An on-line testing technique for the scheduler memory of a GPGPU

IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received December 27, 2019, accepted January 12, 2020, date of publication January 20, 2020, date of current version January 28, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2968139

An On-Line Testing Technique for the Scheduler Memory of a GPGPU

STEFANO DI CARLO, (Senior Member, IEEE), JOSIE E. RODRIGUEZ CONDIA, (Student Member, IEEE), AND MATTEO SONZA REORDA, (Fellow, IEEE)

Department of Control and Computer Engineering (DAUIN), Politecnico di Torino, 10129 Turin, Italy

Corresponding author: Josie E. Rodriguez Condia (josie.rodriguez@polito.it)

This work was supported in part by the European Commission through the Horizon 2020 RESCUE-ETN Project under Grant 722325.

ABSTRACT The highly parallel processing capabilities and reduced power performance of General Purpose Graphics Processing Units (GPGPUs) have been crucial factors for their massive use in multiple fields, such as multimedia and high-performance computing applications. Nowadays, more demanding areas, such as automotive, employ GPGPU devices where safety and reliability are mandatory design constraints. Nevertheless, the structural complexity, the transistor density, and the implementation in the latest silicon technologies introduce challenges to match safety and reliability requirements. In these technologies, wear-out and aging are factors that may significantly increase the occurrence of permanent faults during the lifetime operation. Moreover, these faults may generate unacceptable misbehaviors during the execution of an application. These constraints require devising new methods for in-field fault detection, thus verifying the integrity and correct behavior of the device during its whole operational life. This work proposes a technique to generate functional self-test programs targeting the detection of permanent static faults in the memory of the warp scheduler of a GPGPU. The proposed technique can translate fault primitives, which represent the effect of faults in a memory cell, into self-test functions and programs composed of a sequence of operations to excite the fault in the memory and to propagate its effects to a visible location, thus detecting its presence. We focused on the memory in the warp scheduler because it represents a crucial module for the device operation. Furthermore, this memory is present in each Streaming Multiprocessor (SM) of a GPGPU. Some experimental results to validate the method have been gathered, resorting to the NVIDIA Visual Profiler and the Nsight Debugger using the NVIDIA-GEFORCE GTX GPU platform and a structural fault simulator. The CUDA programming environment was used to implement the test procedures.

INDEX TERMS Functional test, general purpose graphics processing units (GPGPUs), memory test, software-based self-test (SBST).

I. INTRODUCTION

The General Purpose Graphics Processing Units (GPGPUs) are well-known processing solutions for data-intensive applications, such as those in the multimedia and the High-Performance Computing (HPC) fields, due to their parallel processing capabilities and the relatively reduced power consumption. Nowadays, these devices are relevant for safety-critical applications, especially in the automotive domain. In this field, these devices are crucial components in real-time processing controllers (e.g., Sensor-Fusion platforms and Advanced Driver Assistance Systems (ADAS)) to make decisions based on multidimensional information coming from in-field sensors, such as cameras, radars, and lidars. Unfortunately, the in-field operation of these devices demands high reliability and safety conditions considering that failures produced by any hardware fault may generate unexpected consequences. For this purpose, industry regulations in the field (e.g., ISO 26262) impose a set of strict reliability conditions, including the need for suitable forms of on-line test detection. These mechanisms must verify the system integrity periodically and identify any permanent fault in the system.

Among the multiple test strategies, functional testing based on self-test programs (often known as Software-based Self-test (SBST) [1]) is increasingly common and now widely

The associate editor coordinating the review of this manuscript and approving it for publication was Fan Zhang.

supported by several semiconductor manufacturers and IP providers in the automotive field, such as STMicroelectronics [2], Infineon [3], Microchip [4], ARM [5] and Renesas [6].

The main idea behind SBST is to provide the system integration companies using these devices with software routines (known as self-test procedures) able to detect permanent faults and verify the correct operation of the internal modules.

One classical technique employed to test memory cells using self-test procedures uses *March algorithms*, which are composed of *March elements*. One *March element* is a sequence of reading and writing operations performed on all the memory words in a specified order. Each sequence generates specific pattern values to be written and evaluate those read from memory.

March elements are developed using *Fault Primitives* (FPs), which represent the faults affecting a memory cell, and these are employed to design the test patterns as self-test procedures.

The collection of self-test procedures composes libraries, and commonly the semiconductor companies design and provide them as complementary tools to check the reliability in a device. The Fault Coverage (FC) achieved by an SBST library is computed concerning the structural faults (e.g., stuck-at faults). Moreover, it is possible to combine these routines and the application code.

The self-test procedures are active when required (e.g., at the power-on, power-off, or periodically), generating test patterns and exciting a target module. Moreover, these routines check the test results and trigger an observability mechanism, such as an interrupt flag or a software exception, when a fault is detected.

A similar approach can also be applied when considering ADAS systems. In this case, the integration of GPGPUs (or similar accelerators in terms of features and architecture) is quite common. GPGPUs are composed of numerous parallel Processing Units (PUs). Moreover, GPGPUs include multiple micro-architectural modules also present in CPU based-systems (e.g., the ALU and the Register File (RF)). Thus, it is possible to reuse or adapt techniques originally proposed for CPUs and employ them during the self-test program design.

In contrast, unconventional methods are needed to evaluate critical modules, such as the warp Scheduler Controller (SC), which is specific to GPGPU architectures. Such a module manages, stores and processes the information regarding the operational state of each thread (e.g., enabling or disabling its execution). A micro-architectural analysis of the SC shows that most of its area is devoted to storing information in a status memory. At the end of each instruction cycle, the SC updates the information in the status memory.

This work proposes a method to develop self-test procedures targeting the detection of faults in the memory of the SC of a GPGPU.

The proposed method defines a set of FPs describing static and permanent faults in memory. These FPs are customized for the SC and translated considering the operational restrictions of the memory in the SC. Then, the FPs are used to extract the corresponding test patterns (TPs), i.e., the sequence of reading and writing operations. These TPs maps into high-level self-test routines or functions for the GPGPU, thus generating test programs. Finally, the same mapping and translation process is performed from March elements into self-test routines, thus providing the same fault detection coverage of the original March elements. In the end, this method can translate any element of a March algorithm targeting the status memory of the SC into a self-test procedure. Fig. 1 shows a general scheme describing the proposed approach.

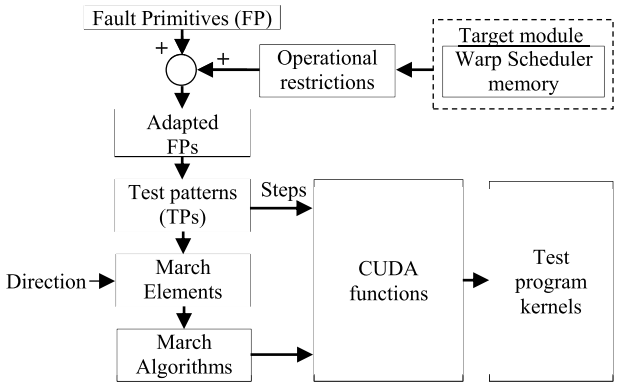

FIGURE 1. General scheme of the proposed approach for mapping TPs targeting FPs into test kernels.

The main contributions of this work include the following:

- The identification of the FPs for all faults (including single and multiple coupling faults) in the status memory of the warp scheduler controller of a GPGPU.
- A method to translate, map and adapt each FP (and associated TPs) or March element into self-test routines and high-level functions targeting the detection of all permanent static faults in the memory cells of the status memory in the warp scheduler of a GPGPU.
- A software mechanism to avoid the operation of the dispatcher units in the SM during the execution of a program kernel in the GPGPU.
- A method to employ the dispatcher units efficiently by using a parallel approach to test the memory in the warp scheduler controller.
- A method to design self-test routines by only using a high-level abstraction language using the CUDA programming environment.

The organization of the paper is the following. Section II briefly describes the internal GPGPU architecture detailing the architecture of the memory in the SC and its functional behavior. Section III summarizes the previous work in the area. Section IV overviews the effects of permanent faults in the SC memory on generic applications. Section V presents the fault primitives (FPs) for a generic memory and describes the main features of the SC memory and its operational restrictions. Section VI defines the methods to generate test

patterns for the target memory employing software-based approaches. Section VII describes the test pattern generation targeting each memory field and

presents a test case algorithm, detailing the general implementation of TPs using high-level functions in the CUDA environment. Section VIII reports on the validation we performed to assess the effectiveness of the proposed techniques and introduces an alternative implementation method for performance optimization. Finally, Section IX draws some conclusions.

II. BACKGROUND

This section provides a general overview of the micro-architecture structure of a GPGPU focusing on the behavior of the warp scheduler and its internal memory.

A. GPGPU MICROARCHITECTURE

In NVIDIA terminology, the micro-architecture of a GPGPU includes several copies of some processing blocks (also known as Streaming Multiprocessors (SMs)).

The SM is the main module inside a GPGPU, and it is optimized to process the same instruction on multiple data sources employing internal execution units (CUDA cores). The controller in the SM manages and traces the execution of the group of assigned threads (also called *warps*). An additional controller (*block controller*) distributes the tasks among the available SMs in the GPGPU.

The basic structure of an SM includes an instruction cache, some logic for fetching and decoding instructions, one warp scheduler, one or more dispatcher units, a register file, Load and Store (LD/TS) modules for memory access, multiple CUDA cores or scalar processors (SPs), and some internal accelerators (Fig. 2).

The classical architecture of an SM includes multiple execution units supporting integer (INT) and floating-point (FP64/FP32) operations. Moreover, modern architectures use accelerator cores, which are specially designed modules for specific tasks or operations, such as Special Functions Units (SFUs) and Tensor cores targeted to perform matrix operations in hardware. Thus, the definition and composition of an SM may vary across technologies and architectures. In some GPGPU architectures, the number of threads in a warp may vary. Moreover, the structure of an SM may include multiple basic SM modules or exclude the accelerator modules.

B. THE WARP SCHEDULER

The warp scheduler controller (SC) is one of the critical modules involved in the execution of an application in a GPGPU. The GPGPU operation starts with a device configuration and is followed by transferring the instructions and the data operands into the device memories. The host performs these previous operations and also manages and monitors the GPGPU operations. After the configuration phase of the device, a block scheduler distributes tasks across the available SMs in the system. The SM executes a task in an SM in groups

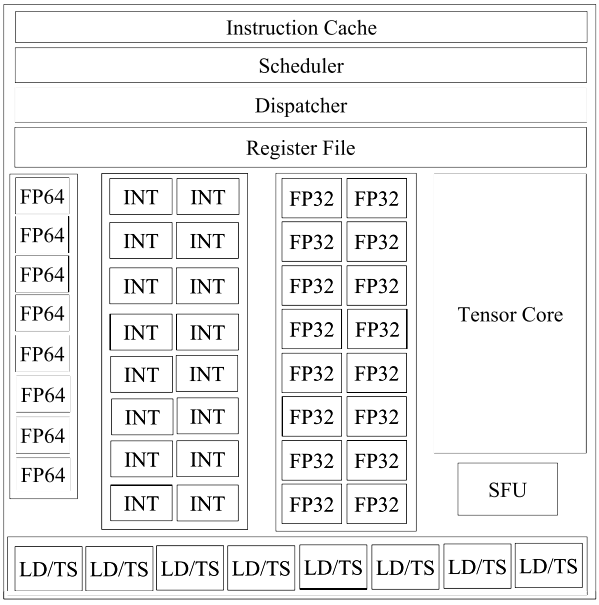

FIGURE 2. A general scheme of the basic processing unit in the SM of a GPGPU (adapted from [9]).

of threads (*warps*) [7] (commonly 32 to 48 consecutive threads). One SC is located inside the SM to control/manage the warp distribution and to verify the task operations [8]. The SC is composed of multiple memories (one principal, and some minor ones) storing updated information regarding warp status execution in the SM. It is essential to clarify that, during kernel operation, all warps share and execute the same instruction. Finally, the host extracts the results from the device memory.

The main SC memory is composed of addressable blocks known as Line-Entries (LEs). Each LE preserves the status information of a thread group during the kernel execution. The SC allocates the information in LEs organized statically and consecutively. The total number of used LEs directly depends on the configuration of an application.

Each LE includes the following fields: a Warp ID field, a Thread Active-Mask (TAM) field, and a Warp Program Counter (WPC) field. Each TAM field may be composed of 48 or 32 bits. The size depends on the GPGPU architecture. In the TAM field, the logical value represents the thread state. Logic "1" corresponds to the active state, while logic "0" corresponds to an inactive thread.

The WPC field is generally composed of 32 bits. Nevertheless, GPGPUs targeting HPC applications include more memory. Thus this field could be composed of up to 35 bits. New designs of the SC module integrate more fields and store the status information for each thread independently [9].

During the device configuration phase, the SC initializes all LEs by setting all the thread states as active. In the execution phase, the SC reads a LE at the beginning of each instruction cycle. Then, the SC updates the information in the LE at the end of the same cycle.

III. RELATED WORKS IN THE AREA

SBST techniques were initially developed for CPU-based systems. Several published works targeted different internal and external modules in these systems. In [1], the authors present a comprehensive overview of SBST techniques for end-of-manufacturing testing. Similar works proposed solutions for other device subsystems, such as cache memories [10], [11], on-chip memories [12], peripherals [13], and communication components [14].

