Testing the divergence stack memory on GPGPUs: A modular in-field test strategy

Josie E. Rodriguez Condia[†], M. Sonza Reorda[‡]
Dip. di Automatica e Informatica, Politecnico di Torino, Torino, Italy
{†josie.rodriguez, ‡matteo.sonzareorda}@polito.it

Abstract[1]**—General Purpose Graphic Processing Units (GPGPUs) are becoming a promising solution in safety-critical applications, e.g., in the automotive domain. In these applications, reliability and functional safety are relevant factors in the selection of devices to build the systems. Nowadays, many challenges are impacting the implementation of high-performance devices, such as GPGPUs. Moreover, there is the need for effective fault detection solutions to guarantee the correct in-field operation of a GPGPU, such as in the branch management unit, which is one of the most critical modules in this parallel architecture. Faults affecting this structure can heavily corrupt or even collapse the execution of an application on the GPGPU. In this work, we propose a non-invasive Software-Based Self-Test (SBST) solution to detect faults affecting the memory in the branch management unit of a GPGPU. We propose a scalar and modular mechanism to develop the test program as a combination of software functions. The FlexGripPlus model was employed to evaluate the proposed strategies experimentally. Results show that the proposed strategies are effective to test the target structure and detect up to 98% of permanent faults.**

Keywords—**General Purpose Graphics Processing Units (GPGPUs) Software-Based Self-Test (SBST), Divergence Stack Memory**

I. INTRODUCTION

Currently, General Purpose Graphic Processing Units (GPGPUs) represent effective solutions for data-intensive applications, such as multimedia, multi-signal analysis, and high-performance computing (HPC). Moreover, these devices are also increasingly considered for safety-critical applications with substantial requirements in terms of reliability and functional safety. In the automotive field, safety-critical applications (such as sensor-fusion systems and Advanced Driver Assistance Systems (ADAS)[1]) usually require real-time execution and high reliability. For this purpose, GPGPUs are implemented in cutting-edge technologies to maximize performance and reduce power consumption. Nevertheless, some studies [2, 3] proved that the latest transistor technologies are prone to be affected by faults during the operative life of the device. The most critical challenges arise when permanent faults affect a module (caused by wear-out or aging [4]), so altering the functionality and the reliability of the device.

A parallel processor, such as a GPGPU, is particularly efficient when executing embarrassingly parallel programs. Nevertheless, real applications are far from this behavior, and most of them are composed of non-easily-parallelizable algorithms. Thus, these applications usually contain Intra-Warp Divergence (IWD), which is produced when a group of threads (warp) has different execution paths with different instructions. In [5], the authors report that sample applications in the CUDA Software Development Kit (SDK), including IWD, use approximately 33% of the total execution time in processing these conditions. Similarly, in [6], the authors profile a divergence map of typical programs and workloads in GPGPUs. Results show that most applications might produce thousands or millions of divergence conditions during the operation of the applications.

The architecture of a GPGPU includes a particular module to manage the IWD. This specific module is often called *Divergence Management Unit* (DMU), Branch Divergence Controller, Branch Controller, or Divergence Controller. DMU is devoted to controlling the operation of multiple paths in the same group of threads. Internally, the DMU evaluates control-flow instructions and uses a stack memory to store relevant information concerning the execution paths. Most DMUs can manage divergences composed of two paths. However, other locations in the stack memory can be employed to manage more than two divergence paths. Thus, the DMU is crucial for the correct operation of an application in the GPGPU, and a fault affecting this unit can propagate through the modules and collapse the entire operation of the device and the executed application.

Currently, test engineers propose some structures and mechanisms targeting the in-field test of a GPGPU and face the new reliability and technology challenges. Current solutions are sometimes based on the addition of *Design for testability* (DfT) structures, such as *Memory Built-in Self-Test* (MBIST)[7], which can be activated during idle times of the operation and stimulate/observe the internal modules of a device detecting possible faults [8]. Other solutions are based on designing special software programs to test a target functionally. These non-invasive and flexible mechanisms are also called *Software-Based Self-Test* (SBST). The last approach is based on hybrid mechanisms, combining hardware structures and software (i.e., custom instructions) to detect [9] or mitigate [10] faults of internal modules in a device. Both solutions (DfT and hybrid) are costly when targeting small modules in a GPGPU and should be developed and included in a design before the production phase.

Some previous works demonstrated that SBST solutions [11] could be successfully integrated into safety-critical applications, such as the automotive ones [12]. Most previous works applied to GPGPUs proposed SBST solutions for some data-path modules [13], including the execution units [14, 15], the register files [16], the pipeline registers [17] and some embedded memories [18]. Moreover, solutions targeting other critical modules in the control-path have been proposed (the warp scheduler [19] and their internal memories [20, 21]). However, to the best of our knowledge, practical solutions to test the DMU using the SBST mechanism are still missing.

