Improving GPU register file reliability with a comprehensive ISA extension

M.M. Gonçalves, J.E. Rodriguez Condia, M. Sonza Reorda, L. Sterpone, J.R. Azambuja

Abstract – This work proposes a comprehensive ISA extension to improve GPU reliability to transient effects. Three additional instructions are proposed, implemented, and combined with software-based datapath duplication. Modified program codes are compared to state-of-the-art software-based fault tolerance techniques in terms of execution time. The circuit area is evaluated against the original GPU architecture, and a fault injection campaign is performed to assess reliability. Results show that this comprehensive ISA extension improves performance and fault detection capabilities of software-based approaches at negligible costs in terms of circuit area. This work can help engineers in designing more efficient and resilient GPU architectures.

1. Introduction

Graphics Processing Units (GPUs) are specialized Integrated Circuits (IC) designed initially to efficiently manipulate computer graphics and image processing, taking advantage of Thread-Level Parallelism (TLP) to handle highly multi-threaded parallel applications. In the past years, GPUs evolved past computer graphics into general-purpose accelerators for High-Performance Computing (HPC), dealing with applications in a wider range of usages, such as oil exploration, bioinformatics, and deep learning, where finishing tasks under strict timing constraints is a must [1]. More recently, designers have been using GPU in safety-critical applications, such as avionics, self-driving vehicles, and medical, where result correctness is mandatory [2].

Faults on electronic components are mainly caused by energized particles from solar activity and cosmic rays, which can cause permanent or temporary effects. 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) [3, 4].

Among the most observed events caused by energized particles are Single Event Upsets (SEUs). An 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, an 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.

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 [3, 4], up to the point where they can experience radiation effects on applications running at ground level [5]. The consequent SEU events on GPUs are critical to both HPC and safety-critical applications, as SDC effects directly affect the result correctness of safety-critical applications, and DUE effects directly affect the timing constraints of HPC applications. 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 must be executed by the processing system, therefore increasing execution runtime, and can be applied to any GPU architecture with an available program source-code [6]. Hardware-based approaches, on the other hand, can be applied with no 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 require access to GPU architecture description [7]. Recently, open-source GPUs have allowed developers to study the effects of radiation, as well as to design and evaluate fault tolerance techniques [8].

This work proposes a comprehensive ISA extension composed of three classes of resilient atomic instructions to improve software-based fault tolerance techniques. The first two classes, which include load and store instructions, target specifically SDC effects for safety-critical applications, while 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.

The ISA extension is implemented and evaluated on an open-source GPU architecture based on the NVIDIA G80 running five case-study applications. Implementation costs are evaluated in terms of circuit area and critical path delay of the modified GPU architecture and execution runtime and memory footprint of the transformed program codes. To assess reliability, a fault injection campaign is performed by simulation, and fault detection capabilities are compared to state-of-the-art software-based fault tolerance techniques.

The main contribution of this work is to propose a comprehensive ISA extension to GPU architectures that specifically targets HPC and safety-critical applications. Even though ISA extensions have been proposed in the past, to the best of our knowledge, this is the first work in the literature to implement one and quantitatively assess its effectiveness in improving HPC and safety-critical applications' reliability. This work also contributes by discussing hardware modifications for reliability improvement in GPU architectures.

2. Related Work

The literature presents works to improve GPU architectures' reliability against radiation effects and improve the performance of classic software-based fault-tolerance techniques. The most straightforward way to protect GPUs is to run the entire program code twice, store its data, and compare the results. This approach can detect all errors at the performance cost of increasing execution time up to 150% [9, 10]. Partial hardening can decrease this performance cost in exchange for less fault coverage by selectively duplicating instructions and registers based on their criticalities [11, 12].

Authors in [13] proposed to optimize instruction-level fault tolerance techniques by reducing the host notification frequency and enabling better instruction scheduling. The techniques focus on hardening instructions and keeping the ECC enabled to protect storage elements. They were able to improve resilience against SDC errors by up to 87%, with an average execution time increase of 36%.

