FlexGripPlus: An improved GPGPU model to support reliability analysis

Microelectronics Reliability 109 (2020) 113660

Contents lists available at ScienceDirect

Microelectronics Reliability

journal homepage: www.elsevier.com/locate/microrel

FlexGripPlus: An improved GPGPU model to support reliability analysis☆

Josie E. Rodriguez Condia*, Boyang Du, Matteo Sonza Reorda, Luca Sterpone

Dipartimento di Automatica e Informatica, Politecnico di Torino, Torino, Italy

A B S T R A C T

General Purpose Graphics Processing Units (GPGPUs) have been extensively used in the last decade as accelerators in high demanding applications, such as multimedia processing and high-performance computing. Nowadays, these devices are becoming popular even in safety-critical applications, such as in autonomous and semi-autonomous vehicles. However, these devices can suffer from the effects of transient faults, such as those produced by radiation effects. Among those effects, Single Event Upsets (SEUs), which are the focus of this paper, can cause application misbehaviors, which may lead to catastrophic consequences. In this work, we first describe how we extended the capabilities of an open-source VHDL GPGPU model (FlexGrip) and developed a new version named FlexGripPlus to study and analyze the effects of SEUs in a GPGPU in a much more detailed manner. We also performed extensive fault injection campaigns using FlexGripPlus, which allowed identifying the most critical effects within the GPGPU architecture. We finally focused on the scheduler controller since it represents a module that is specific to the GPGPU architecture and showed that it has different levels of SEU sensibility depending on the affected location. Moreover, the results of additional analyses varying the number of parallel execution units in the system are presented, demonstrating the correlation between the number of execution units in a GPGPU and the system reliability.

1. Introduction

In recent years, GPGPU devices have become popular solutions in high-demanding data processing applications, including multimedia processing and high-performance computing (HPC). More and more, these devices are also adopted in several data-intensive safety-critical (i.e., autonomous vehicles [1–3]) and mission-critical (i.e., autonomous systems [4]) applications. Moreover, GPGPUs are designed and manufactured using the latest technology process to satisfy energy consumption and performance requirements. However, some studies have shown that these advanced semiconductor technologies are prone to suffer from internal effects impacting their reliability, such as aging and wear-out, and from external ones (i.e., Electromagnetic Interference EMI or radiation effects) [5–8]. Thus, reliability analyses can help to solve or mitigate these effects during the operational phase of such devices.

The nature of a fault affecting a device can be related to the integration scale, the production process variation, and the operating conditions. Currently, new high-performance devices could be affected by aging and wear-out effects originated by the increased sensibility presented as a combination of the previous factors. The complexity of new designs in conjunction with lower voltage, timing, and noise margins reduces the probability of identifying wear-out-free devices correctly. Moreover, some manufacturing processes increase the susceptibility to radiation effects, so transient or intermittent fault effects could be present as fugacious faults [9] and corrupt the operation of the device. Similarly, delays caused by aging mechanisms (i.e., Negative bias temperature instability (NBTI) [10]) could affect critical paths in some modules of a device and, in the long term, behave as transient faults.

Some types of external effects can be modeled as Single Event Upsets (SEUs) and may cause misbehaviors in a safety-critical context leading to catastrophic consequences. In real GPGPU devices, the impact of SEU effects can be analyzed through radiation experiments in a few specialized facilities using expensive and complex equipment. Other methods include software-based fault injection tools instrumenting the compiler, and modifying the application code [11] or using profiling tools to validate the proposed mechanisms [12]. However, in both cases, the analysis of reliability is complex to perform or limited, considering that detailed information about the structure of the device and its implementation are commonly unknown.

An alternative solution is based on model simulation or emulation. In this approach, a model is available and can be instrumented to inject faults in the target module. Then, thanks to the model, we can compute the effect of a fault during the execution of a given application. The outputs can be observed and used to perform a reliability assessment or to identify structural or application weaknesses in a GPGPU. Moreover, these analyses are employed to choose the most suitable mitigation strategy in an application at different levels (e.g., hardware, software, or system) [13].

☆ The European Commission has partially supported this work through the Horizon 2020 RESCUE-ETN project under grant 722325.

* Corresponding author.
E-mail address: Josie.rodriguez@polito.it (J.E.R. Condia).

https://doi.org/10.1016/j.microrel.2020.113660
Received 30 July 2019; Received in revised form 28 February 2020; Accepted 7 April 2020
Available online 06 May 2020

The level of abstraction in a model is a crucial parameter that affects the expected conclusions and is related to the characteristics and the degree of similarity with the behavior of a real device. Hence, for the purpose of the reliability analysis against SEUs, models with low-level descriptions, e.g., implementations using some Hardware Description Language (HDL) at Register Transfer Level (RTL) or gate level, are preferable since they provide a detailed representation of the internal structures. However, such models require higher computational effort for simulation or emulation than the high-level ones.

Regarding GPGPU devices, the number of available models to support reliability analysis is limited. Moreover, most models are described at high-level [14–20] or as a combination of multiple levels of abstraction [21], which renders reliability analysis against SEUs on critical units, such as controllers or arbiters unfeasible. On the other hand, there is a set of low-level GPGPU models available: IPPro [22], FGPU [23], Simty [24], Nyami [25] and FlexGrip [26]. The first four models were designed targeting FPGA platforms using custom architectures. Thus, their custom architectures and implementation target technologies would limit the capability for reliability analysis, especially when commercially available GPGPU devices are concerned. On the other side, the architecture implemented in the FlexGrip model is closer to commercial devices by NVIDIA. FlexGrip also supports partial binary compatibility with the NVIDIA G80 instruction set. Unfortunately, the original version of the FlexGrip model has some restrictions related to technology dependency, instructions format support, and compiler compatibility, limiting the development and study of new applications to support reliability analysis.

In this paper, FlexGripPlus[1] is introduced as a new version of the FlexGrip model with improvements, including a set of architectural and functional changes in the model to increase flexibility while maintaining the architecture of the original design. The FlexGripPlus replaced Xilinx FPGA library dependencies in the original version and now has support for a reviewed and extended set of instructions compatible with the NVIDIA programming environment.

Some representative applications have also been developed for FlexGripPlus besides the five applications from the original version as benchmarks to evaluate the effect of SEUs in data-path and control-path modules. It is worth noting that in previous works [27,28], some initial improvements have already been briefly described. Similarly, preliminary results have been presented in [29,30], characterizing the fault effects on some modules of the GPGPU model.

The main contributions of this work are:

- The detailed description of the approach we used to verify, correct and extend the functionality of the original GPGPU model at the microarchitecture level to develop the FlexGripPlus model;
- The method used for performing the fault simulation campaigns on FlexGripPlus targeting SEUs;
- The description of the new benchmark applications, as well as the required software environment to support further application development;
- The analysis of the results coming from the fault simulation campaigns targeting both the data-path and the control-path of FlexGripPlus. To the best of our knowledge, this is the first work reporting a detailed analysis of the sensitivity of different modules of a GPGPU to SEUs under different encoding styles.

The rest of the paper is organized as follows. Section II briefly describes the general organization of a GPGPU and the FlexGrip model. Then, section III introduces the proposed improvements introduced into FlexGripPlus and the adopted methodology. Section IV presents the fault injection setup, the targeted modules, and the selected benchmarks for the fault injection campaigns. Section V reports the experimental results and the reliability analysis regarding SEUs. Finally, Section VI draws some conclusions and outlines future works.

2. Background

This section introduces the fundamentals of the microarchitecture of a GPGPU device and describes the relationship between the FlexGrip model and the related commercial devices.

2.1. General GPGPU micro-architecture

The GPGPU devices initially targeted for Graphics Processing (hence, the GPU in the name), are commonly based on the Single-Instruction Multiple-Data (*SIMD*) taxonomy [31] and its variations, such as the Single-Instruction Multiple-Thread (*SIMT*) concept by NVIDIA [32]. Thanks to their high parallelism, these devices are more and more adopted in fields other than Graphic Processing, such as scientific computation, where data-intensive workloads can be processed in parallel. A GPGPU device based on SIMT, such as the ones from NVIDIA, is mainly composed of a set of highly parallel execution cores, also known as Streaming Multiprocessors (SMs) or Computing Units (CUs), each of which can execute multiple threads simultaneously utilizing its own registers, caches, local memories, control units, and execution units. The computational capability of a GPGPU device is proportional to the number of available SMs[2] and the local resources available in each SM.

Fig. 1 shows a general scheme of the architecture of a GPGPU device with four SMs. It includes a Block Scheduler to distribute the tasks among the SMs. Each SM block is mainly composed of multiple parallel execution cores (CUDA cores, Execution Units (EUs), or Scalar Processors (SPs)) and other modules, including instruction and data caches, a scheduler controller (or *Warp Scheduler Controller (*SC)), one or more *warp* dispatchers, a register file (RF), and some local memories. Modern GPGPU architectures may also include accelerators implementing complex operations, such as the cross product of matrices (*Tensor-cores*), floating-point operations (FP32/FP64), and transcendental functions (SFU). The scheduler controller and warp dispatchers are crucial modules in the SM to manage the executions of threads (particularly when local resources do not allow all threads, organized into *warps*, to be executed simultaneously) and to manage the occurrences of the intra-warp divergences due to thread-dependent branches [32–35].

2.2. FlexGrip model

The FlexGrip is an open-source soft-GPGPU model based on the NVIDIA G80 microarchitecture and implemented in VHDL. This GPGPU model was developed by the University of Massachusetts [26] and was designed to be fully compatible with the CUDA programming environment using the SM 1.0 compatibility. The FlexGrip model is based on the description of a Streaming Multiprocessor (SM) module and supports 27 assembly (SASS) instructions. Originally it was designed targeting a Xilinx SRAM-based FPGA.