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

In the present work, we propose, for the first time, a functional test strategy based on an SBST approach targeting permanent faults in the DMU located inside a GPGPU. The proposed SBST strategy has been implemented and evaluated, resorting to the FlexGripPlus model, which is a simplified open-source version of the NVIDIA GPU architecture.

This work is organized as follows. Section II introduces a basic overview of the architecture of an NVIDIA GPGPU, as with the special emphasis on the one used to validate the proposed strategy. Section III describes the SBST strategy proposed to test a permanent fault. Section IV reports the implementation details. Section V reports the experimental results, and Section VI draws some conclusions and outlines future works.

II. The FlexGripPlus GPGPU model

FlexGripPlus is a GPGPU model fully described in VHDL [22]. It is an improved version of the original FlexGrip model developed by the University of Massachusetts FPGA [23]. This model implements the Nvidia G80 micro-architecture, and it is also compatible with the commercial programming environment (CUDA under the SM_1.0 compatibility). FlexGripPlus supports 28 instructions of either 32 or 64 bits in 64 formats. The number of parallel execution units is configurable among 8, 16, and 32.

The architecture of an NVIDIA G80 GPU (and of FlexGripPlus) is based on the SIMT (Single-Instruction Multiple-Thread) paradigm. It exploits a custom Streaming Multiprocessor (SM) core with five stages of pipeline (*Fetch*, *Decode*, *Read*, *Execution/Control-flow* and *Write-back*), as shown in Fig. 1. This special-purpose parallel processor executes the same instruction (warp instruction) into a group of multiple threads using the available Execution Units (EUs), or Streaming Processors (SPs), in the SM. One warp is defined as a group of 32 threads. Furthermore, one controller and a warp scheduler controller (WSC) control the submission and execution of the warps into the SPs. In the SIMT paradigm, one warp instruction is fetched, decoded, and distributed to be processed on an independent SP. The *Read* and *Write-back* stages load and store data operands from/to Register Files (RFs), shared, global, or constant memories.

The *Execution/Control-flow* stage includes one DMU, which controls and traces the IWD, which is manifested in a program when the threads of the same warp execute different instructions, so generating multiple execution paths. The DMU can handle two paths in the same level of divergence and up to *n* levels of nested divergence, where *n* represents the number of threads in a warp. The DMU is located in parallel to the *Execution/Control-flow* pipeline stage and can support inter-warp branching at the hardware level. This module manages the control-flow operations to start or retake the flow from conditional branches with multiple paths. This unit also supports up to *n-1* levels of nesting branching.

Inside the DMU, one special purpose memory stores the starting (divergence point) and ending points (convergence point) of the divergence paths of a warp. This information is stored as a stack and processed by the DMU when executing control-flow instructions. More in detail, the memory is organized as a set of 32 Line-Entries (LEs) using the format presented in Fig. 2. The number of LEs directly related to the threads in a warp and the maximum nesting divergence per warp. A divergence point is a location in a parallel program where two paths (*Taken* and *Not-Taken*) are generated to be executed by a warp. Similarly, a convergence point is a location in a program where the IWD paths finish, and the program retakes one execution path.

Each LE in the stack is composed of three fields. Those fields are the thread mask (TM), the flow ID, and the program counter

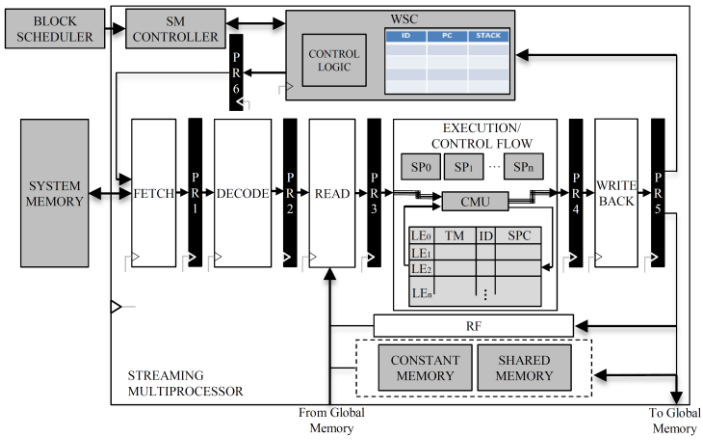

Fig. 1. A general scheme of the SM architecture in FlexGrip and the control-flow structure

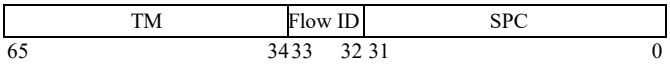

Fig. 2. Organization format of one LE in the stack memory of the GPGPU

of the actual warp under execution (SPC). The TM stores the status of the active threads in a warp. An active logic state represents the number of active threads executing a path (Taken or Not-Taken). The flow ID represents the actual state of the execution of an IWD. This value can be "01" (for a branch condition) or "00" (for a synchronization point or embarrassingly parallel condition). The SPC can store the starting address of the paths or the synchronization point address after paths execution.