SBST can also be employed for on-line testing of a device, thus verifying the integrity of the internal modules during its operation. However, some key points should be suitably adapted, such as the triggering conditions of the test, the results retrieval, the required resources minimization (e.g., in terms of memory), the test time execution, and the coding style. In [15], the authors introduced some solutions facing the issues raised by the usage of SBST solutions. In [16], the authors described multiple solutions adopted in real industry test cases. Similarly, some algorithms for automatically compacting existing test programs to reduce their size [17]–[19], or duration [20] have been recently proposed. In another work [21], the authors proved that formal techniques could be successfully adopted and used for in-field test program generation for pipelined processors. In [22], the authors explored High-level Decision Diagrams for test program generation. In [23], the authors proposed a dynamic scheduling mechanism to apply SBST during the in-field operation of embedded processors. Finally, in [24], an approach is presented to employ information from multiple abstraction layers for designing efficient SBST programs.

Some other works faced new challenges in the area of SBST, including emerging techniques for developing and optimizing the self-test procedures (STLs) in multicore processors, or targeting other architectures, such as VLIW processors [25].

In [26], the authors proposed an SBST approach to test the execution units in dual-issue processors (multiple instructions executed in the same clock cycle) by synchronizing and duplicating the dispatched instructions, thus forcing each pipeline to execute the same instruction.

In [27], the authors presented a method to test by software a branch calculation unit in VLIW processors by employing a set of routines and checking previously-stored golden results. Moreover, in that work, the calculation unit can be replaced by a spare unit using hardware configuring capabilities of the system. In [28], the authors presented a method to test permanent faults in the execution units and repair the program execution on superscalar processors with static scheduling. Similarly, the authors in [29] proposed a method to employ the implicit parallel architecture of Chip-Multithread (CMT) multiprocessors. This technique targets the multiple execution units in the data-path and manages the functional test by splitting and reducing the total time execution.

The use of SBST techniques on accelerators, such as the GPGPUs, can partly be based on the adaptation of standard strategies initially developed in the past for processors. Nevertheless, the architecture complexity and density of these devices complicate this adaptation. These issues mainly arise by the lack of a well-known ISA format and internal architectural structure information to develop the SBST strategies. Moreover, in these technologies, it is common to employ high-level programming environments to tame the application complexity.

A few works proposed methods aimed to test of GPGPU-based systems. In [30], the authors adapted the CPU testing techniques to GPUs using a high-level programming language approach and combining pseudo-assembly instructions to test the integer and the floating execution units in the Streaming Multiprocessor (SM) cores. Moreover, the same method was partially applied to functionally test the special function units (SFUs) in the GPGPUs. Other works introduced application robustness [31] and mitigation strategies [32] for the data-path modules. Those methods employed combinations of high-level languages and in-line assembly code again.

Similarly, some authors targeted and evaluated the effect of faults in data-path units [33], including the register file [34], and pipeline registers [35]. Other work proposed graceful performance degradation strategies to face permanent faults in SM units by employing specially instrumented kernels and coding styles, thus distributing the tasks across the available SMs [32]. In [36], the authors introduced software methods to avoid corrupted units in the GPGPU by instrumenting the code and modifying the block scheduler algorithms.

The authors in [37] described an initial approach to detect faults and mitigate errors in the GPGPU control units. In that work, intensive experiments and software-based methods partially identified the dispatcher policy of the block scheduler. Moreover, this work introduced some mitigation strategies targeting permanent faults during the in-field execution of the GPGPU. In [38], some first techniques tested and detected permanent faults in the SC of the GPGPU and its internal memory. Finally, authors in [39] recently proposed some methods for periodic on-line testing of the execution units (or Scalar Processors (SPs)) in the SM by generating a divergence path and executing partial workloads of a test program.

A detailed analysis of the previous works on GPGPUs shows that most of them targeted the fault testing on data-path units. Moreover, only a few works proposed solutions to detect faults in control modules. Similarly, to the best of our knowledge, SBST test procedures targeting SFUs in a GPGPU are still open study cases.

In the present work, we remove multiple limitations presented in some previous works [38], [40]. In [38], a set of proposed incremental functional test techniques targeted the detection of faults in single memory cells of the SC. The assembly language of the GPGPU described each proposed method during the implementation phase. In [39], several

functional test techniques detected a wide range of static faults present in single memory cells and by the interaction of multiple cells (CFs). In this work, the proposed methods used high-level functions for the implementation. Moreover, these methods can generate the required patterns to implement any March operation or algorithm [41].

IV. EFFECTS OF PERMANENT FAULTS IN THE SCHEDULER MEMORY

A fault located in the memory of the scheduler can generate misbehaviors, which may seriously compromise the correct operation of the device. Some previous works reported that faults affecting this module may have a critical impact on the system execution and can cause wrong memory results or even system hanging [42]. Nevertheless, those works only targeted transient fault effects.

We performed fault simulation experiments on a GPGPU model using four representative applications to extend conclusions and to evaluate the effects of permanent faults. FlexGrip [43], [44] was selected as a GPGPU model to perform the experiments. This open-source model is described in VHDL and includes the basic modules employed in modern GPGPUs for program kernel execution. Using this model, we could inject a permanent fault in each module location and observe the fault effects during the execution of the selected application.

The model implements the G80 architecture by NVIDIA and supports 27 assembly instructions (SASS) and up to 74 different instruction formats. The model is technology independent, so the ModelSim framework is used to simulate the model.

FlexGrip includes an SC containing a memory which stores status warp information, as in real devices. Each memory location includes the TAM and WPC fields. Some other fields are also included in the memory and are related to the configuration parameters coming from the host or external schedulers.

The selected benchmarks employed in the simulation campaigns are briefly described in the following:

- *VectorAdd:* this kernel corresponds to an embarrassingly parallel application adding two independent vectors of 1,024 elements.
- *FFT:* this kernel is the implementation of the butterfly element and operates on an input vector of 64 elements.
- *Edge:* this kernel implements an image detection algorithm employing the Sobel method using a 3x3 stencil element applied to a 16x16 image.
- *MatrixMul:* this kernel performs the multiplication of two independent 32x32 size matrices.

It is worth noting that, due to the limitations of the FlexGrip model, all the above applications employ integer operands, only.

The fault injection campaign employs the RT level description of FlexGrip. A custom fault simulator was designed using a high-level abstraction language (*Python*) targeting the warp memory in the SC of one SM. The fault injector can place permanent faults corresponding to stuck-at faults affecting single bits of the SC memory. The tool includes a fault controller, a fault injector, and a fault checker and classifier.

TABLE 1. Effects of permanent faults affecting the SC memory.

Benchmark	Faults %				
	SDC	Hang	Timeout	Silent	Cumulative Total
VectorAdd	21.87	35.93	0	42.18	57.81
FFT	34.37	51.56	0	14.07	85.93
Edge	9.37	62.5	7.03	21.1	78.90
MatrixMul	1.9	54.68	0	43.46	56.54

The fault controller manages and configures a simulator tool (*ModelSim*) that holds the GPGPU model. The fault injector decodes the target fault location and translates it into representative commands. Then, the injector sends the commands to the simulation environment, and the fault simulation starts. Finally, the fault checker and classifier identify the fault effects and generate a fault report. The method to inject a fault in the simulation framework follows the approach presented in [34].

The fault injection campaign starts with one fault-free simulation, which aims at characterizing the application in terms of memory results and execution time *(number of clock cycles)*. Then, the fault injector loads a fault list and uses each fault in the list to generate the commands to inject in the model during the simulation of a fault. When the fault simulation finishes, the fault checker classifies a fault depending on the produced effect. A *silent data corruption* (SDC) is detected if the memory results differ from those of the fault-free simulation. The *hang condition* is defined as the case when the system is not able the generate memory results or is not able to finish the program execution. Finally, a fault effect is classified as *timeout* or *performance degradation* when the execution time differs between the fault-free and the faulty case. It is worth noting that memory results comparison has the highest priority in the classification of fault effects.

The fault simulator uses a multi-threading fault injection approach dividing each fault list into two pieces. Finally, eight fault campaigns injected 4,096 randomly sampled permanent faults per campaign. According to [45], the number of injections allows us to reach a 2% error margin on the estimated metrics, with a 99% confidence level. Table 1 presents a summary of fault effects, where the rightmost column reports the cumulative percentage of the faults in the SDC, Hang and Timeout categories. From results, a permanent fault can generate different effects depending on the features of the application and used GPGPU resources.

In general, a permanent fault affecting the WPC field generates mainly hanging conditions. On the other hand, faults affecting the TAM field generate most of the SDC conditions.

A high percentage of faults (56.5% to 85.9%) affecting the scheduler memory produces a failure, proving the criticality of this unit in the operation of the GPGPU. Moreover, these faults may generate unacceptable conditions for complex

TABLE 2. Fault primitives for a single memory cell.

Fault type	Fault Model	FP	AFFP(SCH)
DRDF	Deceptive RDF	$<\bar{A}r_{\bar{A}}/A/\bar{A}>$	$<\bar{A},(W_{\bar{A}},r_{\bar{A}},r_{\bar{A}}),A/\bar{A}>$
		$<Ar_A/\bar{A}/A>$	$<A,(W_A,r_A,r_A),\bar{A}/A>$
TF	Transition fault	$<\bar{A}W_A/\bar{A}/->$	$<\bar{A},(W_{\bar{A}},r_{\bar{A}},W_A,r_A),\bar{A}/A>$
		$<AW_{\bar{A}}/A/->$	$<A,(W_A,r_A,W_{\bar{A}},r_{\bar{A}}),A/\bar{A}>$
WDF	Write destructive fault	$<\bar{A}W_{\bar{A}}/A/->$	$<\bar{A},(W_{\bar{A}},r_{\bar{A}},W_{\bar{A}},r_{\bar{A}}),A/\bar{A}>$
		$<AW_A/\bar{A}/->$	$<A,(W_A,r_A,W_A,r_A),\bar{A}/A>$
RDF	Read destructive fault	$<\bar{A}r_{\bar{A}}/A/A>$	$<\bar{A},(W_{\bar{A}},r_{\bar{A}},r_{\bar{A}}),A/\bar{A}>$
		$<Ar_A/\bar{A}/\bar{A}>$	$<A,(W_A,r_A,r_A),\bar{A}/A>$

applications, and some of these effects can be unacceptable for safety-critical applications. Other kinds of faults (e.g., Coupling Faults) affecting the scheduler memory do produce similar results.

V. FAULT PRIMITIVES FOR THE SCHEDULER MEMORY

A. FAULT PRIMITIVES

Fault primitives (FPs) represent memory failures and are the combination of a sequence of memory operations and the observations, including deviations from the expected value. This sequence of memory operations are employed to sensitize a condition in the memory cell and may also be used to verify and to detect any possible failure in a memory cell.

An FP is composed of one or a sequence of memory operations, which can be writing or reading, and the observed effect on the cell. In reading operations, it is common to include the logic output value as a third element. The FPs may target multiple sets of functional faults that can affect a single memory cell and also couples of interfering cells in memory.

The FPs have been used in the past to define memory test techniques, i.e., March algorithms, able to generate an appropriate sequence of patterns (reading and writing operations), thus testing a memory. The complexity of these algorithms grows proportionally to the size of the memory. Additional details regarding a complete theoretical description of memory functional fault models may be found in [46]. For the purpose of this work, we target the procedures required to translate FPs and March algorithms into functional self-test programs able to detect faults in the scheduler memory of GPGPUs.

1) SINGLE CELL STATIC FAULTS

Considering the set of FPs presented in [46], Table 2 reports the full set of static FPs for single memory cells. The term "static" refers to the fact that they represent faults sensitized by a single memory operation. Each row includes the Addressable Functional Fault Primitive for the scheduler, denoted as AFFP(SCH), and also contains the initialization steps required to sensitize the fault.

In Table 2, the "A" symbol represents a logic test stimulus written in the target bit-field or cell of a LE. Similarly, "$\bar{A}$" is the complementary pattern. Each AFFP organizes as follows:

$$AFFP =< initial\_condition, (Stimuli), Fault\_value, Fault\_Free\_value > \quad (1)$$

For example, considering the DRDF, the FP is expressed as: $FP :< \bar{A}r_{\bar{A}}/A/\bar{A} >$, with $(\bar{A}r_{\bar{A}})$ as the initial sequence of operations in the cell (the initial state and one reading operation), the second element (A) describes the effect of the fault in the cell, and the last item $(\bar{A})$ represents the logic output value of the cell. The associated AFFP(SCH) $(AFFP(SCH) :< \bar{A}, (W_{\bar{A}}, r_{\bar{A}}, r_{\bar{A}}), A/\bar{A} >)$ employs the same initial logic state $(\bar{A})$. The second and third memory operations in the sequence $(W_{\bar{A}}, r_{\bar{A}})$ initialize the cell in the SC. The final reading operation $(r_{\bar{A}})$ is one of the operational restrictions presented in the target memory. The fifth element (A) represents the effect of the fault in the cell, and the last parameter $(\bar{A})$ is the fault-free logic output value and used during the test pattern generation process.

