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

MÁRCIO MACEDO GONÇALVES

Towards Resilient Graphics Processing Units: Designing Fault Tolerance Techniques for Radiation-Induced Faults

Thesis submitted as a partial requirement for the attainment of the PhD degree in Microelectronics, with a concentration in Electronic Circuits and Systems Testing.

Advisor: Prof. Dr. José Rodrigo Furlanetto de Azambuja

Porto Alegre
March, 2024

CIP — Cataloging-in-Publication

Gonçalves, Márcio

Towards Resilient Graphics Processing Units: Designing Fault Tolerance Techniques for Radiation-Induced Faults / Márcio Gonçalves. – 2024.

161 f.: il.

Advisor: José Rodrigo Azambuja.

Thesis (Doctoral) – Universidade Federal do Rio Grande do Sul, Programa de Pós-Graduação em Microeletrônica, Porto Alegre, BR-RS, 2024.

1. GPU Reliability. 2. Single Event Upsets. 3. Fault-Tolerance Techniques. 4. Safety-critical Applications. I. Azambuja, José Rodrigo (advisor). II. Title.

ACKNOWLEDMENTS

I would like to express my gratitude to my advisor for the collaborative journey we embarked on together, which began even before this project. His availability and involvement were fundamental to the development of this thesis. I am also thankful for the support of my parents and sisters, who have always been there for me, helping me navigate the complexities of life alongside my academic endeavors. Finally, I extend my heartfelt thanks to my wife, my closest companion throughout this period. She shared in the emotional highs and lows, understanding and empathizing with the challenges and victories we encountered.

ABSTRACT

Graphic Processing Units (GPUs) have emerged as powerful computational tools, enabling high-performance parallel processing and driving significant advancements in various domains. However, their integration into safety-critical applications raises concerns regarding their reliability, particularly in the context of Single-Event Upsets (SEUs) caused by radiation-induced faults. This Thesis aims to evaluate GPU reliability under such conditions and develop SEU mitigation techniques. We employed low-level software techniques and hardware experiments, including hybrid approaches combining software flexibility with hardware efficiency, and focused on selectively hardening critical components against radiation-induced faults. Our research began with commercial GPUs, applying selective hardening combined with Approximate Computing to Nvidia's Kepler architecture to enhance fault tolerance. We then shifted to FlexGrip, a softcore GPU for Field-Programmable Gate Arrays (FPGAs), where we explored software-based fault tolerance techniques for SEU detection in configurable architectures. This included novel technique optimizations and comprehensive ISA extensions to improve resilience against SDC and DUE effects. Our study also involved FGPU, another softcore GPU, assessing reliability through comparisons of software-emulated and hardware-based Floating Point implementations, and the effectiveness of selective Triple Modular Redundancy (TMR). Finally, we explored the potential of Application-Specific Integrated Circuits (ASIC) derived from softcore GPUs, utilizing GPUPlanner to facilitate the transition from RTL designs to ASIC layouts. This research highlights the potential of softcore GPUs as ASIC accelerators for high parallelism applications and marks a significant advancement in the development of reliable, fault-tolerant GPU architectures. Our comprehensive evaluation across commercial and softcore GPUs, and the transition to ASICs, sets the groundwork for more robust GPU integration in safety-critical domains and contributes to the advancement of reliable, high-performance computing solutions for a wide range of critical applications.

Keywords: GPU Reliability. Single Event Upsets. Fault-Tolerance Techniques. Safety-critical Applications.

RESUMO

GPUs emergiram como poderosas ferramentas computacionais, possibilitando processamento paralelo de alto desempenho e impulsionando avanços significativos em diversos domínios. No entanto, sua integração em aplicações que requerem alto grau de confiabilidade suscita preocupações sobre a sua confiabilidade, particularmente no contexto de SEUs causados por falhas induzidas por radiação. Esta tese visa avaliar a confiabilidade das GPUs sob tais condições e desenvolver técnicas de mitigação de SEUs. Empregamos técnicas de software de baixo nível e experimentos de hardware, incluindo abordagens híbridas que combinam a flexibilidade do software com a eficiência do hardware, focando na proteção seletiva de componentes críticos contra falhas induzidas por radiação. Nossa pesquisa começou com GPUs comerciais, aplicando proteção seletiva combinada com Computação Aproximada à arquitetura Kepler da Nvidia para aumentar a tolerância a falhas. Em seguida, mudamos para FlexGrip, uma GPU softcore desenvolvida para FPGAs, onde exploramos técnicas de tolerância a falhas baseadas em software para detecção de SEUs em arquiteturas configuráveis. Isso incluiu a implementação de otimizações de técnicas do estado-da-arte e extensões de ISA para melhorar a resiliência contra efeitos SDC e DUE. Nosso estudo também envolveu FGPU, outra GPU softcore, avaliando a confiabilidade por meio de comparações entre implementações de Ponto Flutuante emuladas por software e baseadas em hardware, e a eficácia da técnica TMR implementada de forma seletiva. Por fim, exploramos o potencial de ASICs derivados de GPUs softcore, utilizando GPUPlanner para facilitar a transição de designs RTL para layouts de ASIC. Esta pesquisa destaca o potencial das GPUs softcore como aceleradores ASIC para aplicações de alto paralelismo e marca um avanço significativo no desenvolvimento de arquiteturas de GPU tolerantes a falhas. Nossa avaliação abrangente, desde GPUs comerciais até softcore, e a transição para ASICs, estabelece as bases para uma integração mais robusta de GPUs em domínios críticos à segurança e contribui para o avanço de soluções de computação de alto desempenho e confiáveis para uma ampla gama de aplicações críticas.

Palavras-chave: Confiabilidade das GPUs. SEUs. Técnicas de Tolerância a Falhas. Aplicações Críticas à Segurança.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

ABFT	Algorithm-Based Fault Tolerance
ABNT	*Associação Brasileira de Normas Técnicas*
ALU	Arithmetic Logic Unit
ANN	Artificial Neural Network
ASIC	Application-Specific Integrated Circuit
AUT	Area Under Test
AVF	Architectural Vulnerability Factor
BRAM	Block Random-Access Memory
CAD	Computer-Aided Design
COTS	Commercial Off-The-Shelf
CUDA	Compute Unified Device Architecture
CUT	Circuit Under Test
DSP	Digital Signal Processor
DUE	Detected Unrecoverable Error
DUT	Device Under Test
ECC	Error-Correcting Code
FlexGrip	FLEXible GRaphIcs Processor
FPGA	Field-Programmable Gate Array
FPU	Floating-Point Unit
FGPU	FPGA general purpose GPU
FIT	Failure In Time
GDSII	Graphic Data System II
GUI	Graphic User Interface
GPGPU	General-Purpose Graphics Processing Unit
GPU	Graphics Processing Unit
GPP	General-Purpose Processor
HLS	High-Level Synthesis
HPC	High-Performance Computing

HPCT	Hardening Post Compiling Translator
IC	Integrated Circuits
ICAP	Internal Configuration Access Port
ILL	Institute Laue–Langevin
ILP	Instruction-Level Parallelism
ISA	Instruction Set Architecture
LUT	Look-Up Table
MBU	Multiple Bit Upset
MOSTFET	Metal-Oxide-Semiconductor Field-Effect Transistor
MTBF	Mean Time Between Failures
MWBF	Mean Workload Between Failures
MWTF	Mean Workload to Failure
NIR	Near-Infrared
OpenCL	Open Computing Language
PTX	Parallel Thread Execution
PR	Predicate Register
RGB	Red, Green, Blue
SASS	NVIDIA Specific Assembly
SASSI	SASS Instrumentor
SASSIFI	SASSI Fault Injector
SDC	Silent Data Corruption
SEE	Single Event Effect
SEL	Single Event Latch-up
SEU	Single Event Upset
SET	Single Event Transient
SIMD	Single Instruction, Multiple Data
SIMT	Single Instruction, Multiple Threads
SM	Streaming Multiprocessor
SPIR	Standard Portable Intermediate Representation

SRAM	Static Random-Access Memory
SBU	Single Bit Upset
TENIS	Thermal and Epi-thermal Neutron Irradiation Station
TLP	Thread-Level Parallelism
TMR	Triple Modular Redundancy
UFRGS	Universidade Federal do Rio Grande do Sul
VHDL	VHSIC Hardware Description Language

CONTENTS

1 INTRODUCTION

Graphics Processing Units (GPUs), commonly referred to as General-Purpose GPUs (GPGPUs) when adapted for tasks beyond graphics rendering, have transcended their original design purpose as specialized Integrated Circuits (ICs) for graphics processing. They have evolved into general-purpose accelerators pivotal in High-Performance Computing (HPC). The traditional focus on computer graphics and image processing has been overshadowed by expansive applications ranging from data analytics to artificial intelligence frameworks like Artificial Neural Networks (ANNs) (ESSEN et al., 2015). ANNs, crucial in pattern recognition, data mining, and robotics, have further propelled the utility of GPUs in real-time object detection, a cornerstone in the functionality of autonomous vehicles (REDMON et al., 2015; NVIDIA, 2020; BOJARSKI et al., 2016). As GPUs have shifted their focus, they have become integral in various domains like oil exploration, bioinformatics, cloud computing, radar systems, and more (HASSANI; AIATULLAH; LUKSCH, 2014; HAKOBYAN; YANG, 2019). Notably, GPUs are also present as accelerators in top500 supercomputers, reinforcing their significance in HPC applications (DONGARRA; MEUER; STROHMAIER, 2015). However, as these devices infiltrate safety-critical applications in aerospace, automotive, and medical sectors, their reliability has increasingly been questioned. Reliability often becomes a secondary consideration in maximizing performance through Thread-Level Parallelism (TLP) and massive data-processing capabilities. While performance metrics are undeniably important, it's crucial to note that high-reliability mandates are non-negotiable in safety-critical settings like automotive and medical applications.

1.1 Motivation and Problem Definition

In line with Moore's law, the semiconductor industry has consistently miniaturized the size of transistors, increasing the number of transistors per silicon area and thus leading to more powerful GPUs (MOORE, 1965). However, this march toward smaller, faster, and more efficient devices comes with a growing susceptibility to radiation-induced faults (JEDEC, 2006; BAUMANN, 2005). As fabrication processes approach their physical limits, devices become more vulnerable to energized particles, whether they originate from cosmic rays or from secondary interactions at the Earth's surface, which can cause both permanent and temporary effects on the system.

The probability of an energized particle causing an effect on an IC depends on a few factors, such as transistor density (denser ICs have more transistors upset by a single particle), operating frequency (higher operating frequencies lead to narrower latch windows), and threshold voltage (smaller threshold voltages require less energy transferred for an upset) (SLAYMAN, 2010; DIXIT; WOOD, 2011). Among the most observed events caused by energized particles are Single-Event Upsets (SEUs). A SEU, also known as a bit-flip, is a temporary, non-destructive event that affects data storage elements, such as memories and registers. On an instruction-processing IC such as a GPU or a microprocessor, a SEU can cause mainly two effects: (i) a Silent Data Corruption (SDC) when the program code is correctly executed, but the result is incorrect, or (ii) a Detected Unrecoverable Error (DUE) when the program code is incorrectly stopped or enters an infinite loop.

The newest GPUs are designed with cutting-edge technology that combines high transistor density, high operating frequency, and low threshold voltages, making them prone to experience radiation-induced transient effects up to the point where they can experience radiation effects on applications running at ground level (OLIVEIRA et al., 2014). The consequent SEU events on GPUs are critical to both HPC and safety-critical applications. SDC effects directly affect the result correctness of safety-critical applications, and DUE effects directly affect the timing constraints of HPC applications. The frequency of these faults in clustered HPC systems has increased to an order of minutes, further emphasizing the urgency for reliable solutions (Tiwari et al., 2015). So, the use of effective fault tolerance techniques is mandatory.

Fault tolerance techniques can be applied by means of software or hardware modifications. Software-based techniques require program code transformation, while hardware-based techniques require hardware modifications. Software-based approaches provide high detection rates at the cost of performance degradation. They insert additional instructions that the processing system must execute, therefore increasing execution runtime, and can be applied to any GPU architecture with an available program source code (GONCALVES et al., 2017). Hardware-based approaches, on the other hand, can be applied with low performance degradation, as replicated hardware can be deployed in parallel with the original, and, as long as the critical path is not altered, the operating frequency can be maintained, but requires access to GPU architecture description (Azambuja et al., 2013).

Although hardware-based fault tolerance techniques can offer resilience, they often require invasive architectural changes and are only sometimes feasible for Commercial off-the-shelf GPUs (COTS GPUs). This limitation brings software-based fault tolerance into prominence (Mahmoud et al., 2018). While these techniques incur a performance overhead due to additional instructions, they offer a viable and sometimes the only pathway for fault tolerance in commercial GPUs. Recently, open-source softcore GPUs have allowed developers to study the effects of radiation and design and evaluate fault tolerance techniques (CONDIA et al., 2020; KADI et al., 2018).

1.2 Purpose of the Thesis

This Thesis is dedicated to an in-depth evaluation of low-level software and hardware fault tolerance techniques alongside comprehensive hardware reliability assessments for GPUs, specifically focusing on enhancing their resilience in safety-critical environments. This work builds upon our previous research (GONCALVES et al., 2017), which underscored the significance of selective fault tolerance methods in protecting critical components and the need for a deeper exploration of hardware aspects in GPUs. Initially, in the complex landscape of commercial GPUs, the constraints imposed by proprietary Instruction Set Architecture (ISA) and undisclosed microarchitectural details prompted us to explore alternative platforms for a more comprehensive and robust analysis. Consequently, we turned to softcore GPUs designed for Field-Programmable Gate Arrays (FPGAs). These platforms provide the architectural openness necessary for in-depth reliability evaluation through emulation, simulation, and targeted radiation experiments. The progression of our research then led us from FPGAs to Application-Specific Integrated Circuits (ASICs), a move representing a strategic evolution. ASICs, with their tailored design, offer significant advantages in terms of both performance and reliability compared to FPGAs. Their hardwired circuitry accelerates processing speed and minimizes vulnerability to SEUs. This transition to ASICs enables the creation of GPU architectures that are more efficient and inherently more reliable, particularly suited for high-demand environments where fault tolerance is a necessity.

1.3 Methodological Approach and Case-Studies

Our methodological approach commences with an empirical study of radiation sensitivity in commercial GPUs, specifically focusing on NVIDIA's Kepler architecture. Despite the limitations inherent to proprietary ISAs, we implement low-level software-based techniques performing a quasi-reverse engineering process, linking Specific Assembly (SASS) registers with Parallel Thread Execution (PTX) registers. We propose and evaluate selective fault tolerance techniques for register files and enhance them through Approximate Computing. Reliability is assessed through neutron beam experiments and simulation.

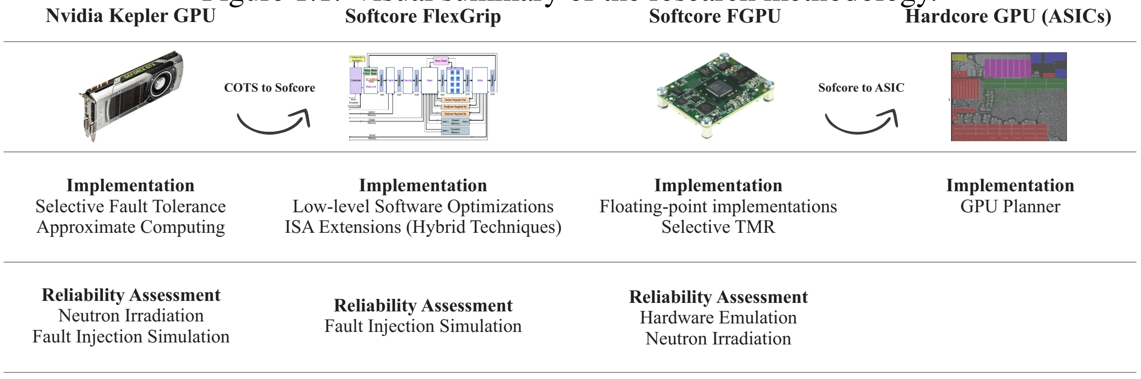
Transitioning from this proprietary realm, we shift our focus to investigating softcore GPUs, starting with FlexGrip, a softcore GPU based on NVIDIA's G80 architecture. This stage of our study focused on analyzing pipeline and register file reliability within FlexGrip, specifically under radiation-induced faults. Our approach included simulation-based experiments for assessing the GPU's vulnerability to SEUs. In this context, we evaluated and optimized low-level software-based fault tolerance techniques, focusing on protecting instruction sets and register files. These techniques were designed to enhance error detection and increase resilience in configurable GPU architectures, enabling a tailored approach to GPU reliability tailored to specific operational needs. Moreover, our research in the FlexGrip environment led to developing hybrid (software and hardware) techniques for error detection and correction within the GPU's pipeline. These techniques included the implementation of comprehensive ISA extensions designed to improve fault detection capabilities and facilitate error correction processes, significantly elevating the fault resilience of the system. FlexGrip, however, presents challenges such as a lack of a stable floating-point unit (FPU) and issues related to memory hierarchy, which limit us to simulation scenarios.

Our research then shifts to FGPU, a more recent FPGA-oriented softcore GPU. Unlike its predecessors, FGPU boasts a functional FPU, enabling hardware-level evaluations. We assess the reliability of applications running on FGPU embedded into an SRAM-based FPGA using both hardware and software floating-point implementations. We also explore the unhardened and hardened reliability curves of isolated components within FGPU to guide decisions on the best candidates for selective hardening through Hardware Selective Triple Modular Redundancy (TMR). Reliability evaluation is conducted through hardware emulation. A pivotal element of our methodology was the integration of hardware emulation to emulate radiation-induced faults, providing an initial understanding of FGPU's vulnerabilities and the effectiveness of our hardening techniques. Notably, we complemented our emulation results with actual radiation tests. These radiation experiments were crucial in confirming the accuracy of our emulation findings, establishing a correlation between emulated conditions and real-world radiation effects.

Concluding our research, we explore the domain of ASICs, presenting a novel solution for implementing GPU architectures in ASICs with various configurations. Our trajectory through commercial and softcore GPUs has demonstrated that reliability is intricately linked to the application in execution, leading us to the idea of designing dedicated GPU accelerators optimized for resilience against radiation effects and tailored to specific application needs. Thus, this exploration signifies a step in introducing the concept of creating fault-tolerant GPU accelerators tailored to specific groups of applications.

Figure 1.1 illustrates the described methodological approach, providing a visual summary of the research transitions from commercial GPUs to softcore GPUs and finally to ASICs.

Figure 1.1: Visual summary of the research methodology.



Source: The author.

1.4 Research Contributions

1. **Commercial GPUs:**
    - Analyzed radiation-induced faults in NVIDIA GPUs, assessing SEU impacts on GPU's registers.
    - Developed selective fault tolerance strategies for GPU register files, enhancing efficiency over non-selective approaches.
    - Proposed acceptance accuracy relaxation methods to improve SEU fault tolerance with reduced overhead.
2. **Softcore GPUs:**
    - Investigated low-level software-based fault tolerance in FlexGrip GPU, including optimizations.
    - Enhanced NVIDIA SASS 1.0 ISA for improved SDC and DUE effect mitigation.
    - Implemented hybrid fault tolerance techniques for GPU pipeline error correction.
    - Conducted reliability studies on FGPU, contrasting soft-FP and hard-FP implementations and their impact on Mean Workload Between Failures (MWBF).
    - Evaluated the effectiveness of selective Triple Modular Redundancy (TMR) in enhancing GPU fault tolerance, particularly in safety-critical applications.
    - Performed radiation experiments on FGPU and correlated findings with emulation results.
3. **Hardcore GPUs:**
    - Developed G-GPU, a domain-specific ASIC accelerator, and introduced GPU-Planner for efficient transition from RTL to ASIC layouts.
    - Demonstrated the suitability of G-GPU for high parallelism applications, advancing adaptable GPU architecture exploration.
4. **Publications:**
    - Published 10 journal articles: IEEE Transactions on Nuclear Science, Microelectronics Reliability, Revista Júnior de Iniciação Científica em Ciências Exatas e Engenharia, and The Journal of Supercomputing.
    - Published 11 conferences proceedings: DATE, ICECS, ISVLSI, LASCAS, LATS, and RADECS.

While this Thesis makes significant strides in evaluating and enhancing GPU fault tolerance, it is important to acknowledge certain limitations. The research primarily revolves around specific GPU architectures like NVIDIA's Kepler and softcore GPUs such as FlexGrip and FGPU, which might limit the extrapolation of our findings to other architectures. Additionally, the results are contingent upon the specific experimental setups used, particularly in radiation testing and fault injection scenarios, which may not fully encapsulate the diverse conditions encountered in real-world applications. The complexity and resource-intensive nature of these experiments also constrain the breadth and depth of our analysis.

1.5 Thesis Organization

The remainder of this Thesis is organized as follows: Chapter 2 provides an overview of GPU architectures for understanding the fundamental aspects of GPU design and functionality. Chapter 3 examines radiation-induced faults, their causes, and implications for GPU reliability. Chapter 4 discusses fault tolerance techniques for GPUs. Chapter 5 introduces the architectures used in our case-studies. Chapters 6, 7, and 8 present our case-studies on the Kepler architecture, FlexGrip, and FGPU, respectively. Chapter 9 explores the transition to ASICs, discussing the development of GPU-based ASIC accelerators and their potential for enhanced fault tolerance. The Thesis concludes with Chapter 10, where we summarize our findings and discuss future work, including the potential for practical implementation and further research in ASIC design for GPUs. The last Chapter, Chapter 11, acknowledges our collaborators and lists the published works that have derived from this research, demonstrating the academic contributions and collective efforts supporting this study.

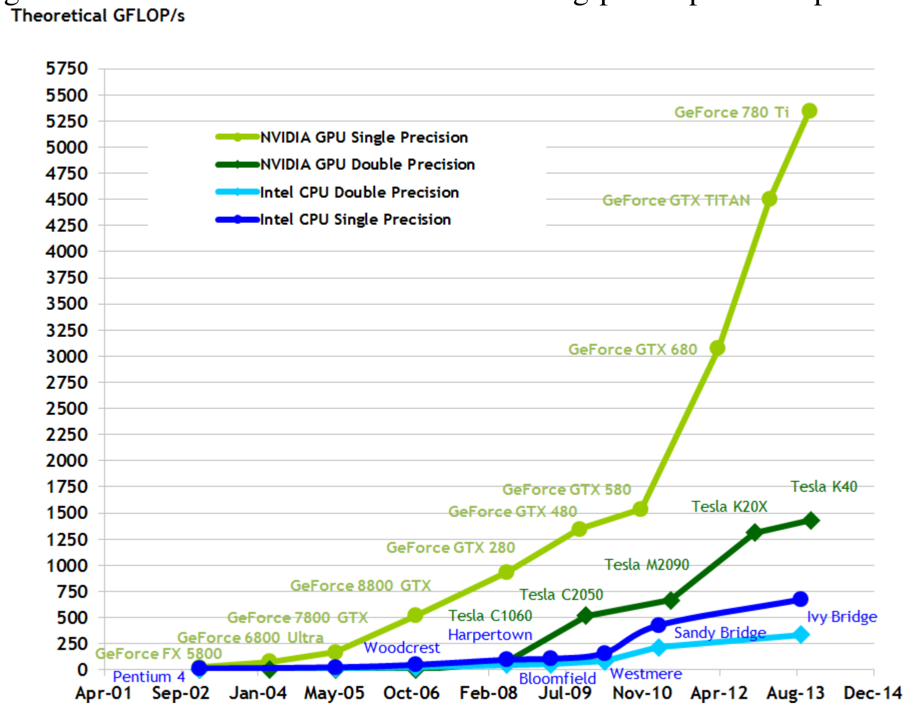
2 GRAPHICS PROCESSING UNITS ARCHITECTURES

This Chapter aims to provide an understanding of the architecture of GPUs, extending from general-purpose computing frameworks like CUDA and OpenCL to specialized implementations in Field-Programmable Gate Arrays (FPGAs) and Application-Specific Integrated Circuits (ASICs). In the first section, the architecture of GPUs is explored, focusing on common programming frameworks such as CUDA and OpenCL. The second section focuses on the context of FPGAs, examining the feasibility and benefits of integrating GPU-like capabilities into FPGA architectures. The third section looks closer at how GPU-inspired concepts can be integrated into ASICs for acceleration and reliability purposes. This Chapter serves as a foundational overview of the architectures and technologies that will be examined in detail in the subsequent Chapters.

2.1 Introduction to GPU Architecture

Originally designed for graphical processing, GPUs consist of a set of multiprocessors capable of running thousands of threads in parallel. Most of the silicon area of a GPU is dedicated to data processing. At the same time, only a small portion is allocated to control units and cache. This architecture makes GPUs more efficient than General-Purpose Processors (GPPs), or CPUs, for large-scale data processing. Figure 2.1 compares the evolution of CPUs and GPUs in terms of floating-point operations per second (NVIDIA, 2015).

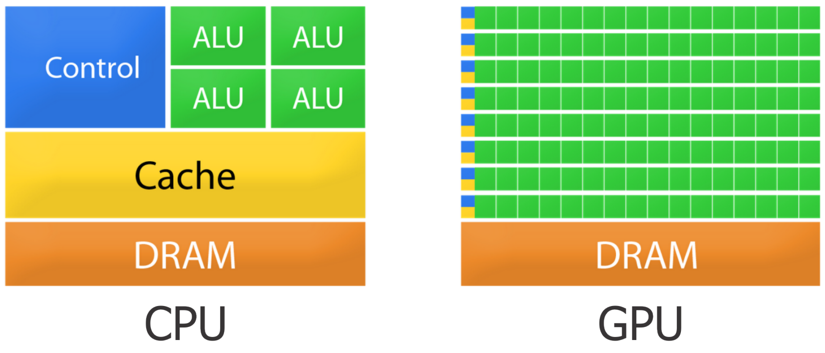
Figure 2.1: GPU vs. CPU: Evolution of floating-point operations per second



Source: (NVIDIA, 2015).

CPUs are built for more generic, sequential algorithms. To optimize performance, they implement intelligent cache mechanisms and control flow features, such as dynamic branch prediction. Therefore, CPUs have robust control units and significant cache memories. In contrast, GPUs have multiple processing units with simpler control units that instantiate and execute many instructions simultaneously. Figure 2.2 compares CPU and GPU architectures.

Figure 2.2: Resource distribution for a CPU and a GPU.



Source: (NVIDIA, 2015).

The underlying computational model for most modern GPUs can best be described as a fusion of Single Instruction Multiple Data (SIMD) and multithreading paradigms, sometimes called Single Instruction Multiple Threads (SIMT). This hybrid model allows for high-throughput, data-level parallelism.

GPUs comprise a hierarchical set of processing units, often called compute units or multiprocessors. These can execute operations concurrently on different data sets. Each compute unit contains multiple smaller processing elements, known as ALUs or cores, responsible for executing individual program threads.

While different vendors have unique terminologies and specifications, the overarching architecture remains conceptually similar, emphasizing parallelism and throughput. Within this landscape, CUDA (Compute Unified Device Architecture) and OpenCL (Open Computing Language) have emerged as two popular frameworks for programming GPUs. CUDA, developed by NVIDIA, is specifically tailored for their line of GPUs. At the same time, OpenCL is an open standard that can run on various hardware platforms, including CPUs, GPUs, and even FPGAs.

Both frameworks offer a rich set of programming constructs for data parallelism, enabling software developers to harness the full computational power of modern GPUs. They provide abstractions such as threads, blocks, and grids in CUDA, or work items, work groups, and NDRange in OpenCL to help users design algorithms that can be broken down into smaller tasks and executed concurrently.

These constructs make GPUs versatile tools, capable of boosting performance across a wide range of applications, from scientific simulations to machine learning algorithms.

2.1.1 Compute Unified Device Architecture - CUDA

Compute Unified Device Architecture (CUDA) is a parallel computing platform and programming model developed by NVIDIA. It enables high-performance computing on GPUs (Graphics Processing Units), aiming for computational and power consumption efficiency. The architecture consists of computational units known as Streaming Multiprocessors (SMs), each capable of running thousands of threads simultaneously. It predominantly employs data parallelism but also offers support for task parallelism.

CUDA's smallest execution unit is a *thread*. These threads are grouped into *blocks*, which in turn are assembled into a *grid*. Each thread within a block runs the same instruction but operates on different data points, achieving data-level parallelism. Multiple blocks may execute in parallel, with the actual scheduling dependent on the number of available Streaming Multiprocessors (SMs) in the GPU. It's important to note that while programmers define grid and block dimensions, the CUDA runtime schedules the blocks on the SMs, a feature that can significantly impact performance.

CUDA incorporates a hierarchical memory model, including global, shared, local, constant, and texture memory. Global memory is accessible by all threads but has the highest latency. In contrast, shared memory is faster but restricted to threads within the same block. Local memory is private to each thread. Understanding this memory hierarchy is vital for optimizing data transfers and access patterns. In CUDA, parallelism is predominantly achieved through data parallelism, with threads executing the same operation on different data.

Programmers specify grid and block dimensions when launching a CUDA kernel, choosing them based on problem size and hardware specifications to maximize performance. Threads within blocks are grouped to optimize shared memory usage within hardware limits. Each thread has a unique ID derived from its block ID and relative thread ID, which can be one-, two-, or three-dimensional vectors. These IDs facilitate mapping to the problem space and allow threads to access specific global and shared memory indexes, ensuring each thread operates on unique data.

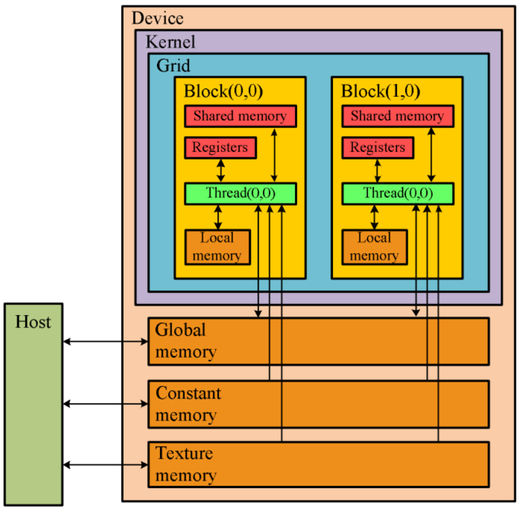
The software flow in CUDA operates at three distinct levels: high-level, intermediate, and assembly level. At the high level, developers write kernel and host code in languages like C or C++, augmented with CUDA-specific extensions. The NVCC compiler processes this code, segregating device and host instructions. Subsequently, a standard host compiler like GCC compiles the host code. In contrast, the device code is compiled into an intermediate form known as PTX (Parallel Thread Execution). PTX serves as a low-level virtual machine and ISA (Instruction Set Architecture), offering a stable programming model for parallel computing. Eventually, the device driver transforms PTX into binary code, or SASS (Specific Assembly), during runtime. This binary code is the device's assembly language.

2.1.2 Open Computing Language - OpenCL

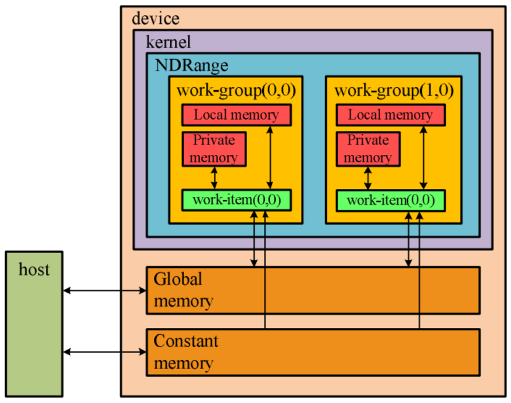
Open Computing Language (OpenCL) serves as a versatile framework for programming across heterogeneous platforms, encompassing CPUs, GPUs, DSPs (Digital Signal Processors), and FPGAs (Field-Programmable Gate Arrays). Distinctively an open standard, OpenCL is not restricted to any specific hardware vendor. The framework comprises a host machine and one or more compute devices, each with its distinct memory hierarchy. These computing devices further house multiple computing units analogous to CUDA's Streaming Multiprocessors (SMs). OpenCL is engineered for flexibility, aiming to be hardware-agnostic and adaptable to various computational scenarios.

OpenCL's execution model boasts significant flexibility, adapting seamlessly to various hardware platforms such as CPUs, GPUs, and even FPGAs. This model adopts a hierarchical structure of computational units analogous to CUDA's architecture. In OpenCL, the primary execution unit is the work-item, which is comparable to a thread in CUDA. These work items are organized into work groups, akin to CUDA's blocks. Multiple work groups form an NDRange, a multi-dimensional array of work items configured with one, two, or three dimensions. This flexibility makes OpenCL inherently adaptable to a wide array of computational challenges. Figure 2.3 delineates the differences between the two models. The CUDA architecture employs a hierarchical structure of a grid composed of blocks with threads sharing local memory and access to global, constant, and texture memory. At the same time, the OpenCL architecture showcases a similar hierarchy with an NDRange consisting of work groups, where each work-item has private memory and shared access to local and global memory. Both frameworks address parallel computation but differ in memory hierarchy and programming abstractions.

Figure 2.3: Side-by-side comparison of GPGPU programming frameworks. (a) CUDA architecture. (b) OpenCL architecture.



(a) CUDA Architecture



(b) OpenCL Architecture

Source: Adapted from (SU et al., 2012).

OpenCL features a complex, multi-tiered memory model that comprises global, local, constant, and private memory types. Global memory is accessible to all work items but has the highest latency. Local memory is shared within a work-group and is faster than global memory. Constant memory is read-only and is optimized for broadcast operations, accessible by all work items. Private memory is specific to individual work items. Understanding the nuances of these memory types is crucial for optimizing performance and data access patterns. Like CUDA, OpenCL also centers on data parallelism as its primary strategy for parallel execution. Each work-item processes the same instruction but on different data, an approach highly efficient for data-parallel computations.

Like CUDA, OpenCL mandates that programmers specify the configuration of work items within work groups and the arrangement of these work groups within the NDRange grid. Optimal performance is achieved by aligning these configurations with problem size and hardware capabilities. OpenCL's support for a diverse range of hardware types provides programmers with additional flexibility in configuration. Each work-item is assigned a unique global ID and a local ID within its work-group. These IDs serve dual purposes: data indexing and execution flow control. Work items within a single work-group can synchronize and share local memory. In contrast, work groups are designed to operate independently.

OpenCL's software flow works across high, intermediate, and assembly levels. At a high level, OpenCL code is written in a C-based language enriched with parallelism-focused extensions. This code is then compiled into an intermediate form known as SPIR (Standard Portable Intermediate Representation), serving a purpose analogous to CUDA's PTX. SPIR acts as a hardware-agnostic layer that makes the code portable across different OpenCL-compatible devices. At runtime, SPIR code is further compiled into the device's native assembly language for execution.

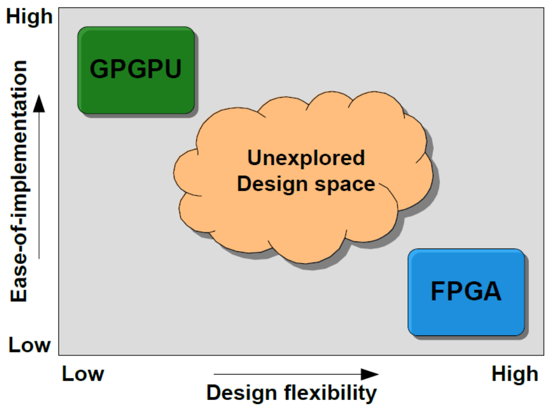
2.2 Softcore GPUs

Softcore GPUs act as hardware descriptions of general-purpose GPUs that are implementable in Field-Programmable Gate Arrays (FPGAs). Softcore GPUs allow rapid prototyping and implementation thanks to their inherent flexibility. This characteristic addresses some of the limitations associated with High-Level Synthesis (HLS), particularly regarding architecture customization and development speed. Softcore GPUs offer a more adaptive architecture, facilitating customization to align with the requirements of specific applications. Moreover, they simplify the development process by supporting widely-used GPGPU programming languages such as OpenCL and CUDA.

One key advantage of softcore GPUs is their ability to tailor computational resources to application-specific needs. Unlike fixed hardware structures, softcore GPUs can be scaled to optimize processing cores, memory hierarchy, and interconnects for performance, power, or area. This flexibility can lead to more efficient use of FPGA resources, especially in systems where full-scale GPGPU integration is not feasible or necessary. The adaptability of softcore GPUs also extends to their capability to incorporate domain-specific optimizations, such as selective hardening of computation units or subsetting of instruction sets, which can yield significant power savings and performance gains for targeted applications. Furthermore, they present an opportunity for system integration in environments lacking native GPGPU support, granting these systems access to GPGPU-like parallelism and throughput.

Figure 2.4 visually positions softcore GPUs within the design space between traditional FPGAs and GPGPUs, illustrating the trade-off between ease of implementation and design flexibility, which is central to the concept of softcore GPUs. This balance offers both computational power and the capability to tailor to specific application requirements.

Figure 2.4: The trade-off between ease-of-implementation and design flexibility for softcore GPUs synthesized on FPGAs. This illustration encapsulates the design space where softcore GPUs offer a balanced approach between the high performance of GPGPUs and the flexibility of FPGAs.



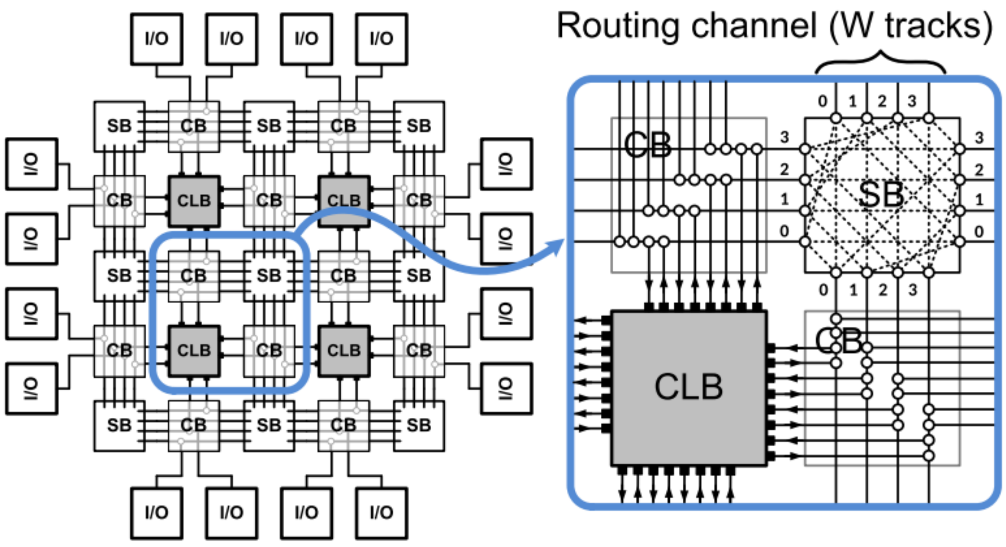
Source: adapted from (MERCHANT, 2013).

2.2.1 Understanding the FPGA Architecture

FPGAs, or Field-Programmable Gate Arrays, are versatile and reconfigurable hardware platforms that offer a middle ground between the flexibility of software and the performance of dedicated hardware. Unlike Application-Specific Integrated Circuits (ASICs), which are designed for a fixed function, FPGAs offer the flexibility to be programmed and reprogrammed for a myriad of tasks, a feature depicted in the architecture shown in Figure 2.5. Their core architecture comprises a matrix of configurable logic blocks (CLBs) interconnected by programmable routes, as illustrated on the left side of the Figure. These logic blocks include fundamental building blocks like Look-Up Tables (LUTs), specialized components such as Digital Signal Processors (DSPs) for high-speed arithmetic operations, and Block RAMs (BRAMs) for low-latency internal data storage. The right side of the Figure zooms into the detail of the routing channel, showcasing the versatility of connections through switch blocks (SB) that interlink the CLBs.

While the hardware serves as the architectural bedrock, the design and customization are guided by specialized Computer-Aided Design (CAD) tools, most notably Xilinx's Vivado Design Suite. Starting from High-Level Synthesis, Vivado can take code written in high-level languages like C++ or domain-specific languages like CUDA and OpenCL and translate it into hardware description languages like VHDL or Verilog. This process is crucial, as it converts abstract computational concepts into concrete hardware implementations, aligning with the configurable nature of FPGA architecture shown in Figure 2.5. This synthesized high-level representation is then further processed into a gate-level netlist, outlining the logical gates and their interconnections. The placement and routing phase, as exemplified by the programmable routes in the Figure 2.5, is where the logic is meticulously mapped onto the specific architecture of the FPGA.

Figure 2.5: Schematic representation of FPGA architecture highlighting the matrix of CLBs and the detailed structure of a routing channel with SBs.



Source: Adapted from (QIN et al., 2018).

The final step in this process is the generation of a bitstream, a set of configuration data that, when loaded onto the FPGA, programs its logic blocks and interconnects to perform the desired digital function, effectively transforming the FPGA into a tailored solution for a particular application.

The synergy between hardware and software, the fusion between physical resources and logical design, elevates the compelling utility of FPGAs. They are a robust platform capable of implementing everything from simple logic circuits to sophisticated parallel algorithms, making them indispensable in modern digital design paradigms.

2.2.2 Softcore GPUs and their Potential in Reliability Domain

In the domain of system reliability, softcore GPUs present a myriad of investigative possibilities. They are a versatile platform for assessing a broad spectrum of fault-tolerance techniques, encompassing hardware-based methods like Triple Modular Redundancy (TMR) and software-based strategies. One of the key advantages of softcore GPUs is their adaptability, which allows for rigorous testing through both Register-Transfer Level (RTL) simulation and hardware emulation. Specifically, RTL simulation enables highly accurate assessments. At the same time, hardware emulation allows for implementing additional hardware modules directly alongside the Design Under Test (DUT). These modules can inject faults in real-time, directly from within the platform, as the design is running an application. Additionally, softcore GPUs are particularly well-suited for radiation testing on platforms such as SRAM-based FPGAs, granting researchers invaluable data on the softcore GPUs' resilience to radiation-induced faults.

2.3 Hardcore GPUs

While FPGA-based softcore GPUs provide a flexible and dynamic platform for reliability testing, ASICs (Application-Specific Integrated Circuits) offer an alternative that excels in hardware optimization and fault tolerance. In drawing a parallel to the FPGA implementations discussed earlier, it becomes crucial to explore how the transition to ASICs can offer specific advantages in the context of reliability.

ASICs present the capability to convert softcore GPUs from hardware description languages into highly optimized, application-specific hardware. This capability enables the introduction of hardware-level redundancies into the design. Unlike FPGAs, where flexibility is a key strength, ASICs allow for micro-level optimizations that are particularly desired for high-reliability applications. These redundancies can be tailored to fit the requirements of a particular application domain, thereby improving fault tolerance without compromising performance.

Despite the benefits of HLS in facilitating early-stage prototyping (CANIS et al., 2011), it often falls short in providing the precision required for ASIC designs aiming for peak performance and reliability (WENG et al., 2020). While the cost associated with designing application-specific accelerators is typically high, thereby posing an economic barrier to widespread adoption, the ability to convert softcore GPUs to ASICs mitigates this challenge. It does so by marrying the efficiency of domain-specific accelerators with the flexibility and programmability of general-purpose architectures (PEREZ et al., 2022).

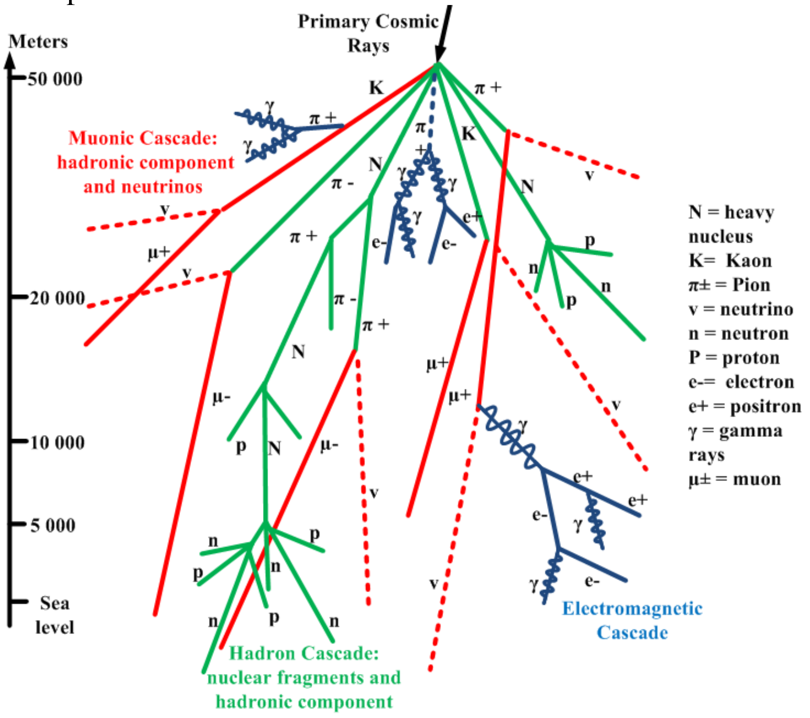
In summary, transitioning softcore GPUs from FPGA-based implementations to hardcore ASICs brings forth a considerable edge in system reliability. It opens doors to fine-grained, application-specific hardware optimizations, significantly boosting system fault tolerance and overall reliability.

With a solid foundation in GPU architectures and the methodological approaches for studying their reliability, we now turn our attention to a critical aspect of GPU performance in adverse conditions: radiation-induced faults. Understanding how these faults impact the reliability of GPU systems is essential for developing robust fault-tolerance techniques. The following Chapter delves into the mechanisms of radiation-induced faults, their effects on semiconductor circuits, and the methodologies employed to assess and enhance the reliability of GPU architectures in radiation-prone environments.

3 RADIATION-INDUCED FAULTS AND RELIABILITY ASSESSMENT

Cosmic rays are a celestial starting point for understanding radiation-induced faults in semiconductor circuits. These rays are high-energy particles originating from outer space and traveling across the universe. The Earth's magnetic field plays a pivotal role in shielding against these high-energy cosmic particles by deflecting or trapping them in radiation belts. As these primary cosmic rays penetrate the Earth's atmosphere, they collide with atmospheric molecules, initiating a cascade of secondary particles process depicted in Figure 3.1. This phenomenon, known as an 'air shower,' modulates the energy spectrum of the incoming radiation, culminating in a variety of particles such as neutrons (n), protons (p), alpha particles (α), electrons or positrons (e), gamma-ray photons (γ), pions (π), and muons (μ). These interactions are more frequent at higher altitudes, leading to an abundance of secondary particles that pose a greater risk of inducing faults in electronic circuits.

Figure 3.1: Extensive Air Shower initiated by a high-energy cosmic ray particle entering the Earth's atmosphere.



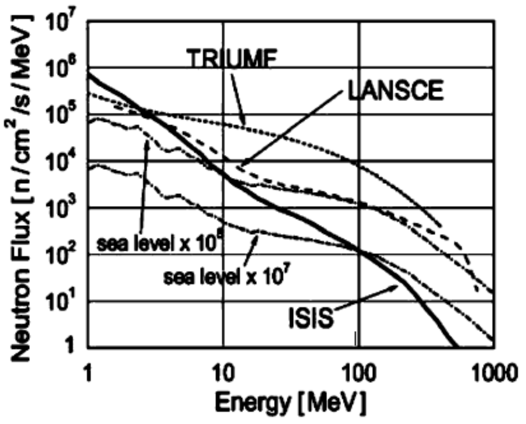
Source: (MARTEN; SUTTON, 2002).

However, the radiation impacting electronic circuits is not solely from cosmic origins. Terrestrial radiation—emitted from the decay of radioactive elements such as uranium, thorium, and radon found in the Earth's crust—also significantly affects semiconductor devices, especially at ground level, where we find applications like autonomous vehicles and high-performance computing systems. Neutrons, byproducts of these radioactive decays, can indirectly ionize silicon when interacting with the material, possibly leading to Single-Event Effects (SEEs) (AL-KHAWLANY; KHAN; PATHAN, 2018).

Understanding the interactions between radiation and electronic circuits begins with two key parameters: energy and flux. Energy characterizes the potential of radiation particles to induce faults. At the same time, flux measures the rate at which these particles encounter a target area, such as a semiconductor device. Higher energy levels and flux rates increase the likelihood of faults within circuits, potentially causing errors that disrupt normal operation. This relationship underscores the importance of assessing the energy spectrum and the radiation flux to which electronic systems are exposed.

The observed effects of radiation on high-performance computing systems can be quantitatively assessed by examining the comprehensive data from the Titan supercomputer, as reported in the study (Tiwari et al., 2015). Figure 3.2 depicts the neutron flux spectrum across various energy levels from different radiation facilities, such as TRIUMF, LANSCE, and ISIS, as well as the natural neutron flux at sea level multiplied by factors of 10^7 and 10^8. These data highlight the spectrum of energy levels that electronic systems may encounter, ranging from ground-level to aviation and space operation conditions. Figure 3.3 provides a histogram of the time between failures for all GPUs within the Titan supercomputer, illustrating the distribution and frequency of failures over time. This histogram provides valuable insights into operational reliability and the potential impact of radiation-induced faults on system stability. This data highlights the essential need for robust fault tolerance mechanisms to mitigate radiation-induced errors, including those affecting applications at the ground level.

