Protecting GPU's Microarchitectural Vulnerabilities via Effective Selective Hardening

Protecting GPU's Microarchitectural Vulnerabilities via Effective Selective Hardening

Josie E. Rodriguez Condia*, Paolo Rech*, Fernando Fernandes dos Santos†, Luigi Carro†, Matteo Sonza Reorda*
*Politecnico di Torino - Department of Control and Computer Engineering DAUIN
{josie.rodriguez, matteo.sonzareorda, paolo.rech}@polito.it
†Federal University of Rio Grande do Sul (UFRGS)
{ffsantos, carro}@inf.ufrgs.br

***Abstract*—Graphics Processing Units (GPUs) are today adopted in several domains for which reliability is fundamental, such as self-driving cars and autonomous machines. Unfortunately, on one side GPUs have been shown to have a high error rate and, on the other side, the constraints imposed by real-time safety-critical applications make traditional, costly, replication-based hardening solutions inadequate.**

This paper proposes an effective microarchitectural selective hardening of GPU modules to mitigate those faults that affect instructions correct execution. We first characterize, through Register-Transfer Level (RTL) fault injections, the architectural vulnerabilities of a GPU model (FlexGripPlus). We specifically target transient faults in the functional units and pipeline registers of a GPU core. Then, we apply selective hardening by triplicating the locations in each module that we found to be more critical. The results show that selective hardening using Triple Modular Redundancy (TMR) can correct 85% to 99% of faults in the pipeline registers and from 50% to 100% of faults in the functional units. The proposed selective TMR strategy reduces the hardware overhead by up to 65% when compared with traditional TMR.

***Index Terms*—Graphics Processing Unit (GPU), Reliability, Selective Hardening**

I. Introduction

The computational power and flexibility of modern Graphics Processing Units (GPUs) have boosted their adoption in High Performance Computing (HPC) and in embedded applications, especially when Artificial Intelligence (AI) algorithms based on Deep and Convolutional Neural Networks (DNNs and CNNs) are exploited. Some domains that benefit from GPUs efficiency in executing AI algorithms, such as autonomous systems for automotive, robotics, and space exploration, are also characterized by strict requirements in terms of dependability. The recent market shift, from consumer to safety-critical applications, has gradually raised the interest, and posed questions, about GPUs reliability.

GPU vendors have worked to improve their devices' reliability, for example by designing more robust memory cells [1] or developing (software) hardening solutions targeting the application domains where dependability is crucial [2], [3]. In the meanwhile, the research community has been extensively studying GPU reliability through fault injection/simulation [4], [5] or beam experiments [6], [7]. The effort towards improving GPUs reliability also requires the availability of effective methods to identify the most critical modules in the hardware and to estimate the improved reliability or safety, as described in universally adopted reliability standards, such as ISO26262 [3].

Designing effective and efficient hardening solutions requires to identify the main sources of failures in GPUs. Unfortunately, performing an extensive Register-Transfer Level (RTL) fault injection on complex algorithms in a GPU is unfeasible, as it would take too long (characterizing *one* GPU module executing a simple 5 layers CNN takes more than 740 *hours*) [8]. Inspired by previous works [9], [10], we propose to extend and adapt to GPUs the idea of characterizing the fault effects analysing basic instructions in combination with some specific complex codes. While this task has been performed for conventional CPUs, it has never been investigated for GPUs. Previous solutions do not scale to the higher complexity of the hardware and the intrinsic extreme parallelism of the software in GPUs.

In this work, we propose a fine-grain RTL fault injection analysis on a set of micro-benchmarks that stimulate specific GPU ISA instructions and on a set of applications. The RTL analysis extracts useful information about the modules, and the flip flops inside a module, that are more likely, once corrupted, to affect the execution of the instruction or application. The use of both micro-benchmarks and applications allows to identify weaknesses of modules stimulated by specific instructions and to highlight behaviours that are characteristic of the executed code. Hence, we can more effectively identify those elements which are worth being hardened.

We applied the proposed approach resorting to an RT-level model (FlexGripPlus [11]) mimicking the behavior of NVIDIA GPUs. We identified the modules that are more likely to impact the execution and applied a selective hardening strategy, based on the triplication of critical Flip Flops (FFs). We choose triplication to provide correction rather than simple detection. While triplication has a higher overhead, in fact, in applications such as object detection the current GPU technologies are barely compliant with real-time constraints (40 frames per second). In the event of a fault, then, the system

This work has been partially supported by the European Commission through the Horizon 2020 RESCUE-ETN project under grant 722325 and PERIOD under the grant agreement No 886202.

does not have time to re-process the data before the next frame.

