An open source embedded-GPGPU model for the accurate analysis and mitigation of SEU effects

B. Du, Josie E. Rodriguez Condia, M. Sonza Reorda, L. Sterpone
Politecnico di Torino, Torino, Italy

Abstract[1]**—In this paper, we propose a new hardware and synthesizable model of an embedded General Purpose Graphic Processing Unit (GPGPUs) designed for analyzing and mitigating radiation effects. Comparative SEU injection experiments confirms the model effectiveness.**

Keywords— SEU, General Purpose Graphics Processing Units GPGPUs, Graphics Processors, Fault Injection.

I. Introduction

General Purpose Graphic Processing Units (GPGPUs) are effective solutions in data-intensive applications, such as multi-signal analysis, and video processing, thanks to their parallel architecture. Nowadays, these devices are promising alternatives for embedded real-time and safety-critical systems. In the automotive field, these devices are commonly adapted in sensor fusion systems and Advanced Driver-Assistance Systems (ADAS) [1], which form specialized systems targeting complex applications, including Automatic Parking and Cruise Control, Pedestrian and Pattern Recognition, and Forward Collision Warning.

New GPGPU technologies include more parallel execution cores or streaming multiprocessors (SM) in NVIDIA's terminology increasing the performance and the throughput. Nevertheless, these new devices employ the latest technology scaling approaches. Moreover, it is well known that these 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. The SEUs may cause unexpected behaviors and unacceptable erroneous operations in safety-critical applications.

In the past, some mitigation techniques were proposed to reduce the error effect including the adoption of special software coding strategy and the algorithm implementation [3]. A detailed analysis is required to evaluate the radiation effect in the application, such as error rate and effect on critical units, and to select the most suitable countermeasures. Moreover, these methods are also employed to provide guidelines for effectively writing the application code, trading off performance and reliability.

Fault injection on device models, at different abstraction levels, are common solutions to support the effect analysis and to select mitigation strategies. Nevertheless, in the GPGPU field, there are some challenges. There are a few available models of GPGPU, and most of them are described at a high level of abstraction [4-7], thus preventing a detailed analysis of radiation effects. Moreover, there are a few synthesizable RTL behavioral GPGPU models, which can be used to analyze the SEU effects on different abstraction levels [8][9][10]. In [11] the authors introduced a SEU fault simulation injection methodology applied to a behavioral/RTL GPGPU device. Moreover, the previous [12] and other works preformed transient faults experiments and provided an initial overview of the effects on data-path units of a GPGPU, such as the register file and the pipeline registers. Conclusions show that the error rate in these modules is directly related with the employed benchmark and the parallelization level in the application (e.g., in terms of thread distribution).

On the other hand, radiation experiment is an alternative technique to analysis the SEU effects. The main advantage of this method is the usage of real devices in the experiments instead of a representative model.

In [13][14] 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 and detailed explanations about the observed behaviors in particular modules, since internal structure details and behavior of the device are not fully available.

In this work, we introduce a set of improvements to the original FlexGrip GPGPU model in order to make it suitable for radiation effect analysis. Furthermore, we performed a comparative fault injection experiment using the SASSIFI approach [15] on a Nvidia Pascal with 256 CUDA cores embedded in the Jetson TX2 embedded GPGPU kit. The comparative analysis demonstrated the similarities of the behaviors thus the effectiveness of our model for the study of radiation induced SEU on GPGPU architectures.

II. Background

FlexGrip is an open source model of a GPGPU described in VHDL. This model was developed by the University of Massachusetts and originally optimized targeting a Xilinx FPGA [8]. The model implements the Nvidia G80 micro-architecture and it is also compatible with Nvidia's CUDA environment under SM_1.0 compatibility. FlexGrip employs a compiled CUDA-binary code (.SASS file) as program kernel. 27 instructions of either 32 or 64 bits are supported by FlexGrip. The kernel configuration parameters, such as Grid dimension, Block dimension and Blocks per core, and the memory content in the constant memory and other GPGPU configuration parameters, such as the number of registers per

[1] This work has been partially supported by the European Commission through the Horizon 2020 RESCUE-ETN project under grant 722325.

thread and the number of blocks per SM core, should be manually defined for each application before execution.

