An extended GPGPU model to support detailed reliability analysis

Josie E. Rodriguez Condia†, Matteo Sonza Reorda‡,
Politecnico di Torino, Dept. of Control and Computer Engineering, Torino, Italy
{†josie.rodriguez, ‡matteo.sonzareorda}@polito.it

Abstract[1]—**General Purpose Graphics Processing Units (GPGPUs) have been used in the last decade as accelerators in high demanding data processing applications, such as multimedia processing and high-performance computing. Nowadays, these devices are becoming popular even in safety-critical applications, such as autonomous and semi-autonomous vehicles. However, these devices can suffer the effects of transient faults, such as those produced by radiation effects. These effects can be represented in the system as Single Event Upsets (SEUs) and are able to generate intolerable application misbehaviors in safety-critical environments. In this work, we extended the capabilities of an open-source VHDL GPGPU model (FlexGrip) in order to study and analyze in a much more detailed manner the effects of SEUs in some critical modules within a GPGPU. Simulation results showed that the scheduler controller has different levels of SEU sensibility depending on the affected location. Moreover, a reduced number of execution units, in the GPGPU can decrease the system reliability.**

Keywords—***Fault simulation, Functional safety, GPGPUs, SEUs SEUs, Transient faults.***

I. INTRODUCTION

In the last decade, GPGPUs have been used as accelerators in highly demanding data processing applications including multimedia processing and high-performance computing. Nowadays, these devices are increasingly adopted in several data-intensive safety-critical applications, such as autonomous and semi-autonomous cars [1]. These devices are manufactured employing aggressive technology scaling techniques in order to satisfy performance and energy requirements. Nevertheless, some studies have shown that these advanced semiconductor technologies are prone to suffer from external transient radiation effects [2-5]. These effects can be represented as Single Event Upsets (SEUs) and may generate intolerable misbehaviors in safety-critical environments.

In real devices, the impact of SEU effects is analyzed through radiation experiments in special facilities using complex and expensive equipment. Other methods include programming environments to inject soft-errors in the application code [6]. However, in both cases, detailed structural information about the device architecture and implementation are, commonly, unknown and detailed analysis of the fault effects is complex to perform. Moreover, the injection tools are helpful in targeting data-path modules, but these cannot inject faults in most control-path units.

In both cases, results are employed to assess device reliability or to identify structural or application weaknesses. Moreover, results are also employed to design mitigation strategies [7]. Potential solutions may include acting on the program coding style and on the application algorithm [8].

A detailed analysis could be crucial to choose the most suitable countermeasures to achieve given reliability and can provide some guidelines in the application development. Moreover, it contributes to identifying critical modules and the incidence of faults on the application failure rate.

Solutions to perform those analyses are based on fault injection via simulation on representative models at various abstraction levels. In the GPGPUs field, there are relatively few available models and fault injectors. Moreover, most of them are described using a high abstraction level [9-14] or a mix of them [15], thus foiling a complete and detailed analysis of SEU effects on complex units such as control-path modules. On the other hand, there are a few Register-Transfer-Level (RTL) behavioral GPGPU models, such as and FGPU [16] and FlexGrip [17], which can be used to analyze the SEU effects in these special-purpose modules. Unfortunately, FGPU was designed using a new microarchitecture and it is not closely related to commercial devices, thus limiting the analysis conclusions in real devices. On the other hand, FlexGrip implements a commercial microarchitecture with some technology dependency restrictions. Moreover, this model has a limited set of supported instructions, thus limiting the development of new applications which could support the detailed analysis mentioned above.

In the work reported in this paper, we first performed a detailed analysis of the FlexGrip model in order to remove some of these limitations and bugs. Moreover, we developed a new release version which has no direct dependency on a technology platform and is able to execute an increased set of instruction formats compatible with commercial compilers.

Using the new model, some representative applications were designed and used as benchmarks for SEU fault injection campaigns. Finally, results were analyzed describing the effect of SEUs in some critical data-path and control-path modules. Fault campaigns employed multiple application parameters and GPGPU configuration modes.

The paper is organized as follows: Section II summarizes the FlexGrip model and the improvements introduced in its new version. Section III presents the fault injection methodology, the

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

targeted modules, and the selected benchmarks. Section IV reports some experimental results, and Section V finally draws some conclusions and future works.

II. FLEXGRIP GPGPU MODEL

A. *FlexGrip architecture*

FlexGrip is an open source soft-GPGPU model described in VHDL developed by the University of Massachusetts [17] employing the Nvidia's G80 microarchitecture. This model is compatible with the CUDA programming environment under the 1.0 architecture. 27 instructions are supported by the model. The model was originally designed to be synthesized for Xilinx FPGAs platforms.

