On the evaluation of SEU effects in GPGPUs

B. Du*, Josie E. Rodriguez Condia[†], M. Sonza Reorda[‡], L. Sterpone[§]
Politecnico di Torino, Torino, Italy
{*boyang.du, [†]josie.rodriguez, [‡]matteo.sonzareorda, [§]luca.sterpone}@polito.it

Abstract[1]—**General Purpose Graphic Processing Units (GPGPUs) are effective solutions for high-demand data applications which involve multi-signal, image and video processing thanks to their powerful parallel architecture. In the last years, GPGPUs have been considered also for safety-critical applications, such as autonomous and semi-autonomous car driving systems. New GPGPU devices include an increasing number of parallel cores in order to increase throughput and performance. This increment in the number of cores and the requirements in terms of power consumption force designers to use aggressive semiconductor technologies. Nevertheless, those new devices can be seriously affected by radiation effects, modeled as Single Event Upsets (SEUs). SEUs could generate unexpected operation effects in the applications which could be unacceptable for the safety-critical ones. This work analyzes the SEU effects resorting to an open-source model of a GPGPU based on the Nvidia's G80 architecture and aims at complementing previous analysis based on radiation experiments.**

Keywords—***SEU, General Purpose Graphics Processing Units GPGPUs, Graphics Processors, Fault Simulation***

I. Introduction

General Purpose Graphic Processing Units (GPGPUs) represent an effective solution in applications requiring high performance data-intensive operations, such as multi-signal analysis, image and video processing, thanks to their highly parallel architecture. Nowadays, these devices are also considered for embedded real-time safety-critical applications, such autonomous machines and autonomous car driving systems.

Examples in the automotive domain include sensor fusion systems and Advanced Driver-Assistance Systems (ADAS)[1], which form specialized systems devoted to different applications, including Automatic Parking, Automatic Cruise Control, Pedestrian and Pattern Recognition and Forward Collision Warning. The use of ADAS systems is also considered as an intermediate step towards the development of semi-autonomous and fully-autonomous cars. These systems are based on a large number of sensors (*stereo, 360 degree and long distance cameras, radars and Lidars*) producing large amount of data which requires to be processed and decisions to be made under real time constraints.

The new GPGPU devices include more parallel cores, called streaming multiprocessors (SM) in Nvidia's terminology, in order to increase the throughput and operation performance. These increments, in the number of cores, and in requirements, in terms of performance and power consumption, force designers to use aggressive technology scaling approaches. However, it is known that latest semiconductor technologies can be particularly affected by radiation effects [2]. Radiation particles can affect the system through transient faults, such as Single Event Upsets (SEUs) in sequential logic or memory cells by corrupting the content of the stored logic value. SEUs may generate unexpected misbehaviors and consequences that could be unacceptable in safety-critical applications. Hence, effective solutions are required to detect and mitigate those faults affecting GPGPU devices [3]. Possible solutions also include acting on the software coding style and on the adopted algorithm [4].

In order to choose the most suitable countermeasures, a detailed analysis is first needed in order to identify the module criticality and the incidence of faults on the application failure rate. This analysis may also provide guidelines for effectively writing the application code, trading off performance and reliability. One solution to support the analysis is based on fault injection on models at various abstract levels. Nevertheless, some challenges exist in the GPGPU field. In fact, very few GPGPU models exist, and most of them are described at a high level of abstraction [5-8], thus preventing a detailed analysis of SEU effects on critical and complex units such as control-path modules. On the other hand, there are a few RTL behavioral GPGPU models which can be used to analyze the SEU effects in those special-purpose modules.

In [9] the authors describe a SEU fault injection methodology using a behavioral/RTL GPGPU model. Moreover, this and other works [10] introduced an initial overview of the SEU effects on data-path units of a GPGPU, such as the register file and the pipeline registers. The results show that the error rate in these modules is quite variable according to the adopted benchmark and parallelization strategy (e.g., in terms of thread distribution). However, control-path units were not analyzed.

A different approach to the analysis of SEU effects relies on radiation experiments. In [11, 12] the authors presented results from radiation experiments on GPGPUs showing that SEU effects, detected in application results, depend on the affected module in the GPGPU. Moreover, these effects are correlated with the module usage by the benchmarks. Nevertheless, in these approaches it is hard to provide convincing explanations about the observed behaviors of critical units, such as control-path modules, since details about the internal structure and behavior of the device are not available.