The micro-architecture is based on the Single Instruction Multiple Thread (SIMT) 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.

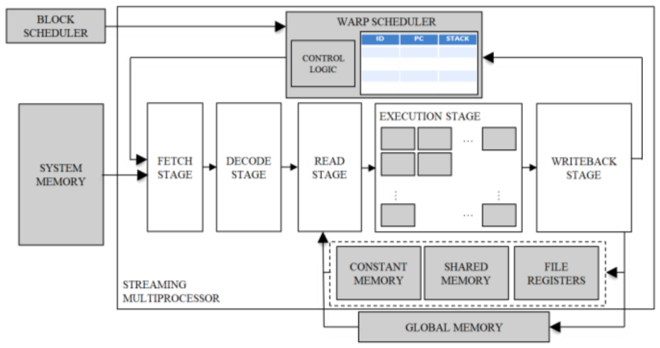

Fig. 1. FlexGrip architecture: the SM.

The execution units (EU) in FlexGrip are able to process only integer operations. Moreover, the number of EU in the system also defines the Thread-level parallelism (TLP) of the GPGPU. The model supports the configuration of 32, 16 and 8 SPs. A (branch unit) module, in parallel to the EU, is able to support inter-warp branching at hardware level. This module manages the control-flow operations in order to start or retake the flow from conditional branches with multiple paths. This unit also support up to 32 levels of nesting branching.

III. GPGPU MODEL FOR RADIATION-EFFECT ANALYSIS

A detailed analysis in the model and to each internal module denoted some operational limitations. Thus, we performed a set of improvements, which allow us to analyze transient fault effects on internal modules. Moreover, these improvements simplify and increase the flexibility of the model for applications development and multi-technology implementation. The improvements can be divided in four categories: technology dependency, instruction format support, compilation restrictions and hierarchical mapping.

FlexGrip was originally designed to be implemented on specific FPGA technologies. Moreover, some internal modules were automatically generated employing high level compilation tools, such as Matlab and Xilinx Vivado. Thus, multiple IP cores are included in multiple parts of the design. Nevertheless, these codes are not easily understandable and cannot be analyzed in an easy manner.

We modified each module by removing any reference or dependency to specific technology libraries and compilation tools and replacing them with equivalent generic implementation. Moreover, the name of signals and interconnections was clarified in order to simplify the analysis during the fault injection campaigns. At the end, 38.8% of the modules were corrected or modified for this purpose. The model can now be imported in simulation environments, such as ModelSim.

Considering the instruction format, FlexGrip was designed to be compatible with the CUDA programing environment and execute SASS instructions. Nevertheless, the use of high-level Electronic Design Automation (EDA) tools during design and its optimizations seems to be one of the factors for some missing instruction formats. Thus, some internal parts in modules, such as intermediate registers, decoding logic, and interconnections were removed by the optimization.

The previous behavior was checked during the development of custom applications employing the CUDA environment. In these applications, some instructions failed during execution. Exhaustive analysis and revisions were performed on the simulation traces. However, in some cases, the analyzed signals behavior showed that some supported instructions were only partially implemented. This restriction limits the transient fault analysis and its incidence under different applications. Moreover, it reduces the model’s flexibility and capability. The improvement reported here required a methodical revision of all supported assembly instructions (SASS) in the model and the addition or correction of the missing implementation for the instructions under the expected format. As a result, we identified a minimal subset of instructions required to implement some basic applications, and focused our work on fixing existing bugs, thus allowing a complete support of these instructions.

As the SASS Instruction Set Architecture (ISA) has not been released by Nvidia, the opcode format of some instructions was decoded employing the CUDA compilation tools (NVCC and CUOBJDUMP). Multiple applications were designed targeting selected instructions in order to force the compiler to generate the instruction opcodes. Then, all the required changes (e.g., missing registers, connections or incomplete modules) were introduced in order to fully support the selected minimal set of instructions with all the potential instruction format variations. After this process, the set included 27 instructions and 74 formats.