Our results show that with the proposed approach we can *correct* from 85% to 99% of faults in the pipeline registers and from 50% to 100% of faults in the execution units with an overhead *lower than traditional duplication* (that only provides detection).

The remainder of the paper is organized as follows. Section II provides some background information and overviews the related works about hardening mechanisms in parallel devices. Section III describes the proposed approach to characterize the modules and identify the target locations for the selective hardening. Section IV reports the experimental results and their analysis, and Section V draws conclusions and proposes future works.

II. Background and Related Work

Several techniques have been proposed to mitigate the effects of transient faults in computing devices. The available solutions correspond to hardware, software, or hybrid techniques. The hardware solutions are devoted mainly to applications with heavy requirements in terms of functional safety and reliability. In this scenario, the additional costs are justified by the improved features and capabilities. The most classical hardware strategies include *Double and Triple Modular Redundancy* (DMR, TMR), and *Error-Correcting Codes* (ECCs). Other alternatives are selective hardening [12] and custom-optimized [13] techniques. For GPUs, the adoption of these solutions requires a careful evaluation of the introduced area and power consumption overhead. Their higher complexity and the huge amount of computing units they are based on make GPU hardening more challenging than for CPUs. Some GPU mitigation solutions based on Built-In Self-Repair (BISR), exploiting spare modules to replace faulty units, have also been proposed [14]–[16]. Furthermore, some authors proposed the reconfiguration of computational modules [17], [18] and memories [19] in GPUs once a fault is detected. Lately, GPU redundant mixed-precision hardware has been exploited for low-cost error detection [20].

Software mitigation solutions can also be adopted in GPUs. These solutions are based on code adaptations to mitigate effects using functional [21] and algorithmic [22], [23] methods. Nevertheless, the performance degradation and memory overhead can be relevant, compromising the real-time constraints of some GPU applications. In [24], the authors introduced a software-based redundant multithreading mechanism multiplying the threads to be executed. However, the performance overhead depends on the workload and the method cannot always be generalized.

This paper aims to identify the most critical elements in some target GPU modules, thus allowing a fast yet effective flow for applying selective hardening to mitigate faults in the most common modules used for parallel execution and operation management, including the functional units and the pipeline registers. To quantitatively assess the effectiveness of the proposed approach, we resorted to the FlexGripPlus RTL model [11] of an NVIDIA GPU architecture.

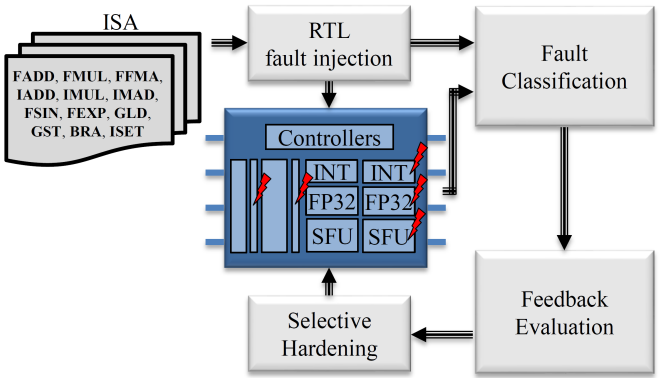

Fig. 1. A general scheme of the proposed methodology to evaluate microarchitectural reliability, perform the fault effect evaluation and select the modules to be hardened in a GPU core.

III. Proposed Methodology

A. Overview

The proposed reliability evaluation and hardening approach is divided in four steps: *i)* RTL fault injection, *ii)* Fault classification, *iii)* Feedback evaluation, and *iv)* Selective Hardening, as depicted in Figure 1.

In the first step, a **microarchitectural RTL fault injection** is performed on the FlexGripPlus GPU model. For this purpose, we inject transient faults (i.e., single bit-flips) in the main computing modules of the GPU core. We consider the Pipeline Registers (PRs), the FP32 and INT functional units, the Special Function Units (SFUs), and the SFU local controller.

In this step, we select a group of instructions from the Instruction Set Architecture (ISA) of the GPU to build a set of micro-benchmarks. In particular, we select the most commonly used assembly instructions and consider the functional modules they use for their execution (details in Section III-B). Moreover, we evaluate a set of applications (FFT, Edge, MxM, nn, and VectorAdd-Vadd) as complementary information to observe the fault distribution dependence on the software and select the main candidates for selective hardening. Further details on the applications we used can be found in [25].

In the second step, **fault classification**, we measure the probability for each fault to reach visible states in the instruction outputs (i.e., to compute the Architectural Vulnerability Factor, AVF [26]). Moreover, the classification determines the type of impact on the instruction output values (single/multiple threads corrupted, how much the corrupted deviates from the expected results, etc.).