One solution to clarify the issue is based on resorting to fault simulation of SEU effects on target modules using some GPGPU model. In this work, we started from an open-source GPGPU model (FlexGrip)[13], and improved it to remove some limitations and bugs. The new version of the model allows us to study in a much more detailed manner the effects of SEUs in different target modules. At the same time, representative applications were designed and selected as benchmarks for SEU fault injection campaigns. These program kernels are representative of those employed in real signal and image processing applications.

[1] This work has been partially supported by the European Commission through the Horizon 2020 RESCUE-ETN project under grant 72232.

This work presents the results of a detailed analysis about the SEU effects on data-path and control-path modules for different applications using different parallelism levels and under various GPGPU configuration modes. To the best of our knowledge, this is the first work that presents results of SEUs injection campaigns on control-path modules of an architectural RTL GPGPU model. It is also the first time that relatively complex applications comparable to real-world ones are considered.

The paper is organized as follows. In Section II the key characteristics of the FlexGrip GPGPU model are introduced. Section III presents the proposed method and the setup employed in the SEU fault injection campaigns, the targeted modules and the selected benchmarks. Section IV reports the experimental results, and Section V finally draws some conclusions.

II. FLEXGRIP

FlexGrip is a RTL VHDL behavioral model of a GPGPU module developed by the University of Massachusetts and originally targeting a Xilinx FPGA [13]. The module is based on the Nvidia G80 Tesla architecture and is compatible with Nvidia's CUDA Compilation Toolkit under SM_1.0 compatibility level. The module uses a compiled CUDA-binary code (.SASS file) as kernel program. 27 instructions of either 32 or 64 bits are supported by FlexGrip.

In FlexGrip, the kernel parameters, such as Grid dimension, Block dimension and Blocks per core, should be manually configured before simulation. Additionally, memory values in constant memory and other GPGPU configuration parameters, such as the number of registers per thread and the number of blocks per SM core, must be configured for each application.

The GPGPU architecture is based on the SIMT (*Single Instruction Multiple Thread*) paradigm and exploits a custom SM core with a five stages pipeline (*Fetch*, *Decode*, *Read*, *Execution/Control-flow* and *Write-back*), as shown in Fig. 1. Moreover, the SM employs a controller and a warp scheduler unit for instruction thread management. In the SIMT architecture, one instruction is fetched, decoded and distributed to be executed on an independent processing unit, or Scalar Processor (SP), in the SM. The Read and Write-back stages load and store data operands from and to Register Files (RFs), shared, global or constant memories. Only integer operations are supported by FlexGrip. For the purpose of this work, the GPGPU model has been improved giving support to 28 instructions in up to 74 instruction formats. Additionally, the technology dependence on Xilinx FPGA has been removed to target on the OpenCell library[14]. Moreover, some bugs and unsupported features related with module interconnections, instructions implementation and nesting thread divergence management have been fixed. Those additional changes allow the model to execute more complex applications.

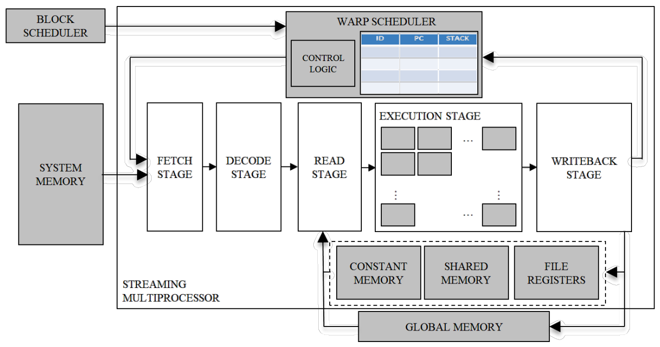

Fig. 1. FlexGrip architecture: the SM.

The GPGPU Thread-level parallelism (TLP) in Flexgrip is customizable and depends on the total number of SPs in the execution stage. The model supports the configuration of 32, 16 and 8 SPs.

The SP configuration number has direct impact on the warp scheduler distribution. In a 32-SPs configuration, the maximum TLP is achieved with 32 threads executed in parallel, one thread per SP. For a 16-SPs configuration, the warp threads are organized in two lanes per SP and every SP should execute 2 threads per warp in sequence. Finally, in the 8-SP core configuration, a 32-threads warp requires four lines and every SP should execute 4 threads per warp.