The internal description of SM in FlexGrip is mainly composed of a five-stages pipeline (*Fetch*, *Decode*, *Read*, *Execution,* and *Write-Back*) following the G80 microarchitecture [32], as shown in Fig. 2. The total number of SPs in the *Execution* stage is configurable during the synthesis to 8, 16, or 32 parallel cores.

The FlexGrip model includes a basic memory hierarchy composed of system memory, a global or main memory, a shared memory, and constant memory. It is worth noting that cache memories are not included in the model. FlexGrip also includes two task schedulers (*Block*

[1] Available at: *https://github.com/Jerc007/Open-GPGPU-FlexGrip-*.

[2] Through the paper, we use SM and CU indifferently as they are at the same level of parallelism. The same applies to CUDA core, Execution Units (EUs) and Scalar Processors (SPs) etc.

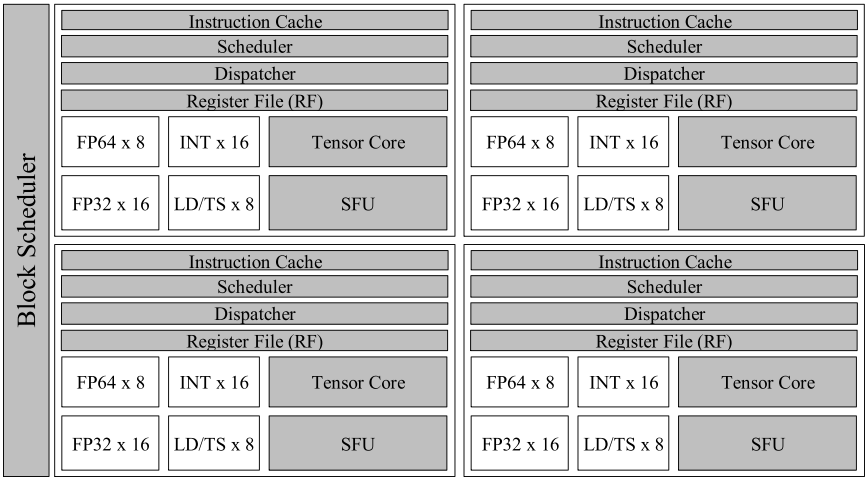


Fig. 1. A general scheme of the GPGPU micro-architecture. Adapted from [33,34].

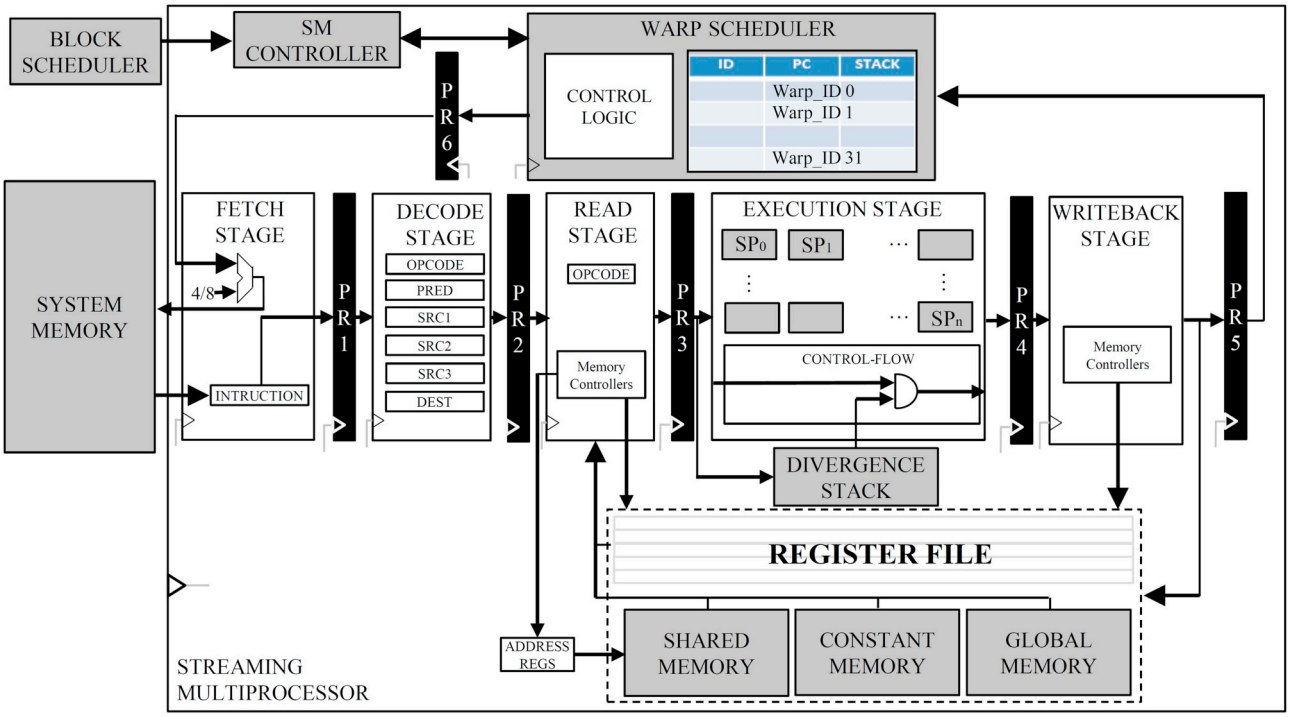


Fig. 2. The general scheme of the SM in the FlexGrip model [26].

Scheduler and *Warp Scheduler*), both adopting a round-robin algorithm as a distribution policy to manage intra-warp divergences caused by a branch instruction. A custom branch module is implemented with a stack memory (denoted as "Divergence Stack" in Fig. 2) to handle up to 32 levels of divergence. Please note that there are registers between different stages of the pipeline (black boxes named "PRx" in Fig. 2), which may also be subject to SEUs. The Structural comparison between the schemes of the FlexGrip model and the commercial GPGPUs (from NVIDIA) shows that both share the same basic functional blocks, including the block and warp scheduler controllers, a register file, the parallel execution units, and the pipeline stages.

Nevertheless, the memory hierarchy in FlexGrip differs from the one in the commercial devices by the missing cache memories. Moreover, FlexGrip has no floating-point modules or special purpose accelerators. These structural limitations in FlexGrip can affect the adoption of the most recent applications. However, we argue that the analysis of reliability and fault effects on data-path and control-path modules performed in FlexGrip can still be meaningful with respect to the commercial devices considering the architectural similarities between them.

3. FlexGripPlus

The development of more complex and detailed analysis and correlate results with real devices required the correction of some limitations present in the original FlexGrip version. So, FlexGripPlus has been implemented with some modifications and improvements based on the original version of FlexGrip.

The major limitations of the original FlexGrip include the dependency on the Xilinx FPGA library, the limited number of supported instructions, and the incomplete compliance with the NVIDIA application development environment. During the FlexGripPlus development, the microarchitecture of the original model has been kept untouched while each initially supported instruction and the newly added

Table 1
Control-flow instructions supported in FlexGripPlus.

Mnemonic	Description	Formats
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
CAL	Call to subroutine	CAL CAL.NOINC
NOP	No operation	NOP NOP.S

instructions have been carefully verified for correctness. To achieve this, a methodology to improve the FlexGrip model has been used based on the following steps:

1. Rigorous structural analysis of the FlexGrip model
2. Development of basic test programs to verify each instruction supported by the design
3. Simulation and interpretation of results
4. Definition of potential corrections
5. Application of the revisions and verification with previous programs
6. Development of new applications based on typical applications and workloads for validation purposes.

In the beginning, the analysis of the internal structures in FlexGrip revealed that some modules, including the distribution controller of RF, the *Decode* stage, and the *Execution* stage, are incomplete. These incomplete modules can lead to incorrect results under certain circumstances and had to be fixed. Then (in step 2), a large number of specific applications were developed to generate all the potential format of the supported instructions and verify the correct execution of each instruction. Each program was developed using the CUDA environment when possible, or directly at the assembly level. During this step, incompatibility issues between the instruction decoding implemented in FlexGrip and the binary code generated by the CUDA environment have been identified and fixed. Furthermore, for each modification, steps 3 to 5 have been repeated to verify the correctness of the new implementation. This methodology should serve as a guideline if the model has to be further improved and new modules have to be added.

Through the process of implementing FlexGripPlus from the original FlexGrip, the introduced changes can be divided into three groups.

3.1. Technology dependency

The FlexGrip model was initially designed as soft-core targeting a Xilinx FPGA, and some internal modules, such as the memories, were automatically generated as IP cores using the Xilinx System Generator tool. The descriptions of these IP cores are not human-friendly to read or analyze and significantly limit the possibility to perform the required gate-level analysis when SEUs are concerned.

In FlexGripPlus, we carefully modified each module by removing any reference or dependency on specific technology libraries, replacing them with equivalent generic descriptions. Through this process, the names of signals, interconnections, and modules were clarified to simplify the analysis on each signal after a fault simulation campaign. In the end, about 40% of the modules were modified for this purpose. It is worth noting that the model can now be easily imported into different simulation environments, such as ModelSim, QuestaSim, or Xcelium Parallel Simulator. Moreover, the removal of the technology dependency from the model allows mapping the model into other platforms or technologies that were not previously supported, such as ASICs. The FlexGripPlus model can be synthesized using proprietary or independent technology libraries, such as the ASIC 45 nm OpenCell [36] or the 15 nm OpenCell [37] libraries.