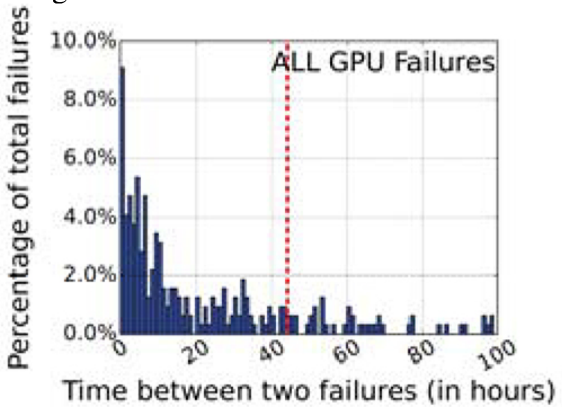
Figure 3.2: Neutron Flux Spectrum.



Source: Adapted from (VIOLANTE et al., 2007b).

In the following section, we delve deeper into the mechanisms of radiation-induced faults and their effects on circuits. By understanding these interactions, we can develop strategies to improve the resilience of electronic systems against the unpredictable nature of radiation.

Figure 3.3: Time Between GPU Failures.



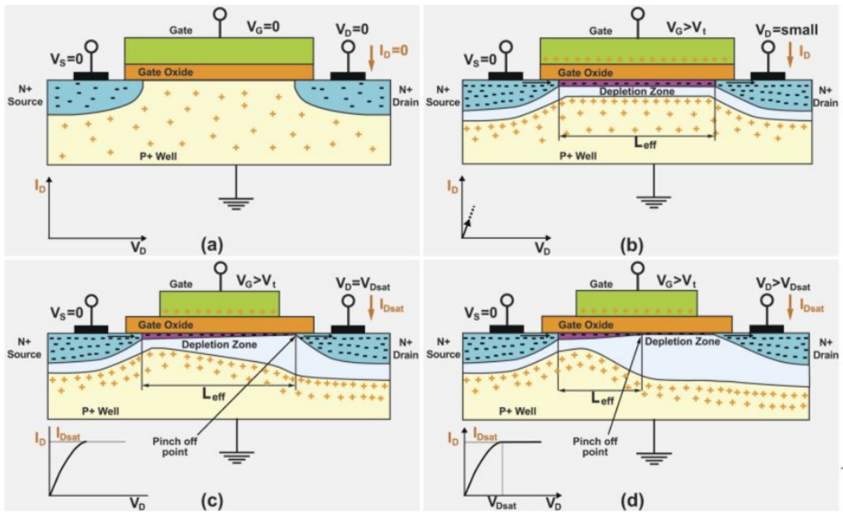
Source: Adapted from (Tiwari et al., 2015).

3.1 Radiation-Induced Faults and their Effects on Circuits

Semiconductors, particularly Metal-Oxide-Semiconductor Field-Effect Transistors (MOSFETs), are the cornerstone of modern electronics, found in everything from microprocessors to memory devices. The fundamental operation of a MOSFET relies on its ability to control the current flow via an electric field. When a voltage higher than the threshold is applied to the gate, it creates a conductive channel between the source and drain, allowing current to flow. Conversely, when the gate voltage is below the threshold, the channel is non-conductive, and the current is inhibited.

Figure 3.4 shows the progressive stages of a MOSFET's operation: (a) with no applied gate voltage, the channel is in a state of depletion; (b) as the gate voltage surpasses the threshold, a conductive channel forms; (c) with a further increase in voltage, the channel reaches saturation; and (d) at even higher voltages, a 'pinch-off' occurs, where the channel near the drain narrows, restricting current flow (MKS Instruments, 2017).

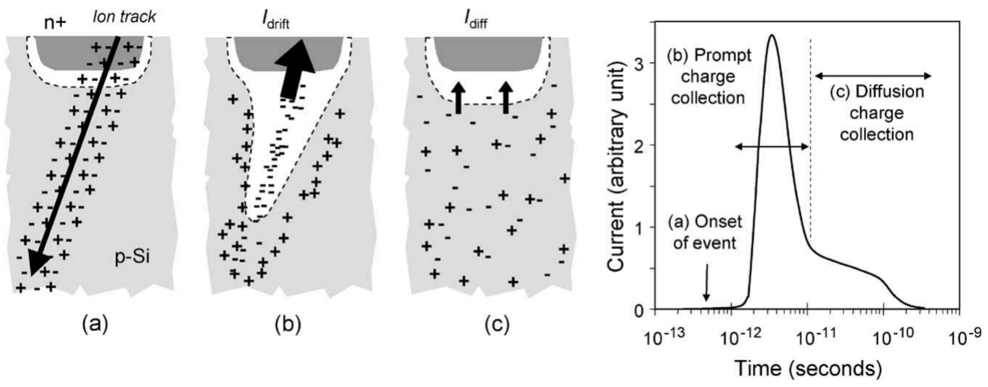
Figure 3.4: Operational stages of a MOSFET.



Source: Adapted from (MKS Instruments, 2017).

The interaction of radiation particles with these semiconductor devices is a critical concern in electronics. The vulnerability of MOSFETs to radiation-induced faults is rooted in their operational sensitivity to charge accumulation. This sensitivity is amplified due to the miniature scale of modern transistors, where the effects of ionizing particles can have a more pronounced impact on the channel conductivity. As depicted in Figure 3.5, when a high-energy particle, such as a neutron, strikes a silicon lattice, it can displace silicon atoms from their positions, creating a track of electron-hole pairs along its path. The graph traces the resulting current, marking (a) the onset of the event, (b) the prompt charge collection, and (c) the diffusion charge collection over time, highlighting the transient nature of radiation-induced currents in semiconductor devices. This disturbance is particularly significant for MOSFETs because their operation hinges on precisely controlling these charge carriers. The immediate effect is the creation of a transient current as these carriers are quickly collected (prompt charge collection), followed by a longer period where the remaining carriers diffuse towards the electrodes (diffusion charge collection) (BAUMANN, 2005).

Figure 3.5: Sequential stages of radiation interaction with a semiconductor. (a) ion track as a charged particle penetrates the p-type silicon substrate. (b) drift of electron-hole pairs generated by the ionizing event. (c) diffusion of carriers after the initial drift.



Source: Adapted from (BAUMANN, 2005).

This transient response in semiconductors is not merely a fleeting fluctuation; it is the precursor to Single Event Effects (SEEs), which can temporarily alter the operational states of a MOSFET, leading to soft errors in digital circuits. Permanent damages, or 'hard errors,' occur when the energy transferred from the particle to the silicon lattice is sufficient to cause lasting atomic displacements. These defects can disrupt the lattice structure, leading to altered electrical properties, increased leakage currents, and potential device failure. Transient effects can cause temporary malfunctions, particularly critical in high-reliability applications such as aerospace and medical devices.

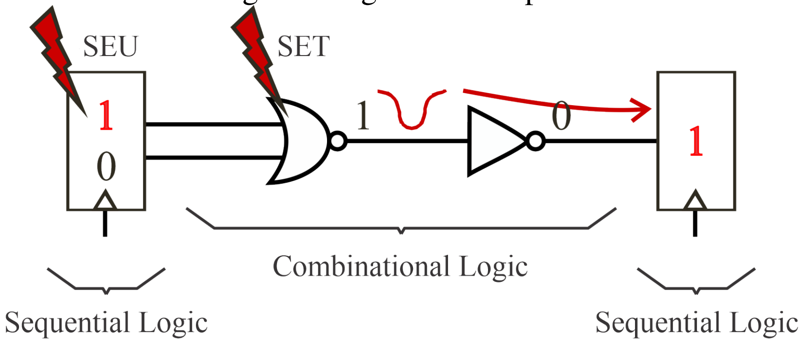
3.1.1 Single-Event Effects

Radiation-induced faults within semiconductor circuits can lead to two primary types of Single-Event Effects (SEEs): destructive 'hard errors' and non-destructive 'soft errors.' Hard errors, such as Single Event Latchup (SEL), can cause permanent damage to the device, necessitating a power cycle or system reset for recovery. SEL occurs when a charged particle activates a parasitic structure within the device, resulting in a high current state capable of causing thermal damage if not promptly addressed (BAUMANN, 2005).

In contrast, soft errors, which include Single Event Transients (SETs) and Single Event Upsets (SEUs), disrupt the circuit's operation without causing lasting physical damage. SETs are fleeting voltage disturbances within the circuit's logic, which, if captured by a latching mechanism, can result in incorrect logical operation or data corruption. SEUs occur when radiation particles alter the data state within a memory element, such as a bit in a register or a flip-flop. Although these events can corrupt data, they are considered 'soft' because normal functionality can be restored by overwriting the corrupted data with the correct values.

Figure 3.6 shows examples of the effects of an SEU and an SET on a circuit. On the left, the effect of an SEU can be observed, where a particle impinges on a memory element, represented as a register, and changes its value from '00' to '10.' This effect impacts the rest of the circuit, changing the value of the register on the right from '0' to '1.' On the NOR gate of the circuit, the incidence of a particle causes a voltage pulse over the combinational logic, which propagates through the circuit to the sequential logic on the right, registering the incorrect value '1' instead of '0.'

Figure 3.6: Illustration of SET and SEU effects in digital circuits: An SEU changing a memory bit's state and an SET generating a transient pulse in combinational logic.



Source: The Author.

In modern devices, the implications of SEEs are further compounded. The high density of nanometric transistors within these circuits, along with their reduced operating voltages and increased operating frequencies, escalates their susceptibility to radiation-induced faults. This vulnerability is threefold:

1. The reduced operating voltage of transistors, aimed at lowering power consumption, makes the components more sensitive to charged particles. These particles could be misinterpreted as internal circuit signals.
2. Elevating the circuit's operating frequency makes it more likely for transient pulses, introduced by charged particles, to align with clock edges.
3. The close proximity of densely packed transistors increases the risk of multiple transistors being affected by a single charged particle incident on the silicon substrate.

3.1.2 Definition of Fault, Error, and Failure

Undesired effects in circuits, such as a disturbance caused by a radiation particle, can corrupt the system's functioning and cause serious issues, depending on the ongoing application. However, this is only sometimes the case, as these disturbances can be masked by hardware or software (in microprocessor systems) or may result in errors irrelevant to the running application. In this context, to classify the effects caused by these disturbances, we will use the definitions of fault, error, and failure as presented by (AVIZIENIS et al., 2004).

When a radiation particle strikes an integrated circuit and causes a disturbance, like an SET, this event is defined as a fault. Faults can be masked by both hardware and software. In hardware, a fault in the circuit might be masked due to combinational logic, electrical factors, and the sampling window of the registers. In these cases, the fault may not propagate to the system outputs. When a fault propagates to the information level, altering an application's variable, it can still be masked during software execution. For instance, this happens when a fault changes the value of data stored in a register that the program won't use or will be overwritten.

However, when a fault is not masked by either hardware or software, altering the system's expected output in terms of data or control, it becomes an error. This error may, in turn, lead to a failure. Failure is the undesirable behavior corresponding to system malfunctioning, usually observable by the user. Nevertheless, not all errors become failures. For example, a fault causing a calculation error in a single pixel of a television image doesn't constitute a failure. However, if a fault causes an error that freezes the image for a long time, this phenomenon may be considered a failure. In applications requiring a high degree of reliability or even those that are critical, where human lives are involved, failures are unacceptable.

3.1.3 Characterization of SDC and DUE Effects

Silent Data Corruption (SDC) occurs silently, without immediate detection, leading to erroneous outcomes despite the correct execution sequence of the program. For instance, if a radiation particle induces a fault that is not caught by hardware or software checks and alters the data being processed, the system continues its operation, producing incorrect results unbeknownst to the user.

In contrast, Detected Unrecoverable Errors (DUEs) are system errors that are identified by the system but are beyond the capability of the system's error-handling mechanisms to correct or recover autonomously. These errors often result in noted system behavior such as crashes, halts, or infinite loops. For example, an exception—such as division by zero caused by a corrupted data value—can lead to a DUE if the system's exception handling cannot resolve it and restore normal program flow.

It is worth noting that not all faults detected by the system result in error correction or system recovery. In instances where fault tolerance techniques are specifically developed for detection, it's accurate to say that these techniques may lead the system to signal a DUE. Such detection-focused techniques alert to the presence of an error; however, their lack of mechanisms for correction or recovery renders the detected faults unrecoverable by the system's own resources. The system's capacity to identify an error is crucial; yet, absent the ability to amend or maintain safe operation, this detection signals that standard operations are jeopardized, necessitating external intervention.

Reflecting on the implications of SDC and DUE underscores their relevance to prior discussions on faults, errors, and failures. SDCs, as a particular category of errors, pose a uniquely critical threat in safety-critical systems where silent errors may lead to catastrophic consequences without immediate detection or warning. The insidious nature of SDCs means that they can undermine the integrity of a system's outputs, potentially causing severe outcomes before any issue is apparent. On the other hand, while DUEs present a clear and detectable disruption, their primary challenge lies in high-performance contexts, especially over extended operations. DUEs necessitate recovery or restart procedures, leading to significant downtime or performance degradation. This distinction is pivotal in designing fault-tolerant systems for environments where reliability cannot be compromised and the cost of error—whether silent or visible—is unacceptably high.

3.2 Reliability Assessment Methods

Considering the potential repercussions of radiation-induced disturbances on semiconductor circuits, it becomes imperative to rigorously assess the reliability of these circuits before subjecting them to harsh environments. Although the effects of faults exist at sea level, the rate is still inadequate for comprehensive testing of fault tolerance techniques. Moreover, these projects often require scaling in challenging environments like high altitudes and outer space. Therefore, fault emulation and testing are indispensable.

One approach to closely simulate real-world conditions is by utilizing particle accelerators, such as Cyclotrons and Spallation Neutron Sources. Globally available facilities like the Alamos Neutron Science Center (LANSCE) in the United States, the ISIS Neutron and Muon Source in the United Kingdom, and the TRIUMF facility in Canada, employ a variety of energized particles, including heavy ions, protons, and neutrons. These facilities enable the simulation of cosmic radiation effects on semiconductor circuits. However, operational costs and the challenge of precisely targeting specific hardware components remain significant drawbacks (VIOLANTE et al., 2007a; LISOWSKI; SCHOENBERG, 2006). The Thermal and Epi-thermal Neutron Irradiation Station (TENIS) at the Institute Laue-Langevin (ILL) in Grenoble, France, specializes in applying thermal and epithermal neutrons. TENIS provides a distinctive environment for assessing the impact of low-energy neutrons, offering a complementary perspective to the high-energy neutron and ion tests conducted at LANSCE, ISIS, and TRIUMF. This diverse array of testing facilities ensures a comprehensive evaluation of semiconductor circuit reliability under various radiation conditions, enhancing the understanding of potential vulnerabilities and the effectiveness of mitigation strategies.

Simulation-based testing, on the other hand, can be conducted at both electrical and logical levels. This approach's main advantage is the precise control over fault injection, thus enabling in-depth analysis of fault effects on varying system structures (CARREIRA; MADEIRA; SILVA, 1998). Reliable simulation tools like HSPICE by Synopsys for electrical-level simulation and ModelSim by Mentor for logic-level are commonly used (SYNOPSYS, 2022; MENTOR GRAPHICS, 2022).

For Field Programmable Gate Arrays (FPGAs), fault emulation is possible using reconfiguration mechanisms (LEGAT; BIASIZZO; NOVAK, 2010). Hardware modules can also be added to the circuit to emulate system faults explicitly. FPGAs offer the advantage of quicker fault emulation, albeit with an increase in implementation complexity.

In summary, simulation-based fault injection provides superior control over testing, facilitating detailed analyses. Cyclotron-based methods offer a realistic test environment but at higher operational costs. FPGA-based testing serves as a balanced approach.

For a comprehensive reliability assessment, a multi-method approach may be considered. This method amalgamates the advantages of each technique to offer an exhaustive view of the system's resilience. This strategy ensures that the system meets functional safety standards like IEC 61508 and ISO 26262 (BROWN, 2000; STANDARD, 2018) while preparing for real-world operational challenges.

3.2.1 Metrics for Reliability Evaluation

Before diving into the metrics, it's important to note that the term "Failure" used in this section is employed in a specific, quantitative context to measure system reliability.

This concept is different from the general definition of "Failure" discussed in the previous section, which refers to the observable malfunctioning of the system. In the metrics discussed below, "Failure" is quantified and used to evaluate the system's robustness and longevity.

In the context of reliability in systems exposed to radiation-induced faults, various metrics are used to evaluate susceptibility, longevity, and robustness. These metrics are pivotal for comprehending how faults propagate through the system and manifest as errors in both GPUs and FPGAs.

- **FIT (Failure In Time)**: FIT is a metric used to quantify the reliability of hardware. It represents the number of failures that can be expected in one billion (10^9) hours of operation. It is useful for predicting the error rate over a large period.
- **AVF (Architectural Vulnerability Factor)**: AVF measures the likelihood of a fault causing an error within a system, providing a rate or probability of error due to faults. It is used to identify system parts most susceptible to errors.
- **MTBF (Mean Time Between Failures)**: MTBF is defined as the average time between two successive failures, indicating the reliability of a system. Higher MTBF values signify a more reliable system and are used to compare different systems or configurations.
- **MWBF (Mean Workload Between Failures)**: MWBF takes into account the workload processed between failures, considering error rates, AVF, and runtime to provide a view of system reliability.
- **MWTF (Mean Workload to Failure)**: MWTF accounts for AVF, performance, and area, offering a dimension for reliability evaluation. It is the average workload completed before the system fails, normalized over specific configurations to isolate architectural and application-specific aspects.
- **MFTF (Mean Faults to Failure)**: This metric quantifies the average number of faults that occur before a failure is observed in a system. A higher MFTF value indicates better fault tolerance or a lower likelihood of failure due to faults.
- **MTTF (Mean Time to Failure)**: MTTF measures the average operational time between system failures. It is an indicator of the system's reliability and expected lifetime. A longer MTTF suggests a more reliable and enduring system.
- **MEBF (Mean Executions Between Failure)**: This metric represents the average number of times a system can execute a task or operation between failures. It provides insight into operational reliability and can guide the frequency of maintenance or system checks.
- **MΦTF (Mean Fluence to Failure)**: MΦTF is defined as the average particle fluence that a system can withstand before a failure occurs. It is particularly relevant in environments with high radiation levels, as it helps assess the system's resilience to particle-induced faults.

- **SEU Cross-Section**: This metric measures the susceptibility of hardware to radiation-induced soft errors. It is characterized statically, based on hardware technology, and dynamically influenced by the hardware's operational state and the application executed, reflecting the system's vulnerability to radiation under different scenarios.

3.3 Reliability of Softcore GPUs in SRAM-based FPGA

When deploying softcore GPUs on SRAM-based FPGAs, specific considerations need to be addressed regarding reliability, particularly under conditions susceptible to radiation-induced faults. These considerations are distinct from those of ASICs designed to function as GPUs. This section elucidates the unique vulnerabilities and reliability issues that manifest when a softcore GPU is implemented on an SRAM-based FPGA compared to its ASIC counterpart.

3.3.1 Structural Vulnerabilities of SRAM-based FPGAs

SRAM-based FPGAs store their configuration in volatile SRAM cells prone to SEUs. This vulnerability is critical because the configuration of the FPGA, including the softcore GPU, is defined by the state of these cells. SEUs can lead to accumulative and systemic issues, corrupting the FPGA's configuration and potentially causing widespread logic or functional failures. The FPGA's layout, comprising configurable logic blocks and routing lines, is particularly susceptible to such faults, leading to persistent corruption until the configuration is reloaded.

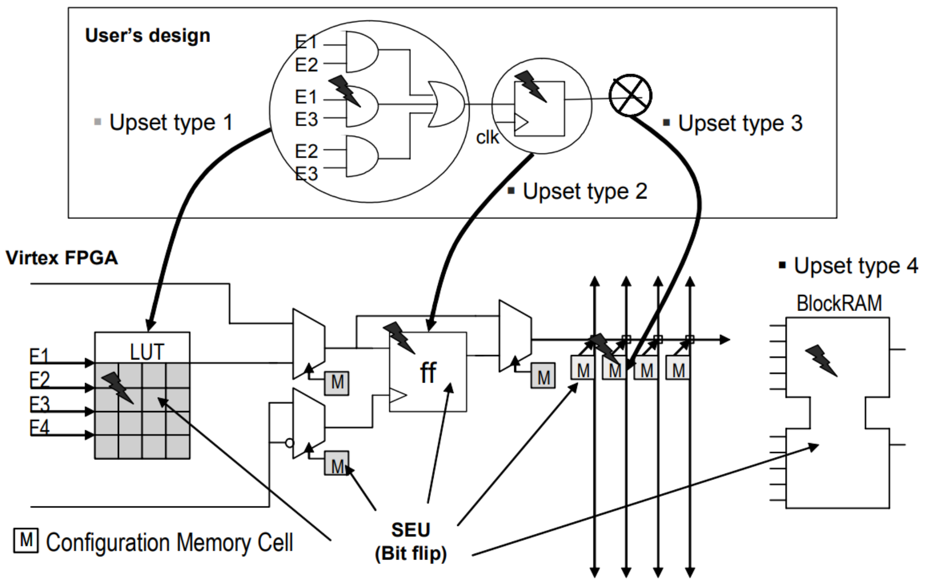
Softcore GPUs on FPGAs inherit these vulnerabilities and introduce new ones based on their architectural design. Particular attention must be paid to Register Transfer Level (RTL) descriptions, as errors here can disrupt the execution flow and lead to accumulative errors that compound over time, degrading performance and reliability. These issues must be scrutinized alongside the standard concerns for any design on an SRAM-based FPGA.

3.3.2 Comparison with ASIC Implementation

Figure 3.7 from the study (KASTENSMIDT; CARRO; REIS, 2006) contrasts the impact of SEUs on SRAM-based FPGAs with the user's design in ASICs. In FPGAs, SEUs can cause permanent logic or memory alterations until the configuration is refreshed; these alterations can accumulate over time, potentially degrading system performance or leading to failures if not addressed. Conversely, ASICs primarily experience transient effects that do not alter the physical configuration. SEUs in FPGA LUTs can result in persistent faults, analogous to stuck-at faults in combinational logic (upset type

1). Routing upsets in FPGAs can disrupt connections, potentially leading to a cumulative deterioration of the interconnect integrity, akin to creating open or short circuits (upset type 3).

Figure 3.7: Comparative illustration of SEU impacts on FPGA vs. ASIC architecture.



Source: Adapted from (KASTENSMIDT; CARRO; REIS, 2006).

While SEUs in the sequential logic of an FPGA (upset type 2) are typically transient and corrected in the next clock cycle, those in embedded memory blocks like BRAMs can lead to persistent data corruption (upset type 4). This corruption can accumulate, further exacerbating the issue and necessitating robust fault-tolerance techniques beyond simple reconfiguration. SRAM-based FPGAs are thus more vulnerable than ASICs, which do not face bitstream corruption due to their immutable architecture.

By contrast, ASICs would be specifically tailored for GPU functions, which minimizes the scope of faults. However, despite their lower susceptibility to radiation-induced faults, ASICs lack the reconfigurability of FPGAs and require a higher initial investment, making them less favorable for applications demanding adaptability and rapid prototyping.

3.4 Supporting Tools for Reliability Assessment

Deploying fault simulation and emulation tools is indispensable for rigorous reliability assessment across hardware platforms. For instance, simulation-based methods, represented by works like (LI; SASANKA, 2010) and (SHEAFFER; LUEBKE; SKADRON, 2009), are essential for evaluating commercial GPUs. In contrast, the RTL fault injection tool from Universidade Federal do Rio Grande do Sul (UFRGS) allows for meticulous examination of hardware, giving crucial precise insights for developing effective fault-tolerance strategies. Additionally, UFRGS contributes to the field with a specialized tool for fault emulation in FPGAs, further broadening the scope and depth of hardware reliability studies.

3.4.1 Fault Simulator for COTS GPUs

Among the various fault injection tools available, SASSIFI (SASSI Fault Injector) stands out due to its comprehensiveness and precision. SASSIFI was designed to inject SEUs at different abstraction levels within CUDA kernels, categorizing injected faults into classes such as register file faults, shared memory faults, and global memory faults to closely mimic the vulnerabilities that might arise from radiation-induced faults (MENON et al., 2014). The SASSIFI framework operates at a low level, building upon SASSI (STEPHENSON MARK SASTRY HARI et al., 2015), a Shader Assembly Instrumentation tool that facilitates the instrumentation of instructions in the GPU assembly language, SASS. This low-level operation allows for a detailed analysis of the fault resilience of commercial GPUs, aiding researchers in understanding and mitigating the vulnerabilities inherent in these systems.

SASSIFI operates in multiple modes, allowing researchers to tailor their experiments. It can perform random injections, randomly selecting a thread and a clock cycle before injecting a fault. Moreover, it offers advanced capabilities such as profile-guided injections, which utilize profile data to optimize the fault injection process. This versatility makes SASSIFI highly suitable for evaluating commercial GPUs, providing a layered approach to understanding the impact of faults at various abstraction levels.

Regarding fault injection, SASSIFI can manipulate instructions executed by the GPU directly within the CUDA kernels. For instance, during an instruction-level fault injection, SASSIFI might alter a specific SASS instruction by changing its opcode, effectively modifying the operation being performed. This kind of manipulation can simulate the potential impact of a fault, helping to analyze the repercussions of minor alterations in the instruction stream on the overall computation.

Furthermore, register file faults are simulated by modifying the values stored in the registers during runtime. For example, a bit-flip fault in a register could be simulated by inverting a single bit, changing the value stored, and potentially altering the program's behavior. This kind of fault can be introduced at various moments during execution to study the temporal effects of these faults, providing insights into how a SEU might propagate through the system.

In conclusion, SASSIFI's detailed and comprehensive approach to fault simulation at the instruction and register levels makes it a valuable tool for assessing the vulnerabilities in Nvidia GPUs that may arise from radiation-induced faults in commercial GPUs.

3.4.2 Fault Simulator for softcore GPUs

Fault injection tools at the RTL offer a distinct advantage in terms of precision and control. Operating at this level allows for accurately modeling and assessing hardware faults. They provide more realistic scenarios for fault injection, thus enabling detailed analysis and more effective fault-tolerance strategies.

In our framework, the fault injection process is orchestrated by a host implemented in Python. Initially, the host system initiates a clean simulation run to gather baseline data about the application, including its execution time and output memory state (GOLDEN results). Upon acquiring this information, the host randomly selects a signal bit from a predefined list and also randomly determines a time window within the execution time of the application. The simulation is then paused, and the selected bit's state is toggled (0 to 1 or 1 to 0) through a command at the RTL level. This approach allows for high-precision fault injection tailored to specific design aspects.

Simulations can be run using either ISIM or ModelSim, both of which are capable of compiling and executing the design. The fault injection logic is implemented in TCL, while the host logic is in Python. Inputs to the fault injection system include a list of design signals to be upset, golden results for reference, and a flag signal indicating fault detection, which can subsequently be used to implement fault-tolerance mechanisms. The fault injection operates autonomously for a predetermined number of faults.

3.4.3 Fault Emulator for FPGAs

The emulation-based fault injection engine from the Universidade Federal do Rio Grande do Sul (UFRGS) stands out for its comprehensive architecture and unique features (BENEVENUTI; KASTENSMIDT, 2019). The engine for Fault Emulation in FPGAs is built around the Xilinx Internal Configuration Access Port (ICAP). The system comprises five main modules:

- **Design Under Test (DUT):** The DUT interfaces with a test vector to represent the evaluated FPGA design.
- **Design Controller:** Manages parameters, test vectors, and golden results for the DUT.
- **Reporting Module:** Sends diagnostic information to a campaign coordinating computer.
- **Fault Injection Module:** Interacts with the Xilinx ICAP interface to read and write into the FPGA's configuration memory frames, thereby injecting faults.
- **System Controller:** Coordinates the entire fault injection process, including fault injection, DUT diagnosis, and reporting.

To ensure the reliability of the fault injection process, floorplanning constraints are applied to define the Area Under Test (AUT). All logical modules of the Circuit Under Test (CUT) must be contained within this AUT. The tool injects single faults at randomized locations within the AUT, achieved through an FPGA-based pseudo-random number generator. The fault injection operates autonomously for a predetermined number of faults. One key feature is that it does not accumulate the faults over time; instead, it provides a snapshot of the design's sensitivity to different faults.

The evaluation of radiation-induced faults and their effects on semiconductor circuits emphasizes the importance of assessing system reliability under varying radiation conditions. By understanding these interactions, we can identify the vulnerabilities in electronic systems and establish methodologies for reliability assessment. The methods, including simulation, particle accelerator testing, and fault emulation, provide comprehensive insights into how faults propagate and affect system performance. These insights are crucial for developing robust fault tolerance techniques.

As we transition to the next Chapter, it is evident that the knowledge gained from reliability assessments lays a strong foundation for devising fault tolerance strategies. These strategies are essential to ensure the continuous and reliable operation of systems, especially in radiation-prone environments. In the following Chapter, we will explore various fault tolerance techniques for GPUs, focusing on architectural transformations in software, hardware, and hybrid solutions that can enhance system resilience against radiation-induced errors.

4 FAULT TOLERANCE TECHNIQUES FOR GPUS

Fault tolerance refers to the ability of a system to continue functioning correctly even in the presence of hardware or software faults. It is crucial in various domains, including scientific computing, autonomous vehicles, and data centers, where system reliability is paramount.

While numerous fault tolerance techniques span different levels of system design, from technology changes to layout-level modifications, this Thesis focuses on architectural transformations that can be applied in software, hardware, or a hybrid of both. Technological improvements in semiconductor materials and modifications in the layout of transistors, such as the insertion of guard rings or the design of trench isolation, fall outside the scope of this work. Our discussion will not cover these structural changes but will instead center on the strategies that enhance fault tolerance at the architectural level.

Fault tolerance techniques at the architectural levels are essentially implemented through transformations in either software, hardware or a combination of both. Three fundamental components often form the basis for these transformative approaches:

- **Redundancy**: In both hardware and software contexts, redundancy involves replicating resources for fault tolerance. In hardware, this often means duplicated components like processors or memory modules. In software, redundancy can be achieved through replicated variables or memory locations. These replicated resources are primarily used for error detection and correction.
- **Checking**: This step involves comparing the outputs or states of the redundant resources to identify discrepancies. Any mismatch is a strong indicator of an error, making this step crucial for error detection.
- **Notification**: The system initiates corrective measures or alerts the relevant parties once an error is detected. Detection is especially important in contexts where immediate correction may not be feasible, but knowing about the error is crucial.

Strategies like selective hardening and approximate computing are introduced to enhance the effectiveness of these key components. Selective hardening is a strategy that refines resource allocation for fault tolerance, often reducing performance overhead while possibly compromising fault coverage. On the other hand, approximate computing serves as a paradigm that allows for a certain level of acceptable error, thus potentially reducing the need for strict checking procedures.

The following sections will delve into specialized approaches and tools designed to implement these core components and strategies, alongside a review of related works. These include Software-Based Techniques, Hardware-Based Techniques, ISA Extensions, Approximate Computing and Criticality, Selective Hardening, and Supporting Tools For Fault Tolerance.

4.1 Software-Based Techniques

Software-based fault tolerance techniques for GPUs can be implemented across various levels of abstraction, from high-level frameworks like CUDA for NVIDIA GPUs, where threads and variables are replicated, to low-level approaches, such as assembly, involving replicating instructions and registers. Additionally, Algorithm-Based Fault Tolerance (ABFT) techniques offer high detection rates with low execution time overheads but cater to a specialized set of applications. Most of the existing literature has concentrated on high-level abstraction techniques for fault tolerance, with key contributions from Oliveira et al. (OLIVEIRA et al., 2014), Wadden et al. (Wadden et al., 2014), and Gupta et al. (GUPTA et al., 2017).

The work by (OLIVEIRA et al., 2014) serves as an insightful example of leveraging the intrinsic parallelism of GPUs for fault tolerance through *Duplication With Comparison* (DWC). DWC is implemented at the application level by duplicating blocks or threads and comparing the results. The study explores spatial and temporal DWC strategies, highlighting their flexibility and adaptability in utilizing the parallel architecture of GPUs. However, these benefits come with performance trade-offs, ranging from 90% to 151% degradation. Additionally, the authors emphasize that duplicated processes must be carefully distributed to prevent errors in shared resources, such as cache memories or critical resources like the scheduler, from propagating to both copies, thereby compromising the method's detection capabilities.

Wadden et al. (Wadden et al., 2014) proposed a compiler-based approach that converts GPU kernels into redundantly threaded versions. They observed high overheads for inter-thread communication and synchronization, with performance costs being application-dependent and exceeding 100% in some cases. Gupta et al. (GUPTA et al., 2017) extended this work by introducing compiler optimizations to reduce synchronization overhead. Software-based instruction-level duplication was found to incur lower synchronization overheads, as both duplication and consistency checking are performed within each thread (GONCALVES et al., 2017; Mahmoud et al., 2018).

Intermediate languages offered by vendors have also been explored for fault tolerance. Specifically, the work in (KALRA et al., 2020) employed heuristics to identify and protect critical instructions at the PTX-level, achieving a significant reduction in DUEs and SDCs, underscoring the potential of PTX-level interventions to bolster system reliability without comprehensive code duplication.

More recently, low-level approaches have been explored for fault tolerance in GPUs, which will be the focus of the next section.

4.1.1 Low-Level Software-Based Techniques

Program code transformations at the low level for enhancing reliability must incorporate the three fundamental components of fault tolerance—Redundancy, Checking, and Notification—initially proposed by Oh et al. (OH; SHIRVANI; MCCLUSKEY, 2002). In the context of code transformations, these components are implemented as follows:

- ***Datapath Duplication***: This transformation aligns with the concept of Redundancy. It duplicates all datapath operations, requiring a duplication of the registers used. By leveraging static code analysis, *spare registers* are identified for duplication. This modification effectively doubles the number of datapath operations being executed, thereby exploiting Instruction Level Parallelism (ILP) more effectively.
- ***Consistency Checking***: This transformation corresponds to Checking. It verifies the consistency between the original and duplicated datapath by using comparison instructions followed by conditional branching to an error subroutine. This modification introduces a data dependency, which affects the ILP gains.
- ***Host Notification***: This transformation is related to Notification. In the event of fault detection, it notifies the host machine. This modification could be either a trap instruction or a memory write instruction to the global memory. The execution of this transformation is conditional upon the outcome of the *Consistency Checking*.

Table 4.1: Code transformations example

Original Code	**Hardened Code**	**Description (Code Transformation)**
ADD R1, R2, R3	ADD R1, R2, R3	Original Datapath
	ADD R1’, R2’, R3’	Datapath Duplication
	CMP R1, R1’	Consistency Checking
	TRAP	Host Notification

As shown in Table 4.1, the original assembly instruction ‘ADD R1, R2, R3‘ undergoes a series of transformations for fault tolerance. The instruction is first duplicated, represented by ‘ADD R1’, R2’, R3", followed by a consistency check ‘CMP R1, R1", and finally a host notification via a ‘TRAP‘ instruction.

Building on the foundational work of Oh et al., Gonçalves et al. (GONCALVES et al., 2017) applied these transformations to the assembly code of a softcore GPU. They focused on detecting faults in the GPU’s register files by intertwining replicated assembly instructions. This approach led to a 99% reduction in errors at the cost of a 78% increase in execution time.

More recently, the work presented in (Mahmoud et al., 2018) introduces a set of software-based fault tolerance techniques collectively known as SInRG. These code transformations are implemented directly in NVIDIA's production compiler and have been evaluated on commercial GPUs. The naive approach of immediate duplication and verification resulted in a performance overhead of 69%. However, by deferring host notification, the authors were able to reduce this overhead to 39%. Architectural fault injection campaigns demonstrated an average SDC coverage exceeding 87%.

The authors also explored using a single register space to verify each instruction result immediately. While this approach reduced register allocation, it increased the average runtime overhead to 49% and introduced a vulnerability, as the source registers for the instructions were left unprotected. These results suggest that a selective hardening strategy targeting critical registers or instructions could further optimize SInRG's runtime performance and resource utilization.

In this Thesis, we extend the work of Gonçalves et al. (GONCALVES et al., 2017) by leveraging assembly code transformations for software-based fault tolerance, independent of the compiler or high-level programming languages. Specifically, we introduce three novel optimizations—*Traceback*, *Move*, and *Delayed Notification*—aimed at reducing performance overhead while improving fault detection capabilities.

4.2 Hardware-Based Techniques

Hardware-based fault-tolerance techniques offer reliability enhancements directly at the hardware level. These methods are generally more transparent to the application and often incur less performance overhead than their software-based counterparts. Notable techniques include parity bits, Error-Correcting Code (ECC), and Triple Modular Redundancy (TMR).

4.2.1 Parity Bits

Parity bits are a straightforward yet effective error detection method commonly employed in commercial devices. By adding an extra bit to each data unit, this technique ensures that the number of '1' bits in the data, including the parity bit, is either always even or always odd. This simple check allows for detecting errors when data is read or received. However, the utility of parity bits is limited in several ways.

Firstly, they can only detect an odd number of bit errors and cannot identify which specific bit is incorrect. Unlike more advanced techniques like ECC, parity bits offer no error correction capabilities. Additionally, while the overhead is generally minimal, including parity bits does slightly increase the size of the data, which could be a concern in scenarios where bandwidth or storage is limited. Calculating and verifying parity bits also

Table 4.2: Graphical explanation of parity technique with error detection

Original Data	Parity Bit (Even)	Data with Parity	Transmitted Data	Error Detected
1101	1	11011	11011	No
1011	1	10111	10111	No
1001	0	10010	10010	No
1111	0	11110	11110	No
1101	1	11011	11010	Yes

introduces a small computational overhead, especially when dealing with large volumes of data. Lastly, the technique lacks the flexibility for selective hardening, where only critical data or operations are protected.

In summary, while parity bits provide a basic level of error detection, they are often used in conjunction with other, more robust fault-tolerance techniques to enhance system reliability.

4.2.2 Error-Correcting Codes - ECC

Error-Correcting Codes (ECC) is a prominent hardware-based fault tolerance technique commonly integrated into commercial devices. It is designed to identify and correct errors at the hardware level automatically. ECC is particularly effective at detecting and correcting single-bit errors, making it a reliable choice for many applications where data integrity is crucial. These computations generate redundant check bits from the original data bits through specific algorithms. When data is read, ECC algorithms compare the stored check-bits with the current data state to detect and correct any discrepancies caused by bit errors. This process is adept at correcting single-bit errors and detecting double-bit errors, employing the Single Error Correct, Double Error Detect (SEC-DED) scheme to ensure data integrity even with minor corruption.

However, ECC comes with its own set of drawbacks. One of the primary limitations is the performance overhead incurred due to the extra computations needed for error detection and correction. Additionally, ECC often requires extra hardware resources, leading to increased area overhead on the chip. Another limitation is its inflexibility for selective hardening, as ECC is generally applied uniformly across the hardware, making it less adaptable to specific application needs.

Notably, the work by (OLIVEIRA et al., 2014) and (Tiwari et al., 2015) demonstrated increased DUEs under radiation experiments when ECC was enabled. This increase suggests that while ECC is effective under certain conditions, there may be more reliable options in some scenarios, particularly in radiation-sensitive environments.

4.2.3 Triple Modular Redundancy - TMR

Triple Modular Redundancy (TMR) enhances system reliability by triplicating hardware modules and using a voting mechanism to produce a single output. This method is highly effective for mitigating single and multiple fault scenarios, ensuring system functionality as long as at least two of the three modules are correct.

TMR's flexibility extends to hybrid implementations that combine hardware and software modules, allowing for more efficient resource allocation in systems with limited hardware. Variants like Selective TMR focus on triplicating only the critical portions of the code or hardware, balancing resource overhead with fault tolerance. Adaptive TMR takes this further by dynamically selecting hardware and software modules based on the system's current status and available resources.

However, TMR comes with its own set of challenges. The technique significantly increases hardware resource requirements and power consumption, which can be problematic in energy-sensitive applications. The added complexity in implementing TMR, especially in its hybrid or adaptive forms, complicates system design and verification. While TMR is effective against single faults, its efficacy diminishes against correlated or multiple faults affecting more than one module simultaneously.

In the context of GPUs, implementing TMR poses challenges due to the proprietary nature of COTS GPUs. Most studies in this area have relied on simulation for verification and validation. Thus, our proposal takes a novel approach by implementing TMR directly into a softcore GPU, allowing for empirical testing and providing a more realistic evaluation of its effectiveness.

4.3 Software-Hardware Approaches: ISA Extension

Hybrid solutions that combine software and hardware approaches offer the best of both worlds: hardware performance and software flexibility. However, the complexity of implementing these solutions is a drawback, as they require both hardware and software modifications. Instruction Set Architecture (ISA) extensions are a prime example of this approach. They allow for the creation of new instructions by identifying available bits in the instruction code, followed by hardware updates to execute these new instructions. In the context of reliability, ISA extensions can handle key components like replication, comparison, and notification while introducing atomic instructions to enhance specific operations' reliability.

Recent works on this topic have explored hybrid approaches to improve GPU resilience. For instance, a cooperative hardware-software mechanism leverages the register file ECC hardware to detect pipeline errors, achieving low instruction-duplication overhead (SULLIVAN et al., 2018). Another study introduces a new XOR instruction for hardware-based host notification, thus eliminating the need for separate consistency checks and notification instructions. This work also proposes hardware-based hardening steps, updating a signature register with each instruction's execution. An extra metadata bit in the instructions informs the hardware when to update this register. While promising, these ISA extensions have yet to be empirically tested (Mahmoud et al., 2018).

However, it's worth noting that the landscape for GPUs differs from that of microprocessors and FPGAs. Many studies, including those proposing ISA extensions for commercial GPUs, have not been empirically tested on real hardware, relying solely on simulation for verification.

4.4 Approximate Computing and Criticality

In certain applications, both safety-critical and otherwise, there exists a tolerance for inaccuracy within a predefined range (MITTAL, 2016). For example, seismic wave applications can tolerate misfits up to 4% (GUAN et al., 2015). This result flexibility opens the possibility that a corrupt output may be correct if it falls within the acceptable accuracy range. Recent studies have even advocated relaxing strict output correctness to improve performance and efficiency (VENKATAGIRI et al., 2018).

One approach to leveraging approximate computing is applying approximate computing techniques to the original program to minimize its execution time before hardening (APONTE-MORENO; PEDRAZA; RESTREPO-CALLE, 2019). Subsequently, fault tolerance techniques are applied to the resulting assembly code, where all registers are protected. This method has shown up to a 53% reduction in overhead for an error acceptance rate of 10% while maintaining fault coverage. However, these studies often overlook the potential for selective hardening strategies that could further optimize performance and resource usage.

In the context of algorithms based on loop computations, the impact of transient faults can be mitigated by simply increasing the number of iterations (RODRIGUES et al., 2019). This inherent fault tolerance allows for small discrepancies in data values to be managed and corrected over multiple iterations. While more iterations introduce additional latency, they also make the system more resilient to faults.

Moreover, defining a threshold for error acceptance can enhance system reliability against transient faults without requiring any modifications to the original application (RODRIGUES et al., 2019; VENKATAGIRI et al., 2018). For example, fault injection experiments have shown that most faults do not cause output errors larger than a predefined percentage, thereby increasing system resilience.

Another direction to explore is using mixed-precision architectures in conjunction with approximate computing (SANTOS et al., 2020). This approach has shown promising results in reducing runtime overhead while maintaining an acceptable level of fault coverage, particularly in high-performance computing applications.

4.5 Selective Hardening

Selective hardening focuses on the most critical components of a system to provide fault tolerance where it is most needed. This approach is universally applicable, seamlessly integrating with all the fault tolerance techniques discussed in this Chapter across various levels of abstraction. Selective hardening optimizes resource allocation while maintaining system reliability, whether in hardware-based, software-based, or hybrid solutions.

Selective hardening techniques based on this topic have been developed to offer a balanced trade-off between performance cost and fault coverage. By selectively duplicating key structures — such as instructions, threads, and registers — these techniques manage to significantly reduce performance overhead without compromising reliability (SUNDARAM et al., 2008; KALRA et al., 2020).

In this Thesis, we suggest a strategy that employs selective hardening in conjunction with approximate computing to assess and optimize the protection of the most vulnerable application registers in a commercial GPU. Separately, we also implement selective TMR on a softcore GPU.

4.6 Supporting Tools for Fault Tolerance

While this Chapter primarily focuses on fault tolerance techniques for GPUs, it is crucial to highlight the tools that support the implementation of these techniques (SWIFT; REHMAN; SMITH, 2005; CRAFT; SMITH; JOHNSON, 2012). Among these, the Hardening Post Compiling Translator (HPCT) stands out for its simplicity and versatility in enabling fault tolerance (AZAMBUJA et al., 2011). Developed at UFRGS, HPCT is crafted in Java, benefiting from the language's robustness and platform-independent nature, making it an ideal choice for various development settings.

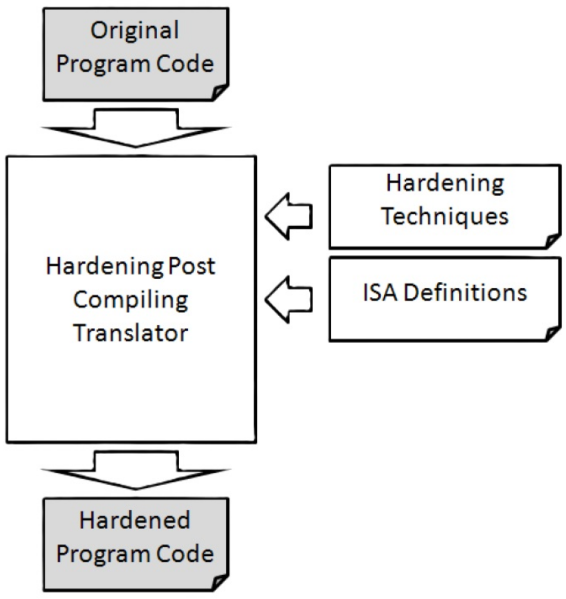
HPCT aids in converting original program codes into versions that are more resilient to faults. It operates independently of the programming language and compiler, working directly with binary code to produce processor-specific, fault-tolerant binary code. The operation of HPCT encompasses several essential steps:

- **Analysis:** It analyzes the binary code to identify branch instructions, the program flow graph, and registers for potential hardening use.
- **Transformation:** HPCT modifies instructions and instruction blocks, including replication for fault tolerance and updating addresses for branch instructions as needed.
- **Execution Flow Graph Extraction:** Constructs the program's execution flow graph to pinpoint critical code sections for hardening.
- **User Interface:** Features a graphical user interface (GUI) allowing users to select

fault tolerance techniques and specify processor architecture details.

- **Output:** Generates transformed binary code with integrated fault tolerance mechanisms, ready for deployment on the target processor.

Figure 4.1: HPCT workflow diagram.



Source: The Author.

In summary, HPCT processes the program's binary code, applying chosen hardening techniques alongside ISA definitions and processor architecture details provided by the user through a GUI. This workflow, detailed in Figure 4.1, showcases the adaptability and comprehensive functionality of HPCT, rendering it an invaluable tool for improving application reliability across different computing platforms.

4.7 Related Works Overview

This section summarizes the research contributions discussed throughout this Chapter, laying the groundwork for fault tolerance techniques in GPUs. The presented works span a range of approaches, including software-only methods, hardware enhancements, hybrid techniques that integrate both software and hardware innovations, and optimization strategies such as Selective Hardening (SH) and Approximate Computing (AC).

The Table 4.3 encapsulates the contributions of each cited work, categorized by architectural approach and optimization strategies. Works without a checkmark were focused on evaluation only, without introducing software or hardware modifications for protection. This summary serves as a reference for the current state of the art in GPU fault tolerance techniques and a precursor to our proposed advancements. The works listed in italics are our core contributions, which will be detailed in the following Chapters.

The fault tolerance techniques for GPUs discussed in this Chapter highlight the importance of ensuring the reliable operation of GPUs in the presence of faults, particularly radiation-induced faults. The various strategies, from software-based methods to hardware-based and hybrid approaches, provide a comprehensive toolkit for enhancing system resilience.