The **feedback evaluation** is intended to extract the most critical and common errors observed. Then, these errors are tracked-back to the original micro-structural source, so locating the specific hardware component and location that have caused the error. These locations are used as candidate locations for hardening. Finally, in the **selective hardening** step, the identified critical locations are hardened (triplicated) according to module, type of resource in the GPU core, and complexity.

The next sub-sections explain in more details the procedures employed in the four steps of the proposed method, targeting the FlexGripPlus model as a case study.

B. RTL Fault Injection

We use the RT-level GPU model (FlexGripPlus) [11] to perform the microarchitectural reliability evaluation through a set of fault injection campaigns. FlexGripPlus is an open-source VHDL-based GPU model, which implements the Nvidia G80 architecture [27], with structural details for the most representative modules, and compatible with the commercial CUDA programming environment.

A custom RT-level fault injection framework [28] was developed, using one general controller to manage the *ModelSim* environment hosting the model. This controller injects one fault at a time (corresponding to a single transient) in the targeted GPU module, according to a previously generated fault list. For the proposed microarchitectural analysis, we inject faults in the Pipeline Registers (PRs), the Integer cores (INTs), the Single Precision Floating Point Units (FP32s), the Special Function Units (SFUs) and the control logic (see Figure 1). Table I describes the main features of the targeted modules, including their size, and instructions used.

FlexGripPlus accepts as input one parallel program (*kernel*) for the GPU model. For the purpose of this work, we designed several programs (*micro-benchmarks*) to characterize the effects of RTL faults in the targeted set of instructions. The chosen instructions are:

- Floating point operations (FADD, FMUL, FFMA - Fused MUL and ADD)
- Integer operations (IADD, IMUL, IMAD - MUL and ADD)
- Trascendental functions (SIN, EXP)
- Load/Store (GLD, GST)
- Branch (BRA)
- Integer set predicate/register (ISET).

Although these assembly instructions represent only a small part of all the ≈200 different opcodes in a typical ISA of a GPU, they account for more than 70% of the instructions that compose applications taken from universally adopted benchmark suites for HPC and safety-critical applications (e.g., Rodinia [29], NVIDIA SDK, CNNs [30], [31]).

The programs using **arithmetic instructions** (IADD, IMAD, IMUL, FADD, FMAD, FMUL, FSIN and FEXP) contain four consecutive operations. Each operation accepts different parameters from memory, so executing the same instruction with different values. These programs are configured using 64 parallel threads (2 warps) with each thread executing the same instruction. It must be noted that the framework injects only one fault in each target location of the GPU core per micro-benchmark simulation. This parallel configuration allows us to observe any possible fault effect corrupting several threads as the effect of one single fault propagation and not by multiple fault injections. Each program contains the 64 threads executing the same instruction without interactions between threads.

TABLE I
EVALUATED MODULES, SIZES AND INSTRUCTIONS USED PER MODULE

Module	RTL Size (Flip-Flops)	Type	Instructions
FP32	4,451	Execution/ Data	FADD, FMUL, FFMA
INT	1,542	Execution/ Data	IADD, IMUL, IMAD
SFU	3,133	Execution/ Data	FSIN, FEXP
SFU controller	288	Control	FSIN, FEXP
PRs	3,007	Control/Data	ALL

An ideal RT-level fault injection requires the evaluation of each instruction working on the same or equivalent values used during the operation of the device. Other approaches make use of random patterns. However, both cases require extensive simulation times, which is clearly unfeasible in most scenarios. Thus, we limit the analysis to three input ranges (*Small*, *Medium*, *Large*). We test the floating point and integer opcodes with three different pre-defined input ranges that we identified based on the results of extensive fault injection experiments: *Small* (S, both inputs in the range $6.8\text{x}10^{-6}$ to $7.3\text{x}10^{-6}$), *Medium* (M, in the range 1.8 to 59.4), and *Large* (L, in the range $3.8\text{x}10^{9}$ to $12.5\text{x}10^{9}$). For the instructions using the SFU (FSIN and FEXP), we selected three inputs according to their operational constraints (in the range 0 to $\pi/2$), so avoiding any internal range reduction procedures. To avoid the bias of our results we perform a fault injection campaign on 4 different randomly selected values for each input range.

We also consider **memory movements** (GLD and GST) and **control-flow instructions** (BRA, ISET). The memory movement micro-benchmarks perform one load operation followed by a store operation. For the control-flow operation, we allocate a limited number of setting instructions before a branch operation. A fault is detected when a set register is not correctly assigned or when the branch condition fails. We anticipate that (not surprisingly) in most cases faults affecting control-flow instructions severely affect the execution of the GPU leading to a hang.