The 4.8% of the whole model description required an addition or modification in its implementation in order to be able of execute the expected instruction formats and its variations. Finally, some bugs and unused interconnections were removed from the project hierarchy in order to clean the modules and remove any redundant logic, which may create problems during the fault campaigns.

The compilation has been extended to full CUDA compatibility. FlexGrip is able to execute applications compiled employing the CUDA-toolkit by Nvidia. Moreover, a SM 1.0 micro-architectural compatibility must be selected. However, the CUDA compiler is protected and, as commented before, the opcode of the instructions is not released. In multiple attempts to design new applications for FlexGrip we discovered several SASS instructions not supported by the model, so in order to maintain the compatibility with the CUDA-toolkit, a SASS checker tool was developed to check the supported SASS instruction formats. This tool is able to identify and notify the user of those unsupported instructions formats in FlexGrip. Additionally, a SASS parser tool was designed to directly write SASS assembly instructions and replace the unsupported ones. Using both tools, a new application can be designed, verified and corrected without the necessity of debugging the instructions in the model, thus reducing the application development time. With this, we implemented a version of FFT and Edge Detection algorithm as benchmark applications.

Finally, the designed model has been implemented considering layout constraints. Each module of the SM is associated to a specific placement constraint file that allows the effective mapping and placement on isolated areas on the considered design. This will permit the adoption of reconfiguration based SEU injection or effective bitstream evaluation.

IV. Comparative Fault Injection Analysis

We performed two fault injection experiments in order to evaluate the effects of SEU in the registers of GPGPUs. The former has been implemented on the developed GPGPU model. We developed a custom fault injector to identify and analyze the SEU effects on specific internal modules of the SM. This fault simulator is based on the ModelSim framework. The latter, we adopted the SASSIFI fault injection approach developed by Nvidia to perform the same type of fault injection but executed on the Nvidia Pascal embedded GPGPU embedded on the Jetson TX2 board.

The fault simulator was designed to perform a set of SEU fault campaigns on the improved version of FlexGrip and follows the fault injection methodology introduced in [11] and to compare the fault injection experiments with the ones performed using SASSIFI on the Pascal GPGPU. Moreover, it includes a multi-thread approach [17] [18] and the utilization of a de-rating factor (UDR) [19] of the targeted modules. The UDR factor considers only the registers employed by an application during execution time, thus reducing the total amount of faults to be injected and the corresponding simulation time in the fault campaign. The fault simulator injects SEUs in memory cells or register signals by flipping its value. For the purpose of this work, we use the SEU injector capabilities in the fault injection campaigns. A fault simulation starts with (compiling and) loading the GPGPU model in ModelSim. In this process, the control manager loads the GPGPU configuration, the application instructions and the initial data memory values. The user provides the kernel instructions and the model configuration before the fault simulation starts. The fault list generated by the fault simulator is also saved in a specific file in order to be used on the Pascal with SASSIFI. Finally, a classifier sorts 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 Silent classification is used if there are not mismatches in memory results or execution time. It is worth noting that one simulation is performed for each considered fault.

On the Pascal embedded GPGPU, we instrumented the benchmark applications with the SASSIFI approach considering the fault list generated by the fault simulator targeting on the proposed model. Since the registers used by the compiled applications on the Pascal were different from the Flex Grip, we relocated them in order to achieve the same statistic of the fault locations.

V. Experimental Results

We implemented three different applications. *Vector_Add* kernel presents high data-intensive operations, thus each operand should use the RF for temporary storage. On the other hand, other two applications (*FFT, Edge-Detection-Sobel*) are based on different combinations of multiple path execution (divergence) and data operations. Each kernel program was developed, compiled in CUDA-C with SM_1.0 and adapted to the supported FlexGrip instruction set. The same CUDA-C codes have been compiled with compatibility 3.0 for Pascal. Table I introduces the major operation features of the selected applications. Please note that the execution cycles of the Pascal GPGPU, settled to execute the code on 32 cores, are around 15% shorter than FlexGrip benchmark code. This is due to two aspects: the former is the optimized CUDA assembly obtained by the 3.0 compiler versus the 1.0, the latter can be related to the small difference of instruction set used for FlexGrip that results in longer application code.

Table I. Benchmark Application Characteristics on FlexGrip and Pascal GPGPUs.