With a solid understanding of these fault tolerance techniques, we can now explore their practical application through case-study architectures. In the next Chapter, we will delve into the GPU architectures utilized to evaluate the reliability of GPU models. The focus will be on the specific GPU architectures chosen for these case studies, setting the stage for the experimental evaluations presented in the subsequent chapters

Table 4.3: Summary of research contributions in fault tolerance techniques for GPUs

Reference	Architectural Approaches			Optimization Strategie	
	SW	HW	Hybrid	SH	AC
(OLIVEIRA et al., 2014)	✓				
(Wadden et al., 2014)	✓				
(GUPTA et al., 2017)	✓				
(SUNDARAM et al., 2008)	✓				
(GONCALVES et al., 2017)	✓				
(Mahmoud et al., 2018)	✓	✓	✓		
(KALRA et al., 2020)	✓			✓	
(SULLIVAN et al., 2018)	✓	✓	✓		
(MITTAL, 2016)	✓				✓
(GUAN et al., 2015)	✓				✓
(VENKATAGIRI et al., 2018)	✓				✓
(APONTE;PEDRAZA;RESTREPO,2019)	✓				✓
(RODRIGUES et al., 2019)	✓				✓
(SANTOS et al., 2020)	✓				✓
(Tiwari et al., 2015)					
(GONCALVES et al., 2019)	✓			✓	
(GONCALVES et al., 2020)	✓			✓	✓
(GONCALVES et al., 2020)	✓	✓	✓	✓	
(GONCALVES et al., 2022)	✓			✓	
(BRAGA; GONÇALVES; AZAMBUJA, 2023)	✓	✓	✓		
(BRAGA; GONCALVES; AZAMBUJA, 2023)	✓	✓	✓		
(GONCALVES et al., 2020)					
(BRAGA et al., 2021)		✓		✓	
(BENEVENUTI et al., 2022)					
(PEREZ et al., 2022)					

Note: References in *italic* indicate works that are further discussed in this Thesis. SW: Software-Based Techniques, HW: Hardware-Based Techniques, Hybrid: Hybrid (ISA Extension) Approaches, SH: Selective Hardening, AC: Approximate Computing.

5 CASE-STUDY ARCHITECTURES

This Chapter explores the architectures of NVIDIA's Kepler, FlexGrip, and FGPU, highlighting their contributions to computational efficiency and adaptability. Kepler GPUs are celebrated for their advancements in HPC and graphics, laying the foundation for future computational innovations. FlexGrip and FGPU, leveraging FPGA platforms' flexibility, offer tailored solutions for handling complex computations. This examination details each architecture's operational mechanisms and internal configurations.

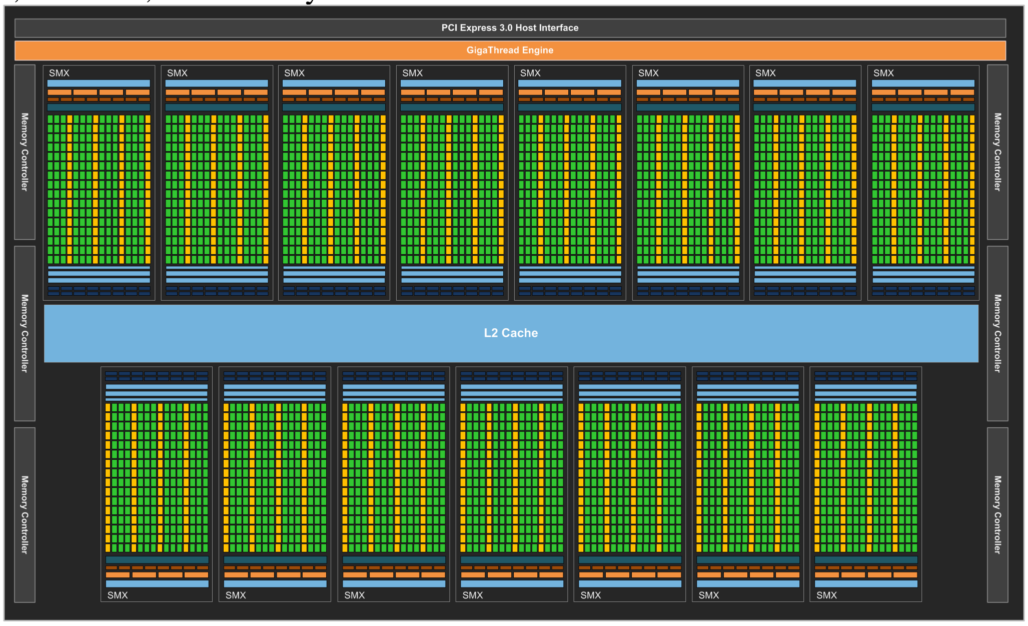
5.1 NVIDIA Kepler Architecture

The NVIDIA Kepler architecture, exemplified by the K20 and K40 GPUs, is designed to excel in scientific computing. The architecture is a paradigm of computational efficiency and power.

The K20 GPU is foundational to the Kepler series, with 2,496 CUDA cores and up to 5.6 GB of memory, engineered for demanding scientific tasks. Advancing this design, the K40 GPU comprises 2,880 CUDA cores and doubles the memory to up to 12 GB, catering to the most complex scientific computations.

Figure 5.1 shows the NVIDIA Kepler GPU architecture layout. It features a network of interconnected SM units, each a hub of parallel processing capability. The diagram also exhibits memory controllers that orchestrate the flow of data between the GPU and its memory and a large L2 cache that serves as a buffer for frequently accessed data, enhancing the overall speed and efficiency of the system.

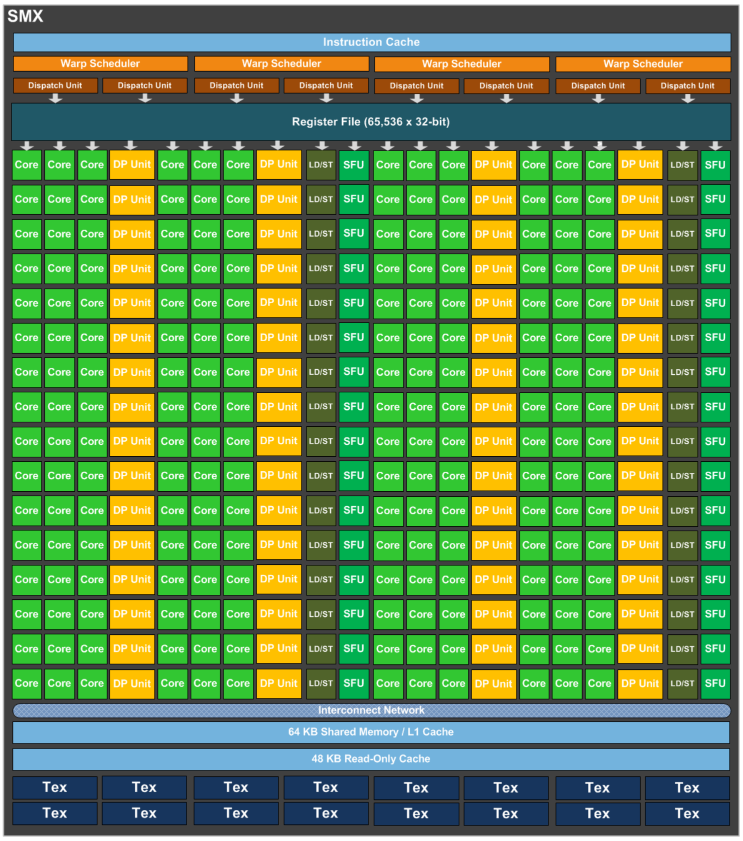
Figure 5.1: NVIDIA Kepler GPU architecture, showcasing the interconnection of SM units, L2 cache, and memory controllers.



Source: (NVIDIA Corporation, 2012).

Figure 5.2 provides a detailed view of a Kepler architecture SM unit, illustrating its key components and their interactions. CUDA cores are arrayed to perform parallel processing effectively. Warp schedulers and dispatch units direct the instructions flow, optimizing core utilization. Register files enable quick data retrieval, while shared memory facilitates thread communication and synchronization within a warp. The texture units (Tex) are specialized for texture mapping and data sampling tasks. The diagram also shows the L1 cache for fast access to data and instructions, reducing latency. Special Function Units (SFU) handle complex mathematical functions, and Double Precision (DP) units ensure precise calculations.

Figure 5.2: Internal structure of a Kepler SM unit, highlighting the CUDA cores and control logic.



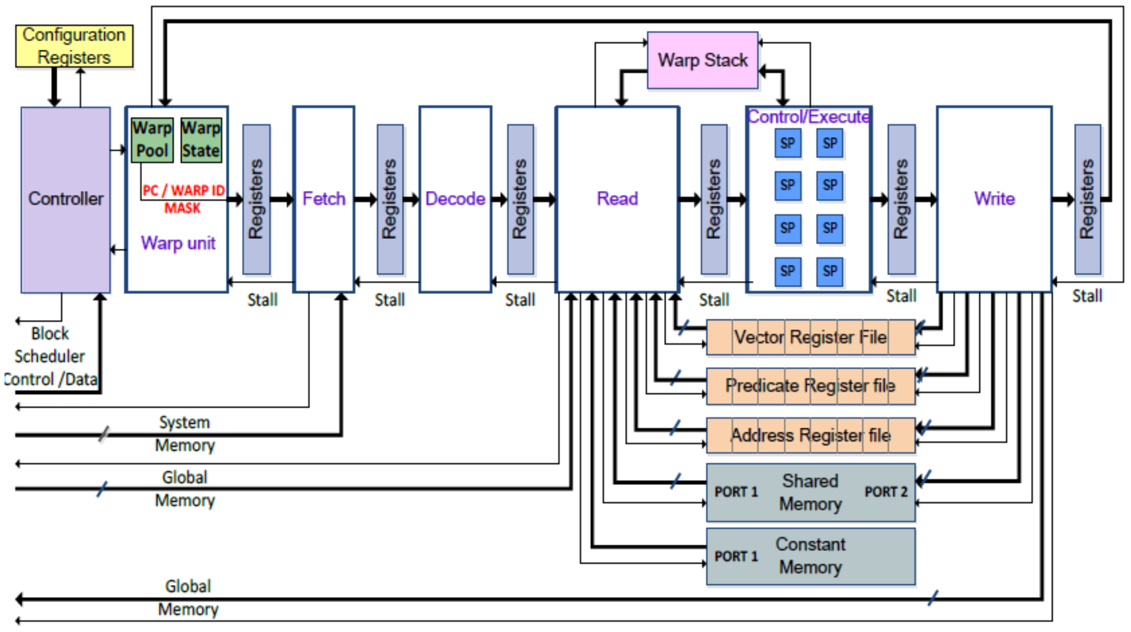
Source: (NVIDIA Corporation, 2012).

The CUDA programming model provides developers with the tools to harness the Kepler GPUs' capabilities. Programs are written in CUDA C/C++, which are then translated into PTX language, allowing the GPU to perform complex computations with high throughput.

The deployment of K20 and K40 GPUs within prominent supercomputers, such as Titan and Stampede, demonstrates their substantial contributions to computational science, managing large-scale, complex simulations with unparalleled efficiency.

5.2 Flexible Graphics Processor - FlexGrip

Figure 5.3: General microarchitecture of an SM core in the FlexGrip model.



Source: (ANDRYC; MERCHANT; TESSIER, 2013).

Flexible Graphics Processor (FlexGrip) is an open-source, configurable softcore general-purpose GPU model described in VHDL. It implements the NVIDIA G80 microarchitecture and supports the CUDA programming environment. The GPU model is compatible with up to 52 assembly instructions (SASS) and follows the Single-Instruction Multiple-Thread (SIMT) paradigm (ANDRYC; MERCHANT; TESSIER, 2013).

The FlexGrip architecture consists of an array of Streaming Multiprocessors (SMs) that execute threads in parallel. Each SM is managed by a Block Scheduler Controller, which distributes the workload to each available SM in the system. Internally, the SM is divided into a five-stage pipeline: Fetch, Decode, Read, Execute, and Write. The pipeline is managed by a Warp Scheduler Controller that oversees the concurrent execution of a group of 32 parallel threads, also known as a warp.

The Block Scheduler Controller manages the SMs and distributes workloads to each available SM. Warps are fetched, decoded, and distributed to be processed in the Scalar Processors (SPs) at the Execute stage. The Read and Write stages handle the loading and storing of data operands from and to various types of memory.

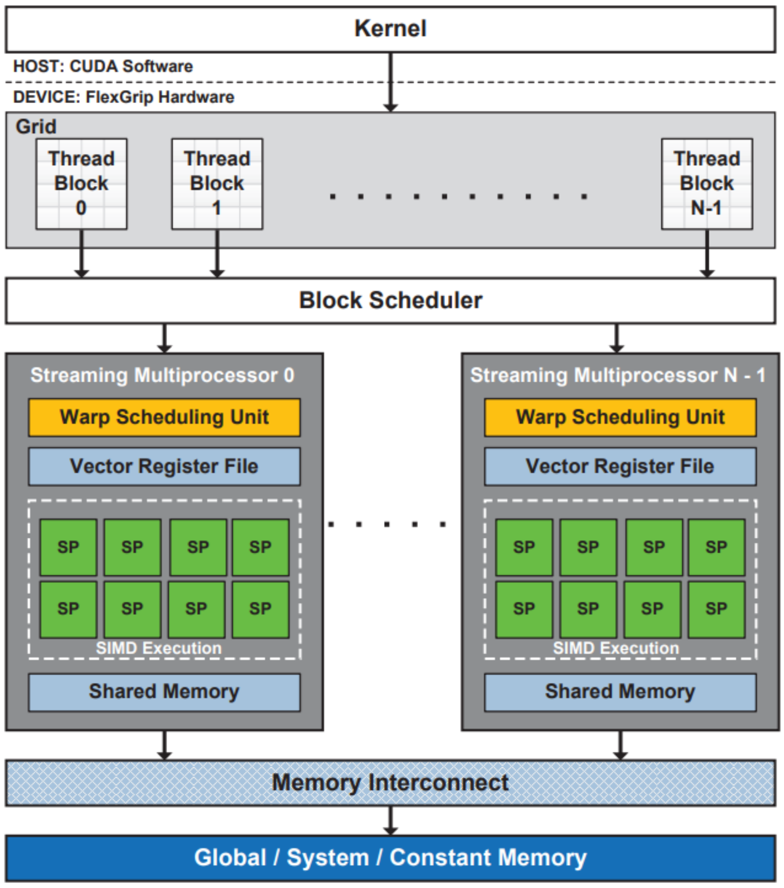
Pipeline Registers (PRs) are situated between the pipeline stages to store both data path and control path signals. These registers are crucial for exploiting Instruction-Level Parallelism (ILP) and ensuring high-performance parallel execution. The number of PRs varies according to the number of SMs and SPs, affecting both data integrity and control flow.

The system memory in FlexGrip includes a General-Purpose Register File (GPRF), an Address Register File (ARF), a Predicate Register File (PRF), local memory (L_mem), constant memory (C_mem), shared memory, and global memory. The GPRF, ARF, and PRF are located inside the SM and are organized in banks. The GPRF is the primary and fastest memory resource, with a size that can vary depending on the configuration of the SM.

FlexGrip can be configured to operate with 8, 16, or 32 cores inside each SM, providing flexibility in modifying the data path length in the pipeline registers and the register size per core in each bank of the GPRF. The number of registers per thread in the GPRF is application-dependent and can reach up to 64 registers per thread on each bank.

As presented in section 2.1.1, in an NVIDIA GPU, the kernel is invoked by a host, typically a CPU. This process involves dispatching the kernel code to the GPU's System Memory for execution. Before launching an application, the GPU must be configured by defining the grid's size and the number of blocks and threads. Moreover, the data to be processed by the threads must be loaded into global memory. To execute an application on FlexGrip, these configuration parameters and the initial states of shared and global memories are directly set within the GPGPU's HDL. The kernel instructions, which represent the algorithm to be executed, are also included within the HDL.

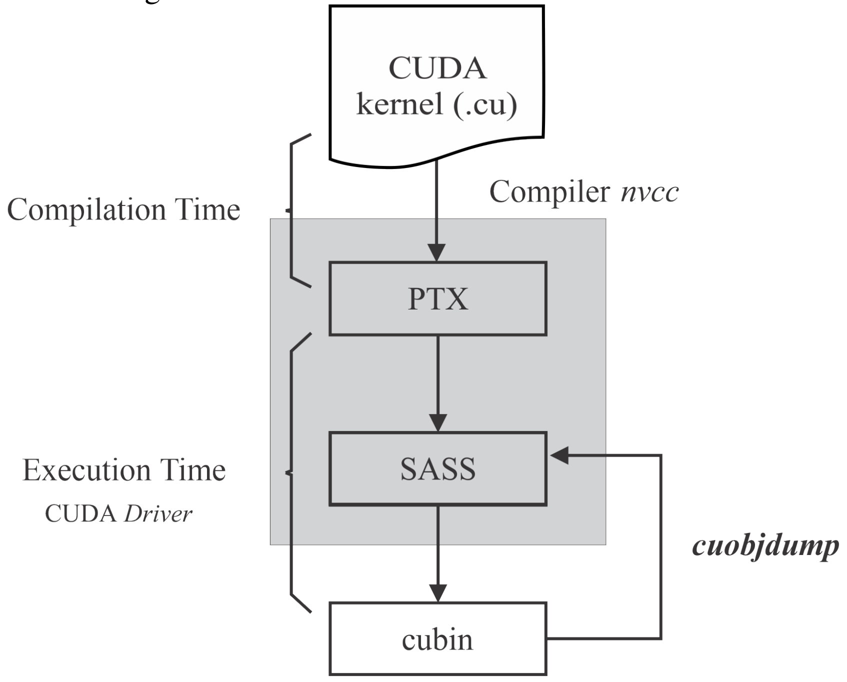
Figure 5.4: FlexGrip GPU with CUDA software during kernel execution.



Source: (ANDRYC; MERCHANT; TESSIER, 2013).

Figure 5.4 illustrates the FlexGrip GPU microarchitecture and its interface with CUDA software during kernel execution. The diagram shows a high-level overview of the execution flow from the host to the device. The kernel, once invoked, is managed within the FlexGrip hardware framework, which is depicted as a series of thread blocks within a grid.

Figure 5.5: Software flow of an NVIDIA GPU

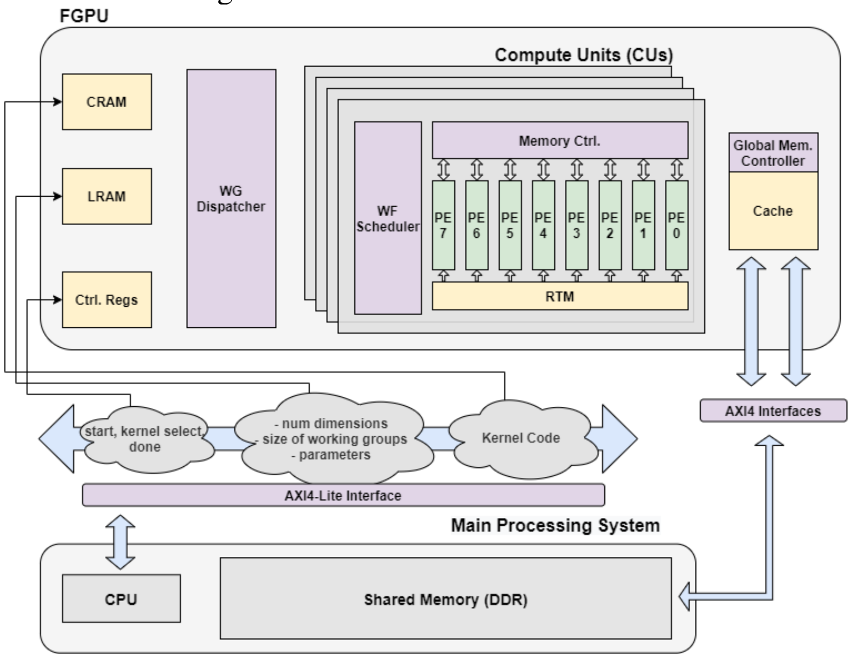


Source: The Author.

The process of compiling CUDA algorithms is shown in Figure 5.5. During compile time, the compiler receives the CUDA kernel, converting it into PTX. During runtime, the PTX code goes through the CUDA driver, which is responsible for generating the binary cubin code, which is then directed to the GPU for effective execution. During the compilation process, the PTX code is converted to SASS; this format represents the native instructions that are interpreted by NVIDIA's hardware. However, the SASS code is generated at runtime and is not visible to the end user. Still, this code can be obtained through the cuobjdump tool provided by NVIDIA's CUDA toolkit. This tool generates the SASS code from the previously described cubin code. Finally, to execute a kernel on FlexGrip, the hexadecimal codes resulting from the SASS file must be copied to FlexGrip before running an application.

5.3 FPGA general purpose Graphical Processing Unit - FGPU

Figure 5.6: Softcore FGPU overview.



Source: The Author.

FPGA general purpose Graphical Processing Unit (FGPU) is a configurable, open-source SIMT softcore processor optimized for FPGA platforms. Still, it can also be ported to ASIC platforms with precise adaptations. It is designed to accelerate workloads that fit within the SIMT paradigm and features an OpenCL compilation framework. The FGPU's architecture, as depicted in Figure 5.6, is modular, focusing on scalability and flexibility (KADI et al., 2018).

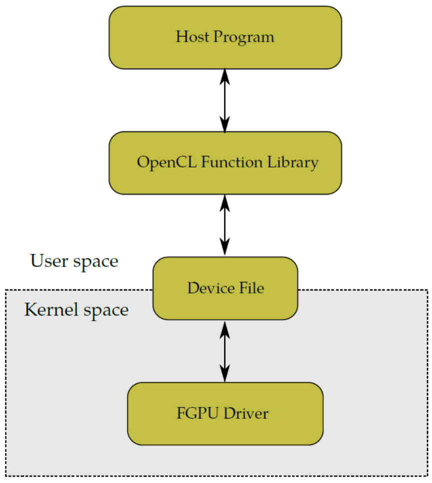
The core component of FGPU is the Compute Unit (CU), a Single-Instruction Multiple Data (SIMD) machine consisting of eight identical Processing Elements (PEs) labeled PE0 - PE7. These PEs come with dedicated Register Files (RFs), Arithmetic-Logic Units (INT), and optional hardware-implemented Floating-Point Units. The CU can be spatially replicated up to eight times, and a single CU can run up to 512 work items, which are computational kernels in OpenCL.

FGPU features a Runtime Memory (RTM) and a dedicated Data Cache, a central, direct-mapped, multi-port, and write-back system capable of simultaneously serving multiple read/write requests. It also integrates numerous data movers that can parallelize the data traffic on up to four AXI Data interfaces.

FGPU supports full thread divergence, meaning each work item can take a different path in the Control Flow Graph (CFG). Work items are grouped into Wavefronts (WFs) that execute concurrently within a CU. These WFs are further combined into Work Groups (WGs), which share a Program Counter and are assigned to a specific CU. The size of these parameters (work items, WFs, and WGs) is entirely configurable during implementation.

For execution to commence, a host processor, typically a hardcore Arm® microprocessor, sends accelerator instructions and execution parameters via an AXI4-Lite interface. This information is stored in Code RAM (CRAM) and Link RAM (LRAM) within the FGPU. A start signal is also sent and stored in a control register, triggering the execution. The FGPU framework includes an LLVM-based OpenCL compiler with clang as the front end. The backend maps instructions exclusively to the FGPU ISA, which consists of 49 instructions. It supports soft- and hard-FP implementations for single-precision floating-point instructions, including addition, subtraction, multiplication, division, and comparisons. Hard-FP support is provided by Xilinx FP Operator (FPO) IPs, while soft-FP is implemented through LLVM.

Figure 5.7: FGPU software stack integration.



Source: (KADI et al., 2018).

Figure 5.7 illustrates the software stack architecture for FGPU. At the top level, the Host Program interacts with the FGPU through the standard OpenCL Function Library. These OpenCL functions are translated into system calls, which are then directed to the FGPU driver via a character device file in user space. This allows the Host Program to communicate with the FGPU without requiring root privileges or compromising system security. In the kernel space, the FGPU driver acts as a mediator, mapping the FGPU's control registers, CRAM, and LRAM into its address space. The driver manages memory allocations, including OpenCL buffers, and maintains cache coherency between the FGPU hardware and applications running on the host processor. This approach negates the need for traditional data transfers over a bus, thereby eliminating the typical bottleneck associated with separate host-device memory transfers.

The FGPU API provides the necessary functionality to control the device from the host, including downloading the executable code into CRAM and setting up the LRAM and control registers according to the OpenCL-API specifications. Through this API, developers can implement and run a wide range of GPGPU applications on the FGPU platform, leveraging its full capabilities for parallel data processing.

In the subsequent chapters, we will delve into specific case-studies, examining the experimental evaluations performed on each GPU architecture. These case-studies will provide detailed insights into the reliability and fault tolerance techniques applied to NVIDIA's Kepler, FlexGrip, and FGPU architectures, culminating in the exploration of ASIC implementations for enhanced fault resilience.

6 SELECTIVE FAULT TOLERANCE FOR REGISTERS IN NVIDIA KEPLER GPU

Like most computing devices, modern GPUs are designed with a Reduced Instruction Set Computing (RISC) architecture. RISC simplifies the source code in a sequence of basic instructions. One of the main differences between Complex Instruction Set Computing (CISC) and Reduced Instruction Set Computing (RISC) architectures is that CISC architectures allow instructions to read directly from the main memory, while RISC architectures require inputs to come from the register file only. Due to this, RISC is also known as a load-store architecture: data in the main memory must be loaded to the register file to be processed. Moreover, data in the register file is not accessible by the user in high-level programming languages. In high-level programming, the content of the registers must be stored in the main memory to make it accessible. From a reliability point of view, the register file is a critical resource, as knowing the probability of an error in a register propagating to the output may be sufficient to characterize the vulnerability of an application. The GPU's parallel architecture requires a wider register file than the traditional CPU, as thousands of active threads must be fed data. *Kepler* GPUs, as the one used in our study, have 64K registers.

As register files need to be extremely fast, they are built with the most advanced technology and are among the most vulnerable resources in modern computing systems. Moreover, as registers are integrated into the fabrication of computing cores, protecting registers ECC is more expensive than caches or Dynamic Random Access Memory (DRAM). GPU Register File is large (i.e., 65KB per streaming multiprocessor), and it is the fastest type of memory on GPU; thus, it is very hard to protect without increasing the circuit's power consumption (WUNDERLICH; BRAUN; HALDER, 2013; TAN et al., 2011). Even if ECC is present, it may be less effective than the ECC for caches or main memory as register words are hard to interleave. As such, the probability for Multiple-Bit Upsets (MBUs) in the register file is higher than in the caches or main memory, requiring a more advanced ECC than the classical Single Error Correction Double Error Detection (SECDED).

In the current study case, we aim to use selective hardening fault tolerance techniques by means of hardware and software to protect the most vulnerable registers of a state-of-the-art commercial GPU. To do so, we perform beam radiation experiments and architectural-level fault injection to evaluate the sensitivity of the GPU register file to radiation effects. We rank registers according to their vulnerability and apply selective fault tolerance techniques based on collected data (GONCALVES et al., 2019).

This Chapter presents a reliability exploration of the Nvidia Kepler GPU architecture. First, we delve into Selective Fault Tolerance for Register Files. Then, we explore the integration of approximate computing techniques to enhance this selective fault tolerance even further.

6.1 Methodology for Reliability Evaluation and Hardening

This section outlines the experimental framework employed to assess the reliability and fault tolerance of NVIDIA Kepler GPUs. Our approach includes neutron radiation tests, register file reliability assessments through simulation, and the automated application of hardening techniques.

6.1.1 Reliability Assessment through Neutron Radiation

We performed a radiation experiment through a neutron beam to evaluate the NVIDIA Kepler GPUs' reliability. We evaluated SDC and DUE Failure In Time (FIT) rates. The FIT rate provides a good metric, representing the number of SDCs and DUEs that happen every billion (10^9) hours. However, it offers limited visibility as it is impossible to correlate the observed errors and the caused faults (their causes). We chose three algorithms as case-study applications from Rodinia benchmark (CHE et al., 2009) as identified as good codes for evaluating processors' reliability (QUINN et al., 2015): (1) Hotspot, which simulates the heat dissipation on a surface; (2) Needleman-Wunsch (NW), used for DNA sequencing; and (3) Quicksort, an efficient sorting algorithm.

Radiation experiments were performed at the Los Alamos Neutron Science Center (LANSCE) facility, exposing two K40 NVIDIA GPUs to a neutron flux between $1.0 \times 10^5 n/(cm^2/s)$ and $2.5 \times 10^6 n/(cm^2/s)$ during 200 hours that, when scaled to the natural environment, should cover around 46 thousand years.

Tests have been performed for all case-study applications, where the host computer sent input vectors to both GPUs and compared results with a golden file to detect SDC effects. A watchdog detects crashes and hangs (i.e., DUEs). We collected at least 100 SDCs and 100 DUEs for each benchmark, which is sufficient to apply a Poisson distribution and restrict the error bars to less than 15% of the reported values. To ensure a realistic environment, we tuned the experiment to cause 10^{-3} faults/execution, ensuring that the probability of more than one neutron generating an error in a single code execution remains negligible.

6.1.2 Register File Reliability Assessment through SASSIFI

We performed a fault injection campaign to analyze better how SDC- and DUE-induced faults affect the Kepler GPU. By doing so, we evaluated the AVF, which is the probability of a low-level corruption in the register file propagating to the output vector and causing an error. The fault injection campaign was performed by NVIDIA SASSIFI (see Section 3.4.1). Based on the SASSI instrumentation tool, SASSIFI injects transient faults in the GPU's register file without disrupting the perceived final instructions schedule or register usage. Previous SASSIFI results have been successfully correlated with radiation experiments (SANTOS; RECH, 2017) and shown to be faster than state-of-the-art related GPU fault injectors (HARI et al., 2017).

The fault injection campaign was performed automatically on two K20 GPUs, targeting registers used by the applications. Faults have been injected randomly in time and location, one per program execution. At the end of each execution, results stored in memory were compared with a golden file in the same process as the radiation experiment. For each application version, we injected 10,000 faults, distributed among application-used registers, achieving ±1% statistical error considering a 95% confidence level (Leveugle et al., 2009).

6.1.3 Automated Hardening through HPCT

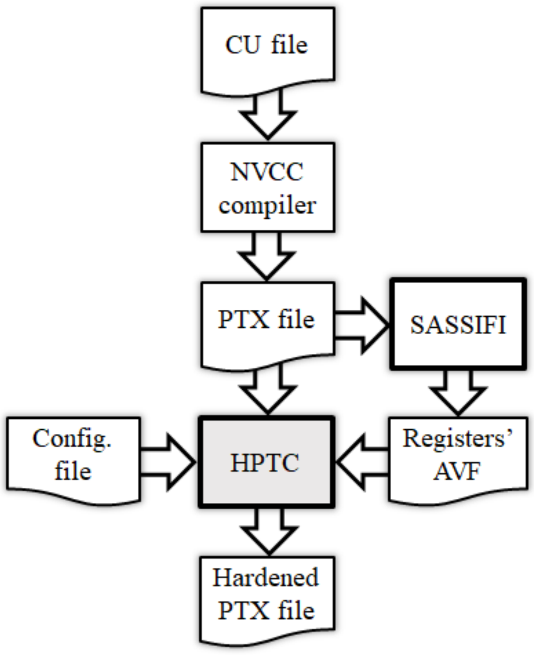
When considering NVIDIA's compilation flow, it is important to mention that it involves four files: CU, PTX, SASS, and BIN. CU is the source code containing both CPU and GPU code, PTX is the NVIDIA virtual machine pseudo-assembly, SASS is the low-level assembly language, and BIN is the binary microcode, which executes natively on the GPU. NVIDIA allows designers to generate the BIN from CU and PTX but does not allow it from SASS because the assembler is proprietary and not available. Therefore, to harden the closest-to-BIN file, we must harden the PTX, which still has to go through a transformation phase instead of the SASS. The main drawback is that we must guarantee that our hardening strategies remain after the transformation from the hardened PTX to the BIN by forcing the compilation to no optimization (-O0), which incurs impracticable runtime overheads (more than five times the original). Based on the software-implemented techniques in the Literature, it is possible to make the overhead acceptable (less than two times the original) by hardening the SASS.

The code transformations explored in this Chapter build upon the foundational work by (OH; SHIRVANI; MCCLUSKEY, 2002), as introduced in Section 4.1.1. We use the HPCT tool and follow the methodology flow presented in Figure 6.1 to automatically apply the software-implemented technique to the case-study applications written in CU. Initially, we input the CU source code to NVIDIA's compiler NVCC, which generates the PTX file. We then input the PTX file into SASSIFI to evaluate registers' AVF to SDC and DUE, as will be detailed in Section 6.3. Finally, we provide HPCT with the PTX file, the registers' AVF, and a configuration file. The result is a hardened PTX file.

6.2 Reliability Evaluation Results

This Section presents the empirical results of our reliability evaluation on NVIDIA Kepler GPUs. We focus on two critical aspects.

Figure 6.1: Software-implemented fault tolerance technique's flow



Source: The Author.

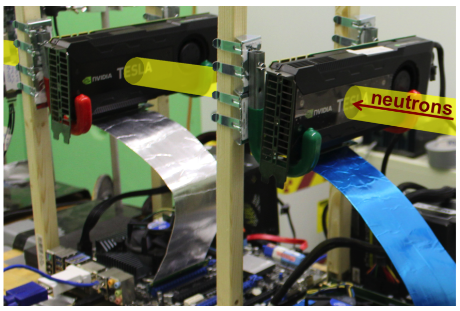
6.2.1 Impact of Neutron Radiation on FIT Rates

Figure 6.2(b) shows the results of the neutron beam experiments as normalized SDC and DUE Failure In Time (FIT) for Hotspot, NW, and Quicksort. Reported data have been normalized by the smallest value of all FITs (i.e., Hotspot DUE rate) to prevent the leakage of business-sensitive data. As shown in Figure 6.2(b), while the SDC rate strongly depends on the executed code (about one order of magnitude of difference between Quicksort and NW), the DUE rate is almost constant. This is explained by the fact that the DUE rate has a component that is dependent on the underlying hardware (scheduler, interfaces, etc.) and not only on the code. Quicksort and NW, respectively, have 1.5 and 2 times higher DUE rates than Hotspot, as the former are control-flow-based algorithms. Some data errors can lead to application crashes (as detailed in the following subsection). Even though results are plotted in arbitrary units, these data show that using fault tolerance techniques is mandatory for using such devices in safety-critical applications.

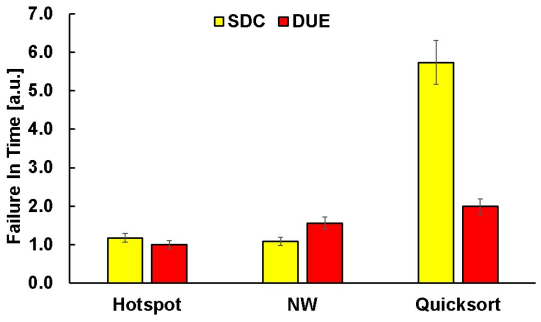
6.2.2 Architectural Vulnerability Assessment of GPU Registers

Figure 6.3 shows the AVF of each GPU register used to compute the selected benchmarks. Figure 6.3(a) shows the SDC AVF and 6.3(b) the DUE AVF. Each K40 thread has 255 available registers on the register file, but most applications use only a few of them. Hotspot uses more registers than the other tested benchmarks (52 registers per thread for Hotspot, 20 for NW, and 46 for Quicksort). Comparing Figure 6.3 with Fig-

Figure 6.2: Beam test setup at LANSCE and FIT for tested benchmarks.



(a) Part of our beam test setup at LANSCE.



(b) FIT for all tested benchmarks normalized on a Poisson distribution with a 95% confidence interval.

Source: The Author.

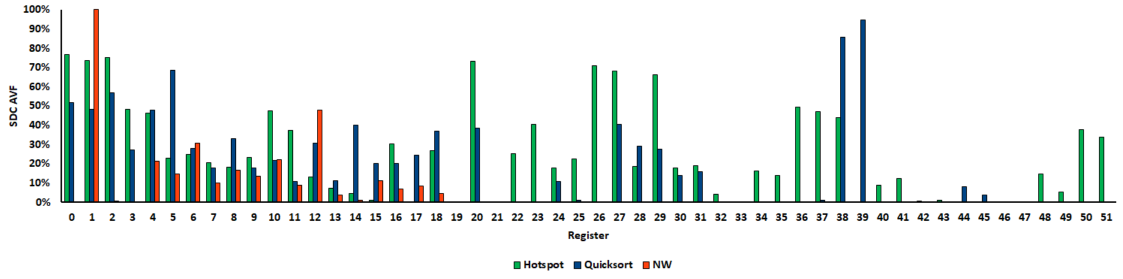
ure 6.2(b), it is clear that using more registers does not necessarily imply a higher FIT rate. This result is to be expected, as AVF only provides information about fault propagation without considering the probability of fault generation or faults in inaccessible resources. It is possible to notice that SDCs are less probable than DUEs for all codes. This interpretation is again in apparent contrast with Figure 6.2(b). On beam experiments, the GPU hardware is fully exposed (including the big data Cache, arithmetic and logic units, etc.), which increases the probability of data corruption. The higher AVF for DUE is also explained, considering that registers can store data, memory addresses, and index variables.

Faults in data mostly result in SDC, while faults in indexes or memory addresses probably generate a DUE. However, while faults injected on data registers can be masked by later operations (i.e., the corrupted result is multiplied by 0 or is filtered), faults injected on addresses and indexes have a very high probability of generating wrong memory accesses or of changing the algorithm control-flow, resulting in a DUE. It is important to notice that faults in data registers could also lead to a DUE (e.g., corrupted data is used to update a control-flow variable), and faults in indexes could lead to an SDC as well (e.g., a loop is shortened, resulting in incomplete data), but these situations are less probable.

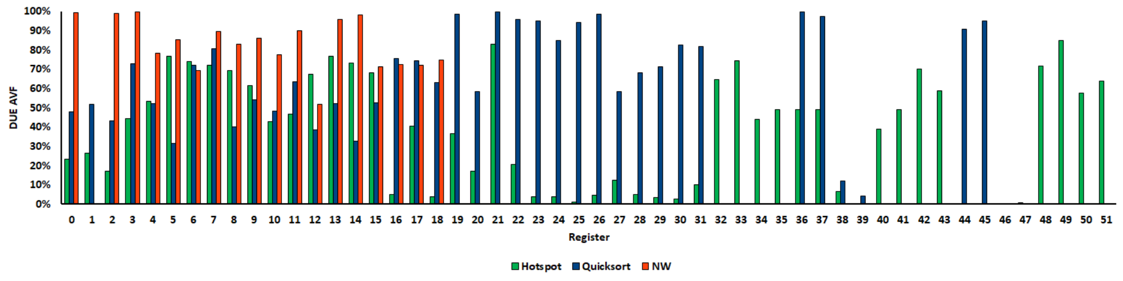
Fig. 6.3 also shows that the registers have very different AVFs to a given effect. For example, when considering DUE effects for Quicksort, registers 36 and 37 show an AVF lower than 0.1, while registers 37 and 38 show AVFs close to 1. These data confirm that there is a huge variation in AVF in the register file for the same type of effect and also that selective fault tolerance techniques should prioritize specific registers in order to increase efficiency in terms of fault tolerance per area, execution time, and power consumption.

Another interesting aspect of Figure 6.3 is that a register AVF can also vary according to each effect. An example can be seen in register 1 for NW, where SDC AFV is close to 1, while DUE AVF is close to 0. In other words, it means that some registers are more or less sensitive to causing SDC or DUE effects in the GPU. Such results are very important since fault tolerance techniques may be either targeted at mitigating SDC effects or DUE effects and because some applications are more concerned with one effect

Figure 6.3: Individual register AVF for all applications. (a) shows register AVF to SDC effects. (b) shows register AVF to DUE effects



(a) Register AVF to SDC



(b) Register AVF to DUE

than the other. Examples are safety-critical applications, which must avoid SDCs in order to guarantee a correct result, and HPC applications, which must guarantee time-frames and, therefore, avoid DUE effects that can slow down performance.

By using the data collected during these experiments and in the fault injection campaign, we can propose efficient hardening strategies based on hardware and software implementations that prioritize and protect only the most vulnerable registers to a given effect. The following Sections describe in detail how these strategies can be applied and their effectiveness in terms of fault coverage, area, execution time, and power consumption overhead.

6.3 Hardware-Implemented Selective Hardening

Hardware-implemented fault tolerance techniques for GPUs must be efficient and effective because their costs in the area, performance degradation, and power consumption can be extreme. As detailed in Section 6.2.2, not all the registers, once corrupted, impact the application output. Protecting the whole system would result in unnecessary overhead in run time or area usage. In other words, protecting all registers would provide a good but inefficient fault-tolerant solution.

We evaluate the benefit and overhead of a hardware selective hardening strategy for GPU RFs. To do so, we first identify the design's most sensitive and critical registers. Then, to evaluate the efficacy and efficiency of selective hardening techniques, we measure the overhead (either time or area) and the benefit (fault coverage) of protecting the identified critical registers. The more registers we protect, the higher the fault coverage and the higher the overhead.

Due to the impossibility of performing architectural modifications in modern GPUs and to simplify our analysis, we assume that the protection of a register (1) detects all faults in the register and (2) increases linearly the overhead. Overhead can be area overhead (registers are duplicated) or time overhead (error correction code is applied). While we are aware that (1) and (2) are simplifications, they are only used to evaluate the benefit and overhead of the proposed technique.

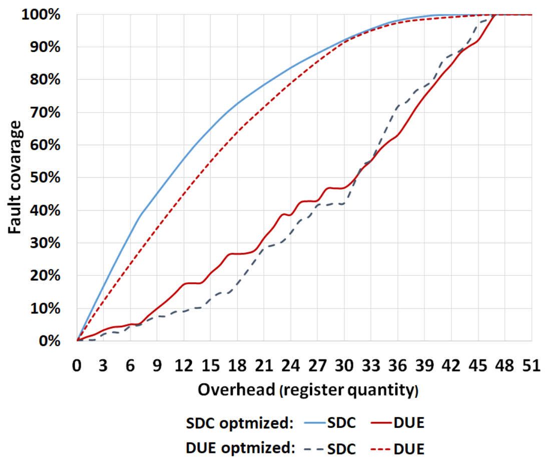
According to data presented in Section 6.5.2, GPU registers have different AVFs not only among each other but also for either SDC or DUE effects. A selective hardening approach should then target either DUE or SDC effects, as the protection of a register with high SDC AVF, for example, could hardly reduce DUEs. To prioritize and select the registers to protect, we ranked registers according to their SDC and DUE AVF, from most to least vulnerable.

Figures 6.4 show for Hotspot (Figure 6.4(a)), NW (Figure 6.4(b)), and Quicksort (Figure 6.4(c)) applications, the fault coverage as a function of overhead, which is linearly dependent on the number of hardened registers. We plot the achieved fault coverage as a function of the number of protected registers. We propose two selective hardening, one optimized for SDC effects and one for DUE effects. In Figure 6.4, *dotted lines* show the efficiency of DUE-optimized selective hardening in reducing DUEs (red dotted line) and SDCs (blue dotted line), while *continuous lines* show the efficiency of SDC-optimized selective hardening (red for DUEs and blue for SDCs).

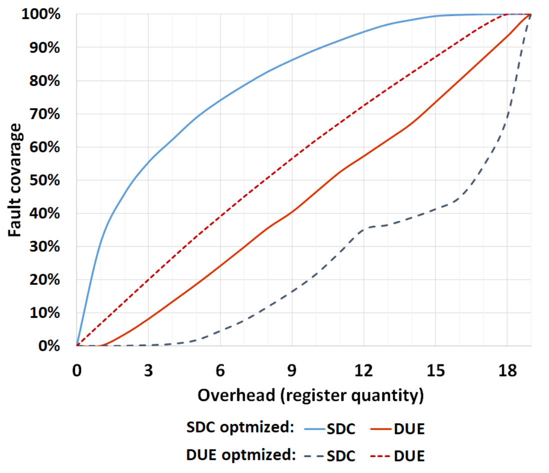
The Hotspot application, plotted in Figure 6.4(a), shows that both DUE and SDC fault coverages, for both SDC-optimized selective hardenings (*continuous blue line*) and DUE-optimized (*dotted red line*), saturate close to 39 hardened registers. Until then, they both follow a super-linear function, where SDC-optimized takes little advantage over DUE-optimized. Out of the 52 registers, for the SDC-optimized approach, 30% of fault coverage can be achieved with the hardening of 6 registers (12%), while 60% with 13 (25%), and 90% with 28 (54%). For the DUE-optimized approach, 30% of fault coverage can be achieved with the hardening of 8 registers (16%), while 60% with 17 (33%), and 90% with 29 (56%). This discrepancy happens because SDC and DUE effects are not equally distributed among registers, where SDCs have a more concentrated distribution than DUEs.

The NW application, plotted in Figure 6.4(b), shows a different trend than Hotspot. The SDC-optimized (*continuous blue line*) selective hardening shows a steeper function for the first hardened registers, presenting a super-linear function. In contrast, the DUE-optimized (*dotted red line*) is almost linear up until 18 hardened registers, when it saturates. Out of 20 registers, for the SDC-optimized approach, 30% of fault coverage can be achieved with the hardening of 1 register (5%), while 60% with 4 (15%), and 90% with 10

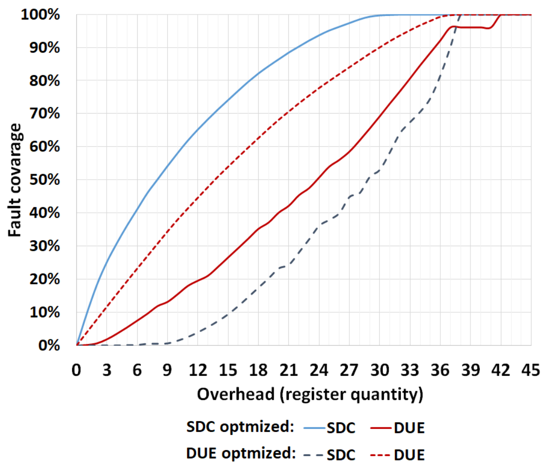
Figure 6.4: Hardware-implemented selective hardening efficiency. (a) Hotspot. (b) NW. (c) Quicksort



(a) Hotspot application



(b) NW application



(c) Quicksort application

(50%). For the DUE-optimized approach, 30% of fault coverage can be achieved with the hardening of 5 registers (25%), while 60% with 10 (50%), and 90% with 16 (80%). This discrepancy occurs mainly because only a few registers are responsible for most SDC effects, while, excluding one register that presented no effects, DUEs are almost equally distributed among used registers.

The Quicksort application, plotted in Figure 6.4(c), shows a similar trend to Hotspot in terms of DUE-optimized (*dotted red line*), but with a slightly steeper curve for SDC-optimized (*continuous blue line*) approach. Out of 46 registers, for the SDC-optimized approach, 30% of fault coverage can be achieved with the hardening of 4 registers (9%), while 60% with 11 (24%), and 90% with 22 (48%). For the DUE-optimized approach, 30% of fault coverage can be achieved with the hardening of 8 registers (17%), while 60% with 17 (37%), and 90% with 30 (65%). Such results are due to an SDC distribution among registers close to the NW but a DUE distribution close to the Hotspot.

It is interesting to notice that because of the low AVF of some registers either to SDC or DUE effects, all SDC-optimized and DUE-optimized show super-linear benefit-overhead trends. In other words, our proposed approach always results in better efficiency than fully protecting the register file (better fault coverage can always be achieved with less overhead). On the other hand, when targeting one specific effect, the other mostly shows worse efficiency. As one can see in Figures 6.4, fault coverage for DUE effects for SDC-optimized fault tolerance (*continuous red lines*) and SDC effects for DUE-optimized fault tolerance (*dotted blue lines*) show, along most of the x-axis, sub-linear functions.

6.4 Software-Implemented Selective Hardening

Software-implemented fault tolerance techniques are usually the only option when dealing with COTS or protected parts, which is the case with modern GPUs. Nonetheless, software hardening strategies must be efficient and effective since every instruction added to the original program code will eventually be processed by the GPU and incur execution time overhead, which can be translated into performance degradation and power consumption overhead. Fault tolerance by means of software implementation can be applied to a given code in different abstraction levels, from a high-level C++ code to a low-level assembly code. Either way, a selective software fault tolerance technique aims to harden the most vulnerable parts of the code.

Software hardening could be more easily applied than hardware selective hardening. In fact, hardware implementations may have to deal with issues such as irregular layouts when partially hardening a register file (especially if implemented with RAM modules), software and compiler modifications that may affect data distribution among physical registers, varying register AVF among applications, applications that may use different quantities of registers, among others. On the other hand, software-implemented techniques do not have to deal with any of these issues.

We follow the previous hardware-implemented approach to efficiently and effectively harden the GPU until we identify the design's most sensitive and critical registers. Then, using data acquired and plotted in Figure 6.4, we set constraints to 30%, 60%, and 90% fault coverage and protect all case-study applications.

An example of the transformation performed by VAR can be seen in Figure 6.5. As one can see, the left column shows the original program code, while the right one shows the protected code. Instructions 3 and 7 are replicated through instructions 4 and 8, respectively, over replicated spare registers %r1' and %r2'. The rest of the inserted instructions are used to check data consistency between registers and replicated registers data and to branch to the error subroutine.