C. Fault classification

Once the fault is propagated to any of the available outputs of the target module (instruction output register, memories, or control signals), its effect is classified by comparing the output values and signals with the golden ones obtained in a fault-free simulation, as *Silent Data Corruption* (SDC, i.e., output values mismatch), *Detected Unrecoverable Errors* (DUE, i.e., hang), or *Masked* (i.e., no effect).

The classification stage generates a report per fault campaign, which includes the effect (SDC, DUE, Masked) of each injected fault based on (1) the used instruction, (2) the input value range (3) the target module (where the fault is injected). We also classify the fault effect as *individual* (one single thread affected) or *multiple* (more threads affected). The general report allows to measure the AVF for each module and

instruction as the ratio between the number of observed errors (SDCs/DUEs) and the total number of the injected faults.

D. Feedback Evaluation Analysis

We carefully track all faults propagated to any of the available outputs of the GPU model and then we proceed to determine the original location of the fault causing the observed SDC or DUE. For this purpose, we combine the generated reports and the output results of each micro-benchmark and application. We classify and group the locations, causing the fault propagation, and structures per module. These identified sensitive locations per module (e.g., FFs) represent the main candidates for the selective hardening.

E. Selective Hardening

The reports from the feedback evaluation are analyzed to identify the affected locations. Then, a general report is built containing all sensitive locations. Finally, the hardening target locations are extracted from the general report.

According to the general report, we propose a selective hardening strategy that triplicates only the most vulnerable locations in the modules. It is worth noting that each module is evaluated and the most suitable locations for hardening are identified. Although selective hardening can be employed at several levels of granularity, we work at the register level, so if a flip-flop in a register (or sequential structure) is sensitive to faults, the complete register is targeted for the hardening. Finally, we evaluate and compare the hardware overhead costs and the effect in terms of fault tolerance in the GPU modules and the complete GPU core.

IV. Experimental results

In this Section, we detail the results of the RTL characterization of faults effect in the targeted GPU modules. The fault injection campaigns were performed on a server of 12 Intel Xeon CPUs running at 2.5 GHz and with 256 GB of RAM.

In the experiments, 4 GPU modules (PRs, FP32/INT functional units, special functional units) are characterized using the fault injection described in Section III-B. Each fault injection campaign considers one micro-benchmark using one of the 12 selected assembly instructions and, for each micro-benchmark, we characterize four random values for each of the three input ranges (S/M/L). For each of the 144 fault-injection campaigns we inject 12,000 faults in 4 GPU modules (FP32, INT, SFU, SFU controller, PRs). A similar procedure, although using only the standard workload, is applied for the fault injection campaigns in the five applications (FFT, Edge, MxM, nn, and VectorAdd (Vadd)). Overall we present data from more than $1.88\text{x}10^6$ fault injections. This guarantees a statistical margin error lower than 3%.

A. Fault injection and fault effect evaluation

Figure 2 depicts the AVF distributions for the Functional Units (INT, FP32, and SFU) and the Pipeline Registers (PRs). We have not evaluated data movement and control-flow instructions (GLD, GST, BRA, and ISET) when injecting faults in the functional units, as these modules remain idle.

To have a fine grain evaluation, we divide the injection locations in each module into two groups. We consider the injections in flip flops close to the module's input and output registers (IO) as the first group of evaluation. The second group is composed of the internal sites (IS). Similarly, for the PRs, we divide faults injected in the data path (Data) and control path (Control).

A general overview of the results shows that, for all custom micro-benchmarks (IADD, IMUL, IMAD, FADD, FMUL, and FMAD), faults in the functional units are more likely to produce SDCs than DUEs. On the contrary, for most applications (FFT, Edge, nn, MxM, Vadd) injections are more likely to lead to DUEs. This different trend can be explained considering that, in the applications, INT and FP32 are also employed to calculate memory addresses or indices. In contrast, there is the minimal percentage of DUEs caused in the SFU module (not visible in Figure 2), which is mainly generated by the impact of faults on a local controller. Once a fault corrupts the operation of the controller, it hangs the execution.

Analyzing the fault injection results we found that most faults in the INT and FP32 affect only one of the instantiated threads. On the contrary, evaluating nn, FSIN, and FEXP benchmarks, we found that faults in the special function units (SFUs) affect several threads. This behavior can be explained considering that GPU cores use dedicated functional unit cores (INT and FP32) to operate each related instruction (ADD, MUL, and MAD). On the contrary, the GPU core only includes a few (two) SFUs and is shared among the different threads. Multiple SDCs are caused by faults in the control units of the SFUs. The effect of a fault occurring when managing a thread is propagated also to other threads.