This GPGPU model is based on a Streaming Multiprocessor (SM) including a memory system and two schedulers (*Block* and *Warp*). The block scheduler is employed to manage and distribute the block tasks among the SMs. The Warp scheduler is used to control the execution of the group of 32 threads tasks denoted as *warp*. Both schedulers employ a round-robin algorithm. The SM is composed of five pipeline stages (*Fetch*, *Decode*, *Read*, *Execute* and *Write-back)* to process warp instructions, see Fig. 1. The total number of execution units (Scalar Processors, or SPs) in the *execution* stage is selectable before synthesis and can be used to select the best performance and power consumption of the GPGPU. The SP programmability can be selected among 8, 16 and 32 cores.

The procedure to execute a warp instruction on the SM starts when the warp scheduler selects one available warp and dispatch one instruction address to the fetch stage. The *Fetch* stage processes the address and finds the equivalent instruction. The *Decode* stage interprets the instruction formats and selects the required execution units and memory operands. The *Read* stage loads from the memory system the required operands. Then, the *Execution* stage processes the warp instructions employing parallel execution units and temporary registers for each thread. The *Write-back* stage stores the results in registers or memory locations. Finally, a new instruction is dispatched by the warp scheduler.

This model includes a custom branch management unit for thread (*intra-warp*) divergence. This module is composed of a control unit and a divergence stack memory to store the addresses of warp convergence points. This model supports up to 32 levels of divergence.

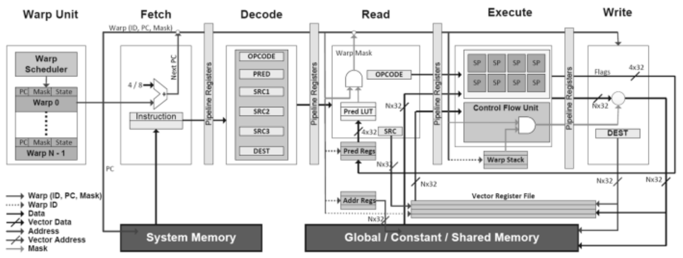

Fig 1. The general scheme of the SM in the FlexGrip model. Adapted from [17].

Although the FlexGrip model implements the NVIDIA's G80 architecture, this architecture includes the micro-architectural basics and modules which are still present in modern GPGPU architectures, such as basic thread and block management methods, integer execution units, memories hierarchy, intra-warp divergence, and pipeline stages. Besides, the supported instructions are basically executed in the same fashion on modern architectures, thus faulty analysis of this model may have similarities with modern GPGPU devices.

B. *Improvements in the FlexGrip GPGPU model*

The improvements we introduced in the model allow us to analyze transient fault effects on internal modules. Moreover, these modifications simplify and increase the flexibility of the model in the development of new applications.

The original GPGPU model was designed to be implemented in a particular technology and execute a predefined set of applications. Nevertheless, a detailed analysis of each internal module showed some limitations. Moreover, the development of new applications revealed some restrictions in the supported instruction formats. The improvements we introduced can be divided into three groups:

- Technology dependency
- Instruction format support and interconnection among sub-modules
- Compiler restrictions.

1) *Technology dependency*

FlexGrip was originally designed for FPGA implementation on specific technologies. Moreover, some internal modules were automatically described employing high-level compilation tools, such as Matlab and Vivado. However, these descriptions are not easily understandable and cannot be analyzed in an easy manner.

We modified each module by removing any reference to specific technology libraries and compilation tools dependency and replacing them with equivalent generic descriptions. Moreover, the name of the signals and interconnections was clarified in order to simplify the analysis during the fault campaigns. In the end, 38.8% of the modules were corrected or modified in order to remove the technology dependency. The model can now be imported in model simulation environments, such as ModelSim. Moreover, this can be synthesized employing other technology libraries, such as the ASIC OpenCell [18] library.

2) *Instruction format support*

FlexGrip design was compiled employing some high-level Electronic design automation (EDA) tools and some unused instruction formats were removed. Thus, internal modules, such as intermediate registers, decoding logic, and interconnections were not fully described by optimizations in those tools. These optimizations reduce the model flexibility limiting the development of new applications.

The previous behavior was checked during the development of custom applications employing the CUDA-toolkit environment. In the new 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, those restrictions limit the model flexibility and its potential employability.

The improvement reported here required a methodical revision of all supported assembly instructions (SASS) in FlexGrip and the addition or correction of the missing description to implement the instruction under the expected format.

