On the testing of special memories in GPGPUs

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

Abstract[1]—**Nowadays, data-intensive processing applications, such as multimedia, high-performance computing and safety-critical ones (e.g., in automotive) employ General Purpose Graphics Processing Units (GPGPUs) due to their parallel processing capabilities and high performance. In these devices, multiple levels of memories are employed in GPGPUs to hide latency and increase the performance during the operation of a kernel. Moreover, modern GPGPU architectures implement cutting-edge semiconductor technologies, reducing their size and power consumption. However, some studies proved that these technologies are prone to faults during the operative life of a device, so compromising reliability. In this work, we developed functional test techniques based on parallel Software-Based Self-Test routines to test memory structures in the memory hierarchy of a GPGPU (FlexGripPlus) implementing the G80 architecture of Nvidia.**

Keywords—***General Purpose Graphics Processing Units (GPGPUs), Software-Based Self-Test (SBST), functional testing, predicate register file, address register file, fault simulation.***

I. INTRODUCTION

General Purpose Graphics Processing Units (GPGPUs) are special-purpose units mainly used as accelerators in data-intensive applications, such as image and video processing, and more recently in high-performance computing. Currently, these technologies are also promising solutions for safety-critical applications, e.g., in the automotive field [1]. Some of these applications (e.g., Sensor Fusion and Advanced Driver Assistance Systems, or ADAS) require devices able to process a large amount of data under real-time constraints. These data usually come from multidimensional sensors (e.g., cameras and radars). GPGPUs are suitable devices for these applications considering their highly parallel capabilities, high performance, and moderate power consumption. Commonly, designers use cutting-edge semiconductor technologies for their implementation to obtain high performance and reduced power consumption.

Nevertheless, some studies [2] have demonstrated that these technologies are prone to introduce faults (such as permanent faults) during the operative life of the device, so compromising the operations and restraining the reliability and safety of a GPGPU. Moreover, some studies analyzed the sensibility of the memory hierarchy of a GPGPU and verified its high susceptibility to faults [3]. The memory hierarchy is composed of multiple memory resources to reduce the latency when executing parallel programs. However, each resource is susceptible to faults. Moreover, the detection of permanent faults in some structures during the operative phase of the device is still an open issue for these architectures.

The detection of permanent faults during in-field operations can be performed using two different functional test strategies. The first strategy is based on Design for Testability (DfT) approaches, which are based on adding specialized structures to test a target module. This solution generates and applies test patterns, and finally detects faults locally: all test operations are done in hardware, thanks to suitably added modules. These modules are included in the design phase, increasing the area and the power consumption of the device.

DfT solutions are practical and commonly used for the end-of-production test. However, they are less effective when used for in-field test, mainly due to the strict real-time execution constraints. In fact, DfT solutions usually destroy the status of the system (i.e., the content of the memory elements), which must be saved before the test and then resumed after it. Some other detection and mitigation solutions, such as those based on Error-Correcting-Codes (ECCs), are costly solutions and can be adopted only for big memory structures within the GPGPUs. Moreover, they mostly target transient faults, and permanent faults may impair their effectiveness with respect to transient faults.

The second approach uses Software-Based Self-Test (SBST), which is based on a set of specially-designed software routines. Each routine is activated when required (e.g., at the power-on, or periodically), and adequately sensitizes the target module, verifying the generated results and propagating a flag or signature stating whether a fault has been detected or not. Commonly, the semiconductor company designs these test routines resorting to rigorous structural metrics to compute the achieved Fault Coverage (FC) for a given fault model in a target design. Currently, many semiconductor and IP provider companies (e.g., Infineon [4], STMicroelectronics [5], Cypress [6], Renesas [7], Microchip [8], and ARM [9]) provide SBST solutions for their products.

Some previous works dealt with SBST solutions targeting GPGPUs. In [10, 11], the authors developed different solutions to test the memory inside the warp scheduler. Similarly, in [12], the authors applied a multi-program approach targeting the pipeline registers in the GPGPUs. Finally, in [13, 14], multiple strategies targeting data-path modules in the GPGPU were proposed. These works proved that functional solutions could effectively be used for in-field tests.

In this work, we propose and explore some SBST strategies targeting the Address Register File (ARF) and the Predicate Register File (PRF), which are specialized memory modules inside a GPGPU. Although their size is relatively small (hence, making their test via DfT or ECC too expensive or infeasible), their correct behavior is critical for the safe GPGPU behavior.

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

Moreover, efficient in-field test techniques for these modules are still missing. Additionally, a compact mechanism to test the Vector Register File (RF) is proposed.