Data in Figure 2 attests that the PRs are more likely to be corrupted when executing control-flow (BRA, iSET) or data movement (GLD, GST) operations. Applications that heavily use these kind of operations (FFT, nn, MxM) are also the ones with higher AVF for the PRs. Interestingly, all instructions (arithmetical, control-flow, and memory movement) use uniformly about 47% of the registers in the PRs. The stress on the PRs does not depend on the instruction, and cannot be the cause for the particularly high AVF for control-flow and data movement instructions. The higher AVF is then only related to the type of instruction and the effect of the fault on the computation.

Most SDCs and DUEs caused by faults in the PRs are originated by control path corruption (about an average of 97.6% of the observed faults). Most of the observed DUEs are caused by corruptions of pipeline control registers that, despite being few (about 16%), are highly critical. Additionally, we found that faults in these pipeline control registers can corrupt multiple threads (up to 18 threads per warp). On the average, $\approx$5% of SDCs caused by faults in the pipeline affect multiple threads. While most PRs ($\approx$84%) store operands for each parallel core and might produce single thread corruptions, registers devoted to control signals are critical as they manage the operation of several threads and cause multiple threads corruptions. Moreover, the evaluation of the memory movement and control

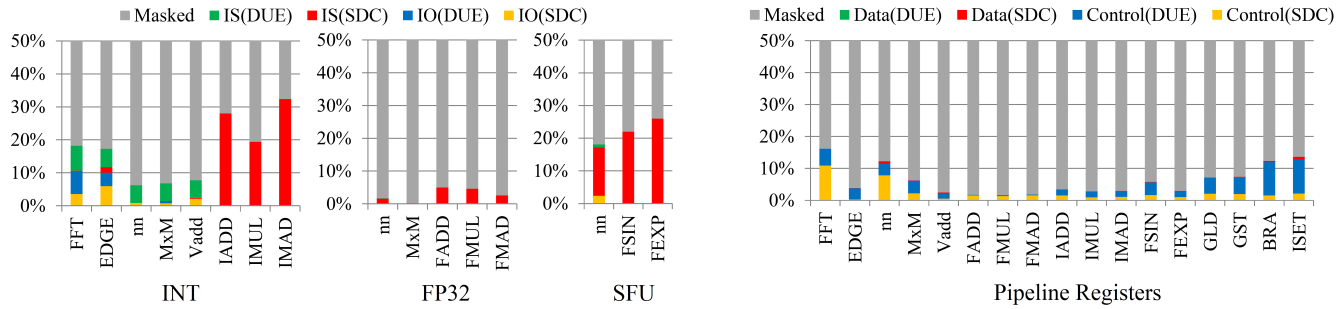

Fig. 2. AVF on the functional units (FP32, INT, SFU) and PRs for the different applications and micro-benchmarks.

flow micro-benchmarks (GLD, GST, BRA, ISET) reveals that faults in the data-path of the PRs can also corrupt the control flow.

As observed in Figure 2, each functional unit has an AVF that depends on the executed instruction (or application). This is because each assembly instruction, involving functional procedures, uses a certain number of submodules in a given structure (e.g., the IADD instruction uses a parallel adder structure instead of a multiplier in the INT module). Several submodules may then remain inactive during the execution of a specific instruction and be activated for others.

Particularly interesting is the case of the INT module, that has a SDCs AVF for the micro-benchmarks but a high DUEs AVF for applications (>60% of the observed faults). This should not surprise, as the applications mostly employ integer instructions for control flow or parallelism management, such as the assignation of thread pointers and memory addressing. A (data) fault in these operations is likely to collapse the GPU operation.

The functional units AVF for the floating point instructions (FADD, FMUL, FMAD) is much smaller than for the integer instructions (IADD, IMUL, IMAD). This is caused by the higher complexity and area of the floating point units, that are more than 3x larger than the integer units (see Table I). A larger area increases the number of injection sites, thus reducing the probability to hit a critical resource for computation. However, the distribution shows that the fault effect is mainly dominated by SDCs in the internal modules of both functional units. Similarly, the applications using INT and FP32 modules (nn and MxM) present the same behavior.

B. Feedback evaluation

With the fault injection results, we trace back the locations, in the GPU core, that produced the observed errors. This allows us to identify the most critical structures for each module.

In case of the PRs, we found that most critical locations are part of the control-path registers (see Control (DUE) and Control (SDC) in Figure 2), which are employed to store and manage the active threads in a program. The affected registers are devoted to store the predicates (*predicate flags*), active threads (*current mask*), memory parameters (*addresses* and *pointers*), among others. When corrupted, these locations affect several threads.