Figure 6.5: Program code transformation

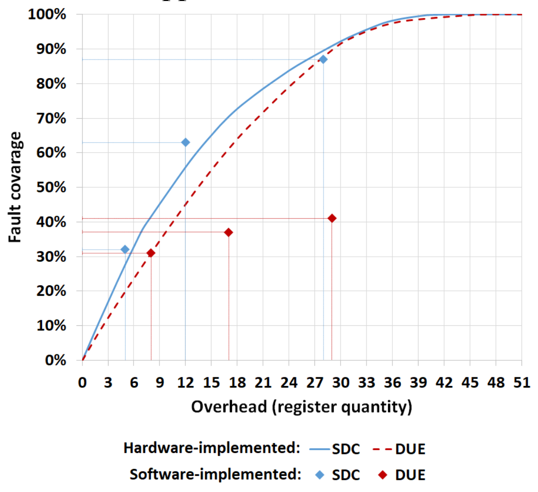
Original Code	Hardened Code
	`1: setp.ne %p1, %r2, %r2';`
	`2: @%p1 bra DETECT;`
`3: ld %r1, [%r2];`	`3: ld %r1, [%r2];`
	`4: ld %r1', [%r2'];`
	`5: setp.ne %p1, %r1, %r1';`
	`6: @%p1 bra DETECT;`
`7: add %r1, %r1, 2;`	`7: add %r1, %r1, 2;`
	`8: add %r1', %r1', 2;`
	`9: setp.ne %p1, %r1, %r1';`
	`10: @%p1 bra DETECT;`
	`11: setp.ne %p1, %r2, %r2';`
	`12: @%p1 bra DETECT;`
`13: st [%r2], %r1;`	`13: st [%r2], %r1;`

Source: The Author.

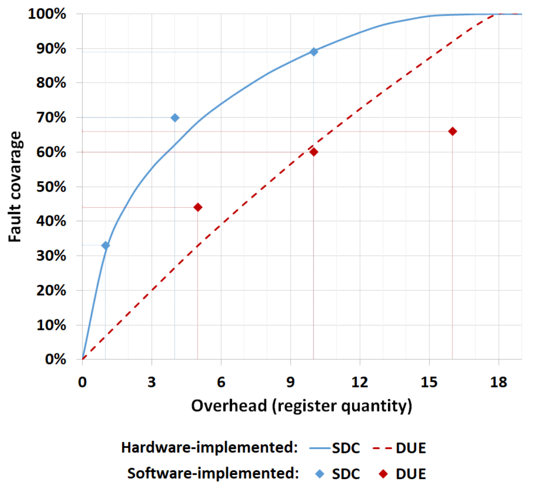
The fault coverage evaluation is done by a second fault injection campaign in the same fashion as the one performed in Section 6.1. Figures 6.6 show for Hotspot (Figure 6.6(a)), NW (Figure 6.6(b)), and Quicksort (Figure 6.6(c)) applications, the fault coverage as a function of protected registers. The lines partially replicate the data from Figures 6.4. At the same time, the dots represent the achieved fault tolerance for SDC (blue dots) and DUE (red dots) fault coverage. It is important to note that hardware- and software-implemented techniques are generic. Therefore, the X-axis should be adjusted regarding power consumption to get a fair comparison. Nonetheless, the plotted data can provide interesting insights into software-implemented efficiency regarding fault coverage per replicated register.

The Hotspot application, plotted in Figure 6.6(a), shows interesting results for both SDC and DUE software-implemented fault tolerance. When targeting 30%, 60%, and 90% fault coverage for SDC effects (continuous blue line), it achieved 32%, 63%, and 87% (blue dots), respectively. On the other hand, when targeting 30%, 60%, and 90% fault tolerance for DUE effects (dotted red line), it achieved 31%, 37%, and 41% (red dots). Results show that applied software-implemented techniques can efficiently detect

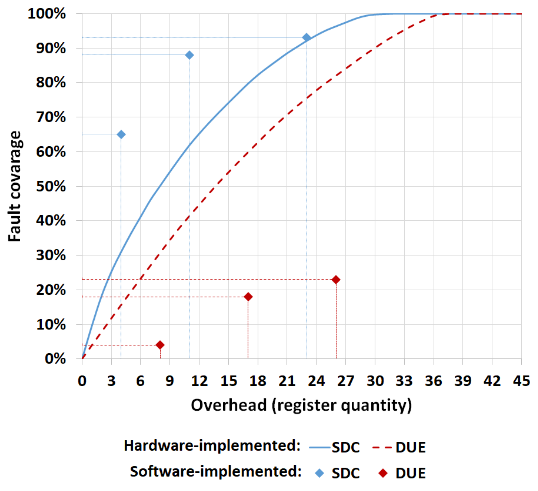
Figure 6.6: Software-implemented selective hardening efficiency. (a) Hotspot application. (b) NW application. (c) Quicksort application.



(a) Hotspot application



(b) NW application



(c) Quicksort application

SDC effects at the same rates as hardware-implemented techniques but cannot follow the same trend when considering DUE effects, where it saturates around 40% of fault coverage.

The NW application, plotted in Figure 6.6(b), shows results close to Hotspot. When targeting SDC effects, fault coverage was able to achieve results close to the hardware-implementation ones: 33% for 30%, 70% for 60%, and 89% for 90%. When targeting DUE effects, the NW hardening showed slightly better results than Hotspot but still lower than the hardware-implementation ones: 33%, 44%, and 66% for 30%, 60%, and 90%, respectively.

The Quicksort application, plotted in Figure 6.6(c), shows different results. For SDC fault coverage, it was able to detect 65% for 30%, 88% for 60%, and 93% for 90%. As one can notice, there is a much steeper curve than hardware-implemented approaches, even though both reach 90% at the same register overhead. DUE fault coverage, on the other hand, is much worse than previous applications. For 30%, 60%, and 90% hardware-implemented fault coverage, it accomplishes 4%, 18%, and 23%, respectively, showing worse results than hardware-implemented ones. The main reason is that Quicksort is the most control-flow-oriented application, having many function calls and different thread comparison instructions. Therefore, inserted instructions are more sensitive to DUE effects than previous applications, easily decreasing SDC effects but hardly decreasing DUE effects.

It is interesting to notice that our proposed software-implemented fault tolerance approach, even though implemented in the PTX file instead of the SASS, was able to achieve SDC fault coverage equal to or better than hardware-implemented ones for the same number of protected registers, showing that, with optimized application of software redundancy, it could also offer better efficiency in terms of fault tolerance per power consumption (power consumption by area for hardware-implemented and by run time for software-implemented approaches). On the other hand, it presented worse results than hardware implementation when targeting DUE effects. Results also show that, to fully harden a GPU, a combination of hardware-implemented and software-implemented fault tolerance techniques could be the best option for fault tolerance efficiency.

6.5 Improving Selective Fault Tolerance by Relaxing Application Accuracy

Some applications, safety-critical or not, have the attribute of tolerating inaccurate results as long as they are in a predefined known range. In this sense, the approximate computing paradigm exploits the gap between the level of accuracy required by the application and the level of accuracy provided by the computing system to improve energy and performance efficiency. By combining selective fault tolerance, which can target only the most critical parts of the system, and approximate computing, which trades off computation accuracy with expended effort, we are able to relax individual register criticality and target only the most critical in an approximate computing perspective, thus reducing the penalties imposed by software-based fault tolerance techniques.

Therefore, we propose to relax application accuracy to improve selective fault tolerance techniques in GPU register files (GONCALVES et al., 2020). We first offer a methodology to decrease application accuracy by introducing an error margin in the results (Section 6.5.1). After that, an evaluation of Kepler GPU registers' criticality to SDC-induced faults (Sections 6.5.2 and 6.5.3). Then, we use the obtained data to improve selective fault tolerance techniques regarding performance and resource usage (Section 6.5.4).

The experiments follow the methodology presented in Section 6.1, with the addition of a new Case-study application, LavaMD. This application simulates interactions among 192 particles in large 3D spaces. It's important to note that we excluded DUE-induced errors since our focus is on safety-critical applications. The analysis centers on SDC-induced errors.

6.5.1 Proposed Methodology for Relaxing Application Accuracy

Our proposed methodology to relax application accuracy uses approximate computing principles to exploit the gap between the level of accuracy required by the application and the level of accuracy provided by the computing system. It inserts a margin of error in which application results will be considered correct and, by doing so, can modify and indirectly reduce individual register criticalities.

Since accuracy margins vary among different applications, our proposed methodology aims to perform two separate analyses. In the first one, we relax accuracy to cover a wide range of error margins. By doing so, we intend to evaluate, in a broad spectrum, how SDC-induced faults affect the outputs of the chosen case-study applications. Then, we use the obtained data for the second analysis, limiting the relaxation accuracy range to a narrower window and evaluating each register's criticality individually. Combining these two analyses, we intend to find out how small relaxations on application accuracy impact individual register criticality. As applications vary in behavior and input data, we relax result accuracy in two ways. For the Hotspot, NW, and LavaMD, we relax accuracy by introducing a percentage margin in which all results are considered correct once a single incorrect result out of the accepted accuracy margin could lead the application to failure. For the Quicksort, we relax accuracy by introducing a percentage margin of total errors in the output vector. As a supporting algorithm for larger applications, Quicksort depends on its input vector values, especially because it mainly switches memory positions. For example, if all vector values were the same, we would only see a reduced number of SDCs, as the application would mask most of the faults. Still, if the vector were composed of sparse values, a little accuracy relaxation would not be able to reduce register criticality.

In the following subsections, we present the analyses proposed by our methodology: subsection 6.5.2 discusses the broad analysis of register file reliability to SDCs, and subsection 6.5.3 discusses the narrower analysis of individual register criticalities to SDCs.

6.5.2 Analysis on Register File Reliability

In this first analysis, our methodology evaluates the impact of SDCs on the output of the chosen case-study applications. We analyze the reliability increase to SDCs for all applications, with a range of acceptance error margins varying from $10^{-9}\%$ to 100% on a logarithmic scale. As one can see in Figure 6.7, the chosen range is able to cover a broad spectrum in reliability increase. It is important to notice that, depending on the target application, one might need to widen the accuracy relaxation range further to observe the reliability increase better.

Figure 6.7 plots register file reliability increase for relaxed application accuracy for all case-study applications: Hotspot (yellow circles), NW (brown crosses), LavaMD (green squares), and Quicksort (blue triangles). The graph considers the sum of all individual registers' AVF for each application shown in previous Figure 6.8. As the accuracy relaxation increases, more errors are considered correct, and the reliability of SDCs also increases.

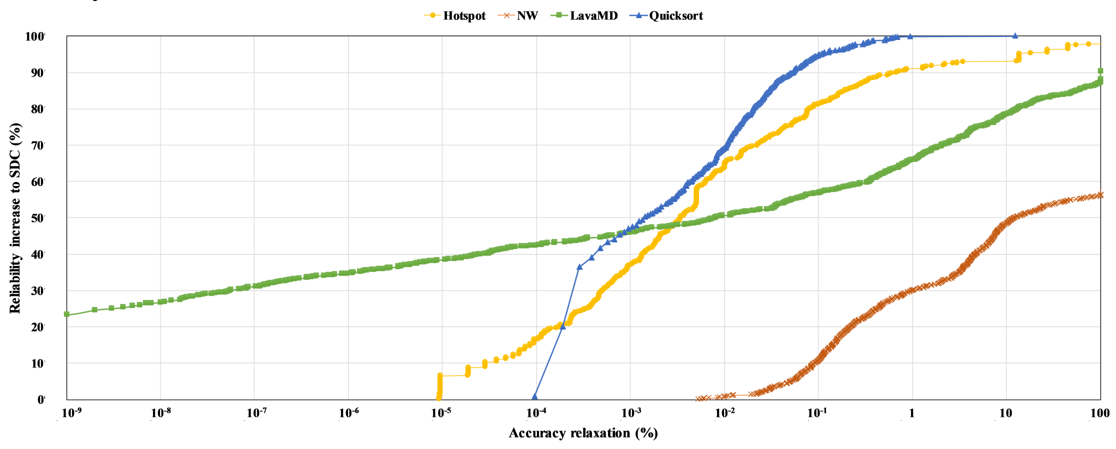
The Hotspot, plotted in yellow circles, shows a high increase rate in reliability for accuracy relaxations between $10^{-5}\%$ and 1%, indicating a reliability increase of up to 90.8% for an accuracy relaxation of 1%. This improvement occurs because most faults cause a small variation in the output when SDC-induced faults corrupt application registers.

Results for the NW application, plotted in brown crosses, show almost no increase in reliability at less than $10^{-2}\%$ relaxation, reaching only 56.4% reliability increase at 100% accuracy relaxation. It shows the least reliability increase per accuracy relaxation among all case-study applications. On the other hand, even though it presented the worst results, one could still increase reliability by 29.9% when increasing accepted result accuracy to 1% and, if one could push acceptance to 10%, a 48.4% reliability increase could be achieved.

Results for the LavaMD application, drawn in green squares, indicate a milder slope when compared to other applications. The reliability increase grows almost linearly, becoming steeper at $10^{-2}\%$. It provides gains in reliability starting at the lower limit of $10^{-9}\%$ until the higher limit of 100%. For instance, one could increase reliability from 23.3% to 65.9% for a tolerance range from $10^{-9}\%$ to 1%, and up to 78.5% when accepting a 10% error margin.

Lastly, the triangular blue line shows results for the Quicksort. This algorithm shows similar behavior to the Hotspot, presenting a 99.8% reliability increase rate at 1%

Figure 6.7: Register file reliability increase to SDC-induced errors for relaxed application accuracy.



accuracy relaxation. Although only 0.2% errors are bigger than 1%, the biggest error observed in Quicksort is 12.4%, indicating the accuracy relaxation required to make the Quicksort fully tolerant to SDC-induced faults.

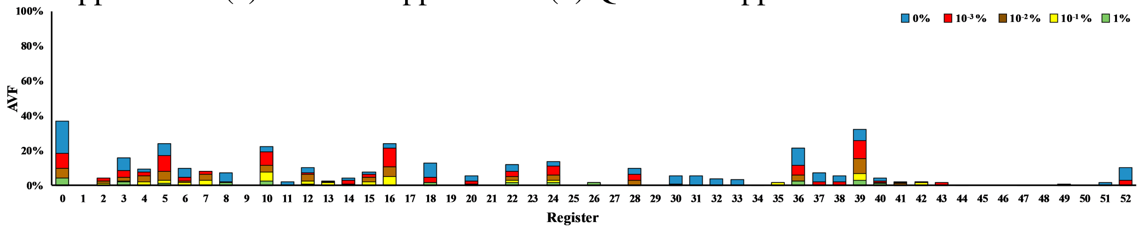
6.5.3 Analysis on Register Criticality

To better visualize how accuracy relaxation affects register criticality and based on results plotted and discussed in previous subsection 6.5.2, this second analysis considers reduced accuracy relaxation margins between 10^{-3}% and 1%. Such margins are conservative when compared to the state-of-the-art works and feasible for many real applications. The previous analysis showed that the register file reliability increased on all case-study applications when relaxing result accuracy in these restricted margins. Therefore, a fine-grained analysis of individual register criticality could provide a better understanding of overall register criticality. Figure 6.8 presents individual register's AVF to SDC-induced faults in our four case-study applications for all application-used registers.

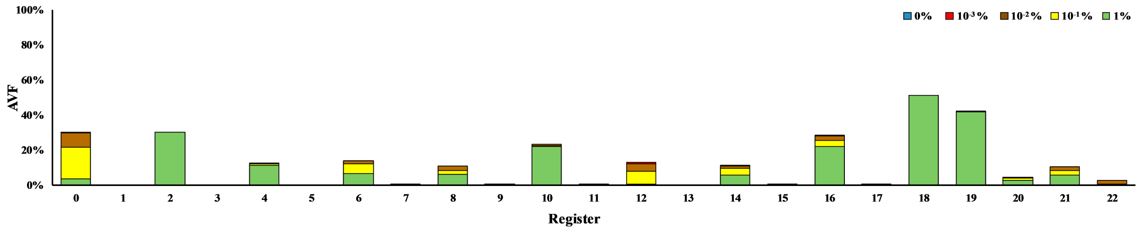
The bars plotted in Figure 6.8 are composed of different application accuracies. A blue bar represents a 0% accepted accuracy (used by state-of-the-art selective fault tolerance techniques), and it changes color as application accuracy is relaxed up to a 1% accepted accuracy in a logarithmic scale. A single-color bar represents a register that can only be relaxed at a single pace, while a five-color bar represents a register that can be relaxed at multiple paces. A nonexistent bar represents a register that is not susceptible to SDC-induced faults. Interestingly, while relaxing application accuracy, register criticality ordering also changes, influencing selective fault tolerance parameters.

Figure 6.8(a) presents the results for the Hotspot. They show that when relaxing acceptance accuracy to the minimum 10^{-3}%, most registers already drop criticality, and registers 11, 33, 34, 44, and 47 become fully tolerant to SDC-induced faults. When

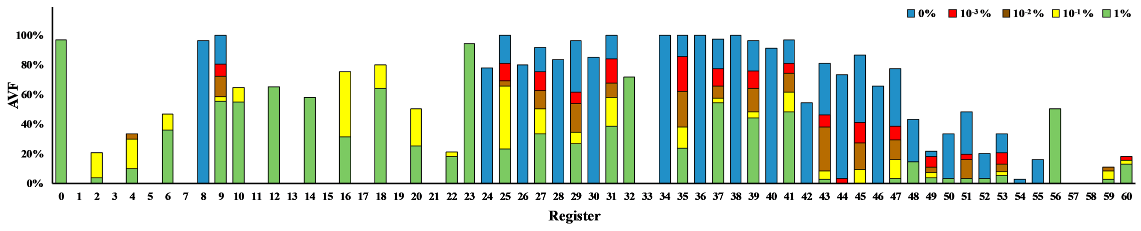
Figure 6.8: Register AVF to SDC with relaxed criticality. (a) Hotspot application. (b) NW application. (c) LavaMD application. (d) Quicksort application.



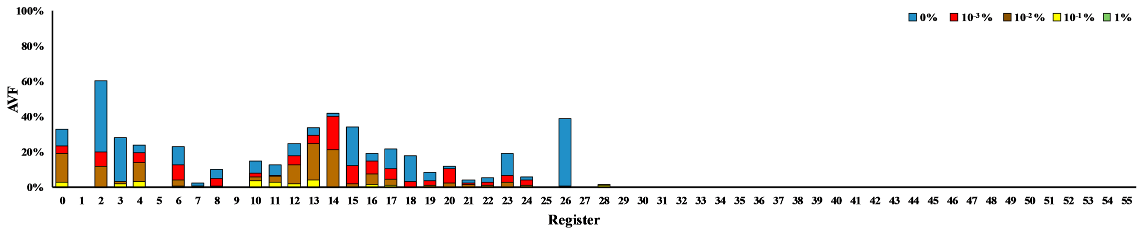
(a) Hotspot application



(b) NW application



(c) LavaMD application



(d) Quicksort application

further relaxing application accuracy, register AVF drops further to the point where, at 1% accepted accuracy, only 25 registers remain sensitive to SDC-induced faults, all below 5% AVF, dropping from an average register AVF of 6.7% at 0% accuracy to 0.6% at 1% accuracy.

Results for the NW application, plotted in Figure 6.8(b), show a different tendency. Relaxing application accuracy between $10^{-3}\%$ and $10^{-2}\%$ accepted accuracy has almost no effect on register AVF. Effects will only be seen at $10^{-1}\%$ and 1% steps, where a few registers decrease AVF (0, 6, 8, 12, 14, 16, 21, and 22). From these registers, 0, 12, and 22 were able to have their AVFs reduced to almost 0. Such results happen because the number of errors caused by a single fault was much higher when compared to the Hotspot (500+ errors against up to 10). Our methodology considers a single value out of the margin as an error. Because of that, AVF drops have been mostly noticed when relaxing accuracy over 500%.

Figure 6.8(c) presents the results for the LavaMD application. They show a combination of Hotspot and NW. 12 registers were not susceptible to SDC-induced faults. From the remaining 48, 13 became fault-tolerant to SDC-induced faults at $10^{-3}\%$ and a total of 15 at $10^{-2}\%$. On the other hand, relaxed criticality did not affect 6 registers up to 1%. The LavaMD is a perfect application to show how ordering by register criticality changes when relaxing accepted accuracy. For example, registers 34, 36, and 38 drop from the most critical to the least critical with a $10^{-3}\%$ relaxation.

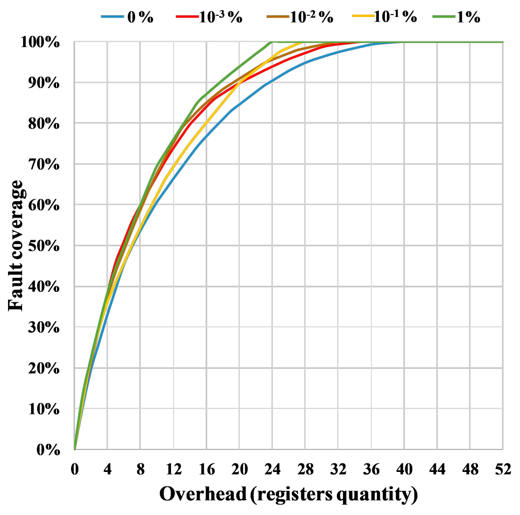
Results for the Quicksort application, plotted in Figure 6.8(d), present yet another tendency. From its 55 used registers, only 24 (44%) are sensitive to SDC-induced faults. The hardening of only 44% of resources yields 100% fault tolerance to SDC-induced faults. When applying our approach to relax application accuracy, registers present a behavior similar to the Hotspot, where individual register AVFs drop at different paces, modifying register criticality ordering and thus further affecting selective fault tolerance techniques.

6.5.4 Selective Hardening Combined with Approximate Computing

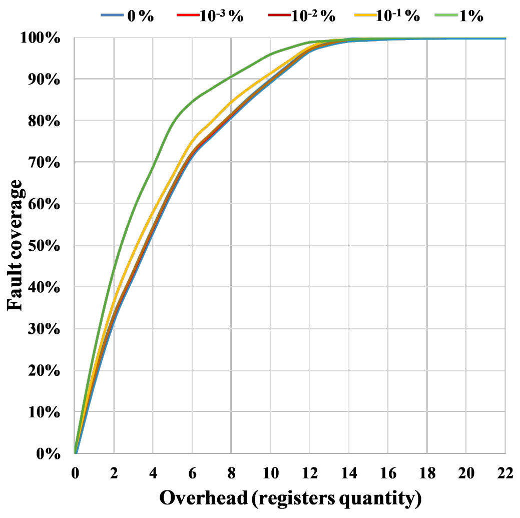
Figure 6.9 presents data from selective fault tolerance techniques. For all case-study applications, it shows fault coverage as a function of hardened registers (either by means of hardware or software). Each line represents a different acceptance accuracy, including 0% (regular selective hardening) and a logarithmic scale from $10^{-3}\%$ to 1%. The following results are compared to full hardening (replication of all registers) and regular selective hardening (replication of the sensitive registers considering a 0% error margin) techniques. Selective hardening techniques usually reduce full hardening overheads in performance and resource usage, as they only protect a subset of all registers. Our approach further reduces overheads by reducing the subset of registers to be protected.

The Hotspot, plotted in Figure 6.9(a), has a total of 53 registers, of which 41 are

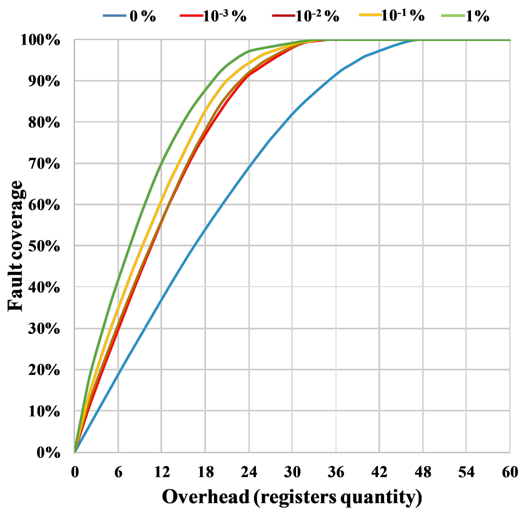
Figure 6.9: Selective hardening efficiency for relaxed criticality. (a) Hotspot application. (b) NW application. (c) LavaMD application. (d) Quicksort application.



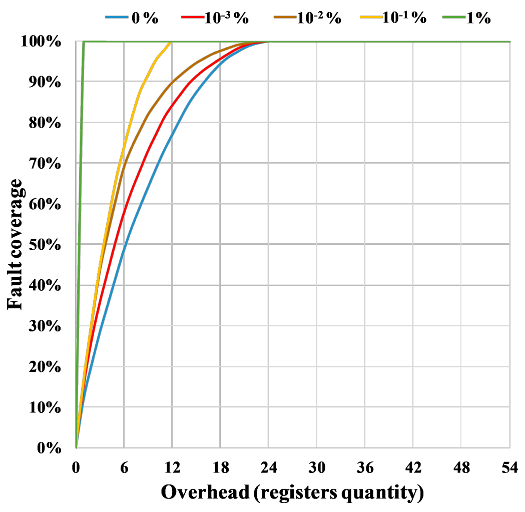
(a) Hotspot application



(b) NW application



(c) LavaMD application



(d) Quicksort application

sensitive to SDC-induced faults. At 100% fault coverage, selective techniques would be able to reduce full hardening overhead by 22.6%. Our approach is able to further reduce overhead at a pace of 4 registers per relaxation step down to 25 registers at 1% acceptance accuracy, resulting in a 39% reduction in overhead when compared to selective fault tolerance techniques (52.8% to full hardening).

The NW application, presented in Figure 6.9(b), has a total of 23 registers, and 21 are sensitive to induced faults. When relaxing criticality, hardening the same 21 registers is required to achieve 100%

Figure 6.9(c) shows data for the LavaMD application. From its 61 registers, 49 are sensitive to SDC-induced faults, meaning that selective fault tolerance techniques would be able to reduce overheads by 19.6% over full hardening while keeping 100% fault coverage. Our approach at 10^{-3}% acceptance accuracy can reduce this overhead to 36 registers and down to 34 registers at 1% accuracy. It represents a 44.2% decrease from full hardening and a 30.6% decrease from selective hardening.

The Quicksort, shown in Figure 6.9(d), has 56 registers, of which 25 are sensitive to SDC-induced faults. While maintaining 100% fault coverage, our approach is able to reduce sensitive registers to 13 at 10^{-1}% and to 1 at 1%, or 98.2% and 96% reduction from full and selective hardening, respectively.

6.6 Case-study Summary

This study case provided guidelines for improving GPU register file reliability by implementing and applying selective fault tolerance techniques implemented by means of hardware and software. In the first moment, a radiation experiment performed using a neutron beam was conducted to evaluate the FIT of a set of case-study applications running on NVIDIA K40 GPUs. Then, a fault injection campaign was performed with SASSIFI, an architectural-level fault injector, to evaluate individual register AVF to SDC and DUE effects. While radiation experiments showed that GPU systems must be hardened to perform safety-critical applications, fault injection results showed that AVF varies widely among different registers, applications, and effects (SDC and DUE).

Based on collected data from the fault injection campaign, we proposed a fault tolerance strategy to rank registers according to either SDC or DUE effects and selectively harden them with known hardware-implemented fault tolerance techniques. Results showed that, for all applications, super-linear functions were found when targeting a specific effect. In other words, the results presented always showed better efficiency regarding fault coverage per overhead than applying random or full register file hardening. On the other hand, fault coverage for the not-targeted effect presented worse efficiency.

We then applied the same strategy to software-implemented fault tolerance techniques by using the same register AVF ranking and applying software-implemented techniques to all case-study applications. Results from a second fault injection campaign, compared to previous hardware-implemented ones, in terms of the number of protected registers, showed equal or better efficiency for SDC fault coverage but worse efficiency for DUE fault coverage. Such results should be adjusted according to area overhead (hardware-implemented) and performance degradation (software-implemented). Still, they provide interesting insights on selective fault tolerance for commercial state-of-the-art GPUs as they are.

In addition, we proposed to decrease acceptance accuracy to improve fault tolerance techniques in GPU register files by either increasing SDC fault coverage or reducing overheads. Then, we proposed a methodology to relax acceptance accuracy, reduce register criticality, and evaluate individual register AVF to SDCs. Based on the collected data, we reordered registers according to relaxed criticalities and combined them with selective fault tolerance techniques.

Results showed that, by only relaxing application accuracy, it was possible to improve the GPU register file's reliability against SDC-induced faults in an average of 71.6% for 1% of application accuracy relaxation. Results also showed that our approach is able to reduce overheads when compared to selective hardening by an average of 41.4% while maintaining 100% fault coverage. When lowering fault coverage constraints below 100%, our approach presented even higher gains, up to the point where, at 10% overhead (10% of the system registers are hardened), we were able to increase fault coverage by an average of 80.1%, when compared to selective fault tolerance technique. It is also important to mention that our approach can be applied to hardware-implemented, software-implemented, and hybrid techniques.

Building upon the previously discussed selective hardening techniques in Chapter 4, this study case introduced an innovative approach that integrated the selective hardening strategy with approximate computing techniques to optimize the protection of the most vulnerable application registers in commercial GPUs. This approach is particularly relevant when compared with existing techniques such as NVIDIA's SInRG (Mahmoud et al., 2018), which, while effective in offering substantial SDC coverage through instruction protection and general ECC implementation for register reliability, does not prioritize selectiveness in register protection, thus potentially leading to significant performance overheads and increasing DUE effects, as demonstrated by (OLIVEIRA et al., 2014). Our methodology addressed this challenge by selectively targeting critical registers and enhanced efficiency through a combination with approximated computing. Moreover, the results demonstrated the potential of our approach to significantly reduce overheads compared to selective hardening alone while maintaining or even improving fault coverage, thus presenting a compelling advancement in the GPU register file reliability field.

The subsequent Chapter immerses in the domain of the FlexGrip architecture, where we delve into the implementation of low-level software-based hardening techniques and comprehensive ISA extensions for hybrid fault tolerance implementation.

7 LOW-LEVEL SOFTWARE-BASED HARDENING IN FLEXGRIP

This chapter begins with the implementation of established low-level software-based fault tolerance techniques for GPUs, showcasing how these can be applied through assembly code transformations. Following this, we propose three specific software-only optimizations: *Traceback*, *Move*, and *Delayed Notification*, and then embark on a comprehensive Design Space Exploration. This exploration assesses the impact of these newly proposed optimizations on GPU reliability across multiple configurations. The optimizations are tailored to enhance performance and reliability trade-offs in GPU architectures and are selectively applied to critical areas such as memory-access and predicate-setting instructions (GONCALVES et al., 2022).

Subsequently, the discussion extends to the proposal and implementation of ISA extensions, aiming to provide a more comprehensive solution for GPU reliability. These extensions are developed to augment the fault detection and correction capabilities of GPUs beyond the software-based methods (GONCALVES et al., 2020).

In the latter sections, the chapter transitions to address the detection and correction of faults within the GPU pipeline. Here, we present a systematic approach to enhancing pipeline reliability through a hybrid XOR technique and software-managed parity, assessing their effectiveness in mitigating radiation-induced faults (BRAGA; GONÇALVES; AZAMBUJA, 2023; BRAGA; GONCALVES; AZAMBUJA, 2023).

7.1 Methodology for Reliability Evaluation and Hardening

This section outlines the methods for assessing FlexGrip reliability through fault injection and details the assembly code hardening techniques used to mitigate error propagation within GPUs.

7.1.1 Reliability Assessment through RTL Simulation

Fault injection campaigns were performed through the Fault Injection Simulator presented in Section 3.4.2 to measure the impact of software-based hardening techniques on reliability. We measured reliability by evaluating the probability of low-level corruption, such as a bit-flip in the register files or the pipeline registers, propagating to the output vector and causing an error quantified by the AVF. This probability was determined by dividing the number of errors by the number of injected faults. We did not inject faults in the memories, assuming they were protected by design (e.g., ECC). Still, we intend to evaluate them in future studies. To evaluate the system's reliability, we classified the injected faults based on their potential impact on the system AVFs: Masked, DUE, SDC, and Detected.

7.1.2 Assembly Code Hardening through HPCT

The software-based hardening techniques are implemented through program code transformations at the assembly level. Thus, they insert/remove assembly instructions into/from the program code, access registers and memory addresses, and use the datapath and the controlpath through assembly instructions. The main benefit of applying software-based techniques at the assembly level is that they are compiler-independent. Therefore, all compiler optimizations can be performed without removing the added redundancies, and we have better control over the code transformation. Also, one can directly target specific registers instead of variables, directly protecting the register files. The main drawback is that the pipeline is not directly accessible. Therefore, its hardening becomes a byproduct of the register file hardening.

To protect NVIDIA GPU codes at a low level, it is necessary to modify the SASS source, as outlined in Section 5.2. While alterations at the PTX level are possible, they may lead to undesirable changes during the compilation process, affecting the optimization of the final assembly.

We employed HPCT, which we upgraded to support the FlexGrip ISA and the proposed techniques, to apply the software-implemented techniques to the case-study applications automatically. This process involved inputting the SASS code into HPCT, which automatically carried out the code transformations, resulting in a hardened SASS file. The program code transformations discussed in this chapter are based on the original work of Oh et al., introduced in Section 4.1.1.

7.2 Software-based Hardening Techniques

In the following, we discuss the program code transformations alongside the three proposed optimizations to improve performance in reliability, fault effects, and fault notification time at different costs.

7.2.1 Program Code Transformations

In this section, we further detail the implementation of the three core transformations, as initially proposed by Oh et al. (OH; SHIRVANI; MCCLUSKEY, 2002) and introduced in Section 4.1.1: (1) datapath duplication, (2) consistency checking, and (3) host notification.

Datapath duplication (transformation 1 - T1) is responsible for duplicating all datapath operations, a process that aligns with the concept of Redundancy discussed in the previous section. This transformation forces the hardware to execute twice the datapath operation in an intertwined fashion, thereby exploiting Instruction Level Parallelism (ILP) from the GPU architecture more effectively than running the code twice. The duplicated instructions operate over copy registers, completely separating the original and duplicated datapath operations. As we consider the memory hardened by other means (e.g., ECC), we do not duplicate store instructions, thus not duplicating memory addresses.

Consistency checking (transformation 2 - T2) corresponds to the Checking component discussed earlier. It is tasked with verifying the consistency between the original and duplicated datapath using comparison instructions followed by conditional branching to an error subroutine. This transformation introduces a data dependency that affects the ILP gains. In this section, we explore the insertion of consistency checking after two classes of instructions: memory access and predicate setting, which influence the program's data flow and control flow, respectively.

Host notification (transformation 3 - T3) aligns with the Notification component, notifying the host in the event of a fault detection. This transformation could be either a trap instruction or a memory write instruction to the global memory. These instructions are not executed during correct application execution and are activated a single time when a fault is detected. Their execution is conditional upon the outcome of the consistency checking, necessitating a predicate register check each time a host notification is added to the program code.

Figure 7.1 exemplifies the three transformations in column *Non-optimized Hardened Code*. Datapath duplication (T1) is depicted in green, replicating lines 1, 3, and 5 with lines 2, 4, and 6. As one can notice, the instructions are the same, but they operate over replicated registers (e.g., R3' instead of R3). Note that the store instruction in line 15 is not duplicated. Consistency checking (T2) is highlighted in blue through instructions 7, 16, and 18, which check consistency for memory access instructions, and instructions 10 and 12, which perform consistency checks for predicate setting instructions. Finally, host notification transformation (T3) is shown in red and inserted after each consistency checking instruction in lines 8, 11, 13, 18, and 19.

7.2.2 Proposed Optimizations

The implemented code transformations take advantage of ILP but still duplicate the whole datapath (except for store instructions) and insert consistency checks and host notifications. Therefore, we expect them to incur high execution time overheads, even considering ILP gains. To reduce performance degradation (i.e., increase the performance of software-based hardening techniques), we propose three optimizations targeting the program code transformations: Move optimization, Traceback optimization, and Delayed Notification optimization. These optimizations aim to improve performance by trading off reliability, detection of specific effects, and host notification delay.

Figure 7.1: Software-based hardening technique transformation examples.

	Unhardened Code	Non-optimized Hardened Code	Optimized Hardened Code			
			Move	Traceback MEM	Traceback PRED	Delayed Notification
1:	MOV R3, 4	MOV R3, 4	MOV R3, 4	MOV R3, 4	MOV R3, 4	MOV R3, 4
2:		MOV R3', 4;	MOV R3', 4;		MOV R3', 4;	MOV R3', 4;
3:	ADD R1, R1, 1;	ADD R1, R1, 1;	ADD R1, R1, 1;	ADD R1, R1, 1;	ADD R1, R1, 1;	ADD R1, R1, 1;
4:		ADD R1', R1', 1;	ADD R1', R1', 1;	ADD R1', R1', 1;		ADD R1', R1', 1;
5:	LOAD R2, [R1];	LOAD R2, [R1];	LOAD R2, [R1];	LOAD R2, [R1];	LOAD R2, [R1];	LOAD R2, [R1];
6:		LOAD R2', [R1'];	MOV R2', R2;	LOAD R2', [R1'];		LOAD R2', [R1'];
7:		SETP.NE PE, R1, R1';	SETP.NE PE, R1, R1';	SETP.NE PE, R1, R1';		@!PE SETP.NE PE, R1, R1';
8:		@PE ERROR;	@PE ERROR;	@PE ERROR;		
9:	SETP.NE P0, R3, R0;	SETP.NE P0, R3, R0;	SETP.NE P0, R3, R0;	SETP.NE P0, R3, R0;	SETP.NE P0, R3, R0;	SETP.NE P0, R3, R0;
10:		SETP.NE PE, R3, R3';	SETP.NE PE, R3, R3';		SETP.NE PE, R3, R3';	@!PE SETP.NE PE, R2, R2';
11:		@PE ERROR;	@PE ERROR;		@PE ERROR;	
12:		SETP.NE PE, R0, R0';	SETP.NE PE, R0, R0';		SETP.NE PE, R0, R0';	@!PE SETP.NE PE, R3, R3';
13:		@PE ERROR;	@PE ERROR;		@PE ERROR;	
14:	@P0 BRA 1;	@P0 BRA 1;	@P0 BRA 1;	@P0 BRA 1;	@P0 BRA 1;	@P0 BRA 1;
15:	STORE [R4], R1;	STORE [R4], R1;	STORE [R4], R1;	STORE [R4], R1;	STORE [R4], R1;	STORE [R4], R1;
16:		SETP.NE PE, R1, R1';	SETP.NE PE, R1, R1';	SETP.NE PE, R1, R1';		@!PE SETP.NE PE, R1, R1';
17:		@PE ERROR;	@PE ERROR;	@PE ERROR;		
18:		SETP.NE PE, R4, R4';	SETP.NE PE, R4, R4';	SETP.NE PE, R4, R4';		@!PE SETP.NE PE, R4, R4';
19:		@PE ERROR;	@PE ERROR;	@PE ERROR;		
20:						@PE ERROR;

Source: The Author.

7.2.2.1 Move optimization

Memory access instructions (i.e., load and store) are the instructions that require the most clock cycles to be executed in FlexGrip (e.g., a load instruction requires around four times more clock cycles than a move instruction). In this sense, we propose the *Move* optimization. This optimization replaces replicated load instructions with move instructions that copy the loaded data to the copy register. Thus, instead of adding a replicated load instruction, it adds a faster move instruction. Doing so directly affects the datapath duplication (T1) by reducing the performance overhead and adding a point of failure to the hardened code.

Figure 7.1 shows an example of this optimization in column *Move opt*. When compared to the non-optimized hardened version (*Hardened code*), the only difference is line 6, where *Hardened code* replicates the original load instruction in line 5 (LOAD R2, [R1]) with a second load instruction (LOAD R2', [R1']), and the Move optimization uses a move instruction instead (MOV R2', R2).

While replicating a load instruction with a move instruction is able to reduce execution time, it inserts a point of failure on the software-based hardening technique, thus trading off on reliability. Suppose a fault affects the register written by the load instruction before the move instruction can copy its value to the replicated registers. In that case, the corrupted value will propagate to the replicated register, both the value and its replica will be corrupted, and the consistency checking will not signal a fault. Even though the load instruction takes an increased amount of clock cycles to execute, only a fault that hits the instruction in its late write-to-register stage would actually upset the destination register. Therefore, this point of failure is smaller than the fetch-to-fetch time.

7.2.2.2 Traceback optimization

Datapath duplication and consistency checking are the leading causes of performance degradation in software-based hardening techniques, even when considering ILP. One alternative is to selectively apply selective hardening techniques by targeting only the most critical parts of the program code. However, related works still cannot pinpoint the most critical parts of a program code. Some related works target subroutines and functions at the program level, while others target variables, registers, and memory addresses. Instead, we propose the Traceback optimization to target instructions and their data in a more fine-grained approach.

It targets specific instructions and all data used during their execution. To do so, we choose a group of target instructions with a high probability of causing an error to the application if affected by a fault (i.e., memory access instructions for data-flow errors and predicate setting instructions for control-flow errors) and evaluate all previous instructions that lead to the execution of these target instructions (i.e., all instructions that computed the data read by a given target instruction).

To implement the Traceback optimization, we start by defining a group of instructions as target instructions (e.g., memory access or predicate setting instructions). Then, we analyze the static program code and draw its control flow. Next, for each target instruction *i*, we evaluate which instructions have written their registers and add them to the group of target instructions. We run this procedure recursively until there are no new instructions to be added to the target instruction group. Finally, we selectively apply the previously discussed program code transformations to this group of instructions.

The choice of which instructions to add to the target instruction group is a complex task that depends on in-depth code analysis, fault injection campaigns, and design space exploration. As this analysis is out of the scope of this work, we took a simplified approach and selected two instruction groups: memory access instructions (Traceback MEM) and predicate setting instructions (Traceback PRED). By doing so, we expect to target data-flow and control-flow errors, respectively. Even though this simplifies the problem, we expect it to provide good results for a proof-of-concept.

Figure 7.1 shows the Traceback optimization applied for memory access instructions on the *Traceback MEM* column and for predicate setting instructions on the *Traceback PRED* column. The original code has two memory access instructions in lines 5 and 15, which have their registers R1 written by instruction 3, and one predicate setting instruction in line 9, which has its register R3 written by instruction 1. Thus, the Traceback optimization selectively hardens lines 3, 5, and 15 for the memory access instructions and lines 1 and 9 for the predicate setting one.

The Traceback optimization selectively targets specific instructions to enhance the effectiveness of program code transformation, leaving some code unhardened. It lets designers focus on particular fault effects like data-flow or control-flow errors. This optimization is highly effective for applications with distinct logic for control flow and data flow, such as a constant loop computing a value. However, hardening may yield similar code to non-optimized versions for applications with intertwined control and data flow logic, such as computations determining loop iterations from inputs.

7.2.2.3 Delayed Notification optimization

The host notification program code transformation informs the user that a fault has been detected in a previous consistency check. When applying the non-optimized program hardening, the consistency check compares two values and overwrites a predicate register with a flag indicating if a fault was detected. As every consistency check overwrites the previous value of the flag, the host notification has to be done before the next consistency check. The proposed Delayed Notification optimization changes how consistency checks write predicate registers. It thus provides the designer with options to perform host notifications less frequently. To do so, it employs conditional instructions (usually implemented but not restricted to GPU ISAs, such as in CUDA) to replace the comparison instruction with a conditional comparison instruction. By doing so, the predicate register is only written once when changing state to "fault detected," thus never being reset. Therefore, multiple consistency checks can be paired with a single host notification, up to using a single host notification instruction for the complete program code.

Figure 7.1 shows the Delayed Notification optimization in the *Delayed Notification opt.* column. Compared to the non-optimized program hardening, it replaces all consistency checks (SETP.NE) in lines 7, 10, 12, 16, and 18 with conditional consistency checks (@!PE SETP.NE). By doing so, it removes all host notification instructions in lines 8, 11, 13, 17, and 18 and inserts a new host notification instruction in line 20.

The Delayed Notification optimization allows designers to decrease host notification frequency. Doing so improves performance at the cost of a larger fault notification period. Therefore, it does not decrease reliability in terms of fault detection. However, a latent fault in the system might increase the chance of a fault causing an error. Also, the longer the system takes to inform the host of a fault, the longer the user will take to correct the fault.

7.3 Application Hardening

The chosen case-study applications are simple but representative when considering resource usage and execution flow orientation: vector sum (VectorSum), matrix multiplication (Matrix), Fast Fourier Transform (FFT), and bitonic sort (Sort). The VectorSum is the shortest application because it only sums two vectors in the memory; therefore, it is a pure data flow-oriented application. The Matrix is mostly a data flow-oriented application. Still, it has a small fixed loop that iterates over the matrices. The FFT is a mix of control-flow and data-flow orientation. It has a more complex control than the Matrix but still performs heavy multiplications and additions. Finally, the Sort is mostly control-flow oriented, as it moves data according to data comparisons. Even though micro-benchmarks, these applications make the building blocks of major HPC and safety-critical applications (HASSANI; AIATULLAH; LUKSCH, 2014).

Table 7.1 shows the resource usage for the four case-study applications running on the 8-, 16-, and 32-core configurations.

Table 7.1: Program memory and runtime requirements for all FlexGripPlus configurations

Application	Program Memory (bytes)	Runtime (μs)		
		8 cores	16 cores	32 cores
FFT	1,344	964	588	406
Matrix Multiplication	264	320	224	177
Sort	288	824	610	502
VectorSum	563	141	103	85
Average	615	562	381	292

Table 7.2 shows the percentage execution time overhead over Table 7.1, individually, for the datapath duplication (T1) - DD - and the consistency checking (T2) with host notification (T3) - CC. Data have been calculated by measuring datapath duplication alone (T1) and its difference to all transformations combined (T1, T2, and T3), thus considering ILP and architectural characteristics of the GPU configuration. Therefore, to effectively harden a program code (and assess execution time overhead), one must combine the datapath duplication column with the consistency checking and host notification column, adding their respective percentage overheads. The datapath duplication column considers data for memory access and predicate setting instruction duplication (DD), Traceback optimization for memory access (DD [MEM]) and predicate setting (DD [PRED]), and the Move optimization ([M]). The consistency checking and host notification column considers data for the Traceback optimization for memory access (CC [MEM]) and predicate setting (CC [PRED]) and the Delayed Notification optimization ([D]). Data for checking both memory access and host notification requires adding columns MEM and PRED (optionally with [D]).

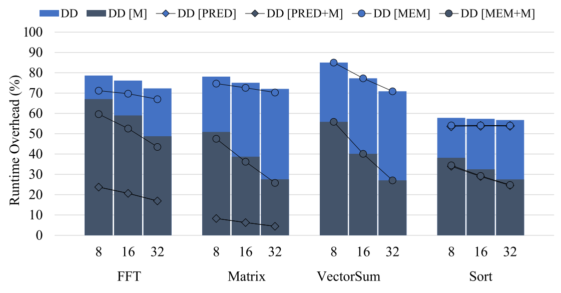
Figure 7.2 draws and discusses data from Table 7.2, considering the execution time overhead for all transformations running on 8-, 16-, and 32-core GPU configurations. Figure 7.2(a) depicts isolated overheads for datapath duplication (DD). It shows the following datapath duplication versions: DD, non-optimized; DD [M], optimized with Move; DD [PRED], optimized with Traceback PRED; DD [PRED+M], optimized with Traceback PRED and Move; DD [MEM], optimized with Traceback MEM; and DD [MEM+M], optimized with Traceback MEM and Move. Figure 7.2(b) shows the same hardening versions for consistency checking and host notification (CC). Finally, Figs. 7.2(c) and 7.2(d) show hardening versions for all transformations (DD+CC) without and with the Delayed

Table 7.2: Execution time overhead for datapath duplication, consistency checking, and host notification (%)

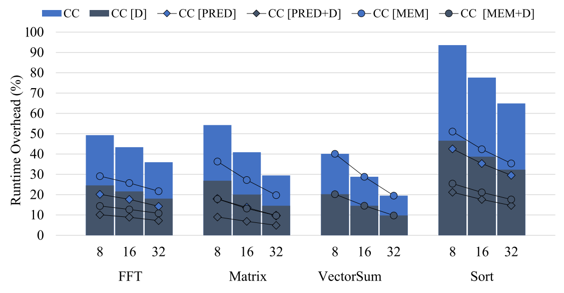
		Datapath duplication (DD)						Consistency checking and host notification (CC)			
Application	Cores	Full	Full [M]	MEM	MEM [M]	PRED	PRED [M]	MEM	MEM [D]	PRED	PRED [D]
FFT	8	79	67	71	60	24	24	29	14	20	10
	16	76	59	70	53	21	21	26	13	18	9
	32	72	49	67	43	17	17	22	11	14	7
Matrix	8	78	51	75	48	8	8	36	18	18	9
	16	75	39	73	36	6	6	27	13	14	7
	32	72	28	70	26	4	4	20	10	10	5
Sort	8	58	38	54	34	54	34	51	25	43	21
	16	57	32	54	29	54	29	42	21	35	18
	32	57	27	54	25	54	25	35	18	30	15
VectorSum	8	85	56	85	56	9	9	40	20	-	-
	16	77	40	77	40	7	7	29	15	-	-
	32	71	27	71	27	5	5	19	10	-	-
Average	8	75	53	71	50	24	19	39	19	27	13
	16	71	43	69	40	22	16	31	16	22	11
	32	68	33	66	30	20	13	24	12	18	9

Notification optimization, respectively.