ISA extensions to improve software-based fault tolerance techniques were also proposed in [13], where a new XOR instruction is used to perform host notification by hardware. The authors propose to replace consistency checks with a signature register that is updated as each instruction executes, adding or subtracting its destination register values based on whether the instruction is an original or a duplicate. The technique includes an extra bit to the instruction formats to inform the hardware when the signature register must be updated. Experiments indicate that these ISA extensions could lower the average runtime overhead to 30%. Still, the proposed techniques focus on datapath protection and rely on the use of ECC to protect the memory structures, such as the register files. However, the use of ECC has been proved to increase the occurrence of DUE effects [9], which may not be acceptable in some fault-tolerant designs, especially when considering HPC applications and their timing constraints.

The authors in [14] analyzed the sensitivity of GPU architectures to SDC and DUE effects. They demonstrated that unhardened memory access instructions make the application mainly susceptible to SDC errors to the point that, by hardening these instructions, they were able to achieve an average reduction in SDC effects of 97%. On the other hand, results showed that unhardened predicate setting instructions make the application mainly susceptible to DUE effects. By hardening these instructions, they were able to achieve a 100% reduction in errors caused by DUE effects. The authors also proposed software optimizations capable of reducing by 34% the average runtime overhead at the cost of a 1% SDC increase.

This work proposes a comprehensive ISA extension to improve performance and fault detection capabilities of software-based fault-tolerance techniques against SDC and DUE effects. The proposed ISA extension is deployed in tandem with state-of-the-art software-based techniques, enabling developers to target SDC- and DUE-induced errors individually. The proposed improvements can be tailored for different scenarios, helping engineers in designing more efficient and resilient GPU architectures.

3. Comprehensive ISA Extension

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 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.

As memory access and set predicate instructions are the main sources of SDC and DUE effects [14], respectively, 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 separate 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, for consistency checking, one extra instruction per instruction-used register must be fully executed by the GPU. 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 and how our ISA extension takes advantage of them, as well as the supporting hardware modifications required to support the new instructions.

3.1. 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. As an 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.

For DUE faults, the same idea applies but considering 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, there are usually portions of the program code that are exclusively used for either the datapath or the controlpath. Therefore, full replication of used registers and their operation instructions, as performed by previous works, 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 4.3.

3.2. Hardware Support

Modern GPUs consist 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.

Considering modern GPU architectures, 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, recent open-source GPUs described in HDL allow designers to implement the required hardware support [8].

4. Implementation

As a base GPU to implement our proposed comprehensive ISA extension, we chose the FlexGripPlus architecture [8]. FlexGripPlus is an open-source soft-core general-purpose GPU described in VHDL that implements the NVIDIA G80 architecture and the SASS 1.0 ISA. The GPU is programmed in CUDA and supports 28 instructions. It runs the native NVIDIA G80 SASS code and accepts a set of kernel configuration parameters, such as grid and block dimension, blocks per core, and registers per thread. These parameters are manually defined before system operation.

It follows the SIMT paradigm through 32 threads, denoted warp, which is fetched, decoded, and distributed to be processed in the Scalar Processors (SPs) at the Execute stage, configured to implement 32, 16 or 8 SPs. The Read and Write stages load/store data operands from/to the register files and shared, global, or constant memories. The GPRF is used to store data operands and addresses during the kernel execution. The PRF is used to store the result of logic-arithmetic or comparison instructions. The local memory is mainly employed to store data arrays, while the shared memory stores data operands that can be used among threads belonging to the same block, and the global memory stores the initial inputs and the final results of a program kernel.

As case-study applications, we chose five case-study applications: matrix multiplication, Fast Fourier Transform (FFT), vector sum, bitonic sort, and edge detection. All case studies are simple applications but differ in their use of the GPU's controlpath and datapath. In terms of data flow 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 FlexGripPlus architecture, and software modifications to the compilation flow to most effectively employ our proposed instructions with software-based hardening techniques.

4.1. 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 into becoming resilient atomic ones.

To solve the first challenge, instead of creating new opcodes, we used a single extra bit. 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 first consider the GPU register file and how instructions access it. FlexGripPlus 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 partially replicate instruction registers, 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 a 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).

4.2. 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 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 FlexGripPlus architecture, we synthesized the original design and three modified versions of the FlexGripPlus 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 15 nm cell library [15], and a 500 MHz constraint, and evaluated circuit area, number of logic cells, power, and critical path delay.