3.2. Instruction format support

The original FlexGrip implementation was intended to be compatible with the CUDA programming environment through SASS instructions. However, through simulation and analysis using the test programs we developed, some internal modules in control- and datapath units, such as decoding logic, intermediate registers, and block interconnections, were found to be not present or only partially implemented.

Hence, we started from the initially supported instructions in FlexGrip, and each instruction was verified against the NVIDIA CUDA programming environment. As NVIDIA has not officially released the SASS op-codes (i.e., the instruction formats), the op-code and formats of some instructions were decoded using the CUDA binary tools, namely ***nvcc*** and ***cuobjdump***, and the *Decuda* open-source project [38]. Additionally, multiple benchmark applications were designed targeting specific instructions to force the compiler to produce the target instruction op-code under various formats. Some issues, such as the compiler optimizations, and the change or removal of instructions, required the explicit addition of numerous output memory locations in the micro-kernels.

All the required changes (e.g., descriptions of missing registers, connections, combinational logic, or the correction of incomplete modules) were introduced in the FlexGripPlus model to fully support the set of instructions with all the format variations, including 28 instructions and 74 formats. Table 1 shows the supported control-flow instructions and their formats. Table 2 lists the supported arithmetic and logic instructions. Finally, Table 3 shows the data-handling and memory instructions. The parameter COMP_TYPE in Table 2 refers to the comparison type and modifies the state of a predicate flag as the effect of an arithmetic or logic operation. The COND parameter in Table 1 is related to the conditional execution of an instruction, considering the state of a predicate flag. The **g[]** and **c[0 × 1][]** fields correspond to instructions using operand sources coming from the shared memory and the constant memory, respectively.

In the end, about 4.8% of the code in FlexGrip was modified for this purpose. Most of those modifications were performed in the Decode and Read stages of the pipeline in the SM.

3.3. Compiler restrictions

The primary approach to develop new applications for the FlexGrip model was based on the CUDA programming environment. After the instruction compatibility issues mentioned above were fixed in FlexGripPlus, there was still a gap between all the instructions that can be generated by the CUDA compiler and the instructions supported in FlexGripPlus. Hence, three software tools have been developed to check and preserve the compatibility between the CUDA programming environment and the FlexGripPlus implementation.

Firstly, an assembly language checker tool named *SASS checker* has been developed to check the binary code generated by the CUDA compiler ***nvcc*** against the instructions supported by FlexGripPlus. If any unsupported instruction is generated by ***nvcc***, then the second tool, an assembly code writer tool named *SASS parser*, will try to convert any unsupported instruction into one or more supported instructions with the equivalent operation. However, in case of failure of such a conversion attempt, the tool will report an error, and the user manually modifies the source code to avoid the generation of unsupported instruction.

Thirdly, a memory configuration tool named *Index corrector* can verify and correct the mismatches in the addressing indices, which are used to address the memories during the execution of an application. The *index corrector* is required considering that the locations to store the

Table 2
Arithmetic and logic instructions in FlexGripPlus.

Mnemonic	Description	Formats
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/ IMAD32I		IMAD32.U16 RZ, RXL|H, RYL|H, RZ 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 3
Data and memory instructions in FLexGripPlus.

Mnemonic	Description	Formats
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 data register to address register	R2A AX, RX
A2R	Move address register to data register	A2R RX, AX
ADA	Move Address register to Address register	ADA AX, AY

variables in a program are managed and decided by the compiler tool without significant user intervention. However, in multiple applications, it was required to verify and correct the indices used to address the memories of the system since the way FlexGripPlus manages the global, shared, and constant memories are not exactly the same as in NVIDIA devices.

With the three tools, the time to develop new applications as benchmarks for reliability analysis using FlexGripPlus is greatly reduced. They are also used in step 6 mentioned previously to use new applications to verify modifications done in FlexGripPlus.

In Table 4, a comparative analysis of the main features of FlexGrip and FlexGripPlus is presented to list the effects of the introduced modifications and the improvements mentioned above.

It is worth noting that one additional instruction (ADA) was implemented, which is fully compatible with the programming environment. The ADA instruction manages the interaction among the address register modules in the GPGPU and is commonly used to access the shared memory.

4. Fault injection setup

This section introduces a custom cross-platform fault injection environment developed to evaluate the effects of SEUs in the FlexGripPlus model. It is worth noting that technological features were not considered in the experiments to represent SEU effects. The goal of our experiments was rather to compute the fault rate and the Architectural Vulnerability Factor (AVF) [39] of different modules in the FlexGripPlus model. Sub-section A describes the fault injection environment. Sub-section B presents the target modules in FlexGripPlus to be evaluated. Finally, sub-section C briefly describes the benchmarks selected to perform the fault campaigns.

4.1. Fault injection environment

The fault injection environment was developed based on the ModelSim simulator framework. Nevertheless, this can be easily adapted into other simulator frameworks, such as the Xcelium Parallel Simulator or QuestaSim. The developed environment follows the guidelines introduced in [40] using commands provided by the simulator to inject faults. Additionally, the environment can reduce the total time of the fault simulation by taking advantage of 1) parallel capabilities of the modern computers to run multiple simulations simultaneously and 2) a module de-rating factor (UDR) to pre-process the fault list and reduce the number of locations to inject faults. The UDR is computed utilizing results from a fault-free simulation. The information is analyzed (i.e., switching activity or correlation) in the target module, and finally, those unused locations by an application are removed [41,42].

The fault injection environment is implemented in *Python* and is composed of three main modules, as shown in Fig. 3: *1)* a *fault controller*, *2)* a *fault injector,* and *3)* a *fault checker and classifier*.

The *fault controller* manages the execution of a fault campaign, including initialization of target design in the specific simulator, starting and ending of a fault injection run. In the case of FlexGripPlus, the *fault controller* loads the FlexGripPlus design, the parameters of the benchmark application (kernel), the initialization data of different memories, including the application itself in the System Memory and the input data in the Global Memory. In the meantime, the optimized fault list (generated by another tool, not shown in Fig. 3) is loaded by the *fault*

Table 4
Comparative analysis of main features in the FlexGrip and FlexGripPlus models.

	FlexGrip	FlexGripPlus
Instructions	• 27 instructions (partially supported)	• 28 Instructions fully supported • 78 formats of instructions verified
Programming environment	• Partially compatible with the CUDA programming environment. • A manual mechanism to compile the CUDA (.cu) file and adapt to the model	• Partially compatible with the CUDA programming environment. • A tool to translate the CUDA (.cu) description into the final binary file used in the model • A tool to develop applications at the assembly level (SASS) • A tool to verify the compatibility of the generated assembly code.
Applications	• Only 5 benchmarks	• More than 20 verified applications
Simulation or Implementation platforms	• Simulation (ISIM, VIVADO simulator) • FPGAs (Xilinx)	• Simulation (ModelSim, Xcelium Parallel Simulator) • FPGAs • ASICs
Memory management support		• An additional support to manage the address register file employed to access the Shared memory

controller and divided into chunks to be launched simultaneously. Once everything is ready, fault injection commands will be sent to the *fault injector*.

The *fault injector* decodes a fault injection command from the *fault controller* and generates the command(s) to be executed by the target simulator, e.g., ModelSim. Currently, two types of faults are supported by the environment: 1) permanent Stuck-At faults, and 2) transient bit-flip faults. As the focus of this paper is on SEUs, the fault injection command generated in the fault simulation campaign includes the signal name (fault location), the event time (fault occurrence time), and duration (related to the clock frequency of target design). The *fault injector* then starts a simulation run with the generated command(s) and waits for the run to finish to process another fault injection command.

Finally, the *fault checker and classifier* monitor the outputs until the termination of the simulation run, gather information, and classify the fault accordingly. In the case of FlexGripPlus, the faults are classified into the following categories: 1) *Silent Data Corruption* (SDC) when the injected fault affects only the final results in memory, 2) *Detected Unrecoverable Error* (DUE) when the DUT hangs or crashes during the simulation, 3) *Timeout* when the SEU produces performance degradation in simulation time and 4) *Masked* when the injected fault does not generate any impact.

When using the developed fault injection environment for fault simulation campaigns targeting FlexGripPlus reported in this paper, the fault list is generated with a careful selection of fault location considering the actual used registers and memory locations used in each benchmark application, while the fault occurrence time is selected randomly.

4.2. Targeted modules

Two data-path and two control-path modules were targeted for evaluation in FlexGripPlus.

4.2.1. Data-path modules

4.2.1.1. Register File (RF). This module is located inside the SM and is composed of 16,384 32-bit registers. Each register can be accessed in the *Read* and *Write-back* stages by different threads depending on the parameters of the running kernel. The data stored in the registers can be computational data or addresses, which can affect the execution flow of the application.

4.2.1.2. Pipeline Registers (PRs). These registers are located among the stages of the pipeline and the warp scheduler, as shown in Fig. 2. As these registers hold data and control signals, SEUs in them can lead to data corruption or interruption of the execution flow of the application. The size of PRs in FlexGripPlus is reported in Table 5.

4.2.2. Control path modules

4.2.2.1. Warp Scheduler (SC). This module manages the warp execution inside the SM. A warp status memory is implemented inside the SC to store status information about the active warps. The warp status memory contains 32 128-bit wide entries. The information stored on each entry is composed of the active thread mask (aTM), the actual warp program counter (wPC), and some additional parameters. The information about the active warp is updated after each instruction cycle.

