus model for selecting the number of SPs in the SM are also employed in the description of the mitigation modules. The same code style is used to describe the BISR modules. These structures are parametrically generated depending on the total number of SP and SSP cores aiming to reduce the hardware overhead in the system among configurations.

5.2.2 Experimental Results

The FlexGripPlus model was configured with one SM core in 3 modes (8, 16, and 32 SPs), so it is possible to analyze the benefits and limitations of the proposed BISR strategy with different numbers of SPs in the device. Firstly, we ran extensive simulations to validate the correctness of the proposed BISR strategy. Moreover, The quantitative cost and the benefits of the strategy were evaluated. For each considered case, the number of introduced SSPs ranged from 0 to 7. The analyses were performed resorting to a gate-level version of the FlexGripPlus model. The estimation of the hardware overhead was done resorting to the Design Vision tool by *Synopsys* using the ultra-compiler configuration. The 45 nm NanGate Open-cell library was employed for the experiments [99].

Hardware Overhead

The modules modified for implementing the BISR strategy are the *Decode*, *Read*, and *Execute* stages. These modules were modified at the RT level. Then, the GPU model was synthesized at gate level and compared in size with respect to the original design. Table 5.2 reports the hardware overhead results: for each configuration, it is reported the required number of cells and the percent of area overhead, computed concerning the corresponding configuration in the original version of the model.

The hardware overhead introduced by the BISR strategy can be split into two parts: from one side, there is the cost to implement the instruction, the switching modules, and the switching controller. The "0 SSPs" configuration is used to quantify the hardware cost in the BISR structure. The hardware overhead of these structures represents a low percentage of the whole hardware: for all SP core configurations, the hardware cost is in the range from 0.8% to 1.7%. From the other side, there is the cost for the SSPs, which linearly grows with their number and becomes largely dominant when the SSPs increase. In fact, the addition of one SSP core (6,623 cells) introduces hardware overhead greater than 3% in all SP configurations. Clearly, the hardware overhead rate grows with more SSPs and is higher when the number of SPs is lower. The optimum choice of both parameters depends on the design requirements, e.g., connected to the computation required by the application, by the probability of faults (given by the operating environment and by the semiconductor technology), by the target reliability and by the duration of the mission. In any case, it is worth noting that the hardware overhead remains below 20% for all the considered GPU configurations.

The last two columns in Table 5.2 report some figures allowing the evaluation of the relative size of the SPs with respect to the total size of the FlexGripPlus model. From the results, it is shown that the percent area of the whole SM that can be protected resorting to the BISR strategy ranges from about 25%, in the 8 SPs configuration, to about 55% with 32 SPs. It is worth noting that the adopted BISR mechanism was aimed to mitigate faults in the SP cores, only. Other solutions can be used to mitigate faults in other modules.

Performance and power overhead

Since the BISR strategy requires the introduction of some complex switching modules to flexibly interconnect all the SPs and SSPs with the rest of the system, it clearly impacts the GPU overall performance. We performed an experimental analysis of this phenomenon resorting to the data produced by the synthesis tool. In particular, the impact on the performance of the adapted BISR strategy has been evaluated by analyzing the changes in the critical path delay for all configurations.

Results showed that for a large number of SSPs (6 or 7), the performance degradation reaches up to 20%. This is mainly caused by the logic included in the input and output switches and inside the switch controller. More in detail, for one SSP, the timing degradation is up to 15% and up to 16% for 2 SSPs. Clearly, it should be noted that the reported results have been obtained without executing any specific optimization to reduce such a performance overhead.

The power overhead can be neglected for this BISR strategy by considering that all inactive SSPs and SPs act as cold standby modules. Moreover, other structures remain in a configuration state. Thus, only static power by leakage

Table 5.2: Hardware overhead of the BISR mechanism for multiple configurations of the GPU.

Version	**SP cores**	**SSPs**	**cells**	**Area overhead (%)**	**Total SP cells in the design**	**SP/SSP cores cells (%)**
FlexGripPlus	8	0	229,515	-	52,984	23.08
	16	0	280,132	-	105,968	37.82
	32	0	386,100	-	211,936	54.89
FlexGripPlus + BISR	8	0	231,343	0.8	52,984	22.90
		1	237,279	3.4	59,607	25.12
		2	243,063	5.9	66,230	27.24
		4	254,692	11.0	79,476	31.20
		6	266,182	16.0	92,722	34.83
		7	271,757	18.4	99,345	36.55
	16	0	283,160	1.1	105,968	37.42
		1	290,034	3.5	112,591	38.81
		2	296,164	5.7	119,214	40.25
		4	309,318	10.4	132,460	42.82
		6	321,529	14.8	145,706	45.31
		7	335,139	19.6	152,329	45.45
	32	0	392,476	1.7	211,936	53.99
		1	400,902	3.8	218,559	54.51
		2	410,280	6.3	225,182	54.88
		4	425,172	10.1	238,428	54.07
		6	440,576	14.1	251,674	57.12
		7	460,372	19.2	258,297	56.10

current is consumed during operation. In a real implementation of the strategy, the transistor technology (i.e., 12 or 7nm) presents leakage currents in the order of 10nA/µm to 12nA/µm. Thus, the final power overhead of the BISR strategy is negligible in comparison with the dynamic power consumption produced in the GPU.

Reliability advantages

The goal of the adapted BISR strategy is to allow a GPU-based system to continue working even after one or more faults arose within the SPs. This strategy is independent of the considered fault model, provided that a suitable technique to detect it and identify the affected SP core is available.

The reliability estimation of this strategy is based on the probability of correct operation of the system after a time t. Considering a GPU composed of **n** SPs and **m** SSPs, the execution of the system is correct if all thread instructions are operated without failures in the available execution units of an SM. Moreover, the probability of proper operation in the GPU can be described as the probability of GPU failure (when at most **m+1** SPs or SSPs fail). Both units (SPs and SSPs) are identical and operate independently among them, hence the probability of correct operation at time t in the SPs ($P_{SP(t)}$) and the SSPs is equal. In this way, the probability of proper execution, using the BISR mechanism (R_{BISR}), follows a cumulative distribution function as reported in equation 5.4.

$$R_{BISR} = \sum_{i=0}^{m+1} \binom{n+m}{i} \left[P_{SP(t))}\right]^{n+m-i} \cdot \left[1 - P_{SP(t))}\right]^{i} \tag{5.4}$$

Extracting terms from equation 5.4 (see equation 5.5), the first one at right represents the probability of correct operation in the original GPU ($P_{GPU(t)}$) corresponding to the first element of the sum, so failing after a single faulty unit. On the other hand, the second term at right represents the added probability of a correct operation using the BISR strategy.

$$\begin{gathered} R_{BISR} = \sum_{i=0}^{0} \binom{n}{i} \left[P_{SP(t)}\right]^{n-i} \cdot \left[1 - P_{SP(t)}\right]^{i} + \\ \sum_{i=1}^{m+1} \binom{n+m}{i} \left[P_{SP(t)}\right]^{n+m-i} \cdot \left[1 - P_{SP(t)}\right]^{i} \cdot \left[P_{SW(t)}\right] \cdot \left[P_{C(t)}\right] \end{gathered} \tag{5.5}$$

$$\begin{gathered} R_{BISR} = P_{GPU(t)} + \sum_{i=2}^{m+1} \binom{n+m}{i} \left[P_{SP(t)}\right]^{n+m-i} \cdot \\ \left[1 - P_{SP(t)}\right]^{i} \cdot \left[P_{SW(t)}\right] \cdot \left[P_{C(t)}\right] \end{gathered} \tag{5.6}$$

with:

$$P_{GPU(t)} = \left[P_{SP(t)}\right]^{n} \tag{5.7}$$

As $P_{GPU(t)}$ is a component of the probability for the BISR mechanism, it proves that $R_{BISR} > P_{GPU(t)}$. Thus, the GPU reliability improves according to the second term in equation 5.6. This term also includes the probability of the correct operation of the switching structures $P_{sw(t)}$ and in the controller $P_{c(t)}$. The dominant factor is the total number of SSP units (**m**) added in the BISR structure. Moreover, there is a direct relationship between the number of SSPs (**m**) and R_{BISR}. However, the BISR strategy may be feasible when considering a balance among overhead, cost, and reliability. In principle, **m** cannot be higher than **n**.

Fig. 5.8 represents the increment of the reliability of the BISR version (R_{BISR}) with respect to the original version for multiple values of $P_{ST(t)}$ and SSPs (m). From the graph, it can be noted that R_{BISR} is strongly dependent on the values of **m** and PST. In fact, R_{BISR} presents a maximum reliability peak whose position varies for each combination of PST and **m**. This peak value can be used to select the

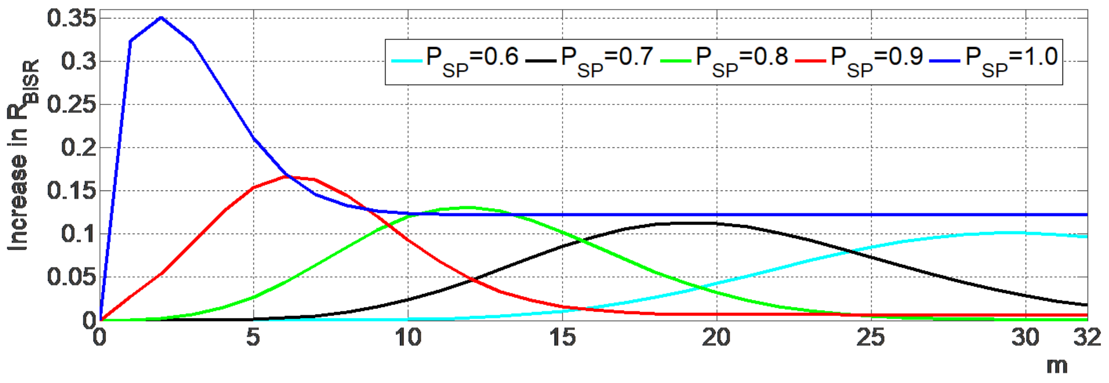


Figure 5.8: Improvement in the reliability of the BISR structure for multiple probabilities of correct execution under multiple configurations of the SSPs (m).

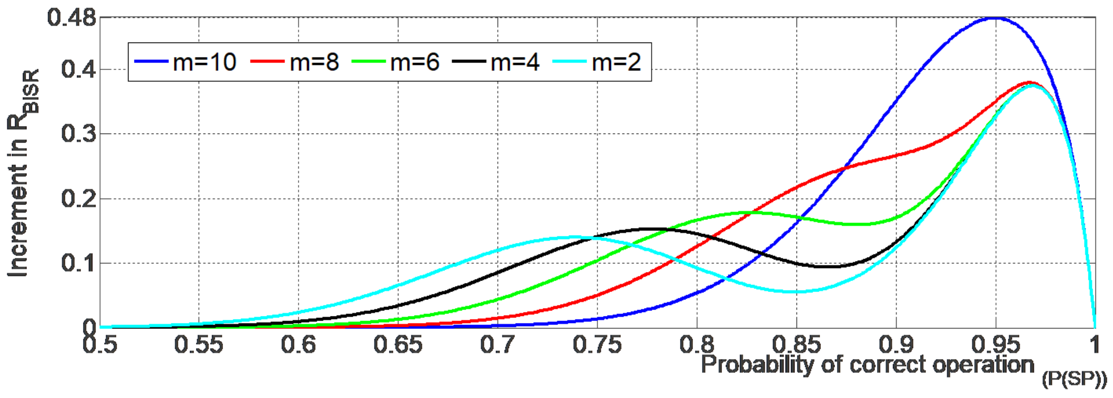


Figure 5.9: Improvement in the reliability of the system RBISR with respect to the probability of correct execution for various values of SSPs (m).

number of SSPs in the GPU considering a target probability of correct execution in the system. After this value, the effectiveness of the strategy drops down.

The graph in Fig. 5.9 describes the relation among $P_{ST(t)}$ and the increment of R_{BISR} for multiple values of **m**. This graph represents the gained benefits in terms of reliability for multiple BISR configurations. As expected, the increase in the number of SSPs (m) has a proportional positive impact on the reliability of the target structure. As can be seen from the graph, the BISR mechanism provides almost 10% of increased reliability, even when the probability of correct execution is dropped by up to 20%. Figures 5.8 and 5.9 can be employed to select the best trade-off among the parameters to reach a given target reliability.

Comparison with other techniques

A comparison with other well-known strategies such as lock-step and TMR can be performed. In principle, a TMR mechanism is highly reliable. However, this is not feasible to be used in the SPs due to the excessive hardware and dynamic power overhead. The lock-step strategy provides a high percentage of fault tolerance for most modules in a GPU. Nevertheless, it requires the duplication of each module. Thus, the hardware overhead is equal to or greater than 100%. A similar situation can be found in terms of power consumption. In contrast, the adapted BISR strategy takes advantage of the regularity of the SPs to reduce the hardware overhead to less than 20% even in the worst case. Moreover, the inactive SPs remain in cold stand-by mode reducing the power consumption of the mitigation strategy.

It must be also underlined that the proposed BISR strategy, based on dynamic reconfiguration, is particularly well suited for long-term missions (which are common for example in the automotive domain) since it allows avoiding the issues created by fault accumulation.

5.3 DYRE: a flexible solution to increase GPU's reliability

This section introduces a flexible solution to perform the detection and the mitigation of faults in GPU cores, so protecting and extending the operative life of a GPU. The proposed solution is based on the addition of some spare modules to perform two in-field operations: the detection and the mitigation of permanent faults of any type of fault that affect the GPU outputs. The proposed solution takes advantage of the regularity of the execution units in the GPU architecture to avoid significant design changes and limit the hardware overhead. The proposed solution was evaluated in terms of area, performance, and power overheads resorting to FlexGripPlus.

As introduced in sections 5.1 and 5.2, mostly all solutions for detection and mitigation of faults are based on hardware, software and hybrid approaches. This subsection proposes a combined and flexible solution called *DYnamic REconfigurable structure for in-field detection and mitigation of faults (DYRE)* that is based on the coalescence of the classical DWC mechanism, for fault detection, and the BISR approach, for fault mitigation. The DYRE architecture is intended to increase the reliability and operative-life, by supporting both the detection and the mitigation of permanent faults in the execution cores of a GPU. This mechanism allows reconfiguring the GPU architecture to identify (through comparisons) and mitigate (by module replacement) possible faults arising during the in-field operation. The architecture of a GPU architecture adopting DYRE can be dynamically re-configured using custom instructions purposely added to the instruction set. Finally, the DYRE architecture is designed to avoid major changes in the original GPU design and to minimize the impact on execution performance.

The next subsections describe the proposed architecture to detect permanent faults in the execution units of GPU cores and mitigate their effects during the in-field operation. Moreover, the DYRE strategy is evaluated experimentally, obtaining The evaluation of the hardware, power, and performance cost involved by the DYRE architecture and of its benefits in terms of reliability enhancement. The analyzes and results show that the overall GPU reliability of the execution cores is improved by 20% to 40% when using the DYRE architecture. Moreover, DYRE introduces less than 1% of performance degradation, less than 5% of hardware costs, and less than 8% of additional power consumption.