Table I 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 in 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.

4.3. Software Implementation

The case-study applications were written in CUDA and compiled with the NVIDIA NVCC compiler. The compilation process created CUDA binaries, from which the assembly code was extracted with the cuobjdump tool from the CUDA toolkit. The assembly was then translated to bytecode, which can be directly loaded into FlexGripPlus' program memory. To automatically apply software-based hardening techniques to the case-study applications, we improved the Post-Compiling Hardening Tool (PCHT) [16] to provide the previously discussed software-support. We input the assembly codes to PCHT, which then automatically applied the code transformations and generated hardened assembly codes.

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 I – Hardware implementation overhead

	Original Design	Hardware overhead (%)		
		ISAset	ISAls	ISAsetls
Area (mm^2)	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

	Original Code	Software-based			Software-based + ISA Extension		
		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 PE,R1,R1'		@!PE SETP PE,R1,R1'			
6:	SETP P0,R2,R3	SETP P0,R2,R3	SETP P0,R2,R3	SETP P0,R2,R3	SETP P0,R2,R3	**raSETP P0,R2,R3,offset**	**raSETP P0,R2,R3,offset**
7:			@!PE SETP PE,R2,R2'	@!PE SETP PE,R2,R2'			
8:			@!PE SETP PE,R3,R3'	@!PE SETP 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 PE,R1,R1'		@!PE SETP PE,R1,R1'			
12:		@!PE SETP PE,R4,R4'		@!PE SETP PE,R4,R4'			
13:		@PE BRA ERROR	@PE BRA ERROR	@PE BRA ERROR			

Fig. 1. Program code transformation by software-based techniques and software-based techniques with proposed ISA extension.

Fig. 1 shows code transformations 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 II shows the execution time for the original applications, and their overhead for the 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 around 50%. Data also show that data flow-oriented applications, such as FFT and matrix multiplication, presented few 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. Thus, it cannot be hardened purely by software-based techniques.

Table II: Execution time overhead

Application	Original (µs)		Hardening technique overhead (%)		
			Memory	Set Predicate	All
FFT	964	SW	93.1	88.7	103.2
		ISA	56.3	24.5	65.6
Matrix Mult.	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 Detect.	1,096	SW	–	–	–
		ISA	49.5	75.1	66.1

5. Evaluation

The fault injection campaign was automatically performed through simulation at RTL in the ModelSim simulator. 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, achieving a 1% statistical error considering a 95% confidence level [17].

Tables III and IV 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, and therefore there is a small window in which a fault can affect the memory.

6. Conclusions and Future Work

This work presented a new comprehensive ISA extension to the NVIDIA SASS 1.0 ISA to improve GPU reliability against transient effects. We proposed and discussed in detail three additional instructions, targeting memory access, and set predicate instructions to mitigate SDC and DUE effects. The implementation considered software and hardware support to address ISA, hardware, and software improvements to the FlexGripPlus open-source GPU. The proposed resilient atomic instructions were then incorporated into software-based hardening techniques and automatically applied to five case-study applications. Finally, a fault injection campaign was performed by simulating 280,000 faults at RTL.

Hardware synthesis results showed no performance degradation and less than 1% area and power overheads for the modified architectures. Execution runtime overheads showed a decrease in around 50% compared to state-of-the-art software-based techniques. For hardening the matrix multiplication against DUE effects, our proposed ISA required a 9% increase in execution time. Fault injection results showed that our proposed ISA extension could improve overall software-based fault detection, especially when considering DUE effects.

In the future, we intend to improve our hardware implementations and add reliability-specific hardware modules to decrease overheads in fault-tolerant GPUs further.