The CMU employs two LEs to manage the operations when IWD is produced. The first LE stores the synchronization point (also known as convergence point) in the SPC field and the number of active threads at the moment of starting the divergence in the TM field. The second LE stores the starting address for the not-taken path in SPC and the threads to execute this path in TM. It is worth noting that when nesting divergence is produced, the CMU uses a new set of LEs to store the status information for the further divergence.

The execution of a synchronization instruction (*SSY*) affects the address pointer in the memory, and it is moved to the next LE. When the program reaches the convergence point, the pointer returns to the previously addressed LE. During the execution, the first LE is used only for storing purposes. In contrast, the second LE is employed during the management of the divergence, and control-flow instructions can affect this LE with writing or reading operations. In this way, when the operation of the first path ends, the information in the second LE is employed to start the not-taken path until the convergence point is reached.

Two main operations can be employed to manage the LEs in the stack memory. Initially, a new LE is addressed with *SSY*, during divergence or nesting divergence generation. The return from an addressed LE to the previous one can be performed using exit control-flow instructions, such as (*NOP.S*).

III. Proposed Method to test the Divergence Stack

The proposed approach employs the functionality of the CMU and the stack memory to generate self-test routines for each LE in the memory. These self-test routines are compacted into modular functions to propose scalable and modular test solutions. The modularity provides the required controllability and observability features to inject test patterns and propagate the fault effect, respectively.

A. Controllability

The injection of test patterns is performed by forcing the execution of divergence paths targeting each one of the threads in

a warp, so a sequence of divergence paths is generated to detect permanent faults in the TM bit field of each LE. As an effect of the divergence procedures, the TM field stores the threads following the not-taken path. Moreover, the SPC field stores the starting address of the not-taken path.

We propose two possible methods to control the address pointer of the LEs and inject test patterns in both fields (TM and SPC) of the LEs.

The first method (*Nesting*), see Fig. 3 (Top), can generate test patterns by using a sequence of nesting IWD routines, where the generation of each divergence produces as an effect, the movement of the address pointer to a deeper LE. The divergence is produced by comparing the *Thread.id* of each thread in a warp with a constant value. The main idea is to generate an ordered number of comparisons and follow a specific path, so causing the required test pattern in the TM field of each LE.

On each comparison, one or a group of threads is disabled, so defining a pattern to be stored into the deeper LE and generating two execution paths. This method is useful in managing the addressing of the LEs and injecting patterns into the TM field. Moreover, it can be described in the CUDA programming environment without significant modifications. The routines on each path (*Taken* and *Not-taken*) expose the presence of a permanent fault in the TM. The previous process is repeated for half the number of threads in a warp. Once, *Taken* routine finishes, the DMU submits the *Not-taken* path routine for processing purposes.

As it can be observed in the scheme in Fig. 3, the main idea is to always execute the routine in the *Taken* path, which generates new divergence paths and forces the test of other levels of LEs. The fault detection can be explained considering that once a divergence is generated, two LEs store the synchronization point and the address to start the not-taken path (as test patterns). Thus, a fault can be detected when retrieving the stored values, or when the number of threads executing a path is different from the expected one, so making the fault effects visible. The total number of possible nesting divergences is 16 when targeting any architecture with a warp composed of 32 threads. Nevertheless, this mechanism is scalable to more threads in a warp.

The *Nesting* strategy can inject test patterns on the even LEs of the stack memory. However, the odd ones are missing. The generation of test patterns for these fields requires the explicit addition of one synchronization instruction SSY before start the comparisons causing the divergence. The effect of the SSY instructions is the movement of the address pointer to the next or deeper LE in the stack memory. Then, the same previous procedure can be applied again, so testing the odd LEs. The comparison values are loaded using immediate instructions.

Nevertheless, the main issue of this strategy is the disabling state of the threads. When a thread is disabled, this cannot be turned active again until the divergence paths are executed, and a convergence point is reached. Thus, it is not possible to test or detect a permanent fault in a deeper LE location if a thread is disabled. This restriction implies that the comparisons should be performed multiple times, targeting different threads in the TM field. Thus, the strategy may suffer from considerable code length and excessive execution times.

The second method called *Sync-Trick*, see Fig. 3 (Center), exploits the functionality of *SSY* instruction to deceive the DMU when testing the stack memory. This method allocates *SSY* operations in strategically selected locations in the test program of the CMU to generate the change in the address pointer of the LEs.