2) COUPLING CELL PERMANENT FAULTS

The coupling FPs are associated with the interaction and effect between two independent cells, an aggressor (a) cell, and a victim (v) cell. These cells can belong to the same LE or not. Table 3 shows the considered FPs. X, Y, and Z represent logic values.

In the proposed approach, the State Faults (SF) FP $< \bar{A}/A/- >< A/\bar{A}/- >$, the State Coupling Faults (CFst) FP$< \bar{A}; \bar{V}/V/- >< \bar{V}/V/- >< \bar{A}; V/\bar{V}/- >< A; V/\bar{V}/- >< A; \bar{V}/V/- >$, and the Incorrect Read Faults (IRF) FP $< \bar{A} >< A >< r_{\bar{A}}/\bar{A}/A >< r_A/A/\bar{A} >$ are not considered, mainly because of the absence of a clear mechanism to cause the initial conditions or to detect these faults in the scheduler memory.

The AFFP(SCH)s in both cases, single and coupling, present some similarities in the associated Sensitizing Operation Sequences (SOSs). Thus, it is feasible to collapse identical patterns. The single-cell AFFPs of RDF and DRDF share the same sensibility patterns. Similarly, some coupling faults (CFrd, CFir, and CFdrd) were grouped using the same SOS, since the only difference among them is the number of consecutive reading operations. Therefore, the SOSs with the lowest number of reading operations were neglected, and the CFdrd SOS is employed to sensitize those coupling faults. In the end, the number of patterns was reduced to 30.

At this point, the AFFP(SCH) is partially complete and must be adapted considering the operational restrictions valid for the SC. The next sub-section describes the operational restrictions of the target memory.

B. SCHEDULER MEMORY OPERATIONAL RESTRICTIONS

As said previously, this memory presents some operational constraints. These are considered in the FP adaptation process and are crucial to generate the test patterns. The memory in

TABLE 3. Static Fault Primitives for coupling cells in memory.

Fault type	Fault Model	FP	AFFP(SCH)
CFtr	Transition coupling fault	$< X^a 0, {W_1}^v \,/\, 0^v \,/\, - >$	$< \bar{X}^a 0^v, ({W_{\bar{X}}}^a, {r_{\bar{X}}}^a, {W_0}^v, {r_0}^v, {r_{\bar{X}}}^v, {r_0}^v)\,, 0^v, X^v >$
			$< X^a 0^v, ({W_X}^a, {r_X}^a, {W_0}^v, {r_0}^v, {r_X}^v, {r_1}^v)\,, 0^v, X^v >$
		$< X^a 1, {W_0}^v \,/\, 1^v \,/\, - >$	$< \bar{X}^a 1^v, ({W_{\bar{X}}}^a, {r_{\bar{X}}}^a, {W_1}^v, {r_1}^v, {r_{\bar{X}}}^v, {r_0}^v)\,, 1^v, \bar{X}^v >$
			$< X^a 1, ({W_X}^a, {r_X}^a, {W_1}^v, {r_1}^v, {r_{\bar{X}}}^v, {r_1}^v)\,, 1^v, \bar{X}^v >$
CFds	Disturb Coupling fault	$< X^a Z^v, {W_y}^a \,/\, \bar{Z}^v \,/\, - >$	$< \bar{X}^a \bar{Z}^v, ({W_{\bar{X}}}^a, {r_{\bar{X}}}^a, {W_{\bar{Z}}}^v, {r_{\bar{Z}}}^v, {W_{\bar{X}}}^a, {r_{\bar{X}}}^a, {r_{\bar{Z}}}^v)\,, Z^v, \bar{Z}^v >$
			$< \bar{X}^a Z^v, ({W_{\bar{X}}}^a, {r_{\bar{X}}}^a, {W_Z}^v, {r_Z}^v, {W_{\bar{X}}}^a, {r_{\bar{X}}}^a, {r_Z}^v)\,, \bar{Z}^v, Z^v >$
			$< \bar{X}^a \bar{Z}^v, ({W_X}^a, {r_X}^a, {W_{\bar{Z}}}^v, {r_{\bar{Z}}}^v, {W_{\bar{X}}}^a, {r_{\bar{X}}}^a, {r_Z}^v)\,, Z^v, \bar{Z}^v >$
			$< X^a Z^v, ({W_X}^a, {r_X}^a, {W_Z}^v, {r_Z}^v, {W_{\bar{X}}}^a, {r_{\bar{X}}}^a, {r_Z}^v)\,, \bar{Z}^v, Z^v >$
			$< X^a \bar{Z}^v, ({W_X}^a, {r_X}^a, {W_{\bar{Z}}}^v, {r_{\bar{Z}}}^v, {W_X}^a, {r_X}^a, {r_{\bar{Z}}}^v)\,, Z^v, \bar{Z}^v >$
			$< X^a Z^v, ({W_X}^a, {r_X}^a, {W_Z}^v, {r_Z}^v, {W_X}^a, {r_X}^a, {r_Z}^v)\,, \bar{Z}^v, Z^v >$
		$< X^a Y^v, {r_X}^a \,/\, \bar{Y}^v \,/\, - >$	$< \bar{X}^a \bar{Y}^v, ({W_{\bar{X}}}^a, {r_{\bar{X}}}^a, {W_{\bar{Y}}}^v, {r_{\bar{Y}}}^v, {r_{\bar{Y}}}^v, {r_{\bar{X}}}^a, {r_{\bar{Y}}}^v)\,, Y^v, \bar{Y}^v >$
			$< \bar{X}^a Y^v, ({W_{\bar{X}}}^a, {r_{\bar{X}}}^a, {W_Y}^v, {r_Y}^v, {r_Y}^v, {r_{\bar{X}}}^a, {r_Y}^v)\,, \bar{Y}^v, Y^v >$
			$< X^a \bar{Y}^v, ({W_X}^a, {r_X}^a, {W_{\bar{Y}}}^v, {r_{\bar{Y}}}^v, {r_X}^a, {r_{\bar{Y}}}^v)\,, Y^v, \bar{Y}^v >$
			$< X^a Y^v, ({W_X}^a, {r_X}^a, {W_Y}^v, {r_Y}^v, {r_X}^a, {r_Y}^v), \bar{Y}^v, Y^v >$
CFwd	Write destructive coupling fault	$< X^a Y^v, {W_Y}^v \,/\, \bar{Y}^v \,/\, - >$	$< \bar{X}^a \bar{Y}^v, \left({W_{\bar{X}}}^a, {r_{\bar{X}}}^a, {W_{\bar{Y}}}^v, {r_{\bar{Y}}}^v, {W_{\bar{Y}}}^v, {r_{\bar{Y}}}^v\right), Y^v, \bar{Y}^v >$
			$< X^a \bar{Y}^v, ({W_X}^a, {r_X}^a, {W_{\bar{Y}}}^v, {r_{\bar{Y}}}^v, {W_{\bar{X}}}^v, {r_{\bar{X}}}^v), Y^v, \bar{Y}^v >$
			$< \bar{X}^a Y^v, ({W_{\bar{X}}}^a, {r_{\bar{X}}}^a, {W_Y}^v, {r_Y}^v, {W_Y}^v, {r_Y}^v), \bar{Y}^v, X^v >$
			$< X^a Y^v, ({W_X}^a, {r_X}^a, {W_Y}^v, {r_Y}^v, {W_Y}^v, {r_Y}^v), \bar{Y}^v, X^v >$
CFrd	Read destructive coupling fault	$< X^a Y^v, {r_Y}^v \,/\, \bar{Y}^v \,/\, \bar{Y}^v >$	$< \bar{X}^a \bar{Y}^v, ({W_{\bar{X}}}^a, {r_{\bar{X}}}^a, {W_{\bar{Y}}}^v, {r_{\bar{Y}}}^v, {r_{\bar{Y}}}^v, {r_{\bar{Y}}}^v), Y^v, \bar{Y}^v >$
			$< X^a \bar{Y}^v, ({W_X}^a, {r_X}^a, {W_{\bar{Y}}}^v, {r_{\bar{Y}}}^v, {r_{\bar{Y}}}^v, {r_{\bar{Y}}}^v), Y^v, \bar{Y}^v >$
			$< \bar{X}^a Y^v, ({W_{\bar{X}}}^a, {r_{\bar{X}}}^a, {W_Y}^v, {r_Y}^v, {r_Y}^v, {r_Y}^v), \bar{Y}^v, Y^v >$
			$< X^a Y^v, ({W_X}^a, {r_X}^a, {W_Y}^v, {r_Y}^v, {r_Y}^v, {r_Y}^v), \bar{Y}^v, Y^v >$
CFir	Incorrect read coupling fault	$< X^a Y^v, {r_Y}^v \,/\, Y^v \,/\, \bar{Y}^v >$	$< \bar{X}^a \bar{Y}^v, ({W_{\bar{X}}}^a, {r_{\bar{X}}}^a, {W_{\bar{Y}}}^v, {r_{\bar{Y}}}^v, {r_{\bar{Y}}}^v), Y^v, \bar{Y}^v >$
			$< X^a \bar{Y}^v, ({W_X}^a, {r_X}^a, {W_{\bar{Y}}}^v, {r_{\bar{Y}}}^v, {r_{\bar{Y}}}^v), Y^v, \bar{Y}^v >$
CFdrd	Deceptive read destructive CF	$< X^a Y^v, {r_Y}^v \,/\, \bar{Y}^v / Y^v >$	$< \bar{X}^a Y^v, ({W_{\bar{X}}}^a, {r_{\bar{X}}}^a, {W_Y}^v, {r_Y}^v, {r_Y}^v, {r_Y}^v), \bar{Y}^v, Y^v >$
			$< X^a Y^v, ({W_X}^a, {r_X}^a, {W_Y}^v, {r_Y}^v, {r_Y}^v, {r_Y}^v), \bar{Y}^v, Y^v >$

the SC cannot undergo any possible operation or sequence of operations. In particular, the following operational restrictions are present (for each of them, we first describe the restriction, then summarize the effects, and finally explain how to take it into account):

1) By default, the initial state of all threads in a warp is the active state. It means that the TAM field starts with a "1" value in all bits. This condition is triggered during the GPGPU configuration phase.

Restriction effect: This condition reduces the available number of test patterns to be applied, and it also forces the addition of an initial condition during test pattern injection, thus reducing the performance of the test.

Potential solution: It is not possible to avoid the initial conditions. Nevertheless, it is possible to split the target test fields during a test procedure and apply multiple test patterns replacing the effect of the missing one. Multiple injections procedures are required for this purpose.

2) In the TAM field, when a warp ends the execution of an instruction, the SC updates the content of the associate LE including an implicit reading procedure. In contrast, writing procedures are present each time an instruction starts its execution or when the application flow changes the number of active threads in a warp.

Restriction effect: This condition complicates the injection of consecutive specific writing and reading operations in the LEs. The implicit reading operation may increase the complexity of the injection of some specific operations in this field.

Potential Solution: In this memory, it is not possible to avoid the operational restriction. However, this restriction does not affect the testing procedures of static coupling faults.

3) Stimulus applied into the TAM field can generate thread divergence causing a program-flow division into two paths (the Taken and the Not-Taken). The SC manages these paths and executes them serially.

Restriction effect: This condition adds some undesirable writing and reading operations and latency. Moreover, initialization procedures may be required between pattern injections of consecutive patterns.

Potential Solution: This restriction cannot be avoided. Nevertheless, the additional path (Not-Taken) can be neglected for the coupling faults test and considered as a transition of the initialization phase for the next pattern injection.

4) The execution of a divergence path (Taken or Not-Taken) is not stoppable when a thread group is operating in the SM resources. Moreover, the use of conventional functions has no effect on stopping these threads.

Restriction effect: This condition involves the development of additional mechanisms to manage the stopping and restarting operations of warps belonging to LEs. Moreover, this behavior imposes restrictions on test patterns selected for the TAM field. Finally, many writing and reading procedures are required to test the field, considering that the imposed

restrictions limit the injection pattern, and at least one active state bit should be maintained during evaluation.

Potential Solution: A method to stop the warp execution lies in resorting to some thread synchronization methods (i.e., __syncthreads()) and semaphore variables to identify the warp state. Nevertheless, at least one thread must remain in the active state to retain the warp in a hanging condition with the possibility to restart it again. On the other hand, a LE with the TAM field filled with inactive threads denotes a finished warp and cannot be relaunched for this program kernel.

5) The GPGPU has two warp dispatcher units. These units dispatch the available warps into the SM resources based on performance and optimization features. Moreover, dispatchers use complex data-hazards mechanisms and elaborated dispatching policies. Thus, the execution order of a group of warps may be complex to predict. Furthermore, clear structural descriptions and internal operational details are not provided by the device manufacturer [7]–[9].

Restriction effect: This behavior may compromise the execution of test patterns in consecutive LEs and adds latency among the injection of patterns into the memory LEs.

Potential Solution:software-based mechanisms can skip the operation of these modules to obtain the expected interaction behavior among consecutive LEs during test injection. This mechanism adds semaphore variables and internal loops to maintain the state condition of a target warp and the associated LE.