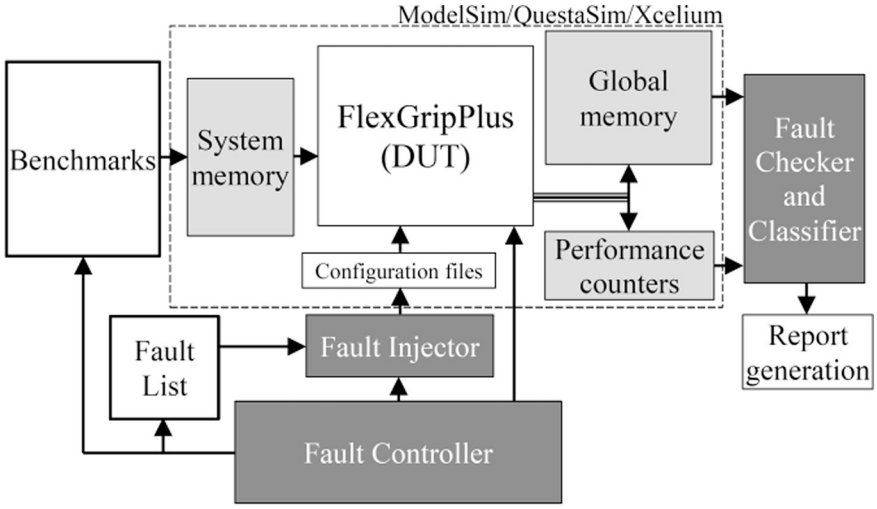


Fig. 3. A general scheme of the simulation environment for FlexGripPlus.

Table 5
Size of PRs in FlexGripPlus.

Location (between)	Size (# bits)
Fetch and *Decode* (F-D)	237
Decode and *Read* (D-R)	408
Execute and *WriteBack* (E-W)	6575
Read and *Execute* (R-E)	3456
WriteBack and *Warp Scheduler* (Wr-W)	133
Warp Scheduler and *Fetch* (W−F)	140

4.2.2.2. Divergence Stack Memory (DSM). This special purpose memory contains 32 32-bit wide entries storing the divergence addresses and the status information of the warp caused by branch instructions. The two execution paths (*Taken* and *Not-taken*) are stored using two entries in the DSM in terms of the starting (divergence) and finishing (convergence) points along with the execution flow. Additionally, each entry in DSM stores the warp index, the flow state condition, and the aTM value to trace the number of executed threads on each path.

4.3. Benchmarks

Six applications were carefully selected as benchmark applications to evaluate the behavior of FlexGripPlus against SEUs under different workload profiles.

1) **FFT:** The application is based on the Coley-Turkey algorithm [43] and implements the butterfly element using CUDA. Since FlexGripPlus does not provide support for division operations, they were replaced by a software-based approach using logarithmic and logical operations.
2) **Edge detection (Edge):** The application is based on the Sobel algorithm applying an image filter of 3 × 3 to a 2-dimensions input.
3) **Vector_Add:** This is one of the original applications developed for FlexGrip. It calculates the sum of two vectors.
4) **Bitonic-Sort (Sort)**: This is another original application developed for FlexGrip. The application sorts a sequence of consecutive data elements stored in an array. This application includes multiple combinations of data movements between memory and registers, and conditional control-flow instructions, generating multiple paths during execution.
5) **M3:** This application implements a Software-Based Self-Test (SBST) algorithm introduced in [44]. M3 targets the memory in the SC. It is composed of multiple control-flow instructions utilizing mainly the control-path modules.
6) **Matrix Multiplication (MxM):** This application is based on the General Matrix Multiplication (GEMM) routine, which is optimized using the square tiling approach. The input matrices are firstly divided into blocks. Then, partial results are obtained by multiplication of corresponding blocks. Finally, partial results are accumulated to get the final results of matrix multiplication. The implementation is limited to 32 × 32 input matrices.

5. Experimental results

As mentioned before, FlexGripPlus can be configured to have 8, 16, or 32 SP cores in SM. Moreover, the benchmark applications (kernels) can be launched using different thread distribution strategies, so different combinations of configurations have been used in the fault injection campaigns.

An initial set of fault injection campaigns was carried out using FlexGripPlus with 8, 16, or 32 SP cores to evaluate the impact of SEUs on different hardware organizations. The benchmark applications were launched using different thread distributions: 32 and 64 threads per block (TPB) except **MxM** was configured to 512 and 1024 TPB, given its

Table 6
Fault injection campaign result in terms of MWBF.

Benchmark	*TPB*	*RF* *MWBF (fault ∗ bits/cc)*			*PRs* *MWBF (fault ∗ bits/cc)*			*SC warp status memory MWBF (fault ∗ bits/cc)*			*SC logic MWBF(fault ∗ bits/cc)*			*DSM* *MWBF (fault ∗ bits/cc*		
		#SP														
		8	*16*	*32*	*8*	*16*	*32*	*8*	*16*	*32*	*8*	*16*	*32*	*8*	*16*	*32*
FFT	32	3.7	5.6	7.6	165.4	202.1	96.1	570.3	1,696	565.4	20.0	34.3	102.2	207.6	269.1	399.8
	64	8.5	6.8	11.5	292.2	353.7	157.7	7.5	33.6	766.2	25.4	85.6	140.1	63.7	155.7	259.9
Edge	32	10.6	16.4	22.0	543.6	599.3	301.3	174.5	974.1	2,570.7	104.7	285.4	210.4	1,338.0	1,903.0	2,688.0
	64	40.1	34.3	43.5	740.0	1,123.7	739.9	81.8	220.7	12,468.5	84.8	186.9	780.5	390.6	1,084.0	2,158.0
Vector_Add	32	57.0	79.7	139.2	3,878.8	4,219.8	4,735.5	361.1	2,166	16,163.7	615.7	970.7	1,766.9	–	–	–
	64	60.2	83.9	111.6	2,849.2	5,820.8	6,037.7	194.7	585.0	2,208.1	640.7	1,084.0	1,985.0	–	–	–
M3	32	473.6	354.0	227.1	976.9	339.5	236.6	11.8	8.7	46.4	46.9	106.9	18.9	30.6	54.2	21.8
	64	1,591.8	3270.2	2,349.7	1,830.5	654.0	637.8	436.9	996.2	998.6	61.3	53.2	2,366.5	26.3	6.3	8.4
Sort	32	307.7	165.7	31.4	173.1	176.1	125.6	207.7	238.3	282.5	212.3	252.2	393.0	358.4	14.2	17.6
	64	0.0	80.7	266.1	116.8	127.7	139.7	362.1	801.8	1,649.7	203.3	327.4	944.6	2.2	3.2	12.0
MxM	512	10,114.2	14,509.4	15,910.2	101.7	73.3	133.3	11,592.1	14,216.3	13,655.6	88.1	128.2	177.7	6.8	15.1	–
	1024	23,879.1	34,620.6	36,327.3	21.0	256.4	300.0	26,787.7	34,428.2	36,149.6	203.4	357.2	411.0	19.0	4.6	–

Number of Faults (%).

characteristics. The second group of fault injection campaigns targeted a special analysis of SEU effects in the SC under different workload distributions: 32, 64, 128, 256, 512, and 1024 TPB (though not for all the selected benchmark applications). It is worth noting that fault injection campaign employed the RT-level description of FlexGripPlus.

5.1. Overall analysis with different configurations

The Relative Mean Workload Between Failures (MWBF) [45] is selected as a metric for reliability analysis against SEUs. The motivation for this choice is that MWBF accounts not only for the percentage of faults producing a failure but also for the different execution time, thus taking into account the higher fault probability. When MWBF was calculated, a constant fault rate was assumed for all applications and all configurations, and the DUE errors were neglected. Moreover, the execution time of each benchmark was calculated, only considering the time interval from the execution of the first instruction up to the kernel termination.

Table 6 reports the results of the first set of fault injection campaigns in terms of MWBF, expressed in terms of the fault rate per bits per clock cycles (cc). Please note that the evaluation of SC is performed individually for the warp status memory and control logic.

Though it is difficult to perform direct comparisons with previously reported experiments in literature, similar conclusions regarding how the number of available SPs and different thread distributions impact the reliability of application execution can be found. In [46], the authors concluded that an increment in the number of blocks reduces the MWBF of an application. Similarly, in [47], experiments proved that an application running in a GPGPU is more reliable when increasing the block size instead of increasing the number of blocks. The same trend can be observed in Table 6 from the results of, for example, *FFT* when RF, PRs, and SC are targeted, and *Edge* when RF, PR are evaluated, etc. However, the opposite trend can also be observed from results of, for example, *Vector_Add* when warp status memory in SC is targeted, and *M3* when DSM is targeted. The trend also changes even with the same application when the FlexGripPlus is configured to different numbers of SPs in SM, for example, *Edge* with 8 or 16 SPs available and 32 SPs when warp status memory in SC is targeted.

Please note that, in this paper, we are targeting individual modules instead of the whole devices as in [46,47], so that the impact of reliability against SEUs of each individual modules are different, as reported in Table 6 when FlexGripPlus is configured with different number of SPs and application launched with different thread distribution profiles. Besides, current FlexGripPlus implementation is still limited to one SM that Block Scheduler did not make any contributions to the fault injection campaign results.

5.2. Detailed analysis per module

The Fault error rate was computed for each module under test and for each application categorized in three different effects, i.e., SDC, DUE, and Timeout. The obtained results are presented as follows w.r.t. each targeted module.

5.2.1. Register File (RF)

In total, 30 fault injection campaigns using the *FFT*, *Edge*, *M3, Sort,* and *MxM* benchmark applications have been performed, each considering 34,816 faults, resulting in the confidence of 99.46%. Meanwhile, for the *Vector_Add* application, 10,240 faults have been injected for 32 SPs configuration, and 8192 faults for 16 and 8 SPs. With the aforementioned fault injection environment, the fault simulation time was reduced from about 200 h to less than 25 h: the adoption of the UDR factor allowed us to reduce by up to 95% the total amount of injected faults. Fig. 4 reports the error rate percentage for each application.