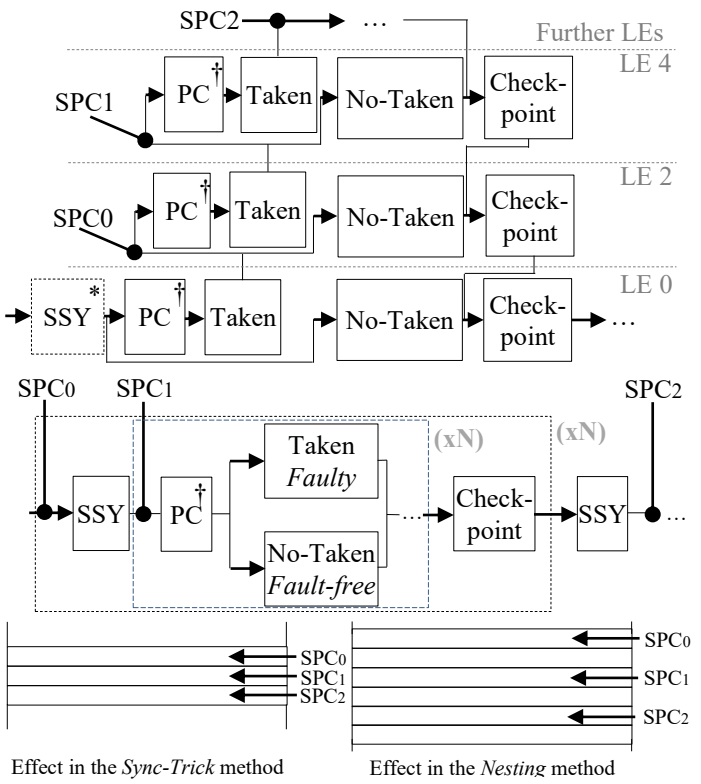

Fig. 3. General schemes of the program-flow of the proposed SBST strategies. *Nesting* method (top), *Sync-Trick* method (center), (bottom) effect in the address pointer of the stack memory of the CMU. (*)Optional function to test the odd LEs. (†) Optional functions to distribute the test functions in the system memory

More in detail, the *SSY* is explicitly located before each sequence of controlled divergence functions to test the TM of a LE. Then, this instruction forces the controller to allocate a new level of LE in the memory without the need to generate the IWD explicitly. The advantage of this mechanism is that each LE can be addressed without the need of disabling specific threads to create new addressing in the memory with the address pointer. Thus, this strategy replaces the generation of nesting divergence by the management of the address pointer in the memory. A sequence of simple IWD operations, generating the *Taken* and *Not-taken* paths, is employed to test the target LE. This process can be repeated N times to use different active threads and memory addresses (as test patterns). Then, a new SSY instruction is generated addressing a deeper LE and restarting the test procedure. It is worth noting that this mechanism is effective to move across one direction and reach deeper LEs in the memory. However, the returning phase (to a previous LE) requires the achievement of the convergence point address, which is initially stored by the *SSY* instruction. This strategy cannot be directly described in the high-level programming environment, and modifications at the assembly level are required.

B. Observability

The fault effect propagation is obtained by using the Signature per Thread (SpT) mechanism [17, 20]. This mechanism uses a set of signatures to map and to propagate the effect of a permanent fault in the stack memory into the global memory. Each SpT is updated, taking advantage of the paths in the controlled divergence routines. Thus, the same mechanism employed to perform the fault injection is used to increase its observability in the structure under test. Each SpT computes and accumulates intermediate results for each verified LE. The SpTs are finally grouped and stored in global memory for later analyses.

C. Modular test strategies

The observability and controllability methods for the TM field are complemented with some check-pointing routines, which are

devoted to testing the SPC field. These routines are located after the convergence point. In this way, any permanent fault in the SPC is detected if the synchronization point or the starting addresses of the *Not-Taken* path present any permanent fault. The check-point routines verify, through a check-point signature, the correct flow execution of a program. In the check-point procedure, this routine compares an expected check signature value in the program with the actual value accumulated during the execution of the test program. When the comparison matches, the accumulated signature is updated, otherwise the test program finishes propagating in memory the error in the SPC field of the evaluated LE. This strategy can be applied to any of the two controllability methods (*Nesting* or *Sync-Trick*).

The check signature values are predefined before execution and are loaded through immediate instructions. A fault in the SPC field would generate an unexpected addressing in the system memory. The permanent fault is detected by changes in the execution time or the signatures stored in the global memory.