C. ADAPTING FPS TO TEST THE SCHEDULER MEMORY

We can identify a test pattern for each one of the considered FPs. This pattern is directly derived from the associated AFFP. The following example shows the steps required to generate and adapt the test patterns from the associated FP. Consider the FP:

$$FP :< X^a Z^v, W_y^a/\bar{Z}^v/- > \tag{2}$$

In this example, this FP describes one disturb coupling fault between two cells (a) and (v) with an initial logic state X and Z, respectively. A writing process in the aggressor (a) cell generates the logic toggle of the victim (v) cell. The associated AFFP is:

$$AFFP :< X^a Z^v, W_x^a/\bar{Z}^v/Z^v > \tag{3}$$

From the AFFP, it is possible to derive an initial test pattern (TP). The TP adds the condition to check the state of the (v) cell and is:

$$TP :< X^a Z^v, (W_X^a, r_Z^v) > \tag{4}$$

The first two terms in (4) represent the initial logic state in (a) and (v) cells. The consecutive terms describe the patterns to evaluate the fault. Thus, the TP starts a writing process in the (a) cell to generate the fault condition, and then, a reading procedure is performed on the (v) cell to verify the fault effect in the affected cell. This TP is valid for general-purpose memories with regular writing and reading procedures. However, the special purpose memory, in the SC, includes a set of constraints presented in the previous section; thus, to include the operational restrictions and initialization conditions into the AFFP and TP, some additional procedures are placed as a part of the pattern. Equations (5) and (6) present the adapted versions of the AFFP and TP for the SC memory.

The additional memory procedures in (5) and (6) were selected as follows: the first four terms in parenthesis, in AFFP(SCH), and TP(SCH), describe the initial logic states for both cells. The second and fourth terms are the unavoidable reading operations on each cell of the SC memory by the operational restrictions. The fifth term represents the sensitive operation of the FP. The sixth term is the implicit reading procedure due to the previous operation. Finally, the seventh parameter, in TP(SCH) is the additional operation to observe the failure in the memory cell.

$$AFFP(SCH) :\prec X^a Z^v, (W_X^a, r_X^a, W_Z^v, r_Z^v, W_{\bar{X}}^a, r_{\bar{X}}^a), \bar{Z}^v, Z^v > \tag{5}$$

$$TP(SCH) :< X^a Z^v, (W_X^a, r_X^a, W_Z^v, r_Z^v, W_{\bar{X}}^a, r_{\bar{X}}^a, r_Z^v) > \tag{6}$$

It is worth noting that during the scheduler operation, it is possible to perform reading operations in the memory for the TAM and WPC fields. Considering the TAM fields, the only condition is that each bit field maintains the same logic value. On the other hand, the WPC changes by executing a new instruction. Thus a reading operation can be partially performed. These toggling bit fields cannot be included in the previous pattern, so the previous pattern should be applied more than once to test the WPC field thoroughly. As explained below, the TP(SCH) can be directly derived from the associated AFFP(SCH), since the same writing and reading operations are performed. Tables 2 and 3 present the complete list of AFFP(SCH)s.

VI. METHODS TO GENERATE SBST PATTERNS FOR THE SCHEDULER MEMORY

As introduced in Section II, the SC memory is a special purpose memory and includes some restrictions in terms of writing and reading operations. Moreover, the accessing method to each LEs cannot follow the conventional procedures used in processors to access data memories. Thus, alternative methods should be proposed to inject test patterns based on writing and reading operations.

In this work, we focus on detecting permanent faults and static coupling faults in the WPC and the TAM fields of the SC memory.

This section presents the software methods employed to perform reading and writing operations in the SC memory for the TAM and WPC fields.

A. THE SC MEMORY BEHAVIOR TO READ AND WRITE OPERATIONS

The in-field controller operation imposes restrictions when perform writing ($W_{(x)}$) and reading ($R_{(x)}$) sequences in the

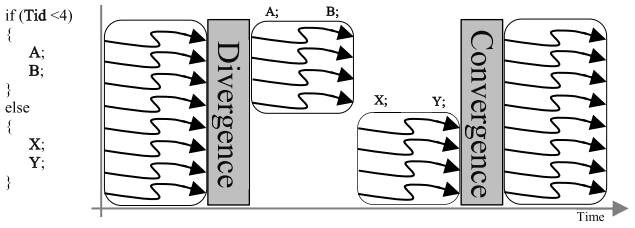

FIGURE 3. A general scheme of the classical intra-warp divergence management for the NIVIDIA's Pascal and previous GPGPU architectures.

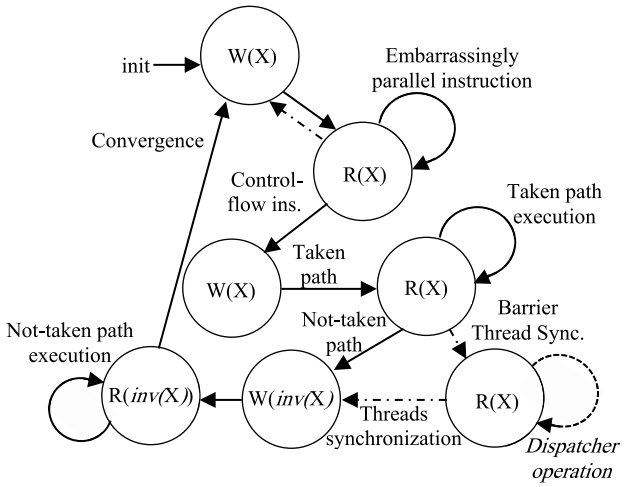

FIGURE 4. A general scheme of the write and read operations generated by a divergence path function in a bit of the TAM field. The nesting divergence case is not considered in the scheme.

SC memory. These restrictions depended on the target field in the LE, as presented in section IV.B.

The methods to perform $W_{(x)}$procedures into the WPC and the TAM fields are similar among them and use control-flow instructions.

Conditional control-flow instructions can produce a writing operation in the TAM field. However, their execution may generate thread divergence creating two paths (the *Taken* and the *not-Taken*), which are managed by the warp scheduler and are executed independently in a serial fashion until reaching a convergence. Then, the threads are executed again in parallel (see Fig. 3). Thus, the execution of one conditional control-flow instruction generates a process where all the instructions in the first thread path (*Taken*) are executed, generating a $W_{(x)}$, followed by the execution of all instructions in the second path (*Not-Taken*), forcing an inverse writing $W_{(x')}$ operation on the same bit field. New architectures include deeper granularity and can process divergence threads independently by storing more parameters into the SC memory [9], thus partially reducing the latency of divergence thread paths.

The time execution of each writing procedure on each path directly depends on the total number of instructions in the path. For the TAM field, the reading sequence is implicitly added after the execution of each instruction. Fig. 4 represents the effect of a conditional control-flow instruction which modifies the state of one-bit of the TAM field in terms of $W_{(x)}$ and $R_{(x)}$ operations. The divergence instruction can generate writing operations of both logic values (1 and 0) on the same bit field consecutively.

The access to the WPC field in a LE employs sequential execution of instructions. Nevertheless, this execution is a naïve method, and some high-part fields are complex to stimulate. In a GPGPU, each warp employs the same program counter to fetch and execute an instruction. Unconditional control-flow instructions to specific locations can be used to generate the test patterns in this field. Taking into account that each warp executed by the SM resources employs a shared program counter, it is required to add mechanisms to stimulate these fields. Nevertheless, the same behavior of the TAM field is also presented in this field, and $R_{(x)}$ operations are generated after a $W_{(x)}$ sequence.

We consider the divergence management mechanism implemented in NVIDIA Pascal and previous architectures to face the TAM fields [47].

VII. TEST PATTERN GENERATION

The generation of TPs for the targeted memory considers and analyzes each possible restriction caused during the mapping process from FPs into software functions.

Regarding operational restrictions, which were introduced in section IV.B, the first and the second ones cannot be avoided, and test patterns targeting the TAM fields must face the starting condition of all threads active (*all bits equal to 1*). Some FPs require initialization conditions of bits in logic 0. Thus additional patterns must be generated and applied before starting the test sequences for the target FPs.

The second restriction describes the impossibility of performing a $W_{(x)}$ without one consecutive implicit $R_{(x)}$ operation at the end of each instruction cycle. The FPs definition, design, and implementation TPs must include those consecutive $W_{(x)}$ and $R_{(x)}$ procedures.

Regarding the third listed constraint describes the impossibility to stop a divergence path when it started. A technique to control and reduce the effect uses the selection of a limited number of operations and control-flow functions presented on each path. A set of input patterns (see Table 4) may be employed to divide the self-test program into chunks. These test patterns are carefully selected to increase the fault detection inside and among LEs.

The third restriction arises during the execution of a not-taken path. Initially, the SC submits threads executing the divergence taken-path. Then, the SC inverts the state of the threads in a warp activating those that were inactive and inactivating those that executed the taken path. In this way, an inverse $W_{(x)}$ operation, in the TAM field, starts the not-taken path execution. The not-taken path can be temporary skipped or delayed by the addition of nested divergence paths. These nesting conditions introduce additional operations in a procedure. Nevertheless, these do not affect the test pattern generation or the adaptation of a March operation into high-level functions.

TABLE 4. Selected test patterns for coupling fault detection in a LE.

Test Pattern	Description
1111…0000… / 0000…1111…	First-half X, second-half $\overline{X}$
00001111... / 11110000...	First four bits X, second four $\overline{X}$
0011…0011… / 1100…1100…	First two bits X, second four $\overline{X}$
1010…1010… / 0101…0101…	Alternated X and $\overline{X}$
11111111111111111111…	All in ones

Concerning the fourth constraint, the previous test patterns must remain at least one thread (*bit*) in the active state (*logic 1*). This condition explains the missing pattern (all in 0s) in Table 4.

The fifth restriction (warp control issues by dispatcher units) is faced by using combinations of thread synchronization functions, i.e., *thread barrier instructions*, semaphore mechanisms (*local variables*), and control-flow loops. These elements are placed in strategic locations in a function to stop/skip the operation of the dispatchers and to control the warp submission into the SM resources. The loops retain the state of LEs during a stopping condition.

This technique can hang the operation of an active warp temporarily and dispatch a desired one. It is worth noting that some undesired consecutive $R_{(x)}$ operations may be a product of the skipping method for the dispatchers. Thus, adding latency in the program execution. On the other hand, these $R_{(x)}$ operations in a LE do not produce consequences in the test pattern generation and the March operation or algorithm adaptation, as previously described.

During SC operation, this unit manages the content of the WPC and the TAM parameters employing $W_{(x)}$ operations, which are the product of control-flow instructions (*conditional* and *unconditional*). Considering that in most GPGPU technologies, the divergence paths are serially executed in different operation cycles, the proposed technique uses the taken-path as the principal path for test pattern generation. Thus, the inverse $W_{(x)}$ operations are neglected and are not used in most of the cases.

The next section proposes and details a method to skip the dispatcher operation employing the mechanisms introduced below. The consecutive sections present the sequence of steps to produce $W_{(x)}$ and $R_{(x)}$ procedures into the WPC and the TAM parameters of a LE.

A. A METHOD TO SKIP THE DISPATCHER UNIT OPERATION IN THE SM

The basic operation of the dispatcher units is the management and warp submission to the SM resources. The operation of this unit depends on multiple parameters to increase performance. Those parameters include the application coding style, data operands sources, internal hazards, and availability. Moreover, the dispatchers also consider the instruction conflict and latency and internal warp distribution policies; thus, in general, warps are not dispatched following a sequential approach with a direct incidence in the LEs. This operation restricts the application of test patterns on (*a*) and (*v*) cells. Thus its operation should be controlled or avoided.

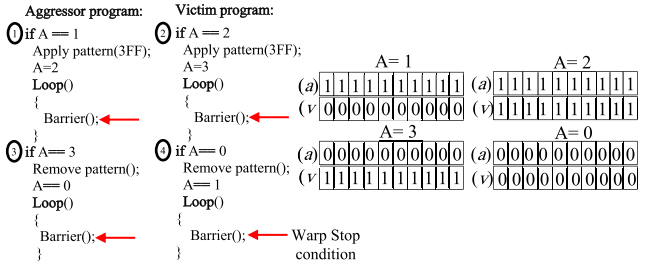

FIGURE 5. Example of the method used to skip the operations of the dispatcher unit employing semaphores and synchronization functions between consecutive aggressor and victim cells.

The proposed method employs software approaches to start and stop the warp execution and keep the belonging LE state. It is worth noting that the main idea is to stop the execution of an active warp temporarily instead of entirely terminate the warp operation and sequentially inject patterns to keep the coherency of the TP injection. Moreover, a set of variables that behave as semaphores and thread synchronization checkers contribute for this purpose as control and communication mechanisms among threads. Finally, the same kernel procedures operate in (*a*) and (*v*) cells.

Fig. 5 represents a basic example of the mechanism employed to skip the operation of the dispatcher. In this example, a kernel program executes concurrently warps corresponding to consecutive LEs behaving as (*a*) and (*v*)cells. The kernel includes one external comparison parameter (A). Moreover, (A) is shared between both warps. As shown in the example, the injection pattern and the starting cell can be selected by changing the (A) value in the semaphore.

The process starts with a warp selection through (A) value comparison. In this example, (A) begins in 1, and the warp corresponding to the (*a*) cell is selected to start its execution. Then, a pattern is applied (3FFh), the semaphore variable is updated with (A=2), and the barrier instruction stops the warp.