As the SASS op-code, i.e., the instruction formats for the GPGPU, has not been released by Nvidia, the op-code format of some instructions was decoded employing the CUDA compilation tools (*NVCC* and *CUOBJDUMP*) through design of multiple applications targeting the selected instruction in order to force the compiler to generate the expected instruction op-code. Then, the missing register, connections or incomplete modules were carefully corrected in order to support all the potential instruction format variations. After this process, the 27 supported instructions were revised and 74 formats are supported.

Table 1, 2 and 3 introduce the arithmetic and logic instructions, the data handling, and memory instructions and the control-flow instructions, respectively. In Table 1, COMP_TYPE refers to comparison type and it depends on the predicate flag generated by an arithmetic or logic operation. In Table 3, COND parameters refer to predicate conditions. *g[]* and *c[0x1][]* correspond to shared memory and constant memory locations, respectively.

TABLE 1. SUPPORTED ARITHMETIC AND LOGIC SASS INSTRUCTIONS.

Mnemonic	Description	The revised format in the improved version
I2I	Integer to integer conversion	I2I.U32.U16/**S16** RZ, RX(L|H) / **g[].U16** I2I.U32.S32 RZ, |RX| / **-RX** I2I.U32.U16.BEXT RZ, RX(L|H) / **g[].U8** I2I.S32.S16.BEXT RZ, RX(L|H) / **g[].S8**
IMUL/	Integer multiplication	IMUL.U16.U16 RZ, RX(L|H) / **g[].U16**, RY(L|H) IMUL.S16.S16 RZ, RX(L|H) / **g[].S16**, RY(L|H)
IMUL32/		IMUL32.U16.U16 RZ, RX(L|H) / **g[].U16**, RY(L|H)
IMUL32I		IMUL32I.U16.U16 RZ, RX(L|H), Imm IMUL32I.S16.S16 RZ, RX(L|H), Imm
SHL	Shift left	SHL RZ, RX, RY / **Imm** SHL RZ, g [], Imm SHL.U16 RZ(L|H), RX(L|H), Imm
SHR	Shift right	SHR.S32 RZ, RX, RY / **Imm** SHR.S32 RZ, g [], Imm SHR.U16 / **S16** RZ(L|H), RX(L|H), Imm SHR RZ, g[], Imm SHR RZ, RX, RY / **Imm**
IADD/	Integer add	IADD RZ, RX / **-RX**, RY IADD RZ, g[], RX / **-RX** IADD RZ, RX, c[0x1][]
IADD32/		IADD32 RZ, RX, RY / **-RY** IADD32 RZ, g [0x..], RX / **-RX** IADD32.U16 RZ(L|H), RX(L|H), RY(L|H) / **-RY(L|H)**
IADD32I		IADD32I RZ, RX / **-RX**, Imm IADD32I RZ, g[], Imm
IMAD/	Integer multiply and Add	IMAD.U16/ **S16** RZ, RX(L|H), RY(L|H), RW IMAD.U16/ **S16** RZ, RX(L|H), c[0x1][], RY IMAD. RZ, RX(L|H), c[0x1][], RY
IMAD32/		IMAD32.U16 RZ, RXL|H, RYL|H, RZ
IMAD32I		IMAD32I.U16/ **S16** RZ, RX(L|H), Imm, RZ
LOP	Bitwise logical Operation	LOP.AND/**OR**/XOR/**PASS_B** RZ, RX/ **g[]**, RY LOP.AND/**OR**/XOR/**PASS_B** RZ, RX, c[0x1] [] LOP.U16.AND/**OR**/XOR/**PASS_B** RZ(L|H), RX(L|H), RY(L|H)
ISET	Integer comparison	ISET RZ, RX, RY / **c[0x1][]**, COMP_TYPE ISET RZ, g[], RX, COMP_TYPE ISET.S32 RZ, RX, RY / **c[0x1][]**, COMP_TYPE ISET.S32 RZ, g[], RX, COMP_TYPE

TABLE 2. SUPPORTED DATA HANDLING AND MEMORY SASS INSTRUCTIONS.

Mnemonic	Description	The revised format in the improved version
MVC	Load from constant memory	MVC RX, c [0x1] []
GLD	Load from global memory	GLD.U32|U16|S16|U8|S8 RZ, global14[]
GST	Store to global Memory	GST.U32|U16|S16|U8|S8 global14[], RX
MOV/	Move register to register/load from shared memory	MOV RZ, RX / **g[]** MOV.U16 RZ(L|H), RX(L|H) / **g[].(U16|U8)**
MOV32		MOV32 RZ, RX / **g[]** MOV32.U16 RZ(L|H), RX(L|H)
MVI	Move immediate to destination	MVI RX, Imm
R2G	Store to shared Memory	R2G.U32.U32 g [], RX R2G.U16.U16 g [], RXL|H R2G.U16.U8 g [], RX
R2A	Move general purpose register to address register	R2A AX, RX
A2R	Move address register to general purpose register	A2R RX, AX

4.8% of the whole model description required an addition or modification in its description to execute the expected instruction formats and its variation. 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.