The SPC field is partially tested. This issue is mainly caused by the short length of the test program in both strategies. A control-flow routine (PC), see Fig. 3, can be included before or in one of the paths of each IWD to test the high bits in the SPC field. Moreover, the test routines are redistributed across the system memory, so generating the missing test patterns.

Some GPU instructions (in the format of 32 and 64 bits) are located before each relocated function in the memory. These instructions avoid hanging conditions by permanent faults in the SPC field. In this way, when the program counter is affected by a fault, and it jumps to any unexpected memory location, it is always possible to retake control of the program and finish the execution of the GPGPU. Nevertheless, it is expected degradation in performance by the effect of the permanent fault.

Figure 3 (Top and Center) presents the basic schemes of the operational flow for the *Nesting* and *Sync-Trick* mechanisms. In both schemes, the execution of the synchronization instruction (*SSY*) provokes a change in the addressed LE by the pointer in the stack memory. The address pointers SPC_0, SPC_1, and SPC_2 represent the effect in the stack memory when executing the functions on each method. In the *Nesting* scheme, the divergence instructions and the implicit *SSY* instructions are represented in the division of the paths.

The use of these additional functions (Check-Point and PC) is entirely optional, considering that these strategies are costly in terms of memory overhead for an in-field execution. It is worth noting that the proposed technique takes into account the operational restrictions to develop the test programs using the Stuck-at fault model. Other fault models would require the adaptation of the *Sync-trick* mechanism. However, it would be hard or impossible to follow the *Nesting* strategy.

IV. Implementation

Following the schemes in Fig. 3, we implemented the SBST strategy using the high-level programming environment (when possible) and combined with instructions in the assembly language (SASS) of the GPGPU. Blocks and dotted lines in Fig. 3 represents the division in the description of the SBST code as a set of functions for both methods.

The implemented code for both test methods is composed of the following functions: *i)* Initialization function, *ii)* synchronization function (*SSY*), *iii)* flow control function (*PC*), *iv)* the test pattern injection and SpT update function (*Taken* and *No-Taken*), and *v)* the check-point function (*Check-point*).

Each function is described independently and can be attached depending on the target of a test program. The initialization function defines and initializes the registers for each thread. Moreover, this function initializes the addresses to store the SpTs and check-point signatures. The functionality of other functions was introduced in the previous section.

The modular description of both SBST strategies has the advantage of allowing the fast development of multiple test programs with different test objectives. In the *Nesting* method, the modularity is used to manage the nesting divergence functions and to add or remove the optional functions targeting the test of the SPC field. In contrast, the modularity presents considerable advantages for the *Sync-Trick* method. The code description of this method is scalable and modular in such a way that it is possible to add or remove part of the description to target the individual test of LEs in the stack memory.

This modularity gives us the possibility to address any or a group of LEs in the stack memory and to generate an independent test program. The division of the test contributes to reducing the execution time of the test program during the in-field operation of a GPGPU.

The *Sync-Trick* method can employ two approaches to evaluate LEs in memory. The first approach (*Accumulative or Acc*) aims the test of a consecutive group of LEs and accumulates the signatures in memory. This approach must always start from the first LE and can finish at any of the other 31 LEs in the stack.

On the other hand, the second approach (*Individual or Ind*) targets the testing of an individual LE and then the retrieving of signature results to the host. This approach only focuses on one of the LEs in the memory and is intended to have a reduced execution time. The performance cost (execution time) of both approaches (*Acc* and *Ind*) can be calculated using the equations (1) and (2).

$$ST(Acc) = T \cdot n + Ch \cdot n + SSY \cdot (n-1) \quad (1)$$

$$ST(Ind) = T + Ch + SSY \cdot (n-1) \quad (2)$$

Where *n* represents the target LE in the stack memory. SSY, T, and Ch represent the execution time of the synchronization, test pattern injection, and check-point functions, respectively. The initialization function was not included considering that it is constant for both cases, and it is negligible in terms of duration.

From equations (1) and (2), it is clear that the cost of the Accumulative version (Acc) is higher than the *Ind* version. The cost is mainly caused due to the different approaches in each case. In the *Acc* version, the program is intended to test the number of selected LEs sequentially. In contrast, the *Ind* approach targets the test on one LE, so the test pattern and check-point functions are used once. The number of synchronization functions depends on the target level of LE in the stack memory.

On the other hand, the performance cost of the *Nesting* method is described by the expression in equation (3).

$$Ns = N \cdot Ch \cdot \sum_{i=0 \vee 1}^{m} SSY + T \quad (3)$$

Where *N* represents the total number of threads in a warp, and *m* is the target LE to be tested. *CH*, *SSY,* and *T* have the same meaning from equations (1) and (2). As introduced previously, the target LE could be even or odd. Thus, the starting value of *i* in the summation could be 0 or 1.