It can be observed that *FFT* and *Edge* show similar distribution when the RF module is targeted. *Vector_*Add does not produce any DUE or Timeout as it does not contain branch instruction in the implementation. *Sort* and *M3* both have DUE as the majority due to the large percentage of branch instructions that can be affected by SEUs in the registers. For *MxM*, it shows a mixture of both as it is a data-intensive application, while some branches in the implementation can also be affected by SEUs to cause DUE.

When comparing results from different configurations, from some applications, the trend of error rate is not so evident since the error rate is affected by several aspects, for example, the number of registers actually utilized during execution, the duration of each data stayed inside registers, etc. However, some interesting effects can be observed. In *FFT*, when the TPB increased from 32 to 64, a DUE decrement can be seen due to the fact the execution time is reduced, while as the number of registers utilized increased, the SDC rate is not decreased as DUE (except when the number of SP is configured to 8. In *Sort and M3*, an growth of DUE rate can be observed when the TPB increase from 32 to 64 as opposite in *FFT*, because *Sort* contains large portion of branch instructions whose execution depends on the value of data so that when more registers are utilized higher the probability of a DUE because of data corruption in register. In *MxM*, this trend of DUE rate increment is not as obvious as in *Sort*, because the branch instructions in *MxM* do not depend on the input data values, though it still depends on the loop variables. These observations are similar to those shown in [40] for applications with a high percentage of control-flow instructions.

Similar trends in different applications can be observed when the number of SPs in SM is increased from 8 to 16 then to 32, for example, the increment of DUE in the *Sort* application. However, the other applications do not present such consistent and visible increase, as when the number of SPs changes, not only the register utilization changes but also the threads have to be organized into warps differently, which causes changes in registers load and store pattern.

5.2.2. Pipeline Registers (PRs)

In total, 144 faults injection campaigns were performed targeting PRs in FlexGripPlus. A total of 30,000 SEUs were injected per configuration. The fault injection results in terms of the averaged fault rate in the entire structure are shown in Fig. 5.

It can be observed from the results that increase TPB will lead to an increment of error rate, including SDC and DUE. This behavior can be explained as the additional time cost for processing warps of the same block increases the probability of an SEU in PR to be propagated through pipeline. Another clear trend that can be observed is that the error rates decrease when more SPs are available in SM. For Timeout, this trend is not consistent across different applications and configurations.

For *M3*, DUE is the majority for all the configurations as it is control-flow oriented application. For *Vector_Add*, as it is data-oriented (without branch instruction), the majority effect land in SDC. For the other applications, it is less obvious.

When analyzing the PRs between different stages, as listed in Table 5, the SEU sensitivity fluctuates from 1.2 to 13.5 times, shown in Fig. 6, which indicates the existence of critical PRs. It turns out the PRs storing the instruction decoding and warp status information are the ones of highest sensitivity against SEU contributing a large proportion of SDC and DUE during fault injection campaigns. In [40], the authors presented similar conclusions when evaluating the PRs of a GPGPU. Although a direct comparison of the results cannot be performed, both works show that the *E-Wr* PR is among the most SEU sensitive PRs, as indicated in Fig. 6.

5.2.3. Warp Scheduler Controller (SC)

The SC module was divided into two parts for analysis purposes: the internal memories and the sequential logic elements. Thirty-six fault campaigns have been carried out.

Contradicting to the criticality of this module, the fault injection

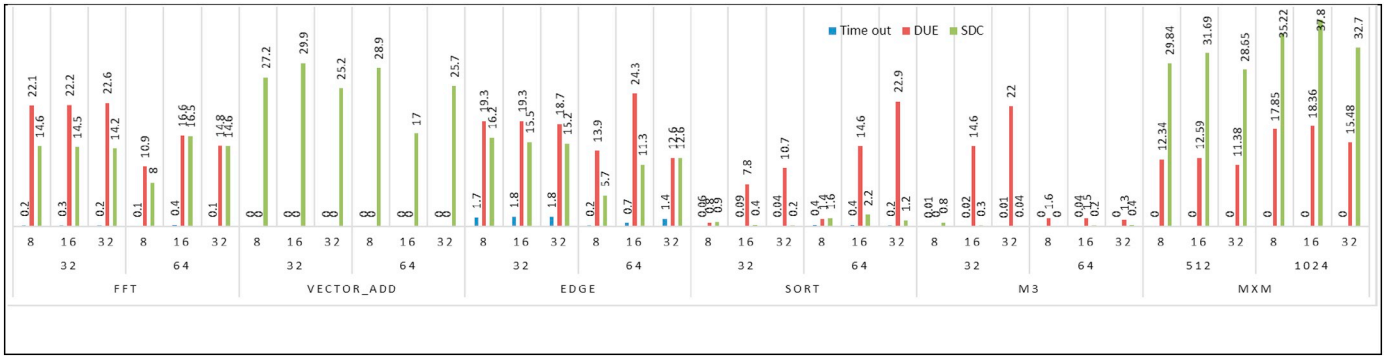


Fig. 4. Fault rate results in the RF in FlexGripPlus (the horizontal axis, from top to bottom, are #SPs, TPB and application name).

campaign results show low sensitivity against SEU, in which, though the sequential logic corresponds to 14.3% of the entire elements in the SC, it generates between 85% and 92% of DUE among the detected faults for all the tested applications.

The unexpectedly low error rate in the SC internal memories is caused by a loop existing between the SC and the SM pipeline stages, which masks the fault effects. The execution of an instruction in the SM requires SC to control and check the execution, which causes the data in SC memories frequently refreshed before it is read after data corruption due to SEU. Nevertheless, in most benchmarks, which are configured with a fewer number of TPB, the fault-masking behavior is effective when the workload is relatively small. However, it becomes ineffective in *MxM* as the probability of data in SC memories read before overwritten after an SEU increases with the workload.

In the operation of the SC, it is expected that an SEU causes a single SDC effect and corrupt one value in the results. However, it was observed that multiple SDC conditions could also be presented in the output memory (i.e., lines, blocks, or random locations with erroneous results). Table 7 reports the percentage of SEUs that causes SDC effects in the SC module divided into those generating single and multiple effects in the memory.

An in-depth analysis shows that the multiple SDC conditions are caused by SEUs affecting the wPC and other fields of the memory in the SC, which are used redundantly by numerous threads (*base addresses employed to access the RF or the shared memory*). Similarly, the multiple SDCs are caused by logic elements in the SC related to the management of the information of the warps.

In general, for all the analyzed applications, errors corrupting the wPC also affected the group of threads and propagate in the execution, so causing multiple SDC. In contrast, faults in the aTM field produce most single SDC effects. In some control-flow-based applications (*EDGE, FFT, Sort,* and *M3*), a small number of multiple SDC were caused by errors present in the aTM field. Those errors modified the execution of one thread in the execution paths and caused additional operations or the missing of operations, which affected subsequent executions in the program flow.

More in detail, the distribution of single and multiple SDC effects differs on the applications and depends on parameters such as the coding style and internal modules employed that are correlated with the SC module. In the *MxM* application, the high percentage of multiple SDCs is caused by the existent connection between the status information of a warp stored in the SC and the RF and the shared memory modules. However, the trend is not present in other applications. *Vector_Add* and *M3* have limited use of the RF and Shared memory, so the contribution of multiple SDCs by misbehaviors in the management of these modules is low.

In other work [48], the authors reported the effect and criticality of SDCs affecting neural network applications in GPGPUs. In results, the authors also included results for the *MxM* application. A comparison between the results of the *MxM* with those introduced in the present work shows equivalent trends. In [48], the authors found that the distribution of SDCs caused by multiple errors in the output lay in the range from 45 to 65%, without error margins. In the listed results in Table 7, the *MxM* applications have an equivalent tendency for the pool memory with a distribution of SDCs in the range of 54 to 65%.

In contrast, the tendency is not followed by the logic part of the SC, and it presents a lower range (27–39%). However, it should be noted that the experiments performed in [48] injected fault in all modules of a GPGPU. In contrast, we perform fault injection campaigns targeting specific parts of the SC module only. In any, case the obtained results show the criticality of the SC module and the susceptibility to SEUs of the *MxM* application.

On the other hand, the trend in the distribution of SDCs for other applications is different. In *Vector_Add*, *FFT*, *Edge,* and *M3*, the trend shows a higher percentage of SDCs caused by single output errors than multiple ones, as analyzed previously.