The dispatcher picks any other available warp and dispatches it to the system. However, if the submitted warp is not the expected one, a loop mechanism stops it again. It should be clarified that a previously stopped warp continues its execution after the last executed instruction before the stopping condition. Employing this mechanism, the target (*v*) warp, with an A=2 state, can start the execution in an ordered manner.

Once (*v*) is dispatched, the semaphore condition is true, and the pattern is injected on the cell, followed by a new update in the semaphore value and a barrier function. This last instruction restarts the dispatcher operation and picks any available warp again.

In final steps, the dispatcher submits the warp corresponding to (*a*) again and enters in a path to remove the pattern. This pattern can be the Not-Taken path or an additional nesting path for a new pattern injection. Moreover, the semaphore is again updated. Finally, the dispatcher submits (*v*) and removes the pattern. In the end, the cells are ready to start

a new pattern injection. The loop requires an additional local variable to calculate the number of loop execution times. It is worth noting that we did not consider the Not-taken path to include operations related to the injection of new patterns.

B. TEST PATTERNS FOR THE TAM FIELD

The scheme in Fig. 4 describes the sequence of steps and stimulus instructions to generate patterns in one LE. The following steps are derived from this scheme and perform $W_{(x)}$ procedures in the target parameter (TAM).

This method presents effectiveness in evaluating some coupling faults between two cells belonging to different LEs (CFir, CFdrd, and CFrd).

1) SEQUENCE OF STEPS FOR PATTERN GENERATION IN A SINGLE CELL

1) Execute an embarrassingly parallel function (**F**) (*Initial condition*).
2) Execute a divergence generator function ($W_{(x)}$ *in the target bit(s) of TAM*).
3) Execute the taken path ($R_{(x)}$*in the full TAM*).
4) Execute the not-taken path *(Inverse* $W_{(x)}$ *and* $R_{(x)}$*procedures on selected bit(s) field*).
5) Convergence point execution (**CP**) (*parallel execution of instructions and implicit*$R_{(x)}$*procedures*).

The previous steps form the basic $W_{(x)}$ procedure into a LE. On the other hand, the proposed method may neglect the Not-Taken path for test patterns generation. Thus, step 4 can be ignored, for the purpose of test pattern generation, and is mainly employed as a connection to start a new March operation.

In the Taken Path, of a thread group, additional functions are added to stop and hang the warp execution. Then, the warp scheduler selects an available warp, and the dispatcher launches it in the SM resources. The CFdrd, the CFrd, and the CFir coupling faults can be suitably tested by means of both divergence paths, due to the number of $W_{(x)}$ and $R_{(x)}$ operations involved. Nevertheless, some additional steps must be added to test a bigger number of coupling fault sets.

2) SEQUENCE OF STEPS FOR PATTERN GENERATION IN MULTIPLE CELLS

The detection of coupling faults, among cells of different LEs, requires more steps, including synchronization operations, nesting divergence functions, and warp selection routines to assure the correct pattern generation and evaluation of each fault set. The following steps are based on the scheme presented in Fig. 4, and the interaction between two cells. Experimental observations were used to determine the interaction among cells. The following list describes the suggested steps to test this type of coupling faults:

1) Execute an embarrassingly parallel function (**F**).
2) Select a target warp or LE ((a) cell).
3) Execute the first divergence function (W(x) operation in the target bit(s) field of (a) cell).
4) Execute the taken path of divergence (R(x) the full TAM field for (a) cell).
5) Execute a barrier function in the (*a*) cell path, thus stopping the warp execution and launching a new warp.
6) Select a new target warp (*(v) cell*).
7) Execute a second divergence function ($W_{(x)}$ *in the target bit(s) of a (v) cell).*
8) Execute the taken path for the second divergence ($R_{(x)}$ *operations of the full TAM field in the (v) cell).*
9) Execute a barrier function in the (*v*) cell path, launching a new warp.
10) Execute the not-taken path for the (*a*) cell *(inverted* $W_{(x)}$ *operation in the target bit(s) of the TAM field, followed by* $R_{(x)}$ *operations).*
11) Execute a barrier function in the (*a*) cell path.
12) Execute the not-taken path for the (*v*) cell *(inverted* $W_{(x)}$ *procedures in the target bit(s) of the TAM field, followed by*$R_{(x)}$ *operations).*
13) Execute a barrier function in the (*v*) cell path.
14) Execute the convergence point (**CP**).

The step range, from 9 up to 13, originally is part of the neglected operations required to start a new test pattern sequence. Nevertheless, these steps potentially can be employed to test additional coupling fault conditions in the TAM parameter *(CFtr, CFds).*

In the first divergence, an external pattern is applied to the target (*a*) cell. This pattern divides the warp by selecting the active threads during the evaluation of a (*v*) cell. This division is performed by a divergence which splits the warp into two equal groups of consecutive threads (group 1: 0 to 15 and group 2: 16 to 31). Nevertheless, the functions must be evaluated independently, in both groups, to cover all coupling faults.

In this method, the $R_{(x)}$ operations are integrated with the $W_{(x)}$ ones. In the simplest case, $R_{(x)}$ operations can be performed by the execution of non-control-flow instructions during the execution of a warp. In the TAM field, one $R_{(x)}$ operation is presented when the number of active threads, in a warp, is the same after executing one instruction. Similarly, for the WPC field, the execution of instructions can maintain the value in most fields, but for the WPC parameter, some bits should be neglected during the fault evaluation.

As introduced previously, $R_{(x)}$ operations are presented after each instruction cycle in both targeted fields. Moreover, these cannot be avoided. The SC checks the LE continuously at the starting and ending points of each instruction cycle to preserve the SM coherency during the warp operation. Other halted and stopped LEs are read when those turn into the active state by the SC management.

Step 8 describes the addition of a nested divergence function. This nested divergence function is effective in retaining a warp in an active state and maintain at least one active thread in the TAM field, thus limiting the effect of the fourth constraint in the memory. Moreover, this is suitable to detect the coupling faults of the groups (CFir and CFrd). The divergence

function can be located after steps 4 or 10, considering an (*a*) cell. Similarly, this function can also be placed after step 8 for a (*v*) cell.

A third nested divergence function is required to detect the faults in the CFwd set. The new divergence sensitizes some missing conditions and guarantees the required $W_{(x)}$ operations in the memory. The third nested divergence function operates selecting only the thread 0 (first) or the thread 31 (last) for each path. This divergence should be applied in both cases also to test coupling faults in those fields. Some barrier instructions are added on each path for synchronization purposes. In the sequence of steps, the 14th step is replaced and the 4th, 5th, 10th, and 11th are added to generate a new operation on the (*a*) cell. Thus, the third nested divergence function is executed during the second nested divergence, in the not-taken path.

This sequence can be used to evaluate any interaction among the (*a*) and (*v*) cells. Finally, barrier functions and shared variables may be placed on the steps to control the displacements across the LEs in any memory direction*(dropping or incrementing)*.

C. TEST PATTERNS IN WPC

The proposed technique considers those GPGPU architectures with a shared WPC parameter in all threads of a warp. In this strategy, the test program design and execution must target the highest possible parallelism. Thus the approach avoids thread divergence to maintain the parallel execution of the threads in a warp. For this purpose, a set of routines are located in specific and strategic addresses of the system memory. The target memory addresses, for each routine, are based on the test patterns introduced in Table 4.

Each routine is mainly composed of embarrassingly parallel operations and thread barriers. The primary function calls each routine using unconditional control-flow instructions. Inside the routines, a set of barrier functions halts the operation of a warp and starts the execution of a new one. The same method used to select the warps, in the TAM parameter evaluation, is also used for testing the WPC parameter.

Some WPC bit fields, in the highest part of the memory, are difficult to evaluate by the complexity to locate the test routines. The inclusion of additional GPGPU functions and program kernels in the memory solves the test location issue.

The host locates these functions and programs in memory during the configuration of the device. Finally, the method is flexible, and the implementation may consider division into pieces to evaluate specific WPC fields through independent kernels in short execution times.

1) SEQUENCE OF STEPS FOR PATTERN GENERATION IN SINGLE CELLS

The following sequence of steps describes the procedures of writing and reading into the WPC parameter to evaluate a single cell.

1) Execute an embarrassingly parallel instruction (F) ($R_{(x)}$ *procedures)*.
2) Execute an unconditional control-flow routine *(calling a function and* $W_{(x)}$ *and* $R_{(x)}$ *procedures)*.
3) Return from the routine, and then compare the signature. Start a new call to another routine ($W_{(x)}$ *and* $R_{(x)}$ *procedures)*.
4) *(Restart and repeat steps* 2 *and* 3 *when required)*.

As previously discussed, $R_{(x)}$ operations are integrated into the $W_{(x)}$ operations in the target memory. For each memory parameter (TAM or WPC), the processing thread computes a signature *(d_signature)*. This parameter is an observation mechanism to evaluate and detect any fault in the LE. Moreover, a mismatch in the signature represents the presence of a fault in the memory cell. Finally, the method evaluates a signature at the end or in the middle of each test program routine.

2) SEQUENCE OF STEPS FOR PATTERN GENERATION IN MULTIPLE CELLS

As previously introduced for the TAM parameter, the test pattern generation of coupling faults among LEs requires more elaborated steps. The (*a*) and (*v*) target cells are carefully chosen by the warp selector mechanism, which may generate divergence, in case of intra-warps cells, or not when the cells belong to different LEs. The following sequence of steps can generate test conditions and patterns.

1) Execute an embarrassingly parallel instruction (F).
2) Execute unconditional control-flow operations calling and selecting an (*a*) cell ($W_{(x)}$ *operation in the (a) cell)*.
3) Execute the routine and a barrier instruction in the path of the (*a*) cell launching a warp containing the (*v*) cell ($R_{(x)}$ *operation in the target (a) cell of a LE)*.
4) Selection of the target (*v*) cell and execution of unconditional control-flow operations ($W_{(x)}$ *operation in the (v) cell)*.
5) Execute the routine and a barrier instruction in the path of the (*v*) cell, dispatch of a new warp containing (*a*) cell.
6) Return to the main program path and the start of the new operation of embarrassingly parallel instructions ($W_{(x)}$ *operation in the returning stage,* $R_{(x)}$ *in the parallel execution)*.

This sequence of steps allows the test pattern generation and evaluation of the groups, CFtr and CFds, of coupling faults. Nevertheless, other steps must be added to evaluate missing coupling fault groups, such as CFir. In this case, the new steps are located after the 5^{th} step, and these include the execution of additional functions in the routine, thus generating implicit $R_{(x)}$operations in the LE. New barrier instructions are also placed to generate elaborated stimulus in both cells. The generation of test patterns for CFwd coupling faults needs a new $W_{(x)}$operation in the (*a*) cell employing a new routine, which is included after the 6^{th} step.

Example of March operations	Adapted approach of March operations
$\updownarrow (W_{(0)})$	$\Downarrow (W_{(0)}, R_{(0)})^*$, $\Uparrow (W_{(1)})^*$, $\Uparrow (R_{(1)})^*$, $\updownarrow (W_{(0)})$
$\Uparrow (R_{(0)}, W_{(1)})$	$\Downarrow (W_{(0)}, R_{(0)})^*$, $\Uparrow (R_{(0)})^*$, $\Uparrow (R_{(0)}, W_{(1)})$
$\Downarrow (W_{(0)})$	$\Downarrow (W_{(1)}, R_{(1)})^*$, $\Uparrow (R_{(1)})^*$, $\Downarrow (W_{(0)})$
$\Uparrow (R_{(0)})$	$\Downarrow (W_{(0)}, R_{(0)})^*$, $\Uparrow (R_{(0)})$
$\Downarrow (W_{(1)})$	$\Downarrow (W_{(1)}, R_{(1)})^*$, $\Uparrow (R_{(1)})^*$, $\Downarrow (W_{(1)})$
$\Uparrow (R_{(1)})$	$\Downarrow (W_{(1)}, R_{(1)})^*$, $\Uparrow (R_{(1)})$

FIGURE 6. A normal sequence of March operations and the adapted approach for the memory in the scheduler. (∗) Example of initialization operations.

D. MARCH ALGORITHMS ADAPTATION

March algorithms are well-known methods to detect coupling faults in memories. Thus, those methods can be employed to generalize the proposed technique to detect permanent and static coupling faults in the SC memory.

Those algorithms are composed of a set of writing and reading operations *(or March operations)* to be performed in the SC memory. The injection direction in operation is considered as an additional parameter to consider in the implementation phase.

It is not possible to inject consecutive March operations in the SC memory by its operational restrictions. Nevertheless, this restriction can be partially avoided by applying March operation patterns in a segmented fashion. In this method, a March operation is firstly performed, followed by a set of initialization operations. Then, the second March operation is injected and, finally, new initialization operations are included for other March operations. Fig. 6 shows an adaptation example employing the segmented approach.

The application of segmented patterns into the target memory locations does not reduce the effectiveness of the proposed method and the adaptation of a March operation.

This initiation sequence is composed of additional $W_{(x)}$ and $R_{(x)}$ operations required to access the target memory locations or to avoid the normal scheduler execution. In some cases, these initialization operations include March operations in other LEs. Thus, March operations are described independently as a set of program kernels.