Table 1 reports the results of the performance parameters for the *Nesting* method, and the *Sync-Trick* method under the *accumulative* and *individual* approaches. It is worth noting that results reported in Table 1 were obtained by simulations in the ModelSim environment using the FlexGripPlus model.

The reported results show the performance parameters for the two possible methods that can be used to test the LEs in the stack

memory of the CMU. All versions present an overhead in the global memory of 64 locations (256 bytes) devoted to saving the SpTs and the Check-point signatures.

TABLE 1. PERFORMANCE PARAMETERS OF THE SBST PROGRAMS USING THE TWO APPROACHES TO DETECT PERMANENT FAULTS IN THE LES

Approach	LE	Instructions	Execution time (Clock cycles)	System memory overhead (Bytes)
Sync-Trick Ind	1	403	33,449	1,612
	2	404	34,211	1,616
	10	412	34,589	1,648
Sync-Trick Acc	1 - 2	794	66,637	3,176
	1-10	3,922	326,423	15,688
	All	12,524	1,030,473	50,096
Nesting	1	683	37,986	2,732
	1-2	1,323	83,569	5,292
	1-10	6,443	528,086	25,772
	All	19,883	2,567,209	79,532

Regarding the performance results of both versions, it can be noted that the *Sync-Trick* (*Ind*) approach maintains an average performance cost to test any LE in the stack memory. The only difference among these programs is the number of SSY instructions included to address a selected LE. Similarly, the *Sync-Trick* (*Acc*) version can test a group of LEs consecutively. However, it requires additional execution time and cannot be stopped once the test program starts.

On the other hand, Table 1 reports the required execution time to test the first and the second LEs in the stack using the *Sync-Trick Ind* (rows 2 and 3, column 4) and *Sync-Trick Acc* (row 5, column 4) approaches. The *Individual* approach requires 76 additional clock cycles to test the LEs, but it has the advantage of test each LE independently. In contrast, the Accumulative method must check both LEs consecutively. Thus, the *Ind* approach can be adapted for in-field operation by the limited number of clock cycles required during the execution.

The performance parameters show that the *Nesting* approach has a proportional relation among the number of instructions and the number of LEs to test. Similarly, the relationship between the execution time and the number of LEs to test present an increasing exponential ratio. In the end, the *Nesting* method requires more than double the execution time to test the entire stack than the *Sync-Trick* using the *Acc* approach. The execution time could be the relevant parameters to take into account when targeting the in-field operation.

V. EXPERIMENTAL RESULTS

The RT-level description of the FlexGripPlus model was employed in the experiments. The fault injector environment follows the methodology described in [17], and we injected permanent faults following the Stuck-at-Fault model. A total of 4,224 permanent faults were injected in the stack memory of the CMU of the FlexGripPlus model for each fault campaign.

The fault simulation campaign was performed using both representative benchmarks and the proposed SBST strategy. These representative benchmarks employ the CMU unit and are carefully selected to compare the detection capabilities they can achieve with the one provided by the proposed SBST approach. Descriptions and details regarding the chosen benchmarks can be found in [17, 24]. For the sake of completeness and comparison, the different versions of the SBST strategy are reported in Table 2. Moreover, both approaches were evaluated with and without the optional SPC functions in the LEs.

The last column of Table 2 reports the testable FC of the benchmarks and the proposed SBST strategy. During the analysis of the memory in the stack, a total of 192 faults were identified as untestable. These are related to the lowest bits of the SPC field of each LE, which does not affect the execution of an instruction. Thus, these faults were removed when computing the FC.

TABLE 2. FC RESULTS FOR THE REPRESENTATIVE BENCHMARKS AND THE PROPOSED SBST STRATEGY

SBST strategy or benchmark		FC (%)				
		SDC	Hang	Timeout	Total	Total testable
MxM		0.0	0.38	0.0	0.38	0.40
Sort		0.15	0.04	0.0	0.19	0.19
FFT		0.14	0.19	0.0	0.33	0.35
Edge		0.15	0.28	0.0	0.43	0.47
Sync-Trick	***Ind***	65.64	2.08	1.01	68.75	72.02
	Acc	64.89	2.84	1.01	68.75	72.02
	Ind + PC	83.00	8.49	2.44	93.93	98.41
	Acc + PC	82.24	9.25	2.44	93.93	98.41
Nesting		54.12	11.81	1.23	67.16	70.04
	+ PC	76.94	13.16	2.81	92.91	97.34

The *Sync-Trick* strategy presents a moderate FC for both cases (*Ind* and *Acc*). Moreover, the FC increases when adding the SP functions and the relocation of the test functions in the memory. These comprehensive approaches (*Ind+SP* and *Acc+SP*) obtain a high percentage of FC for the target structure.