5.3.1 Proposed DYRE architecture

DYRE [204] is a dynamic fault-tolerance architecture intended to detect permanent faults in the SP cores of a GPU SM and mitigate their effects. This mechanism takes advantage of the high regularity and homogeneous composition of the SP cores, the parallel execution of the thread/tasks on the SPs, and the static distribution of the tasks among the SPs to reduce the cost in terms of hardware and performance. The DYRE architecture is based on the addition of one or more spare SPs (SSPs) in the Execute stage of the SM. Each additional SSP can be employed for results comparison or replacement purposes.

In particular, an SSP can:

- Be paired to an SP, so that it performs the same operations on the same input data; hence, the results produced by the paired SP and SSP can be compared, and this allows detecting possible faults affecting one of the two modules;
- Replace a faulty SP core.

The architecture of a DYRE GPU differs from a normal one only in the Execute stage (see Figure 5.10) and includes one or more SSP cores, three crossbar units (input, middle and output), some configuration registers, one comparator block (COMP), a controller unit, and some decoding logic. This structure provides flexibility allowing two non-exclusive operational features: *1)* the in-field detection of faults and *2)* the in-field mitigation of faults in the SPs.

The specific architecture of a DYRE GPU can be flexibly and dynamically decided by executing ad hoc assembly instruction introduced in the GPU instruction set to activate the fault detection and fault mitigation features. The *DETection Trigger (DETT)* instruction configures an SSP to be paired with an SP, thus enabling the comparison between the results produced by the pair for fault detection

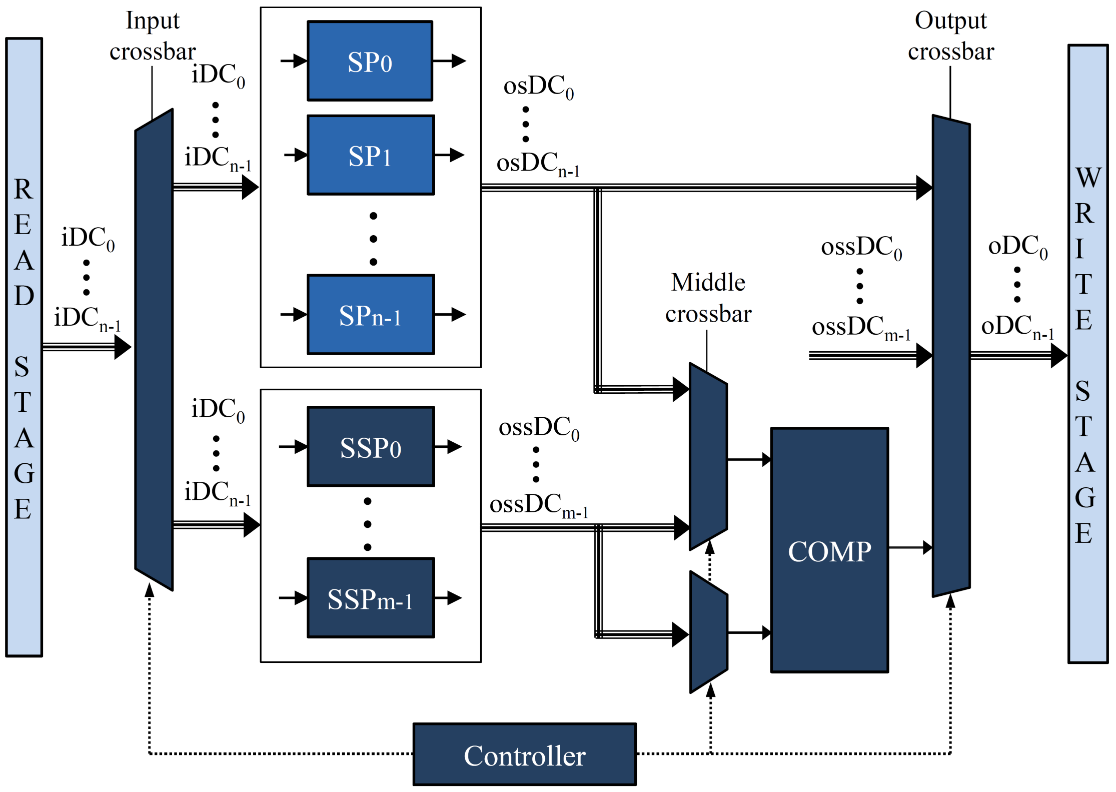


Figure 5.10: A general scheme of the Execute stage of a GPU with the DYRE architecture.

purposes. Similarly, the *MITigation Trigger (MITT)* instruction re-configure the GPU substituting one SP with an SSP for mitigation purposes. Both instructions can reconfigure the DYRE architecture with the cost of one instruction cycle and are intended to be included in a running application, so enabling both features of a DYRE GPU. Both operational features (*detection* and *mitigation*) use the same hardware structures, thus reducing the overall hardware cost. However, a single SSP cannot be used for detection and mitigation at the same time.

Fault Detection

This operational feature is inspired by the DWC mechanism and uses a sphere of redundancy composed of the active SPs in the SM. The DYRE architecture uses this feature to detect faults through the active comparison of results after each executed instruction by an SM. More in detail, When the DETT instruction is executed, the local controller enables the fault detection feature, and one SSP and one active SP are selected to perform all the following instructions in parallel. After this procedure, the detection feature is transparent for the application. The SP and the SSP can be paired by a time interval or the entire execution of the application.

Moreover, the target SSP or SP can be replaced with another core at any moment of the in-field operation by executing again a new DETT instruction.

The structure of the DYRE architecture uses two crossbars (input and middle) to select a target SP. Both crossbars select and duplicate the input and output data channels to feed the SSPs core and the comparator block, respectively.

After each operation, the results of the SP and the SSP are compared. The comparator triggers a faulty flag when a mismatch is detected. The flag is propagated to the next stage and sent to the exceptions unit in the GPU or the Host.

Fault Mitigation

This operational feature is based on an adaptation of the BISR mechanism, and it is intended to mitigate the effect of faults in the cores by disabling and replacing one affected SP core with one of the available SSPs in the system. The SSPs are organized as cold standby modules and are active only when required. Correspondingly, the inactive SP cores are disabled to reduce the power consumption during inactivity.

The static distribution of tasks among the SPs allows the correction of faults by switching the input data from a faulty unit to a fault-free unit. This behavior also reduces further changes in other modules of the GPU. For this purpose, it is possible to mask the replacement of a faulty SP by an SSP. Thus, the fault-mitigation structure operates transparently for the point of view of the memory and the scheduling controller.

More in detail, the execution of the MITT instruction activates two crossbars (*input* and *output*, as depicted in Figure 5.10) to redirect the data-flow of the data channel from one active SP (faulty core) to the selected SSP (fault-free), so mitigating the fault effect. The effect of the MITT remains active for all subsequent instructions.

Methods of use

The DYRE architecture is intended to operate in two cases: *i)* in the Power-on/reset phase of the device and *ii)* during the in-field operation of an application.

At the power-on, the DYRE architecture is inactive. Hence, the SSPs are initially idle as cold standby modules. A specially crafted test program applies patterns to check the possible presence of permanent faults in each SP. This program includes several DETT instructions that activate one SSP and swap the available SPs to perform comparisons with it when executing the same instructions on the same data. If a mismatch is found, the SP is labeled as faulty. The program replaces the faulty SPs by SSPs thought MITT instructions, and the application starts. It is worth noting that the generation of suitable test programs for the SPs is out of the scope of this work. However, the techniques proposed in Chapter 4 showed that generating them is feasible.

On the other hand, the use of DYRE during in-field operation requires the addition of one or several DETT instructions in the application code. Each DETT instruction selects one SSP, so activating the fault detection through comparisons. A fault is detected when, during the execution of the instructions of the application, a comparison produces a mismatch. Then, a subroutine activates the MITT instruction to replace the faulty SP by one SSP. This subroutine can be launched when a mismatch is generated or during the idle times of an application. The replacement subroutine (with MITT) is intended to substitute the faulty core with minimal latency in the execution of the application, considering the low reconfiguration cost of the mitigation feature.

It is worth noting that the DYRE architecture, as the BISR mechanism, does not include any fault administration structure to store the actual configuration state and to be possibly restored after a device power-off or reset. However, a similar proposed solution for the BISR mechanism can also be used for DYRE as well.

Implementation

DYRE was implemented in FlexGripPlus, modifying the *Decode*, *Read*, and *Execute* stages. The hardware to support the DETT and MITT instructions was added in the Decode stage. Similarly, a bypass mechanism and some changes in the memory controllers were performed in the Read stage to add flexibility to the instructions. The implementation allows the adoption of the DYRE architecture with any of the three SP configurations (8, 15, and 32) of the model.

The *Execute* stages include the additional SSPs, the crossbars, and the controllers of the DYRE architecture. The main purpose of the crossbars is the selection of the input and output data channels (iDCx and oDCx) to feed the SPs and SSPs in the system. The input crossbar selects one of the iDCx feeding the active SP cores and can duplicate or switch the input data to one of the SSPs. In case of duplication, the selected SSP redundantly executes precisely the same operation of the selected SP. In contrast, in the case of switching, the input crossbar substitutes the iDCx of one SP core and feeds a selected SSP. The control signals of the SP cores are statically shared among the SP and SSPs in the system.

The middle crossbar is formed of two independent crossbars used to feed the two inputs of the COMP module. COMP is only used during the fault detection operation and is composed of a bitwise comparator that compares the results and output flags from two execution units (SPs or SSPs). On the other hand, the output crossbar manages the results coming from the active SPs and SSPs. This crossbar is used to select the output channels (osDCx and ossDCx) from the active SPs and SSPs and feed the next pipeline. The flexibility of the middle crossbar allows the comparison of two SSPs when the mitigation and duplication mode are simultaneously activated.

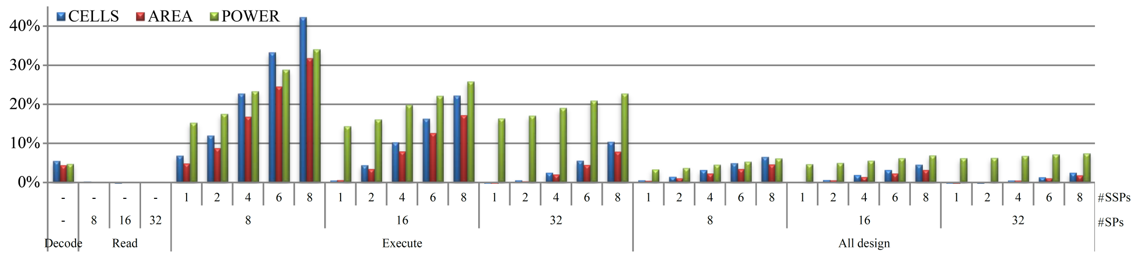


Figure 5.11: Percentage of overhead cost of the DYRE architecture on each adapted module and in the entire GPU.

The input and output crossbars are indeed meta-crossbars, and multiplexer structures used to preserve the same type of input and output data channels in the *Execute* stage and from and to other stages of the SM.

Some configuration registers are employed to select among the operational features (detection, mitigation, or both). The local controller configures the DYRE architecture using decoded commands that come from the DETT and MITT instructions. Some decoding logic is included to manage the two operational features when controlling the crossbar structures.

The DETT and MITT instructions were designed to select the channels or target cores using operands coming from an immediate value or a general-purpose register. This flexibility in the instruction format allows the dynamic selection of the target core during the in-field operation. Both instructions use a format composed of six bits stating the instruction type. Other five bits select the input data channel to be switched for duplication or replacement, and five bits select the target SSP core to be used.

5.3.2 Experimental Results

Two evaluations are performed on the proposed mechanism. Firstly, the overhead assessment determines the cost in terms of hardware, power, and performance of the DYRE architecture. For this purpose, the DYRE architecture is compared with the original design, one based only on fault-detection (DDWC), see section 5.1, and one based only on fault mitigation, see section 5.2. The original GPU and the three fault-tolerance mechanisms were synthesized using the Design Compiler tool using the 15 nm Nandgate Open-cell library and 500MHz Clock. It is worth noting that the internal memories were not synthesized. Fig. 5.11 reports the results of the hardware and power overhead for each setup. Finally, a second evaluation analyses the reliability features of the proposed mechanism.

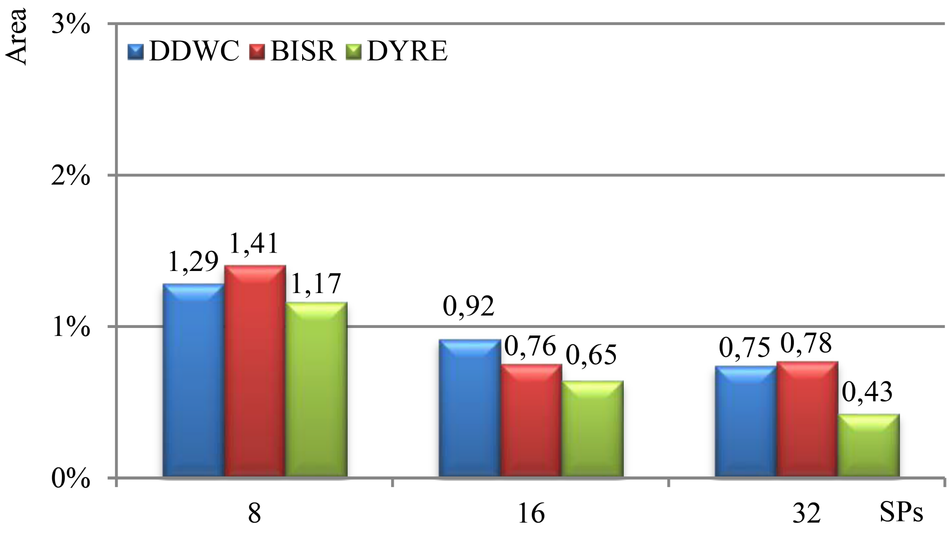


Figure 5.12: Area overhead for the DDWC, BISR and DYRE architectures with respect to the original design evaluated in the 8, 16 and 32 SP cores configurations with one SSP.

Hardware overhead analysis

Two cases were considered for the hardware overhead evaluation: *i)* considering the affected modules, only and *ii)* considering the whole system. In the first case, the assessment was performed considering the modules affected by modifications when implementing DYRE. In the second case, the cost of the entire design is evaluated. All evaluations were performed using the three configurations with 8, 16, and 32 SPs.

According to results, the hardware cost of implementing the instructions in the Decode stage is lower than 5% and almost negligible for the Read stage (≈0.3%). On the other hand, the implementation of the SSPs directly affects the hardware cost in the Execute module. For a configuration of 8 SPs, the cost of using two SSPs is lower than 13%, but it increases to 42% when DYRE is configured to use the same SPs and SSPs. Among the SP configurations, it can be noted that in the Execute module and the entire design, the hardware overhead follows a proportional inverse relation. Thus, large SP configurations present low hardware overhead. In Execute, when adding 25% of SSPs, the cell and area costs are around 10% and 8%, respectively. In the case of adding 50% of SSPs, these costs are about 22% and 17%.