The division of a test kernel in chunks, or multiple test programs, allows the fault evaluation during idle intervals of the in-field operation of the device. The division must also consider the observability mechanism and the Host interaction. For this purpose, the test signatures are stores in free locations of the global memory. Thus, these can be reused in multiple test kernels. Similarly, the Host may locate and trigger the test programs aside from the application kernels.

As a proof of concept, we employed two March algorithms (MATS+ and MATS++) to demonstrate the features of the proposed technique and generated a set of test programs for the TAM and WPC fields for each LE in the SC memory.

1) MATS+ AND MATS++ ALGORITHMS

Each algorithm is adapted following the proposed technique. Moreover, it is described in multiple test kernels; thus, its operation is performed by applying multiple test parts. Table 5 shows the operations in the MATS+ algorithm *(the reader may refer to* [11] *for details regarding March test notation).* It can be noted that initialization steps (in bond) and implicit $R_{(x)}$operations are included in the adaptation of each operation. The initialization steps are needed to generate a specific test pattern in the target LE (or warp) and to avoid the execution of the dispatcher units. This method is used for each LE in the memory.

TABLE 5. MATS+ and (∗) MATS++ operations as a sequence of CUDA kernel executions.

Original March operations (*MATS+*)	Adapted March Operations	Equivalent CUDA BBK kernel
M1: $\updownarrow (W_{(0)})$	**Init. Steps:** $(W_{(X)}, R_{(X)})$; **M1**:$\updownarrow (W_{(0)}, R_{(0)})$	**Test_kernel_dropping** <TOTAL_BLOCKS, TOTAL_THREADS> (TOTAL_THREADS, vector_params[0], d_signature);
M2: $\Uparrow (R_{(0)}, W_{(1)})$	**Init. Steps:** $(W_{(X)}, R_{(X)})$; **M2**:$\Uparrow (W_{(0)}, R_{(0)}, W_{(1)}, R_{(1)})$	**Test_kernel_dropping_x** <TOTAL_BLOCKS, TOTAL_THREADS> (TOTAL_THREADS, vector_params[1], d_signature);
M3: $\Downarrow (R_{(1)}, W_{(0)}, R_{(0)}*)$	**Init. Steps:** $(W_{(X)}, R_{(X)})$; **M3**:$\Downarrow (R_{(1)}, W_{(0)}, R_{(0)})$	**Test_kernel_incrementing** <TOTAL_BLOCKS, TOTAL_THREADS> (TOTAL_THREADS, vector_params[0], d_signature);

Table 5 also presents the adaptation of the algorithm into a set of independent test kernels. In this example, it can be observed that the external pattern is different in each case. The same pattern is applied to the first and third kernels. On the other hand, the second program needs the opposed value to generate the desired pattern. The kernel sequence (dropping, dropping_x, incrementing) must use the patterns in Table 4 to evaluate all target coupling faults. In the end, this program kernel sequence is applied eight times.

2) ALGORITHM IMPLEMENTATION IN THE CUDA ENVIRONMENT

In Fig. 7, the scheme presents the interaction between (*a*) and (*v*) using the method introduced in section IV.A. Although the diagram depicts a parallel program execution in (*a*) and (*v*), the execution of the algorithm is fully sequential for each SM. In fact, on each SM core, only one SC and its internal memory exist. However, this scheme shows the complexity in the test program description to ensure that (*a*) and(*v*) cells interact in the expected fashion.

The method employed to inject patterns in the cells uses a selection mechanism to identify and classify the LEs (warps) as even or odd (as (a)-(v), or (v)-(a)). Moreover, additional conditions in the kernel description were added to check the edge conditions of (a) in the memory borders

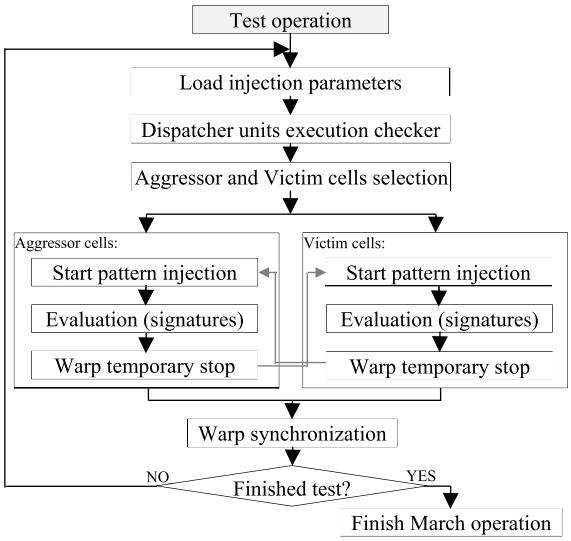

FIGURE 7. A general flow diagram of the test algorithm for the TAM and WPC fields in the scheduler memory.

(i.e., LE number 0 or 31 and target (v) number -1 or 32, respectively). In these cases, the (a) cell is maintained as inactive while the other cells perform the injection sequence. The mechanism employed for thread communication is based on the method to skip the dispatcher units operation, see section VI.A.

The implementation of the program kernels also considered the direction of pattern injection. Independent kernels were designed, as presented in Table 5, to perform the pattern injection in both directions.

The pseudo-code reported in Fig 8 describes a general CUDA implementation of one of the test programs aimed to detect coupling faults. This kernel is executed concurrently by an (a) and a (v) warps (or LEs).

The warp selection mechanism divides both cells and also controls the execution of the program sequentially.

In Fig. 8, we detailed the implementation of the warp selection and detection process. The main difference among them is the total number of conditions required to generate the start of detection of the target LE.

It should be noted that, in both cases, the returning steps after and before warp synchronization are neglected to inject any test pattern into the cells.

In the TAM case, these intervals are generated during the Not-Taken path evaluation. In the WPC algorithm, these intervals belong to the returning procedures from a called routine.

The warp selector mechanism generates independent paths for the (a) and(v) cells. Internally, the path may use divergence operations, stimulating the parameter under evaluation. In (a), a second divergence path is employed to add the necessary initialization condition ($W_{(0)}$) for the assessment of those coupling faults which need this initial state.

The pseudo-code presents the worst-case scenario for test pattern injection. Thus the kernel description can be simplified to avoid the second divergence for some static coupling and permanent faults.

TABLE 6. Example of a TP sequence for coupling fault evaluation.

Target TP operations	$TP_x = \{w_x^a, r_x^a, w_y^v, r_y^v, r_y^v, r_y^v\}$	
March Operation	**The equivalent operation for TAM fields**	**The equivalent operation for WPC fields**
w_x^a	Divergence generation in (a)	Subroutine call in (a) cell
r_x^a	Execution of multiple instructions in the taken path of (a)	Execution of consecutive instructions in (a) cell
w_y^v	Divergence generation in (v)	Subroutine call in (v) cell
r_y^v	Execution of multiple instructions in the taken path of (v)	Execution of consecutive instructions (v) cell
r_y^v	Execution of multiple instructions in the taken path of (v)	Execution of consecutive instructions in (v) cell

3) ADAPTING MARCH OPERATIONS TO TEST THE SCHEDULER MEMORY

Each March operation represents a set of W(x) and R(x) operations in a selected direction. The implementation of a March operation, considering the FPs for the SC memory, included the direction parameter as an additional factor in the adaptation.

a: ADAPTING THE MARCH WRITING AND READING PROCEDURES

Each operation ($W_{(x)}$ and $R_{(x)}$) is translated into equivalent software functions on the target fields (TAM and WPC). Table 6 presents an example of the adaptation of a TP for coupling fault detection composed of multiple $W_{(x)}$ and $R_{(x)}$ operations.

It is worth noting that this example does not present the additional steps after the pattern injection and the Not-taken path operations for (a) and (v) warps. Thus, those steps are considered as initialization operations for the next pattern.

The final step to adopt a March operation is the adaptation of the direction parameter in the kernel. The next subsection presents the method to add and select this parameter in the design.

b: ADAPTING THE MARCH DIRECTION OPERATIONS

The application of a March operation has an associated injection direction ($\updownarrow$, $\Uparrow$, $\Downarrow$). This parameter is related to the FP to test and TP to inject.

The CUDA implementation of the TPs included, on each kernel, an external parameter describing the injection direction (Incrementing and Dropping) of each operation. Moreover, this organization is static. Thus, the warps with lower or higher IDs will be the first selected and have priority access to the SM resources. It is worth noting that warp selection does not generate any divergence in the SM. Nevertheless, the warp

__global__ void Test_kernel_dropping_x (**int*** *divergence_parameters*, **int*** *signature* …)		
{		
Parameter_initialization();		▶Definition and initialization of variables (local and shared).
Thread_warp_size_check_correction();	(§)	▶Warp size checker.
for warp **in** kernel **do:**	(§)	▶Search each Warp ID
if *Warp_Selected*() **then:**	(§)	▶Select a Warp ID in order (*Incrementing* / *Dropping*)
Load_divergence_parameters();	(§)	▶Load the external pattern to be used in *(a)* cells.
if warp **is** *Aggressor* **then:**	(§)	▶Check if warp ID is *(a)* cells.
Aggressor_warp_enabled();	(£)	▶Check if *(a)* cell has associated *(v)* cells.
if divergence_parameter **is** '0' **then:**	(¥)	▶Execution of the Not-taken path (First divergence)
Signature_evaluation_updating();		▶Signature update and $R_{(x)}$ procedure.
Barrier_operation();		▶Warp Stop.
else:	(¥)	▶Execution of Taken path (First divergence)
Signature_evaluation_updating ();		▶ Signature evaluation and $R_{(x)}$ operation.
for Warp_Id > 0 **do:**		▶Check if *(v)* cell has been executed
if *divergence_parameter*(0) **is** '1' **then:**	(¥)	▶Execution of the Nesting functions (Second divergence)
Signature_evaluation_updating ();*Barrier_operation*();		▶ Signature evaluation and $R_{(x)}$ operation.
else if *divergence_parameter*(31) **is** '1' **then:**	(¥)	▶Execution of the Nesting functions (Second divergence)
Signature_evaluation_updating ();*Barrier_operation*();		▶ Signature evaluation and $R_{(x)}$ operation.
else:	(¥)	▶Execution of the Nesting functions (Second divergence)
Signature_evaluation_updating ();*Barrier_operation*();		▶ Signature evaluation and $R_{(x)}$ operation.
Signature_evaluation_updating ();		▶Implicit Read in one instruction cycle.
Warp_ID --;		▶Drop in Warp ID value.
else if warp **is** *Victim* **then:**	(§)	▶Check if warp ID is *(v)*.
victim_warp_enabled();	(£)	▶Check if *(v)* has associated *(a)* cell.
if threads **in** warp ('<16' / *'>15'*) **then:**	(¥)	▶Thread division into lower or **higher** part
Signature_evaluation_updating (); *Barrier_operation*();		▶Signature updating and $R_{(x)}$ procedure.
else:	(¥)	
Signature_evaluation_updating ();		▶Signature updating and $R_{(x)}$ procedure.
for Warp_Id > 0 **do:**		▶Check if *(a)* cell has been executed
Barrier_operation();		▶Warp Stop.
Barrier_operation();		▶Warp Stop.
else:	(§)	▶Warp is not selected to be launched.
Signature_evaluation_updating (); *Barrier_operation*();		▶Signature updating, Implicit $R_{(x)}$ procedure, Warp stop.
Warp_synchronization();	(§)	▶Warp synchronization.
Clear_Parameters();	(§)	▶Cleaning variables.
}		

FIGURE 8. **A pseudo-code (CUDA) describing the program kernel implementation to detect coupling faults. (¥) Functions to generate divergence paths in the *(a)* and *(v)* cells in the LEs. (§) functions to skip dispatchers. (*$*) optional functions to evaluate edge conditions in LEs.**

selection process and direction management are based on the previously described method to skip the dispatcher unit introduced in section VI.A.

One selection loop is added into the test programs to manage the warp selection, considering the direction of the March operation. This selection loop is initialized with a base, an offset, and a limit. The base and the limit define the direction to be applied. The test program starts with the same conditions for each warp in the test kernel; thus, during the kernel execution, the loop selects the warps in the range of the base and the offset values. Then, the base and offset are updated until the limit is reached. This selection mechanism has a higher priority for the warps in the range of selection. However, this method has low priority in the border warps and increases the latency in their execution. Moreover, the dispatching process may add additional conditions to select even or odd warps and generate TPs for multiple coupling faults.

Inside the loop, a base and offset are employed to select the external parameters and to identify the warps that can be dispatched into the SM.

Multiple shared memory locations are employed to control and manage the coherency of pattern injection between consecutive (*a*) and (*v*) warps. This local synchronization mechanism is local for the interaction between consecutive cells belonging to different LEs in the memory.

VIII. EXPERIMENTAL RESULTS

A. PERFORMANCE RESULTS

Some experiments were performed to validate the proposed methods. We employed an NVIDIA©GeForce GTX 960M GPGPU with 32 threads per warp and 1.176 GHz of clock rate. The evaluation and validation of the test programs are performed by employing the NVIDIA©Nsight™5.6 and the NVIDIA©Visual profiler tools. These tools are used to determine resources overhead and performance parameters. Moreover, these also verify the correct operation of the implemented test kernels in the GPGPU.