TABLE 3. SUPPORTED CONTROL-FLOW SASS INSTRUCTIONS.

Mnemonic	Description	The revised format in the improved version
BRA	Branch	BRA CX.COND Imm BRA Imm
BAR	barrier synchronization	BAR.ARV.WAIT b0, 0xFFF
RET	Return from kernel	RET RET CX.COND
SSY	Set synchronization point	SSY Imm
NOP	No operation	NOP NOP.S

3) *Compiler restrictions*

FlexGrip is able to execute applications compiled employing the CUDA-toolkit by NVIDIA. Moreover, an SM 1.0 micro-architectural compatibility must be selected. However, the CUDA compiler is protected and, as commented below, the op-code 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, presented in Tables 1, 2 and 3. This tool is able to identify and notify the user of those unsupported instructions formats in FlexGrip. Most of them are generated by the compiler when the kernel includes arithmetical operations (*division*, *transcendental*, and *format conversions*), control-flow instructions (*call* and *return* from subroutines, and *conditional breaks*) and S2R (special register to register movement) instructions.

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 need of executing the instructions in the model, thus reducing the application time development. Sub-section III.C introduces three benchmarks developed for FlexGrip employing these tools.

III. FAULT INJECTION METHODOLOGY

In order to evaluate the effect of SEUs in the improved version of the FlexGrip model, we developed a fault injection tool employing the ModelSim framework. The injector tool was designed following the guidelines introduced in [19] regarding transient fault injection in behavioral models using the simulator commands. Additionally, the tool implements techniques to reduce the fault simulation time (multi-thread fault simulation and module de-rating factor (UDR) usage). Details about these techniques can be found in [20, 21].

The fault injector was developed employing a high-level language (*Python*) and is composed of a fault controller, a fault injector and a fault checker and classifier. The fault controller manages the fault campaign execution and it is able to start and finish the tool execution.

Initially, the fault controller configures the program kernel parameters, loads the FlexGrip model into the ModelSim environment and the program instructions to be executed. The kernel parameters include the number of SP-Cores presented in the SM, the total number of blocks and threads in the task, the total number of blocks per SM and the file register size.

Once the model is loaded in the simulator, the fault controller starts the fault injector and this loads and decodes the fault to be injected into the GPGPU model. The fault injector reads, from a fault list, the location and the injection time of the fault. Then, the injector translates those parameters into the equivalent commands for ModelSim. The tool is able to handle permanent and transient faults. For the purpose of this work, we employed the transient fault capabilities of the designed tool. One fault simulation is performed for each element in the fault list, once the fault is injected in the model.

Finally, the fault checker and classifier waits for simulation termination and checks memory results and simulation time parameters in order to classify the effects of the fault in the system. This unit classifies the faults in four categories: *Silent Data Corruption* (SDC) when the SEU affects the memory results, *Detected Unrecoverable Error* (DUE) when the model is hanged by the SEU effect, *Timeout* when the SEU produces performance degradation in simulation time and *Silent* when the SEU does not generates any effect.

A. Fault campaign description

In SEU fault injection campaigns, two elements are considered: the SEU location and the SEU injection time. The SEU location depends on the fault universe and spans over the registers and memory elements employed by a benchmark during execution time on each targeted module. The fault universe was carefully checked and selected through a golden execution. The injection time for each fault is randomly chosen.

A fault campaign starts with a golden simulation to define the reference execution time and the reference memory results. Then, the fault controller starts a loop in which this unit loads the fault list and the fault injector applies the equivalent command in the simulation model. The simulation time is selected as twice the reference execution time in order to allow the tool to detect timeout effects. Moreover, the model is instrumented with a memory generator which stores the memory results into a file for each simulation. The fault checker checks the presence of this file and performs the classification phase. Finally, a new fault is loaded for the fault list and the simulation loop starts again. The fault injection campaign finishes when the fault list is empty.

The multi-thread fault injection approach is employed in the tool by dividing the fault list in chunks of faults. Each fault list is composed of the SEU fault location *(signal name*) and the SEU injection time.