For the purpose of this work, we used FlexGripPlus [15], which is an enhanced GPGPU model we developed starting from the open-source FlexGrip model [16], mimicking the G80 Nvidia architecture and allowing us to assess the effectiveness of the proposed SBST solutions quantitatively.

The paper is organized as follows: Section II describes the general organization of the GPGPUs, and the model used in the experiments. Moreover, this section introduces the memory hierarchy. Section III presents the methods to test the memories in the main cores of a GPGPU using a functional approach based on SBST. Section IV reports some experimental results, and Section V finally draws some conclusions.

II. BACKGROUND

A. General organization of a GPGPU

GPGPUs are based on the Single-Instruction Multiple-Data (SIMD) architecture, according to Flynn's taxonomy [17]. The implementation is composed of multiple parallel execution units (also called Streaming Multiprocessors, or SMs). Internally, each SM includes various execution units (EUs, also known as Scalar Processors, or SPs), some cache (*shared*) memories, a Register File (RF), a warp scheduler controller (WSC), and some dispatcher controllers. Moreover, the SM employs multiple pipeline stages to process warp instructions and improve performance. Nevertheless, the implementation details of these structures are commonly unknown. The available SPs can operate on floating-point or integer numbers to handle an assigned task. Each task is also known as a thread, and the SM can process groups of 32 to 128 threads (also known as *Warps* or *Work-groups*) almost in parallel.

In the SM, the WSC submits an available warp to the SPs to execute the same instruction on each thread. In this parallel architecture, it is common that each thread employs different data operands to execute the instructions, thus generating multiple accesses to the memory system for load and store purposes. Modern architectures of GPGPUs include a hierarchy composed of various levels of memories to reduce the latency and race conditions during the load and store operations.

B. The FlexGripPlus model

FlexGripPlus [15, 18] is an open-source VHDL model and is an improved version of the original FlexGrip [16] implementing the G80 micro-architecture by Nvidia. This new model is fully compatible with the commercial programming environment of Nvidia (CUDA-Toolkit under SM 1.0 compatibility level). FlexGripPlus supports 28 instructions of either 32 or 64 bits in more than 64 formats. The SPs in the model can be configured with 8, 16, or 32 cores.

A set of external parameters are defined in the GPGPU model before starting the operation. These parameters are the number of Blocks per core, the Block dimension, and the Grid dimension. Moreover, other settings are configured by setting proper values in the constant memory, such as the number of registers per thread and the number of blocks per SM core.

More in detail, the micro-architecture of FlexGripPlus is based on a variation of the SIMD taxonomy that is called SIMT (Single-Instruction Multiple-Thread) paradigm. The model exploits a custom SM core with five stages of pipeline (*Fetch*, *Decode*, *Read*, *Execution/Control-flow,* and *Write-back*).

The SM uses a WSC to manage the operation of each thread. In this model, one instruction is executed in parallel per warp or group of 32 threads. It means that one warp instruction is fetched, decoded, and distributed into independent SPs to be processed in the SM.

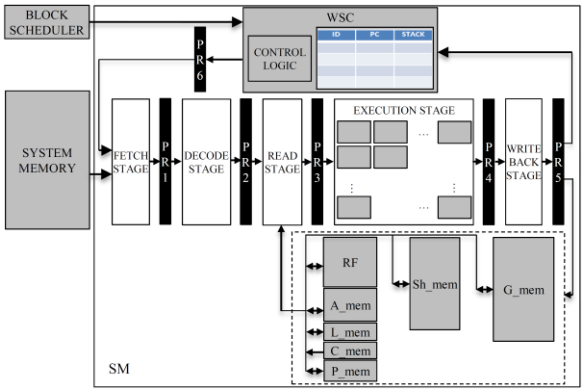

FIG 1. GENERAL SCHEME OF THE SM IN THE FLEXGRIP GPGPU

C. Memory hierarchy in FlexGripPlus

The SIMD taxonomy uses a large set of data operands to operate the same instruction in parallel. This structure generates bottlenecks and race conditions when accessing operands from the memory system. For this purpose, the GPGPUs include multiple memory levels to reduce the latency. These mechanisms are optimized to process data operands mostly organized as arrays or matrices. In this way, each SM includes multiple data memory resources to optimize the information flow for each thread. These resources are the RF, the shared (Sh_mem), Global (G_mem, or main), constant (C_mem), and the local memory (L_mem), as in Fig 1. Moreover, some special-purpose memories store the memory addresses (A_mem) and predicate registers (P_mem).