Furthermore, additional 24 fault injection campaigns have been performed on *Vector_Add,* and *M3* with different TPB configures, and

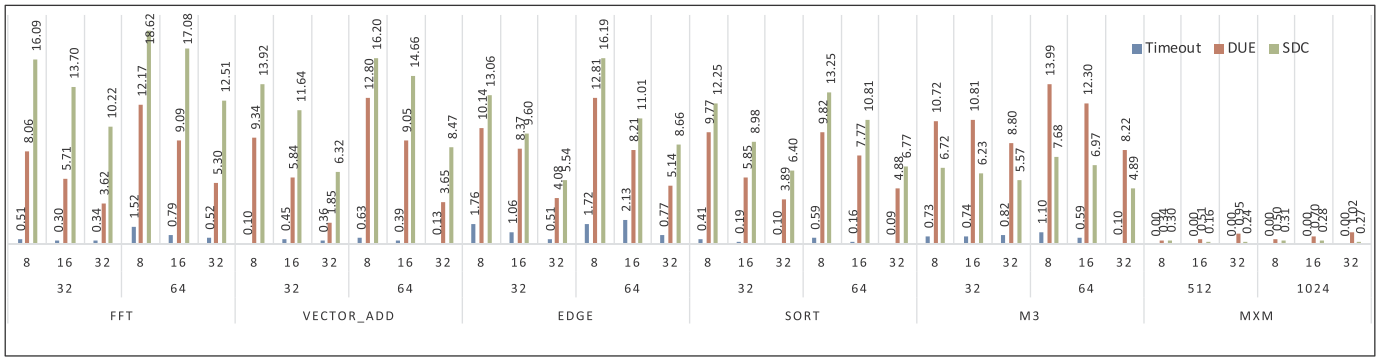
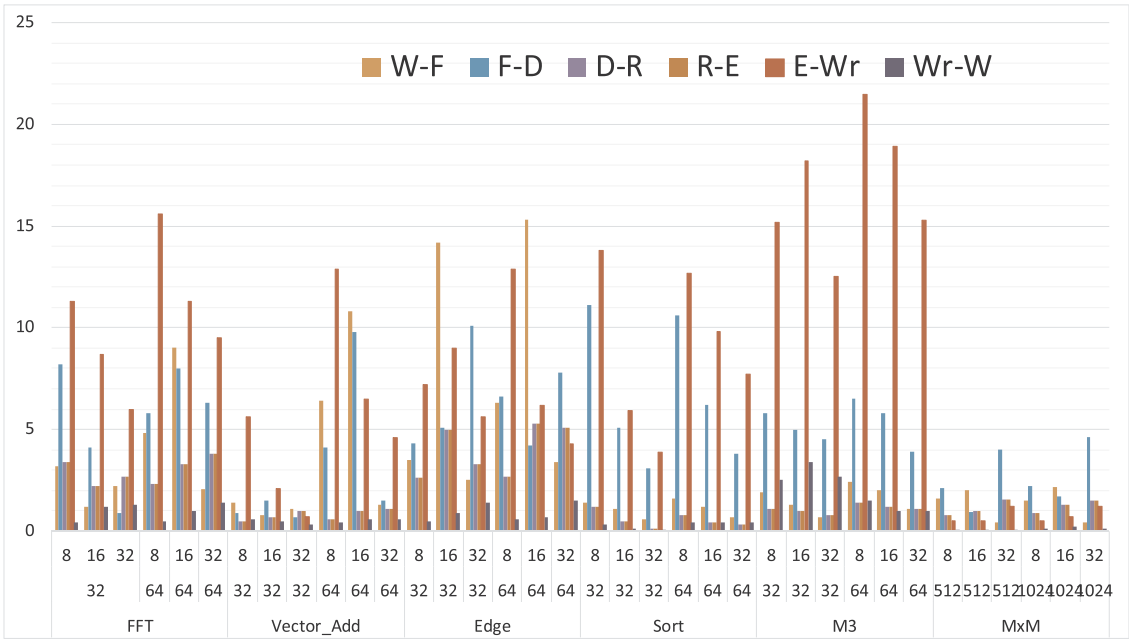
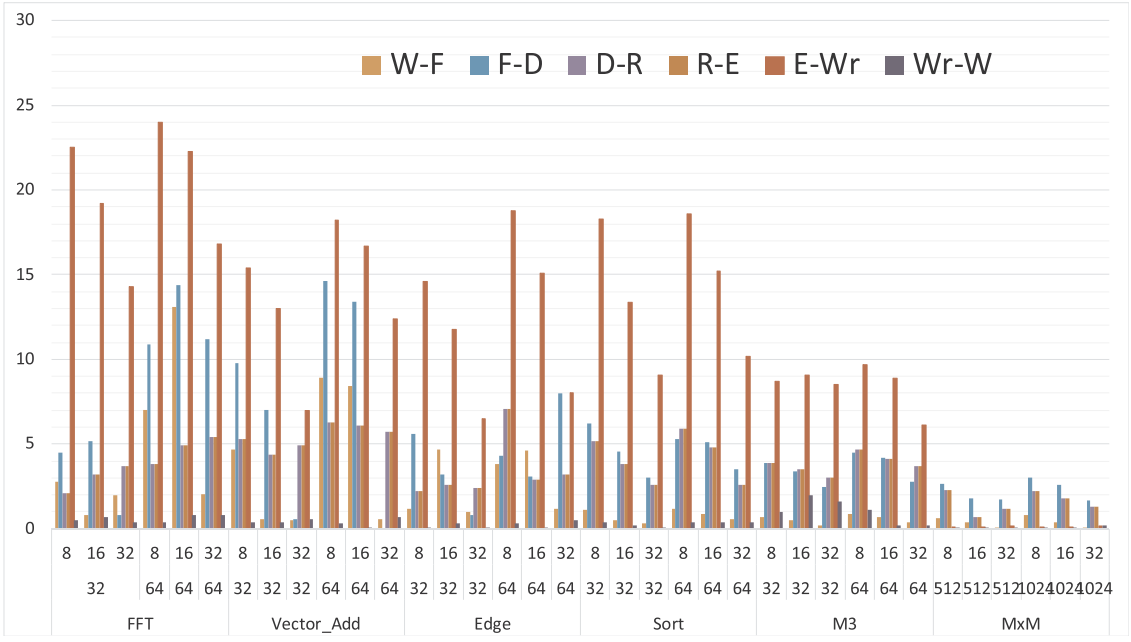


Fig. 5. Fault rate results in the PRs in FlexGripPlus (the horizontal axis, from top to bottom, are #SPs, TPB and application name).



(a) DUE



(b) SDC

Fig. 6. Fault injection results of different PRs between different stages (the horizontal axis, from top to bottom, are #SPs, TPB and application name

the number of SPs fixed to 32. The additional experiments are intended to provide remarks regarding the fault masking effect in the SC memories mentioned before.

The two applications were selected mainly by the distinctive execution behaviors in the SC. *Vector_Add* program includes high data-intensive operations without control-flow or thread divergence operations. In contrast, *M3* is mainly composed of control-flow operations and thread divergence routines. Thus, SC is utilized in the two applications with entirely different patterns. The TPB configurations, used in the experiments, have a range from 32 to 1024 for both applications.

From the results, shown in Fig. 7, when the TPB is configured to be 32 or 64, the fault masking in SC memories is effective to limit the impact of SEUs, though we can still observe a small amount DUE caused by SEUs in the logic part of SC. When the TPB is increased up to 1024, the error rate goes up rapidly, and two different distributions of DUE and SDC can be observed when comparing results from *Vector_Add* and *M3* applications.

Thus, an optimized implementation for performance is, as it often happens, not the best solution when reliability is concerned. Further actions to increase reliability should be adopted, such as ECC in the memory and Triple Modular Redundancy (TMR) in the control logic. Finally, depending on the type of application, different solutions (or in combination) can have effectiveness for improving system reliability against SEU.

Table 7
Distribution of SEU effects in the SC causing single and multiple output effect.

Benchmark	TPB	SPs	SDC (%)					
			Warp Status memory			Logic		
			Single	Multiple	Total	Single	Multiple	Total
FFT	32	8	0.024	0.025	0.049	1.4	1.28	2.68
		16	0.014	0.01	0.024	1.2	1.12	2.32
		32	0.042	0.056	0.098	0.4	0.61	1.01
	64	8	6.53	2.4	8.94	1.1	1.43	2.53
		16	2.19	1.2	3.39	0.7	0.56	1.26
		32	0.12	0.1	0.22	0.64	0.39	1.03
VectorAdd	32	8	1.6	0.54	2.148	1	1.4	2.4
		16	0.3	0.16	0.464	0.9	1.06	1.95
		32	0.06	0.01	0.073	1.10	0.15	1.25
	64	8	6.7	2.43	9.131	1.1	1.59	2.69
		16	3.1	1	4.102	1.2	0.95	2.15
		32	0	0	0	1.01	0.35	1.36
Edge	32	8	0.35	0.18	0.537	0.5	1.14	1.64
		16	0.08	0.06	0.146	0.4	0.51	0.91
		32	0.02	0.03	0.049	0.20	0.51	0.71
	64	8	1.1	1.51	2.612	0.8	1.89	2.69
		16	0.5	1.38	1.88	0.94	1.21	2.15
		32	0.01	0.039	0.049	0.4	0.8	1.2
Sort	32	8	0.002	0.01	0.012	0.03	0.04	0.074
		16	0.002	0.01	0.012	0.05	0.05	0.106
		32	0.01	0.039	0.049	0.10	0.00	0.01
	64	8	0.16	0.31	0.476	0.01	0.025	0.035
		16	0.25	0.33	0.585	0.1	0.27	0.37
		32	0.21	0.36	0.57	0.01	0.005	0.015
M3	32	8	0.41	0.13	0.54	0.3	0.11	0.414
		16	0.28	0.18	0.46	0.12	0.17	0.297
		32	0.14	0.15	0.292	0.23	0.23	0.467
	64	8	0.93	0.6	1.53	0.6	0.21	0.81
		16	1.48	0.7	2.185	0.66	0.24	0.9
		32	1.58	0.65	2.23	0.5	0.49	0.99
MxM	512	8	13.51	20.69	34.2	0.16	0.1	0.26
		16	12.75	18.3	31.05	0.17	0.11	0.28
		32	11.39	13.2	24.59	0.22	0.1	0.32
	1024	8	14.17	25,34	39.51	0.21	0.09	0.3
		16	14.38	23.21	37.59	0.28	0.11	0.39
		32	11.51	21.03	32.54	0.27	0.1	0.37

5.2.4. Divergence Stack Memory (DSM)

In total, 50,688 faults have been injected, targeting *FFT*, *Sort*, *M3,* and *Edge* applications to evaluate the sensitivity of DSM against SEUs.