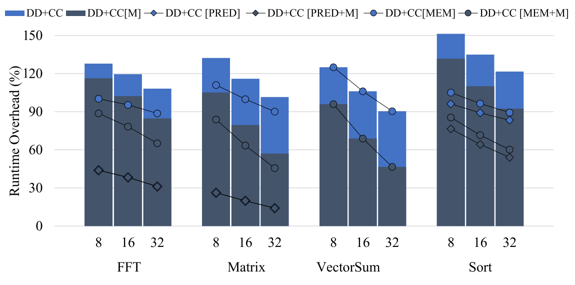
Figure 7.2: Transformations' runtime overhead for 8-, 16-, and 32-core configurations.



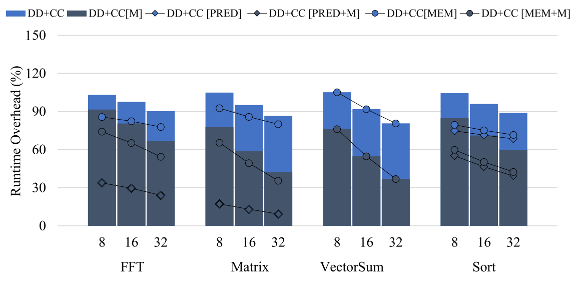
(a) Datapath duplication



(b) Consistency checking and host notification



(c) Transformations without Delayed Notification



(d) Transformations with Delayed Notification

Figure 7.2(a) shows that the Move optimization significantly reduces the datapath duplication overhead for all applications. This result was expected, as we designed this optimization to reduce execution time at a cost in reliability, later evaluated in Section 7.4. On the other hand, the Move optimization also produces an increased reduction for configurations with more cores. This effect can be explained by the fact that an arrangement with more parallel cores increases the number of concurrent global memory accesses.

Thus, by replacing load instructions with move instructions, we could alleviate the global memory pressure, which is then able to reduce performance degradation more efficiently in configurations with more cores. These data indicate that the Move optimization can perform even better in COTS GPUs due to their configuration with many cores.

Figure 7.2(a) also shows the Traceback optimization applied to the case-study applications. To evaluate this optimization, we must consider its application to the memory access and predicate setting instructions individually.

The performance gain is minimal when the Traceback optimization is applied to the memory access instructions. This result happens due to two main factors: most of the program code instructions are used for accessing memory data (i.e., data-flow), thus cannot be removed from the hardening; or the data-flow also uses most of the instructions used for calculating conditional branches, conditional executions, and loops (i.e., control-flow). With a small control flow, data-flow-oriented applications FFT, Matrix, and VectorSum showed little performance improvement. The VectorSum presented no improvement due to not having control-flow instructions. With a complex control flow, the Sort application also showed little performance improvement because of its entangled data and control flows. Unlike the Move optimization, the Traceback optimization is not a trade-off between performance and reliability. Therefore, even with small performance gains, applying it has no drawbacks.

When considering the Traceback optimization applied to the predicate setting instructions, the performance gain is much higher than when applied to the memory access instructions. For the same reasons discussed previously, applications with a heavy data flow can be more aggressively optimized as long as the data and control flow are not entangled. Therefore, one can notice a significant performance gain for the FFT and Matrix applications. Because the VectorSum has no control flow, its optimization would equal the original unhardened application. On the other hand, the Sort application showed the smallest performance gain because its instructions are mostly used for both control and data flow. The optimizations in runtime observed when protecting the FFT and Matrix applications show that the Traceback optimization can harden instructions selectively with significant performance gains.

Figure 7.2(b) shows that, for all applications, the Delayed Notification optimization reduces the runtime overhead by half. This reduction happens because it uses a single notification instruction for a group of checking instructions instead of one notification for each checking instruction. The highest reduction is achieved with a single notification instruction for the complete program code. The main drawback is that this optimization increases the average delay between identifying a fault and notifying the host. However, it should not decrease reliability. The selective protection improves performance significantly, especially when associated with the Delayed Notification. The number of cores demonstrates an impact similar to that observed in the datapath replication with the Move optimization (Figure 7.2(a)).

The total costs for the proposed fault tolerance techniques are presented in Figs. 7.2(c) and 7.2(d), without and with the Delayed Notification optimization, respectively. They add the datapath duplication costs presented in Figure 7.2(a) with the consistency checking and host notification costs presented in 7.2(b). Results without the optimization show that the Move and Delay optimizations together reduce the average performance cost from 134%, 119%, and 105% to 83%, 66%, and 51% for 8, 16, and 32 cores, respectively. When selective memory protection is implemented, the average cost drops to 69%, 55%, and 42% for 8, 16, and 32 cores. When selective predicate instruction is implemented, the average cost is 35%, 30%, and 24% for 8, 16, and 32 cores, respectively. Such results indicate that when performance is essential, selective protection should be applied whenever possible. In addition, the results show that the amount of cores not only speeds up the execution time of applications but also absorbs the cost of performance of fault tolerance techniques.

7.4 Fault injection results

Faults were injected in the original and hardened case-study applications. For each application, we considered three hardened versions: (1) *SDC Hard*, with consistency checking for memory access instructions and the Traceback MEM optimization; (2) *DUE Hard*, with consistency checking for predicate setting instructions and the Traceback PRED optimization; and (3) *Full Hard*, with consistency checking for memory access and predicate setting instructions. We considered the delayed branch optimization for all versions because previous works showed statistically the same fault reduction for both versions. Table 7.3 summarizes the final fault tolerance techniques' names and shows each tested version and optimizations applied.

Table 7.3: Implemented hardened versions

Label	Application hardening	Move	Traceback MEM	Traceback PRED	Delayed Notification
Unhardened					
SDC Hard	X		X		X
SDC Hard [M]	X	X	X		X
DUE Hard	X			X	X
DUE Hard [M]	X	X		X	X
Full Hard	X				X
Full Hard [M]	X	X			X

We injected 10,000 faults for each combination of (1) case-study application (FFT, Matrix, VectorSum, and Sort), (2) software version (original, SDC Hard, SDC Hard [M], DUE Hard, DUE Hard [M], Full Hard, and Full Hard [M]), (3) GPU configuration (8-, 16-

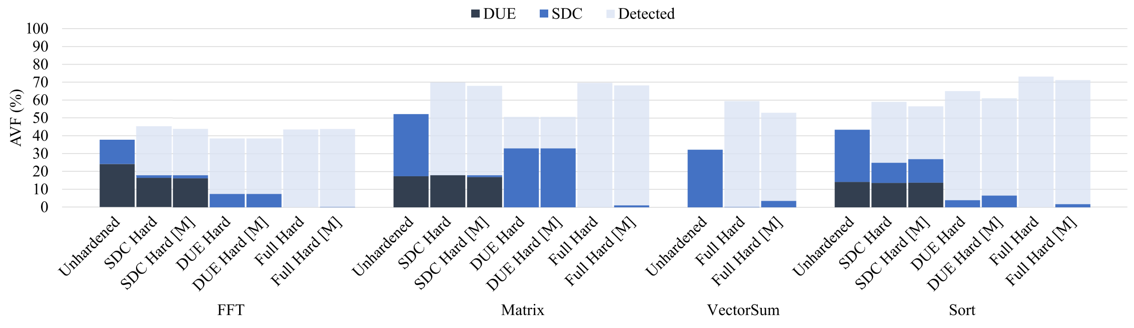
, 32-core), and (4) fault injection location (register file and pipeline registers), reaching 1,440,000 injected faults (VectorSum has only 3 software versions). Each of the 1.44 million faults was injected as a single fault per program execution, meaning that we ran the case-study applications a total of 1.44 million times, each with a single upset.

In the following, we discuss the results for faults injected into the register files and pipeline registers.

7.4.1 Register File

Figure 7.3 averages data for the 8-, 16-, and 32-core configurations for the fault injection in the register files, as they presented similar results. It shows, for all four case-study applications and all hardening versions (because the VectorSum does not have control-flow instructions, it does not have a DUE Hard version, and the SDC Hard is the same as the Full Hard), faults classified according to their effects (DUE, SDC, Detected, and Masked).

Figure 7.3: Fault effects distribution for faults injected into the register files.



Source: The Author.

When considering SDC effects, SDC Hard and SDC Hard [M] techniques effectively reduced DUE and SDC cases in all applications, showing an average error reduction rate of 88% and 82%, respectively. SDC Hard was able to detect faults more effectively for the data-flow-oriented applications (FFT, Matrix, and VectorSum), followed closely by SDC Hard [M]. On the other hand, neither technique could detect all SDCs for the FFT application because its control part is larger than in the other two applications. Also, when considering the Sort application, the SDC Hard and SDC Hard [M] showed poor results, being able to reduce SDCs by 61% and 55%, respectively. They could not effectively detect SDCs mainly because the Sort's control flow includes conditional instructions in its main loop. Therefore, protecting the memory access instructions alone leaves most of its dynamic instructions unprotected and prone to SDCs. An option to improve the SDC Hard techniques would be not simply targeting memory access instructions. Instead, evaluate all instructions more aggressively, considering their impact on causing SDC effects.

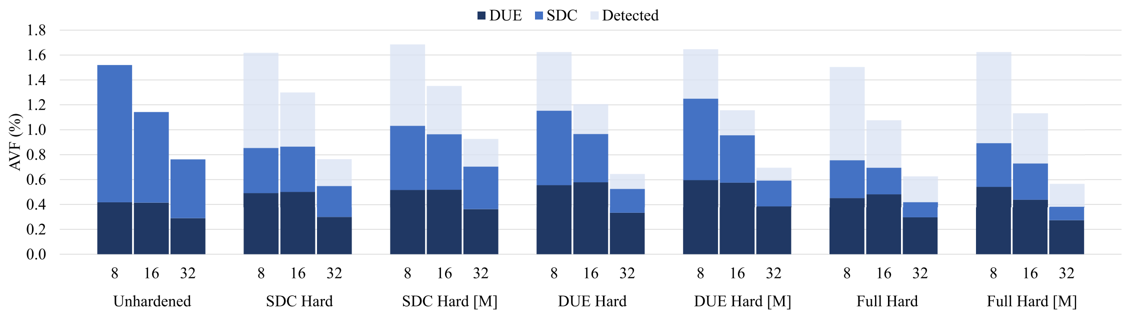
Unlike the memory access techniques, the DUE Hard and DUE hard [M] techniques eliminated all DUEs. Note that DUE Hard and DUE Hard [M] are the same for the data-flow applications because their control flows have no memory access instructions. They differ only for the Sort application. Still, for all applications, they detected all DUE effects. Also, note that both techniques were also able to reduce SDC effects for the FFT and the Sort application. Especially when considering the Sort, they achieved better detection capabilities for SDC effects than the SDC Hard techniques. As mentioned previously, the Sort has a conditional comparison instruction that belongs to its control flow, which can only be hardened by targeting predicate-setting instructions. For this application, a predicate setting instruction is far more relevant for SDC effects than the remaining memory access ones.

Finally, when targeting both SDC and DUE effects, one must use the Full Hard technique, followed closely by the Full Hard [M] (with increased SDC effects). As seen for the SDC and DUE Hard techniques, the SDC ones cannot detect DUE effects whatsoever, while the DUE ones cannot detect most SDC effects (except for the Sort, where it reaches 86%). These results show that there are better options than solely targeting either memory-access or predicate-setting instructions. One should more aggressively select, for each application, which instructions to harden.

7.4.2 Pipeline Registers

Figure 7.4 averages data of the four case-study applications for the fault injection in the pipeline registers, as they produced similar results. It presents faults classified according to their effects (DUE, SDC, and Detected) for all hardening techniques and configurations. Masked effects have been removed for clarity because they represent over 98% of the effects. Although only a small percentage of errors is observed, we can draw tendencies on the fault effects and the software-based hardening techniques detection capabilities.

Figure 7.4: Fault effects distribution for faults injected into the pipeline registers.

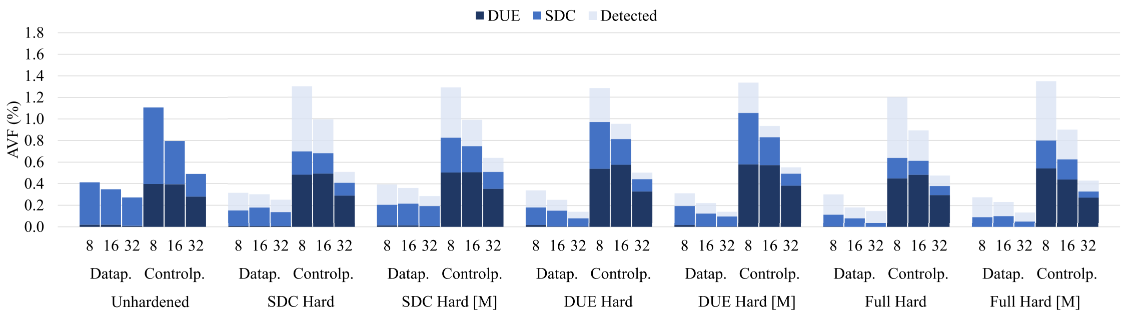


Source: The Author.

Considering the original unprotected application, one can notice that SDC effects happen more than DUE ones, up to 2.6 times for the 8-core configuration. As we increase the number of cores, the SDC rate decreases. At the same time, the DUE effects show a small reduction when moving from the 16- to 32-core configuration. Still, the 32-core configuration shows 1.6 SDCs for each DUE. This reduction happens because the 8- and 16-core configurations put increased pressure on the warp scheduler, resulting in more DUE effects. When we apply the hardening techniques, we achieve a reduction in SDC effects for all techniques, from a 28% reduction for the SDC Hard [M] (32-core) to a 77% reduction for the Full Hard [M] (32-core). Note that the DUE Hard techniques can also detect SDC effects, showing that memory access instructions are not fully correlated with SDC effects. On the other hand, the same cannot be achieved for DUE effects, to the point that, on average, they increase DUEs by 34%. To better understand the cause of this behavior, we classified the fault injection location as in the pipeline's datapath or controlpath.

Figure 7.5 distributes data from Figure 7.4 according to the fault injection location. It shows that fault effects are more common in the controlpath than in the datapath. In the datapath, faults are more easily masked due to a large percentage of unused bits during instruction execution. On the other hand, the controlpath has control bits responsible for the general GPU operation. When affected by a fault, they can more easily propagate it while executing instructions. As assembly instructions do not make these registers visible to the user, our proposed hardening techniques cannot directly target them. Also, by inserting additional instructions, our proposed techniques increase the chance of a DUE effect in the pipeline. To solve this issue, one should consider hardware-based techniques to selectively target specific registers in the pipeline.

Figure 7.5: Fault effects distribution for faults injected into the pipeline registers.



Source: The Author.

All hardening techniques positively affect the reduction of SDCs in the datapath and controlpath, especially those aimed at reducing SDCs and Full Hard. In the datapath, SDCs occur when a fault propagates to data registers. In the controlpath, SDCs occur mainly when a fault alters the instruction's operation, producing an incorrect value that propagates in the program code. In these cases, the faults are detected by our proposed hardening techniques.

7.5 Design Space Exploration

We used the AVF metric to measure and discuss the fault detection capabilities of the proposed software-based hardening techniques in the registers from the pipeline and the register files. The AVF is a useful metric for estimating the probability of failures in the presence of faults for each GPU configuration and the impact of our proposed techniques in reducing this value. On the other hand, the AVF metric fails to account for other essential metrics, such as performance and the number of sensitive bits (i.e., area). Therefore, measuring the reliability impact of using different GPU configurations is difficult since it directly affects the GPU's performance and area. For example, a 32-core configuration executes the generic application faster than an 8-core configuration, thus increasing reliability and having more sensitive bits, thus reducing reliability. On top of that, our proposed hardening techniques impact different core configuration performances in different ways.

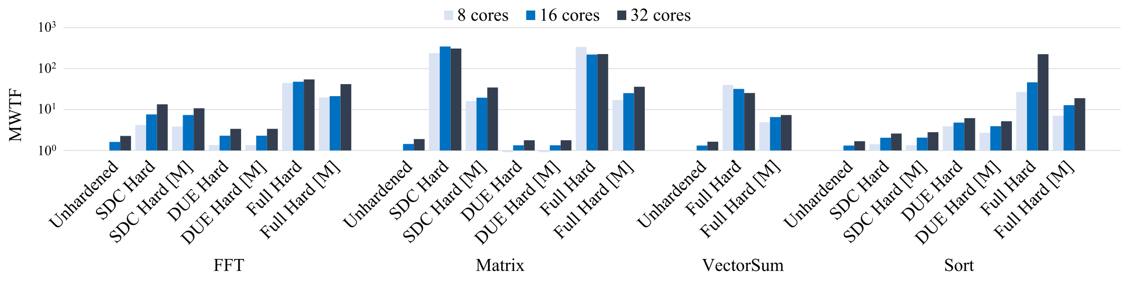
To account for AVF, performance, and area, we herein adopt a second reliability metric called *Mean Workload to Failure* (MWTF) (Reis et al., 2005), defined in Eq. 1. A larger MWTF means that more workload can be completed before the system fails. MWTF considers the AVF to a given effect (SDC, DUE, or both), workload execution time (i.e., application's runtime), and the raw error rate (i.e., GPU configuration's raw number of sensitive bits). Because the raw error rate also depends on the circuit technology and environmental conditions, we normalize MWTF over the 8-core configuration running original applications. The normalization process is performed by dividing, for each application, the MWTF of all implemented versions by the MWTF of the unhardened original version running on the 8-core GPU configuration. By doing so, we consider that all configurations are running on the same technology and environmental conditions. Therefore, we remove these factors from the equation, reducing the raw error rate to the number of available registers in each GPU configuration.

$$\text{MWTF} = (error\ rate \times AVF \times runtime)^{-1} \tag{7.1}$$

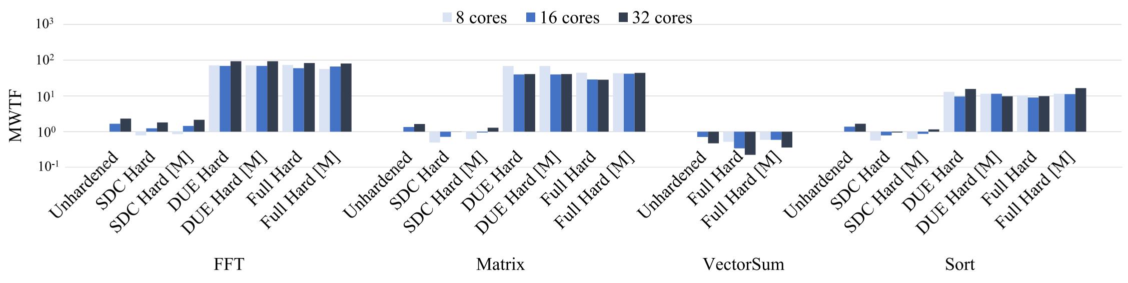
Figure 7.6 shows the normalized MWTF, where Figs. 7.6(a), 7.6(b), and 7.6(c) consider as AVF the SDC, DUE, and both effects, respectively. Note that the VectorSum application does not have a Traceback optimization version because it has no control-flow instructions. Thus, it only has the Full Hard and Full Hard [M] hardened versions, which are equivalent to the SDC Hard and SDC Hard [M] versions, respectively.

The Mean Workload to SDC Failure results (Figure 7.6(a)) show that hardening techniques improve MWTF in all applications, especially those optimized for reducing SDCs. The main reason for this improvement is that techniques can reduce SDC effects in both the register files and the pipeline, thus drastically reducing AVF and improving the MWTF up to 348 times. In most cases, the increase in cores improves the MWTF, indicating that the execution time is quite relevant. This trend can also be seen in the FFT

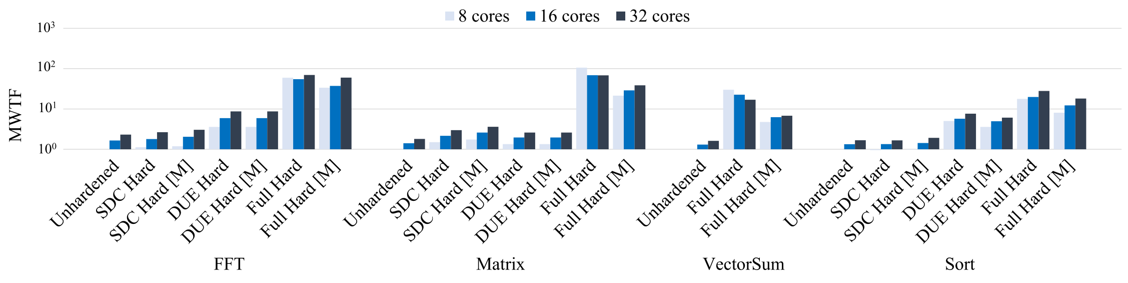
Figure 7.6: MWTF normalized over original 8-core configuration. (a) Mean Work to SDC Failure. (b) Mean Work to DUE Failure. (c) Mean Work to Failure.



(a) Mean Work to SDC Failure



(b) Mean Work to DUE Failure



(c) Mean Work to Failure

application running on the 32-core configuration, where the Full Hard [M] presented a better MWTF than the Full Hard, even though it is less effective in reducing errors. In other cases (Matrix with Full Hard and VectorSum with SDC Hard), reducing sensitive bits by reducing cores is the best option.

The Mean Workload to DUE Failure results (Figure 7.6(b)) show that the proposed techniques for reducing DUEs improve MWTF by 140 times, on average. On the other hand, SDC Hard and SDC Hard [M] make the applications more sensitive to DUE errors. This trend happens mainly because these hardening techniques do not target DUE effects. Thus, their additional instructions increase execution time and sensitivity to DUE effects without helping detect them. An increased execution time makes the pipeline more sensitive to DUE effects due to the additional assembly instructions executed by the GPU. When analyzing the impact of implemented techniques, we can see that the results vary from application to application. For example, the DUE Hard with the 32-core configuration for the FFT is the best option. In contrast, for the Matrix, the DUE Hard application running on the 8-core configuration improved MWTF. Note that, for all applications, DUE Hard is even more efficient than the Full Hard. This result shows that increasing the number of assembly instructions also increases DUE effects, indicating the high potential advantages of selective protection.

The Mean Workload to Failure results (Figure 7.6(c)) show that software-based hardening techniques improve MWTF, achieving up to 106 times improvement for the 8-core configuration on the Matrix application. As with selective protection of DUEs or SDCs, the ideal core configuration setting depends on the application. These results show that MWTF is an interesting metric, especially for applications that must calculate large workloads and take a long time to complete their tasks. For these cases, the results show that the best fault tolerance technique should consider an optimized balance between its effectiveness in reducing errors, execution time, sensitive bits, and AVF. In this scenario, configurable GPUs can be a good option.

7.6 Improving GPU register file reliability with ISA extension

To further improve software-based fault tolerance techniques, we also propose a comprehensive ISA extension composed of three classes of resilient atomic instructions (GONCALVES et al., 2020). The first two classes, which include load and store instructions, target specifically SDC effects for safety-critical applications. The third class, which includes set predicate instructions, targets DUE effects for HPC applications. The extension is developed to be deployed in tandem with software-based fault tolerance techniques, therefore taking advantage of both software- and hardware-based techniques benefits.

We propose three additional instructions to the NVIDIA SASS 1.0 ISA as a comprehensive ISA extension. The new proposed instructions are resilient atomic ones, being able to check the consistency of read registers, notify the host in case of mismatch, and duplicate a register write to the original register's replica in a single instruction. By doing so, this extended ISA is able to absorb multiple instructions into a single instruction execution and improve runtime overheads.

Memory access and set predicate instructions are the main sources of SDC and DUE effects, respectively (GONCALVES et al., 2020). Our comprehensive ISA extension proposes two classes of resilient atomic instructions, tackling SDC effects with resilient atomic load and store instructions and DUE effects with resilient atomic set predicate instructions. By doing so, we intend to remove additional instructions required by software-based hardening techniques to (i) duplicate load, store, and set predicate instructions, (ii) perform regular consistency checks to compare original and replicated data, and (iii) notify the host in case of fault detection.

When considering load and store instruction hardening by software-based techniques, for duplicating the original instructions, two individual instructions must go through the GPU pipeline, requiring two fetches, two decodes, up to four register file accesses, two executions, and two memory accesses. Additionally, the GPU must fully execute one extra instruction per instruction-used register for consistency checking. Finally, a host notification procedure is required, where a branch instruction and a subroutine for writing predefined memory locations with a predefined value alerting the host of a fault. Our proposed resilient atomic load and store, in a single instruction, can duplicate register access, check the read and the written values for consistency, and notify the host, performing a single fetch, decode, execution, and, most importantly, a single memory access.

When considering the set predicate hardening performed by software-based instructions, the procedure is basically the same: it duplicates the original instruction, inserts consistency checks, and optionally notifies the host. Even though our proposed resilient atomic set predicate instructions do not spare memory access, they absorb the original instruction's replica, the consistency checks, and the eventual host notification, decreasing execution time overhead compared to state-of-the-art software-based fault tolerance techniques.

The implementation of these instructions requires software and hardware support. The software support has to be able to generate program code considering these new instructions and insert them in a context where software-based hardening techniques can duplicate portions of the code and effectively use the new instructions. The hardware must be able to execute these new instructions and notify the host in case of fault detection. In the following Sections, we discuss the existing software-based techniques, how our ISA extension takes advantage of them, and the supporting hardware modifications required to support the new instructions.

7.6.1 Software Support

Software-based fault tolerance techniques detect faults by performing code transformations at different abstraction levels, from application code to assembly. The most common code transformation is to replicate a portion of the program code, regularly check it for consistency, and notify the user in case of a mismatch between the original code and its replica.

When considering SDC faults, where the execution flow of the program is correct, the best portion of the code to be replicated is the datapath, which includes memory access instructions and all the logic that leads to writing its registers, leaving branch instructions unprotected. For example, consider the store instruction "store R0, offset [R1]", where R0 is written to the memory address pointed by R1. In this case, the store instruction must be replicated and checked for consistency, while all instructions that form the logic cones that calculate the values of R0 and R1 must be simply replicated.

The same idea applies to DUE faults but considers set predicate instructions that write predicate registers used by conditional branch instructions. It is interesting to notice that, even though the controlpath and the datapath might share resources in the program code, portions of the program code are usually exclusively used for either the datapath or the controlpath. Therefore, as performed by previous works, full replication of used registers and their operation instructions can lead to unnecessary execution time overheads.

To support our proposed comprehensive ISA extension, two code transformations must be supported: (1) replace load, store, and set predicate instructions by resilient atomic ones and (2) duplicate instructions that belong to logic cones that lead to load, store, and set predicate instructions. Their implementations are later discussed in Section 7.6.3.3.

7.6.2 Hardware Support

As presented in Chapter 7, FlexGrip consists of arrays of Streaming Multiprocessors (SMs) used as Single-Instruction Multiple-Thread (SIMT) processors. Each SM has an individual pipeline with fetch, decode, read, execute, memory access, and write-back stages, besides a warp scheduler and a deep memory hierarchy, which contains General-Purpose Register Files (GPRFs), Predicate Register Files (PRFs), shared memories, constant memories, global memories, caches, among other storage elements.

The hardware support must be able to (i) decode new instructions, (ii) provide access to the register files and the different memory elements of the memory hierarchy, and (iii) effectively notify the host of fault detection. The first two items must be implemented across the pipeline, influencing the GPU's datapath and controlpath. The last item should be included in the GPU's exception circuitry. To maintain performance, hardware modification cannot increase critical path delays.

Even though commercial GPUs have restricted descriptions of their IPs, FlexGrip allows designers to implement the required hardware support.

7.6.3 Implementation

For these experiments, we chose five case-study applications: matrix multiplication, Fast Fourier Transform (FFT), vector sum, bitonic sort, and edge detection. All case studies differ in their use of the GPU's controlpath and datapath. In terms of dataflow and control-flow characteristics, matrix multiplication and vector sum are mostly data-flow-oriented, with few conditional deviations. The FFT, bitonic sort, and edge detection applications are mostly control-flow-oriented, with many conditional deviations. The matrix multiplication, FFT, vector sum, bitonic sort, and edge detection case-study applications have 64, 64, 256, 32, and 64 threads each, respectively.

The following Sections discuss the implementation of the three classes of resilient atomic instructions:raLoad (load instructions), raStore (store instructions), and raSetP (set predicate instructions). Discussions include ISA extension and instruction formats modification, hardware modifications to the FlexGrip architecture, and software modifications to the compilation flow to most effectively employ our proposed instructions with software-based hardening techniques.

7.6.3.1 ISA Implementation

The implementation of the resilient atomic instructions in the SASS 1.0 ISA poses two challenges: (1) to differentiate the resilient atomic opcodes from the original ones and (2) to allocate the addressing of the replicated register. As the new classes of instructions must perform the same functionalities as the original ones, used bits cannot be removed from the original instructions. Hence, we must use spare bits to improve the original instructions to become resilient atomic ones.

We used a single extra bit to solve the first challenge instead of creating new opcodes. We then defined it as '0' for the original instructions and '1' for their resilient atomic versions. Even though this approach requires one extra bit, it is compatible with legacy code, as unused bits are set to '0'.

To address the second challenge, we must consider the GPU register file and how instructions access it. FlexGrip can have up to 128 registers per thread, and load, store, and set predicate instructions can address up to 2 registers in their instruction formats. So, to directly access the extra registers, all resilient atomic instructions would require 14 spare bits. Unfortunately, they do not have as many spare bits: load and store have 12, and the set predicate has 7. Hence, two options arise: (1) to encode a subset of registers or (2) to encode an offset between registers and their replicas.

The first option limits the scope of replicated registers but allows programmers to replicate instruction registers partially, as they are individually addressed. On the other hand, the limited scope of replicated registers might force the software transformation to reallocate registers accordingly. The second option is less costly bitwise, as all replicated registers share the same address offset, but forces the hardware to calculate the effective address and the software transformation to allocate replicated registers with the same offset. Also, programmers must either harden all registers in an instruction or none.

Due to the number of available bits, our implementation used the first option for the resilient atomic load and store instructions (1 for opcode, 5 for the first replica, and 6 for the second replica) and the second option for the resilient atomic set predicate instructions (1 for opcode and 6 for offset).

7.6.3.2 Hardware Implementation

Hardware modifications have been made to the Decode, Read, and Write pipeline stages. We have also implemented an additional hardware exception for host notification. The Decode stage has three source registers (src1, src2, and src3) and one destination register (dest1). The load and store instructions use a single source register (src1), and the set predicate instructions use two source registers (src1 and src2). Thus, for the implementation of raLoad and raStore, we used src2 as the source register replica, and for raSetP we used src3. Because the Decode stage did not originally support a second destination register, we implemented it through dest2. We also implemented supporting control flags for the correct execution of the proposed instructions.

The Read and Write stages were adapted to consider an additional source operand for the new resilient instructions. For the raStore and raLoad, the additional source is directly the register address, but for the raSetP, it is an offset that must be added to the original register addresses to find their replicas. Global memory addresses from the memory access instructions are calculated by a specific module, which was adapted to read a second value from the register file (src2) and check it for consistency. The raSetP was adapted to calculate the replicated registers' addresses and check them for consistency. The raLoad was modified to copy the data loaded from memory to both dest1 and dest2 operands. Finally, we added an extra hardware exception for host notification.

To evaluate how the hardware modifications impact the FlexGrip architecture, we synthesized the original design and three modified versions of the FlexGrip architecture: (i) ISAset, with modifications for raSetP, (ii) ISAls, with modifications for raLoad and raStore, and (iii) ISAsetls, with modifications for the complete ISA extension. We performed the synthesis with 8 SP cores, a 15nm cell library (MARTINS et al., 2015), and a 500 MHz constraint. The evaluation considered circuit area, number of logic cells, power, and critical path delay.

Table 7.4 shows the synthesis reports for all three modified architectures. When considering the hardware implementation for the complete ISA extension, the circuit area showed an overhead of 0.179%, while the number of logic cells increased by 0.369%. Reduced versions of architecture ISAset and ISAls showed lower overheads, with a higher overhead caused by the raSetP due to a more expensive circuitry to calculate replicated addresses. Power measurements showed an increase of 0.125%, being ISAset and ISAls equally responsible for it. Finally, and most importantly, the critical path delay showed negligible improvements in less than 0.01% for all architectures. The impacts of the hardware modifications to support the proposed ISA extension in terms of circuit area, logic cell number, power, and critical path delay show that the discussed hardware implementations can be done without imposing significant performance penalties.

7.6.3.3 Software Implementation

We generated six hardened versions for each case-study application, divided into two classes: software-based hardening techniques and software-based hardening techniques with ISA extension. Each class protects memory access instructions (Memory), targeting SDC effects, set predicate instructions (Set Predicate), targeting DUE effects, and both (All).

Table 7.4: Hardware implementation overhead (%)

	Original Design	Hardware overhead (%)		
		ISAsetp	ISAls	ISAsetls
Area (mm²)	196,338	0.047	0.030	0.179
Cells (#)	377,798	0.152	0.031	0.369
Power (mW)	95,074	0.053	0.056	0.125
Delay (ns)	1,99205	-0.002	-0.001	-0.008

Table 7.5 shows code transformation examples for each hardened version. The original code (black) contains add, load, set predicate, conditional branch, and store instructions. For the software-based technique class, add and load instructions are replicated (green) over original registers' copies R1' and R2' to maintain consistency between register duplications. The store instructions should only be duplicated in case of memory replication. Set predicate instructions are inserted (blue) for consistency checking after memory access instructions (Memory), after set predicate instructions (Set Predicate), or after both (All). Finally, host notification is performed by a conditional branch instruction (yellow). For the ISA extension, the add instruction replication is maintained. The remaining load, store, and set predicate registers and their respective consistency checks are replaced (red) by raLoad and raStore (Memory), raSetP (Set Predicate), or both (All) instructions. The ISA extension also absorbs the host notification.

Table 7.5: Program code transformation by software-based techniques and software-based techniques with proposed ISA extension.

	Software-based			Software-based + ISA Extension		
Original Code	Memory	Set Predicate	All	Memory	Set Predicate	All
1: ADD R1, R1, 1	ADD R1, R1, 1	ADD R1, R1, 1	ADD R1, R1, 1	ADD R1, R1, 1	ADD R1, R1, 1	ADD R1, R1, 1
2:	ADD R1', R1', 1	ADD R1', R1', 1	ADD R1', R1', 1	ADD R1', R1', 1	ADD R1', R1', 1	ADD R1', R1', 1
3: LOAD R2, [R1]	LOAD R2, [R1]	LOAD R2, [R1]	LOAD R2, [R1]	**raLOAD R2, R2', [R1, R1']**	LOAD R2, [R1]	**raLOAD R2, R2', [R1, R1']**
4:	LOAD R2', [R1']	LOAD R2', [R1']	LOAD R2', [R1']		LOAD R2', [R1']	
5:	@!PE SETP.NE PE, R1, R1'		@!PE SETP.NE PE, R1, R1'			
6: SETP.NE P0, R2, R3	SETP.NE P0, R2, R3	SETP.NE P0, R2, R3	SETP.NE P0, R2, R3	SETP.NE P0, R2, R3	**raSETP.NE P0, R2, R3, offset**	**raSETP.NE P0, R2, R3, offset**
7:		@!PE SETP.NE PE, R2, R2'	@!PE SETP.NE PE, R2, R2'			
8:		@!PE SETP.NE PE, R3, R3'	@!PE SETP.NE PE, R3, R3'			
9: @P0 BRA 1	@P0 BRA 1	@P0 BRA 1	@P0 BRA 1	@P0 BRA 1	@P0 BRA 1	@P0 BRA 1
10: STORE [R4], R1	STORE [R4], R1	STORE [R4], R1	STORE [R4], R1	**raSTORE [R4, R4'], R1, R1'**	STORE [R4], R1	**raSTORE [R4, R4'], R1, R1'**
11:	@!PE SETP.NE PE, R1, R1'		@!PE SETP.NE PE, R1, R1'			
12:	@!PE SETP.NE PE, R4, R4'		@!PE SETP.NE PE, R4, R4'			
13:	@PE BRA ERROR	@PE BRA ERROR	@PE BRA ERROR			

Table 7.6 shows the execution time for the original applications and the overheads for their hardened versions. The software-based overheads (SW) show average increases of 94% for SDC detection (Memory), 85% for DUE detection (Set Predicate), and 104% for both (All). When using our proposed ISA extensions, these same values drop to 45%, 41%, and 54%, respectively, showing a decrease in execution time overhead of

around 50%. Data also show that dataflow-oriented applications, such as FFT and matrix multiplication, presented lower overheads than control-flow-oriented ones when targeting set predicate instructions for DUE effects. The vector sum application does not have predicate registers. The edge detection application uses all predicate registers and thus cannot be hardened purely by software-based techniques.

Table 7.6: Execution time overhead (%)

Application	Original (us)		Hardening technique overhead (%)		
			Memory	Set Predicate	All
FFT	964	SW	93.1	88.7	103.2
		ISA	56.3	24.5	65.6
Matrix Multiplication	320	SW	95.9	87.1	104.9
		ISA	41.9	9.3	46.9
Vector Sum	141	SW	105.2	-	-
		ISA	45.3	-	-
Bitonic Sort	824	SW	83.2	79.0	104.4
		ISA	30.3	55.9	36.6
Edge Detection	1,096	SW	-	-	-
		ISA	49.5	75.1	66.1

7.6.4 Evaluation

Faults were injected into original and hardened versions of the case-study applications. For each application version, we injected 10,000 faults, one per program execution, adding up to 280,000 simulations. Faults have been randomly distributed among original application-used registers from the GPRF, as unused and replicated registers were not sensitive to faults.

Tables 7.7 and 7.8 show the number of SDC and DUE effects in the original applications and the hardened version percentage reductions. For SDC effects, data show an average error reduction of 88.6% and 95.4%, respectively, for software-based techniques and ISA extension when the memory access instructions were protected (Memory). For DUE effects, data show an average error reduction of 99.9% and 100%, respectively, for software-based technique and ISA extension when set predicate instructions were hardened (Set Predicate). When targeting both, both versions achieved 100% fault detection for all applications but the FFT. Such results indicate that fault detection capabilities of software-based hardening techniques can be improved with our proposed ISA extension.

The software-based hardening techniques with our proposed ISA could not detect all errors for all applications. This happens mainly because our proposed ISA performs consistency checks before accessing the memory. Therefore, there is a small window in which a fault can affect the memory.

Table 7.7: SDC reduction (%)

Application	SDC Effects		Hardening technique		
			Memory	Set Predicate	All
FFT	1,452	SW	89.0	39.9	100.0
		ISA	84.9	17.3	96.1
Matrix Multiplication	3,461	SW	99.9	-4.8	99.9
		ISA	99.9	-5.7	99.9
Vector Sum	3,164	SW	100.0	-	-
		ISA	100.0	-	-
Bitonic Sort	1,739	SW	61.1	87.5	100.0
		ISA	92.5	71.1	99.8
Edge Detection	258	SW	-	-	-
		ISA	99.2	80.2	100.0

Table 7.8: DUE reduction (%)

Application	DUE Effects		Hardening technique		
			Memory	Set Predicate	All
FFT	2,321	SW	30.1	100.0	99.9
		ISA	32.3	100.0	100.0
Matrix Multiplication	1,755	SW	0.2	99.9	99.8
		ISA	4.0	100.0	100.0
Vector Sum	1	SW	99.9	-	-
		ISA	99.9	-	-
Bitonic Sort	1,368	SW	1.9	99.9	99.9
		ISA	5.3	100.0	100.0
Edge Detection	1,952	SW	-	-	-
		ISA	12.7	99.9	99.9

7.7 Hybrid Techniques for Error Mitigation in GPU Pipelines

In the preceding sections of this chapter, our focus was predominantly on the reliability of register files within GPUs, exploring the sensitivity of pipeline registers, and assessing the impact of various fault tolerance techniques. Although these techniques have shown effectiveness in detecting errors in register files, their efficiency in identifying errors within pipeline registers has been limited. Moving forward, we shift our attention to the pipeline itself, a critical yet vulnerable component of GPU architecture.

We introduce advanced techniques specifically tailored for pipeline error management, aiming to detect and correct faults. Our methods build on hybrid approaches incorporating ISA extensions, as mentioned in key works like (Mahmoud et al., 2018; SULLIVAN et al., 2018). These software-directed techniques are rooted in hardware functionality, enhancing error detection and host notification capabilities.

Our strategies represent a comprehensive advancement in fault management: they integrate the three critical steps of instruction replication, comparison for checking, and host notification—all executed via hardware. Beginning with an XOR-based method adapted from current literature, we refine it further to enable fault correction, particularly within the datapath. Subsequently, we implemented a Parity mechanism focused on the control-path, facilitating both detection and correction through software directives. This dual-pronged approach enhances the GPU pipelines' reliability and presents a significant innovation in the realm of software-directed hardware-based fault tolerance.

7.7.1 Pipeline Evaluation Methodology

Pipeline Registers (PRs) play a crucial role in GPU architecture, serving a dual function. They facilitate data transfer between pipeline stages and handle the control execution of instructions. This includes managing program counters, active wire masks, memory address offsets, and other critical elements. Our approach involves modifications to the pipeline architecture and the introduction of new instructions in the ISA to support this enhanced fault detection and correction mechanism.

Regarding reliability, it's important to note that faults affecting the datapath PRs often lead to data-flow errors (SDCs). In contrast, faults in the control-path PRs primarily cause control-flow errors (DUEs). The quantity of PRs varies based on the SMs and SPs configuration in a GPU architecture. Each SM has its control set, and each SP has its datapath. Consequently, the count of PRs in the control path is linked to the number of SMs, while the quantity in the datapath corresponds to the number of SPs. Our FlexGripPlus configuration, which includes an SM with 32 SPs, resulted in a total of 1841 bits of PR in the control path and 6144 bits of PR in the datapath.

For the XOR Technique, four case-study algorithms were used to exercise the GPU during the fault injection: Matrix Multiplication, Fast Fourier Transform (FFT), Vector Sum, and Bitonic Sort. To further our analysis with the Parity technique, we added M3 and Edge Detection algorithms, thus enhancing the diversity and robustness of our case study pool.

We executed a comprehensive fault injection campaign across all case-study applications and their corresponding hardened variants, spanning the four defined hardening classes: Operation instructions (*Op*), Memory instructions (*Mem*), Predicate instructions (*Pred*), and All instructions (*All*). This campaign encompassed the injection of 20,000 faults for each application version. These faults were subsequently categorized according to their impact on the FlexGrip GPU (Masked, DUE, or SDC faults).

7.7.2 Software Support

Our techniques divide responsibilities into software and hardware support. Software support enhances application code by selectively duplicating critical ISA instruction classes and modifying opcodes to indicate protected classes. This approach is uniformly applied to both the XOR and Parity techniques, ensuring consistency in software-level fault management. The hardware execution of these techniques, including the distinct mechanisms for XOR and Parity, will be detailed in the following sections.

In more detail, software support modifies the program code and creates a new application program by performing the following modifications: (i) selective hardening of instruction classes, (ii) modification of opcodes to mark hardened instructions to the Decode stage, and (iii) adjustment of binary codes to match the modifications. Meanwhile, hardware support must do the following: (i) decode new instruction code generated by software, (ii) integrate hardware modules for storing XOR information, (iii) execute the XOR technique flow, and (iv) implement instruction correction for detected faults.

Enhancements to HPCT offer comprehensive protection for the entire FlexGrip ISA, including selective techniques for specific instructions and registers. We categorized protection into four variants: Operations (*Op*), Memory (*Mem*), Predicates (*Pred*), and Full Coverage (*All*). *Op* applies XOR to specific instructions like MOV, MVI, IMUL, IADD, and LOP. *Pred* protects predicate instructions, crucial for GPU flow control, including ISET. *Mem* protects global memory access instructions like GLD and GST. *All* combines these protections, excluding NOP, synchronization instructions, and certain control-flow instructions (i.e., BRA, BAR, RET, SSY, and CAL).

Table 7.9 illustrates an example of assembly code and its respective hardened versions for the four categories mentioned: Operation (*Op*), Memory (*Mem*), Predicate (*Pred*), and Full Coverage (*All*). The first column presents the original application code, while the subsequent columns display the corresponding enhanced versions. Unhardened instructions can be seen in black, while instructions that have undergone modifications are highlighted in green. Note that we do not replicate the instructions in lines 2, 6, and 7.

Table 7.9: Software transformation example

	Original Code	Op	Pred	Mem	All
1:	ISET.C0 R1, R2, LE;	ISET.C0 R1, R2, LE;	ISET.RES.C0 R1, R2, LE;	ISET.C0 R1, R2, LE;	ISET.RES.C0 R1, R2, LE;
2:	RET C0.NE;	RET C0.NE;	RET C0.NE;	RET C0.NE;	RET C0.NE;
3:	IADD32 R3, g [0x6], R3;	IADD32.RES R3, g [0x6], R3;	IADD32 R3, g [0x6], R3;	IADD32 R3, g [0x6], R3;	IADD32.RES R3, g [0x6], R3;
4:	MOV R1, g [0x7];	MOV.RES R1, g [0x7];	MOV R1, g [0x7];	MOV R1, g [0x7];	MOV.RES R1, g [0x7];
5:	GST global14[R3], R8;	GST global14[R3], R8;	GST global14[R3], R8;	GST.RES global14 [R3], R8;	GST.RES global14 [R3], R8;
6:	BRA C0.NE, 0x90;	BRA C0.NE, 0x90;	BRA C0.NE, 0x90;	BRA C0.NE, 0x90;	BRA C0.NE, 0x118;
7:	NOP;	NOP;	NOP;	NOP;	NOP;

The modified instructions are differentiated from the original instructions by adding a *.RES* to their terminology. For example, the *MOV* instruction becomes *MOV.RES* after modification. To signal which instructions must be protected by the hardware modules, HPCT changes the binary code of the original instruction by selecting an unused bit and setting it accordingly. This change is made only in the replicated instruction classes, and the other bits retain their original values.

The following subsections detail the hardware support that defines our XOR and Parity approaches and the experimental results that validate our methods.

7.7.3 Software-Directed Hardware-Implemented XOR Approach

The FlexGrip hardware undergoes significant modifications in the Decode and Write pipeline stages. In the Decode stage, the system checks whether an incoming instruction needs protection based on a software-enabled signal. If protection is required, the Decode stage interrupts the previous pipeline stage (Fetch) and instructs subsequent steps to apply XOR to that instruction. After, the same instruction is forwarded to the next stage (Read), and the previous step (Fetch) is freed to receive a new instruction.

When the initial protected instruction reaches the Write pipeline stage, the regular write operation to the destination register is saved in a temporary register. Consequently, no data is written to the GPU's output registers, stopping the instruction flow. Upon the second instruction reaching the Read stage, an XOR operation is performed between its outcome and the previous instruction's outcome stored in the temporary register. If the XOR result is not zero, an error notification triggers another re-execution of the current instruction. Subsequently, a new correction instruction is executed to rectify the application's state.

This hardware-based approach offers advantages like simplified software development, faster re-execution, and efficient handling of replication, verification, and notification within the Decode and Write stages. However, it can't detect faults affecting the preceding Fetch stage and is susceptible to control path PRs errors, as the XOR operation exclusively focuses on the instruction's result in the output register.

Table 7.10 displays execution times for selected case-study applications, along with execution time overhead for their hardened versions. The *Mem* version has an av-

Table 7.10: Execution time and execution time overhead

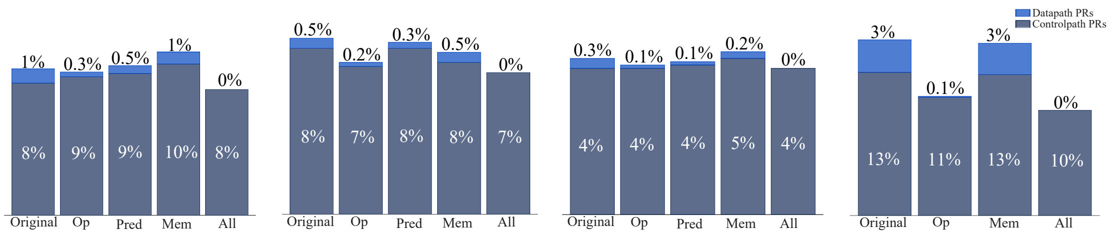
Application	Original (μs)	Operation (Op)	Predicate (Pred)	Memory (Mem)	All
FFT	406.3	1.42x	1.03x	1.28x	1.74x
Matrix Mult.	177.3	1.22x	1.02x	1.50x	1.74x
Bitonic Sort	501.5	1.10x	1.07x	1.35x	1.53x
Vector Sum	84.7	1.18x	–	1.49x	1.66x
Average	**292.5**	**1.23x**	**1.04x**	**1.41x**	**1.66x**

erage overhead of 1.41x, the *Op* version averages 1.23x, and the *Pred* version averages 1.04x. Notably, the FFT application bucks this trend, showing lower execution time in the *Mem* version compared to the *Op* version. This divergence stems from memory access instructions, which typically require three times more clock cycles than standard instructions, as discussed in previous sections.