B. Targeted modules

One Data-Path and two Control-Path modules were targeted during fault campaigns. Their characteristics are briefly described in the following:

1) Data-path module

File Registers: The 32 bit-size registers are employed as source and destination operands and addresses during a warp instruction execution. These registers are organized and distributed according to the total number of warps and blocks to be executed in an SM.

2) Control Path modules

SM Warp Scheduler: The warp scheduler manages the warp execution inside an SM. This unit is able to select an available warp, dispatch the warp instruction to the SM and check its execution. This module is composed of various memories and control logic. The internal warp memory is employed to store the status information of each warp execution. This information is updated after each instruction execution and is composed of the active thread mask (aTM), the actual program counter and some additional warp configuration parameters.

Divergence Stack memory: This unit stores the divergence addresses generated by a divergent warp. A special-purpose memory stores the address, warp index and aTM to trace the number of executed threads on each divergence path.

C. Benchmarks

Three applications were developed for the improved version of FlexGrip to evaluate the SEU effects on the targeted modules. They are briefly described in the following.

FFT: This typical signal processing application was implemented based on the Coley-Turkey algorithm [22]. In this application, the butterfly element was described employing the CUDA-C environment. Although the model does not provide support for division operations, they were replaced by a software approach based on logarithm methods using shift and logical displacements.

Edge detection: This common image processing application is based on the Sobel algorithm and was programmed with a 3x3 size dimensions stencil element. The stencil describes an image filter and it is applied to a 2-dimensions input. As described below for FFT, the division operations are implemented employing the same logarithmic approach.

Vector add: This typical embarrassingly parallel application operates on two individual arrays and stores the result in a specified memory area. This program kernel is selected considering that most applications include execution segments with fully data-parallel operations. This application employs data-path modules and execution units to process the operations.

IV. EXPERIMENTAL RESULTS

The fault campaigns considered two different sets of parameters, the GPGPU model configuration, and the benchmark configuration. The GPGPU model was configured employing 8, 16 and 32 SP-cores. Moreover, the benchmarks were configured with two application threads per block (TPB) distributions: *A→ 32 threads* and *B→ 64 threads*. Benchmarks under each configuration are named as follow: benchmark name, thread configuration, SP-cores configuration. For example, VectorAdd with 32 TPB and 16 SP-Cores is named as *V_32_16*.

A simplified version of the Mean Workload Between Failures (MWBF) metric introduced in [23] was employed to correlate the number of detected faults, the faults applied, the benchmark execution time, and the data processed for every benchmark. This metric represents the correctly executed workload on each application before experiencing a fault. Thus, a higher MWBF means higher fault reliability. It is worth noting that, DUE errors are not considered in the MWBF computation. The SM warp Scheduler was divided into two parts (memory, and logic). Table 3 reports the gathered results, expressed in terms of clock cycles.

TABLE 3 MWBF RESULTS (PROCESSED BYTES PER CLOCK CYCLES)

		FFT		EDGE		Vector Add	
Module	Config / SP-Cores	A	B	A	B	A	B
File register	32	7.6	11.5	22.0	43.5	139.2	111.6
	16	5.6	6.8	16.4	34.3	79.7	83.9
	8	3.7	8.5	10.6	40.1	57.0	60.2
Warp memory	32	565.4	766.2	2,570.7	12,468.5	16,163.7	2,208.1
	16	1,695.9	33.6	974.1	220.7	2,165.9	585.0
	8	570.3	7.5	174.5	81.8	361.1	194.7
Warp logic	32	102.2	140.1	210.4	780.5	1,766.9	1,985.3
	16	34.3	85.6	285.4	186.9	970.7	1,083.9
	8	20.0	25.4	104.7	84.8	615.7	640.7
Divergence Stack memory	32	399.8	259.9	2,688.4	2,158.0	-	-
	16	269.1	155.7	1,903.3	1,084.2	-	-
	8	207.6	63.7	1,338.1	390.6	-	-

In the target modules, the SEU sensitivity depends on the SP-cores configuration. Thus, dropping the number of SP-cores reduces the reliability of the system. This behavior is constant for each module and kernel configuration. The file register is more reliable to SDC and timeout errors by increasing the TPB. In contrast, the divergence stack, the warp logic, and the warp memory seem to be more reliable with kernels configured with a lower number of TPB. A detailed analysis for each module is provided in the following sub-sections.

A. *Data-Path module results*

1) *Register File Results*

27 multi-thread fault injection campaigns were performed injecting 34,816 faults for the *FFT* and *Edge* programs. For *VectorAdd*, 10,240 faults were injected in 32-SP cores and 8,192 faults in the 16- and 8-SP cores configurations. The fault list was divided into ten parts and fault simulations were performed in parallel reducing the fault simulation time from about 150 hours to less than 16 hours. UDR factor also reduces the total amount of faults to inject in up to 95%.