FlexGrip employs an additional module to support thread-level branching at hardware level (*branch unit*). This module manages control-flow operations in order to generate or retake the flow from conditional branches with multiple paths.

In the SIMT architecture, a conditional control-flow instruction causes *divergence paths* and a set of threads, whose size is lower than warp size, selects different execution paths (Taken and Not-Taken). On the other hand, a synchronization instruction creates a synchronization point defining an instruction location for the previous paths to *converge*. Once path divergence occurs, all the threads will continue the instruction execution until all of them reach the synchronization point. The organization of the branch unit includes a warp divergence stack memory able to store thread information such as thread mask, program counter and flow ID.

III. PROPOSED APPROACH FOR FAULT INJECTION

A custom fault injector was developed to identify and analyze the SEU effects on specific internal modules of the SM. Sub-section B introduces the targeted modules for fault injection. Finally, a description of the designed benchmarks is presented in sub-section C.

A. Custom Fault Injector

The instrument was designed to perform a set of SEU fault campaigns on the FlexGrip model and follows the fault injection methodology introduced in [9]. Moreover, it includes a multi-thread approach [15, 16] and the utilization of a de-rating factor (UDR) [17] of the targeted modules. The UDR factor considers only the registers and memory locations employed by an application during the simulation, thus reducing the total amount of faults to be injected and the corresponding simulation time of a fault campaign.

The custom fault injector was designed in a high-level language (*Python*) and it is able to link a set of configuration files with the execution of the behavioral/RTL simulator (in our case *ModelSim* by Mentor) of the GPGPU model. This tool is composed of a set of modules: a control manager, a fault decoder-generator, a fault injector and a checker and classifier module. The functions, in the fault simulator, can handle the execution of multiple fault campaigns of the model.

The injector can introduce two types of faults: permanent faults, based on the Stuck-at fault model, and transient faults, based on the SEU model.

It is worth noting that it is complex to represent and inject SEU faults in a behavioral/RTL model without timing information details. The fault simulator injects Single Bit Upsets (SBUs) in memory cells or register signals in order to generate the equivalent SEU effects. For the purpose of this work, we use the SEU injector capabilities in the fault campaigns.

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

A golden simulation is performed in order to obtain the reference memory results and the performance parameters, which are later used in the classification stage. The fault control manager loads the fault list for the campaign. This fault list is composed of the signal locations where to inject each fault in the model. Depending on the fault model, additional parameters are required, such as the injection time and the injection period. The fault decoder-generator reads from the SEU fault list one fault at a time and generates an equivalent behavioral injection command for the ModelSim environment, using the *force–deposit* syntax.

Then, the control manager starts the fault injection campaign. The execution time limit is defined as twice the golden execution time of the program kernel. This value is employed in order to check performance degradation by the fault effect. Once the simulation finishes, the memory results and performance parameters are stored. Finally, the checker and classifier verify the generation of memory results. This classifier catalogues the fault effects in four categories: Silent Data Corruption (SDC), Time-Out (Performance Degradation), Hang (Detected Unrecoverable Error (DUE)) and Masked (Silent).

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

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

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

In order to evaluate the SEU effects on the FlexGrip model, four target modules were selected, commonly used by most of the kernel programs. It is worth noting that each module presents a different level of use and criticality in kernel execution.

B. *Target Modules*

Two control-path (SM scheduler and Divergence stack) and one data-path (Register Files) module of the FlexGrip were selected for SEU fault injection campaigns. The general description of each module is presented as following:

SM Scheduler: This module manages and controls thread execution in the available SPs inside the SM. This unit also tracks the actual status and thread operation. Furthermore, this is a critical unit during the GPGPU operation and a fault can generate misbehaviors generating hang or SDC effects. This module is selected by its criticality in thread execution and the sensitivity to permanent faults observed in other works [18]. For the purpose of this work, we target the warp pool memory in the scheduler injecting SEU faults in it and classifying its sensitivity to them. This memory stores crucial information for the warp execution.

Divergence Stack: This module stores the branching information related to convergence points (*addressing points*), generated by control-flow instructions during the execution of a program kernel. Each one of the 32 entry lines in the memory is composed of a program counter field (*convergence point*), a mask of active threads and the active Warp ID. Misbehaviors in this module may corrupt the thread execution flow, generating performance, hang and SDC effects. In a program kernel, a synchronism convergence points, *(SSY)* instruction, generate the pull to store the instruction pointer and the threads index. Once the convergence point is reached by the program, the information is pushed and the entry line is released.