The memory hierarchy includes several controllers and arbiters to access every memory resource. Initially, a master memory controller activates a separate memory controller when accessing an operand from that particular resource. In the FlexGripPlus architecture, the controllers are located inside the *Read* and the *Write-back* pipeline stages to perform the load and store of operands, respectively.

When processing a program, the compiler usually selects the best trade-off in terms of performance to locate the data operands using the available memory resources in the SM. In particular, the RF stores individual operands. The L_mem stores the operands behaving as arrays. Similarly, C_mem stores constant variables during the operation of a program, and Sh_mem stores those operands used among the threads in a block. Finally, G_mem is used to locate all input data sources and the output results of a program.

The master memory controller decodes the commands coming from the incoming fetched warp instruction. This controller selects the target memory resource and submits a request to the specific memory controller. It is worth noting that both the *Read* and the *Write-back* stages can activate up to 3 simultaneous operations on the memories considering the required number of sources or destinations by the instruction. Some modules operate in parallel and determine the target memory locations to perform the reading or writing operations, depending on the source or destination number.

Memory arbiters manage and order access into the target memories. These arbiters organize the memory access for the threads in a warp, considering that up to 32 loads or stores can be generated per warp parallelly.

The RF is a massive structure composed of 16KB general-purpose registers and located inside of an SM. The WSC divides the RF among the available SPs and the configured threads in a program kernel. The RF is one of the most critical units in the operation of a thread in the SM since most

instructions require a load or a store from/to memory. Moreover, the RF feeds the execution units with the data operands for each thread. The RF also stores the indices for memory addressing, the kernel parameters, and the data and address operands during the execution of one warp instruction.

The P_mem or predicate register file (PRF) stores the predicate flags after each comparison or logic-arithmetic instructions. When the model is configured with 8 SPs, 512 registers of one bit-size are assigned per SP. These registers are distributed in groups of four registers among the available threads. The four registers store the logical state of the zero (Z), the sign (S), the carry (C), and the overflow (O) flags for each thread. The flags remain constant in the subsequent clock cycles until the execution of a new instruction affects their state.

The A_mem or address register file (ARF) adresses the shared and constant memories with additional indices indirectly. The shared memory is commonly used in programs to optimize the performance and is used to access sectors of data organized as arrays or matrices by multiple threads in a program kernel efficiently. Moreover, the ARF reduces the latency in data used frequently by a kernel. Each one of the eight SPs has an associated ARF module composed of 512 registers of 32 bit-size holding up to 128 threads. In this way, four registers (A0, A1, A2, and A3) are assigned to each thread.

III. METHODOLOGY FOR FUNCTIONAL TEST

The proposed strategy uses a functional test approach based on SBST programs to detect permanent faults according to the stuck-at fault model in the RF, PRF, and ARF. Although the target modules correspond to memory, and several sophisticated fault models have been proposed for memories, for the purpose of this paper we only deal with the stuck-at faults affecting single cells: in fact, these memories are likely to be implemented as SRAMs, and the strict time constraints for in-field test would not allow targeting more complex fault models.

A. General features in the memory hierarchy of a GPGPU

The strategy takes advantage of the characteristic features of each target memory and the behavior of the controllers in the memory hierarchy. These features are:

- The target structures ARF and PRF are fully independent for a given SP core. Similarly, the RF is divided among the available SP cores in an SM and managed independently
- Each memory location can be accessed by threads operating in a kernel program, so the maximum thread parallelism allows the lowest latency during a test procedure.

The proposed general strategy functionally tests each memory cell in the target memories considering these two features. Moreover, taking into account that the structures or parts of them are fully independent for a given SP and can be assigned to each thread, the test program must use the maximum capacity of active threads in the SM to access each memory location.

The application of each test pattern employs one out of two strategies. The first employs direct memory movements among registers and a target memory location. The second uses intra-warp divergence. Both methods are effective mechanisms to provide test controllability and generate the required test patterns in the target structures.

The Signature per Thread strategy (SpT) [12] can provide the observability of a fault in a parallel architecture. The SpT allows the individual observation of a fault present in one of the target locations by performing sensitizing operations and updating exclusive signatures according to the presence or absence of a fault. One SpT is assigned to each thread and stores status information of every target location in a module. The SpTs are stored in one or more consecutive memory locations in the G_mem, considering the target of the test: fault detection or diagnosis, respectively.

The thread divergence generation can update each SpT through two execution paths (the *faulty* path and the *fault-free* path) depending on a comparison checking the presence of a fault. On each path, the program loads and updates an SpT with a representative value to propagate the fault in the memory. At the end of the test routine, some external comparisons are performed between the golden values and the SpTs to detect a fault. Since the target structures for the test are memories, the golden values are directly loaded from immediate instructions, so avoiding the use of any memory resource.