An in-depth analysis of the results shows that the Individual approach allows detecting 100% of the faults in the TM of all LEs by looking at the results produced by the test procedure (Silent Data Corruption, or SDCs). In contrast, the *Acc* version makes a small percentage (0.75%) of faults in the TM field visible because they hang the GPGPU. This behavior can be explained considering that in the *Ind* approach, each LE is evaluated individually, and so all detections can be labeled as SDC. On the other hand, for the *Acc* method, a permanent fault in one LE affects the synchronization point, thus corrupting the convergence point and causing the Hang condition. More in detail, a Stuck-at-0 fault is a sensitive case during the run of the test program. A fault affecting one LE when used as synchronization causes the Hang condition.

The *Nesting* approach has a marginally lower FC than *Sync-Trick* with an increment in more than double the percentage of faults causing Hanging and Timeout. This fault effect is equivalent to the effect presented in the *Acc* version of *Sync-Trick*. In this case, the *Nesting* method generates IWD to move the address pointer among the LEs in the stack memory, testing all LEs even if a fault is detected, so the next LEs are also evaluated. The continuous evaluation generates issues when a fault affects the LE used for synchronization purposes. Thus, the test program may lose the convergence point and produces the Hang or Timeout condition. According to results, the *Nesting* strategy seems to be more susceptible to Hang and Timeout effect than the *Sync-Trick* using the *Acc* approach.

In both approaches, the addition of the relocation in memory and the SPC functions increases the testable coverage in the stack memory. However, as explained previously, these optional functions can be employed when it is possible to use the entire system memory to relocate the test functions in specific memory locations, or the application code allows this adaptation. Similarly, both SBST approaches can detect a considerable percentage of the permanent faults in the stack memory. However, a direct comparison involving the performance parameters from Table 1 shows that the *Nesting* approach consumes more than double the execution time and 37% of additional instructions. In conclusion, the *Sync-Trick* strategy seems to be a feasible candidate for in-field operations. Moreover, the *Ind* strategy can be divided into parts and adapted with the application code.

A comparison of the FC obtained by the proposed SBST strategies and the representative benchmarks shows that the FC using these specialized programs is higher and effective for this module than the FC obtained with typical applications. Thus, the FC capabilities of the representative benchmarks are lower and can be considered as almost negligible. This behavior can be

explained, considering that most applications only use part of the CMU and the stack memory to manage the IWD. The main issue is that most applications employ only the first levels of stack memory to handle the divergence.

The *matrix multiplication* application generates one level of divergence. Thus, other levels inside the Stack memory are not employed, and the fault effect in not detected or propagated into the application. Similarly, the *Sort* application can generate IWD depending on the input data operands, but it remains limited to the first LE in the stack memory. However, the percentage of detection of 0.33% and 0.19% are negligible in comparison with the proposed test strategies.

The *FFT* benchmark produces two levels of IWD. This behavior slightly increases the percentage of faults detected. Nevertheless, the percentage is small. Finally, the *Edge* application causes two levels of IWD and can detect some faults as SDC and hanging conditions. However, the total coverage of all representative kernels is minimal.

The previous scenario supports the idea that executing applications and checking their results (as it is often done when using a functional test approach) is definitely not enough to verify the functionality of a crucial module in the GPGPU. Thus, special test programs are required to guarantee the correct operation of a module inside a device used in a safety-critical application.

The main advantage of the proposed method to test the stack memory is the modularity and scalability of the SBST strategy. This scalability allows the configuration and the selection of the number of LEs to be tested. Moreover, the test program can be divided into multiple parts when using the Sync-Trick approach.

It is worth noting that the implementation of the test programs required the combination of high-level descriptions (≈20% of the code), when possible, and the addition of assembly functions (≈80%). For all proposed approaches, the synchronization functions (*SSY*) were described in assembly language, considering that these instructions are not possible to specify at CUDA or PTx levels. Moreover, the optimizations of the compiler also remove part of the descriptions. Thus, these parts are rebuilt at the assembly level. These limitations show that the development of test programs for these complex structures in GPGPUs requires access to the assembly language of the micro-architecture to provide efficient solutions. The implementation effort could be reduced by the design of an automatic tool to include the subroutines at the assembly or binary level. Moreover, such a tool could also be employed to target other modules in the GPGPU.

Although the proposed SBST strategies targeted the test of the stack memory in the CMU of a GPGPU with the G80 micro-architecture, we still claim that the strategy can be adapted to other architectures of GPGPUs.

VI. CONCLUSIONS

We introduced and evaluated two functional test strategy (named *Sync-Trick* and *Nesting*) based on the Software-Based Self-Test (SBST) approach aimed for the in-field test permanent faults in the stack memory of the divergence management unit of a GPGPU. The experimental results show that the proposed strategies are effective in detecting up to 98% of the faults in the target structure.