TABLE II
MODULES AND IDENTIFIED CRITICAL SUB-STRUCTURES

Module	Size of the module (FFs)	Structure	Size of the structure (FFs)
PRs	3,007	Data path	1,536
		Predicate Flags	256
		current mask	192
		Memory addresses	384
		Address pointers	224
		Other	167
INT core	595	Input/Output registers	131
		Internal structures	395
FP32 core	4,451	Input/Output registers	0
		Internal structures	2,598
SFU core	3,133	Input/Output registers	32
		Internal structures	1,952
SFU controller	288	Internal structures	288

The analysis of the execution units revealed that, unfortunately, it is not possible to identify locations for the INT and FP32 modules that are more critical than others. Thus, the identification of sub-modules in these units is not as effective as with the PRs. In the execution units there are a few locations that cause the corruption of the output (such as the output registers of the module). However, there are several internal structures that once affected by faults propagate the fault effect. Table II reports only the identified critical locations in the modules and main candidates for the selective hardening. Other locations in the GPU modules, even when corrupted, do not affect computation, due to fault masking or missing activation patterns, so these were discarded as potential reliability targets in the present work. It is worth noting that deeper analyzes, using more kernels would be required to guarantee the injection of most activation patterns, so allowing the complete identification of insensitive sub-modules.

C. Selective hardening

We applied a conservative approach of selective hardening, which consists in triplicating all flip-flops of a critical register (or sub module) and including a majority voter downstream. All the identified sub modules (see Table II) and their flip-flops are the main targets for selective hardening. In detail, complete sub modules (i.e., registers, counters, state machines, etc) are protected even when only internal parts were classified as sensitive to faults.

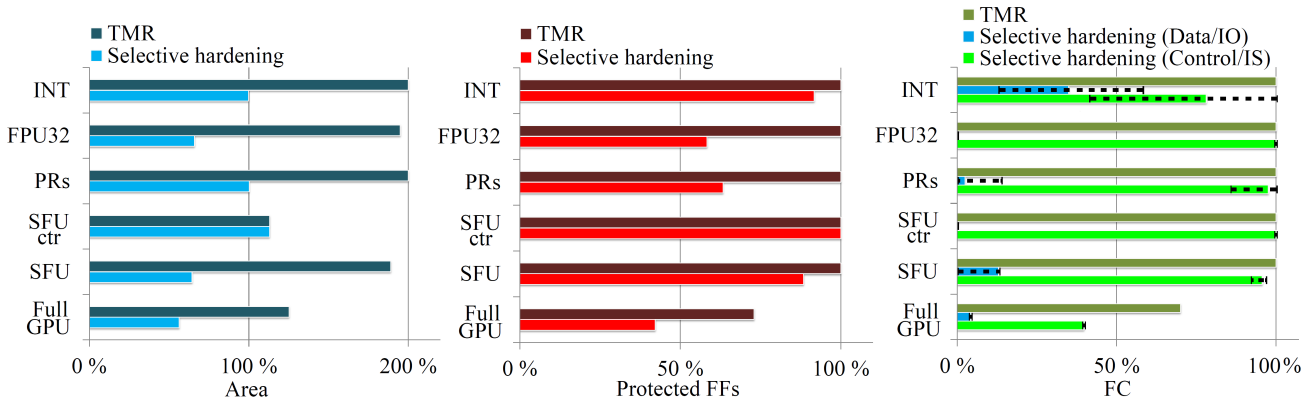

Fig. 3. Area overhead (*left*), percentage of protected FFs (*center*), and Fault Coverage (FC) (*Right*) of the selective hardening TMR and traditional TMR in the evaluated modules. The Functional units are divided as IS/IO and the PRs are divided as control path/data path.

The hardware overhead cost (of selective hardening in the modules and the full GPU) is determined by performing hardware synthesis of the individual module and its hardened version. Furthermore, we also synthesize the modules with a full TMR for comparison purposes. The synthesis framework employs the 15nm Open-Cell NAND-gate library [32].

Figure 3 shows the performance results of the selective vs traditional hardening in the GPU. The Figure shows the relative hardware overhead (Area) for the hardened modules. Moreover, we also show the percentage of protected FFs in both hardening approaches (the proposed hardening and the full TMR) and the Fault Coverage (FC). It is worth noting that all other GPU modules are considered unprotected modules in the estimation of the cost and benefit of the strategies.

If we consider the full GPU, triplicating all the critical modules would increase the area of 125% (memories and other modules are not hardened), while our selective hardening would increase the area of just 52%. Moreover, the selective hardened version covers up to 42.2% of FFs in the GPU core.