B. Proposed general strategy

The proposed methodology is composed of four steps. For each target structure, these steps may be subject to minor adaptations. The steps are:

1) Load test pattern

Immediate instructions (in this case, the pattern is included in the instruction op-code) loads and applyies a pattern to a target memory location. The same mechanism is used to load the golden results for comparison purposes. This approach avoids the use of any memory resource in the SM and potential faults affecting these structures. Four test patterns are employed: 0xFF…, 0xAA…, 0x00…, and 0x55… targeting stuck-at-0 (S/0) and stuck-at-1 (S/1) faults, respectively. If a transparent test is required, the actual values in the registers of the RF or the ARF and their inverted values can also be used replacing the proposed test patterns without major changes.

2) Applying a test pattern

The application of a test pattern uses one of two possible mechanisms: intra-warp divergence and direct memory movements. In case of transparent testing, the actual value is considered as starting pattern.

A test pattern for the PRF requires an indirect approach. In this case, a set of consecutive thread divergence operations (control-flow instructions) are executed, forcing a change on a target predicate flag. A subsequent comparison is employed to update the SpT and propagate the fault. On the other hand, one direct movement instruction applies one test pattern to a target location in the RF or ARF.

3) Parallel propagation of test patterns

The maximum thread capacity in an SM is configured in the test programs to test all memory locations. The same functions are executed on each thread, and the test pattern is propagated almost in parallel. It is worth noting that some minor latency can be present when spreading a test pattern in a target module due to the scheduling of warps in the SM. However, the independent access of each thread into every assigned memory location avoids the inaccuracy of the test caused by the latency so that this latency effect can be neglected.

4) Evaluation and updating of the SpT

When the execution of all threads propagates a test pattern, the SpT is loaded from global memory, and one comparison is performed between a golden value and the value from a register or predicate flag. The fault identification is achieved by using thread divergence paths, starting from a divergence point, see Fig.2 . From this divergence point, two paths are generated (the *fault-free* path and the *faulty* path). Each comparison is performed following a predefined fault-free path, which updates the SpT with golden values only. In this way, a fault is detected in the SpT when one or multiple faulty paths are taken (as an

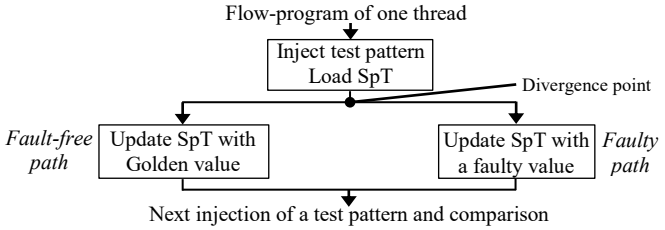

FIG 2. A GENERAL SCHEME OF THE METHOD USED TO UPDATE THE SPTS

effect of faults in the module), and the updated value is different from the golden one.

After the convergence point, the SpT is stored in memory. Finally, the previous process is applied again for the missing test patterns or the inverted value. It is worth noting that the functional test of the state machines in the memory hierarchy controller of an SM is out of the scope of this work, and it is planned as future work.

IV. IMPLEMENTATION

We use the native language of the Nvidia GPGPUs (Shader Assembly language, or SASS) to implement each test program, considering the compatibility supported by FlexGripPlus.

A. Predicate register file PRT

In this module, each thread uses control-flow instructions to generate controlled divergence paths to apply a test pattern indirectly into the registers of this structure. Initially, the first group of divergence paths evaluates S/0 on each register. Then, other groups of divergence paths test the S/1 condition. The activation (logic one) of a flag in the register and a successive comparison can detect a permanent S/0. Similarly, deactivation (logic zero) and a comparison can detect the S/1 condition in a register field. A predicate flag is activated or inactivated as a product of one comparison (*X_SET type*) instruction.

Each divergence path uses carefully selected logic-arithmetic operations to propagate a fault to the main memory. The divergence paths update and store back the SpT to identify and to disseminate a fault in a memory cell. In this way, the golden path updates the SpT with an acumulative golden value. On the other hand, the faulty path performs a faulty update. Thus, a later evaluation can identify a fault in one of the flags of the PRT as an error observed in the G_mem memory. It is worth noting that the test program targets an individual flag in the register per divergence paths.

One convergence point indicates that a thread finished the divergence operation, and a new one starts targeting a different flag in the predicate register. Thus, multiple consecutive and independent divergence paths are effective to test the PRF.