Register Files: This data-path module is highly used by most kernel programs, since most instructions include operations with one or more registers. In Flexgrip, this unit is composed of 512 32-bits registers in a 32 SP-core configuration. For 16-SP cores there are 1,024 registers per register file. Finally, 2,048 registers are available in the 8-SP core configuration. On the other hand, the total number of registers employed by an application directly depends on the kernel instructions. In a register file, the registers per thread are assigned through dividing the total number of registers by the number of threads per block to be executed.

C. *Benchmarks Description*

Three representative benchmarks were selected to evaluate the incidence of SEU effects in the model. Every benchmark generates different stimuli over each targeted module. One kernel (*Vector_Add)* presents high data-intensive operations, mainly employing data-path modules and with a low incidence on control-path modules. Additionally, we considered two applications (*FFT*, *Edge-Detection-Sobel*) based on different combinations of control-flow strategies, divergence generation, and data operations. Each kernel program was developed, compiled in CUDA-C with SM_1.0 and adapted to the supported FlexGrip instruction set. Table 1 introduces the major operational features of the selected applications. A general description of each benchmark is presented as following:

FFT: The kernel description of the one-dimension FFT algorithm is based on the Cooley-Turkey algorithm [19] using the butterfly propagation algorithm. It was necessary to adapt the kernel description to the instructions supported in FlexGrip. The division instruction (not supported in hardware) was replaced by logarithmic division method based on shift and logical displacements. It is worth noting that the initial data set is ordered by the host to feed the FFT kernel. The same procedure is followed by the host to reorder the results.

Edge Detection: This algorithm is based on the convolution of an input image matrix and a 3x3-size stencil element, representing a 2D filter. Initially, the data matrix is divided according to the total number of thread configurations in the kernel and then the convolution is processed sequentially. The kernel description is adapted to the supported instructions of FlexGrip. The division instruction is replaced by a logarithmic method mentioned above. This application includes multiple loop, thread dependency execution, and dense arithmetical operations, which require an intensive use of register resources and introduces thread divergence in the kernel.

Vector_Add: This is a typical parallel and high data-intensive application based on the execution of concurrent additions over independent data sets from two input vectors. The result vector is stored in a free memory location. This application commonly uses data-path units and execution modules of the GPGPU. This benchmark is selected because most elaborated program kernels include parallel sections during their execution. Moreover, the lack of control-flow instructions can give clear information of the SEU effects on data-path units.

TABLE 1 MAIN BENCHMARK FEATURES UNDER A THREAD DISTRIBUTION OF 32 THREADS PER BLOCK AND 2 BLOCKS CONFIGURATION.

Benchmark	Code size (*Words*)	.SASS Instructions	Execution time *(cycles)*	Configuration (*SP cores*)
FFT	334	174	584,265	32
			777,555	16
			1,153,845	8
Edge	712	373	688,305	32
			905,525	16
			1,374,265	8
VectorAdd	18	12	28,565	32
			33,385	16
			42,785	8

IV. EXPERIMENTAL RESULTS

The fault campaigns setup is described. Sub-sections B and C D introduce results and discussions of SEU effects on each targeted module.

A. Experiment setup

In the SEU fault injection campaigns, two main elements are considered: the SEU location and the SEU injection time. The SEU location defines the fault universe and it is composed of the registers and memory elements employed by a benchmark during its execution on each targeted module. The faults were carefully checked and selected during the golden execution.

The SEU injection time considers the time intervals in the kernel execution on FlexGrip. Those time intervals are: configuration, execution, global-memory storage and kernel termination. The SEU injection range, by definition, does not consider kernel configuration times and memory storage times and must correspond to the execution interval only. One SEU injection time (i.e., one clock cycle) is selected randomly from the SEU injection range for each SEU location.

Fault campaigns were performed on the targeted modules considering different TLP configurations (8, 16 and 32 SP-cores) and different thread distributions (*A* and *B*). These thread distributions are defined depending on the number of threads per block. *A* configuration distributes every benchmark with 32 threads and two blocks per grid. In contrast, *B* configuration uses 64 threads per block and one block per grid.

The Architectural Vulnerability Factor (*AVF*) [20] is computed for each targeted module under all TLP and threads distributions and for the selected benchmarks. This metrics is calculated as the ratio between the total number of failures, affecting the simulation output, by the total number of SEU faults injected. The following sections present the results of the fault injection campaigns for every target module. The AVF results are presented in Table 2 for each targeted module.