Results in Fig. 2 shows that *FFT* and *Edge* benchmarks present a similar behavior. In both cases, the error rate reduces by increasing the number of SP-cores and by increasing the number of TPB. In *FFT*, a slight increment in the SDC error-rate is generated by increasing the TPB. This behavior can be explained through the relation of the model execution time and kernel configuration. In principle, data stored in active registers for long periods are more prone to SEU effects (case B) than registers with periodical write and read activity (case A).

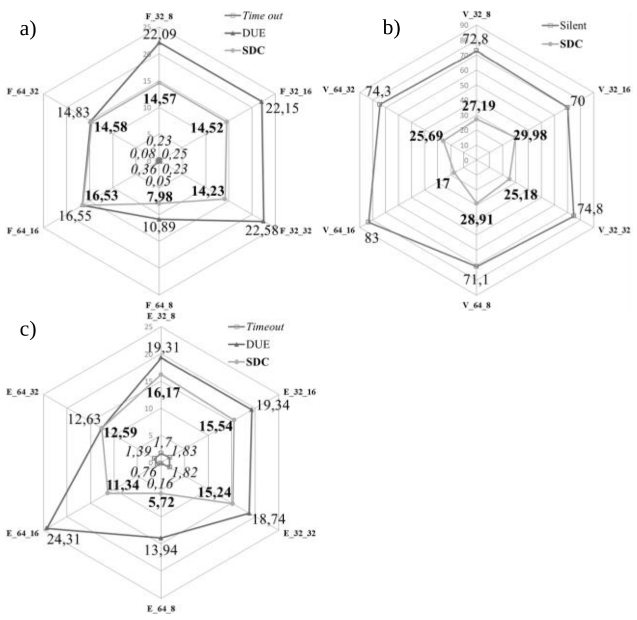

Fig. 2. Register File results for FFT (a), VectorAdd (b) and Edge (c) kernels.

In simulations, the *A* configuration models required longer execution time. However, the individual block execution time is lower than the time required by *B* configurations. Moreover, *FFT* in *A* configuration uses half of the registers of those employed in *B* configuration and employs them to process threads data, in different interval times, belonging to different blocks. In this case, the increment in TPB increases the SDC error rate, as it happens in the 32 and 16-SP cores configurations.

Another factor affecting the error rate is the instruction type. *FFT* includes control-flow instructions depending on predicate conditions, which are generated evaluating register operands. Thus, some registers are included in control-flow operations. Those registers can be considered as control-flow critical registers (CFRs). If an SEU fault affects one of these CFRs, most of the effects are reflected as DUE.

According to results, a higher number of CFRs is generated by decreasing the TPB. This can be explained considering the registers employed in the *A* configuration and the CFRs mapped among threads with the same address locations. During kernel execution, one register location will store, in different time intervals, data belonging to two CFRs, increasing the probability to generate a DUE.

A different behavior is shown by the *Vector_Add* benchmark. An increment in the TPB corresponds to an increase in the SDC error rate. This trend is visible for all SP-cores configurations and depends on the increased SEU sensibility due to the additional time required by the SM to dispatch other warps belonging to the same block. Moreover, the execution time to process an instruction under a large number of threads (*B* configuration) is the double of a block with fewer threads (*A* configuration). Additionally, SEU effects slightly increase by reducing the SP-core configuration. This behavior can be explained by the additional time employed by the scheduler to process one instruction, of each thread, with the limited number of SP cores. The number of SEU faults generating DUE and Time-Out effects is zero as this application does not use any control-flow instruction.

In the *Edge* benchmark, we can observe an inverse relationship between the SDC error rate and the TPB. This behavior is visible in each SP-Core configuration. It can be explained noting that this kernel includes a large number of control-flow, divergence generation, and arithmetic-intense instructions. Regarding the DUE error rate, results also show an inverse relation between TPB and the error rate. This can be explained due to the SEU sensibility of CFRs. Results (*Edge Detection* and *FFT*) are similar to those shown in [19] for control-flow applications.

B. *Control-Path results*

1) *Warp Scheduler results*

36 fault campaigns were performed targeting this module. The model flexibility allows us to divide the module into two parts for analysis purposes: the internal memories (*Warp, State,* and *Predicate*) and the sequential logic components in the module. Results are presented in Fig 3.