On the other hand, the hardware overhead in the logic of the design is lower than 7% for all configurations. In the case of two SSPs, the cell and area overhead is lower than 2%, so causing a minimum impact on the design when using DYRE. When the SM is configured with 32 SPs, the addition of one or two SSPs caused negative percentages of hardware overhead. However, these values are due to the optimization constraint in the synthesis tool, and the effect is translated as power overhead for these configurations

5.3.3 Power and performance analysis

From Fig. 5.11, the power consumption in the Decode module indicates a minimum overhead (<5%), and it is almost negligible in the Read module for all SP and SSP configurations. In the Execute module, the addition of one or two SSPs in all SP configurations causes a moderate average cost of power from 14 to 17%. When DYRE is configured to include 50% of SSPs for each SP configuration, the power cost is moderate (around 23.7% and 25.9%). Moreover, the overhead reaches up to 34% when the number of SSPs and SPs is equivalent. Nevertheless, the entire logic cost remains stable and is lower than 8% in all configurations.

In terms of performance, the DYRE architecture does not introduce more than 1% of degradation in the critical path for all the evaluated configurations.

Although the synthesis of the model used only the clock constraint, the results in Fig. 5.11 show the distribution and the trend to consider when implementing the DYRE solution. In this way, the addition of two SSPs can be affordable in terms of hardware (<2%) and power (<8 %) costs.

An overhead comparison of DYRE with the fault-detection (DDWC) and the fault-mitigation (BISR) architectures is reported in Fig. 5.12. Each strategy was implemented and synthesized for the three possible SP configurations. In principle, results show that hardware overhead in DYRE is the lower of the three strategies (<5%) and decreases when increasing the number of SPs in the design. However, the main overhead is represented in terms of power consumption.

Regarding power consumption, the DYRE architecture increases the power of the system from 4.55% to 8.72% with respect to the original design. This can be explained considering that the additional structures (controller, multiplexers, and the comparator block) remain active, so consuming static power even when the DYRE architecture is inactive. However, this cost might be reduced, including power optimization strategies. It should be noted that power optimization techniques were not used in any of the three strategies during the experiments.

5.3.4 Reliability analysis

The reliability of the DYRE architecture is estimated by determining the probability of correct operation, which depends on the number of available and fault-free SP and SSP modules. The proper execution of the system is obtained when all thread-operations are performed without failures affecting the execution cores. This probability of correct operations can be inverted and expressed as the probability of failure (when some SPs or SSPs fails). The dual-modules feature of the DYRE architecture influences the reliability calculation and the number of cumulative faults affecting SPs or SSPs before the overall architecture encounters a functional error.

During fault-free operations, both groups of SP and SSP modules are identical and operate in parallel independently among them. Considering this scenario, the probability of correct operation of the DYRE architecture (R_{DYRE}) can be computed by adopting a binomial distribution function using **n** SPs and **m** SSPs module, respectively. R_{DYRE} is composed of the probability of a fault in an SP($P_{Core(t)}$) at a given time t and a K limit related to the active operational features (*mitigation* and *detection*), as reported in equation 5.8.

$$R_{DYRE} = \sum_{i=0}^{k} \binom{n+m}{i} \left[P_{core(t))}\right]^{n+m-i} \left[1 - P_{core(t))}\right]^{i} \tag{5.8}$$

In detail, when the fault mitigation feature is active, a failure in the overall system occurs when k=(m+1) execution units (SPs or SSPs) are faulty, hence it is not possible to complete the thread operations without errors. On the other hand, if both features are active, the system produces a failure when k=m execution unit fails since one SSP is used as a comparator during the in-field fault detection. Finally, in case the fault detection feature is enabled or both detection and mitigation features are disabled, there are no available SSPs dedicated to fault mitigation, therefore k=0 and m=0.

In order to determine the advantage, in terms of probability of correct operation, for the SM using the DYRE architecture, we introduce the equation 5.9. Equation 5.9 is composed of two terms. The first term corresponds to the probability of correct operation of the SM without the DYRE architecture ($P_{SM(t)}$). In contrast, the second term represents the improved probability of correct operation by adopting the DYRE architecture ($\triangle R_{DYRE}$). This term also includes the probabilities of correct operation for the switching modules ($P_{sw(t)}$) and the controller ($P_{c(t)}$).

$$\begin{gathered} R_{DYRE} = \sum_{i=0}^{0} \binom{n}{i} \left[P_{core(t))}\right]^{n-i} \cdot \left[1 - P_{core(t))}\right]^{i} + \\ \sum_{i=1}^{k} \binom{n+m}{i} \cdot \left[P_{core(t))}\right]^{n+m-i} \cdot \left[1 - P_{core(t))}\right]^{i} \cdot \left[P_{sw(t))}\right] \cdot \left[P_{c(t))}\right] \end{gathered} \tag{5.9}$$

As it can be noted in equation 5.9, the number of SSPs (**m**) determines the probability of correct operation in the GPU. The behavior of $\triangle R_{DYRE}$ concerning the probability of correct operation on SPs ($P_{core(t)}$) has been plotted in Fig. 5.13. The graph describes the relationship between $P_{core(t)}$ and $\triangle R_{DYRE}$ for multiple values of **m**. The almost stable behavior of about 20% to 40% of positive increment impacts $\triangle R_{DYRE}$ when **m** increases and $P_{core(t)}$ thoroughly decreases.

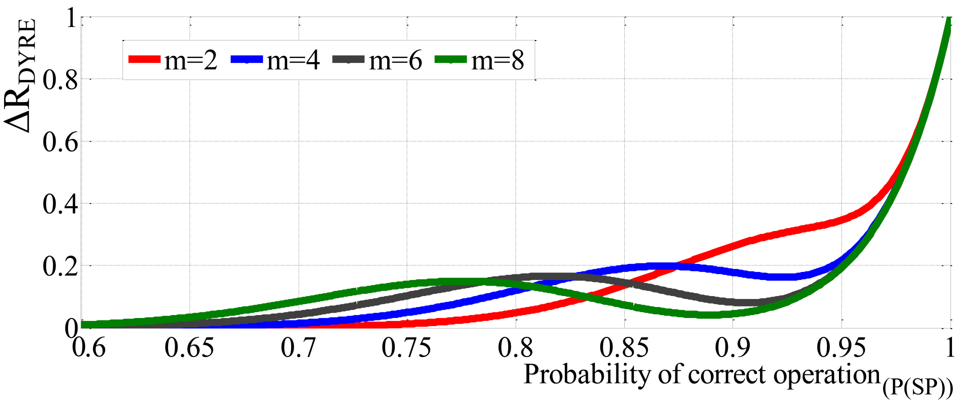


Figure 5.13: Reliability benefit in the system for multiple probabilities of correct operation.

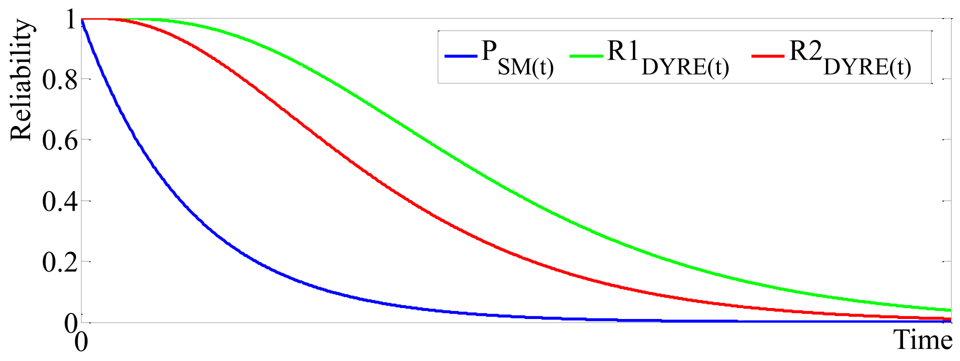


Figure 5.14: Reliability comparison of a standard GPU and other using the two features of the DYRE architecture with two SSPs.

Furthermore, the comparison between the reliability behavior of a standard GPU ($P_{SM(t)}$) and the one of an architecture adopting the two features of the DYRE architecture (*mitigation* only ($R1_{DYRE}$) and *detection+mitigation* ($R2_{DYRE}$)) is plotted in Fig. 5.14., using a typical probability function ($P_{core(t)} = e^{-\alpha t}$) in both cases. This figure shows the reliability when adding two SSPs in the system. As it can be observed, the reliability of a DYRE GPU (R1 and R2) remains higher than without DYRE, so extending its operative life. Under these conditions, a detailed analysis revealed that in some points the reliability is increased by up to 57%, when the mitigation and detection features are active, and 72% with the mitigation only feature. However, it must be noted that a precise evaluation of the reliability features of both modes of operation of the DYRE architecture (*detection* and *detection+mitigation*) requires the consideration of technical characteristics of the final transistor technology.

5.4 Conclusions

This chapter introduced three hybrid and flexible mechanisms to increase the reliability of GPU cores by adding and activating special infrastructures during the operative stage of the system. These mechanisms are called DDWC, BISR and DYRE.

DDWC targets the in-field detection of permanent faults in the execution units (SPs) in SMs of a GPU. DDWC is based on the duplication with comparison strategy and exploits the structural regularity of the SP cores. The SP core to be monitored can be dynamically selected, resorting to one ad-hoc instruction. Thanks to its flexibility, low hardware overhead, and moderate performance degradation, this strategy could be effectively employed to increase the reliability of GPUs when they are adopted in safety-critical applications.

Experimental results show that the proposed DDWC mechanism introduces a limited area overhead while it provides a significant increase in the in-field fault detection capabilities of the GPU. Its flexibility allows selecting the best trade-off between fault detection latency and performance overhead.

The dynamic BISR targets the mitigation of permanent faults possibly affecting the execution units (SPs) in the SM of a GPU. The BISR mechanism is based on the addition of spare modules, which are controlled via software using a custom instruction. This instruction allows removing a faulty SP from the set of active ones, substituting it with one of the available spare SP cores. Results show that the structures required to implement the proposed technique introduce a relatively low hardware overhead (<4% with a single spare core). Moreover, we showed that the area of the modules where faults can be tolerated with the BISR structure can achieve about 55% of the total SM area.

The strategy seems particularly suitable for long-term missions since it allows mitigating the effects of fault accumulation in the SP cores. Although the validation experiments were performed on a specific NVIDIA-based GPU architecture, the proposed solution can be easily extended to other architectures as well.

Finally, DYRE targets the detection and mitigation of permanent faults affecting the execution units in GPUs. DYRE provides a solution that can be employed during the operative life of a GPU and extend the reliability capabilities by 20% to 40% for most configurations of the execution units of these devices. Moreover, the overall reliability can be increased in up to 72% when using the mitigation feature. The experimental results let us affirm that adding the proposed DYRE mechanism into a GPU design requires a minimum to moderate cost that directly depends on the number of additional cores included to support fault detection and/or mitigation.

Chapter 6

Conclusions

6.1 Summary

The main conclusions and contributions of the research activities are listed in this chapter. Moreover, some future activities and possible directions in the field of fault testing and mitigation of GPU architectures are also described. It is worth noting that a considerable part of the performed research activities was possible due to the availability of a microarchitectural GPU model (FlexGripPlus), which also represents a first and important contribution of this work.

6.2 Main contributions

6.2.1 Reliability evaluation of GPUs

The detailed microarchitectural evaluation of individual modules in a GPU core allows identifying specific effects and trends in the execution of applications in GPU devices when affected by transient faults. For example, the impact of transient faults in the scheduler controller denoted a proportional relation with the workload and the percentage of use of this module. Interestingly, other modules showed different relations, such as the register file (highly sensitive) and the divergence stack (poorly impacted by faults). Moreover, for the first time, individual microarchitectural evaluations were performed on GPU modules, which contribute to determine and understand the most sensitive modules in the parallel architecture.

The evaluation of the scheduler controller allows the evaluation of the fault sensitivity of this module. In fact, it was possible to quantitatively evaluate the proportional relation between increasing the workload of an application and the impact on fault sensitivity of the scheduler controller. Similarly, the microarchitectural evaluation provides relevant information on the main sensitive sources in the pipeline registers of the GPU core. The control path registers in the pipeline are highly sensitive and are responsible for more than 80% of the observed faults but only represent about 16% of the size of the module. In contrast, other modules in the data path of the GPU denoted higher impact to transient faults, such as the register file and the pipeline registers. According to results, the pipeline registers and the register file are sensitive locations, these conclusions acts in accordance to related works in the field suggesting and proposing mechanisms to mitigate the fault effects on those modules. Unfortunately, in some cases (i.e., the variation in the number of parallel cores), the evaluations do not provided clear trends and complementary reliability evaluations are required to determine effects and trends of the fault impact on GPUs.

6.2.2 Functional software-based self-testing of GPUs

The main core of this research work was devoted to devising functional test strategies aiming at fault detection in GPUs during the in-field operation. The functional test strategies were developed based on the SBST strategy, and several modules were targeted and tested using custom, multi-kernel, modular and automated approaches.

The custom approach was used to develop test strategies for particular modules in GPU architectures such as the scheduler controller and its memories. The proposed approach also considered the targeted fault model and reached more than 80% of faults detected in each case. This approach proves that deterministic procedures can be employed in the development of test programs for GPUs. However, microarchitectural details of the target GPU are required in order to guarantee a high fault coverage. Similarly, the use of lower level languages (i.e., Assembly language) in the development and implementation of the test programs was required, when observing that the traditional compiler optimize the high-level code for performance, so translating the test routines incorrectly.

In the case of the multi-kernel approach, this method was employed on modules relevant for the GPU operation but included configurable parts that remain fixed during the operation of a parallel program, such as the pipeline registers. In this case, several parallel test programs were required to provide the test patterns and propagate most fault effects. Moreover, new parallel strategies of fault propagation were devised, including the *signature per thread* approach to identify and detect faults on parallel and singular units in the GPU. Moreover, it was proved that these signatures could also provide diagnosis features during the test. This approach can be exploited on configurable modules in the GPU, but also can be used when testing other modules in the GPU, which require of specific kernel configurations, such as the memory interconnections not targeted in the present work.

We also explored the use of high-level constructs in combinations with assembly instructions. Several advantages, constraints, and limitations were found regarding the compiler optimizations and the lack of direct mapping from a high-level constructs with assembly instructions.

The proposed modular approach to design generic test programs is able to exploit the main microarchitectural features of a target module in combination with the target fault model. In this case, a generic test program is designed and intended to be mapped as several software routines, then it is finally mapped considering the available instructions in the GPU. Moreover, the modular testing approach allows the exploration of different implementations for a given test program. This approach was effective in detecting permanent faults in several internal memories of the GPU core and the divergence stack of the convergence management module.

The adoption of automatic approaches from CPUs into GPUs proved to be effective on regular structures of the parallel architectures, such as functional units. In the results, the automatic approaches, based on ATPG-based and pseudorandom-based methods, denoted a high capability to test permanent faults on the functional units of the GPU. Thus, the automatic approach can be employed as a complementary testing technique for the GPU.

In the end, the developed functional test strategies were evaluated on the complete GPU core. According to results, up to 92.6% of the permanent faults in the GPU core can be detected using SBST strategies. These results support the claim that SBST strategies can be effectively employed as complementary and alternative solutions for in-field test of GPU devices. Nevertheless, additional effort is still required when functionally testing the memory hierarchy in the GPU. For the purpose of this work, the test of memory hierarchy in GPUs is still considered as an open point.

Regarding the functional-safety analysis, this analysis provided the main features of the SBST strategy when applied into GPUs. In the analysis, the SBST strategy showed a promising performance when employed as safety mechanism for GPUs targeting the safety-critical domain. In the result, the analysis shows that functional solutions based on SBST strategies applied to GPU cores can assess an ASIL B level. Thus, STLs based on SBST strategies can be consider as alternative and complementary safety mechanisms for GPUs.