Table III: SDC Reduction

Application	SDC Effects		Hardening technique (%)		
			Memory	Set Predicate	All
FFT	1,452	SW	89.1	39.9	100.0
		ISA	84.9	17.5	96.1
Matrix Mult.	3,461	SW	100.0	- 4.6	100.0
		ISA	100.0	- 5.6	100.0
Vector Sum	3,164	SW	100.0	–	–
		ISA	100.0	–	–
Bitonic Sort	1,739	SW	61.1	87.5	100.0
		ISA	92.6	71.4	100.0
Edge Detect.	258	SW	–	–	–
		ISA	99.6	80.2	100.0

Table IV: 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 Mult.	1,755	SW	0.2	99.9	99.8
		ISA	4.0	100.0	100.0
Vector Sum	1	SW	100.0	–	–
		ISA	100.0	–	–
Bitonic Sort	1,368	SW	1.9	99.9	99.9
		ISA	5.3	100.0	100.0
Edge Detect.	1,952	SW	–	–	–
		ISA	12.7	99.9	99.9

7. Acknowledgments

This work has been partially supported by the European Commission through the Horizon 2020 RESCUE-ETN project under grant 722325, Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) – Finance Code 001, CNPq, and FAPERGS.

References

[1] M. Snir et al., “Addressing failures in exascale computing,” *High Performance Computing Applications*, vol. 28, no. 2, pp. 129–173, 2014.

[2] M. Bojarski et al.,“End to end learning for self-driving cars,” *arXiv preprint arXiv:1604.07316*, 2016.

[3] C. Slayman, “Soft errors—past history and recent discoveries,” in *2010 IEEE International Integrated Reliability Workshop Final Report*. IEEE, 2010, pp. 25–30.

[4] A. Dixit and A. Wood, “The impact of new technology on soft error rates,” in *2011 Int. Reliability Physics Symp.*. IEEE, 2011, pp. 5B–4.

[5] P. Rech et al., “An efficient and experimentally tuned software-based hardening strategy for matrix multiplication on gpus,” *IEEE Trans. on Nuclear Science*, vol. 60, pp. 2797–2804, 2013.

[6] M. Gonçalves et al.,“A low-level software-based fault tolerance approach to detect seus in gpus’ register files,” *Microelectronics Reliability*, vol. 76, pp. 665–669, 2017.

[7] J. R. Azambuja et al., “Evaluating neutron induced see in srambased fpga protected by hardware-and software-based fault tolerant techniques,” *IEEE Trans. on Nuclear Science*, vol. 60, no. 6, pp. 4243–4250, 2013.

[8] J. E. R. Condia et al., “Flexgripplus: An improved gpgpu model to support reliability analysis,” *Microelectronics Reliability*, vol. 109, p. 1-14, 2020.

[9] L. L. Pilla et al., “Software-based hardening strategies for neutron sensitive fft algorithms on gpus,” *IEEE Trans. on Nuclear Science*, vol. 61, no. 4, pp. 1874–1880, 2014.

[10] M. Dimitrov, M. Mantor, and H. Zhou, “Understanding software approaches for gpgpu reliability,” in *Proc. of Workshop. on General Purpose Processing on GPU*, 2009, pp. 94–104.

[11] A. Sundaram et al., “Efficient fault tolerance in multi-media applications through selective instruction replication,” in *Proceedings of Workshop on Radiation effects and fault tolerance in nanometer technologies*, 2008, pp. 339–346.

[12] M. Goncalves et al., “Selective fault tolerance for register files of graphics processing units,” *IEEE Trans. on Nuclear Science*, vol. 66, no. 7, pp. 1449–1456, 2019.

[13] A. Mahmoud et al., “Optimizing software-directed instruction replication for gpu error detection,” in *Int. Conf. for HPC, Networking, Storage and Analysis*. IEEE, 2018, pp. 842–853.

[14] M. M. Goncalves et al., “Evaluating software-based hardening techniques for general-purpose registers on a gpgpu,” in *2020 IEEE Latin-American Test Symp.* IEEE, 2020, pp. 1–6.

[15] M. Martins et al., “Open cell library in 15nm freepdk technology,” in *Proc. of the Int. Symp. on Physical Design*, 2015, pp. 171–178.

[16] J. R. Azambuja et al., “Detecting sees in microprocessors through a non-intrusive hybrid technique,” *IEEE Trans. on Nuclear Science*, vol. 58, no. 3, pp. 993–1000, 2011.

[17] R. Leveugle et al., “Statistical fault injection: Quantified error and confidence,” *in Proc. Design, Automation & Test in Europe Conference & Exhibition*. IEEE, 2009, pp. 502–506.