At first glance, results contradict the criticality of this module in the GPGPU operation. Nevertheless, a deep analysis of its architectural organization and the role employed by the scheduler helps to clarify results meanings.

The error rate in the sequential logic is caused by the SEU sensibility and criticality of the internal registers employed in processing and storing the warp information. Although the sequential logic corresponds to 14.3% of the elements in the scheduler controller, the percentage of DUE effects lies in a range between 85% and 92% in all kernels. It means that errors in those registers directly compromise kernel termination.

The unexpectedly low fault error rate in the warp memory is caused by a loop existing between the scheduler and the SM pipelines stages. This loop helps to mask and reduce SEU effects in memory since affected information is presented simultaneously in the pipeline registers and in the targeted memory. After each instruction execution, this memory is written (refreshing the information) and correcting any SEU. Moreover, this special memory allows performing the write and read process in a few clock cycles, during a new instruction load, reducing the error propagation. SEU effect on the state and predicate memories is zero for the selected benchmarks.

Results show that increasing the TPB raises the SDC and DUE error rate. The program, under the *B* Configuration, uses more memory locations and requires the execution of two warps to process one instruction including warp line exchange. This exchange generates a temporary short in the loop and the memory location cannot correct any SEU.

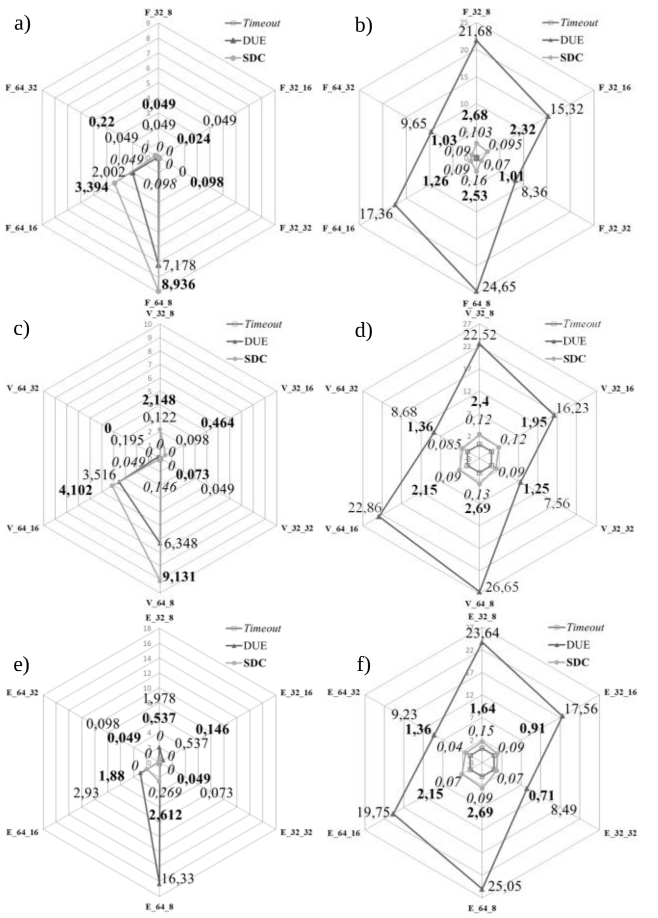

Fig. 3. Warp Scheduler results in warp memory (a, c and e) and sequential logic (b, d, and f) for *FFT* (a, c), *VectorAdd* (b, d) and *Edge* (e, f) benchmarks.

A reduction in SP-Cores produces a direct increment in the error rate. It can be explained by the additional work performed by the scheduler (twice and four times) for thread execution in the 16 and 8 SP-Cores configurations.

2) *Divergence Stack memory*

Vector_Add program was not considered in the fault campaigns because this kernel does not use the Divergence Stack memory. Multi-thread fault campaigns with 50,688 faults were performed for the *FFT* and *Edge* benchmarks. Results are presented in Fig. 4. These show that the divergence stack memory does not generate a relevant contribution to the error rate by SEU effects. This behavior is explained by the partial usage during kernel execution. Each memory location (*line*) is employed for the time fraction of a divergence generation. Thus, each line has a different SEU sensibility. A detailed inspection to this unit, for both kernels, revealed that its usage is limited to less than two-thirds of the total simulation time. Moreover, each additional pushed line presents fewer activities generating a low SEU sensibility in this unit.

The difference in terms of error rate between the two benchmarks is explained analyzing the instructions, its description, and the divergence paths length. Moreover, the number of synchronization point instructions (SSY) determines the usage of each memory location. *Edge* kernel uses seven independent SSY instructions with a short path length and seems to be reliable to SEU effects. In contrast, *FFT* includes two SSY instructions and long divergence paths. This long interval time between writing and reading seems to increase the SEU sensitivity.