TABLE 2 AVF RESULTS FOR THE TARGETED MODULES UNDER VARIOUS TLP AND THREAD CONFIGURATIONS.

Benchmark	FFT		EDGE		VectorAdd		SP-Cores
Thread Config	A	B	A	B	A	B	
Register file AVF (%)	37.03	29.49	35.80	26.61	17.0	25.0	32
	36.92	33.44	36.72	36.40	25.18	28.91	16
	36.89	18.87	36.87	19.81	27.19	29.98	8
Warp scheduler memory AVF(%)	0.10	0.27	0.15	0.15	0.12	0.20	32
	0.07	5.44	0.698	4.81	0.56	7.67	16
	0.10	16.21	2.51	19.21	2.27	15.63	8
Divergence stack AVF(%)	1.24	1.41	0.91	0.90	-	-	32
	1.5	1.66	1.21	1.22	-	-	16
	2.25	2.48	1.82	1.81	-	-	8

B. Register File Results

For this module 27 fault injection campaigns were performed, injecting SEUs in the register files under all configurations described below. The total number of faults injected is determined by establishing the total number of registers used by each application under each TLP configuration. Then, the total number of bit fields is multiplied by a constant in order to define the faults per location. 34,816 faults were injected for the *FFT* and *Edge Detection* applications under all SP configurations. In *VectorAdd,* 10,240 faults were injected for 32-SP cores and 8,192 faults for the 16 and 8-SP cores configurations.

The employed multi-thread simulation approach reduced a fault campaign of more than 150 hour to less than 16 hours. Besides, it also reduced the total amount of faults to inject in the campaign by up to 95%.

An initial analysis of the results shows that the *FFT* and *Edge* benchmarks present a similar behavior: by changing the TLP and increasing the number of threads per block (*B* configuration) reduces the total error rate (*AVF*). A different behavior is shown by the *Vector_Add* application. In this case, raising the number of threads per block (i.e., increasing the parallelism) generates a direct increase in the failure rate provoked by SEUs. Fig. 2 presents in more details of the results given in Table 1.

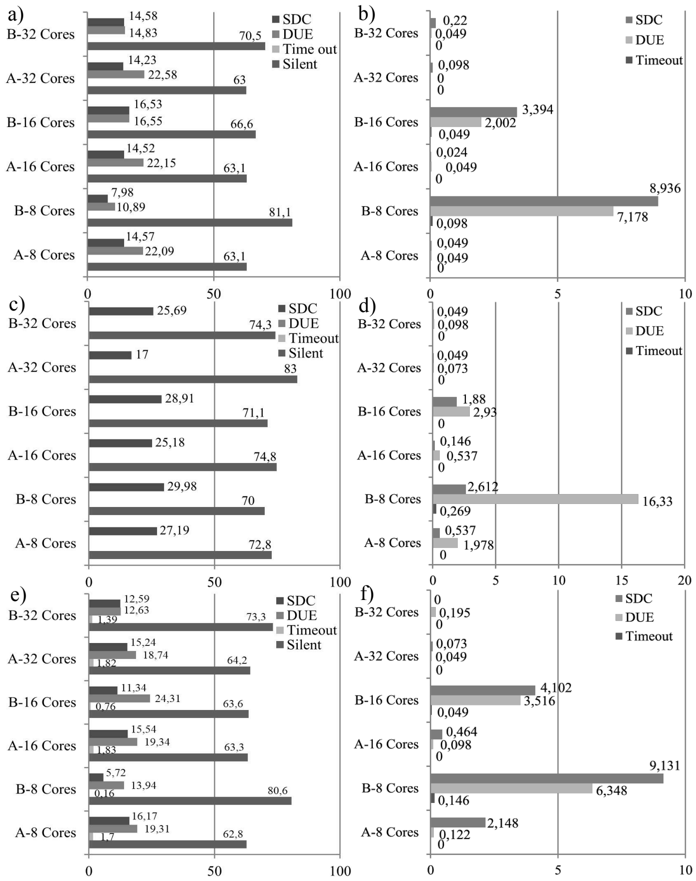

Fig. 2. Register File (a, c and e) and warp schedule memory (b, d and f) results for FFT (a, b), VectorAdd (c, d) and Edge (e, f) benchmarks.