6.2.3 Fault mitigation strategies for GPUs

Three flexible strategies were proposed and evaluated, combining hardware structures and controlled, employing custom instructions added to the ISA of the GPU. The three strategies target the protection of the functional units by performing in-field fault detection, fault mitigation, or both. The first strategy (DDWC) exploits the high regularity of the functional units in the GPU core to implement a dynamic duplication with comparison and provide a mechanism to evaluate and detect faults during the execution of an application. The results showed that less than 20% of hardware overhead is required for an implementation of this strategy.

The second strategy (BISR) targets the mitigation and repair of fault by adding spare functional units in the GPU. The proposed strategy is programmable during the in-field operation, so allowing the change of a faulty module by an available spare one. The cost of the BISR strategy is similar to the one of the DDWC.

Finally, the third strategy (DYRE) allows in-field fault detection and fault mitigation by combining the main features of the DDWC and BISR strategies and allowing the simultaneous operation of both reliability features. This strategy is controlled by one custom instruction allowing the enabling and disabling of one or both features during the GPU execution. The reliability estimation showed that the reliability benefits could reach up to 50%. Furthermore, the hardware overhead is lower than 10% for this strategy.

The observed performance of the flexible strategies in the GPUs shows that alternative mechanisms can be included in GPU devices with the main purpose of extending the in-field operation with a limited cost in hardware and performance. In some cases, the limited hardware changes (<10%) on the original design can extend the reliability benefits in up to 50% (in the DYRE architecture) in comparison with the original design. These results also suggest that one or more flexible mechanisms can be employed as alternative fault-tolerance and provide in-field test and in-field mitigation capabilities on GPUs.

6.3 Future works

As future works, the development of hardware, hybrid, and software fault-tolerance strategies for GPUs can be explored using the new microarchitectural GPU model. FlexGripPlus allows the implementation and also evaluation in terms of hardware, performance, and power overhead of any hardware-based solution, as performed in the proposed strategies of chapter 5. Moreover, the access to microarchitectural details also allows the tracing of individual modules when executing any reliability strategy. These advantages are not always possible and feasible in architectural models of GPUs, so FlexGripPlus can be used to implement and evaluate such strategies. Moreover, the GPU model's available ISA and architectural description also allow the exploration of new extensions based on software-based hardening for GPUs. Similarly, the exploration and development of software solutions based on pseudo-assembly languages, such as PTx represents a challenge for GPU architectures and has not been fully exploited.

Similarly, other future works include the exploration of functional test techniques based on SBST for other modules in the GPU, such as crossbar units and global scheduler controllers that were not targeted in the present work. In fact, these modules have minimal observability mechanisms in real devices. Moreover, the missing microarchitectural details limit the injection of test patterns. However, innovative or indirect mechanisms to test such modules could be explored for GPUs.

Reliability evaluations and hardening solutions can be extended to other particular modules in the GPU architecture, such as the scheduler controllers and the special accelerators, including Tensor Core Units (TCUs) and Texture Mapping Units (TMUs).

The exploration and development of fault detection and mitigation strategies in the GPU may also require targeting other fault models, such as transition delay faults affecting the execution unit of GPUs.

Finally, the high complexity of modern applications may suggest exploring and developing high-level fault models to be employed during the development and validation of online fault tolerance and mitigation mechanisms. Hence, accurate high-level models for parallel architectures have not being exploited at all.

Chapter 7

Publication list

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

7.1 Part of the research work

The following list enumerates the publications that contain the main contributions supporting the present thesis.

- B. Du, **J. E. R. Condia**, M. Sonza Reorda and L. Sterpone, "About the functional test of the GPGPU scheduler," *2018 IEEE 24th International Symposium on On-Line Testing And Robust System Design (IOLTS)*, Platja d'Aro, Spain, 2018, pp. 85-90, doi: 10.1109/IOLTS.2018.8474174.
- B. Du, **J. E. R. Condia**, M. Sonza Reorda and L. Sterpone, "On the evaluation of SEU effects in GPGPUs," *2019 IEEE Latin American Test Symposium (LATS)*, Santiago, Chile, 2019, pp. 1-6, doi: 10.1109/LATW.2019.8704643.
- B. Du, **J. E. R. Condia** and M. Sonza Reorda, "An extended model to support detailed GPGPU reliability analysis," *2019 14th International Conference on Design & Technology of Integrated Systems In Nanoscale Era (DTIS)*, Mykonos, Greece, 2019, pp. 1-6, doi: 10.1109/DTIS.2019.8735047.
- S. Di Carlo, **J. E. R. Condia** and M. Sonza Reorda, "On the in-field test of the GPGPU scheduler memory," *2019 IEEE 22nd International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*, Cluj-Napoca, Romania, 2019, pp. 1-6, doi: 10.1109/DDECS.2019.8724672.
- **J. E. R. Condia** and M. Sonza Reorda, "Testing permanent faults in pipeline registers of GPGPUs: A multi-kernel approach," *2019 IEEE 25th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, Rhodes, Greece, 2019, pp. 97-102, doi: 10.1109/IOLTS.2019.8854463.

- **J. E. R. Condia**, P. Narducci, M. Sonza Reorda and L. Sterpone, "A dynamic reconfiguration mechanism to increase the reliability of GPGPUs," *2020 IEEE 38th VLSI Test Symposium (VTS)*, San Diego, CA, USA, 2020, pp. 1-6, doi: 10.1109/VTS48691.2020.9107572.
- **J. E. R. Condia**, P. Narducci, M. Sonza Reorda and L. Sterpone, "A dynamic hardware redundancy mechanism for the in-field fault detection in cores of GPGPUs," *2020 23rd International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*, Novi Sad, Serbia, 2020, pp. 1-6, doi: 10.1109/DDECS50862.2020.9095665.
- **J. E. R. Condia** and M. Sonza Reorda, "On the testing of special memories in GPGPUs," *2020 IEEE 26th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, Napoli, Italy, 2020, pp. 1-6, doi: 10.1109/IOLTS50870.2020.9159711.
- **J. E. R. Condia** and M. Sonza Reorda, "Testing the Divergence Stack Memory on GPGPUs: A Modular in-Field Test Strategy," *2020 IFIP/IEEE 28th International Conference on Very Large Scale Integration (VLSI-SOC)*, Salt Lake City, UT, USA, 2020, pp. 153-158, doi: 10.1109/VLSI-SOC46417.2020 .9344088.
- **J. E. R. Condia**, B. Du, M. Sonza Reorda, L. Sterpone, "FlexGripPlus: An improved GPGPU model to support reliability analysis," in *Microelectronics Reliability*, Vol. 109, 113660, 2020, ISSN 0026-2714, doi: 10.1016/j.microrel.-2020.113660.
- S. Di Carlo, **J. E. R. Condia** and M. Sonza Reorda, "An On-Line Testing Technique for the Scheduler Memory of a GPGPU," in *IEEE Access*, vol. 8, pp. 16893-16912, 2020, doi: 10.1109/ACCESS.2020.2968139.
- **J. E. R. Condia**, P. Narducci, M. Sonza Reorda, L. Sterpone, "DYRE: a DYnamic REconfigurable solution to increase GPGPU's reliability," in *The Journal of Supercomputing*, 2021. doi:10.1007/s11227-021-03751-2.
- **J. E. R. Condia** and M. Sonza Reorda, "Modular Functional Testing: Targeting the Small Embedded Memories in GPUs, " in Cross-Layer Reliability of Computing Systems, *VLSI-SOC (28th IFIP/IEEE International Conference on Very Large Scale Integration) 2020 book series, Springer Nature*, 2021, ISBN: 978-3-030-81640-7.
- **J. E. R. Condia**, F. F. Dos Santos, M. Sonza Reorda and P. Rech, "Combining Architectural Simulation and Software Fault Injection for a Fast and Accurate CNNs Reliability Evaluation on GPUs, " *2021 IEEE 39th VLSI Test Symposium (VTS)*, Virtual Event, 2021, doi: 10.1109/VTS50974.2021.9441044.

- J. D. Guerrero-Balaguera, **J. E. R. 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)*, 2021, pp. 81-86, doi: 10.1109/DDECS52668.2021.9417025. **Best paper Award** in the field of testing.

7.2 Other publications

The following list of research works covers other side activities that were not explicitly included as part of the thesis, but supported several steps in the development process of the research.

- B. Du, **J. E. R. Condia**, M. Sonza Reorda, L. Sterpone, "An open source embedded-GPGPU model for the accurate analysis and mitigation of SEU effects," *2019 30th European Conference on Radiation and Its Effects on Components and Systems (RADECS)*, Toulouse, 2019, pp 1-6, doi: 10.5281/zenodo.4662662.

- M. M. Goncalves, J. R. Azambuja, **J. E. R. Condia**, M. Sonza Reorda and L. Sterpone, "Evaluating Software-based Hardening Techniques for General-Purpose Registers on a GPGPU," *2020 IEEE Latin-American Test Symposium (LATS)*, Maceio, Brazil, 2020, pp. 1-6, doi: 10.1109/LATS49555.2020.9093682.

- **J. E. R. Condia**, M. M. Goncalves, J. R. Azambuja, M. Sonza Reorda and L. Sterpone, "Analyzing the Sensitivity of GPU Pipeline Registers to Single Events Upsets," *2020 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, Limassol, Cyprus, 2020, pp. 380-385, doi: 10.1109/ISVLSI49217.2020.00076.

- M.M. Gonçalves, **J. E. R. Condia**, M. Sonza Reorda, L. Sterpone, J.R. Azambuja, "Improving GPU register file reliability with a comprehensive ISA extension," in *Microelectronics Reliability*, Vol. 114, 113768, 2020, ISSN 0026-2714, doi:10.1016/j.microrel.2020.113768.

- **J. E. R. Condia**, J. D. Guerrero-Balaguera, C. F. Moreno-Manrique and M. Sonza Reorda, "Design and Verification of an open-source SFU model for GPGPUs," *2020 17th Biennial Baltic Electronics Conference (BEC)*, Tallinn, Estonia, 2020, pp. 1-6, doi: 10.1109/BEC49624.2020.9276748.

- A. Bosio, S. Di Carlo, G. Di Natale, M. Sonza Reorda, **J. E. R. Condia**, "Design techniques to improve the resilience of computing systems: software layer, " in Cross-Layer Reliability of Computing Systems, *IET - The Institution of Engineering and Technology*, pp.95-112, 2020, ISBN: 978-1785617973.

- **J. E. R. Condia**, P. Rech, F. F. dos Santos, L. Carro, M. Sonza Reorda, "Protecting GPU's Microarchitectural Vulnerabilities via Effective Selective Hardening," *2021 27th IEEE International Symposium on On-Line Testing and Robust System Design (IOLTS)*, Virtual event, 2021, doi: 10.1109/IOLTS52814.2021.9486703.
- F. F. dos Santos, **J. E. R. Condia**, L. Carro, M. Sonza Reorda, P. Rech, "Revealing GPUs Vulnerabilities by Combining Register-Transfer and Software-Level Fault Injection," *51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks(DSN)*, Virtual event, 2021, doi: 10.1109/ DSN48987.2021.00042.
- J. D. Guerrero-Balaguera, **J. E. R. Condia**, and M. Sonza Reorda, "A Novel Compaction Approach for SBST Test Programs," *2021 30th Asian Test Symposium (ATS)*, Virtual event hosted by Japan, 2021, pp. 1-6.

Appendix A

The FlexGripPlus GPU model

As briefly introduced in Chapter 2, the FlexGripPlus model provides a microarchitectural description of a GPU core using the G80 architecture by NVIDIA. This section describes the control modules of the FlexGripPlus model and the Instruction Set Architecture of the GPU model (SASS) supported and compatible with the NVIDIA Programming environment (CUDA) under the SM_1.0.

General microarchitecture of FlexGripPlus

This section describes the main modules in the SM of the FlexGripPlus model. These modules are: *i)* The warp scheduler controller, *ii)* Convergence management unit, *iii)* Memory Hierarchy, and *iv)* the functional units.

A.0.1 Warp Scheduler controller

A warp scheduler controller unit manages the thread execution in the FlexGripPlus architecture with a limit of 32 threads per warp.

The warp scheduler controller is mainly composed of a warp generator, warp scheduler, warp checker, and two main memories to store information about the warp threads, namely the warp state memory and the warp pool memory. The warp pool memory stores the ID, the execution program counter, and the thread mask for each warp. The warp state memory stores the warp's actual state (active, wait, stop and finish). Each entry line in both memories corresponds directly to a warp that is or will be executed in the SM. For every thread in a warp, the fence registers store in a bit the thread's state, indicating whether the thread is in use or a waiting state. The warp scheduler reads and checks the fence registers to identify when all warp threads are waiting. If this condition is true, it means that all previous divergence paths converged, and the synchronization barrier is released so that warps can return to normal operation.

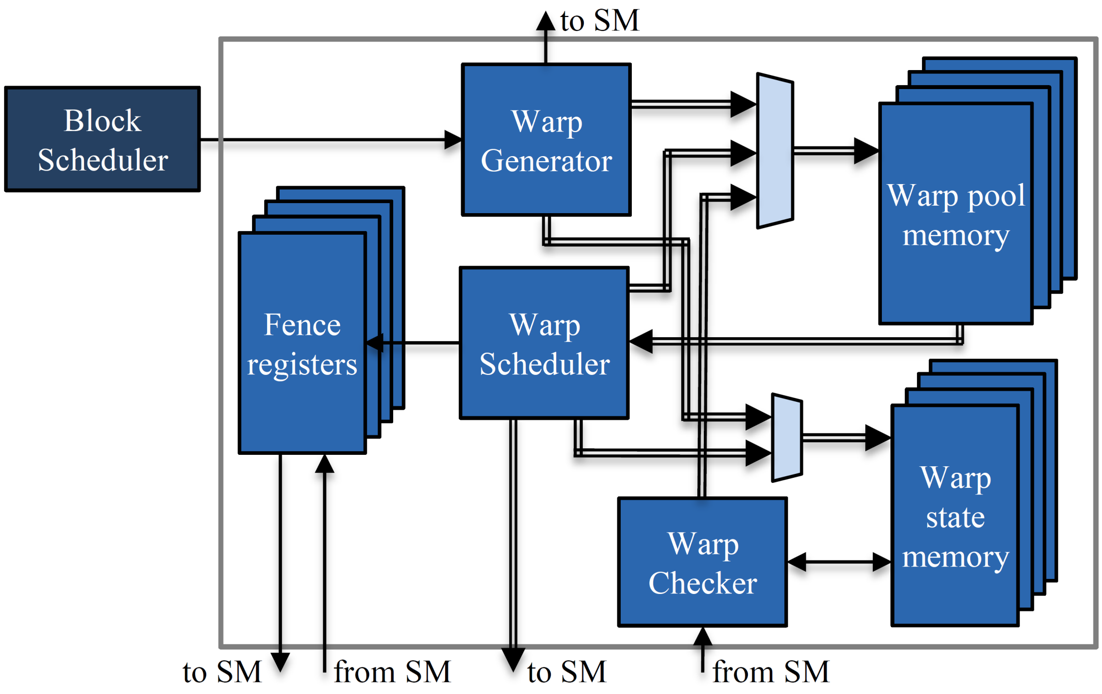


Figure A.1: A general scheme of the scheduler controller in one SM of the FlexGripPlus model.