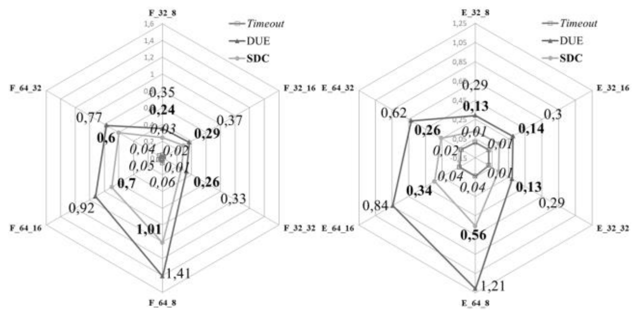

Fig. 4. Divergence stack result for *FFT* (left) and *Edge* (right) benchmarks.

Regarding the DUE and SDC error rates, they depend on the affected location. The difference, for both applications, is mainly caused by the ability of the program counter and mask fields to generate hang conditions. An SEU in the program counter may generate Timeout or DUE errors. Similarly, the effect in the aTM may generate SDC, by inactive threads, or DUE effects, by threads missing the taken path. Finally, an SEU in the warp ID field generates Timeout effects.

The model with *A* Configuration uses the same lines in the divergence stack, but these lines are employed in different time slots and the execution time per block is lower than that required in *B* Configuration. The additional time in *B* Configuration seems to be responsible for the increasing SEU sensitivity. A decrement in TPB could help to reduce, in more than twice, the SDC error rate.

V. CONCLUSIONS

We introduced an improved version of the open source GPGPU model FlexGrip. This detailed model description was crucial to explain the behavior observable in the control unit modules when are affected by transient faults. Although the FlexGrip model does not completely match the architecture of the most recent GPGPU devices, we still claim that the performed analysis may be valid for some of them as well. The new model version is technology independent. Moreover, each instruction was checked and the supported formats were listed. Additionally, further tools have been implemented to provide assistance in the development of new applications employing the CUDA environment.

SP-cores customization in the model could be useful for area and energy optimization. However, according to Table 3, a lower number of SP-cores increases the SEU sensibility and reduces system reliability. We performed several fault injection campaigns to analyze the effects of SEUs in different modules within the GPGPU with different applications. The results showed that the behavior of the error rate (measured via the MWBF metric) when changing the configuration parameters depends on the application. Thanks to the availability of the FlexGrip model, we provided explanations about the observed phenomena.

VI. FUTURE WORKS

We are currently working to extend the analysis of the SEU effects to other modules within the GPGPU architecture employing different program kernel characteristics.

We also plan to extend the instruction and hardware support of FlexGrip model following the SM 1.0 microarchitecture compatibility. Moreover, new execution units, such as floating point units are also potential extensions for the model. The support to different warp scheduler controller algorithms is also planned as future work.

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] S. Hamdioui, D. Gizopoulos, G. Guido, M. Nicolaidis, A. Grasset, and P. Bonnot, "Reliability challenges of real-time systems in forthcoming technology nodes," in *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2013, pp. 129-134.

[3] V. Sridharan, N. DeBardeleben, S. Blanchard, K. B. Ferreira, J. Stearley, J. Shalf, *et al.*, "Memory errors in modern systems: The good, the bad, and the ugly," *ACM SIGARCH Computer Architecture News,* vol. 43, pp. 297-310, 2015.

[4] H. L. Hughes and J. M. Benedetto, "Radiation effects and hardening of MOS technology: devices and circuits," *IEEE Transactions on Nuclear Science,* vol. 50, pp. 500-521, 2003.

[5] 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.

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] A. Vallero, D. Gizopoulos, and S. Di Carlo, "SIFI: AMD southern islands GPU microarchitectural level fault injector," in *2017 IEEE 23rd International Symposium on On-Line Testing and Robust System Design (IOLTS)*, 2017, pp. 138-144.

[13] N. Farazmand, R. Ubal, and D. Kaeli, "Statistical fault injection-based AVF analysis of a GPU architecture," *Proceedings of SELSE,* vol. 12, 2012.

[14] S. Tselonis and D. Gizopoulos, "GUFI: A framework for GPUs reliability assessment," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2016, pp. 90-100.

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] T. Santini, P. Rech, G. Nazar, L. Carro, and F. R. Wagner, "Reducing embedded software radiation-induced failures through cache memories," in *2014 19th IEEE European Test Symposium (ETS)*, 2014, pp. 1-6.