As the results are shown in Fig. 8, the DSM has a relatively low sensitivity against SEUs when comparing to other modules presented above. One reason for this is that DSM is less utilized in the applications and even fewer cases when multiple branches activate multiple level entries in DSM. However, the general trend shows that a fault affecting the SDM is critical and can cause a DUE collapsing the operation of the system.

Similarly, a change in the SP configuration seems to affect the sensibility of faults in the SDM module. This behavior can be observed in the *FFT*, *Edge,* and *M3* applications under 64 TPB. In each case, increasing the number of SPs is inversely proportional to the susceptibility to SEUs and is explained by the reduction in the management operations performed by the SC for a large number of SP cores in the SM. However, there is not any direct interaction among the DSM and the number of SPs, so the observed reduction in the 64 TPB is mainly caused by the correlation between reduced management operations in the SC and shorter operation times on the routines executed in a divergence path.

Among the four tested applications, *Sort* includes only one conditional control flow instruction generating multiple execution paths. However, the divergence in this benchmark is data-dependent, so the generation of a new path depends on the comparison of two operands from memory. This behavior explains that the error rate did not change so much with different configurations in *Sort*.

The *M3* is also different from others as it intends to generate multiple intra-warp divergences sequentially in the first 32 threads, leading to an intensive switching activity in SC. But it does not generate nesting divergence paths, i.e., it does not use multiple level entries in the DSM. So, when the TPB configuration is changed from 32 to 64, the switching activity in SC is reduced while the level of utilization of DSM due to divergence paths is not increased, leading to decreased error rate, as shown in Fig. 8.

For *FFT* and *Edge*, similar trends can be observed as error rates increase with TPB and decrease with the number of SPs. This behavior is mainly due to the switching activity when the different combinations of TPB and the number of SPs affect the organization of warp execution.

Regarding the distribution of the DUE and SDC error rates, it depends on the affected location within an entry in DSM. An SEU in the wPC field may create Timeout or DUE (or SDC). Similarly, an SEU affecting the aTM field may generate SDC, by interrupting thread execution (i.e., unfinished computation), or DUE by causing threads to miss the synchronization point. Finally, an SEU in the warp ID field produces Timeout effects.

As seen in the comparison between *FFT* and *Edge*, a decrement in TPB can help to reduce more than 50% of the SDC error rate, which is coherent with the conclusion introduced in [47,49].

5.3. General comments

In contrast to previous work, this paper presents fault injection results based on simulation under different configurations involving different numbers of SPs and different TPB configurations, targeting separate modules in the proposed FlexGripPlus. Direct comparison with results of previous works, where the SEUs were injected into a GPGPU device indiscriminately [46–49] or at the instruction level [18] is hard. However, similar trends of reliability impact of SEUs with respect to different configurations, particularly in terms of TPB, can be found. Some cases exist, where opposite trends under certain combinations of settings are observed, as presented in the previous subsections. Results reported in this paper prove that modules inside a GPGPU device can be affected differently when trying to balance performance and system (application) reliability. Hence, different modules will require different approaches to achieve some target reliability figures.

6. Conclusions and future works

In this paper we presented the new FlexGripPlus model, the methodology we followed to develop it, the fault injection simulation environment we used to gather extensive results about the sensitivity of different GPGPU modules to SEUs and different encoding styles, and the results gathered using six applications targeting different modules and with different configurations.

Besides the improvements made towards increasing the set of supported instructions, another significant improvement in FlexGripPlus is the technology independence. This independence allows the usage of the model at a lower level without any limitation related to the targeted gate library and the simulation tool. More importantly, in this way, it is possible to investigate the SEU effects with fault injection techniques targeting specific modules.

Although FlexGripPlus implements the NVIDIA G80 microarchitecture (as inherited from the original FlexGrip model), it includes all principal and critical modules, which are also present in modern GPGPU architectures. Moreover, the compatibility of the model with commercial programming tools allows the use of the same tools as in real application development (with some limitations). Thus, it is possible to perform reliability analysis in FlexGripPlus considering similar modules across generations of GPGPU architectures.

The performed fault injection campaigns provided detailed results about the sensibility to SEUs of individual modules of the GPGPU under different encoding styles (e.g., varying the TPB parameter). The

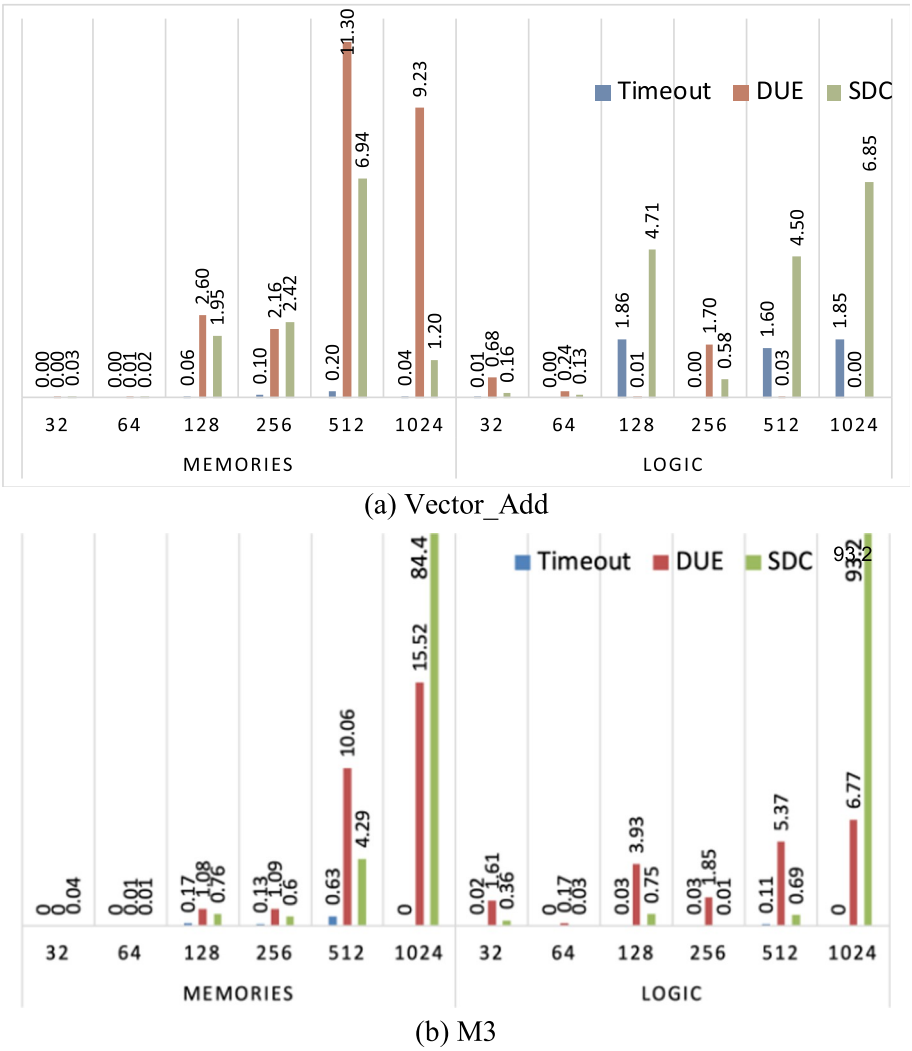


Fig. 7. Fault injection results of SC memories and logic under different TPB configurations (the horizontal axis, from top to bottom, are #SPs, TPB, and application name).

evaluation was performed employing representative applications with diverse workloads to sensitize each module with different patterns. In some cases, it was possible to determine correlations with previous works. However, the existence of some inconsistency across the different applications and configurations prompts for further investigation for evaluating specific modules in GPGPU devices against SEUs.

In general, a major result stemming from the gathered results is that different modules behave in a rather different manner when changing the TPB parameter and show different sensitivity to SEUs. The specific characteristics of each application may further change the above behaviors. Previous results gathered at the GPGPU level could not catch these aspects, which must be taken into account when optimizing an application code for performance, reliability, or in conjunction.

Although FlexGripPlus does not entirely match the architecture of

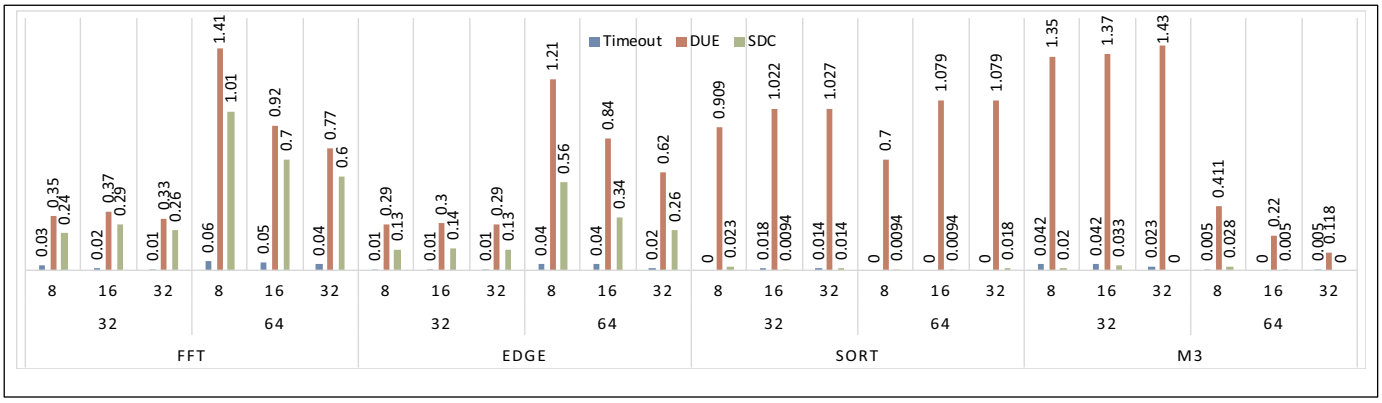


Fig. 8. Fault injection results of DSM (the horizontal axis, from top to bottom, are #SPs, TPB and application name).

the most recent GPGPUs, we still claim that the performed analyses to be valid considering the similarities in structures of modern devices.