More in detail, the warp generator defines the total number of warp pool lines to be employed by the application and assigns the entry lines in the pool memory needed to manage them. Moreover, this unit writes the initial value of every warp entry line in the pool memory.

The warp checker modifies the fields in the warp lines after comparing the real state values from the SM's pipeline stages and the predefined values in the warp pool memory. Moreover, this module writes and reads the state memory. The warp scheduler is composed of a state machine and coordinates every warp execution in the SM.

The configuration of FlexGripPlus starts in the block scheduler. This block defines and controls (among other configuration parameters useful for the SM execution) the block and warps dimension for the application. The block scheduler controls and manages the execution of block threads in the SM, adopting a round-robin scheme. It sends the configuration parameters to the warp scheduler controller unit and into one additional local controller (SMP controller) to control the execution of multiple warps.

The SMP controller is an extension of the block scheduler and calculates the addresses per block to address all SM's memory resources. This module is only operative during the configuration and submission phases of a parallel program in the SM.

A.0.2 Convergence management unit

This specific module is also often called Divergence Management Unit (DMU), Branch Divergence Controller, Branch Controller, or Divergence Controller. DMU is devoted to controlling and tracing the operation of multiple paths in the same group of threads. Internally, the DMU evaluates control-flow instructions and uses a stack memory to store relevant information concerning the execution paths. Most DMUs can manage divergences composed of two paths. However, other locations in the stack memory can be employed to manage more than two divergence paths. Thus, the DMU is crucial for the correct operation of an application in the GPU.

The DMU module is located in the SM's Execute stage. It comprises stack memory and a state machine managing each control flow instruction generating any conditional (depending on any predicate flag) or unconditional thread branch operation. Moreover, this module also controls the operations indicating any possible intra-warp divergence and the calling to a specific routine, the return from a routine, and exit conditions. Thus, enabling and disabling paths of divergence. More in detail, the stack stores the starting and ending points of any conditional assessment.

Finally, for the branch unit's correct execution, the G80 architecture supports synchronization barriers that allow the convergence of all threads in a warp and a block. At the microarchitectural level, this generation implies using convergence points, which are employed to define a memory address for the program counter. In case of divergence, all threads reach this instruction address. Then the execution of the parallel program can continue.

A.0.3 Memory hierarchy

FlexGripPlus is mainly based on the operation of the SIMD taxonomy, so each SM uses a large set of data operands to execute in parallel the same instruction on the available functional units. This structure generates bottlenecks and race conditions when accessing operands from the memory system. For this purpose, the GPUs include multiple memory levels to reduce latency. These mechanisms are optimized to process data operands mainly organized as arrays or matrices. In this way, each SM includes multiple data memory resources to optimize the information flow for each thread. These resources are the Register File, the shared memory, the Global (or main) memory, the constant memory, and the local memory. Moreover, some special-purpose memories in the core are devoted to storing the memory addresses (indirect addressing) and predicate registers per thread.

In terms of performance and latency, the Register File, the address register file, the predicate registers, and the shared memory present similar access times for the threads. These are distributed as banks and accessed in parallel. In contrast, the local, constant and global memories are access in the GPU model as a sequence of operations. It is worth noting that the GPU model does not include any caches in the memory hierarchy.

More in detail, the memory hierarchy includes several controllers and arbiters to access every memory resource. Initially, a master memory controller activates a separate memory controller when accessing an operand from that particular resource. In the FlexGripPlus architecture, the controllers are located inside the Read/Issue and the Write-back pipeline stages to perform the operands' load and store, respectively.

When processing a program, the compiler usually selects the best trade-off in terms of performance to locate the data operands using the available memory resources in the SM. In particular, the Register File stores individual operands. The Local memory stores the operands behaving as arrays. Similarly, the constant memory stores constant variables during a program's operation, and the shared memory stores those operands used among the threads in a block. Finally, Global memory is used to locate all input data sources and the program's output results.

The master memory controller decodes the commands coming from the incoming fetched warp instruction. This controller selects the target memory resource and submits a request to the specific memory controller. It is worth noting that both the Read and the Write-back stages can activate up to 3 simultaneous operations on the memories considering the required number of sources or destinations by the instruction. Some modules operate in parallel and determine the target memory locations to perform the reading or writing operations, depending on the source or destination number.

Memory arbiters manage and order access into the target memories. These arbiters organize the memory access for the threads in a warp, considering that up to 32 loads or stores can be generated per warp in parallel.

The Register File is a massive structure composed of 16KB general-purpose registers and located inside of an SM. The scheduler controller divides the Register File among the available SPs and the configured threads in a program kernel. The Register File is one of the most critical units in the operation of a thread in the SM since most instructions require a load or a store from/to memory. Moreover, this unit feeds the execution units with the data operands for each thread. The Register File also stores the indices for memory addressing, the kernel parameters, and the data and address operands during the execution of one warp instruction.

The predicate register file stores the predicate flags after each comparison or logic-arithmetic instructions. When the model is configured with 8 SPs, 512 registers of one-bit size are assigned per SP. These registers are distributed in groups of four registers among the available threads. The four registers store the logical state of the zero (Z), the sign (S), the carry (C), and the overflow (O) flags for each thread. The flags remain constant in the subsequent clock cycles until the execution of a new instruction affects their state.

Finally, the address register file addresses the shared and constant memories with additional indices indirectly. Shared memory is commonly used in programs to optimize performance. It is used to access sectors of data organized as arrays or matrices by multiple threads in a program kernel efficiently. Moreover, the address register file reduces the latency in data used frequently by a kernel. Each of the eight SPs has an associated address register module composed of 512 registers of 32 bit-size holding up to 128 threads. In this way, four registers (A0, A1, A2, and A3) are assigned to each thread.

A.0.4 Functional units

The FlexGripPlus model includes a set of functional units on each SM core. The main feature of the original FlexGrip model is maintained, and the number of cores is configurable before simulation or synthesis. The available functional units in FlexGripPlus include integer cores (INT), floating-point units in single precision format (FP32), and Special Function Units (SFUs) in single precision. The possible configuration in the SM is combinations of (8,8,2), (16,16,2), and (32,32,4) units of INT, FP32, and SFUs modules, respectively.

The integer core is a combinational unit devoted to performing logic, integer arithmetic, and conversion operations for signed and unsigned values in byte (8 bits), half-word (16 bits), and word (32 bits) sizes. Similarly, the Floating point unit is an extension performed in the FlexGripPlus model, and it can execute floating-point operations in the single-precision format (32 bits). The Float multiply-add operation was also included as part of the FP32 module, and this extension also includes the conversion unit (Int-to-Float) and (Float-to-Int).

The SFU extension in the model allows the execution of trigonometric and transcendental operations supported by the CUDA programming environment. Moreover, a range reduction module was included for specific angle reductions. For this purpose, two different SFU modules are available for the model and can be exchanged during the compilation process. The first SFU version is based on a combination of iterative and compacted non-iterative arithmetic circuits describing the operations of the SFU. These compacted non-iterative circuits are commonly employed in accelerators dedicated to mobile applications. The second SFU module is based on a biquadratic non-iterative minimax approximation that allows the execution of all functions on an overlapped modules optimized for precision and area.

A.0.5 Supported instructions

The SASS assembly language is used in FlexGripPlus and it is compatible with codes generated from the CUDA programming environment using the SM_1.0. Tables A.1, A.2, A.3, A.4 and A.5 provides the pnemonics, the formats per instruction of the control-flow, arithmetic and logic, data and memory handling. floating-point and trigonometric instructions, respectively.

The parameter COMP_TYPE in the Tables refers to the comparison type and modifies the state of predicate flags as the effect of an arithmetic or logic operation. Similarly, the COND parameter is related to the conditional execution of an instruction, considering the state of a predicate flag. The g[] and c[0x1][] fields correspond to instructions using operand sources coming from the shared memory and the constant memory, respectively. Detailed information regarding the instructions' op-code formats can be found in the programmer's manual of the FlexGripPlus model.

Table A.1: Control-flow instructions supported in FlexGripPlus.

Mnemonic	**Description**	**Formats**
BRA	Branch	BRA CX.COND Imm BRA Imm
BAR	barrier synchronization	BAR.ARV.WAIT b0, 0xFFF
RET	Return from kernel	RET RET CX.COND
SSY	Set synchronization point	SSY Imm
NOP	No operation	NOP NOP.S
TRAP	Trap interruption	TRAP
CAL	Call to subroutine	CAL.NOINC CAL

Table A.2: Arithmetic and logic instructions in FlexGripPlus.

Mnemonic	**Description**	**Formats**
I2I	Integer to integer conversion	I2I.U32.U16/S16 RZ, RX(L|H) / g[].U16 I2I.U32.S32 RZ, |RX| / -RX I2I.U32.U16.BEXT RZ, RX(L|H) / g[].U8 I2I.S32.S16.BEXT RZ, RX(L|H) / g[].S8
IMUL/	Integer multiplication	IMUL.U16.U16 RZ, RX(L|H) / g[].U16, RY(L|H) IMUL.S16.S16 RZ, RX(L|H) / g[].S16, RY(L|H)
IMUL32/		IMUL32.U16.U16 RZ, RX(L|H)/g[].U16, RY(L|H)
IMUL32I		IMUL32I.U16.U16 RZ, RX(L|H), Imm IMUL32I.S16.S16 RZ, RX(L|H), Imm
SHL	Shift left	SHL RZ, RX, RY / Imm SHL RZ, g [], Imm SHL.U16 RZ(L|H), RX(L|H), Imm
SHR	Shift right	SHR.S32 RZ, RX, RY / Imm SHR.S32 RZ, g [], Im SHR.U16 / S16 RZ(L|H), RX(L|H), Imm SHR RZ, g[], Imm SHR RZ, RX, RY / Imm
IADD/	Integer add	IADD RZ, RX / -RX, RY IADD RZ, g[], RX / -RX IADD RZ, RX, c[0x1][]
IADD32/		IADD32 RZ, RX, RY / -RY IADD32 RZ, g [0x..], RX / -RX IADD32.U16 RZ(L|H), RX(L|H), RY(L|H) /-RY(L|H)
IADD32I		IADD32I RZ, RX / -RX, Imm IADD32I RZ, g[], Imm
IMAD/	Integer multiply and Add	IMAD.U16/ S16 RZ, RX(L|H), RY(L|H), RW IMAD.U16/ S16 RZ, RX(L|H), c[0x1][], RY IMAD. RZ, RX(L|H), c[0x1][], RY
IMAD32/		IMAD32.U16 RZ, RXL|H, RYL|H, RZ
IMAD32I		IMAD32I.U16/ S16 RZ, RX(L|H), Imm, RZ
LOP	Bitwise logical Operation	LOP.AND/OR/XOR/PASS_B RZ, RX/ g[], RY LOP.AND/OR/XOR/PASS_B RZ, RX, c[0x1] [] LOP.U16.AND/OR/XOR/PASS_B RZ(L|H), RX(L|H), RY(L|H)
ISET	Integer comparison	ISET RZ, RX, RY / c[0x1][], COMP_TYPE ISET RZ, g[], RX, COMP_TYPE ISET.S32 RZ, RX, RY / c[0x1][], COMP_TYPE ISET.S32 RZ, g[], RX, COMP_TYPE

Table A.3: Data handling and memory instructions in FlexGripPlus.

Mnemonic	Description	Formats
MVC	Load from constant memory	MVC RX, c [0x1] []
GLD	Load from global memory	GLD.U32|U16|S16|U8|S8 RZ, global14[]
GST	Store to global Memory	GST.U32|U16|S16|U8|S8 global14[], RX
MOV/ MOV32	Move register to register/load from shared memory	MOV RZ, RX / g[] MOV.U16 RZ(L|H), RX(L|H) / g[].(U16|U8) MOV32 RZ, RX / g[] MOV32.U16 RZ(L|H), RX(L|H)
MVI	Move immediate to destination	MVI RX, Imm
R2G	Store from register to shared Memory	R2G.U32.U32 g [], RX R2G.U16.U16 g [], RXL|H R2G.U16.U8 g [], RX
R2A	Move data register to address register	R2A AX, RX
A2R	Move address register to data register	A2R RX, AX
ADA	Movement from address register to address register	ADA Ax, Ax, Offset

Table A.4: Floating Point Unit (FPU) instructions supported in FlexGripPlus.

Mnemonic	Description	Formats
FADD32 / FADD / FADD32I	Floating-point addition	FADD32 Rx, Ry / g[Ax + Imm], Rz FADD.COND Rx (Cx), Ry, -Rz / c[0xX][Imm] FADD32I Rx, Ry, Imm
FMUL / FMUL32 / FMUL32I	Floating-point multiplication	FMUL Rx(Cx.COND), Ry / g [Ax + Imm], Rz / c[0xX][Imm] FMUL32 Rx, Ry / g [Ax+Imm], Rz FMUL32I Rx, Ry, Imm
FMAD / FMAD32 / FMAD32I	Floating-point multiply and addition	FMAD Rx, Ry / - g [Ax+Imm], Rz / c[0xX][Imm], Rw FMAD32I Rx, -Ry, Imm, Rz
F2F	Floating-point conversion	F2F.F32.F32 Rx (CX.COND), -Ry / |Ry|
F2I	Conversion from Floating-point to Integer	F2I.S32.F32.COND Rx, Ry
I2F	Conversion from Integer to Floating-point	I2F.F32.S32/U32 Rx (CX.COND), Ry
FSET	Floating-point set	FSET.C0 o[0x7f] (Cx.COND), Rx / |Rx|, Ry / c[0xX][Imm], COND
RCP / RCP32	Reciprocal value	RCP Ry (Cx.COND), Rx RCP32 Ry, Rx

Table A.5: Special Function Unit (SFU) instructions supported in FlexGripPlus.

Mnemonic	Description	Formats
SIN	Single precision SIN (32 bits)	SIN Rx, Rx
COS	Single precision COS (32 bits)	COS Rx, Rx
RRO	Range Reduction Operator (phase)	RRO Ry, Rx, (SIN/EX2)
EX2	Find the base-2 exponential of a value	EX2 Ry, Rx
RSQ	Reciprocal of the square root in single-precision (32 bits)	RSQ Ry, Rx
LG2	Calculates the Log, base 2, of a value	LG2 Ry, Rx

Bibliography

[1] B. Fleming. “Microcontroller Units in Automobiles [Automotive Electronics]”. In: *IEEE Vehicular Technology Magazine* 6.3 (2011), pp. 4–8. DOI: 10.1109/MVT.2011.941888.

[2] J. P. Trovao. “Trends in Automotive Electronics [Automotive Electronics]”. In: *IEEE Vehicular Technology Magazine* 14.4 (2019), pp. 100–109. DOI: 10.1109/MVT.2019.2939757.

[3] J. C. Knight. “Safety critical systems: challenges and directions”. In: *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*. 2002, pp. 547–550. ISBN: 1-58113-472-X.

[4] H. Ardebili et al. “10 - Trends and challenges”. In: *Encapsulation Technologies for Electronic Applications (Second Edition)*. Second Edition. Materials and Processes for Electronic Applications. William Andrew Publishing, 2019, pp. 431–479. ISBN: 978-0-12-811978-5. DOI: 10.1016/B978-0-12-811978-5.00010-9.

[5] S. Khan et al. “BTI impact on logical gates in nano-scale CMOS technology”. In: *2012 IEEE 15th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*. 2012, pp. 348–353. DOI: 10.1109/DDECS.2012.6219086.