Table 7 reports the performance results of the implemented test programs following the proposed technique for different LE sizes in the SC memory. The second column in the table

TABLE 7. Performance parameters of the implemented test programs to evaluate different LE sizes. (+) active kernel functions, only.

		LE parameter					
		TAM			WPC		
LE size		32	16	8	32	16	8
BBK execution *(uS)*		778.98	359.30	168.30	573.50	187.79	91.52
Total execution *(mS)*		18.69	8.62	4.04	13.86	4.51	2.19
Number of instructions executed per pattern	BBK incrementing kernel *(KB)*	276.25	78.28	8.43	186.48	50.36	14.46
	BBK dropping kernel *(KB)*	276.61	78.18	8.40	436.13	73.49	15.58
	Total *(MB)*	829.11	234.74	25.26	809.08	174.21	44.49
		6.63	1.88	0.202	6.473	1.397	0.356
GPGPU memory overhead	System memory *(KB)*	1.84	1.84	1.84	2.42(+)	2.42(+)	2.42(+)
	Shared memory *(B)*	260.0	132.0	68.0	4.0	4.0	4.0

also reports the required idle time intervals to perform an individual test sequence of the whole procedure.

The BBKs were originally designed and implemented to be executed in one SM of the GPGPU with a configuration of one block per grid.

The implementation of the proposed method used an equal number of local registers, and this number is independent of the line-entry size. The TAM kernels required 28 registers, and the WPC kernels required 37.

The number of instructions to evaluate a parameter (3 test kernels) in the LEs of the memory is relatively high (809 and 809KB for WPC and TAM parameters, respectively). This behavior is explained by the selection of the programming environment (CUDA-C) to implement the functions and the additional mechanism employed to manage and avoid the dispatcher units. Moreover, the total number of instructions is also proportional to the number of line entries tested.

As shown in Table 7, some milliseconds are required to execute all program kernels. Moreover, the kernels present a low cost in terms of the system memory overhead to perform the test of one pattern for 32 LEs (2.4KB in the WPC parameter and 1.84KB in the TAM field). Nevertheless, the test program for the WPC parameter is also composed of some inactive kernels in memory to place the routines in the target memory locations to be tested. This test procedure requires the entire system memory space. Hence, the evaluation of the WPC parameter during device operation should be limited to the Power-On/Power-Off intervals.

Regarding the share memory resources (Table 7), the TAM kernels require a low number of memory locations. For this kernel, this parameter is directly dependent on the total number of line entries considered and the number of shared thread variables employed as semaphores used to avoid the execution of the dispatcher units. On the other hand, the WPC kernels employ a constant number of shared memory locations, as this kernel does not require semaphore variables among the threads.

Six test kernel functions are needed to control the warp execution and to stop the operation of the dispatchers. For this purpose, the conditional functions (divergence functions) are employed during the evaluation of coupling faults between cells. These functions include additional instructions in the test program implementation. In contrast, the instruction size of the BBK is relatively small (≈280KB) and is executed in less than 780μs. In the TAM test programs, the number of shared memory locations has a linear dependency on the evaluated LEs. In contrast, the WPC test programs employ a constant amount of shared variables independently of the SC memory size. This constant amount can be explained considering that the techniques for testing the WPC parameter are more straightforward than those employed to evaluate the TAM field, including the warp selection mechanism to stop the operation of the dispatchers.

A detailed analysis of the TAM test programs with the NVIDIA Visual Profiler shows that the concurrency operation of these test kernels, under multiple LE size, is 0%. The concurrency operation is an indicator of the level of parallelism in the program. This zero percentage in concurrency is mainly generated in the program by the complexity to follow the required steps and create the patterns to evaluate the TAM parameter in a LE and the TAM. Additional analyses with the Visual Profiler showed that the implementations of the test programs spend almost 60% of the execution time in thread synchronization operations or halting conditions. In the WPC test kernels, the halting and the synchronization functions affect in a lower manner the execution time, which is near to 50%. In both cases, these operations are needed to stop or avoid the execution of the dispatcher modules during the test operation.

B. FAULT DETECTION RESULTS

We employed, as a verification tool, a memory fault simulator to evaluate the effectiveness of the proposed technique and to check the FC.

This simulator reads an input file with the generated memory procedures (writing and reading) during the program kernel execution. A detailed description and additional information about the memory fault simulator can be found in [18].

To the best of our knowledge, comparative functional test techniques targeting the same memory structure in the scheduler of a GPGPU were not proposed in the past, thus limiting the possibilities of a direct comparison. Nevertheless, we selected three representative benchmarks to evaluate, compare, and show the efficacy of the proposed functional test mechanism. Each benchmark was configured with a

TABLE 8. Fault coverage of the MATS++ test kernels for an SC memory with 32 LEs. (**) FPs that were not initially considered in the proposed technique.

Fault primitive	MATS+ Algorithm FC (%)		VectorAdd FC (%)		MxM FC (%)		Edge FC (%)	
	TAM	WPC	TAM	WPC	TAM	WPC	TAM	WPC
TF_1	100	100	100	34.4	100	37.5	1.6	35.5
TF_0			0.0	84.4	0.0	87.5	100	89.5
RDF_1			100	84.4	0.0	38.48	100	40.5
RDF_0			0.0	34.4	100	89.1	1.6	90.5
DRDF_1			100	0.0	0.0	0.0	0.0	0.0
DRDF_0			0.0	0.0	100	0.0	0.0	0.0
WDF_1			0.0	34.4	0.0	37.5	100	35.5
WDF_0			0.0	84.4	0.0	87.5	1.6	85.5
CFds_X, CFir_X, CFtr_X	100	100	-	-	-	-	-	-
CFrd_X, CFwd_X, CFdrd_X	100	100	-	-	-	-	-	-
CFst_X	100 (*)	100 (*)	-	-	-	-	-	-
SF_X	100 (*)	100 (*)	-	-	-	-	-	-
CFid_X	100 (*)	100 (*)	-	-	-	-	-	-

workload equivalent to the one adopted for the functional test. These applications are:

1) Vector addition (VectorAdd): is a parallel program kernel, and it is selected considering that most applications include sections of fully parallel execution.
2) Matrix multiplication (MxM): is commonly used during the implementation of algorithms for image processing and neural network applications.
3) Edge detection (Edge): is a typical application in the image pre-processing and the computer vision fields.

Each test program and benchmark include a debugging function able to trace the memory operations in the SC. This debugging function generates a compatible input file for the memory simulator. Table 8 presents the obtained results for the proposed functional test mechanisms and the selected benchmarks expressed in terms of FC.

According to the results, the proposed technique can test the static single and coupling faults effectively in the SC memory. Although the original test programs design was not targeted to consider some groups of faults, such as the state coupling faults (CFst), the state faults (SF), and inversion coupling faults (CFid), the results in the fault simulation show that the test program kernels indirectly evaluated these groups of faults. The additional nesting divergence functions, for both cells (a and v), directly caused the detection of faults in the CFid group. Moreover, the time delays during the test injection provoke the required initial conditions in those cells that are not directly evaluated, thus, indirectly evaluating the faults of the CFst and SF groups.

During the first attempts to determine FC results from the fault simulations using the proposed method, we obtained some moderate percentages (96%) for some specific patterns in the memory. After a detailed check of these conditions, we concluded that the initial stage of some LEs, targeting the TAM fields, remains in an active state, thus avoiding the evaluation of some FPs with a $W_{(0)}$ as a starting logic state. These restrictions were finally removed through the addition of the second divergence path to consider the conditions of initialization.

A comparison of results among the implemented test programs and the selected benchmarks allow us to affirm that an application by itself is not suitable to generate a considerable number of functional test patterns in the SC memory. The previous behavior is clearly represented in the low percentage of FC of each single cell FP group. The VectorAdd and MxM applications operated fully parallel and did not produce all the required logic state changes for the TAM field. The parallel behavior explains the zero percentage presented in the FC in some FPs groups. A similar situation is shown in the low percentage of the TAM field in the Edge application. This behavior is generated by some divergence paths produced during the evaluation of the convolution algorithm. Nevertheless, the generated divergence is not enough to increase the percentage of FC in most single-cell groups of FPs.

Concerning FC results in the WPC field, these directly depend on the total number of instructions and the number of internal loops described in the program kernel. The longest benchmark (Edge), which also includes up to 25 loops, denotes a higher FC in comparison with the other applications with a lower number of instructions and with none (VectorAdd) or a few (MxM) loops in their description. A direct comparison of FC results between the proposed functional test mechanism and the representative applications shows the need and relevance of the proposed functional test solution in terms of detecting faults in a large number of groups of FPs.

The FC results in the benchmark for the coupling fault groups are not presented in Table 8. However, it is possible to state that the same trend towards a low FC for single-cell FPs is also valid for the multiple cell groups.

C. KERNEL PERFORMANCE OPTIMIZATION

The implemented kernels apply the March operations sequentially following the direction of a March operation. However, the software management of the kernels (by using semaphore variables, loops, and synchronization functions) is effective in generating test patterns in the memory, but it also reduces the execution performance. Thus, the total kernel execution time requires hundreds of microseconds in the experiments. Nevertheless, as detailed previously, it was noted that the operation of the synchronization loops causes a large part of the latency in execution time.

We propose a set of optimizations in the kernel description by modifying the method employed to apply the March operation and reduce the time execution in the kernels. This method arbitrarily uses the operation on groups of two independent consecutive LEs *(Warps),* one as *(a)* and the other as *(v),* following the direction parameter of the original BBK. Thus, the injection of a $W_{(x)}$ or $R_{(x)}$ operation only depends on the

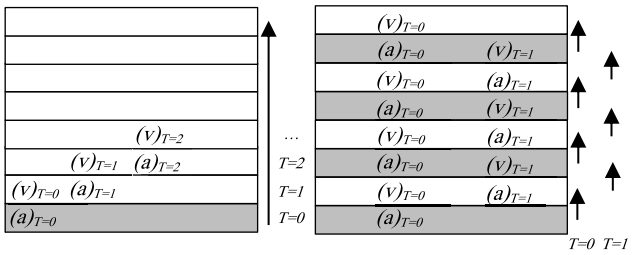

FIGURE 9. Proposed methods to apply a March operation, including the direction parameter. (Left) Sequential method using software techniques to skip the dispatcher unit operation. (Right) Using an arbitrary parallel injection in pairs of LEs.

correlated pairs of LEs (see Fig. 9). This method takes advantage of the operation of the dispatcher units to manage the pattern injection on consecutive LEs. Thus, a high percentage of the latency presented in the previous sequential method can be removed. Nevertheless, it must be considered that synchronization loops cannot be entirely excluded. However, those can be optimized to reduce the final latency.

The first loop, employed in the sequential selection of a warp, was removed and replaced by the simple loop mechanism, which also selects LE as (*a*) or (*v*). Then, on each path, this solution includes optional synchronization loops for each condition. One additional loop and break-loop condition are placed at the end of the kernel execution after the pattern is applied, and the LEs are ready to finish.

The main purpose of the loop and the break-loop conditions is the reduction of inefficient synchronization loops and the associated latency for a large number of warps. In this strategy, the synchronization is only evaluated between the aggressor and victim warps, instead of adopting a global synchronization with all warps, as in the other strategy.

The break-loop condition was designed to be executed at the end when a warp terminates its execution, and intra-warp divergence is not active, thus simplifying the management of non-consecutive warps.

The same semaphore mechanisms are employed to keep the order in the execution. Nevertheless, the total amount of semaphores is incremented up to 32, which is equal to the number of LEs in the scheduler memory.

This optimized solution should be applied twice for the *(a)-(v)* and *(v)-(a)* cells configurations, respectively, to cover all conditions in the memory.

A performance comparison between a sequential version of a test kernel and the optimized version was performed. Table 9 presents the performance parameters for two BBKs (*increasing* and *dropping*).

The effectiveness of the proposed optimizations in the kernel description can be noted by comparing the execution time of the BBKs (original and optimized). For the Dropping BBK version, the execution time is reduced by more than 77%. Similarly, the Increasing BBK version presents a reduction of more than 79%. The total number of registers employed by each kernel remains the same. In contrast, each optimized kernel version uses up to 64 times more shared memory

TABLE 9. Performance parameter for the original and optimized version of a test kernel. (∗) The resource overhead is calculated per kernel.

BBK	Time execution (us)	Registers	Shared memory (bytes)	Instruction size (bytes)
Dropping (*Original*)	208,701	37	4	1,152
Dropping (*Optimized*)	47,868	37(*)	256(*)	2,944
Improved	**160,833**	**0**	**-252**	**-1,792**
Increasing (*Original*)	176,750	37	4	1,728
Increasing (*Optimized*)	35,801	37(*)	256(*)	2,944
Improved	**140,949**	**0**	**-252**	**-1,216**

locations and almost double the number of instructions in the kernel description. Although this optimization required more resources in terms of shared and system memory locations, these can be partially neglected when comparing with the high percentage of execution performance gain.

In Table 9, it should be considered that the selected BBKs do not include a second nesting divergence for pattern injection. This additional nesting would require one extra synchronization loop. Thus each path would include one such loop, as it is presented in the original implementation. Nevertheless, the improvement process is the same as described below.