In general, the selectively hardened modules have a hardware overhead cost in the range of 60% to 100% which is almost 50% less of the overhead imposed by a traditional TMR. More in detail, the selective hardened versions of FP32 and SFU modules increase the area of about 66% and require to harden just between the 60% and 85% of the available flip-flops on each module, respectively. On the other hand, according to the performed analyzes, the selective hardening of the INT unit requires an increment of 100% in area. Interestingly, the SFU controller requires a complete hardening with maximum overhead in area (about 115%). The small size of the module and the high fault sensitivity of this module justify this choice. The hardware cost in the PRs follows a trend similar to the INT module (near 100% overhead). A large part of the hardware cost is devoted to harden the control-path.

The Fault Coverage results, see FC in Figure 3 (on the right), give an indication of the benefits of the selective hardening. The reported FC is the average coverage obtained with the applications and micro-benchmarks. The error bars (black-dotted) indicate the variation of FC among the applications and micro-benchmarks. The results are divided in sub modules for the evaluated functional units (IO/IS) and the PRs (control path/data path).

From results, the FC in the PRs increases when selectively protecting the control path (about 85% to 99%), see Figure 3(Right). In contrast, a low percentage (<15%) is observed when protecting the data path, only. In the execution units, the IS of each module plays an important role in the application of the selective hardening. According to results from Figure 3, the selective hardening of the internal structures can increase the reliability of an application in a range from 50% to 100%. In contrast, the protection of the input and output registers (IO) has a limited benefit: FC between 0%, 13%, and 60% in the FP32, SFU and INT modules, respectively. As observed for the full GPU, the hardening of IS and control in the modules contribute to the FC in about 38%. Finally, combining the previous results, a selectively hardened version of the GPU can reduce of up to 45% the number of faults of the entire design with less than 55% of area overhead.

V. Conclusions

In this work, we proposed a method to evaluate the reliability of several modules in a GPU with the purpose of performing selective hardening. The evaluation is based on the development of a custom set of micro-benchmarks and some applications. Then, we applied and evaluated the impact of a selective hardening strategy to mitigate the fault effects. The results show that the proposed selective hardening techniques are particularly effective when selecting specific sub-structures, such as the control-path registers in the pipeline registers (85% to 99%). Results also showed that the proposed hardening solution reduces by up to 65% and 55% the hardware overhead in the individual modules and the entire GPU core, respectively, with respect to a complete triple modular redundancy strategy.

In the future, we plan to explore other hardening mechanisms and evaluate other modules in the GPUs, including the schedulers and controllers.

REFERENCES

[1] P. Rech, L. Carro, N. Wang, T. Tsai, S. K. S. Hari, and S. W. Keckler, "Measuring the Radiation Reliability of SRAM Structures in GPUS Designed for HPC," in *IEEE 10th Workshop on Silicon Errors in Logic - System Effects (SELSE)*, 2014.

[2] S. K. S. Hari, M. B. Sullivan, T. Tsai, and S. W. Keckler, "Making convolutions resilient via algorithm-based error detection techniques," 2020.

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

[4] NVLABS, "Nvbitfi: An architecture-level fault injection tool for gpu application resilience evaluations." https://github.com/NVlabs/nvbitfi, 2020.

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

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

[7] F. F. d. Santos, P. F. Pimenta, C. Lunardi, L. Draghetti, L. Carro, D. Kaeli, and P. Rech, "Analyzing and increasing the reliability of convolutional neural networks on gpus," *IEEE Transactions on Reliability*, vol. 68, no. 2, pp. 663–677, 2019.

[8] J. E. R. Condia, F. F. dos Santos, M. Sonza Reorda, and P. Rech, "Combining architectural simulation and software fault injection for a fast and accurate cnns reliability evaluation on gpus," in *2021 IEEE VLSI Test Symposium (VTS)*, pp. 1–6, 2021 to appear.

[9] M. Maniatakos, N. Karimi, C. Tirumurti, A. Jas, and Y. Makris, "Instruction-level impact analysis of low-level faults in a modern microprocessor controller," *IEEE Transactions on Computers*, vol. 60, no. 9, pp. 1260–1273, 2011.

[10] N. Karimi, M. Maniatakos, A. Jas, and Y. Makris, "On the correlation between controller faults and instruction-level errors in modern microprocessors," in *2008 IEEE International Test Conference*, pp. 1–10, 2008.

[11] J. E. R. Condia *et al.*, "Flexgripplus: An improved gpgpu model to support reliability analysis," *Microelectronics Reliability*, vol. 109, 2020.