[6] S. Hamdioui et al. “Reliability challenges of real-time systems in forthcoming technology nodes”. In: *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2013, pp. 129–134. DOI: 10.7873/DATE.2013.040.

[7] I. Agbo et al. “Read path degradation analysis in SRAM”. In: *2016 21th IEEE European Test Symposium (ETS)*. 2016, pp. 1–2. DOI: 10.1109/ETS.2016.7519325.

[8] S. Pae et al. “Effect of BTI Degradation on Transistor Variability in Advanced Semiconductor Technologies”. In: *IEEE Transactions on Device and Materials Reliability* 8.3 (2008), pp. 519–525. DOI: 10.1109/TDMR.2008.2002351.

[9] K. Iniewski. *Radiation effects in semiconductors*. CRC press, 2018, pp. 1–431. ISBN: 978-1-4398-2694-2. DOI: 10.1201/9781315217864.

[10] D. Oliveira et al. “Thermal Neutrons: a Possible Threat for Supercomputers and Safety Critical Applications”. In: *2020 IEEE European Test Symposium (ETS)*. 2020, pp. 1–6. DOI: 10.1109/ETS48528.2020.9131597.

[11] H. G. Kerkhoff et al. “Intermittent Resistive Faults in Digital CMOS Circuits”. In: *2015 IEEE 18th International Symposium on Design and Diagnostics of Electronic Circuits Systems*. 2015, pp. 211–216. DOI: 10.1109/DDECS.2015.12.

[12] D. Gil-Tomás et al. “Studying the effects of intermittent faults on a microcontroller”. In: *Microelectronics Reliability* 52.11 (2012), pp. 2837–2846. DOI: 10.1016/j.microrel.2012.06.004.

[13] M. Radetzki et al. “Methods for fault tolerance in networks-on-chip”. In: *ACM Computing Surveys (CSUR)* 46.1 (2013), pp. 1–38. DOI: 10.1145/2522968.2522976.

[14] S. Borkar et al. “Microprocessors in the Era of Terascale Integration”. In: *2007 Design, Automation Test in Europe Conference Exhibition*. 2007, pp. 1–6. DOI: 10.1109/DATE.2007.364597.

[15] D. Shapiro. *NVIDIA DRIVE Xavier, World’s Most Powerful SoC, Brings Dramatic New AI Capabilities*. Nvidia Blog, available online at https://blogs.-nvidia.com/blog/2018/01/07/drive-xavier-processor/, consulted 08/04/2021. Apr. 2018.

[16] Mark White et al. *Scaled CMOS technology reliability users guide*. Tech. rep. Pasadena, CA: Jet Propulsion Laboratory, National Aeronautics and Space . . ., 2010.

[17] M. Psarakis et al. “Microprocessor Software-Based Self-Testing”. In: *IEEE Design Test of Computers* 27.3 (2010), pp. 4–19. DOI: 10.1109/MDT.2010.5.

[18] Nektarios Kranitis et al. “Software-Based Self-Testing of Embedded Processors”. In: *Processor Design: System-on-Chip Computing for ASICs and FPGAs*. Ed. by Jari Nurmi. Dordrecht: Springer Netherlands, 2007, pp. 447–481. ISBN: 978-1-4020-5530-0. DOI: 10.1007/978-1-4020-5530-0_20. URL: https://doi.org/10.1007/978-1-4020-5530-0_20.

[19] K. Andryc et al. “FlexGrip: A soft GPGPU for FPGAs”. In: *2013 International Conference on Field-Programmable Technology (FPT)*. 2013, pp. 230–237. DOI: 10.1109/FPT.2013.6718358.

[20] M. Bojarski et al. *End to End Learning for Self-Driving Cars*. 2016. arXiv: 1604.07316 [cs.CV].

[21] Pavan Kumar Datla Jagannadha et al. “Special Session: In-System-Test (IST) Architecture for NVIDIA Drive-AGX Platforms”. In: *2019 IEEE 37th VLSI Test Symposium (VTS)*. IEEE. 2019, pp. 1–8. DOI: 10.1109/VTS.2019.8758636.

[22] I. S. Olmedo et al. “A Perspective on Safety and Real-Time Issues for GPU Accelerated ADAS”. In: *IECON 2018 - 44th Annual Conference of the IEEE Industrial Electronics Society*. 2018, pp. 4071–4077. DOI: 10.1109/IECON.2018.8591540.

[23] J. Borrego-Carazo et al. “Resource-Constrained Machine Learning for ADAS: A Systematic Review”. In: *IEEE Access* 8 (2020), pp. 40573–40598. DOI: 10.1109/ACCESS.2020.2976513.

[24] W. Shi et al. “Algorithm and hardware implementation for visual perception system in autonomous vehicle: A survey”. In: *Integration* 59 (2017), pp. 148–156. DOI: 10.1016/j.vlsi.2017.07.007.

[25] J. Janai et al. “Computer Vision for Autonomous Vehicles: Problems, Datasets and State of the Art”. In: *Foundations and Trends® in Computer Graphics and Vision* 12.1–3 (2020), pp. 1–308. ISSN: 1572-2740. DOI: 10.1561/0600000079.

[26] R. Hussain et al. “Autonomous Cars: Research Results, Issues, and Future Challenges”. In: *IEEE Communications Surveys Tutorials* 21.2 (2019), pp. 1275–1313. DOI: 10.1109/COMST.2018.2869360.

[27] M. Dumont et al. “Electromagnetic Fault Injection : How Faults Occur”. In: *2019 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*. 2019, pp. 9–16. DOI: 10.1109/FDTC.2019.00010.

[28] C. Constantinescu. “Intermittent faults and effects on reliability of integrated circuits”. In: *2008 Annual Reliability and Maintainability Symposium*. 2008, pp. 370–374. DOI: 10.1109/RAMS.2008.4925824.

[29] C. Sandionigi et al. “When processors get old: Evaluation of BTI and HCI effects on performance and reliability”. In: *2013 IEEE 19th International On-Line Testing Symposium (IOLTS)*. 2013, pp. 185–186. DOI: 10.1109/IOLTS.2013.6604076.

[30] M. J. Mišić et al. “Evolution and trends in GPU computing”. In: *2012 Proceedings of the 35th International Convention MIPRO*. 2012, pp. 289–294. ISBN: 978-1-4673-2577-6.

[31] C. McClanahan. “History and evolution of gpu architecture, A Survey Paper”. In: (2010). Georgia Institute of Technology, pp. 1–6. URL: http://www.mathcs.emory.edu/~cheung/Courses/355/Syllabus/94-CUDA/Docs/.

[32] T. Akenine-Möller et al. *Real-Time Rendering 4th Edition*. A K Peters/CRC Press, 2019, p. 1200. ISBN: 978-1-3153-5986-1.

[33] P. K. Das et al. “History and evolution of GPU architecture”. In: *Emerging Research Surrounding Power Consumption and Performance Issues in Utility Computing*. IGI Global, 2016, pp. 109–135. ISBN: 978-1-4666-8853-7.

[34] J. Nickolls et al. "The GPU Computing Era". In: vol. 30. 2. 2010, pp. 56–69. DOI: 10.1109/MM.2010.41.

[35] T. M. Aamodt et al. "General-purpose graphics processor architectures". In: *Synthesis Lectures on Computer Architecture* 13.2 (2018), pp. 1–140.

[36] M. J. Flynn. "Some Computer Organizations and Their Effectiveness". In: *IEEE Transactions on Computers* C-21.9 (1972), pp. 948–960. DOI: 10.1109/TC.1972.5009071.

[37] M. Mantor. "AMD Radeon™ HD 7970 with graphics core next (GCN) architecture". In: *2012 IEEE Hot Chips 24 Symposium (HCS)*. 2012, pp. 1–35. DOI: 10.1109/HOTCHIPS.2012.7476485.

[38] S. Markidis et al. "NVIDIA Tensor Core Programmability, Performance Precision". In: *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2018, pp. 522–531. DOI: 10.1109/IPDPSW.2018.00091.

[39] M. A. Raihan et al. "Modeling Deep Learning Accelerator Enabled GPUs". In: *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2019, pp. 79–92. DOI: 10.1109/ISPASS.2019.00016.

[40] B. W. Coon et al. *Thread group scheduler for computing on a parallel thread processor*. Nvidia Corporation, US Patent 8,732,713. May 2014.

[41] D. Acocella et al. *Concurrent access of data elements stored across multiple banks in a shared memory resource*. Nvidia Corporation, US Patent 7,750,915. July 2010.

[42] E. Lindholm and othnd others. "NVIDIA Tesla: A Unified Graphics and Computing Architecture". In: *IEEE Micro* 28.2 (2008), pp. 39–55. DOI: 10.1109/MM.2008.31.

[43] M. Daga et al. "Architecture-aware mapping and optimization on a 1600-core gpu". In: *2011 IEEE 17th International Conference on Parallel and Distributed Systems*. IEEE. 2011, pp. 316–323. DOI: 10.1109/ICPADS.2011.29.

[44] NVIDIA Corporation. "NVIDIA VOLTA V100 ARCHITECTURE". In: *White paper* (2018).

[45] N. Juffa. *Pipelined integer division using floating-point reciprocal*. Nvidia Corporation, US Patent 8,140,608. Mar. 2012.

[46] S. Hilker. *Floating point multiply accumulator multi-precision mantissa aligner*. Advanced Micro Devices Inc., US Patent 9,141,337. Sept. 2015.

[47] S. H. Dhong et al. *Power saving in FPU with gated power based on opcodes and data*. International Business Machines Corporation, US Patent 7,137,021. Nov. 2006.

[48] S. Oberman et al. *Fused multiply-add functional unit*. Nvidia Corporation, US Patent 8,106,914. Jan. 2012.

[49] M. Siu et al. *Multipurpose functional unit with combined integer and floating-point multiply-add pipeline*. Nvidia Corporation, US Patent App. 10/986,531. May 2006.

[50] M. Siu et al. *Multipurpose functional unit with multiply-add and format conversion pipeline*. Nvidia Corporation, US Patent 7,428,566. Sept. 2008.

[51] D. C. Tannenbaum et al. *Approach to power reduction in floating-point operations*. Nvidia Corporation, US Patent 9,829,956. Nov. 2017.

[52] A. Li et al. "SFU-Driven Transparent Approximation Acceleration on GPUs". In: *Proceedings of the 2016 International Conference on Supercomputing*. ICS '16. Istanbul, Turkey: Association for Computing Machinery, 2016. ISBN: 9781450343619. DOI: 10.1145/2925426.2926255.

[53] P. K. Meher et al. "50 Years of CORDIC: Algorithms, Architectures, and Applications". In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 56.9 (2009), pp. 1893–1907. DOI: 10.1109/TCSI.2009.2025803.

[54] J. -. Pineiro et al. "High-speed function approximation using a minimax quadratic interpolator". In: *IEEE Transactions on Computers* 54.3 (2005), pp. 304–318. DOI: 10.1109/TC.2005.52.

[55] S. F. Oberman et al. "A high-performance area-efficient multifunction interpolator". In: *17th IEEE Symposium on Computer Arithmetic (ARITH'05)*. 2005, pp. 272–279. DOI: 10.1109/ARITH.2005.7.

[56] R. C. Johnson et al. *Efficient Predicated Execution for Parallel Processors*. Nvidia Corporation, US Patent App. 12/891,629. Mar. 2011.

[57] E. R. Komen. "Efficient parallelism using indirect addressing in SIMD processor arrays". In: *Pattern recognition letters* 12.5 (1991), pp. 279–289. DOI: 10.1016/0167-8655(91)90411-E.

[58] J. R. Nickolls et al. *Single interconnect providing read and write access to a memory shared by concurrent threads*. Nvidia Corporation, US Patent 7,680,988. Mar. 2010.

[59] N. B. Lakshminarayana et al. "Effect of instruction fetch and memory scheduling on gpu performance". In: *Workshop on Language, Compiler, and Architecture Support for GPGPU*. Vol. 88. Georgia Institute of Technology. 2010.

[60] S. Lee et al. “CAWS: Criticality-aware warp scheduling for GPGPU workloads”. In: *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*. 2014, pp. 175–186. DOI: 10.1145/2628071.2628107.

[61] S. Di Carlo et al. “An improved fault mitigation strategy for CUDA Fermi GPUs”. In: *Proc. Dependable GPU Comput. workshop*. 2014, pp. 1–6. URL: http://porto.polito.it/2571949/.

[62] C. M. Wittenbrink, E. Kilgariff, and A. Prabhu. “Fermi GF100 GPU Architecture”. In: *IEEE Micro* 31.2 (2011), pp. 50–59. DOI: 10.1109/MM.2011.24.

[63] A. B. Hayes et al. “Decoding CUDA Binary”. In: *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2019, pp. 229–241. DOI: 10.1109/CGO.2019.8661186.

[64] ISO ISO. “ISO 26262 Road Vehicles-Function Safety-Part 5: Product development at the hardware level”. In: *International Standardization Organization Std* (2018).

[65] Y.C. Chang et al. “Assessing automotive functional safety microprocessor with ISO 26262 hardware requirements”. In: *Technical Papers of 2014 International Symposium on VLSI Design, Automation and Test*. 2014, pp. 1–4. DOI: 10.1109/VLSI-DAT.2014.6834876.

[66] Alkahtani M. et al. “A decision support system based on ontology and data mining to improve design using warranty data”. In: *Computers Industrial Engineering* 128 (2019), pp. 1027–1039. ISSN: 0360-8352. DOI: https://doi.org/10.1016/j.cie.2018.04.033. URL: https://www.sciencedirect.com/science/article/pii/S0360835218301694.

[67] A. Frigerio et al. “Component-Level ASIL Decomposition for Automotive Architectures”. In: *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. 2019, pp. 62–69. DOI: 10.1109/DSN-W.2019.00021.

[68] D.D. Ward et al. “The uses and abuses of ASIL decomposition in ISO 26262”. In: *7th IET International Conference on System Safety, incorporating the Cyber Security Conference 2012*. 2012, pp. 1–6. DOI: 10.1049/cp.2012.1523.

[69] D. A. Gonçalves de Oliveira et al. “GPGPUs ECC efficiency and efficacy”. In: *2014 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. 2014, pp. 209–215. DOI: 10.1109/DFT.2014.6962085.

[70] J. Chen, S. Li, and Z. Chen. “GPU-ABFT: Optimizing Algorithm-Based Fault Tolerance for Heterogeneous Systems with GPUs”. In: *2016 IEEE International Conference on Networking, Architecture and Storage (NAS)*. 2016, pp. 1–2. DOI: 10.1109/NAS.2016.7549404.

[71] D. Sabena et al. “On the evaluation of soft-errors detection techniques for GPGPUs”. In: *2013 8th IEEE Design and Test Symposium*. 2013, pp. 1–6. DOI: 10.1109/IDT.2013.6727092.

[72] M.M. Gonçalves et al. “Improving GPU register file reliability with a comprehensive ISA extension”. In: *Microelectronics Reliability* 114 (2020). 31st European Symposium on Reliability of Electron Devices, Failure Physics and Analysis, ESREF 2020, p. 113768. ISSN: 0026-2714. DOI: https://doi.org/10.1016/j.microrel.2020.113768.