The *FFT* results show a slight increment in the SDC error-rate when increasing the number of threads per block in the benchmarks. This behavior has a direct relation with the kernel execution time for each configuration. In principle, the data stored in an active register for a long time are more exposed to SEU effects (case *B*) than registers with multiple write and read activity (case *A*). In the experiments, the *A* configuration models required a longer execution time. Nevertheless, the effective block execution time is lower than for *B* configuration.

Moreover, *FFT* in the *A* configuration uses half of the registers of the *B* configuration and employs them to process threads, in different interval times, belonging to different blocks.

In this scenario, an increment in the number of threads per block should increase the SDC error rate, as it happens for the results with 32 and 16-SP cores. However, a detailed analysis of *FFT* instructions shows that these include control-flow instructions, which also depend on predicate conditions (*flags*). These predicate conditions are generated evaluating register results; it means that, some registers are involved in control-flow operations. Those registers can be considered as control-flow critical registers (CFRs).

If a SEU fault affects one of these CFRs, the effect on the application is DUE. According to results, a high number of CFRs is generated by decreasing the thread distribution or parallelism (*A* Configuration). This behavior can be explained considering the number of registers employed in *A* Configuration and the CFRs mapped among threads in the same register locations, which means that, during kernel execution, one register location will store, in different time intervals, data belonging to two CFRs, increasing the probability of DUE.

A different behavior is observed for the *Vector_Add* benchmark. In fact, an increment in the number of threads per block corresponds to an increase in the SDC error rate. This trend is consistent on each TLP configuration and can be explained by the increased SEU sensitivity by the additional time required by the warp scheduler to dispatch other warps from the same block. Moreover, the execution time to process an instruction under a long thread distribution (*B* Configuration) is the double of a block with fewer warps (*A* Configuration).

Additionally, the SEU effects slightly increase by reducing the TLP. In this case, the main factor is the additional time required by the scheduler to process an instruction of each thread with the limited number of SP cores. On the other hand, the number of faults producing DUE and Time-Out behaviors is zero because this kernel has no control-flow instructions.

In *Edge Detection* application, for each TLP configuration, the total number of SDC errors decreases when the number of threads per block increases. This behavior partially contradicts the results of *VectorAdd* and *FFT* benchmarks. Nevertheless, an explanation to this behavior can be found in the fact that *Edge Detection* kernel includes more control-flow, divergence generation and arithmetic-intense instructions than *FFT*. In fact, it is a longer and more elaborated application. Regarding the DUE error rate, results show an inverse relation between threads per block, in the kernel, and the number of DUE errors. The explanation is the same as for the *FFT* benchmark for DUE errors, which decrease due to CFRs. Moreover, these results (*Edge Detection* and *FFT*) are consistent with those introduced in [9] for applications with control-flow instructions.

C. Warp Scheduler memory results

Fig. 2 presents the error rate results for the warp scheduler memory. An initial results overview could contradict the criticality of this module in GPGPU operation. However, an analysis of its architectural organization and the role employed, during kernel execution, helps to clarify the behavior.

Comparing results and simulation execution, the low error rate is mainly caused by a closed loop between the scheduler and the pipelines stages, which contributes to mask and reduce the SEU effects in the memory. This is written each time a warp finished the execution of an instruction. Moreover, this memory allows the writing and reading process in a few clock cycles after the instruction finishes, thus reducing the error propagation.

Regarding threads distribution and parallelism, results show that increasing threads per block generates a direct rise in SDC and DUE errors. The kernel (in *B* Configuration) uses more lines in the warp memory and requires the execution of multiple warps to process one instruction. Moreover, warp line exchange is required to process all threads. This exchange generates a temporary short in the closed loop and if a SEU effect is present, affecting a loaded entry line, can be propagated into the system.

A reduction in TLP produces a direct increment in the error rate. This increment is due to the additional effort required by the scheduler to process the threads. In this case, the scheduler employs twice to four times writing and reading sequences on the warp memory to finish a warp instruction contributing to increase the error rate.

D. Divergence Stack Results

The fault injection campaigns on this module did not considered the *Vector_Add* application because this kernel does not employ the module. The *FFT* and *Edge Detection* benchmarks were evaluated employing 50,688 faults in each fault campaign. The results are presented in Fig. 3.