7.7.3.1 Experimental results

Figure 7.7 shows the percentage of errors for all case-study applications' hardened versions, colored according to the PR's controlpath or datapath function. As previously mentioned, controlpath PRs are responsible for executing the instruction, and datapath PRs are responsible for moving data through the pipeline. Despite variations in execution flow orientation across applications, some consistent trends are evident. The XOR hybrid technique, while effective in detecting datapath PR faults (reducing them to 0% for all applications), has a less significant impact on controlpath PR errors. For instance, in the Vector Sum application, the All group only reduced errors from 16% down to 10%—a modest 37% reduction. This trend is observed across other applications, where error reduction is less pronounced.

Figure 7.7: Fault injection results



(a) FFT (b) Matri Mult. (c) Bitonic Sort (d) Vector Sum

Source: The Author.

Table 7.11 details the error reductions achieved by the XOR hybrid technique, segregated into datapath and controlpath PR errors. For controlpath PR errors, the XOR technique's impact varies, with the All version only reducing these errors by 11%. Other versions, such as Pred, even showed a 4% increase in errors. This variability can be

Table 7.11: Datapath and controlpath PRs error reduction (%)

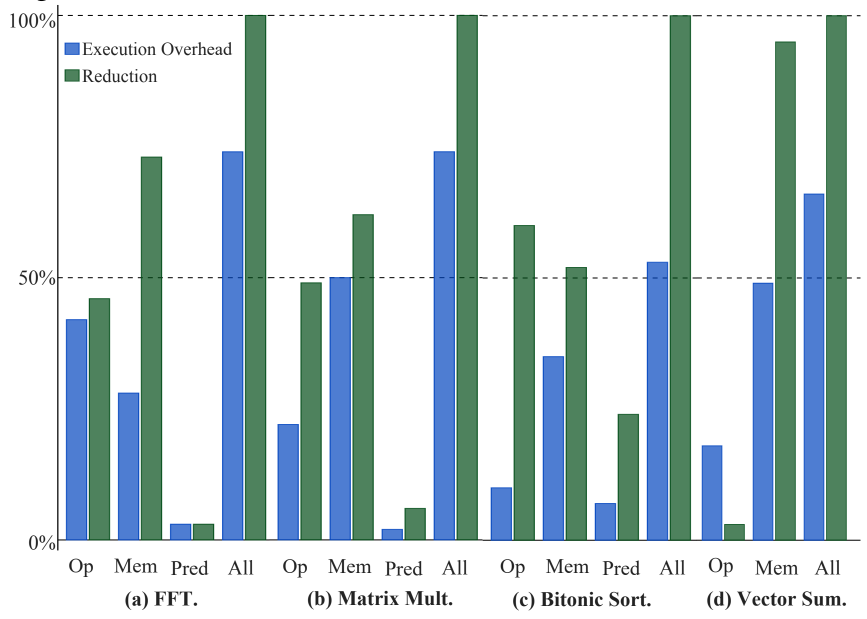
Application	Datapath PRs				Controlpath PRs			
	Op	Pred	Mem	All	Op	Pred	Mem	All
FFT	46	3	73	100	-8	-14	-4	5
Matrix	49	6	62	100	0	8	10	14
Sort	60	24	52	100	-1	-7	0	-1
VectorSum	3	–	95	100	1	–	17	26
Average	40	11	71	100	0	-4	9	11

attributed to the increased complexity in the GPU's pipeline due to the XOR circuitry, which paradoxically makes the controlpath more fault-prone.

In contrast, the XOR technique demonstrates high efficacy for datapath PR errors, achieving a 100% error reduction across all applications, as illustrated in Figure 7.8. Considering its average execution time overhead of 1.66x, this technique emerges as a viable alternative to conventional hardware methods like DWC, considerably increasing hardware usage. Moreover, by adjusting the hardening approach with different versions like Mem, Op, and Pred, one can achieve lower execution time overheads while still attaining significant error reductions.

In summary, while the XOR hybrid technique excels in mitigating datapath PR errors, it faces challenges in effectively addressing controlpath PR errors due to the increased complexity in the GPU's pipeline. This conclusion sets the stage for the subsequent section, where we present the Parity approach as a potential solution for more effectively handling controlpath PR errors, aiming to complement the XOR technique's strengths and address its limitations.

Figure 7.8: Results for execution time overhead and error reduction.



Source: The Author.

7.7.4 Software-Directed Hardware-Implemented Pipeline Parity

The architectural modifications for the software-directed Pipeline Parity comprise four steps: (i) enhance the Decode stage to discern reliability bits in the newly generated instructions from the originals, (ii) integrate parity check circuits into the FlexGrip pipeline stages at each stage, focusing on control registers, (iii) centralize notifications of parity check on a single exception signal, and (iv) add correction when a failure is detected.

Identical to the hybrid XOR technique, the Decode stage has been adapted to take into account the bit position allocated by the SASS compiler to store the reliability bit. Upon detecting a hardened version, decoding initiates a control sequence to re-execute the original instruction and activate the parity circuit for parity calculation and storage or comparison. One limitation is its inability to detect control flow errors that precede the Decode stage.

The first execution of the instruction calculates the parity of all the output registers (except the data ones); that is, the parity verifies the number of bits that have the value 1 in the output. Any bit-flip fault injection into the parity registers will be different. Parity checks are performed at each pipeline stage, ensuring control register integrity throughout the pipeline. The consistency checking between the parities of the two executions is performed. If an error is detected, the hardware performs a new execution of the control logic of the instruction that identified the error, thus correcting the error and resuming normal execution of the application.

Table 7.12 shows the performance overhead ranged from 1.01 to 1.23 times the original application (1% to 23% runtime overhead), demonstrating that even with all instructions hardened against pipeline register control flow errors, the performance impact remains manageable. Tailoring fault tolerance through software customization of hardening strategies further enhances cost-effectiveness for specific applications.

Table 7.12: Execution time and execution time overhead

Application	Unhardened (μs)	Hardened Time Overhead			
		Operation (Op)	Memory (Mem)	Predicate (Pred)	All
FFT	406.3	1.18x	1.03x	1.02x	1.23x
Matrix Mult.	177.3	1.09x	1.03x	1.01x	1.14x
Vector Sum	84.7	1.07x	1.03x	–	1.10x
Bitonic Sort	501.5	1.05x	1.03x	1.03x	1.11x
Edge Detection	3276.1	1.11x	1.02x	1.02x	1.15x
M3	316.7	1.16x	1.15x	1.00x	1.19x
Average	793.8	1.11x	1.05x	1.02x	1.15x

7.7.4.1 Experimental Results

Table 7.13 shows error reduction for the proposed hardening variants, and Figure 7.9 combines its data with execution time overhead to evaluate the efficiency of each hardening class. Our proposed fault tolerance technique failed to drastically reduce pipeline errors as it only targets a portion of pipeline records. However, except for the FFT application hardened with *Pred*, all versions presented error reduction and correction for all of them, in addition to not causing a significant impact on execution time, showing that we can effectively increase GPU reliability with our proposed approach.

The *All* hardening strategy achieved the most significant error reduction, reducing pipeline failures by up to 57% with a cost in execution time reaching up to 11% for the Bitonic Sort application. The Vector Sum app recorded the smallest decrease, with a 36% reduction. When considering the impact on execution time, FFT stood out as the most expensive, as its cost increase was more than double that observed in Sort and Vector Sum. These data highlight that the benefit of our approach may vary depending on the type of application.

The *Mem* implementation stood out as one of the most effective choices, causing a small increase of approximately 3% in execution time in all applications except M3, with a 15% increase. Additionally, it provided an average error reduction of 23%, with the largest reduction reaching 33% in the Matrix Mult. application. This approach proved to be the most effective in terms of reducing errors in relation to increasing execution time.

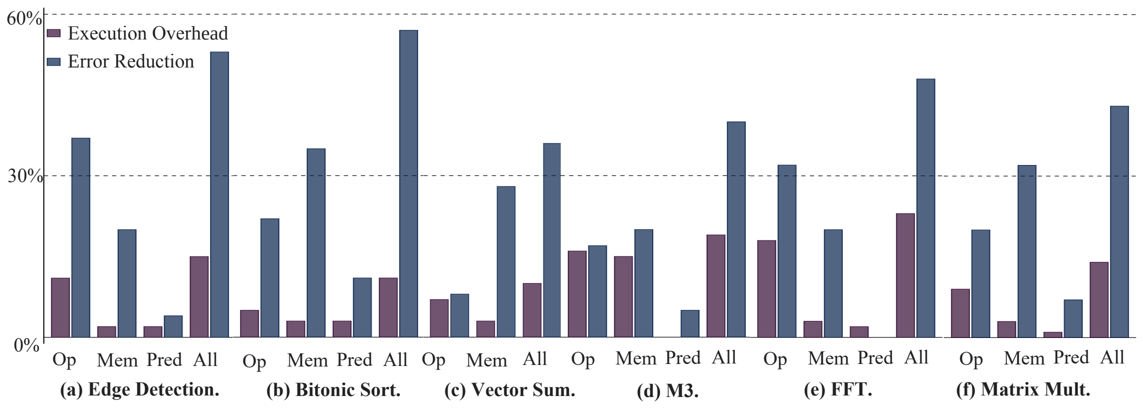
On the other hand, implementing the *Op* protection has had varying results depending on the application. It presented better error reduction than the *Mem* for the FFT, Bitonic Sort, Edge Detection, and M3 applications and worse for the Matrix Multi. and Vector Sum. However, it also required more execution time. When considering the higher overhead costs, *Op* proved less effective across all applications.

Lastly, the *Pred* protection category imposed the smallest execution time penalties since it focuses on a narrower set of instructions to be hardened. However, it also showed the smallest decrease in errors. Importantly, for all instruction classes, we were able to correct all detected faults.

Table 7.13: Fault injection results and error reduction

Application	Unhardened (%)	Error Reduction over Unhardened (%)			
		Operation (Op)	Memory (Mem)	Predicate (Pred)	All
FFT	8.7	32	20	0	48
Matrix Mult.	8.1	21	33	9	44
Vector Sum	13.3	8	27	–	36
Bitonic Sort	4.6	22	17	13	57
Edge Detection	2.7	30	19	4	56
M3	6.5	20	20	2	40
Average	7.3	22	23	6	47

Figure 7.9: Results for execution time overhead and error reduction.



Source: The Author.

7.8 Case-study Summary

This case study evaluated low-level software-based fault tolerance techniques designed to detect SEU effects in configurable GPU architectures. We adapted and implemented state-of-the-art software-based fault tolerance techniques through low-level assembly code transformations and proposed three optimizations to improve the performance of a set of case-study applications. These optimizations were performed with costs in reliability, fault detection for specific effects, and host notification time. These novel technique optimizations, called *Traceback*, *Move*, and *Delayed Notification*, were automatically applied to selectively protect three groups of instructions: memory access instructions, predicate setting instructions, and all instructions. We ran four case-study applications on three GPU configurations to measure the impact of the techniques and optimizations on the configurable GPU. Moreover, we investigated the sensitivity of register files and pipeline registers to radiation-induced faults. A fault injection campaign was performed through simulation at RTL with over 1.4 million faults injected to evaluate the GPU's susceptibility to SEUs.

We measured error rate, AVF, and runtime for 216 scenarios, varying hardening techniques, optimizations, core configurations, and case-study applications to explore the design space for hardening the FlexGrip configurable GPU architecture. We then presented these data with the MWTF metric, which factors measured data and normalized them over each original unhardened application running on the 8-core GPU configuration. Our comprehensive design space exploration shows that the most reliable GPU architecture configuration is not intuitive when equality factoring error rate, AVF, and runtime. Instead, a balance between error reduction, execution time, raw error rate, and AVF must be considered to optimize the GPU architecture for a given application, environment, or device. Finally, we conclude that navigating the reliability design space is mandatory to improve hardened application efficiency, as simply applying the most expensive software-based technique to the largest available hardware does not guarantee the best reliability.

The low-level software-based fault tolerance techniques we've developed promise universal applicability, offering the potential to refine and augment state-of-the-art strategies. These techniques, designed for the assembly-level application, could enhance methods like those presented in (OLIVEIRA et al., 2014) and (KALRA et al., 2020), which are implemented at higher abstraction levels, to improve SDC and DUE detection across various architectures. Notably, our findings suggest that leveraging more computing cores results in reduced overhead, shedding light on the efficiency gains observed in NVIDIA's SINRG (Mahmoud et al., 2018) when applied to commercial GPUs with thousands of CUDA cores, as opposed to the limited core counts in softcore GPU implementations. This indicates that our techniques could yield even greater efficiencies if integrated into the production compiler of a commercial GPU, thereby fully exploiting the computational resources of these advanced architectures to bolster fault tolerance.

Moving forward, we extended this work by introducing a comprehensive ISA extension to the NVIDIA SASS 1.0 ISA. This extension added three additional resilient atomic instructions targeting memory access and set predicate instructions to mitigate SDC and DUE effects. These instructions were incorporated into the existing software-based hardening techniques and automatically applied to five case-study applications. Hardware synthesis results showed no performance degradation and less than 1% area and power overheads. Execution runtime overheads showed a decrease compared to state-of-the-art software-based techniques. A subsequent fault injection campaign involving 280,000 faults confirmed the efficacy of this ISA extension, indicating that it could improve overall software-based fault detection, especially when factoring in DUE effects. To the best of our knowledge, this is the first work in the literature that proposes and implements atomic instructions specifically designed to mitigate SDC and DUE effects independently.

Building upon the foundation laid by our ISA extensions, we shifted our focus from the register files to the GPU pipeline. In our comprehensive evaluation of GPU pipeline hardening techniques, conducted through a fault injection campaign involving 1.000.000 faults, we proposed XOR and Parity methods, each demonstrating unique strengths and limitations. The hybrid XOR fault tolerance technique proved highly effective in detecting and correcting faults within the datapath, achieving 100% protection against dataflow errors. However, its efficacy was not as pronounced in managing controlpath errors, resulting in some instances of reduced GPU responsiveness for certain protection settings. This trend was reflected in the modest error reduction rates and an execution time increase ranging from 1.02x to 1.74x. However, these costs were deemed reasonable compared to DWC-based methods.

On the other hand, the proposed Parity technique, aimed at protecting the GPU pipeline against radiation-induced effects, showed promising results in both error reduction and performance overhead. This technique was implemented as a hybrid of software and hardware modifications, allowing for targeted protection of specific instruction groups. Error reductions of up to 59% were achieved with a runtime overhead of just 11%, and subsequent improvements led to even more efficient error reductions, with performance degradations ranging from 1% to 23%. Notably, all detected faults were corrected, indicating potential for future detection and correction capabilities enhancements.

In comparing our XOR and Parity techniques with NVIDIA's state-of-the-art SwapCodes and SInRG (SULLIVAN et al., 2018; Mahmoud et al., 2018) approaches, several key distinctions emerge. NVIDIA works utilize software control for hardware-mediated error mitigation. SInRG employs a unique XOR instruction comparison and host notification. SwapCodes uses register file ECC for error detection and correction. While these works primarily address datapath errors through instruction replication, our methods extend this concept by implementing instruction replication, error checking, and correction entirely in hardware. Our XOR technique aligns with NVIDIA's focus on datapath errors. Still, it further introduces the Parity technique, specifically targeting controlpath errors—a dimension not explicitly addressed in NVIDIA's approaches. Furthermore, our validation on a Softcore GPU, in contrast to NVIDIA's partial hardware simulations, offers a comprehensive and practical perspective on the effectiveness of these techniques. This holistic approach underlines the potential of our strategies to enhance fault tolerance in GPU architectures, paving the way for future innovations in the field.

Advancing our exploration, the next Chapter focuses on FGPU, a more recent softcore GPU, within the context of SRAM-based FPGAs. This segment aims to deepen our understanding of GPU reliability challenges and solutions in this evolving technological landscape.

8 RELIABILITY EVALUATION OF FGPU IN SRAM-BASED FPGA

This Chapter focuses on evaluating the reliability of applications running on FGPU embedded into an SRAM-based FPGA. The evaluation begins with a twofold approach: first, it investigates the trade-offs between hard Floating Point (FP) implementations, which utilize hardware accelerators, and soft-FP implementations, where FP instructions are emulated by software. The aim is to assess whether the computational speed advantage gained by hard FP justifies its increased resource utilization, especially in the context of radiation-induced faults (GONCALVES et al., 2020). Second, the Chapter explores the application and effectiveness of selective Triple Modular Redundancy (TMR) in enhancing the FGPU's reliability (BRAGA et al., 2021).

To comprehensively understand these aspects, the Chapter conducts a bitstream fault-injection campaign on various case-study applications, calculating their Mean Workload Between Failure (MWBF) to gauge reliability. Additionally, this Chapter investigates the reliability impacts of neutron-induced soft errors, specifically on aerial image classification CNNs executed on the FGPU. This investigation is achieved through both configuration bitstream fault injection and neutron irradiation experiments, providing insights into the robustness of softcore GPUs under such conditions (BENEVENUTI et al., 2022).

Section 8.1 outlines the experimental setup and board used across reliability evaluations. Section 8.2 examines the robustness of different FP implementations. Section 8.3 delves into the selective TMR application and its impact on the FGPU's reliability. The exploration of neutron-induced soft errors and their effects on the FGPU, particularly regarding aerial image classification CNNs, is detailed in Section 8.4. Specifics of the irradiation experimental setup will be provided in this section. Finally, Section 8.5 summarizes the key insights and their implications for advancing the reliability of softcore GPUs within FPGA contexts.

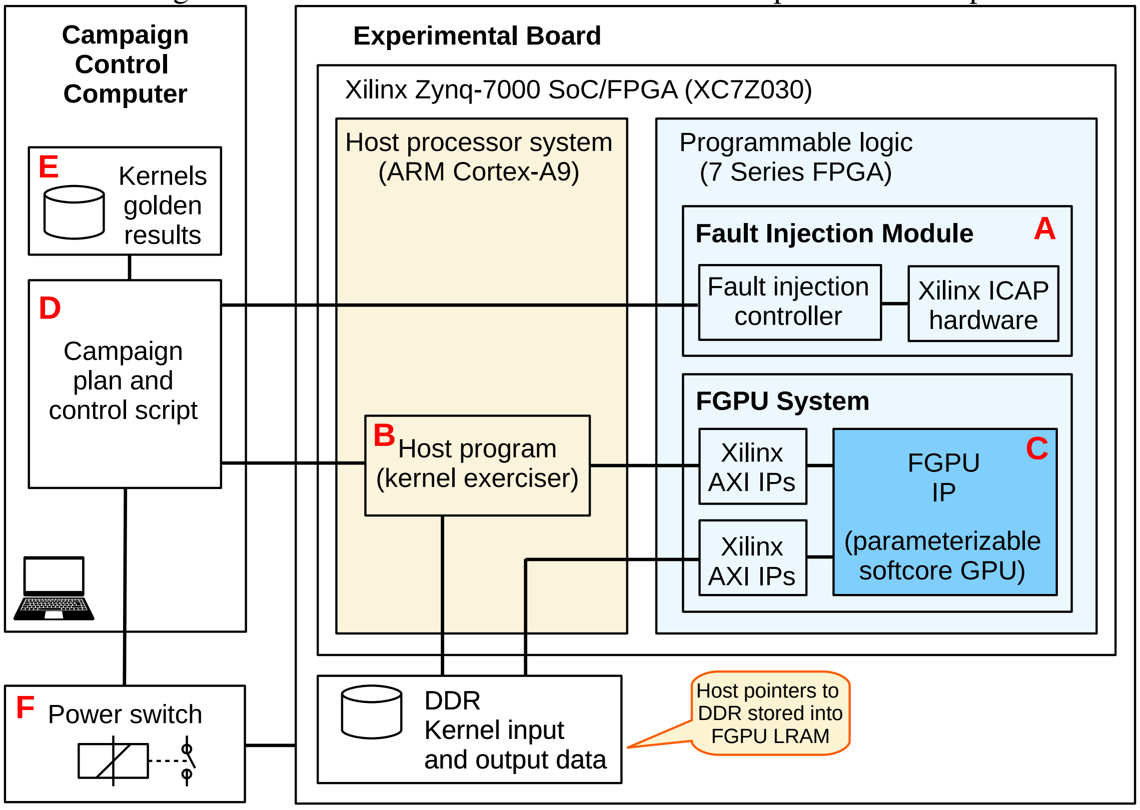
8.1 Methodology for Reliability Evaluation and Hardware Emulation

8.1.1 Reliability Assessment by Hardware Emulation

Our fault injection experimental setup aims to evaluate the impact of radiation-induced effects on the FGPU. We used the FPGA Fault Emulator presented in Section 3.4.3, which performs an injection methodology that applies faults directly to the FPGA's bitstream. Since faults might be masked or affect parts of the bitstream where no logic is implemented, we allow these faults to accumulate while running each of the applications multiple times. We then collect a reliability profile, monitoring the average number of faults required for an error to be observed.

An overview of our toolflow is depicted in Figure 8.1. Random bit-flips are injected into the configuration memory using the Xilinx Internal Configuration Access Port

Figure 8.1: Radiation-induced SEU emulation experimental setup.


Campaign Control Computer
E
Kernels golden results
D
Campaign plan and control script
F
Power switch
Experimental Board
Xilinx Zynq-7000 SoC/FPGA (XC7Z030)
Host processor system (ARM Cortex-A9)
B
Host program (kernel exerciser)
Programmable logic (7 Series FPGA)
Fault Injection Module
A
Fault injection controller
Xilinx ICAP hardware
FGPU System
Xilinx AXI IPs
Xilinx AXI IPs
C
FGPU IP
(parameterizable softcore GPU)
DDR Kernel input and output data
Host pointers to DDR stored into FGPU LRAM

Source: The Author.

(ICAP) (A) while continuously running the host application on the Cortex-A9 processor (B) and the OpenCL kernel on the FGPU (C). The control script monitors the processing time and the outputs of the FGPU. When it detects a functional error due to a timeout (D) or an output error (E), it power-cycles the FPGA board (F). The process is then repeated until the desired number of failure events is collected (i.e., at least 300 for this work). We classify events as SDC in case of a mismatch between the kernel's resulting data and the golden one. Otherwise, we classify the event as a Timeout, either because the processing time exceeded the expected execution time by over 10% or the system crashed.

We uniformly chose a random bit address for fault injection over the whole region of interest. The minimum interval between injected faults was selected to be larger than twice each application's execution time. According to the campaign planning, the control script (D) controls the bit-flip position in the FPGA's configuration memory and the fault injection timing. Each fault injection campaign is conducted separately, with a new campaign initiated for each bitstream-kernel pair.

8.1.2 Experimental Board

Our experimental board is the commercial SoC FPGA board TE0715 from Trenz Electronic, equipped with a Xilinx FPGA SoC Zynq-7000 XC7Z030, 32 MB Flash memory for storing the FPGA bitstreams, and 512 MB DDR3 memory used for data exchange between the embedded ARM Cortex-A9 processor and the FGPU. To fit the GPU into this board, we implemented a single Compute Unit (CU) with eight Processing Elements (PEs) in two configurations: (1) hard-FP, with hardware accelerators for FP operations, and (2) soft-FP, where FP instructions are emulated by software. Thus, the FGPU configurations with and without floating-point support are not implemented together at the FPGA, and different FPGA designs and bitstream files were generated for each engine configuration. The Zynq-7000 model used in these experiments includes an SRAM-based FPGA equivalent to the Xilinx Artix®-7 family manufactured in 28 nm planar CMOS technology.

8.2 Floating-Point Implementations Reliability

This section evaluates the trade-offs between hard-FP and soft-FP implementations in the FGPU embedded into an SRAM-based FPGA. We conducted a bitstream fault-injection campaign on seven case-study applications and calculated their Mean Workload Between Failure (MWBF).

For our case-studies, we selected a benchmark comprising seven kernels, described in Table 8.1. These kernels were compiled to FGPU's ISA with either soft-FP or hard-FP. Table 8.2 shows the execution times, highlighting that hard-FP can speed up execution by factors ranging from 1.21 to 19.68, depending on the computation offload.

Table 8.3 presents implementation costs for soft-FP operations. Soft-FP requires up to 354 static instructions and 16 basic blocks in software to emulate a single FP operation. Unlike hard FP, which deterministically increases area resources, soft FP has a varying execution time. Area resources in FPGAs are quantified by various metrics such as Look-Up Table (LUT), Flip-Flop (FF), and others, collectively termed as essential bits. The implementation cost in area resources for both soft- and hard-FP operations is detailed in Table 8.4.

Table 8.1: Case-study applications

App	Description	FP operations	Workload
bitonic	Bitonic sort	$<$	32k
vec_add	Sum two vectors	$+$	512k
vec_mul	Multiply two vectors	$*$	512k
fft	Fast Fourier Transform	$+,-,*$	128k
div	Divide by a scalar	$/$	256k
matrix_mult	Multiply two matrices	$+,*$	32k
cross_corr	Slide dot product of two arrays	$+,*$	1k

Table 8.2: Execution time @50MHz (μs)

Application	Soft-FP Implementation	Hard-FP Implementation	Speedup
bitonic	343,470	284,974	1.21×
vec_add	60,263	35,663	1.69×
vec_mul	77,160	35,671	2.16×
fft	691,189	350,791	1.97×
div	150,499	21,245	7.08×
matrix_mult	1,126,341	114,018	9.88×
cross_corr	868,513	44,134	19.68×

Table 8.3: Soft-FP implementation costs

FP Instruction	Symbol	Instruction count	Software basic blocks
addition	$+$	252	16
subtraction	$-$	252	16
multiplication	$*$	250	15
division	$/$	354	15
less than comparison	$<$	28	1
greater than comparison	$>$	27	1
float to int conversion	$ftoi$	41	4
int to float conversion	$itof$	76	4

Table 8.4: FGPU resources with Soft- and Hard-FP

Resource	Soft-FP Implementation	Hard-FP Implementation	Variation
LUT	29,459	40,960	1.40×
LUTRAM	723	1,833	2.54×
FF	35,588	53,178	1.49×
BRAM	56	56	–
CARRY	913	2,929	3.21×
DSP MULT	32	64	2.00×
Essential bits	7,397,007	9,608,657	1.30×

8.2.1 Reliability Evaluation Results

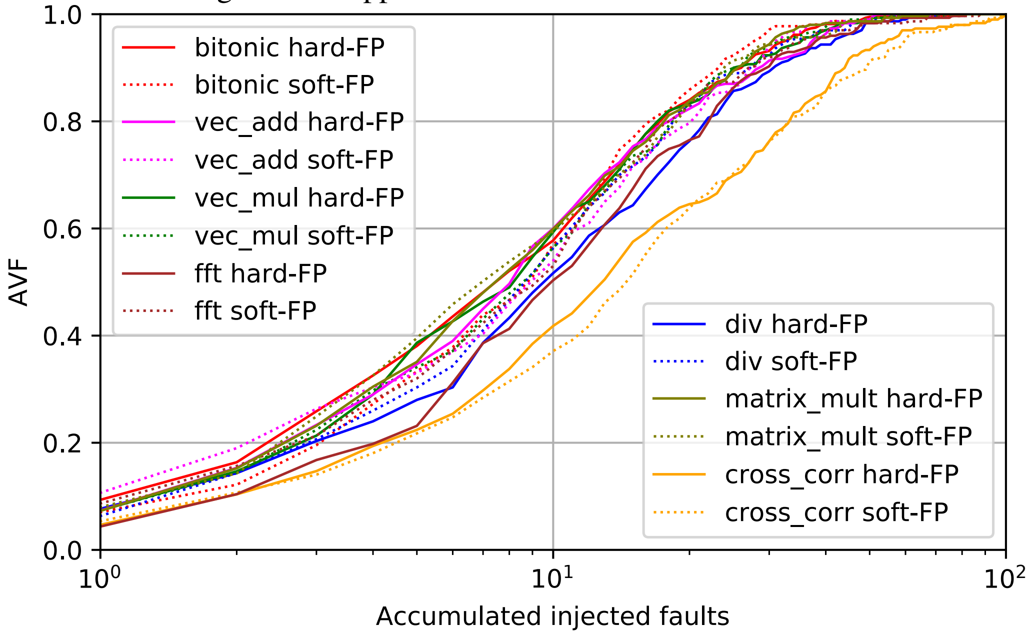
Many metrics measure the reliability of an FPGA and its implemented system. One usual metric to evaluate the susceptibility of a device under radiation is the SEU cross-section, which can be static or dynamic (RECH et al., 2014). The static SEU cross section in SRAM-based FPGAs shows the probability of an ionizing particle causing an SEU in the FPGA's configuration memory. This metric is particular to the technology used in the FPGA's integrated circuit and independent of the design running on the FPGA. The dynamic SEU cross-section shows the probability of an SEU in the FPGA's configuration memory causing an error in the design's execution. It takes into account the number of configuration bits used by the design and the logic-masking effect of the system. Usually, the more configuration bits are used by a design implementation, the higher the dynamic SEU cross-section. An option for estimating dynamic SEU cross section is to calculate AVF, as they are proportional to each other (Velazco; Foucard; Peronnard, 2010).

The AVF is also known as the error rate or the probability of a fault causing an error. On FPGAs, the AVF must be calculated considering fault accumulation, as we are considering the configuration memory bits, which should be written a single time. Therefore, we have used the experimental setup described in Section 8.1 to evaluate AVF for all case-study applications.

Figure 8.2 shows the measured AVF for all applications. Data show that the average AVF for a single fault is 0.07, increasing to 1.0 with less than 100 accumulated faults. One can also notice that the same application AVFs for soft-FP and hard-FP grow closely together, with a difference of less than 0.09. This pattern can be explained by the fact that the 29.9% increase in resource usage needed from the hard FP is not as sensitive to radiation-induced faults as shared resources, such as control registers, dispatchers, global memory controllers, and AXI buses.

The main issue with solely using cross-section as a metric is that it needs to consider the application's execution time. Therefore, two applications that process different amounts of data in the same time window could present the same dynamic cross-section, even though one can compute a more significant workload. To better compare different implementations regarding reliability, we focus our results on the Mean Workload Between Failure (MWBF) metric, introduced in Section 3.2.1.

Figure 8.2: Application AVF for accumulated faults.



Source: The Author.

The MWBF also considers the dynamic SEU cross-section by multiplying the static SEU cross-section by the measured AVF. Still, it weighs in the application's workload and execution time. Thus, MWBF is defined in (8.1), where the workload (bit) in one execution is divided by the static cross-section σ_{static} (cm^2/bit), the AVF ($bit_{critical}/bit_{injected}$), the flux ($particle/cm^2/s$), and execution time t_{exec} (s).

$$MWBF = \frac{workload}{\sigma_{static} * AVF * flux * t_{exec}} \tag{8.1}$$

To calculate our case-study applications' MWBF, we need to consider that our applications run at ground level and use a 28 nm Kintex-7 FPGA. It means that neutrons are the main source of radiation-induced faults, with a flux of 13 $neutrons/cm^2/s$ (Normand, 1996) and that our static SEU cross section is 5.69×10^{-15} cm^2/bit (XILINX, 2019). The application's workload can be seen in Table 8.1, execution time in Table 8.2, and AVF in Figure 8.2 (CARMICHAEL, 2006).

Table 8.5 shows the calculated MWBF for all case-study applications running with soft-FP and hard-FP implementations. Although data can be miss-leading when comparing MWBF between different applications, the relative improvement in MWBF due to the use of hard-FP for the same applications (also shown in Figure 8.3) is very interesting. It shows a strong correlation between MWBF and the execution time speedup achieved by the hard-FP implementation (Table 8.2). This correlation is mostly because the AVF for soft-FP and hard-FP grew closely together with fault accumulation (Figure 8.2).

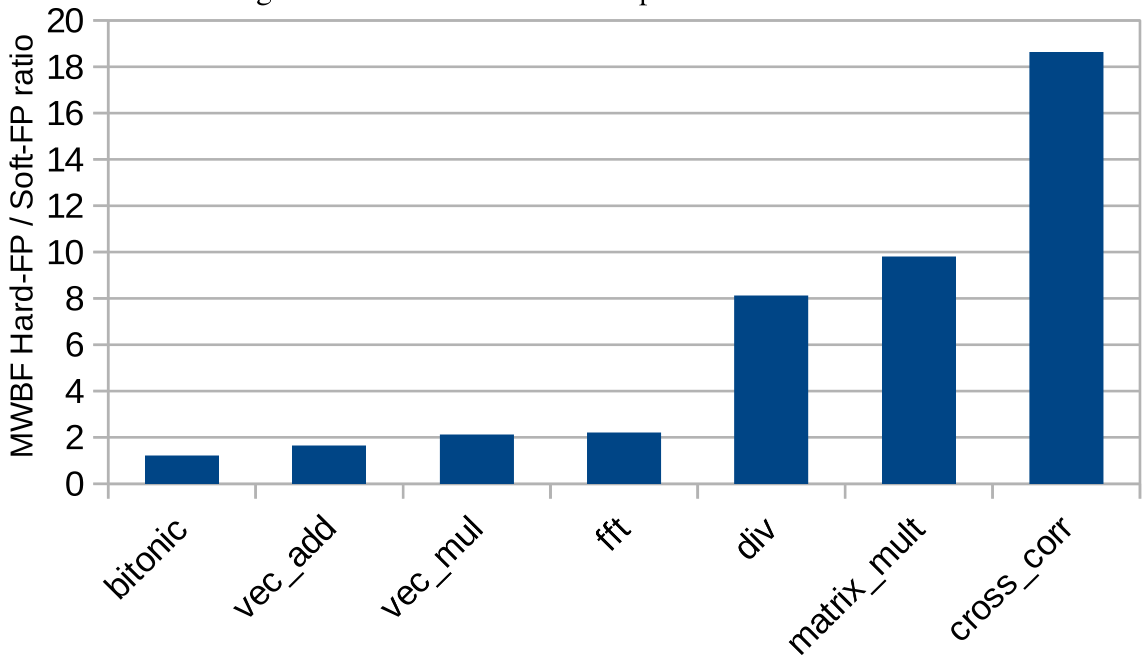
Table 8.5: Application MWBF (bit)

Application	Soft-FP Implementation	Hard-FP Implementation	Variation
bitonic	1,83E+15	2,22E+15	1.21×
vec_add	1,81E+17	2,99E+17	1.65×
vec_mul	1,37E+17	2,93E+17	2.14×
fft	4,05E+15	8,91E+15	2.20×
div	3,64E+16	2,96E+17	8.13×
matrix_mult	5,66E+14	5,55E+15	9.81×
cross_corr	4,20E+13	7,82E+14	18.62×

8.3 Selective Triple Modular Redundancy

This section evaluates the impact of applying selective Triple Modular Redundancy (TMR) to the FGPU when deployed on our experimental board under radiation-induced faults. We employ specific floorplan constraints to focus fault injection on targeted areas within the FGPU architecture. Multiple FGPU bitstreams are generated, each assigning specific architectural modules to the designated floorplan area for fault injection. Subsequently, fault injection campaigns are conducted on the configuration memory, targeting three case-study applications.

Figure 8.3: Hard-FP/Soft-FP improvement in MWBF.



Source: The Author.

TMR is then applied to each of these modules, and its impact on system reliability is assessed. The individual TMR reliability curves are combined with resource usage data to group modules into different sets of hardened modules. These sets are then compared with the original, unhardened system to evaluate the effectiveness of the TMR strategy in enhancing system reliability.

8.3.1 Methodology for Fault Injection in FGPU Modules

We selected three different microbenchmark applications—bitonic sort (bitonic), Fast Fourier Transform (FFT), and cross-correlation (xcorr)—to evaluate the FGPU's reliability. These applications have shown varying levels of program vulnerabilities in our previous study. Bitonic is the most sensitive regarding reliability metrics, followed by FFT and xcorr. These results are expected, as each microbenchmark utilizes different combinations of the FGPU's instruction set and modules. Table 8.6 summarizes these applications, detailing the floating-point instructions used and execution time.

The input data size for all three applications was kept consistent at 2,048 single-precision floating-point elements (IEEE 754™ binary32). The organization of these elements varies by application, as detailed in the text.

Those input data elements are organized differently for each application, being a single vector of 2,048 elements in the bitonic, two vectors of 1,024 elements in the xcorr, and two vectors of 512 complex elements ($a+bi$) in the FFT. For the bitonic and the FFT kernels, the resulting output is stored inside the same vectors (memory address) of the

Table 8.6: Case-study applications

Application	Description	FP Instr.	Kernel Exec. (*ms*)
bitonic	Bitonic sort	$<$	7.9
FFT	Fast Fourier transform	$+,-,\times$	1.7
xcorr	Cross-correlation	$+,\times$	171.9

input data. In contrast, for the xcorr kernel, a third vector with the same size as the input vectors is allocated to hold the output result. Each application uses a different OpenCL NDRange dimensionality and size at the kernel.

As our objective goes beyond investigating the reliability of the entire FGPU, we split its architectural modules into groups. We created five groups of architectural modules: (i) AXI, which includes data and control for the AMBA AXI4 interconnection used to interface the FGPU with the Cortex-A9 and DDR memory, (ii) FGPU Control, which consists of the control registers, WG dispatcher, and memory structures, (iii) CU Control, which includes the CU WF scheduler, runtime memory, and memory controller, (iv) INT Unit, which provides integer operations and register files, and (v) FP Unit, which consists of the floating-point unit. Figure 8.5 shows each of these groups using different colors, and Table 8.7 presents the resource usage by each group in terms of LUTs, FFs, BRAMs, and DSPs.

Table 8.7: FGPU resources usage

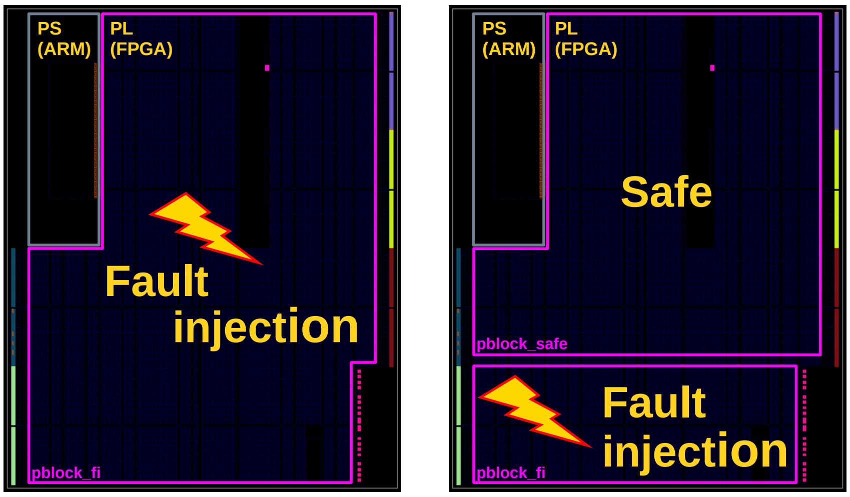
Module	Look-up Tables (LUTs)	Flip-Flops (FFs)	BRAMs	DSPs
AXI	950	1,491	—	—
FGPU Control	18,006	14,022	13	—
CU Control	5,146	8,123	27	—
INT Unit	5,823	12,314	16	32
FP Unit	10,948	17,292	—	32

To physically inject faults in each individual group of modules and identify which are the ones more susceptible to provoke failures, we created two FPGA floorplans, constraining or not the area under test. The first floorplan, shown in Figure 8.4(a) and discussed in Section 8.3.1.1, uses the entire FPGA as the area under test. The second, shown in Figure 8.4(b) and discussed in Section 8.3.1.2, uses an isolated area as an area under test, keeping the rest of the FPGA fabric untouched by faults. Doing so allows us to individually constrain modules to the area under test through floorplan placement.

8.3.1.1 Global Fault Injection Floorplanning

The global fault injection floorplan targets the whole FPGA except for small regions used for the fault injection logic and its communication pins as the area under test.

Figure 8.4: Floorplan setup for fault injection.

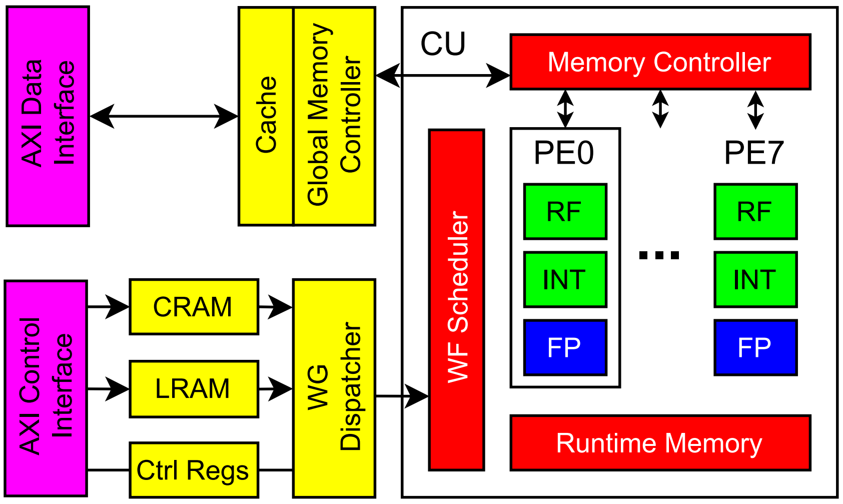


(a) Global fault injection (b) Isolated fault injection

Source: The Author.

In this floorplanning, faults can affect any region of the deployed FGPU, being the best option to evaluate the impacts of radiation-induced faults in the complete running system. Therefore, it gives us more realistic data when considering our applications running in harsh environments. However, it comes at the cost of less observability of how each part of the FGPU contributes to its reliability. Figure 8.6(a) shows an example of the entire FGPU placed in this global fault injection floorplan with its modules colored according to Figure 8.5.

Figure 8.5: Softcore FGPU colored block diagram.

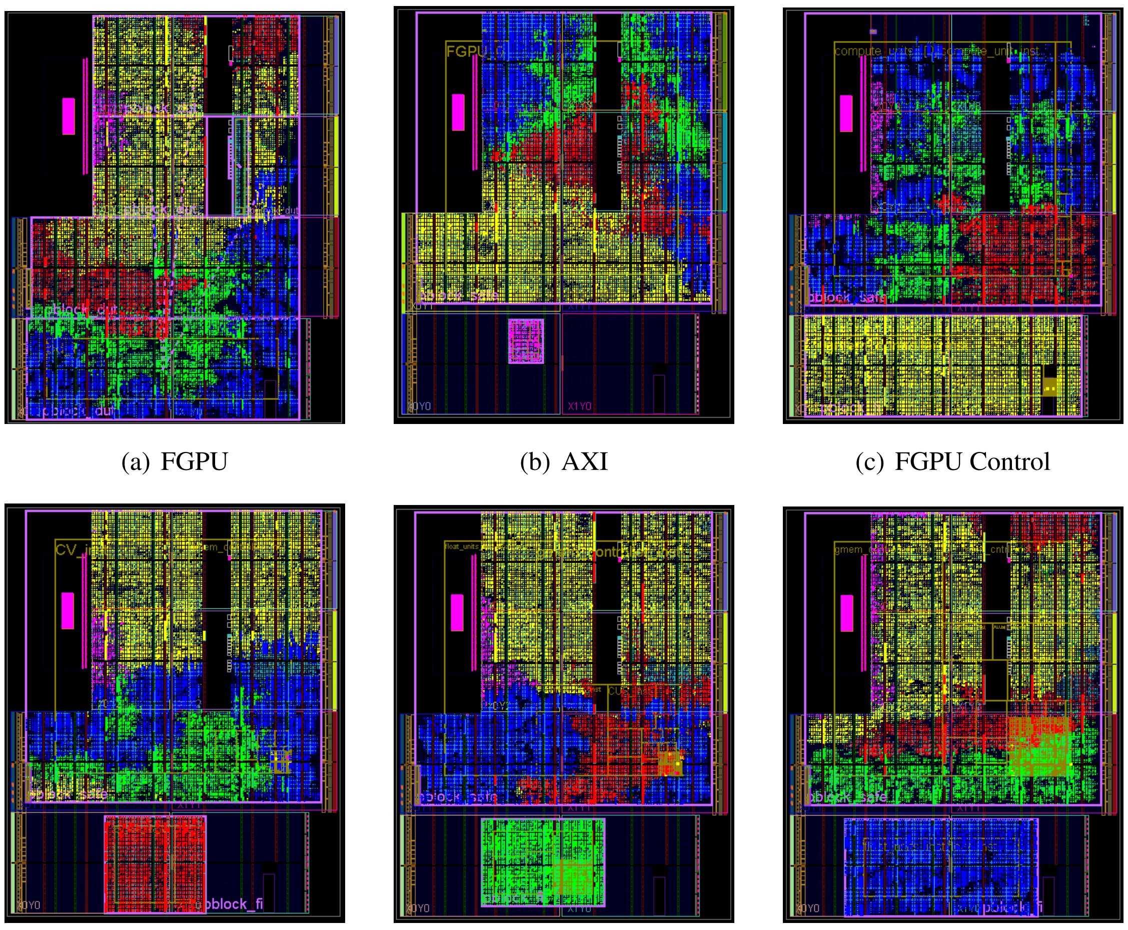


Source: The Author.

8.3.1.2 Isolated fault injection floorplanning

The isolated fault injection floor plan was designed to contain only a fraction of FGPU in the area under test. It separates the area under test from the rest of the FGPU,

Figure 8.6: Floorplans of entire FGPU and isolating individual modules for the bitstream fault injection campaign.



(a) FGPU (b) AXI (c) FGPU Control

(d) CU Control (e) INT Unit (f) FP Unit

Source: The Author.

thus allowing us to inject faults precisely in individual groups of modules. This approach shows us how each part of the FGPU contributes to its overall reliability. The floorplan placement strategy, shown in Figure 8.4, consisted of partitioning the FPGA into safe and fault injection regions. To do so, we reserved an area in the FPGA's fabric that could hold the largest of the chosen groups of modules, leaving the rest of the FPGA's fabric in the safe region. Knowing the mapping between these floorplan regions and the bit addresses in configuration memory allowed us to restrict the bit-flip injection to these regions of interest and emulate radiation effects only in the selected FGPU modules.

We synthesized FGPU five times to adapt it to our placement strategy, one for each isolated group of modules in the area under test. Multiple placement constraints were used for each group of modules to keep a similar fabric usage density. Figs. 8.6b–f show each group of modules isolated according to the experimental floorplan on Figure 8.4(b), also colored according to Figure 8.5, where we can compare the relative area occupancy of each modules group in the FPGA's fabric with the resources usage in Table 8.7.

8.3.2 FGPU Reliability

This Section discusses FGPU's reliability. We collected at least 300 failure events for the entire FGPU (considering the global fault injection floorplan) and for each of the five isolated groups of modules (considering the isolated fault injection floorplan) for all case-study applications. We show the effects of radiation-induced faults on the entire unhardened FGPU and each unhardened module separately. We start by drawing the application's reliability as the probability of tolerating accumulated faults in the configuration memory over time without experiencing a functional failure (SDC or Timeout).

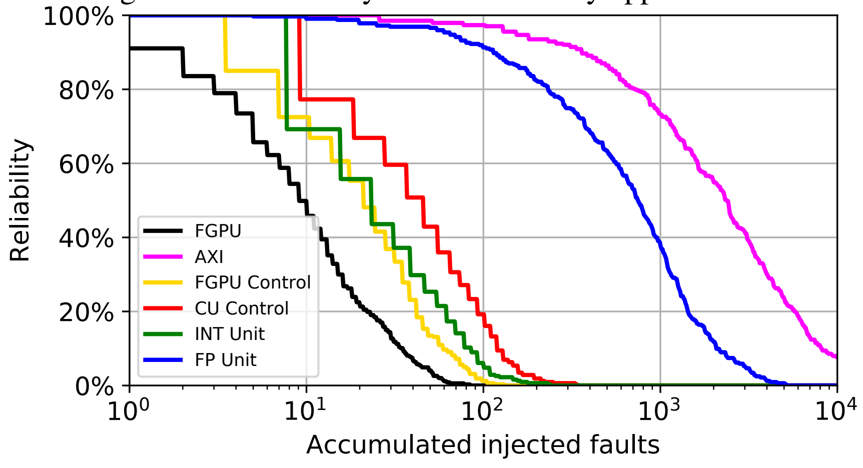
The reliability curves can be seen in Figure 8.7. It shows, in black, the reliability for the FGPU using the whole FPGA's fabric (Figure 8.4(a)) and, in colors, the isolated groups' reliability (Figure 8.4(b)) - i.e., the probability of functional failures occurring when a fault hits one of these groups. One can notice that the AXI is the most reliable group. Even though the host constantly performs pooling requests on the AXI modules, it takes over 100 accumulated faults for all applications to drop reliability below 90%. It happens mainly due to its small resource usage and masking effects. The FP Unit is the second most reliable group, on average. The FP Unit becomes less reliable for the FFT and xcorr applications, closing in the INT Unit. It happens mainly because the bitonic uses only *relational less than* FP instructions, which is the smallest module in the FP Unit, while FFT and xcorr depend heavily on *add/sub* and *multiplication* instructions. Overall, the remaining groups have similar behavior, with the FGPU Control being the most sensitive group.