[73] F. F. Dos Santos et al. “Reduced Precision DWC: an Efficient Hardening Strategy for Mixed-Precision Architectures”. In: *IEEE Transactions on Computers* (2021), pp. 1–1. DOI: 10.1109/TC.2021.3058872.

[74] I. Polian et al. “Selective Hardening: Toward Cost-Effective Error Tolerance”. In: *IEEE Design and Test of Computers* 28.3 (2011), pp. 54–63. DOI: 10.1109/MDT.2010.120.

[75] N. Maruyama et al. “Software-based ECC for GPUs”. In: *Symposium on Application Accelerators in High Performance Computing (SAAHPC’09)*. Vol. 107. 2009.

[76] L. L. Pilla et al. “Software-Based Hardening Strategies for Neutron Sensitive FFT Algorithms on GPUs”. In: *IEEE Transactions on Nuclear Science* 61.4 (2014), pp. 1874–1880. DOI: 10.1109/TNS.2014.2301768.

[77] C. Bao et al. “Algorithm-based fault tolerance for discrete wavelet transform implemented on GPUs”. In: *Journal of Systems Architecture* 108 (2020), p. 101823. ISSN: 1383-7621. DOI: https://doi.org/10.1016/j.sysarc.2020.101823.

[78] D. A. G. Gonçalves de Oliveira et al. “Evaluation and Mitigation of Radiation-Induced Soft Errors in Graphics Processing Units”. In: *IEEE Transactions on Computers* 65.3 (2016), pp. 791–804. DOI: 10.1109/TC.2015.2444855.

[79] P. Rech et al. “Experimental evaluation of GPUs radiation sensitivity and algorithm-based fault tolerance efficiency”. In: *2013 IEEE 19th International On-Line Testing Symposium (IOLTS)*. 2013, pp. 244–247. DOI: 10.1109/IOLTS.2013.6604091.

[80] P. Rech et al. “An Efficient and Experimentally Tuned Software-Based Hardening Strategy for Matrix Multiplication on GPUs”. In: *IEEE Transactions on Nuclear Science* 60.4 (2013), pp. 2797–2804. DOI: 10.1109/TNS.2013.2252625.

[81] H. Wunderlich et al. “Efficacy and efficiency of algorithm-based fault-tolerance on GPUs”. In: *2013 IEEE 19th International On-Line Testing Symposium (IOLTS)*. 2013, pp. 240–243. DOI: 10.1109/IOLTS.2013.6604090.

[82] S. Di Carlo et al. “A software-based self test of CUDA Fermi GPUs”. In: *2013 18th IEEE European Test Symposium (ETS)*. 2013, pp. 1–6. DOI: 10.1109/ETS.2013.6569353.

[83] B. Fang et al. “GPU-Qin: A methodology for evaluating the error resilience of GPGPU applications”. In: *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2014, pp. 221–230. DOI: 10.1109/ISPASS.2014.6844486.

[84] G. Malhotra et al. “GpuTejas: A parallel simulator for GPU architectures”. In: *2014 21st International Conference on High Performance Computing (HiPC)*. 2014, pp. 1–10. DOI: 10.1109/HiPC.2014.7116897.

[85] R. Ubal et al. “Multi2Sim: A simulation framework for CPU-GPU computing”. In: *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2012, pp. 335–344. ISBN: 978-1-4503-1182-3.

[86] A. Bakhoda et al. “Analyzing CUDA workloads using a detailed GPU simulator”. In: *2009 IEEE International Symposium on Performance Analysis of Systems and Software*. 2009, pp. 163–174. DOI: 10.1109/ISPASS.2009.4919648.

[87] S. Collange et al. “Barra: A Parallel Functional Simulator for GPGPU”. In: *2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*. 2010, pp. 351–360. DOI: 10.1109/MASCOTS.2010.43.

[88] V. M. del Barrio et al. “ATTILA: a cycle-level execution-driven simulator for modern GPU architectures”. In: *2006 IEEE International Symposium on Performance Analysis of Systems and Software*. 2006, pp. 231–241. DOI: 10.1109/ISPASS.2006.1620807.

[89] J. Power et al. “gem5-gpu: A Heterogeneous CPU-GPU Simulator”. In: *IEEE Computer Architecture Letters* 14.1 (2015), pp. 34–36. DOI: 10.1109/LCA.2014.2299539.

[90] S. Tselonis et al. “GUFI: A framework for GPUs reliability assessment”. In: *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2016, pp. 90–100. DOI: 10.1109/ISPASS.2016.7482077.

[91] R. Balasubramanian et al. “Enabling GPGPU Low-Level Hardware Explorations with MIAOW: An Open-Source RTL Implementation of a GPGPU”. In: *ACM Trans. Archit. Code Optim.* 12.2 (June 2015). ISSN: 1544-3566. DOI: 10.1145/2764908.

[92] M. Amiri et al. “FPGA-based soft-core processors for image processing applications”. In: *Journal of signal processing systems* 87.1 (2017), pp. 139–156. DOI: 10.1007/s11265-016-1185-7.

[93] F. M. Siddiqui et al. “IPPro: FPGA based image processing processor”. In: *2014 IEEE Workshop on Signal Processing Systems (SiPS)*. 2014, pp. 1–6. DOI: 10.1109/SiPS.2014.6986057.

[94] M. A. Kadi et al. “General-purpose computing with soft GPUs on FPGAs”. In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 11.1 (2018), pp. 1–22. DOI: 10.1145/3173548.

[95] C. Collange. *Simty: a Synthesizable General-Purpose SIMT Processor*. Research Report RR-8944. Inria Rennes Bretagne Atlantique, Aug. 2016. URL: https://hal.inria.fr/hal-01351689.

[96] J. Bush et al. “Nyami: a synthesizable GPU architectural model for general-purpose and graphics-specific workloads”. In: *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2015, pp. 173–182. DOI: 10.1109/ISPASS.2015.7095803.

[97] J. E. R. Condia et al. “FlexGripPlus: An improved GPGPU model to support reliability analysis”. In: *Microelectronics Reliability* 109 (2020), 1136 60. ISSN: 0026-2714. DOI: 10.1016/j.microrel.2020.113660.

[98] B. Du et al. “An extended model to support detailed GPGPU reliability analysis”. In: *2019 14th International Conference on Design Technology of Integrated Systems In Nanoscale Era (DTIS)*. 2019, pp. 1–6. DOI: 10.1109/DTIS.2019.8735047.

[99] J. Knudsen. “Nangate 45nm open cell library”. In: *CDNLive, EMEA* (2008).

[100] M. Martins et al. “Open Cell Library in 15nm FreePDK Technology”. In: *Proceedings of the 2015 Symposium on International Symposium on Physical Design*. ISPD ’15. 2015, pp. 171–178. ISBN: 9781450333993. DOI: 10.1145/2717764.2717783.

[101] J. E. R. Condia et al. “Design and Verification of an open-source SFU model for GPGPUs”. In: *2020 17th Biennial Baltic Electronics Conference (BEC)*. 2020, pp. 1–6. DOI: 10.1109/BEC49624.2020.9276748.

[102] S. K. S. Hari et al. “SASSIFI: An architecture-level fault injection tool for GPU application resilience evaluation”. In: *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2017, pp. 249–258. DOI: 10.1109/ISPASS.2017.7975296.

[103] Q. Lu et al. “LLFI: An Intermediate Code-Level Fault Injection Tool for Hardware Faults”. In: *2015 IEEE International Conference on Software Quality, Reliability and Security*. 2015. DOI: 10.1109/QRS.2015.13.

[104] F. G. Previlon et al. “PCFI: Program Counter Guided Fault Injection for Accelerating GPU Reliability Assessment”. In: *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2019, pp. 308–311. DOI: 10.23919/DATE.2019.8714781.

[105] A. Vallero et al. “Multi-faceted microarchitecture level reliability characterization for NVIDIA and AMD GPUs”. In: *2018 IEEE 36th VLSI Test Symposium (VTS)*. 2018, pp. 1–6. DOI: 10.1109/VTS.2018.8368665.

[106] S. Azimi et al. “Evaluation of transient errors in GPGPUs for safety critical applications: An effective simulation-based fault injection environment”. In: *Journal of Systems Architecture* 75 (2017), pp. 95–106. ISSN: 1383-7621. DOI: https://doi.org/10.1016/j.sysarc.2017.01.009.

[107] D. A. G. Gonçalves de Oliveira et al. “Evaluation and Mitigation of Radiation-Induced Soft Errors in Graphics Processing Units”. In: *IEEE Transactions on Computers* 65.3 (2016). DOI: 10.1109/TC.2015.2444855.

[108] P. Rech et al. “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*. 2014, pp. 455–466. DOI: 10.1109/DSN.2014.49.

[109] R. L. Rech et al. “Impact of Layers Selective Approximation on CNNs Reliability and Performance”. In: *2020 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. 2020, pp. 1–4. DOI: 10.1109/DFT50435.2020.9250821.

[110] D. A. G. Oliveira et al. “Radiation Sensitivity of High Performance Computing Applications on Kepler-Based GPGPUs”. In: *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. 2014, pp. 732–737. DOI: 10.1109/DSN.2014.75.

[111] P. Rech et al. “Neutron radiation test of graphic processing units”. In: *2012 IEEE 18th International On-Line Testing Symposium (IOLTS)*. 2012, pp. 55–60. DOI: 10.1109/IOLTS.2012.6313841.

[112] D. Sabena et al. “Evaluating the radiation sensitivity of GPGPU caches: New algorithms and experimental results”. In: *Microelectronics Reliability* 54.11 (2014), pp. 2621–2628. ISSN: 0026-2714. DOI: https://doi.org/10.1016/j.microrel.2014.05.001.

[113] W. Nedel et al. “Evaluating the effects of single event upsets in soft-core GPGPUs”. In: *2016 17th Latin-American Test Symposium (LATS)*. 2016, pp. 93–98. DOI: 10.1109/LATW.2016.7483346.

[114] L. L. Pilla et al. “Memory Access Time and Input Size Effects on Parallel Processors Reliability”. In: *IEEE Transactions on Nuclear Science* 62.6 (2015), pp. 2627–2634. DOI: 10.1109/TNS.2015.2496381.

[115] W. Nedel et al. “Implementation and experimental evaluation of a CUDA core under single event effects”. In: *2014 15th Latin American Test Workshop - LATW*. 2014, pp. 1–4. DOI: 10.1109/LATW.2014.6841913.

[116] M. M. Gonçalves et al. “Investigating Floating-Point Implementations in a Softcore GPU under Radiation-Induced Faults”. In: *2020 27th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*. 2020, pp. 1–4. DOI: 10.1109/ICECS49266.2020.9294939.

[117] D. Tiwari et al. “Understanding GPU errors on large-scale HPC systems and the implications for system design and operation”. In: *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. 2015, pp. 331–342. DOI: 10.1109/HPCA.2015.7056044.

[118] C. Lunardi et al. “On the Efficacy of ECC and the Benefits of FinFET Transistor Layout for GPU Reliability”. In: *IEEE Transactions on Nuclear Science* 65.8 (2018), pp. 1843–1850. DOI: 10.1109/TNS.2018.2823786.

[119] F. F. d. Santos et al. “Analyzing and Increasing the Reliability of Convolutional Neural Networks on GPUs”. In: *IEEE Transactions on Reliability* 68.2 (2019), pp. 663–677. DOI: 10.1109/TR.2018.2878387.

[120] P. M. Basso et al. “Impact of Tensor Cores and Mixed Precision on the Reliability of Matrix Multiplication in GPUs”. In: *IEEE Transactions on Nuclear Science* 67.7 (2020), pp. 1560–1565. DOI: 10.1109/TNS.2020.2977583.

[121] F. Fernandes dos Santos et al. “Evaluation and Mitigation of Soft-Errors in Neural Network-Based Object Detection in Three GPU Architectures”. In: *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. 2017, pp. 169–176. DOI: 10.1109/DSN-W.2017.47.

[122] J. Tan et al. “Modeling and characterizing GPGPU reliability in the presence of soft errors”. In: *Parallel Computing* 39.9 (2013). Novel On-Chip Parallel Architectures and Software Support, pp. 520–532. ISSN: 0167-8191. DOI: https://doi.org/10.1016/j.parco.2013.01.001.

[123] D. M. Hiemstra et al. “Single Event Effect Evaluation of the Jetson AGX Xavier Module Using Proton Irradiation”. In: *2020 IEEE Radiation Effects Data Workshop (in conjunction with 2020 NSREC)*. 2020, pp. 1–4. DOI: 10.1109/REDW51883.2020.9325840.

[124] D. Sabena et al. “Reliability Evaluation of Embedded GPGPUs for Safety Critical Applications”. In: *IEEE Transactions on Nuclear Science* 61.6 (2014), pp. 3123–3129. DOI: 10.1109/TNS.2014.2363358.

[125] R. L. Davidson et al. “Error Resilient GPU Accelerated Image Processing for Space Applications”. In: *IEEE Transactions on Parallel and Distributed Systems* 29.9 (2018), pp. 1990–2003. DOI: 10.1109/TPDS.2018.2812853.

[126] B. Du et al. “On the evaluation of SEU effects in GPGPUs”. In: *2019 IEEE Latin American Test Symposium (LATS)*. 2019, pp. 1–6. DOI: 10.1109/LATW.2019.8704643.

[127] J. Guthoff et al. “Combining software-implemented and simulation-based fault injection into a single fault injection method”. In: *Twenty-Fifth International Symposium on Fault-Tolerant Computing. Digest of Papers*. 1995, pp. 196–206. DOI: 10.1109/FTCS.1995.466978.

[128] H. Ziade et al. “A survey on fault injection techniques”. In: *International Arab Journal of Information Technology* Vol. 1, No. 2, July (2004), pp. 171–186.

[129] D. Alexandrescu. “Circuit and System Level Single-Event Effects Modeling and Simulation”. In: *Soft Errors in Modern Electronic Systems*. Ed. by Michael Nicolaidis. Springer US, 2011, pp. 103–140. ISBN: 978-1-4419-6993-4. DOI: 10.1007/978-1-4419-6993-4_5.

[130] J. W. Cooley et al. “The Fast Fourier Transform and Its Applications”. In: *IEEE Transactions on Education* 12.1 (1969), pp. 27–34. DOI: 10.1109/TE.1969.4320436.

[131] S. S. Mukherjee et al. “A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor”. In: *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36*. 2003, pp. 29–40. DOI: 10.1109/MICRO.2003.1253181.

[132] P. Rech et al. “Threads Distribution Effects on Graphics Processing Units Neutron Sensitivity”. In: *IEEE Transactions on Nuclear Science* 60.6 (2013), pp. 4220–4225. DOI: 10.1109/TNS.2013.2286970.

[133] Nathan DeBardeleben et al. “GPU Behavior on a Large HPC Cluster”. In: *Euro-Par 2013: Parallel Processing Workshops*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014. ISBN: 978-3-642-54420-0. DOI: 10.1007/978-3-642-54420-0_66.

[134] D. A. Gonçalves de Oliveira et al. “Radiation-Induced Error Criticality in Modern HPC Parallel Accelerators”. In: *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2017. DOI: 10.1109/HPCA.2017.41.

[135] F. F. d. Santos et al. “Analyzing and Increasing the Reliability of Convolutional Neural Networks on GPUs”. In: *IEEE Transactions on Reliability* 68.2 (June 2019). ISSN: 1558-1721. DOI: 10.1109/TR.2018.2878387.