The divergence stack memory does not represent a major contribution to the error rate by SEU effects. The main explanation for this low error rate is the restrained usage during kernel execution. Each entry-line is employed for the fraction of a divergence generation, meaning a different SEU sensitivity per line. This behavior is directly dependent on kernel description, nesting divergence, total number of divergence path instructions, and the number of convergence points. Results support the previous explanation. Moreover, considering that usage of this unit, for both kernels, is less than two thirds of the total simulation time and each additional pushed line presents less activities, the sensitivity to SEU effects reduces drastically.

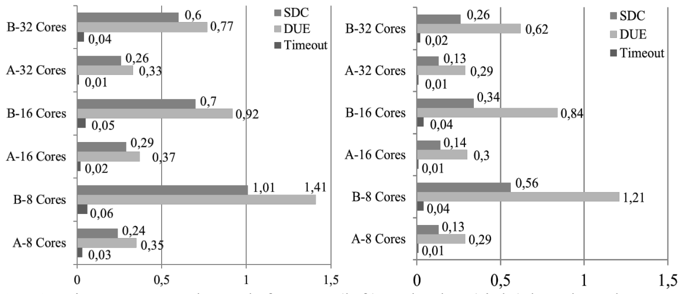

Fig. 3. Divergence stack result for FFT (left) and Edge (right) benchmarks.

The difference in error rate between both benchmarks can be found analyzing the kernel instructions, its description and the divergence paths length. The total number of *SSY* instructions plays an important role in the usage of each entry-line.

A detailed inspection to *FFT* instructions shows that it has two *SSY* operations and long divergence paths. On the other hand, the *Edge Detection* kernel employs seven *SSY* instructions with shorter path length than *FFT*. Besides, some paths are parameter dependent.

According to results, the behavior presented in *Edge Detection*, with multiple independent consecutive synchronization points, it seems to be less affected by SEU effects than a low number of divergence paths, which means a low number of writings in the stack line. In this case, the long interval time between writing and reading seems to increase the SEU sensitivity.

Regarding DUE and SDC error rate, it directly depends on the affected location. A SEU in the program counter may generate Timeout or DUE errors. Similarly, a SEU affecting the thread mask field may generate SDC, by inactive threads, or

DUE effects, by threads missing the taken path. Finally, a SEU in the warp ID field can generate Timeout effects. The difference in DUE and SDC error rate, for both applications, is mainly caused by the sensitivity of program counter and mask fields to generate hang conditions. The applications are prone to skip or lose the execution path by changes in these two fields instead of generating a SDC error.

Regarding the number of threads per block, coherent results were found across TLP configurations. Although, *A* Configuration employs the same number of entry lines in the divergence stack, these lines are employed in different time slots and the execution time per block is lower than required in *B* Configuration. The additional time presented in *B* Configuration seems to be the responsible for increasing the SEU sensitivity. A decrement in the number of threads per block could help to reduce, in more than twice, the SDC error rate generated by SEU effects. By comparison of the TLP configurations, it seems to follow the same SEU sensitivity to execution time as presented on the previous modules.

V. Conclusions

As a major contribution of this paper, we reported a detailed analysis of the effects of SEUs in a GPGPU resorting to an extended GPGPU model and to some realistic applications.

Two signal and image processing applications, *FFT* and *Edge Detection*, were designed and employed as benchmarks to evaluate the effects of SEU faults on several sub-modules of a GPGPU. Additionally, an application (*Vector_Add*) was designed to observe the SEU effects on data-intensive applications, which are commonly used in the practice.

The detailed GPGPU model description was crucial to explain the behavior observable in control unit modules under the considered low error rate. Despite the fact that the FlexGrip model does not exactly match the architecture of the most recent GPGPU devices, we still claim that the performed analysis remains valid for them as well considering that those modules are also employed and implemented in new GPGPU architectures.

In the Register File, a kernel divided in blocks reduces the SEU effects and increases error masking, considering that the same register set is used by independent threads belonging to different blocks. This behavior can be observed if two independent blocks are dispatched to the same SM, as it happens in FlexGrip.

Results suggest that the threads-per-block distribution may play a major role as a mitigation strategy for SEU effects for applications with a high usage of data-path units, such as the Registers Files. On the other side, this distribution seems not to impact in a significant manner on the targeted control units.

Analyzing the results for all modules, an increment in the number of threads per block seems to generate a higher SEU sensitivity on all data-path modules. This can be explained by the additional time to process all threads. Nevertheless, the final effect depends on the kernel behavior and the instructions employed in its implementation.