Figure 8.8 shows the Mean Faults to Failure (MFTF) in the FGPU to better compare how the faults affect its different components when running different applications. In other words, this figure shows the mean number of accumulated faults required to produce a functional failure. As one can notice, there is no clear trend, showing that applications used different groups of modules more heavily than others. For example, the xcorr is the most reliable when considering the INT Unit but the less reliable when considering the FP Unit. The FFT is the most reliable application when considering the CU Control but the least reliable when considering the FGPU Control. These data show that engineers must consider the application when designing hardening techniques for FGPU. Note that the MFTF becomes less important when we apply TMR, as it worsens the curve with many accumulated faults (CARMICHAEL, 2006).

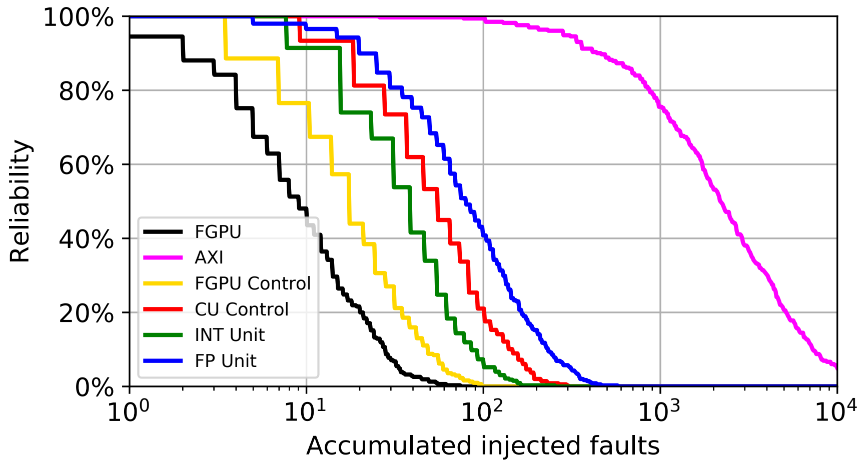
8.3.3 Selective TMR FGPU Reliability

In our and many real-life scenarios, the unhardened FGPU occupies a huge amount of our target device's resources (70% of the configurable logic slices). Due to that, implementing full-scale TMR hardening is not a feasible option for improving reliability. Alternatively, selectively applying TMR to the most sensitive module groups becomes an

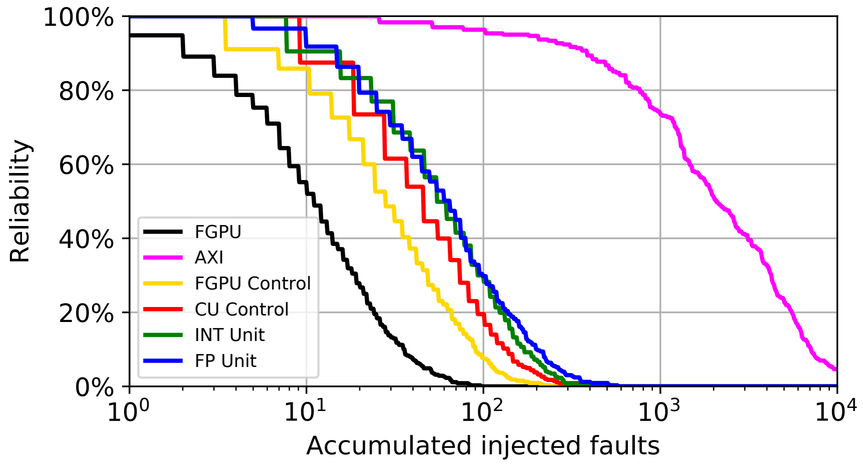
Figure 8.7: Reliability of the case-study applications.



(a) Bitonic



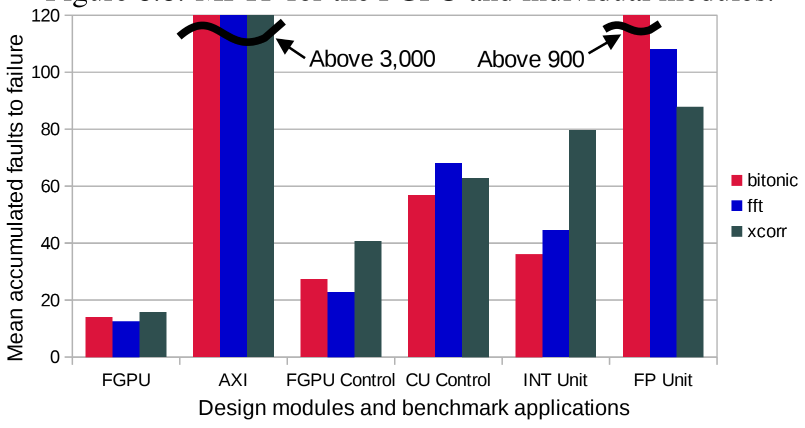
(b) FFT



(c) Cross correlation

Source: The Author.

Figure 8.8: MFTF for the FGPU and individual modules.



Source: The Author.

interesting solution. Notice that this area constraint is always an issue for a configurable scalable architecture because larger FPGA devices will have more Compute Units (CU). In this Section, we use our FPGA board resources as our global constraint and discuss the trade-offs in reliability, area, and power of applying selective TMR hardening on the FGPU.

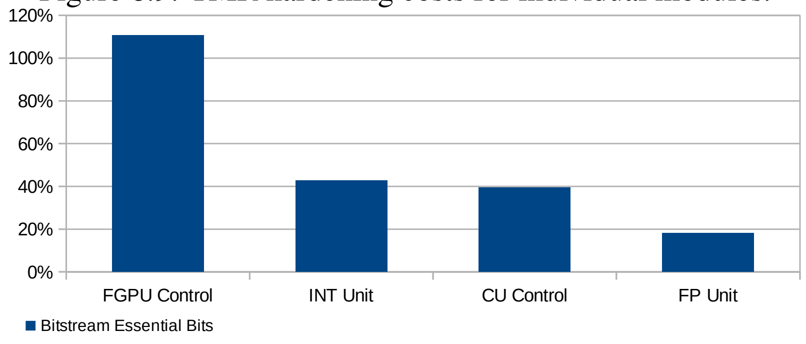
Considering the size and reliability of the AXI in the previous experiment, we removed it from our analysis. For the remaining four modules (FGPU Control, CU Control, INT Unit, and FP Unit), we implemented TMR versions and replaced them, one at a time, in the FGPU architecture.

The overheads of hardening each individual module group can be seen in Figure 8.9. We analyze the overheads regarding bitstream essential bits, estimated power, and resource usage. The number of essential bits reflects the configuration memory size susceptible to bit flips for a given module. The fault rate decrease must be higher than the sensitive area increase to improve reliability over time. Thus, the increase of the essential bits has to be minimal. Estimated power is not directly related to reliability. Still, it is a common constraint in aerospace applications and may limit design space. Resource usage is a necessary metric for LUTs, FFs, BRAMs, and DSPs.

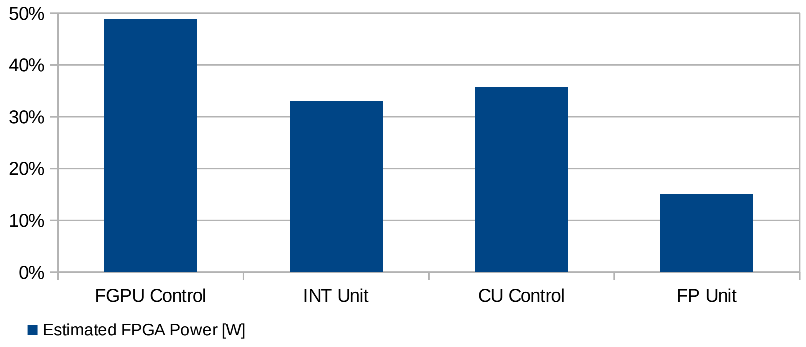
Resource usage obtained during TMR synthesis shows that modules differ in resource usage. While FGPU Control increases the system's LUT usage by over 100%, it requires a smaller amount of FFs and BRAM and does not require DSPs. On the other hand, the FP Unit requires a small amount of LUTs and FFs but a considerable amount of DSPs. Given our FPGA board resource constraints, the TMR version of the FGPU Control cannot be combined with any other TMR module due to the lack of more LUTs. However, the other modules can deploy their TMR versions in the same design.

In the following, we discuss the reliability impacts of applying TMR to a single module and multiple modules. For both evaluations, we performed fault injection campaigns following the same methodology used in Section 8.1 according to the experimental setup shown in Figure 8.4(a), where we inject faults over the entire design.

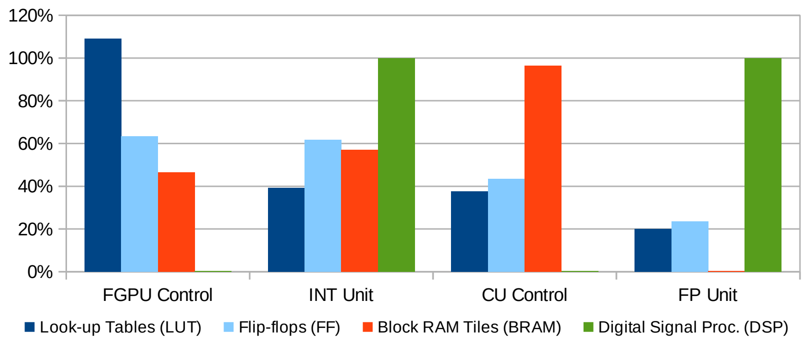
Figure 8.9: TMR hardening costs for individual modules.



(a) Essential bits



(b) Estimated power



(c) Resource usage

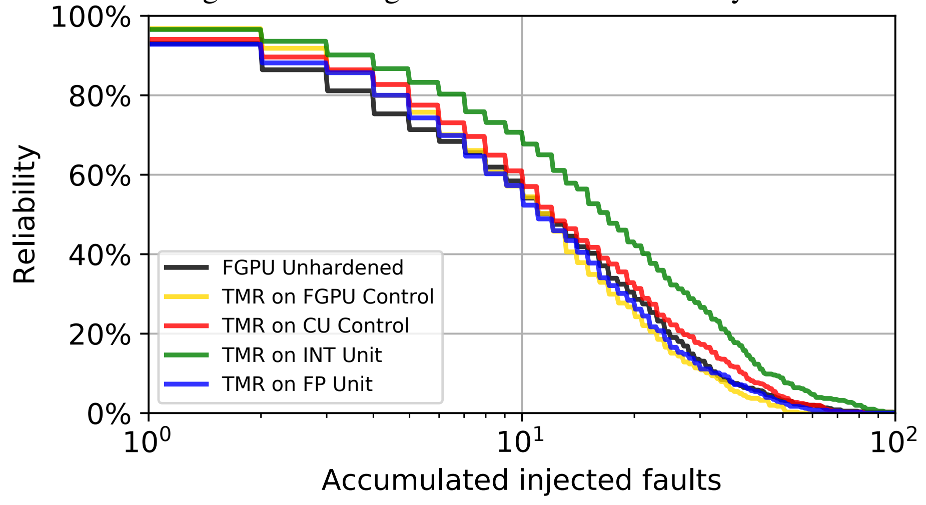
Source: The Author.

8.3.3.1 Single Module TMR

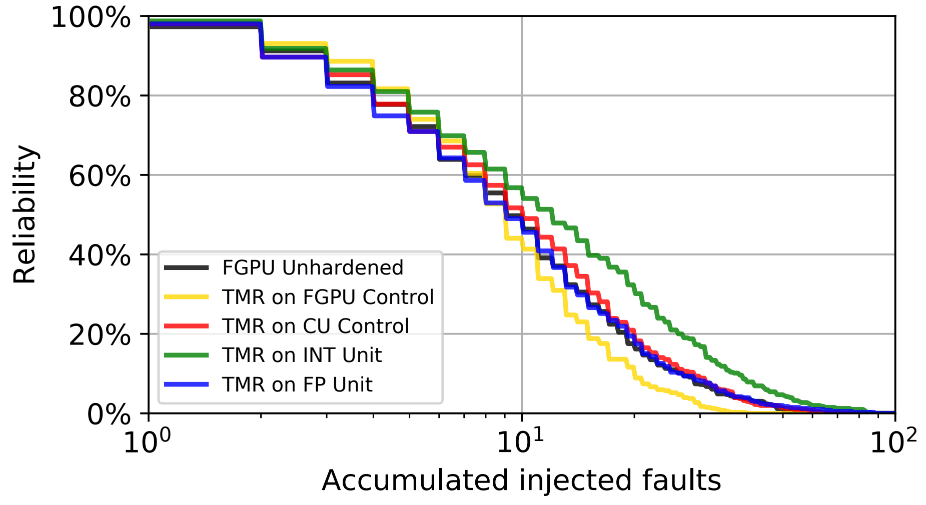
Single module TMR evaluates how effectively we can move FGPU's unhardened curve (Figure 8.7 in black) to the upper right corner so that more faults can affect the bitstream without leading to failures (thus increasing the MFTF). To do so, we implemented four FGPU versions, each with a single module (FGPU Control, CU Control, INT Unit, and FP Unit) hardened with TMR. Figure 8.10 shows the reliability curves for the FGPU unhardened, which serves as a baseline, and for each TMR hardened module for all case-study applications.

The bitonic application showed the best improvement through the INT Unit. This kernel uses only the relational less than floating-point operation, relying mostly on data movement, control, and logical operations from the INT Unit. Hence, the INT Unit

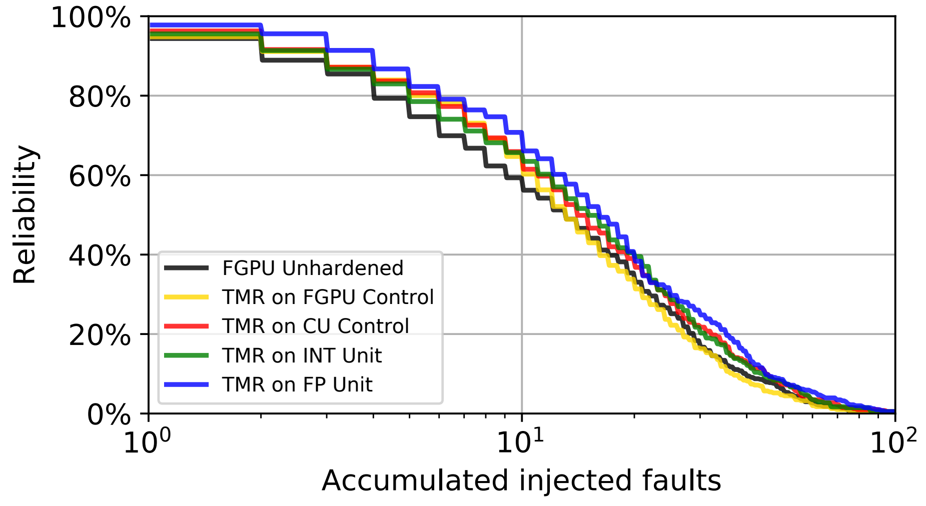
Figure 8.10: Single module TMR Reliability.



(a) Bitonic



(b) FFT



(c) Cross correlation

Source: The Author.

showed a clear advantage compared to the other modules.

Despite having inherent floating-point processing that uses addition, subtraction, and multiplication, the FFT kernel improved the most with the FGPU Control and INT Unit mitigation. The FGPU Control provides better reliability until the reliability drops down to 80%, then the INT Unit takes the lead. This trend happens due to the impact that the TMR replication causes on larger modules, becoming ineffective with fault accumulation.

The xcorr application showed small improvements for all hardened modules except the FP Unit. This kernel spends the most execution time running addition and multiplication floating-point operations. Hence, the FP Unit hardening showed a clear advantage over the others.

8.3.3.2 Multiple Module TMR

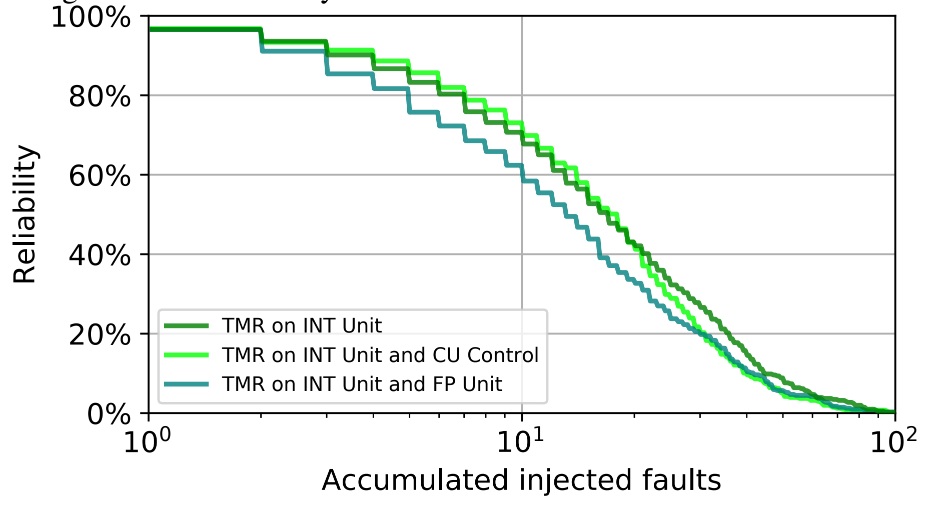
Multiple module TMR aims to improve single module FGPU's reliability curve by combining multiple TMR hardened modules. We took the best candidates from the previous Section as a baseline and combined them with other modules. Figure 8.11 shows the reliability curves for multiple module TMR against single module TMR.

The bitonic application uses the INT Unit as a baseline, which is then combined with the FP Unit or CU Control. Combined with the CU Control, it shows improvements up to around 40% reliability. However, reliability worsens the baseline curve when combined with the FP Unit. According to the previous evaluation (Figure 8.10(a)), this happens due to the FP Unit's additional redundancy weight that adds little to FGPU's reliability.

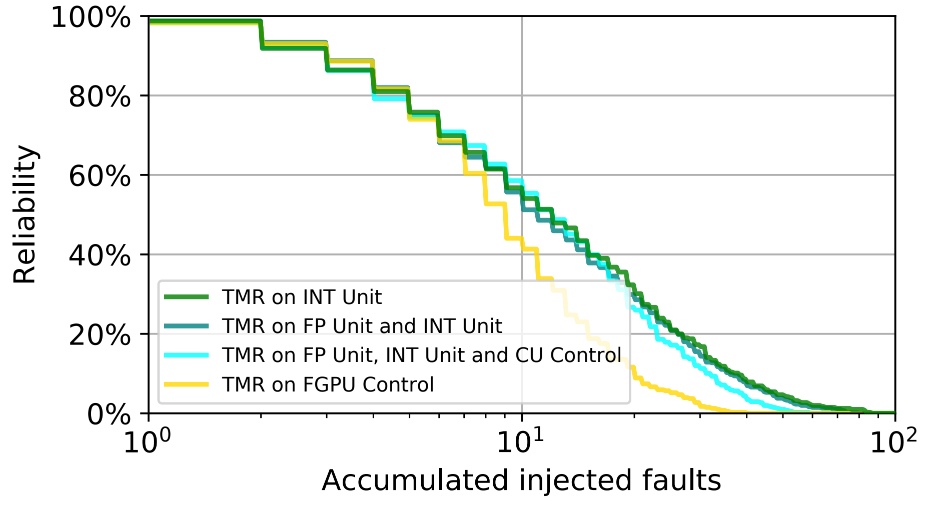
The FFT kernel has two baselines, INT Unit and FGPU Control. Because the FGPU Control cannot be grouped with other modules, we evaluated the INT Unit combined with the FP Unit and combined with the FP Unit and the CU Control. The combinations improve both baselines down to 80% reliability. From then on, they presented better results than FGPU Control but were close to the INT Unit. This is an interesting example of setting a reliability level and varying hardening options in resource usage and power.

The xcorr has the FP Unit as a baseline. Hence, we combined it with the INT Unit and then added the CU Control. The results show a scenario that resembles the bitonic application. Here, as for the bitonic kernel, we have a scenario where combinations hindered FGPU's reliability. The main reason for this trend is that, as shown in the previous evaluation (Figure 8.10(c)), the FP Unit is the only TMR that improves FGPU's reliability. Thus, the replication costs outweigh its reliability benefits.

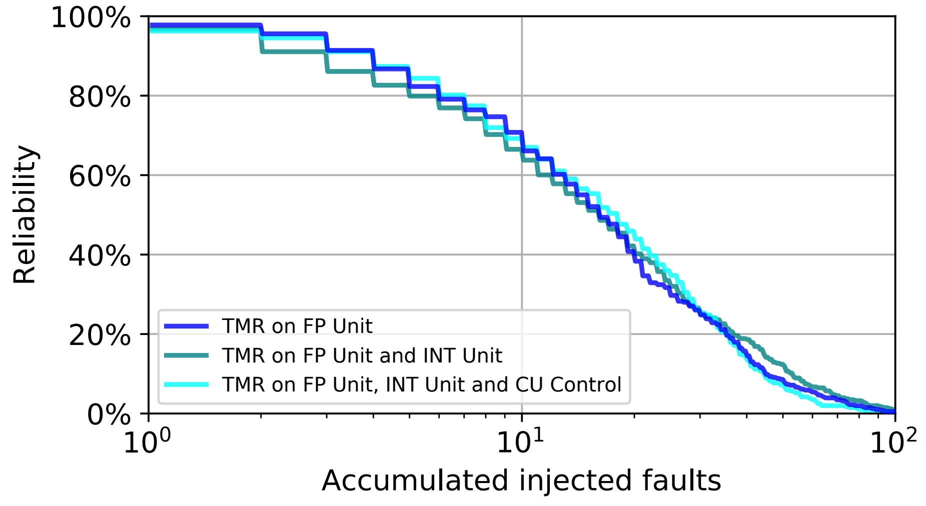
Figure 8.11: Reliability of alternative TMR modules combinations.



(a) Bitonic



(b) FFT



(c) Cross correlation

Source: The Author.

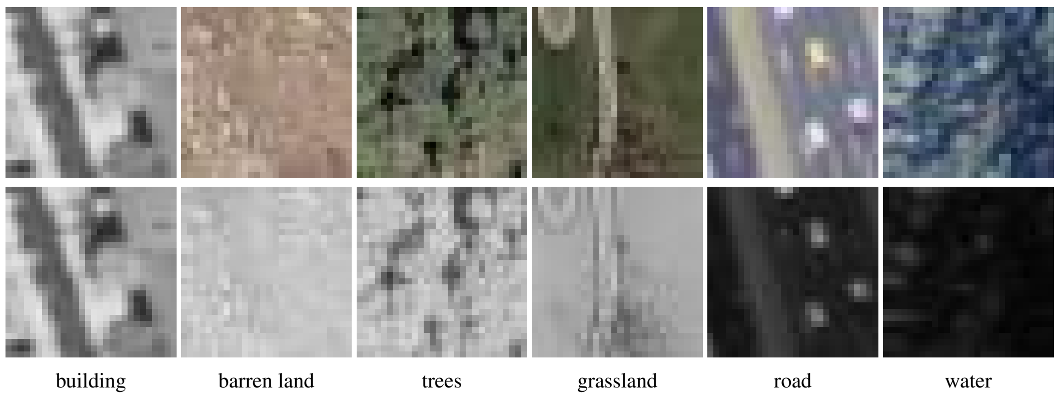
8.4 Reliability Impacts of Neutron-induced Soft Errors

This section investigates how fast and thermal neutron-induced SEUs affect the reliability of aerial image classification CNNs running on a softcore GPU implemented in an SRAM-based FPGA. We perform a configuration bitstream fault injection campaign and two neutron irradiation experiments to evaluate reliability.

8.4.1 Aerial Image Classification Benchmark

CNNs are state-of-the-art machine learning algorithms with high accuracy in solving image classification problems. They are composed of many multiply-accumulate (MACC) operations organized into different layers. CNNs contain convolutional, pooling, and, occasionally, fully connected layers. Weights and bias coefficients at the CNN are trained repetitiously from a large dataset, learning information directly from image, audio, and text without prior processing or feature extraction. In this case, the neural network learns to extract low-level features from the input data. We chose the SAT-6 dataset (BASU et al., 2015) as a study case. It consists of RGB and NIR band images representing 1 meter per pixel of continental United States from the U.S. National Agriculture Imagery Program. The SAT-6 dataset was labeled for the land cover in six classes, as exemplified in Figure 8.12.

Figure 8.12: Sample RGB (top) and NIR (bottom) images from SAT-6 dataset

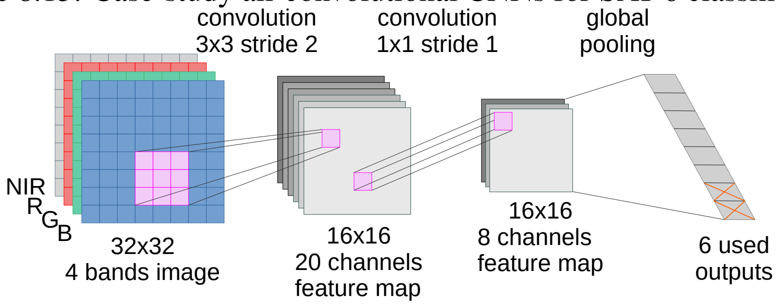


Source: (BASU et al., 2015).

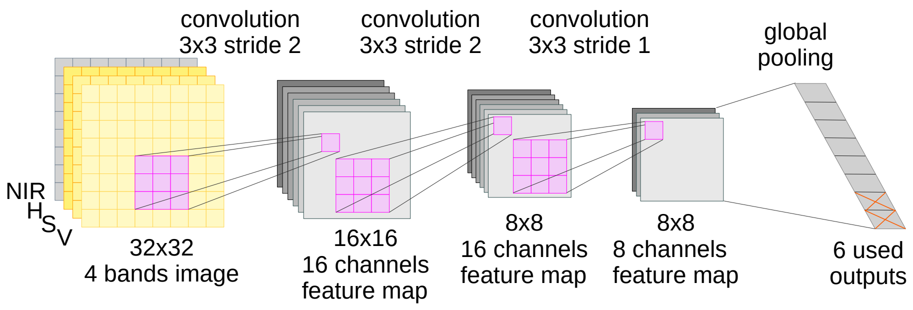
Our CNNs were designed by automatically generating a sub-optimal shallow CNN architecture using genetic algorithms (BENEVENUTI et al., 2021) to explore its geometry's design space. The design space exploration included the number and dimension layers and the resulting kernel size, all constrained by the feasibility rules of the target inference engines and the size of the target FPGA device. We then designed two CNNs, a floating- and a fixed-point one.

As depicted in Figure 8.13, the floating-point CNN design resulted in two convolutional layers with a set of 908 weights and bias coefficients. We applied a fractional fixed-point numeric representation (Q*m*.*n*) for the fixed-point CNN implementation of input images, feature maps, weights, and bias coefficients. To simplify the fixed-point multiplication, input images and feature maps were modified to have only the signal and integer parts (Q7.0). In addition, weights and bias coefficients were modified to have the signal and fractional parts (Q0.7). The multiplication and accumulator results in the convolution were implemented with integer and fractional parts (Q7.8), discarding the fractional part by rounding. Consequently, the inputs and outputs of each convolutional layer fit into an eight-bit signed integer (INT8). The final candidates reached 97.3% accuracy requiring 2.3×10^5 MACC operations for inference (floating-point) and 98.2% accuracy requiring 3.7×10^5 MACC operations for inference with 4,072 weights and bias coefficients (fixed-point).

Figure 8.13: Case-study all-convolutional CNNs for SAT-6 classification.



(a) CNN trained for floating-point



(b) CNN trained for fixed-point

Source: The Author.

8.4.2 CNN Implementation

The FGPU configurations, both with and without floating-point support, were not implemented concurrently on the FPGA. Separate FPGA designs and bitstream files were created for each specific engine configuration. The FPGA design encompasses the soft-core FGPU along with its associated AMBA™ AXI™ interconnection to the DDR memory, as well as the Arm® Cortex®-A9 microprocessor provided by the Xilinx Zynq-7000 (detailed previously in Section 8.1.2).

The set of test images and the CNN coefficients are stored in the DDR memory and are the same for both engines. The Arm Cortex-A9 is the host processor, coordinating the sequential processing of each CNN layer for every test image. Still, it is not involved in any relevant mathematical processing for the CNN. Triplication, voting, and checksum are used on test images and CNN data stored on DDR to avoid interference of SEUs on DDR with the test of the engines in FPGA.

Regarding resource usage, performance, and power, the FGPU configured without supporting hardware for floating-point operations required half the number of dedicated DSPs and a slightly lower amount of resources. In contrast, the processing time is 20× longer when computing in fixed-point arithmetics. The Xilinx Vivado synthesis tool provides rough estimates for power consumption that are very similar for both configurations of the FGPU.

8.4.3 Evaluation

Our random-accumulated fault injection methodology described in Section 8.1.1 consists of reading the FPGA's configuration memory at a random position, flipping a bit, and writing it back. This process is asynchronous to the CNN application running at the FPGA. Therefore, we can inject a fault randomly during any step of the CNN processing.

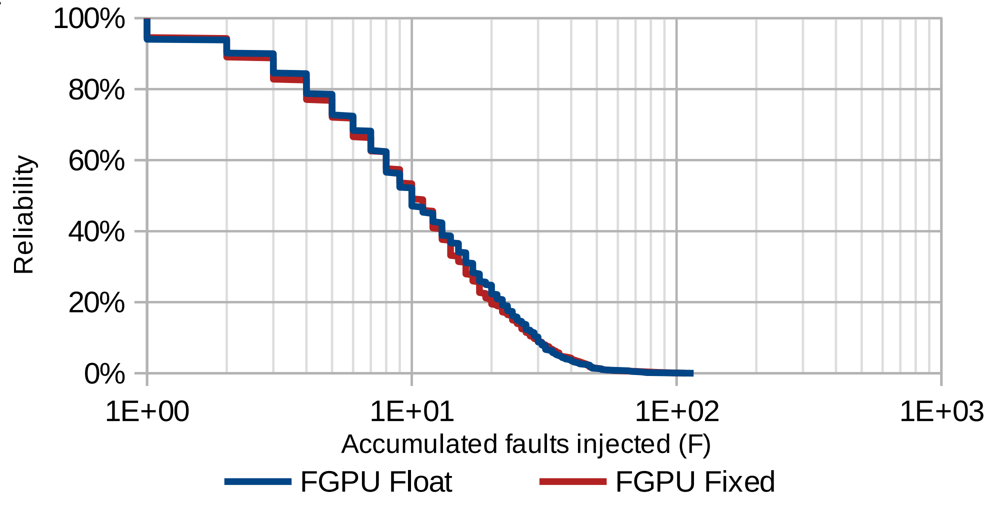
Results from emulated fault injection are used in this work in two ways. Firstly, it helps in screening candidate solutions. Different CNNs and the same inference engine with different parameters can have different FPGA footprints, processing times, and, ultimately, different reliabilities. Secondly, it is used for planning the irradiation campaign. Fault injection preliminary estimates of reliability metrics are used to prepare the radiation facility's time to test each candidate solution.

From fault injection, one can build an empirical probability distribution function for the reliability of each candidate solution, as seen in Figure 8.14(a), and estimate reliability metrics such as mean faults to failure (MFTF), mean time to failure (MTTF), mean executions between failure (MEBF) and mission time (MT). Some of these metrics are presented in Table 8.8.

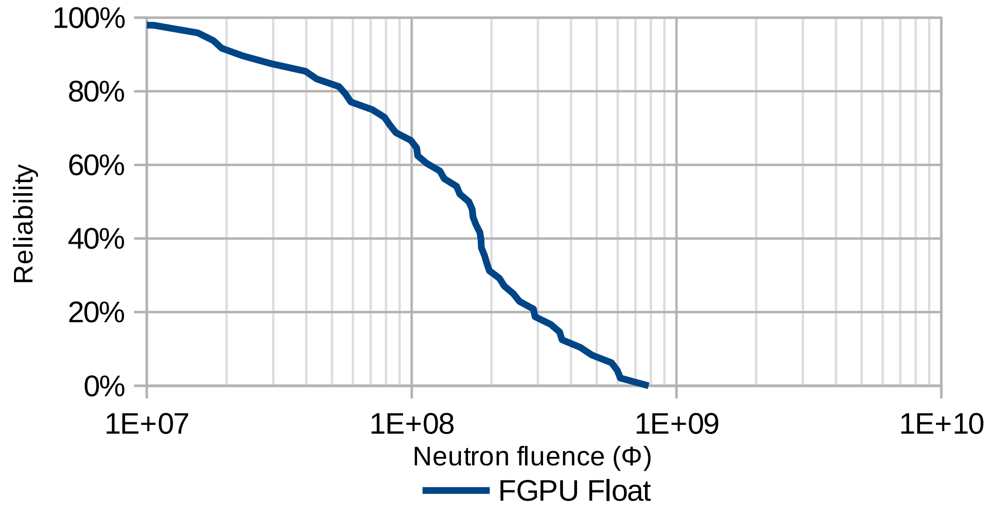
Table 8.8: Results and reliability metrics from fault injection

Metrics	Floating-point	Fixed-point
Faults injected	7,868	5,448
Failure events	569	401
Mean Faults to Failure (MFTF)	13.8	13.6
Normalized Mean Executions Between Failure (MEBF)	21.7×	1.0×

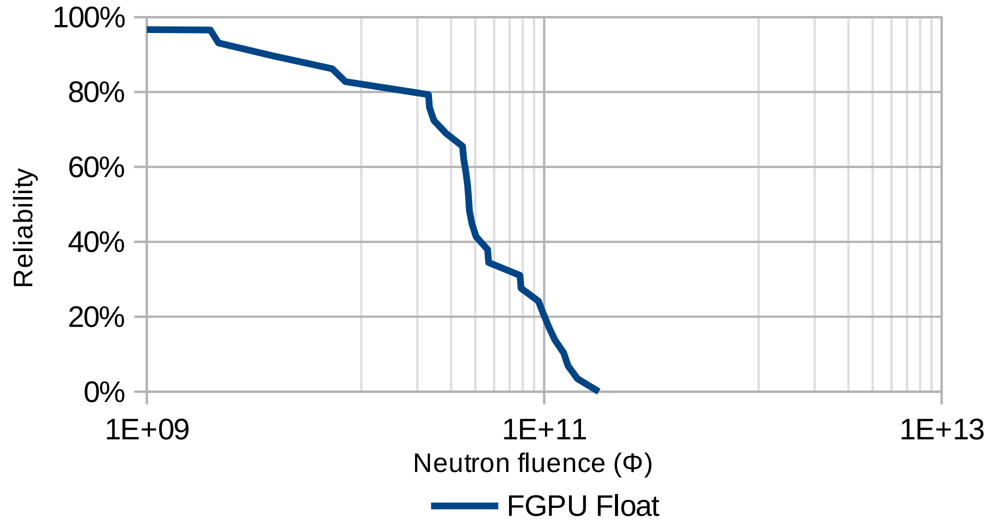
Figure 8.14: Empirical reliability curves obtained from fault injection and neutron irradiation.



(a) Fault injection



(b) 14.1 MeV neutron



(c) Thermal neutron

Source: The Author.

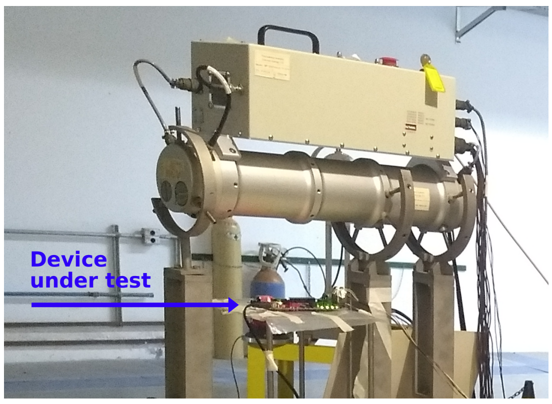
Results from fault injection (Figure8.14(a)) are consistent with previous experiments, with MFTF being lower than observed on other benchmarks due to reduced area. Given the towering increase in the CNN processing against marginal improvement in terms of power and resources usage, and the similar profile of reliability observed in the FGPU without floating-point support (Figure 8.14(a)), we excluded the CNN running in fixed-point over FGPU from the radiation tests.

The FGPU implementation operating in floating-point was tested under radiation using a deuterium-tritium (D-T) 14.1 MeV neutron generator (Figure 8.15(a)) at the facilities of the Institute for Advanced Studies (IEAv), São José dos Campos, Brazil, a research unit under the Aeronautics Science and Technology Department of the Brazilian Air Force. During the experiments, the FPGA board was protected with 0.5 mm cadmium sheet to avoid the influence of thermal neutrons.

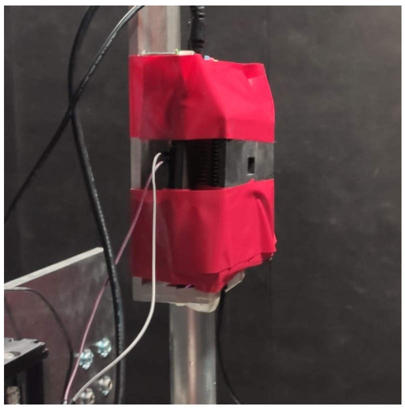
The FPGA configuration memory static cross section for 14.1 MeV neutron was measured previously at this same facility with the same setup, being in the order of $1.8 \times 10^{-15}\,\mathrm{cm^2 bit^{-1}}$ for single-bit upsets (SBU) and $0.6 \times 10^{-15}\,\mathrm{cm^2 bit^{-1}}$ for multiple-bit upsets (MBU) at frontal irradiation (0°). The static cross section was measured as $1.3 \times 10^{-15}\,\mathrm{cm^2 bit^{-1}}$ and $0.3 \times 10^{-15}\,\mathrm{cm^2 bit^{-1}}$ for SBU and MBU, respectively, at backside irradiation (180°).

The FPGA board was irradiated with 14.1 MeV neutrons at frontside (0°) for 53.6 h with an average flux of $2.79 \pm 0.08 \times 10^4\ \mathrm{s^{-1}cm^{-2}}$ at the experimental board distance of 20 cm and $5.50 \pm 0.18 \times 10^4\ \mathrm{s^{-1}cm^{-2}}$ at the distance of 15 cm. Irradiation time, total fluence, number of failure events, and mean fluence to failure ($M\Phi TF$) observed are summarized in Table 8.9. Figure 8.14(b) presents reliability curves for 14.1 MeV neutrons.

Figure 8.15: Experimental setup for neutron irradiation.



(a) Radiation facility 14.1 MeV neutron generator.



(b) Board shielding for thermal neutrons experiments.

Source: The Author.

The FGPU configured for floating-point was also tested under thermal and epithermal neutrons at the Thermal and Epi-thermal Neutron irradiation Station (TENIS) of the Institute Laue–Langevin (ILL), Grenoble, France. At the operating conditions for

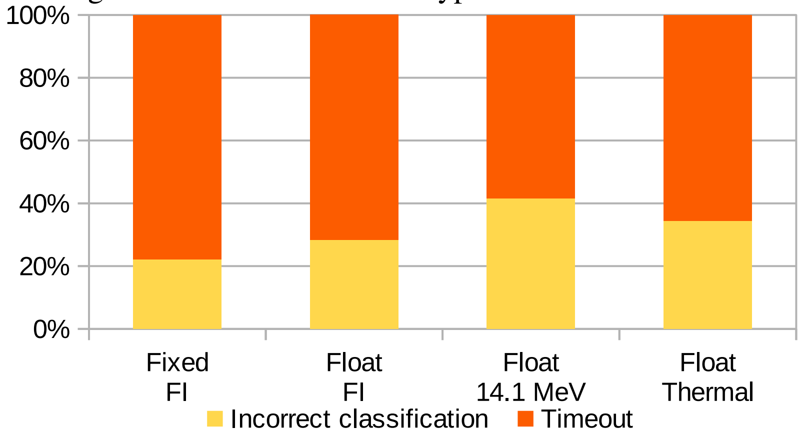
Table 8.9: Results and reliability metrics from irradiation

Metrics	14.1 Mev Neutron	Thermal Neutron
Irradiation time [h]	53.6	0.3
Fluence (Φ) [cm^{-2}]	9.5×10^{9}	1.7×10^{12}
Failure events	48	29
Mean Fluence to Failure - MΦTF [cm^{-2}]	2.0×10^{8}	6.0×10^{10} ($1\times$)

these experiments, the maximum flux, centered at the beam, was 1.9×10^{9} cm^{-2}s^{-1}. The FPGA board was positioned off the beam center for a lower flux adequate for testing the CNN. To minimize radiation interference on other electronic components, the board was covered with a B4C shielding with a hole over the Xilinx Zynq-7000 SoC FPGA device (Figure 8.15(b)).

The FPGA board was irradiated for a total of 0.3 h while testing the CNN. Failure events not directly associated with the CNN processing were discarded, the remaining 29 CNN failure events accounting for a total fluence of 1.7×10^{12} cm^{-2}s. Reliability curves for thermal neutron are presented in Figure 8.14(c), and reliability metrics are summarized in Table 8.9. As one can notice, the three curves show the same reliability curve pattern. Figure 8.16 presents the proportion of each type of functional failure observed in the inference. The prevalence of system hangs, detected by communication timeout (Figure 8.16), suggests that mitigation techniques may focus more on control structures instead of arithmetic operations.

Figure 8.16: Occurrences of types of functional failure



Source: The Author.

8.5 Case-study Summary

This Chapter presented an exploration of the reliability of FGPU, an open-source softcore GPU embedded within SRAM-based FPGAs. This study case was conducted through various methodologies, including fault injection campaigns, neutron irradiation experiments, and the assessment of different floating-point operation implementations.

Initially, the investigation analyzed the trade-offs between software-emulated floating-point (soft-FP) and hardware-accelerated floating-point (hard-FP) implementations within the FGPU. The fault injection campaigns revealed that hard-FP implementations increased the MWBF significantly across different applications, with improvements ranging from 1.2 to 18.6 times over soft-FP implementations. This substantial increase underscored the benefit of hard-FP in terms of reliability despite its higher resource utilization.

Following this, the research focused on the application of selective TMR to specific modules of the FGPU. Individual modules' reliability curves and associated resource usage directed the particular hardening approach. The results indicated that the strategic selection of modules for TMR could lead to a more efficient increase in overall system reliability compared to indiscriminate replication. This nuanced approach provided a more resource-effective way of enhancing reliability, tailor-made for the specific needs and limitations of the FGPU architecture.

Finally, the study addressed the resilience of aerial image classification neural networks running on the FGPU against neutron-induced soft errors. Two methods were employed: emulated fault injection and neutron irradiation experiments. These methods provided consistent results, revealing the susceptibility of the FGPU to soft errors and the need for robust hardening techniques.

In synthesizing the findings from our FGPU experiments within SRAM-based FPGAs, we observed a notable distinction in reliability variation compared to commercial GPUs and prior experiments. Unlike scenarios where the GPU's reliability is application-dependent, our FGPU's reliability in FPGA environments is more consistent, seemingly less influenced by applications and more by the FPGA technology itself, particularly the essential bits and runtime. Software-based fault tolerance techniques may be less effective than anticipated, potentially reducing reliability due to longer run times. This trend aligns with HLS implementations' outcomes (BENEVENUTI et al., 2021), indicating a fundamental aspect of FPGA-based GPU reliability. Though FPGAs offer versatility, they often trail ASICs in performance, suggesting that transitioning FGPU implementations to ASICs could enhance speed and improve reliability. This transition will be the focus of our upcoming Chapter.

9 DESIGNING ASICS WITH GPUPLANNER

In the previous Chapters, we explored the reliability of softcore GPUs within FPGA platforms, delving into their fault tolerance against radiation-induced effects and implementing selective mitigation techniques. As we transition from SRAM-based FPGAs to the realm of ASICs, this Chapter focuses on unveiling the potential of ASICs in achieving high performance and energy efficiency for specialized tasks. We introduce GPUPlanner, an innovative framework for automating the generation of GPU-like accelerators as IP, such as softcore GPUs, from RTL to Graphic Data System II (GDSII), facilitating the design of domain-specific accelerators (PEREZ et al., 2022). We will term these general-purpose accelerators G-GPUs.

This Chapter begins by outlining the GPUPlanner framework, detailing the customization and physical implementation processes that enable the generation of scalable and efficient GPU-like accelerators. We then present an extensive design space exploration, analyzing the performance, power, and area (PPA) trade-offs encountered during the synthesis of various G-GPU configurations. By contrasting these configurations with the widely known RISC-V architecture, we highlight the superior performance benefits for highly parallel applications.

While the primary objective of GPUPlanner is not centered on reliability, the framework sets the stage for future work where fault tolerance can be a focal point. Given the consistent reliability observed in our FPGA experiments, transitioning to ASICs promises performance gains and the potential for improved reliability, especially for applications that leverage the parallelism of GPU architectures.

The Chapter is structured as follows: Section 9.1 introduces the GPUPlanner framework. Section 9.2 provides a detailed discussion of the results obtained from implementing G-GPUs, including performance comparisons with RISC-V. Lastly, Section 9.3 concludes the Chapter with a summary of our contributions and insights into the future of ASIC accelerators in high-performance computing.

9.1 GPUPlanner Framework

Our experimental investigation started by migrating the FGPU, originally designed for FGPA, to ASIC. To this end, a few changes in the architecture were necessary. As compilers for FGPA have a feature to infer memory from RTL automatically, all the memory blocks in the FGPU code were described as regular FFs. In ASIC, memory IPs are hand-instantiated instead of inferred. Thus, the first task was to clearly define the intended behavior from the code and instantiate memory modules. In our experiments, we utilized a 65nm commercial technology. Its memory compiler offers single and dual-port low-power SRAM, with parameters ranging from 16-65536 for addresses and 2-144 bits for word size.

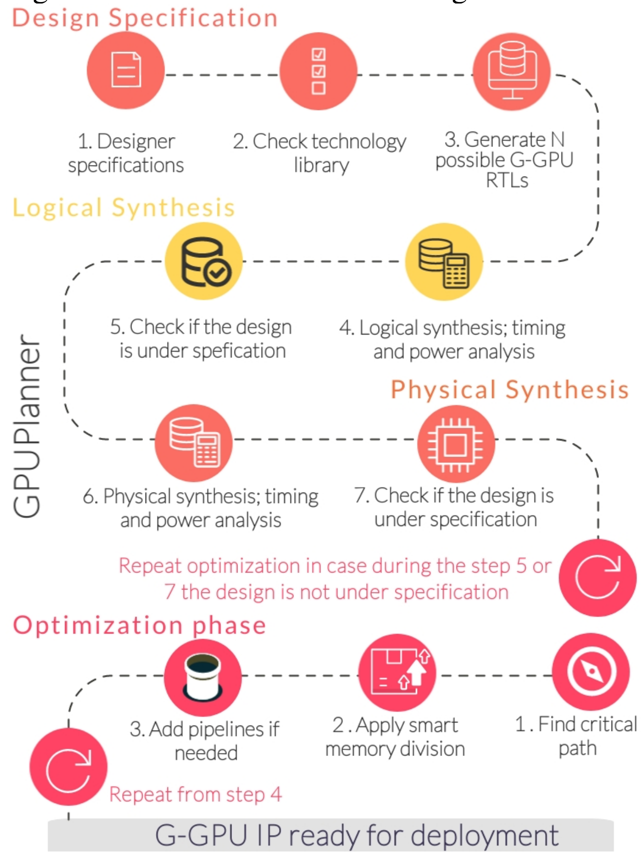
One of our main goals is to achieve the best performance, power, and area (PPA) ratio possible from the G-GPU, exercising the maximum possible design space. The result of this was a selection of versions for different scenarios. The first aspect analyzed was the performance. This is done by finding the maximum operating frequency, which does not violate timing. For the logical synthesis, the value found for the standard version (without any of the optimizations done in this work) is 500MHz. The G-GPU has a similar performance across versions with different numbers of CUs because the CU itself is the bottleneck for performance in this architecture, not the logic for controlling the communication between the modules. As expected, the critical path for the version without any optimization has its starting point at a memory block. Also, the critical path was found inside the CU partition.

Larger memories, either in number of words or word size, display a higher delay for accessing the stored data when compared with smaller memories. This observation guides our design space exploration: dividing the memory blocks in the critical path is a valid strategy for increasing the performance of a design (SUMBUL et al., 2015). Memory division can be applied by dividing the number of words, the size of the word, or both, the latter depending on the performance of the memories available in the given technology. This strategy requires a few alterations in the RTL code. First, the new modules must be instantiated properly, substituting the target memories for the optimization. Second, the address or the input/output data must be concatenated accordingly. To attain faster results, this task was fully automated in our framework. Thus, we only need to point out which memories we want to divide and the number of divisions for applying this strategy.