It is well-known for parallel programs that the usage of synchronization loops reduces the execution performance and could generate conflicts in the intra-warp execution; thus, in general terms, these methods should be avoided. Nevertheless, the added mechanisms were carefully designed to operate in specific regions when the warp does not diverge. Moreover, the loop break condition is limited to convergence warp states: in this way, the coherence of the kernels is not affected.

IX. CONCLUSION

A functional technique was proposed targeting the development of Self-test programs aiming to perform the on-line test of static and coupling faults in the scheduler memory of a GPGPU. This technique was developed and implemented, considering the available micro-architectural information of a GPGPU and a high-level programming environment (CUDA). The results on some representative test cases showed that the proposed approach is effective and can test all the fault primitives, thus validating and ensuring complete coverage of all static and coupling faults in the targeted structure.

Optimizations to effectively implement the same kernel to describe a generic March operation targeting the memory within the scheduler are also presented.

REFERENCES

[1] M. Psarakis, D. Gizopoulos, E. Sanchez, and M. S. Reorda, "Microprocessor software-based self-testing," *IEEE Des. Test. Comput.*, vol. 27, no. 3, pp. 4–19, May 2010.

[2] "Guidelines for obtaining IEC 60335 Class B certification for any STM32 application," STMicroelectron., Geneva, Switzerland, Appl. Note AN3307, 2016.
[3] (2018). *Infineon Technologies*. [Online]. Available: https://www.hitex.com/software-components/selftest-libraries-safety-libs/pro-sil-safetcore-safetlib/
[4] *DS52076A 16-bit CPU Self-Test Library User's Guide*, Microchip Technol., Chandler, AZ, USA, 2012, p. 52.
[5] (2018). *ARM Technologies*. [Online]. Available: https://developer.arm.com/technologies /functional-safety
[6] (2018). *Renesas Technology*. [Online]. Available: https://www.renesas.com/en-eu/products/synergy/software/add-ons.html#read
[7] "NVIDIA's next generation CUDATM compute architecture: FERMI," NVIDIA Corp., Santa Clara, CA, USA, White Paper V1.1, 2009.
[8] J. Nickolls and W. J. Dally, "The GPU computing era," *IEEE Micro*, vol. 30, no. 2, pp. 56–69, Mar. 2010.
[9] "V100 GPU architecture. The world's most advanced data center GPU," NVIDIA Corp., Santa Clara, CA, USA, White Paper Version WP-08608-001_v1.1, 2017, p. 108.
[10] S. Di Carlo, P. Prinetto, and A. Savino, "Software-based self-test of set-associative cache memories," *IEEE Trans. Comput.*, vol. 60, no. 7, pp. 1030–1044, Jul. 2011.
[11] G. Theodorou, N. Kranitis, A. Paschalis, and D. Gizopoulos, "A software-based self-test methodology for on-line testing of processor caches," in *Proc. IEEE Int. Test Conf.*, Sep. 2011, pp. 1–10.
[12] A. Van De Goor, G. Gaydadjiev, and S. Hamdioui, "Memory testing with a RISC microcontroller," in *Proc. Design, Autom. Test Eur. Conf. Exhibit. (DATE)*, Mar. 2010, pp. 214–219.
[13] M. Grosso, W. J. H. Perez, D. Ravotto, E. Sanchez, M. S. Reorda, and J. V. Medina, "A software-based self-test methodology for system peripherals," in *Proc. 15th IEEE Eur. Test Symp.*, May 2010, pp. 195–200.
[14] A. Apostolakis, D. Gizopoulos, M. Psarakis, D. Ravotto, and M. S. Reorda, "Test program generation for communication peripherals in processor-based SoC devices," *IEEE Des. Test. Comput.*, vol. 26, no. 2, pp. 52–63, Mar. 2009.
[15] A. Paschalis and D. Gizopoulos, "Effective software-based self-test strategies for on-line periodic testing of embedded processors," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 24, no. 1, pp. 88–99, Jan. 2005.
[16] P. Bernardi, R. Cantoro, S. De Luca, E. Sanchez, and A. Sansonetti, "Development flow for on-line core self-test of automotive microcontrollers," *IEEE Trans. Comput.*, vol. 65, no. 3, pp. 744–754, Mar. 2016.
[17] M. Gaudesi, M. S. Reorda, and I. Pomeranz, "On test program compaction," in *Proc. 20th IEEE Eur. Test Symp. (ETS)*, May 2015, pp. 1–6.
[18] J. Hudec, "An efficient adaptive method of software-based self test generation for RISC processors," in *Proc. 4th Eastern Eur. Regional Conf. Eng. Comput. Based Syst.*, Aug. 2015, pp. 119–121.
[19] J. Zhou and H.-J. Wunderlich, "Software-based self-test of processors under power constraints," in *Proc. Design Autom. Test Eur. Conf.*, 2006, pp. 1–6.
[20] M. Gaudesi, I. Pomeranz, M. S. Reorda, and G. Squillero, "New techniques to reduce the execution time of functional test programs," *IEEE Trans. Comput.*, vol. 66, no. 7, pp. 1268–1273, Jul. 2017.
[21] A. Riefert, R. Cantoro, M. Sauer, M. Sonza Reorda, and B. Becker, "A flexible framework for the automatic generation of SBST programs," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 24, no. 10, pp. 3055–3066, Oct. 2016.
[22] R. Ubar, A. Tsertov, A. Jasnetski, and M. Brik, "Software-based self-test generation for microprocessors with high-level decision diagrams," in *Proc. 15th Latin Amer. Test Workshop (LATW)*, 2014, pp. 1–6.
[23] N. Bartzoudis, V. Tantsios, and K. McDonald-Maier, "Dynamic scheduling of test routines for efficient online self-testing of embedded microprocessors," in *Proc. 14th IEEE Int. On-Line Test. Symp.*, Jul. 2008, pp. 185–187.
[24] C.-H. Chen, C.-K. Wei, T.-H. Lu, and H.-W. Gao, "Software-based self-testing with multiple-level abstractions for soft processor cores," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 15, no. 5, pp. 505–517, May 2007.
[25] M. Hatzimihail, M. Psarakis, D. Gizopoulos, and A. Paschalis, "A methodology for detecting performance faults in microprocessors via performance monitoring hardware," in *Proc. IEEE Int. Test Conf.*, Oct. 2007, pp. 1–10.
[26] P. Bernardi, C. Bovi, R. Cantoro, S. De Luca, R. Meregalli, D. Piumatti, E. Sanchez, and A. Sansonetti, "Software-based self-test techniques of computational modules in dual issue embedded processors," in *Proc. 20th IEEE Eur. Test Symp. (ETS)*, May 2015, pp. 1–2.
[27] T. Koal and H. T. Vierhaus, "A software-based self-test and hardware reconfiguration solution for VLIW processors," in *Proc. 13th IEEE Symp. Design Diag. Electron. Circuits Syst.*, Apr. 2010, pp. 40–43.
[28] M. Scholzel, T. Koal, S. Muller, S. Scharoba, S. Roder, and H. T. Vierhaus, "A comprehensive software-based self-test and self-repair method for statically scheduled superscalar processors," in *Proc. 17th Latin-Amer. Test Symp. (LATS)*, Apr. 2016, pp. 33–38.
[29] A. Apostolakis, M. Psarakis, D. Gizopoulos, A. Paschalis, and I. Parulkar, "Exploiting thread-level parallelism in functional self-testing of CMT processors," in *Proc. 14th IEEE Eur. Test Symp.*, 2009, pp. 33–38.
[30] S. Di Carlo, G. Gambardella, M. Indaco, I. Martella, P. Prinetto, D. Rolfo, and P. Trotta, "A software-based self test of CUDA Fermi GPUs," in *Proc. 18th IEEE Eur. TEST Symp. (ETS)*, May 2013, pp. 1–6.
[31] S. Di Carlo, G. Gambardella, M. Indaco, I. Martella, P. Prinetto, D. Rolfo, and P. Trotta, "Increasing the robustness of CUDA Fermi GPU-based systems," in *Proc. IEEE 19th Int. On-Line Test. Symp. (IOLTS)*, Jul. 2013, pp. 234–235.
[32] S. Di Carlo, G. Gambardella, I. Martella, P. Prinetto, D. Rolfo, and P. Trotta, "Fault mitigation strategies for CUDA GPUs," in *Proc. IEEE Int. Test Conf. (ITC)*, Sep. 2013, pp. 1–8.
[33] N. Farazmand, R. Ubal, and D. Kaeli, "Statistical fault injection-based AVF analysis of a GPU architecture," *Proc. SELSE*, vol. 12, pp. 1–6, Jan. 2012.
[34] W. Nedel, F. L. Kastensmidt, and J. R. Azambuja, "Evaluating the effects of single event upsets in soft-core GPGPUs," in *Proc. 17th Latin-Amer. Test Symp. (LATS)*, Apr. 2016, pp. 93–98.
[35] M. Gonçalves, M. Saquetti, F. Kastensmidt, and J. R. Azambuja, "A low-level software-based fault tolerance approach to detect SEUs in GPUs' register files," *Microelectron. Rel.*, vols. 76–77, pp. 665–669, Sep. 2017.
[36] D. Defour and E. Petit, "A software scheduling solution to avoid corrupted units on GPUs," *J. Parallel Distrib. Comput.*, vols. 90–91, pp. 1–8, Apr. 2016.
[37] S. Di Carlo, G. Gambardella, I. Martella, P. Prinetto, D. Rolfo, and P. Trotta, "An improved fault mitigation strategy for CUDA Fermi GPUs," in *Proc. Dependable GPU Comput. workshop*, Dresden, Germany, Mar. 2014, pp. 1–6.
[38] B. Du, J. E. R. Condia, M. S. Reorda, and L. Sterpone, "About the functional test of the GPGPU scheduler," in *Proc. IEEE 24th Int. Symp. On-Line Test. And Robust System Design (IOLTS)*, Jul. 2018, pp. 85 90.
[39] M. Abdel-Majeed and W. Dweik, "Low overhead online periodic testing for GPGPUs," *Integration*, vol. 62, pp. 362–370, Jun. 2018.
[40] S. D. Carlo, J. E. R. Condia, and M. S. Reorda, "On the in-field test of the GPGPU scheduler memory," in *Proc. IEEE 22nd Int. Symp. Design Diagnostics Electron. Circuits Syst. (DDECS)*, Apr. 2019, pp. 1–6.
[41] A. Van De Goor, "Using march tests to test SRAMs," *IEEE Des. Test. Comput.*, vol. 10, no. 1, pp. 8–14, Mar. 1993.
[42] P. Rech, L. Pilla, P. Navaux, and L. Carro, "Impact of GPUs parallelism management on safety-critical and HPC applications reliability," in *Proc. 44th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, Jun. 2014, pp. 455–466.
[43] K. Andryc, M. Merchant, and R. Tessier, "FlexGrip: A soft GPGPU for FPGAs," in *Proc. Int. Conf. Field-Program. Technol. (FPT)*, Dec. 2013, pp. 230–237.
[44] B. Du, J. E. R. Condia, and M. S. Reorda, "An extended model to support detailed GPGPU reliability analysis," in *Proc. 14th Int. Conf. Design Technol. Integr. Syst. Nanosc. Era (DTIS)*, Apr. 2019, pp. 1–6.
[45] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert, "Statistical fault injection: Quantified error and confidence," in *Proc. Design, Autom. Test Europe Conf. Exhibit.*, Nice, France, Apr. 2009, pp. 502–506.
[46] S. Di Carlo and P. Prinetto, *Models in Hardware Testing: Lecture Notes of the Forum in Honor of Christian Landrault*. Dordrecht, The Netherlands: Springer, 2009.
[47] W. W. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic warp formation and scheduling for efficient GPU control flow," in *Proc. 40th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Dec. 2007, pp. 407–420.

STEFANO DI CARLO (Senior Member, IEEE) received the M.S. degree in computer engineering and the Ph.D. degree in information technologies from the Politecnico di Torino, Turin, Italy, in 1999 and 2003, respectively. He has been an Assistant Professor with the Department of Control and Computer Engineering, since 2008. His research interests include DFT techniques, SoC testing, BIST, and memory testing. He has published more than 150 articles in peer-reviewed the IEEE and ACM journals and conferences. He regularly serves on the Organizing and Program Committees of major the IEEE and ACM conferences. He is a Golden Core Member of the IEEE Computer Society.

JOSIE E. RODRIGUEZ CONDIA (Student Member, IEEE) received the B.S. and M.S. degrees in electronic engineering from the Universidad Pedagógica y Tecnológica de Colombia (UPTC), Sogamoso, Colombia, in 2013 and 2017, respectively. He is currently pursuing the Ph.D. degree with the Department of Control and Computer Engineering, Politecnico di Torino, Turin, Italy. He was an Adjunct Lecturer with the Electronics Department, UPTC, from 2013 to 2017. His research interests include system functional test, SBST, DfT, parallel architectures, GPGPUs, and embedded system design.

MATTEO SONZA REORDA (Fellow, IEEE) received the M.S. degree in electronics and the Ph.D. degree in computer engineering from the Politecnico di Torino, Italy, in 1986 and 1990, respectively. He is currently a Full Professor with the Department of Control and Computer Engineering, Politecnico di Torino. His research interest includes design and test of reliable electronic circuits and systems.