[12] I. Polian and J. P. Hayes, "Selective hardening: Toward cost-effective error tolerance," *IEEE Design Test of Computers*, vol. 28, no. 3, pp. 54–63, 2011.

[13] M. Gonçalves, J. R. Condia, M. S. Reorda, L. Sterpone, and J. Azambuja, "Improving gpu register file reliability with a comprehensive isa extension," *Microelectronics Reliability*, vol. 114, p. 113768, 2020. 31st European Symposium on Reliability of Electron Devices, Failure Physics and Analysis, ESREF 2020.

[14] J. E. Rodriguez Condia, P. Narducci, M. Sonza Reorda, and L. Sterpone, "A dynamic reconfiguration mechanism to increase the reliability of gpgpus," in *2020 IEEE 38th VLSI Test Symposium (VTS)*, pp. 1–6, 2020.

[15] T. Koal and H. T. Vierhaus, "Logic self repair based on regular building blocks," in *New Trends in Audio and Video / Signal Processing Algorithms, Architectures, Arrangements, and Applications SPA 2008*, pp. 109–114, 2008.

[16] J. E. R. Condia, P. Narducci, M. Sonza Reorda, and L. Sterpone, "Dyre: a dynamic reconfigurable solution to increase gpgpu's reliability," *The Journal of Supercomputing*, p. 1–18, 2021.

[17] W.-J. Lee, S.-H. Lee, J.-H. Nah, J.-W. Kim, Y. Shin, J. Lee, and S.-Y. Jung, "Sgrt: A scalable mobile gpu architecture based on ray tracing," in *ACM SIGGRAPH 2012 Talks*, SIGGRAPH '12, (New York, NY, USA), Association for Computing Machinery, 2012.

[18] K. Kwon, S. Son, J. Park, J. Park, S. Woo, S. Jung, and S. Ryu, "Mobile gpu shader processor based on non-blocking coarse grained reconfigurable arrays architecture," in *2013 International Conference on Field-Programmable Technology (FPT)*, pp. 198–205, 2013.

[19] J. Zhao, G. Sun, G. H. Loh, and Y. Xie, "Energy-efficient gpu design with reconfigurable in-package graphics memory," in *Proceedings of the 2012 ACM/IEEE international symposium on Low power electronics and design*, pp. 403–408, 2012.

[20] F. D. Santos, M. Brandalero, M. Sullivan, R. R. Junior, P. M. Basso, P. Hubner, L. Carro, and P. Rech, "Reduced precision dwc: an efficient hardening strategy for mixed-precision architectures," *IEEE Transactions on Computers*, pp. 1–1, feb 5555.

[21] D. A. G. Gonçalves de Oliveira, L. L. Pilla, T. Santini, and P. Rech, "Evaluation and mitigation of radiation-induced soft errors in graphics processing units," *IEEE Transactions on Computers*, vol. 65, no. 3, pp. 791–804, 2016.

[22] J. Chen, S. Li, and Z. Chen, "Gpu-abft: Optimizing algorithm-based fault tolerance for heterogeneous systems with gpus," in *2016 IEEE International Conference on Networking, Architecture and Storage (NAS)*, pp. 1–2, 2016.

[23] D. Sabena, M. Sonza Reorda, L. Sterpone, P. Rech, and L. Carro, "On the evaluation of soft-errors detection techniques for gpgpus," in *2013 8th IEEE Design and Test Symposium*, pp. 1–6, 2013.

[24] J. Wadden, A. Lyashevsky, S. Gurumurthi, V. Sridharan, and K. Skadron, "Real-world design and evaluation of compiler-managed gpu redundant multithreading," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pp. 73–84, 2014.

[25] B. Du, J. E. R. Condia, and M. Sonza Reorda, "An extended model to support detailed gpgpu reliability analysis," in *2019 14th International Conference on Design Technology of Integrated Systems In Nanoscale Era (DTIS)*, pp. 1–6, 2019.

[26] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin, "A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor," in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, p. 29, IEEE Computer Society, 2003.

[27] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, pp. 39–55, March 2008.

[28] B. Du, J. E. R. Condia, M. Sonza. Reorda, and L. Sterpone, "On the evaluation of seu effects in gpgpus," in *2019 IEEE Latin American Test Symposium (LATS)*, pp. 1–6, 2019.

[29] S. Che *et al.*, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 44–54, 2009.

[30] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

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

[32] M. Martins *et al.*, "Open cell library in 15nm freepdk technology," in *Proceedings of the 2015 Symposium on International Symposium on Physical Design ISPD'15*, p. 171–178, Association for Computing Machinery, 2015.