Following our plan to achieve the best PPA ratio possible, we continually applied the memory division strategy when the critical path contained a memory block. However, the area of the memory blocks is not linear with respect to their size. In fact, two blocks of size $M \times N$ are larger and more power-hungry than a single block of size $2M \times N$ or $M \times 2N$. Therefore, we are increasing the area and power from the division alone. Also, a small extra logic is necessary to accommodate the addressing control of the new blocks (i.e., MUXes to switch between block memories if the number of words is split according to the MSBs of the address). When exercising the memory division to enhance the design performance, we found cases where the critical path was not in memory blocks. To solve such timing issues, pipelines were introduced in those paths. In total, we created twelve different G-GPU solutions, varying the operating frequency and number of CUs.

As a result, we created a tool to generate G-GPU IPs automatically from RTL to GDSII. The flow of GPUPlanner is highlighted in Figure 9.1. For starters, the designer has to define the specifications required from the G-GPU. Our architecture can be configured for different numbers of CUs, ranging from 1 to 8. Increasing the number of CUs enhances the computation capacity of the G-GPU. Also, the designer has to specify the operating frequency of the G-GPU.

Figure 9.1: GPUPlanner's G-GPU generation flow.


Design Specification
1. Designer specifications
2. Check technology library
3. Generate N possible G-GPU RTLs
Logical Synthesis
5. Check if the design is under spefication
4. Logical synthesis; timing and power analysis
GPUPlanner
Physical Synthesis
6. Physical synthesis; timing and power analysis
7. Check if the design is under specification
Repeat optimization in case during the step 5 or 7 the design is not under specification
Optimization phase
3. Add pipelines if needed
2 . Apply smart memory division
1 . Find critical path
Repeat from step 4
G-GPU IP ready for deployment

Source: The Author.

After surveying the possible versions of the G-GPU for desired application scenarios, the designer can generate a specification for each scenario. Then, these specifications are contrasted with the characteristics of the technology intended to be used to create a first-order estimation of the G-GPU PPA. In this phase, there is a possibility of finding several versions suitable for the given specification. Still, it also might happen that a configuration that suits the designer's requirements does not exist. However, our framework is not a static input generator. Instead, we provide a map of how to achieve a realistic PPA that might be close enough to the designer's requirements. This map is a dynamic spreadsheet where the user inputs the delay of the memory blocks required for the non-optimized version of the G-GPU. Our map gives the maximum performance, which memory has to be divided, and where to introduce pipelines to enhance the performance. This is an iterative process and can be repeated until the designer finds the desired performance. Thus, using our map, the designer can rapidly adapt his specifications or create new versions of G-GPU by splitting more memory blocks to increase performance or by introducing on-demand pipelines. Even though applying this strategy is complicated, our framework can handle any memory and technology with little effort. The designer only has to give the basic information of the memory blocks (i.e., name, number of ports, port names, and minimum delay for data access). The only hard constraint is that many G-GPU memories must be dual-port. Further development for single-port memories is scheduled as future work.

After settling the specifications, one or more designs can be feasible, generating a list of G-GPU versions. From a single push of a button, our framework can perform logic and physical synthesis of the list of designs. After the logic and physical synthesis, the resulting PPA is checked to guarantee it is under the initial specification. If the resulting G-GPU is not up to specifications, the designer should modify it and restart the process. In any case, the resulting layouts are ready to be integrated into a system as a tapeout-ready IP.

9.2 Results and Discussion

During the exercise of the GPUPlanner in finding the best trade-offs for a range of operating frequencies, we were able to draw a map of parameters to be adapted to create the versions demonstrated in this work. This map is agnostic of the technology used because our main optimization strategy deals with the intrinsic delay of the memory blocks and the characteristics of the G-GPU architecture. Employing our strategy for other technologies would result in different PPA ratios, depending on the given technology performance. The results depend mainly on the memory performance and the standard cells. However, the points of optimization would be somewhat the same. Users who follow our map will rapidly find the best versions for the given technology.

From the exercise of the GPUPlanner, we found 12 versions worth the PPA trade-off in a general manner. These versions have 1, 2, 4, and 8 CUs. Their variants run at 500MHz, 590MHz, and 667MHz. The characteristics of each version are shown in Table 9.1. In terms of area, the G-GPU size grows linearly with the number of CUs. The optimizations done to augment the performance increased the area by an average of 10%, from 500MHz to 590MHz, and 2%, from 590MHz to 667MHz. Thus, if the power consumption is not a priority, the 667MHz is a good fit for having a negligible increase in area in a trade-off for better performance. These results demonstrate the potential scalability of the G-GPU architecture.

After the logical synthesis, we chose four versions to perform the physical synthesis. Those are the 1CU@500MHz, 1CU@667MHz, 8CU@500MHz, and 8CU@677MHz. A reader can appreciate that these are the extreme cases identified by GPUPlanner. During this phase, the G-GPU is broken into three partitions during implementation: the CU, the general memory controller, and the top. The density of the CU and the general memory controller was set to 70%. Because of our floorplan strategy of breaking the design into partitions, the top has a low density of 30%. Nevertheless, breaking the design into partitions allows the designer to scale G-GPU without any extra effort. Once a CU partition is fully placed and routed, it can be implemented in versions with more than 1 CU by cloning the partition in the final floorplan of the design. Moreover, the user can create a collection of different CU layout blocks and scale the floorplan according to the number of CUs for different application scenarios easily.

Table 9.1: Characteristics of 12 different G-GPU solutions generated by our tool after logic synthesis in Cadence Genus.

#CU & Freq.	Total Area (mm^2)	Memory Area (mm^2)	#FF	#Comb.	#Memory	Leakage (mW)	Dynamic (W)	Total (W)
1@500MHz	4.19	2.68	119778	127826	51	4.62	1.97	2.055
2@500MHz	7.45	4.64	229171	214243	93	8.54	3.63	3.77
4@500MHz	13.84	8.56	437318	387246	177	16.07	6.88	7.14
8@500MHz	26.51	16.39	852094	714256	345	30.79	13.33	13.86
1@590MHz	4.66	3.15	120035	128894	68	4.73	2.57	2.66
2@590MHz	8.16	5.34	229172	221946	120	8.73	4.63	4.81
4@590MHz	15.03	9.72	436807	397995	224	16.41	8.70	9.02
8@590MHz	28.65	18.49	850559	737232	432	31.25	16.81	17.40
1@667MHz	4.77	3.26	120035	130802	71	4.65	2.62	2.72
2@667MHz	8.27	5.45	229172	222028	123	8.72	4.69	4.87
4@667MHz	15.15	9.83	436807	398124	227	16.43	8.75	9.07
8@667MHz	28.69	18.60	848511	730506	435	30.21	19.10	19.76

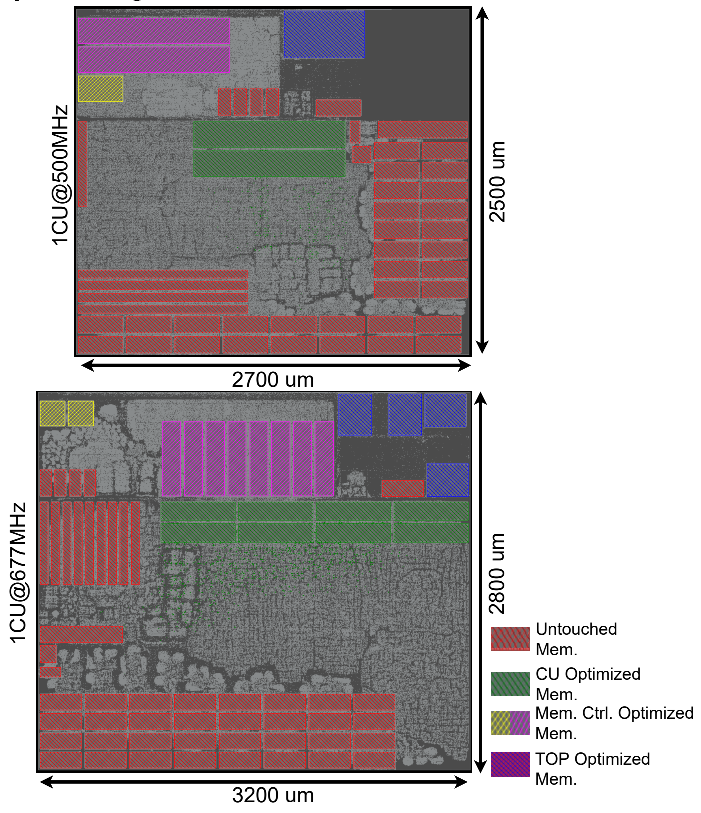
The layouts for the versions with 1 CU and 8 CUs are contrasted in Figure 9.2 and Figure 9.3, respectively. The block memories divided for augmenting the performance are highlighted in green for the CU partition, yellow and pink for the general memory controller, and blue for the top. Note how different the floorplan is between the version with optimizations running at 667MHz (600MHz in the 8 CUs version) and without optimizations running at 500MHz. Block memories must be strategically placed to extract the maximum performance, hence the differences in the floorplan. The layout of the versions 1CU@500MHz, 1CU@667MHz, and 8CU@500MHz have the same performance expected from the logical synthesis (i.e., they can run at the specified clock frequency without any timing violation). However, the layout of version 8CU@667MHz can only run at 600MHz. This finding is explained by analyzing the floorplan of its layout (see Figure 9.3).

The connecting routing wires introduce a significant capacitance because of the long distance between the peripheral CUs and the general memory controller. In turn, this capacitance increases path delay up to a point where it violates the 1.5ns target period. To better explain the difference in wire length routing between 1 and 8 CUs, Table 9.2 shows the total amount of wire length per metal layer[1]. In an attempt to solve this issue, pipelines were introduced between the connections with high delay. Still, this strategy was ineffective in solving the timing violations. To maintain the balanced PPA ratio, the best performance for 8 CUs was 600MHz.

To fully evaluate the usage of G-GPU as an ASIC accelerator, we compared its performance with an implementation of the popular RISC-V architecture. We synthesized both architectures using the same technology utilized during the G-GPU implementation with an operating frequency of 667MHz, the RISC-V having 32kb memory, and the G-GPU with its largest configuration for 1/2/4/8 CUs. As case-study applications,

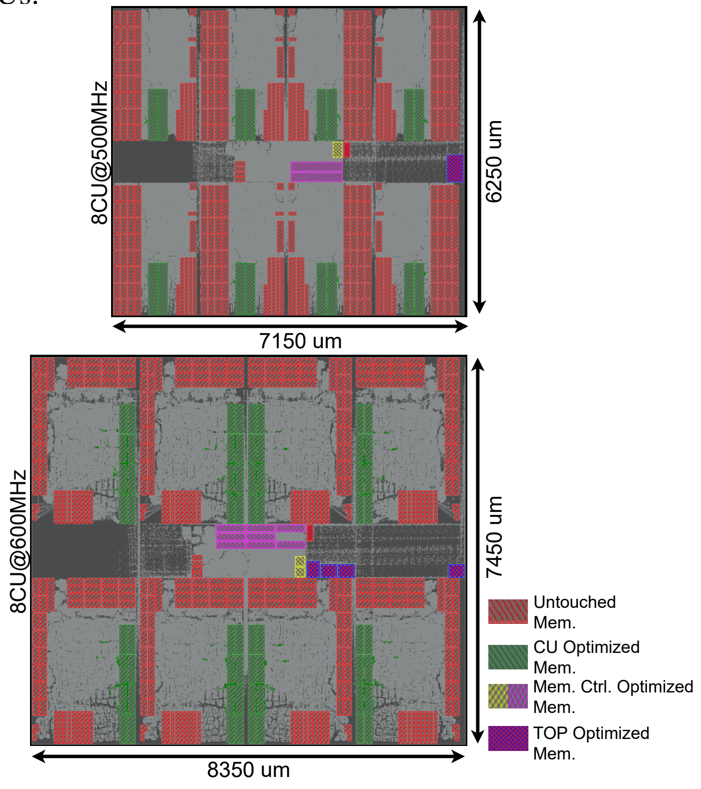
[1]The metal stack contains nine layers for the technology utilized. The metal layers M1, M8, and M9 are reserved for power routing only and have not been drawn in Table 9.2. This is a representative metal stack.

Figure 9.2: Layout comparison between 1CU@500MHz and 1CU@667MHz variants.



Source: The Author.

Figure 9.3: Layout comparison between minimum and maximum performance of a G-GPU with 8 CUs.



Source: The Author.

Table 9.2: Routing length per metal layer for different G-GPU versions and variants.

Metal Layer	Wirelength (μm)			
	1CU@500MHz	1CU@667MHz	8CU@500MHz	8CU@600MHz
M2	3185110	15340072	20314957	25637608
M3	5132356	21219705	27928578	34890963
M4	2987163	9866798	19209669	22387405
M5	2713788	11293663	21953276	26355211
M6	1430594	8801517	14074944	11111664
M7	616666	2915533	6316321	5315697

we chose seven micro-benchmarks from the AMD OpenCL SDK and increased their inputs until RISC-V and its compiler crashed. We further increased the input size of the G-GPU applications to make its computing units fully utilized. To compare the performance of the different-input size applications, we took a pessimistic approach for G-GPU and considered that one could increase RISC-V application input sizes by multiplying its cycle count by the G-GPU/RISC-V input size ratio (which in practice is unfeasible but favors RISC-V). Table 9.3 shows input sizes and measured cycle counts for all case-study applications.

Our first evaluation compares raw performance between G-GPU and RISC-V for the same input sizes. Figure 9.4(a) shows in a bar chart that G-GPU with 8 CUs is up to 223 times faster than RISC-V. However, only applications that enjoy high parallelism are orders of magnitude faster when running G-GPU. G-GPU can be as low as only 1.2 times faster than RISC-V for applications with low to no parallelism. As G-GPU is a domain-specific ASIC accelerator, such results are expected once it is no longer the best option for general-purpose applications. Therefore, a user interested in implementing a G-GPU as an accelerator can utilize these provided data to ponder if this type of architecture is a good fit for his system when considering only the raw speed-up.

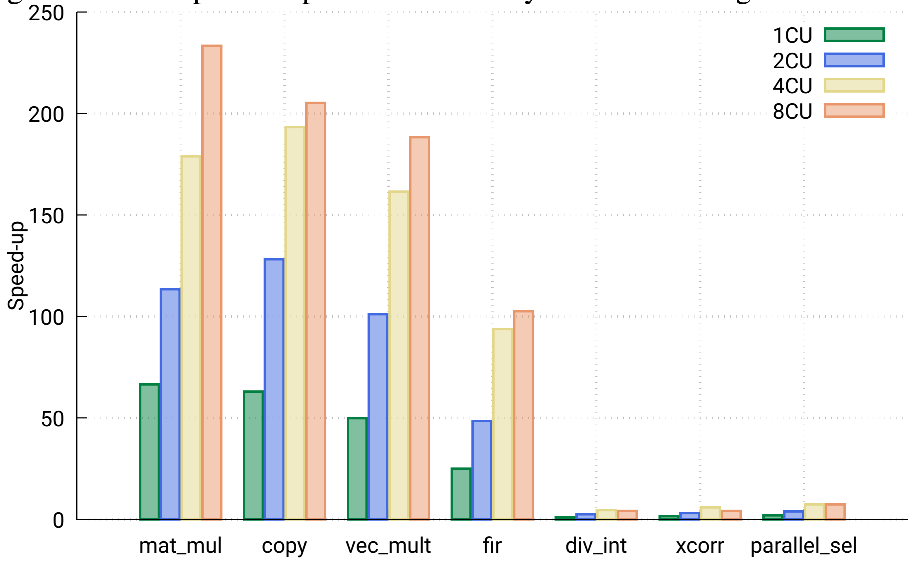
Our second evaluation factors previously measured area into performance speed-up. As designers might be interested in extracting the most from a given available area, we derated the previously measured speed-up by dividing it by the G-GPU/RISC-V area

Table 9.3: Benchmark's input size and cycle count

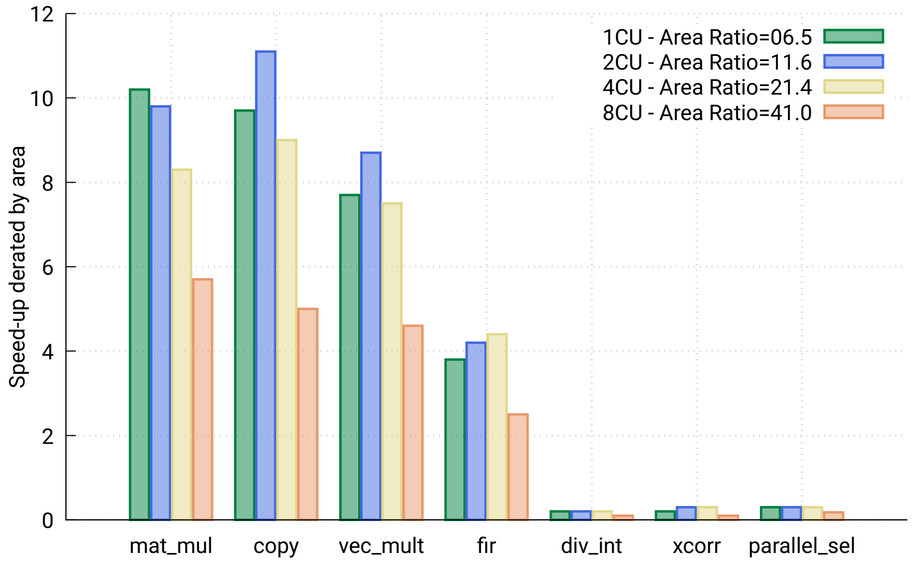
Kernel	Input size		Cycle Count (k-cycles)				
	RISC-V	G-GPU	RISC-V	1CU	2CU	4CU	8CU
mat_multiplication	128	2048	202	48	28	18	14
copy	512	32768	71	73	36	24	22
vec_multiplication	1024	65536	78	100	49	31	26
fir	128	4096	542	694	358	185	169
div_int	512	4096	32	209	105	57	62
xcorr	256	4096	542	5343	2802	1467	2079
paralle_sel	128	2048	765	5979	3157	1656	1660

ratio for each G-GPU CU configuration. This metric is useful to evaluate trade-offs in computation speed-up and area when replacing a RISV-C with a G-GPU. These results are shown in Figure 9.4(b) as a bar chart. G-GPU with 1 CU has an area that is 6.5 times larger than the RISC-V, and it achieves the best increase in performance per area of 10.2 times the RISC-V's. On the other hand, G-GPU with 8 CUs has an area that is 41 times bigger than RISC-V's, thus achieving the best increase in performance per area of 5.7 times faster than RISC-V's. Note that, when factoring area in, the 8-CU G-GPU shows the worst results. This trend happens mainly because data dependency and global memory communication limit parallelism. Thus, it provided increased processing power of a G-GPU configuration with more CUs.

Figure 9.4: Comparative performance analysis of G-GPU against RISC-V.



(a) Speed-up over RISC-V.



(b) Speed-up over RISC-V derated by area.

Source: The Author.

For future work, we plan to update the GPUPlanner to be able to implement the 8-CU G-GPU with performance compared with the versions with fewer CUs. The performance problem of the layouts with 8 CUs can possibly be solved by replicating the general memory controller, shortening the distance between the peripheral CUs, and reducing the delay introduced by the routing wires. Also, we intend to scale FGPU beyond 8 CUs, including a supporting memory hierarchy, and incorporate single-port memories into GPUPlanner.

9.3 Case-study Summary

In this study case, we introduced GPUPlanner, a framework for generating domain-specific ASIC accelerators. This tool automated the transition from RTL to a tape-out-ready layout, underscoring the feasibility of softcore GPUs as high-performance domain-specific ASIC accelerators. When benchmarked against the RISC-V architecture, these GPU-like architectures demonstrated significant advantages for highly parallel applications, offering a promising avenue for the reliability field to deploy robust and efficient softcore GPUs. GPUPlanner enables the broader community to engage in design space exploration of GPU-like accelerators, furthering the potential for tailored, reliable computing solutions. Our findings suggest that these accelerators could significantly advance reliable computing architectures, potentially surpassing current HLS approaches in both performance and reliability, as indicated by comparisons with HLS implementations (CANIS et al., 2011; BENEVENUTI et al., 2021). As we extend GPUPlanner's capabilities to various GPU architectures and technologies, we pave the way for innovative, reliable, high-performance computing platforms.

10 CONCLUSION AND FUTURE DIRECTIONS

This Thesis conducted a thorough investigation into the reliability of GPUs under radiation-induced faults, focusing on SEUs. Our first study case focused on commercial GPUs, particularly Nvidia's Kepler architecture. We analyzed the impact of radiation-induced faults on GPU registers and developed selective fault tolerance strategies, enhancing efficiency over non-selective methods. We further improved SEU fault tolerance by implementing acceptance accuracy relaxation methods while reducing overhead.

Transitioning to FlexGrip, a softcore GPU, we investigated low-level software-based fault tolerance techniques, including novel optimizations. Here, we emphasized the protection of instruction sets and register files against SEUs. We also enhanced the NVIDIA SASS 1.0 ISA for improved mitigation of SDC and DUE effects and implemented hybrid fault tolerance techniques for error correction in the GPU pipeline.

Our research then moved to FGPU, another softcore GPU oriented towards FPGA implementations. We assessed its reliability using hardware and software floating-point implementations, exploring isolated components' unhardened and hardened reliability curves. The effectiveness of selective TMR in enhancing GPU fault tolerance was evaluated. Additionally, we performed radiation experiments and correlated these findings with emulation results to establish a deeper understanding of FGPU's vulnerabilities.

Finally, we explored the domain of ASICs, developing the G-GPU, a domain-specific ASIC accelerator. The introduction of GPUPlanner facilitated the efficient transition from RTL designs to ASIC layouts. Our findings demonstrated the suitability of GPU-like accelerators for high parallelism applications, marking a significant advancement in the exploration of adaptable GPU architectures.

To encapsulate our research across various GPU architectures, Table 10.1 summarizes the case-studies, highlighting the techniques implemented, evaluation methods, and metrics investigated.

In conclusion, this Thesis contributes to GPU reliability in radiation-sensitive environments. We have laid the groundwork for more robust GPU integration in safety-critical domains through our comprehensive evaluation across different GPU architectures and the transition from FPGAs to ASICs. Our approach, combining both software and hardware strategies, offers a pathway toward developing resilient GPUs for various critical applications.

As our study deepened, it became evident that GPU sensitivity is closely linked to the application being executed. This observation leads us to a promising direction for future research: the development of fault-tolerant GPUs designed specifically for targeted sets of applications. Our extensive work in enhancing fault tolerance, underscored by the effective methods we proposed and tested, provides a solid foundation for this exploration. Furthermore, with our established tool for creating ASICs, we envision a path forward in applying these fault tolerance techniques to design robust and resilient ASICs, particularly

Table 10.1: Summary of case-studies

Study Case	**Implemented Techniques**	**Evaluation Methods**	**Metrics Investigated**
Kepler	Selective fault tolerance, approximate computing	Radiation experiments, fault injection	Resources, AVF, FIT
FlexGrip	Low-level software-based fault tolerance, ISA extension, hybrid techniques	Simulation, fault injection	Resources, execution time, AVF, MWTF
FGPU (FPGA)	Selective TMR, Soft-FP and Hard-FP implementations	Hardware emulation, radiation experiments	Resources, execution time, MWBF, MFTF, MEBF, $M\Phi TF$
FGPU (ASIC)	GPUPlanner framework, ASIC design	Performance evaluation	Resources, parallel processing efficiency, ASIC design adaptability

tailored for these specific application groups. This approach not only aligns with the initial discussions and objectives of our research but also opens a significant field for innovation in GPU technology. By focusing on custom ASICs for specific application clusters, we can manage the costs of ASIC development, making this a viable and groundbreaking direction in the realm of fault-tolerant computing solutions.

11 FINAL CONSIDERATIONS

This Thesis aims to serve as a roadmap in the domain of GPU technologies. The scope of this investigation is substantiated by a wide range of experiments conducted across three distinct GPU platforms. Our research methodologies encompass a spectrum of approaches, from low-level software-based fault tolerance techniques to hardware- and hybrid software-hardware approaches. These methodologies have been tested through various means, including simulations, hardware emulations, and neutron beam experiments.

11.1 Collaborative Contributions

This Thesis is part of a broader academic initiative that has led to publications in scientific publications. This work has been made possible through the direct collaboration of the following institutions and individuals:

Universidade Federal do Rio Grande do Sul, Brazil

- Giani Braga
- Ivan Peter Lamb
- Leonardo Gobatto
- Fernando Fernandes
- Fabio Benevenuti
- Raphael Brum
- Fernanda Kastensmidt
- Paolo Rech

Institute for Advanced Studies, Aeronautics Science and Technology Department, Brazil

- Evaldo Carlos F. Pereira Jr
- Rafael G. Vaz
- Odair L. Gonçalez

Brandenburg University of Technology, Germany

- Hector Munoz
- Marcelo Brandalero
- Michael Hubner

Politecnico di Torino, Italy

- Josie E. Rodriguez Condia

- Matteo Sonza Reorda
- Luca Sterpone

Tallinn University of Technology, Estonia

- Tiago D. Perez
- Samuel Pagliarini

Université Grenoble-Alpes, Grenoble, France

- Rodrigo Possamai Bastos

Institute Laue-Langevin, Grenoble, France

- Manon Letiche

11.2 List of Publications

Journal Articles

1. Braga, G.; **Goncalves, M. M.**; Azambuja, J. R. "Software-controlled pipeline parity in GPU architectures for error detection." *Microelectronics Reliability*, vol. 148, pp. 115155, 2023. Elsevier.
2. **Goncalves, M. M.**; Condia, J. E. R.; Reorda, M. S.; Sterpone, L.; Azambuja, J. R. "Evaluating low-level software-based hardening techniques for configurable GPU architectures." *The Journal of Supercomputing*, vol. 78, no. 6, pp. 8081-8105, 2022. Springer.
3. Benevenuti, F.; **Goncalves, M.**; Fonseca, E. C. P.; Vaz, R. G.; Goncalez, O.; Bastos, R. P.; Letiche, M.; Kastensmidt, F. L.; Azambuja, J. R. F. "Investigating the reliability impacts of neutron-induced soft errors in aerial image classification CNNs implemented in a softcore SRAM-based FPGA GPU." *Microelectronics Reliability*, p. 114738, 2022.
4. Braga, G.; Benevenuti, F.; **Goncalves, M. M.**; Hernandez, H. G. M.; Hübner, M.; Brandalero, M.; Kastensmidt, F.; Azambuja, J. R. "Evaluating softcore GPU in SRAM-based FPGA under radiation-induced effects." *Microelectronics Reliability*, vol. 126, pp. 114348, 2021. Elsevier.
5. **Goncalves, M. M.**; Lamb, I. P.; Rech, P.; Brum, R. M.; Azambuja, J. R. "Improving selective fault tolerance in GPU register files by relaxing application accuracy." *IEEE Transactions on Nuclear Science*, vol. 67, no. 7, pp. 1573-1580, 2020. IEEE.
6. **Goncalves, M. M.**; Condia, J. E. R.; Reorda, M. S.; Sterpone, L.; Azambuja, J. R. "Improving GPU register file reliability with a comprehensive ISA extension." *Microelectronics Reliability*, vol. 114, pp. 113768, 2020. Elsevier.
7. **Goncalves, M. M.**; Fernandes, F.; Lamb, I.; Rech, P.; Azambuja, J. R. "Selective

fault tolerance for register files of graphics processing units." *IEEE Transactions on Nuclear Science*, vol. 66, no. 7, pp. 1449-1456, 2019. IEEE.

8. **Goncalves, M. M.**; Saquetti, M.; Azambuja, J. R. "Evaluating the reliability of a GPU pipeline to SEU and the impacts of software-based and hardware-based fault tolerance techniques." *Microelectronics Reliability*, v. 88-90, p. 931-935, 2018.
9. **Goncalves, M. M.**; Saquetti, M.; Kastensmidt, F.; Azambuja, J. R. "A low-level software-based fault tolerance approach to detect SEUs in GPUs' register files." *Microelectronics Reliability*, v. 1, p. 1, 2017.
10. **Goncalves, M. M.**; Tirone, M. S. P. C.; Azambuja, J. R. "Uma Abordagem de Tolerância a Falhas Baseada em Software de Baixo Nível para Detectar SEUs em Bancos de Registradores de GPUs." *Revista Júnior de Iniciação Científica em Ciências Exatas e Engenharia*, v. 1, p. 28-37, 2017.

Conference Proceedings

1. Braga, G.; Gobatto, L.; **Goncalves, M. M.**; Azambuja, J. R. "An Investigation into Fault Detection and Correction in GPU Pipelines with a Hybrid XOR Approach." In: *IEEE Latin American Symposium on Circuits and Systems*, 2024, Punta del Este.
2. Braga, G.; Gobatto, L.; **Goncalves, M. M.**; Azambuja, J. R. "Improving GPU Reliability with Software-Managed Pipeline Parity for Error Detection and Correction." In: *IEEE Latin American Symposium on Circuits and Systems*, 2024, Punta del Este.
3. Braga, G.; **Goncalves, M. M.**; Azambuja, J. R. "Evaluating an XOR-based Hybrid Fault Tolerance Technique to Detect Faults in GPU Pipelines." In: *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2023, Foz do Iguaçu.
4. Perez, T. D.; **Goncalves, M. M.**; Gobatto, L.; Brandalero, M.; Azambuja, J. R.; Pagliarini, S. "G-GPU: a fully-automated generator of GPU-like ASIC accelerators." In: *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 544-547, 2022. IEEE.
5. Benevenuti, F.; **Goncalves, M. M.**; Junior, E. C. F. P.; Vaz, R. G.; Goncalez, O. L.; Azambuja, J. R.; Kastensmidt, F. L. "Neutron-induced Faults on CNN for Aerial Image Classification on SRAM-based FPGA Using Softcore GPU and HLS." In: *European Conference on Radiation and Its Effects on Components and Systems (RADECS)*, pp. 1-4, 2021. IEEE.
6. **Goncalves, M. M.**; Azambuja, J. R.; Condia, J. E. R.; Reorda, M. S.; Sterpone, L. "Evaluating software-based hardening techniques for general-purpose registers on a GPGPU." In: *2020 IEEE Latin-American Test Symposium (LATS)*, pp. 1-6, 2020. IEEE.
7. Condia, J. E. R.; **Goncalves, M. M.**; Azambuja, J. R.; Reorda, M. S.; Sterpone,

L. "Analyzing the sensitivity of GPU pipeline registers to single events upsets." In: *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pp. 380-385, 2020. IEEE.

8. **Goncalves, M. M.**; Benevenuti, F.; Munoz, H.; Brandalero, M.; Hubner, M.; Kastensmidt, F.; Azambuja, J. R. "Investigating floating-point implementations in a softcore GPU under radiation-induced faults." In: *IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pp. 1-4, 2020. IEEE.
9. **Goncalves, M. M.**; Lamb, I.; Brum, R. M.; Azambuja, J. R. "Evaluating the Impact of Accuracy Relaxation in the Reliability of GPU Register Files." In: *IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, 2019, Genoa.
10. Fernandes, F.; **Goncalves, M.**; Azambuja, J.; Rech, P. "Register File Selective Hardening for Graphics Processing Units." In: *Radiation Effects on Components and Systems Conference (RADECS)*, 2018, Gothenburg.
11. **Goncalves, M. M.**; Tirone, M. S. P. C.; Azambuja, J. R. "Evaluating the Efficiency of Software-based Fault Tolerant Techniques to Detect SEUs in GPUs." In: *Radiation Effects on Components and Systems Conference (RADECS)*, 2016, Bremen.

REFERENCES

AL-KHAWLANY, A. H.; KHAN, A.; PATHAN, J. Review on studies in natural background radiation. **Radiation Protection and Environment**, Medknow, v. 41, n. 4, p. 215–222, 2018.

ANDRYC, K.; MERCHANT, M.; TESSIER, R. Flexgrip: A soft GPGPU for FPGAs. In: **Proceedings of the 2013 International Conference on Field-Programmable Technology (FPT)**. [S.l.: s.n.], 2013. p. 230–237.

APONTE-MORENO, A.; PEDRAZA, C.; RESTREPO-CALLE, F. Reducing overheads in software-based fault tolerant systems using approximate computing. In: IEEE. **2019 IEEE Latin American Test Symposium (LATS)**. [S.l.], 2019. p. 95–100.

AVIZIENIS, A. et al. Basic concepts and taxonomy of dependable and secure computing. **IEEE transactions on dependable and secure computing**, IEEE, v. 1, n. 1, p. 11–33, 2004.

AZAMBUJA, J. R. et al. Detecting sees in microprocessors through a non-intrusive hybrid technique. **IEEE Transactions on Nuclear Science**, v. 58, n. 3, p. 993–1000, June 2011. ISSN 0018-9499.

Azambuja, J. R. et al. Evaluating neutron induced see in SRAM-based FPGA protected by hardware- and software-based fault tolerant techniques. **IEEE Transactions on Nuclear Science**, v. 60, n. 6, p. 4243–4250, Dec 2013. ISSN 1558-1578.

BASU, S. et al. DeepSat: A learning framework for satellite imagery. In: **SIGSPATIAL**. Seattle, Washington: [s.n.], 2015. ISBN 9781450339674.

BAUMANN, R. Radiation-induced soft errors in advanced semiconductor technologies. **Device and Materials Reliability, IEEE Transactions on**, v. 5, n. 3, p. 305–316, Sept 2005. ISSN 1530-4388.

BENEVENUTI, F. et al. Neutron-induced faults on cnn for aerial image classification on sram-based fpga using softcore gpu and hls. In: IEEE. **2021 21th European Conference on Radiation and Its Effects on Components and Systems (RADECS)**. [S.l.], 2021. p. 1–4.

BENEVENUTI, F. et al. Investigating the reliability impacts of neutron-induced soft errors in aerial image classification cnns implemented in a softcore sram-based fpga gpu. **Microelectronics Reliability**, Elsevier, v. 138, p. 114738, 2022.

BENEVENUTI, F.; KASTENSMIDT, F. L. Comparing exhaustive and random fault injection methods for configuration memory on SRAM-based FPGAs. In: **IEEE Latin-American Test Symposium**. Santiago, Chile: [s.n.], 2019. p. 87–92.

BENEVENUTI, F. et al. Robust convolutional neural networks in SRAM-based FPGAs: a case study in image classification. **JICS**, v. 16, n. 2, aug 2021.

BOJARSKI, M. et al. End to end learning for self-driving cars. **arXiv preprint arXiv:1604.07316**, 2016.

BRAGA, G. et al. Evaluating softcore gpu in sram-based fpga under radiation-induced effects. **Microelectronics Reliability**, Elsevier, v. 126, p. 114348, 2021.

BRAGA, G.; GONCALVES, M. M.; AZAMBUJA, J. R. Software-controlled pipeline parity in gpu architectures for error detection. **Microelectronics Reliability**, Elsevier, v. 148, p. 115155, 2023.

BRAGA, G. A.; GONÇALVES, M. M.; AZAMBUJA, J. R. Evaluating an xor-based hybrid fault tolerance technique to detect faults in gpu pipelines. In: IEEE. **2023 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)**. [S.l.], 2023. p. 1–6.

BROWN, S. Overview of iec 61508 design of electrical/electronic/programmable electronic safety-related systems. **Computing and Control Engineering Journal**, Citeseer, v. 11, n. 1, p. 6–12, 2000.

CANIS, A. et al. Legup: high-level synthesis for fpga-based processor/accelerator systems. In: **Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays**. [S.l.: s.n.], 2011. p. 33–36.

CARMICHAEL, C. **Triple Module Redundancy Design Techniques for Virtex FPGAs**. San Jose, CA, USA: Xilinx, Inc., 2006. Application Note XAPP197. Available from Internet: <https://www.xilinx.com/support/documentation/application\%5Fnotes/xapp197.pdf>.

CARREIRA, J.; MADEIRA, H.; SILVA, J. G. Xception: A technique for the experimental evaluation of dependability in modern computers. **IEEE Transactions on Software Engineering**, IEEE, v. 24, n. 2, p. 125–136, 1998.

CHE, S. et al. Rodinia: A benchmark suite for heterogeneous computing. In: IEEE. **2009 IEEE international symposium on workload characterization (IISWC)**. [S.l.], 2009. p. 44–54.

CONDIA, J. E. R. et al. Flexgripplus: An improved gpgpu model to support reliability analysis. **Microelectronics Reliability**, v. 109, p. 113660, 2020. ISSN 0026-2714.

CRAFT, A.; SMITH, B.; JOHNSON, C. Craft: Code transformation for fault-resilient software. In: **Proceedings of the ACM Symposium on Fault-Tolerant Computing**. [S.l.: s.n.], 2012. p. 102–110.

DIXIT, A.; WOOD, A. The impact of new technology on soft error rates. In: IEEE. **Reliability Physics Symposium (IRPS), 2011 IEEE International**. [S.l.], 2011. p. 5B–4.

DONGARRA, J.; MEUER, H.; STROHMAIER, E. **TOP500 Supercomputer Sites: November 2015**. 2015. Available from Internet: <http://www.top500.org>.

ESSEN, B. V. et al. Lbann: Livermore big artificial neural network hpc toolkit. In: **Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments**. New York, NY, USA: ACM, 2015. (MLHPC '15), p. 5:1–5:6. ISBN 978-1-4503-4006-9.

GONCALVES, M. et al. Improving gpu register file reliability with a comprehensive isa extension. **Microelectronics Reliability**, Elsevier, v. 114, p. 113768, 2020.

GONCALVES, M. et al. Selective fault tolerance for register files of graphics processing units. **IEEE Transactions on Nuclear Science**, IEEE, v. 66, n. 7, p. 1449–1456, 2019.

GONCALVES, M. et al. A low-level software-based fault tolerance approach to detect seus in gpus' register files. **Microelectronics Reliability**, v. 77, p. 665–669, 2017. ISSN 0026-2714.

GONCALVES, M. M. et al. Investigating floating-point implementations in a softcore gpu under radiation-induced faults. In: IEEE. **2020 27th IEEE International Conference on Electronics, Circuits and Systems (ICECS)**. [S.l.], 2020. p. 1–4.

GONCALVES, M. M. et al. Evaluating low-level software-based hardening techniques for configurable gpu architectures. **The Journal of Supercomputing**, Springer, v. 78, n. 6, p. 8081–8105, 2022.

GONCALVES, M. M. et al. Improving selective fault tolerance in gpu register files by relaxing application accuracy. **IEEE Transactions on Nuclear Science**, IEEE, v. 67, n. 7, p. 1573–1580, 2020.

GUAN, Q. et al. Towards building resilient scientific applications: Resilience analysis on the impact of soft error and transient error tolerance with the clamr hydrodynamics mini-app. In: IEEE. **2015 IEEE International Conference on Cluster Computing**. [S.l.], 2015. p. 176–179.

GUPTA, M. et al. Compiler techniques to reduce the synchronization overhead of gpu redundant multithreading. In: IEEE. **2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)**. [S.l.], 2017. p. 1–6.

HAKOBYAN, G.; YANG, B. High-performance automotive radar: A review of signal processing algorithms and modulation schemes. **IEEE Signal Processing Magazine**, v. 36, n. 5, p. 32–44, 2019.

HARI, S. K. S. et al. Sassifi: An architecture-level fault injection tool for gpu application resilience evaluation. **International Symposium on Performance Analysis of Systems and Software**, p. 249–258, Oct 2017.

HASSANI, R.; AIATULLAH, M.; LUKSCH, P. Improving HPC application performance in public cloud. **IERI Procedia**, v. 10, p. 169–176, 2014. ISSN 2212-6678.

JEDEC. **MEASUREMENT AND REPORTING OF ALPHA PARTICLE AND TERRESTRIAL COSMIC RAY INDUCED SOFT ERRORS IN SEMICONDUCTOR DEVICES**. 2006. <https://www.jedec.org/standards-documents/docs/jesd-89a>. Accessed: 2021-09-19.

KADI, M. A. et al. General-purpose computing with soft GPUs on FPGAs. **ACM Transactions on Reconfigurable Technology and Systems**, Association for Computing Machinery, New York, NY, USA, v. 11, n. 1, jan. 2018. ISSN 1936-7406.

KALRA, C. et al. Armorall: Compiler-based resilience targeting gpu applications. **ACM Transactions on Architecture and Code Optimization (TACO)**, ACM New York, NY, USA, v. 17, n. 2, p. 1–24, 2020.

KASTENSMIDT, F. L.; CARRO, L.; REIS, R. A. da L. **Fault-tolerance techniques for SRAM-based FPGAs**. [S.l.]: Springer, 2006.

LEGAT, U.; BIASIZZO, A.; NOVAK, F. Automated seu fault emulation using partial fpga reconfiguration. In: IEEE. **13th IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems**. [S.l.], 2010. p. 24–27.

Leveugle, R. et al. Statistical fault injection: Quantified error and confidence. In: **2009 Design, Automation Test in Europe Conference Exhibition**. [S.l.: s.n.], 2009. p. 502–506. ISSN 1530-1591.

LI, M.; SASANKA, R. Understanding error resilience of gpgpu. **NVIDIA Technical Report**, 2010.

LISOWSKI, P. W.; SCHOENBERG, K. F. The los alamos neutron science center. **Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment**, Elsevier, v. 562, n. 2, p. 910–914, 2006.

Mahmoud, A. et al. Optimizing software-directed instruction replication for GPU error detection. In: **SC18: International Conference for High Performance Computing, Networking, Storage and Analysis**. [S.l.: s.n.], 2018. p. 842–853. ISSN null.

MARTEN, M.; SUTTON, C. **The Particle Odyssey: A Journey to the Heart of the Matter**. [S.l.]: Oxford University Press, 2002.

MARTINS, M. et al. Open cell library in 15nm freepdk technology. In: **Proc. of the Int. Symp. on Physical Design**. [S.l.: s.n.], 2015. p. 171–178.

MENON, H. et al. Sassifi: Evaluating resilience of gpu applications. **International Symposium on Workload Characterization**, 2014.

MENTOR GRAPHICS. **ModelSim User Guide**. [S.l.], 2022.

MERCHANT, M. Testing and validation of a prototype gpgpu design for fpgas. 2013.

MITTAL, S. A survey of techniques for approximate computing. **ACM Comput. Surv.**, ACM, New York, NY, USA, v. 48, n. 4, p. 62:1–62:33, mar. 2016. ISSN 0360-0300.

MKS Instruments. **MKS Instruments: Semiconductor Devices and Process Technology**. 2017. [Online; accessed 2024-02-07]. Available from Internet: <https://www.mks.com/n/mosfet-physics>.

MOORE, G. E. Gramming more components onto integrated circuits. **Electronics**, v. 38, p. 8, 1965.

Normand, E. Single-event effects in avionics. **IEEE Transactions on Nuclear Science**, v. 43, n. 2, p. 461–474, 1996.

NVIDIA. **NVIDIA DRIVE: Scalable AI platform for Autonomous Driving**. 2020. Available from Internet: <https://www.nvidia.com/en-us/self-driving-cars/drive-px/>.

NVIDIA, C. **Programming Guide :: CUDA Toolkit Documentation**. 2015. Available from Internet: <http://docs.nvidia.com/cuda/cuda-c-programming-guide>. Accessed in: 23 ago. 2016.

NVIDIA Corporation. **NVIDIA Kepler GK110 Architecture Whitepaper**. [S.l.], 2012. Available: <https://developer.nvidia.com/blog/trenches-gtc-inside-kepler/>.

OH, N.; SHIRVANI, P. P.; MCCLUSKEY, E. J. Error detection by duplicated instructions in super-scalar processors. **IEEE Transactions on Reliability**, v. 51, n. 1, p. 63–75, Mar 2002. ISSN 0018-9529.

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

PEREZ, T. D. et al. G-gpu: a fully-automated generator of gpu-like asic accelerators. In: IEEE. **2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)**. [S.l.], 2022. p. 544–547.

QIN, T. et al. Performance analysis of nanoelectromechanical relay-based field-programmable gate arrays. **IEEE Access**, IEEE, v. 6, p. 15997–16009, 2018.

QUINN, H. et al. Using benchmarks for radiation testing of microprocessors and fpgas. **IEEE Transactions on Nuclear Science**, v. 62, n. 6, p. 2547–2554, Dec 2015. ISSN 0018-9499.

RECH, P. et al. Impact of GPUs Parallelism Management on Safety-Critical and HPC Applications Reliability. In: **IEEE International Conference on Dependable Systems and Networks**. Atlanta, USA: [s.n.], 2014. p. 455–466.

REDMON, J. et al. You only look once: Unified, real-time object detection. **Computing Research Repository**, abs/1506.02640, p. 779–788, 2015.

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

RODRIGUES, G. S. et al. Assessing the reliability of successive approximate computing algorithms under fault injection. **Journal of Electronic Testing**, v. 35, n. 3, p. 367–381, Jun 2019. ISSN 1573-0727.

SANTOS, F. F. d.; RECH, P. Analyzing the criticality of transient faults-induced sdcs on gpu applications. In: **Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems**. New York, NY, USA: ACM, 2017. (ScalA '17), p. 1:1–1:7. ISBN 978-1-4503-5125-6.

SANTOS, F. F. dos et al. Reduced-precision dwc for mixed-precision gpus. In: IEEE. **2020 IEEE 26th International Symposium on On-Line Testing and Robust System Design (IOLTS)**. [S.l.], 2020. p. 1–6.

SHEAFFER, J. W.; LUEBKE, D.; SKADRON, K. A hardware redundancy and recovery mechanism for reliable scientific computation on graphics processors. In: **Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware**. [S.l.: s.n.], 2009.

SLAYMAN, C. Soft errors–past history and recent discoveries. In: **IEEE International Integrated Reliability Workshop Final Report**. [S.l.: s.n.], 2010. p. 25–30.

STANDARD, I. 26262 road vehicles-functional safety. **International Organization for Standardization, www. iso. org, date of access**, v. 20171210, 2018.

STEPHENSON MARK SASTRY HARI, S. K. et al. Flexible software profiling of gpu architectures. In: **Proceedings of the International Symposium on Computer Architecture**. [S.l.: s.n.], 2015. (ISCA '15), p. 185–197.

SU, C.-L. et al. Overview and comparison of opencl and cuda technology for gpgpu. In: IEEE. **2012 IEEE Asia Pacific Conference on Circuits and Systems**. [S.l.], 2012. p. 448–451.

SULLIVAN, M. B. et al. Swapcodes: Error codes for hardware-software cooperative gpu pipeline error detection. In: IEEE. **2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)**. [S.l.], 2018. p. 762–774.

SUMBUL, H. E. et al. A synthesis methodology for application-specific logic-in-memory designs. In: **ACM/EDAC/IEEE Design Automation Conference**. [S.l.: s.n.], 2015. p. 1–6.

SUNDARAM, A. et al. Efficient fault tolerance in multi-media applications through selective instruction replication. In: **Proceedings of the 2008 workshop on Radiation effects and fault tolerance in nanometer technologies**. [S.l.: s.n.], 2008. p. 339–346.

SWIFT, T.; REHMAN, S.; SMITH, A. Swift: Software implemented fault tolerance. **Journal of Software Hardening**, v. 4, p. 123–136, 2005.

SYNOPSYS. **HSPICE User Guide**. [S.l.], 2022.

TAN, J. et al. Analyzing soft-error vulnerability on GPGPU microarchitecture. In: IEEE. **2011 IEEE International Symposium on Workload Characterization (IISWC)**. [S.l.], 2011. p. 226–235.

Tiwari, D. 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)**. [S.l.: s.n.], 2015. p. 331–342. ISSN 1530-0897.

Velazco, R.; Foucard, G.; Peronnard, P. Combining results of accelerated radiation tests and fault injections to predict the error rate of an application implemented in SRAM-based FPGAs. **IEEE Transactions on Nuclear Science**, v. 57, n. 6, p. 3500–3505, 2010.

VENKATAGIRI, R. et al. Impact of software approximations on the resiliency of a video summarization system. In: IEEE. **2018 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)**. [S.l.], 2018. p. 598–609.

VIOLANTE, M. et al. A New Hardware/Software Platform and a New 1/E Neutron Source for Soft Error Studies: Testing FPGAs at the ISIS Facility. **Nuclear Science, IEEE Transactions on**, v. 54, n. 4, p. 1184–1189, 2007. ISSN 0018-9499.

VIOLANTE, M. et al. A new hardware/software platform and a new 1/e neutron source for soft error studies: Testing fpgas at the isis facility. **IEEE Transactions on Nuclear Science**, IEEE, v. 54, n. 4, p. 1184–1189, 2007.

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

WENG, J. et al. Dsagen: Synthesizing programmable spatial accelerators. In: **2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)**. [S.l.: s.n.], 2020. p. 268–281.

WUNDERLICH, H.; BRAUN, C.; HALDER, S. Efficacy and efficiency of algorithm-based fault-tolerance on gpus. In: **IEEE International On-Line Testing Symposium**. [S.l.: s.n.], 2013. p. 240–243. ISSN 1942-9398.

XILINX. **Device Reliability Report**. San Jose, CA, USA: Xilinx, Inc., 2019. User Guide UG116. Available from Internet: <https://www.xilinx.com/support/documentation/user\%5Fguides/ug116.pdf>.