	Benchmark	**Code size (*Words*)**	**.SASS Instructions**	**Execution time *(cycles)***	**Configuration (*SP cores*)**
FlexGrip	FFT	334	174	584,265	32
	Edge Det.	712	373	688,305	32
	VectorAdd	18	12	28,565	32
Pascal	FFT	282	136	496,452	32
	Edge Det.	656	327	585,021	32
	VectorAdd	16	10	25,534	32

During experiments, two main elements were considered: the location and the injection time. The SEU location is composed of the registers and memory elements employed by a benchmark during its execution. The locations were carefully checked and selected during the golden execution and were used to instrument the SASSIFI fault injection on the Pascal GPGPU. 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 and memory storage but only corresponds to the execution interval.

In the fault campaign, one SEU injection time (i.e., one clock cycle) is selected randomly from the SEU injection range for each SEU location. The fault campaigns were performed on the targeted modules considering a TLP configuration of 32 SP cores and two thread distribution configurations: *Config. A* distributes every benchmark with 32 threads and two blocks per grid; *Config. B* uses 64 threads per block and one block per grid.

The fault injection results demonstrated similarity between the FlexGrip GPGPU and the Pascal embedded GPGPU. 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 (Config. B) than registers with multiple write and read activities (Config. A). In the experiments, the *Config. A* required a longer execution time. Nevertheless, the effective block execution time is lower than for *Config. B*. Moreover, FFT in the *Config. A* uses half of the registers of the *Config. B* and employs them to process threads, in different interval times, belonging to different blocks.

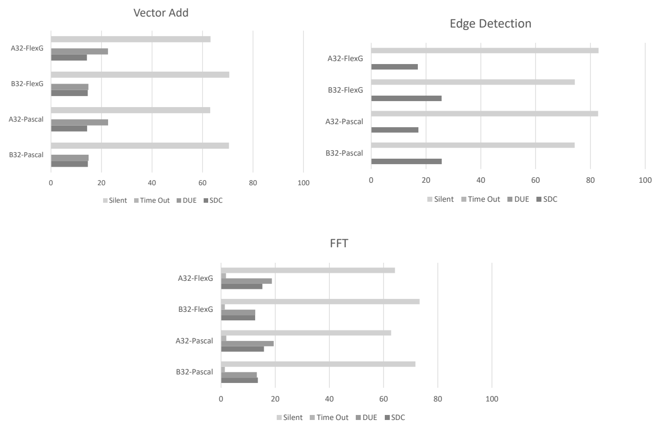

Fig. 2. SEU injection on FlexGrip model and Pascal embedded GPGPU.

VI. Conclusions and Future Works

In this paper, we presented a hardware model of an embedded GPGPU with an extended ISA architecture able to provide compatible results with modern GPGPU architectures. Despite the fact that the developed extended FlexGrip model does not exactly match the architecture of the most recent GPGPU devices we were able to demonstrate its similar behavior regarding SEUs within user register resources. The performed SEU fault injection experiments executed with the simulation environment on the FlexGrip and with SASSIFI on the Pascal GPGPU provide fully compatible results. Three benchmark applications have been compiled with the respective architecture.

As future works, we plan to implement the enhanced FlexGrip model on Flash-based FPGA in order to have a similar technology behavior and study the influence of SETs and SEUs during a radiation test campaign.

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. L. Pilla, P. Rech, F. Silvestri, C. Frost, P. O. A. Navaux, M. S. 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] M. A. Kadi, B. Janssen, and M. Huebner, "FGPU: An SIMT-Architecture for FPGAs," presented at the Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, California, USA, 2016.

[10] R. Balasubramanian, V. Gangadhar, Z. Guo, C. H. Ho, C. Joseph, J. Menon, *et al.*, "MIAOW - An open source RTL implementation of a GPGPU," in *2015 IEEE Symposium in Low-Power and High-Speed Chips (COOL CHIPS XVIII)*, 2015, pp. 1-3.

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

[12] 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.

[13] 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.

[14] 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.

[15] 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," 2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Santa Rosa, CA, 2017, pp. 249-258.

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

[17] 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.

[18] 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.

[19] 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.