The SP-cores customization on FlexGrip is useful for energy and area optimization. Nevertheless, from the reliability point of view the SP reduction increases the SEU effects on each targeted module and hence reduces the reliability of the system.

We are currently working to extend the analysis of the SEU effects to other modules within the GPGPU architecture.

References

[1] W. Shi, M. B. Alawieh, X. Li, and H. Yu, "Algorithm and hardware implementation for visual perception system in autonomous vehicle: A survey," *Integration,* vol. 59, pp. 148-156, 2017/09/01/ 2017.

[2] E. Ibe, H. Taniguchi, Y. Yahagi, K. Shimbo, and T. Toba, "Impact of Scaling on Neutron-Induced Soft Error in SRAMs From a 250 nm to a 22 nm Design Rule," *IEEE Transactions on Electron Devices,* vol. 57, pp. 1527-1538, 2010.

[3] L. B. Gomez, F. Cappello, L. Carro, N. DeBardeleben, B. Fang, S. Gurumurthi, *et al.*, "GPGPUs: How to combine high computational power with high reliability," in *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2014, pp. 1-9.

[4] L. L. Pilla, P. Rech, F. Silvestri, C. Frost, P. O. A. Navaux, M. Sonza Reorda, *et al.*, "Software-Based Hardening Strategies for Neutron Sensitive FFT Algorithms on GPUs," *IEEE Transactions on Nuclear Science,* vol. 61, pp. 1874-1880, 2014.

[5] S. Collange, M. Daumas, D. Defour, and D. Parello, "Barra: A Parallel Functional Simulator for GPGPU," in *2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, 2010, pp. 351-360.

[6] J. Power, J. Hestness, M. S. Orr, M. D. Hill, and D. A. Wood, "gem5-gpu: A Heterogeneous CPU-GPU Simulator," *IEEE Computer Architecture Letters,* vol. 14, pp. 34-36, 2015.

[7] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, 2009, pp. 163-174.

[8] B. Fang, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi, "GPU-Qin: A methodology for evaluating the error resilience of GPGPU applications," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014, pp. 221-230.

[9] W. Nedel, F. L. Kastensmidt, and J. R. Azambuja, "Evaluating the effects of single event upsets in soft-core GPGPUs," in *Test Symposium (LATS), 2016 17th Latin-American*, 2016, pp. 93-98.

[10] M. Gonçalves, M. Saquetti, F. Kastensmidt, and J. R. Azambuja, "A low-level software-based fault tolerance approach to detect SEUs in GPUs' register files," *Microelectronics Reliability,* vol. 76-77, pp. 665-669, 2017/09/01/ 2017.

[11] P. Rech, G. Nazar, C. Frost, and L. Carro, "GPUs reliability dependence on degree of parallelism," *IEEE Transactions on Nuclear Science,* vol. 61, pp. 1755-1762, 2014.

[12] 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 *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, 2014, pp. 455-466.

[13] K. Andryc, M. Merchant, and R. Tessier, "FlexGrip: A soft GPGPU for FPGAs," in *2013 International Conference on Field-Programmable Technology (FPT)*, 2013, pp. 230-237.

[14] J. Knudsen, "Nangate 45nm Open Cell Library," *CDNLive, EMEA,* 2008.

[15] J. Guthoff and V. Sieh, "Combining software-implemented and simulation-based fault injection into a single fault injection method," in *Twenty-Fifth International Symposium on Fault-Tolerant Computing. Digest of Papers*, 1995, pp. 196-206.

[16] H. Ziade, R. A. Ayoubi, and R. Velazco, "A survey on fault injection techniques," *Int. Arab J. Inf. Technol.,* vol. 1, pp. 171-186, 2004.

[17] D. Alexandrescu, "Circuit and System Level Single-Event Effects Modeling and Simulation," in *Soft Errors in Modern Electronic Systems*, ed: Springer, 2011, pp. 103-140.

[18] B. Du, J. E. R. Condia, M. Sonza Reorda, and L. Sterpone, "About the functional test of the GPGPU scheduler," In On-Line Testing Symposium (IOLTS) 2018 IEEE 24th International, Platja d'Aro, Costa Brava, Spain, 2018.

[19] J. W. Cooley, P. A. W. Lewis, and P. D. Welch, "The Fast Fourier Transform and Its Applications," *IEEE Transactions on Education,* vol. 12, pp. 27-34, 1969.

[20] 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 *36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.*, 2003, pp. 29-40.