Both test approaches were designed using a modular and scalable mechanism, so a set of parametric functions were created and then combined to test the target structure using different strategies (*Sync-Trick* and *Nesting*). Moreover, the modularity of the solution allows the division of the test program into parts keeping the FC, so adjusting to potential requirements of in-field operation.

As future works, we plan to evaluate the fault coverage in the divergence controller at RT level and gate-level and propose functional test approaches for other critical modules in a GPGPU.

REFERENCES

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

[2] S. Hamdioui, *et al*, "Reliability challenges of real-time systems in forthcoming technology nodes," in *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2013.

[3] I. Agbo, *et al.*, "Read path degradation analysis in SRAM," in *Test Symposium (ETS), 2016 21th IEEE European*, 2016.

[4] X. Chen, Y. Wang, Y. Liang, Y. Xie, and H. Yang, "Run-time technique for simultaneous aging and power optimization in GPGPUs," in *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2014.

[5] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W.-m. W. Hwu, "An adaptive performance modeling tool for GPU architectures," in *15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2010.

[6] B. Coutinho, D. Sampaio, F. M. Pereira, and W. Meira Jr, "Profiling divergences in gpu applications," *Concurrency and Computation: Practice and Experience,* vol. 25, pp. 775-789, 2013.

[7] A. J. Becker, C. A. S. Pathirane, and R. C. Aitken, "Memory built-in self-test for a data processing apparatus," *UK Patent* US 9,449,717 B2, 2016.

[8] R. Gulati*, et al.*, "Self-test during idle cycles for shader core of GPU," *US Patent* US 10,628,274, 2020.

[9] J. E. R. Condia, P. Narducci, M. Sonza Reorda, and L. Sterpone, "A dynamic hardware redundancy mechanism for the in-field fault detection in cores of GPGPUs," in *2020 23rd International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*, 2020.

[10] J. E. R. Condia, P. Narducci, M. Sonza Reorda, and L. Sterpone, "A dynamic reconfiguration mechanism to increase the reliability of GPGPUs," in *VTS 2020: VLSI Test Symposium,* 2020.

[11] M. Psarakis, D. Gizopoulos, E. Sanchez, and M. Sonza Reorda, "Microprocessor software-based self-testing," *IEEE Design & Test of Computers,* vol. 27, pp. 4-19, 2010.

[12] P. Bernardi, M. Grosso, E. Sanchez, and O. Ballan, "Fault grading of software-based self-test procedures for dependable automotive applications," in *2011 Design, Automation & Test in Europe*, 2011.

[13] M. Abdel-Majeed and W. Dweik, "Low overhead online periodic testing for GPGPUs," *Integration,* vol. 62, pp. 362-370, 2018.

[14] S. Di Carlo, G. Gambardella, M. Indaco, I. Martella, P. Prinetto, D. Rolfo*, et al.*, "A software-based self test of CUDA Fermi GPUs," in *2013 18th IEEE European Test Symposium (ETS)*, 2013.

[15] D. Defour and E. Petit, "A software scheduling solution to avoid corrupted units on GPUs," *Journal of Parallel and Distributed Computing,* vol. 90, pp. 1-8, 2016.

[16] D. Sabena, M. Sonza Reorda, L. Sterpone, P. Rech, and L. Carro, "On the evaluation of soft-errors detection techniques for GPGPUs," in *2013 8th IEEE Design and Test Symposium*, 2013.

[17] J. E. R. Condia and R. Sonza Reorda, "Testing permanent faults in pipeline registers of GPGPUs: A multi-kernel approach," in *2019 25th IEEE International Symposium on On-Line Testing and Robust System Design (IOLTS)*, 2019.

[18] J. E. R. Condia and M. Sonza Reorda, "On the testing of special memories in GPGPUs," in *2020 26th IEEE International Symposium on On-Line Testing and Robust System Design* (*IOLTS*), 2020.

[19] S. Di Carlo, J. E. R. Condia, and M. Sonza Reorda, "An On-Line Testing Technique for the Scheduler Memory of a GPGPU," *IEEE Access,* vol. 8, pp. 16893-16912, 2020.

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

[21] S. Di Carlo, J. E. R. Condia, and M. Sonza Reorda, "On the in-field test of the GPGPU scheduler memory," in *22nd IEEE International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS 2019)*, 2019.

[22] J. E. R. Condia, B. Du, M. Sonza Reorda, and L. Sterpone, "FlexGripPlus: An improved GPGPU model to support reliability analysis," *Microelectronics Reliability,* vol. 109, p. 113660, 2020/06/01/ 2020.

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

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