[136] R. Balasubramanian et al. “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)*. 2014. DOI: 10.1109/HPCA.2014.6835976.

[137] J. E. R. Condia et al. “Combining Architectural Simulation and Software Fault Injection for a Fast andAccurate CNNs Reliability Evaluation on GPUs”. In: *2021 IEEE 39th VLSI Test Symposium (VTS), Virtual Event*. 2021, "in press", pp. 1–6.

[138] Y. LeCun et al. “LeNet-5, convolutional neural networks”. In: *URL: http:// yann. lecun. com/exdb/lenet* 20.5 (2015).

[139] F. G. Previlon et al. “A Comprehensive Evaluation of the Effects of Input Data on the Resilience of GPU Applications”. In: *2019 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. 2019. DOI: 10.1109/DFT.2019.8875269.

[140] V. Sridharan et al. “Eliminating microarchitectural dependency from Architectural Vulnerability”. In: *2009 IEEE 15th International Symposium on High Performance Computer Architecture*. 2009. DOI: 10.1109/HPCA.2009.4798243.

[141] G. Li et al. “Understanding Error Propagation in Deep Learning Neural Network (DNN) Accelerators and Applications”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’17. Association for Computing Machinery, 2017. ISBN: 9781450351140. DOI: 10.1145/3126908.3126964.

[142] S. S. Mukherjee et al. “A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor”. In: *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36*. 2003. DOI: 10.1109/MICRO.2003.1253181.

[143] M. Dimitrov et al. “Understanding Software Approaches for GPGPU Reliability”. In: *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*. GPGPU-2. 2009, pp. 94–104. ISBN: 9781605585178. DOI: 10.1145/1513895.1513907.

[144] H. Jeon et al. “Warped-DMR: Light-weight Error Detection for GPGPU”. In: *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. 2012, pp. 37–47. DOI: 10.1109/MICRO.2012.13.

[145] M. Abdel-Majeed et al. "Warped-RE: Low-Cost Error Detection and Correction in GPUs". In: *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. 2015, pp. 331–342. DOI: 10.1109/DSN.2015.55.

[146] A. Mahmoud et al. "Optimizing Software-Directed Instruction Replication for GPU Error Detection". In: *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. 2018, pp. 842–854. DOI: 10.1109/SC.2018.00070.

[147] J. Tan et al. "RISE: Improving the streaming processors reliability against soft errors in GPGPUs". In: *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2012, pp. 191–200. ISBN: 978-1-4503-1182-3.

[148] H. Kim et al. "Compiler-Directed Soft Error Resilience for Lightweight GPU Register File Protection". In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2020. Association for Computing Machinery, 2020, pp. 989–1004. ISBN: 9781450376136. DOI: 10.1145/3385412.3386033.

[149] L. Bautista Gomez et al. "GPGPUs: How to combine high computational power with high reliability". In: *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2014, pp. 1–9. DOI: 10.7873/DATE.2014.354.

[150] P. Bernardi et al. "Development Flow for On-Line Core Self-Test of Automotive Microcontrollers". In: *IEEE Transactions on Computers* 65.3 (2016), pp. 744–754. DOI: 10.1109/TC.2015.2498546.

[151] A. Riefert et al. "On the automatic generation of SBST test programs for in-field test". In: *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2015, pp. 1186–1191. DOI: 10.7873/DATE.2015.0271.

[152] S. Mittal et al. "A Survey of Techniques for Modeling and Improving Reliability of Computing Systems". In: *IEEE Transactions on Parallel and Distributed Systems* 27.4 (2016), pp. 1226–1238. DOI: 10.1109/TPDS.2015.2426179.

[153] N. Farazmand et al. "Statistical fault injection-based AVF analysis of a GPU architecture". In: *Proceedings of SELSE 12*. 2012, pp. 1–6.

[154] D. Defour et al. "A software scheduling solution to avoid corrupted units on GPUs". In: *Journal of Parallel and Distributed Computing* 90-91 (2016), pp. 1–8. ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2016.01.001.

[155] ST Microelectronics. *AN3307 Application note Guidelines for obtaining IEC 60335 Class B certification for any STM32 application.* Microelectronics, ST, 2016.

[156] Infineon Technologies. *Infineon Self-test libs. C.* https://www.hitex.com/tools-components/software-components/selftest-libraries-safety-libs. Accessed: 05/03/2021.

[157] Microchip Technology Inc. *DS52076A 16-bit CPU Self-Test Library User's Guide.* Microchip Technology, 2012.

[158] ARM Technologies. *ARM Technologies Self-test libs. C.* https://developer.arm.com/technologies/functional-safety. Accessed: 15/11/2018.

[159] *Renesas Technology Self-test libs. C.* https://www.renesas.com/en-eu/products/synergy/software/add-ons.html#read. Accessed: 15/11/2018.

[160] R. Leveugle et al. "Statistical fault injection: Quantified error and confidence". In: *2009 Design, Automation Test in Europe Conference Exhibition.* 2009, pp. 502–506. DOI: 10.1109/DATE.2009.5090716.

[161] S. Di Carlo et al. "Models in Memory Testing". In: *Models in Hardware Testing: Lecture Notes of the Forum in Honor of Christian Landrault.* Dordrecht, 2010, pp. 157–185. ISBN: 978-90-481-3282-9. DOI: 10.1007/978-90-481-3282-9_6.

[162] C. M. Wittenbrink et al. "Fermi GF100 GPU architecture". In: *IEEE Micro* 31.2 (2011), pp. 50–59. DOI: 10.1109/MM.2011.24.

[163] G. Theodorou et al. "A Software-Based Self-Test methodology for on-line testing of processor caches". In: *2011 IEEE International Test Conference.* 2011, pp. 1–10. DOI: 10.1109/TEST.2011.6139154.

[164] J. Hudec. "An Efficient Adaptive Method of Software-Based Self Test Generation for RISC Processors". In: *2015 4th Eastern European Regional Conference on the Engineering of Computer Based Systems.* 2015, pp. 119–121. DOI: 10.1109/ECBS-EERC.2015.26.

[165] S. Di Carlo et al. "Increasing the robustness of CUDA Fermi GPU-based systems". In: *2013 IEEE 19th International On-Line Testing Symposium (IOLTS).* 2013, pp. 234–235. DOI: 10.1109/IOLTS.2013.6604088.

[166] B. Du et al. "About the functional test of the GPGPU scheduler". In: *2018 IEEE 24th International Symposium on On-Line Testing And Robust System Design (IOLTS).* 2018, pp. 85–90. DOI: 10.1109/IOLTS.2018.8474174.

[167] M. M. Gonçalves et al. "A low-level software-based fault tolerance approach to detect SEUs in GPUs' register files". In: *Microelectronics Reliability* 76-77 (2017), pp. 665–669. ISSN: 0026-2714. DOI: https://doi.org/10.1016/j.microrel.2017.07.035.

[168] M. M. Gonçalves et al. “Evaluating the reliability of a GPU pipeline to SEU and the impacts of software-based and hardware-based fault tolerance techniques”. In: *Microelectronics Reliability* 88-90 (2018). 29th European Symposium on Reliability of Electron Devices, Failure Physics and Analysis (ESREF 2018), pp. 931–935. ISSN: 0026-2714. DOI: 10.1016/j.microrel.2018.07.007.

[169] S. Gurumurthy et al. “Automatic generation of instruction sequences targeting hard-to-detect structural faults in a processor”. In: *2006 IEEE International Test Conference*. 2006, pp. 1–9. DOI: 10.1109/TEST.2006.297676.

[170] R. Cantoro et al. “About on-line functionally untestable fault identification in microprocessor cores for safety-critical applications”. In: *2018 IEEE 19th Latin-American Test Symposium (LATS)*. 2018, pp. 1–6. DOI: 10.1109/LATW.2018.8349679.

[171] P. Bernardi et al. “Fault grading of software-based self-test procedures for dependable automotive applications”. In: *2011 Design, Automation Test in Europe*. 2011, pp. 1–2. DOI: 10.1109/DATE.2011.5763092.

[172] M. Abdel-Majeed et al. “Low overhead online periodic testing for GPGPUs”. In: *Integration* 62 (2018), pp. 362–370. ISSN: 0167-9260. DOI: https://doi.org/10.1016/j.vlsi.2018.04.015.

[173] J. E. R. Condia and M. 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)*. 2019, pp. 97–102. DOI: 10.1109/IOLTS.2019.8854463.

[174] J. E. R. Condia and 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)*. 2020, pp. 1–6. DOI: 10.1109/IOLTS50870.2020.9159711.

[175] S. Di Carlo et al. “An On-Line Testing Technique for the Scheduler Memory of a GPGPU”. In: *IEEE Access* 8 (2020), pp. 16893–16912. DOI: 10.1109/ACCESS.2020.2968139.

[176] S. Di Carlo et al. “On the in-field test of the GPGPU scheduler memory”. In: *2019 IEEE 22nd International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*. 2019, pp. 1–6. DOI: 10.1109/DDECS.2019.8724672.

[177] J. E. R. Condia and 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)*. 2020, pp. 153–158. DOI: 10.1109/VLSI-SOC46417.2020.9344088.

[178] A. Apostolakis et al. “Exploiting Thread-Level Parallelism in Functional Self-Testing of CMT Processors”. In: *2009 14th IEEE European Test Symposium*. 2009, pp. 33–38. DOI: 10.1109/ETS.2009.31.

[179] J. Guerrero-Balaguera et al. “On the Functional Test of Special Function Units in GPUs”. In: *2021 24th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*. 2021, pp. 81–86. DOI: 10.1109/DDECS52668.2021.9417025.

[180] J. E. R. Condia et al. “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)*. 2020, pp. 1–6. DOI: 10.1109/DDECS50862.2020.9095665.

[181] S. Alcaide et al. “High-Integrity GPU Designs for Critical Real-Time Automotive Systems”. In: *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2019, pp. 824–829. DOI: 10.23919/DATE.2019.8715177.

[182] Nvidia Corporation. “NVIDIA Announces World’s First Functionally SafeAI Self-Driving Platform”. In: Materials and Processes for Electronic Applications. 2016. URL: https://nvidianews.nvidia.com/news/nvidia-announces-worlds-first-functionally-safe-ai-self-driving-platform.

[183] M. Gonçalves et al. “Selective Fault Tolerance for Register Files of Graphics Processing Units”. In: *IEEE Transactions on Nuclear Science* 66.7 (2019), pp. 1449–1456. DOI: 10.1109/TNS.2019.2903027.

[184] J. 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)*. 2014, pp. 73–84. DOI: 10.1109/ISCA.2014.6853227.

[185] J. W. Sheaffer et al. “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. Eurographics Association, 2007, pp. 55–64. ISBN: 978-1-5959-3625-7.

[186] J. R. Nickolls. *Defect tolerant redundancy*. Nvidia Corporation, US Patent 6,879,207. Apr. 2005.

[187] S. Ainsworth et al. “Parallel Error Detection Using Heterogeneous Cores”. In: *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2018, pp. 338–349. DOI: 10.1109/DSN.2018.00044.

[188] S.L. Hurst. “VLSI testing and testability considerations: an overview”. In: *Microelectronics Journal* 19.4 (1988), pp. 57–69. ISSN: 0026-2692. DOI: 10.1016/S0026-2692(88)80045-4.

[189] T. Koal et al. “Logic self repair based on regular building blocks”. In: *New Trends in Audio and Video / Signal Processing Algorithms, Architectures, Arrangements, and Applications SPA 2008*. 2008, pp. 109–114. ISBN: 978-1-4577-1660-7.

[190] S. Chin-Lung et al. “A processor-based built-in self-repair design for embedded memories”. In: *2003 Test Symposium*. 2003, pp. 366–371. DOI: 10.1109/ATS.2003.1250838.

[191] S. Chin-Lung et al. “An integrated ECC and redundancy repair scheme for memory reliability enhancement”. In: *20th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT’05)*. 2005, pp. 81–89. DOI: 10.1109/DFTVS.2005.18.

[192] M. Schölzel et al. “A comprehensive software-based self-test and self-repair method for statically scheduled superscalar processors”. In: *2016 17th Latin-American Test Symposium (LATS)*. 2016, pp. 33–38. DOI: 10.1109/LATW.2016.7483336.

[193] T. Koal et al. “A software-based self-test and hardware reconfiguration solution for VLIW processors”. In: *13th IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems*. 2010, pp. 40–43. DOI: 10.1109/DDECS.2010.5491821.

[194] R. S. Ferreira et al. “Low latency reconfiguration mechanism for fine-grained processor internal functional units”. In: *2019 IEEE Latin American Test Symposium (LATS)*. 2019, pp. 1–6. DOI: 10.1109/LATW.2019.8704560.

[195] D. J. Palframan et al. “Precision-aware soft error protection for GPUs”. In: *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 2014, pp. 49–59. DOI: 10.1109/HPCA.2014.6835966.

[196] K. Kwon et al. “Mobile GPU shader processor based on non-blocking Coarse Grained Reconfigurable Arrays architecture”. In: *2013 International Conference on Field-Programmable Technology (FPT)*. 2013, pp. 198–205. DOI: 10.1109/FPT.2013.6718353.

[197] J. Zhao et al. “Energy-Efficient GPU Design with Reconfigurable in-Package Graphics Memory”. In: *Proceedings of the 2012 ACM/IEEE International Symposium on Low Power Electronics and Design*. ISLPED ’12. 2012, pp. 403–408. ISBN: 978-1-4503-1249-3. DOI: 10.1145/2333660.2333752.

[198] W. Lee et al. “A scalable GPU architecture based on dynamically reconfigurable embedded processor”. In: *High Performance Graphics* (2011). Embedded Multimedia Systems Group, Samsung Electronics, pp. 5–7.

[199] L. B. Gomez et al. “GPGPUs: How to combine high computational power with high reliability”. In: *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2014, pp. 1–9. DOI: 10.7873/DATE.2014.354.

[200] A. Dhar et al. “Efficient GPGPU Computing with Cross-Core Resource Sharing and Core Reconfiguration”. In: *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2017, pp. 48–55. DOI: 10.1109/FCCM.2017.59.

[201] S. Di Carlo et al. “Increasing the robustness of CUDA Fermi GPU-based systems”. In: *2013 IEEE 19th International On-Line Testing Symposium (IOLTS)*. 2013, pp. 234–235. DOI: 10.1109/IOLTS.2013.6604088.

[202] S. Di Carlo et al. “Fault mitigation strategies for CUDA GPUs”. In: *2013 IEEE International Test Conference (ITC)*. 2013, pp. 1–8. DOI: 10.1109/TEST.2013.6651908.

[203] J. E. R. Condia et al. “A dynamic reconfiguration mechanism to increase the reliability of GPGPUs”. In: *2020 IEEE 38th VLSI Test Symposium (VTS)*. 2020, pp. 1–6. DOI: 10.1109/VTS48691.2020.9107572.

[204] J. E. R. Condia et al. “DYRE: a DYnamic REconfigurable solution to increase GPGPU’s reliability”. In: *The journal of Supercomputing* (2021), pp. 1–17. DOI: doi.org/10.1007/s11227-021-03751-2.

This Ph.D. thesis has been typeset by means of the TeX-system facilities. The typesetting engine was pdfLaTeX. The document class was `toptesi`, by Claudio Beccari, with option `tipotesi=scudo`. This class is available in every up-to-date and complete TeX-system installation.