As an on-going work, we are currently extending the reliability analysis to other modules within FlexGripPlus. We also plan to further extend the instruction and hardware support of the FlexGripPlus model following the SM 1.0 microarchitecture compatibility, including floating-point units and special functional units into the model.

Authorship statement

All persons who meet authorship criteria are listed as authors, and all authors certify that they have participated sufficiently in the work to take public responsibility for the content, including participation in the concept, design, analysis, writing, or revision of the manuscript.

CRediT authorship contribution statement

Josie E. Rodriguez Condia: Conceptualization, Data curation, Formal analysis, Software, Validation, Visualization, Writing - original draft, Writing - review & editing. **Boyang Du**: Conceptualization, Data curation, Formal analysis, Software, Validation, Visualization, Writing - review & editing. **Matteo Sonza Reorda**: Conceptualization, Data curation, Formal analysis, Methodology, Supervision, Validation, Writing - review & editing. **Luca Sterpone**: Conceptualization, Methodology, Supervision, Validation.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] W. Shi, M.B. Alawieh, X. Li, H. Yu, Algorithm and hardware implementation for visual perception system in autonomous vehicle: a survey, Integration 59 (2017) 148–156 2017/09/01/.

[2] V. Campmany, S. Silva, A. Espinosa, J.C. Moure, D. Vázquez, A.M. López, GPU-based pedestrian detection for autonomous driving, Procedia Computer Science 80 (2016) 2377–2381 2016/01/01/.

[3] S. Kato, S. Tokunaga, Y. Maruyama, S. Maeda, M. Hirabayashi, Y. Kitsukawa, et al., Autoware on board: enabling autonomous vehicles with embedded systems, 2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS), 2018, pp. 287–296.

[4] M. Yang, N. Otterness, T. Amert, J. Bakita, J.H. Anderson, F.D. Smith, Avoiding pitfalls when using NVIDIA GPUs for real-time tasks in autonomous systems, 30th Euromicro Conference on Real-Time Systems (ECRTS 2018), 2018.

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

[6] 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 43 (2015) 297–310.

[7] H.L. Hughes, J.M. Benedetto, Radiation effects and hardening of MOS technology: devices and circuits, IEEE Trans. Nucl. Sci. 50 (2003) 500–521.

[8] E. Ibe, H. Taniguchi, Y. Yahagi, K. Shimbo, 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 57 (2010) 1527–1538.

[9] J. Espinosa, D.d. Andrés, P. Gil, Increasing the dependability of VLSI systems through early detection of fugacious faults, 2015 11th European Dependable Computing Conference (EDCC), 2015, pp. 190–197.

[10] C.Y.H. Lin, R.H. Huang, C.H. Wen, A.C. Chang, Aging-aware statistical soft-error-rate analysis for nano-scaled CMOS designs, 2013 International Symposium onVLSI Design, Automation, and Test (VLSI-DAT), 2013, pp. 1–4.

[11] S.K.S. Hari, T. Tsai, M. Stephenson, S.W. Keckler, 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), 2017, pp. 249–258.

[12] S. Di Carlo, J.E.R. Condia, M. Sonza Reorda, An on-line testing technique for the scheduler memory of a GPGPU, IEEE Access 8 (2020) 16893–16912.

[13] 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, 2014 Design, Automation & Test in Europe Conference & Exhibition (DATE), 2014, pp. 1–9.

[14] S. Collange, M. Daumas, D. Defour, D. Parello, Barra: A parallel functional simulator for GPGPU, 2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, 2010, pp. 351–360.

[15] J. Power, J. Hestness, M.S. Orr, M.D. Hill, D.A. Wood, gem5-gpu: a heterogeneous CPU-GPU Simulator, IEEE Comput. Archit. Lett. 14 (2015) 34–36.

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

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

[18] N. Farazmand, R. Ubal, D. Kaeli, Statistical fault injection-based AVF analysis of a GPU architecture, Proceedings of SELSE 12 (2012).

[19] S. Tselonis, D. Gizopoulos, GUFI: a framework for GPUs reliability assessment, 2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2016, pp. 90–100.

[20] R. De Jong, A. Sandberg, NoMali: Simulating a realistic graphics driver stack using a stub GPU, 2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2016, pp. 255–262.

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

[22] M. Amiri, F.M. Siddiqui, C. Kelly, R. Woods, K. Rafferty, B. Bardak, FPGA-based soft-Core processors for image processing applications, Journal of Signal Processing Systems 87 (2017) 139–156. April 01.

[23] M.A. Kadi, B. Janssen, M. Huebner, FGPU: an SIMT-architecture for FPGAs, *2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, Monterey, California, USA, 2016.

[24] S. Collange, Simty: A Synthesizable General-purpose SIMT Processor, (2016).

[25] J. Bush, P. Dexter, T.N. Miller, A. Carpenter, Nyami: A synthesizable GPU architectural model for general-purpose and graphics-specific workloads, 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2015, pp. 173–182.

[26] K. Andryc, M. Merchant, R. Tessier, FlexGrip: a soft GPGPU for FPGAs, 2013 International Conference on Field-Programmable Technology (FPT), 2013, pp. 230–237.

[27] B. Du, J.E.R. Condia, M. Sonza Reorda, An extended model to support detailed GPGPU reliability analysis, 14th IEEE International Conference on Design & Technology of Integrated Systems in Nanoscale Era (DTIS), 2019.

[28] J.E.R. Condia, M. Sonza Reorda, An extended GPGPU model to support detailed reliability analysis, 15th IEEE Workshop on Silicon Errors in Logic – System Effects (SELSE 15), 2019.

[29] B. Du, J.E.R. Condia, M. Sonza Reorda, L. Sterpone, On the evaluation of SEU effects in GPGPUs, 2019 IEEE Latin American Test Symposium (LATS), 2019, pp. 1–6.

[30] M. M. Goncalves, J. R. Azambuja, J. E. R. Condia, M. Sonza Reorda, and L. Sterpone, "Evaluating software-based hardening techniques for general-purpose registers on a GPGPU," in *21st IEEE Latin-American Test Symposium (LATS2020)*, Brazil, 2020, (to appear).

[31] M.J. Flynn, Some computer organizations and their effectiveness, IEEE Trans. Comput. C-21 (1972) 948–960.

[32] E. Lindholm, J. Nickolls, S. Oberman, J. Montrym, NVIDIA tesla: a unified graphics and computing architecture, IEEE Micro 28 (2008) 39–55.

[33] P.N. Glaskowsky, NVIDIA's Fermi: the first complete GPU computing architecture, White Paper 18 (2009).

[34] NVIDIA, NVIDIA Tesla V100 GPU Architecture, (2017).

[35] J.E. Lindholm, B.W. Coon, J. Wierzbicki, R.J. Stoll, S.F. Oberman, Credit-based Streaming Multiprocessor Warp Scheduling, US20110072244A1, Google Patents, 2015.

[36] J. Knudsen, Nangate 45nm Open Cell Library, https://projects.si2.org/events_dir/2008/oacspring2008/nan.pdf, (2008) Retrieved.

[37] M. Martins, J.M. Matos, R.P. Ribas, A. Reis, G. Schlinker, L. Rech, et al., Open cell library in 15 nm FreePDK technology, *2015 Symposium on International Symposium on Physical Design*, Monterey, California, USA, 2015.

[38] W. J. Van der Laan. (2019, 18/07/2019). Decuda project. Available: https://github.com/laanwj/decuda/wiki.

[39] S.S. Mukherjee, C.T. Weaver, J. Emer, S.K. Reinhardt, T. Austin, Measuring architectural vulnerability factors, IEEE Micro 23 (2003) 70–75.

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

[41] H. Ziade, R.A. Ayoubi, R. Velazco, A survey on fault injection techniques, Int. Arab J. Inf. Technol 1 (2004) 171–186.

[42] D. Alexandrescu, Circuit and system level single-event effects modeling and simulation, Soft Errors in Modern Electronic Systems, Springer, 2011, pp. 103–140.

[43] J.W. Cooley, P.A.W. Lewis, P.D. Welch, The fast Fourier transform and its applications, IEEE Trans. Educ. 12 (1969) 27–34.

[44] B. Du, J.E.R. Condia, M. Sonza Reorda, L. Sterpone, About the functional test of the GPGPU scheduler, 24th IEEE International on-Line Testing Symposium (IOLTS) 2018, 2018.

[45] 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.

[46] P. Rech, L.L. Pilla, P.O.A. Navaux, L. Carro, Impact of GPUs parallelism management on safety-critical and HPC applications reliability, Dependable Systems and

Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on, 2014, pp. 455–466.
[47] P. Rech, T.D. Fairbanks, H.M. Quinn, L. Carro, Threads distribution effects on graphics processing units neutron sensitivity, IEEE Trans. Nucl. Sci. 60 (2013) 4220–4225.
[48] F.F.d. Santos, P.F. Pimenta, C. Lunardi, L. Draghetti, L. Carro, D. Kaeli, et al., Analyzing and increasing the reliability of convolutional neural networks on GPUs, IEEE Trans. Reliab. 68 (2019) 663–677.
[49] D.A.G.G.d. Oliveira, L.L. Pilla, T. Santini, P. Rech, Evaluation and mitigation of radiation-induced soft errors in graphics processing units, IEEE Trans. Comput. 65 (2016) 791–804.