The generation of each divergence path is intended to keep thread coherence in the test program. In this way, if a fault is present in one of the threads, a faulty path would be executed, and the SpT of this individual thread would denote the fault by a change in a signature. However, the program execution is not stopped or hanged by the detection of a fault in this module, so allowing the detection by observing the memory content, only.

Once the comparison is executed, a target flag is modified and stored in the PRF. It is worth noting that the effect of this comparison in the flag is extended and remains for multiple instructions cycles. In this way, after the application of a test pattern, two consecutive control-flow instructions are executed, checking the target flag, thus reading the target predicate register. In each case, the *faulty* or *fault-free* paths are executed, updating the SpT, as described below. The previous process can be used to detect S/0 faults. In the case of the S/1, the process is similar. In this scenario, the flags are forced to zero (*cleaning operation*) by carefully selecting one operation. Then, the execution of the two paths updates the SpT to detect ay fault in the flag. Finally, after the evaluation of each flag in the register, a new register is targeted to perform the same procedures. Figure 3 shows the assembly instructions describing the test procedure for the PRF and ARF modules.

B. Vector Register File RF:

The test of permanent faults in the RF can follow the classical method from the literature, such as March algorithms. However, this method can compact the test patterns using the features in the RF module, the available instructions, and the redundant operation of the threads in the program.

The method injects test patterns targeting S/0 and S/1 independently. This independent approach reduces the number of instructions required to update the SpT. Three SpTs (SpT1, SpT2, and SpT3) are employed in the method to test and compact the detection of a possible fault. SpT1 and SpT2 are signatures devoted to store any S/0 and S/1, respectively. SpT3 is the compact signature to detect any fault independently of the type. It is worth noting that using these three signatures, it is possible to perform detection and also diagnosis.

In the first stages of the GPGPU execution, the register R_0 contains the thread indexes. These indexes are used by each thread to access the memory resources, so this register is the last to be tested, avoiding the loss of the thread indices.

In the proposed method, the indices are combined with the target memory locations to store and load the SpT. Initially, R_0 handles the address of the SpT in memory and sustains this value during the program execution. The implemented SBST procedure consists of the following steps:

1) Initialization of the registers in RF (excluding R_0) with one test pattern (all 0s or all 1s)
2) Execution of one logic operation between a target register and a constant value. The AND and the OR bit-wise operators evaluate the S/1 and the S/0 conditions, respectively
3) Store the result in global memory as SpT1
4) Change of the address value in R_0. Repeat steps 1, 2, and 3 using the missing test pattern and logic operation
5) Store the result in global memory as SpT2
6) Move the content of R_0 to other registers, assigning the first test patterns to R_0
7) Load SpT1 and operate with R_0, Store the result as SpT1
8) Repeat steps 6-7 with the missing pattern and store results as SpT2
9) Load SpT1 and SpT2 and compact as SpT3.

SpT3 can represent the fault effect of a permanent fault in the RF. The expected value is all 1s. A mismatch in this value represents the fault in one of the registers assigned to each thread. It is worth noting that one SpT is stored in the memory for each of the configured threads in the test kernel.

The diagnosis is performed by sequential comparisons of the SpT1 and SpT2 with golden values. This method is useful to detect any individual fault in any register of the RF. It should be noted that only one fault can be detected. If two or more permanent faults are present in a register, the last one is reported only, due to the sequential comparison. These comparisons require the use of one predicate flag. The result of this flag determines if a fault is present in a register location, then new comparisons are performed. After each comparison, a cleaning procedure cleans the flag to avoid inconsistent fault detection.

C. Address Register File ARF

The direct movement instruction between a data register and one address register in ARF starts the injection of a test pattern. Then, an address movement (**ADA**), see Fig. 3, propagates the test pattern among the registers. Moreover, it is assumed that the RF module operates correctly.

The test program uses the maximum capacity of threads (1,024) in the SM to test the entire ARF. More in detail, each thread can access four address registers (A0-A3). Moreover, the SpT uses two consecutive memory locations and propagates the effect of a permanent fault in any address register. The first location stores the type of the fault (S/0 or S/1) with logic states and the location of the fault in the register. The second location in memory indicates if a fault is present. It is worth noting that each update to the SpTs uses different paths in the program. These paths are generated using internal comparisons.

This test method applies test patterns 0xAA… and 0x55… and forces a sequence of individual comparisons with a golden value on each thread and identifies faults by mismatches in the target address register. Those comparisons generate test paths. Each test path was optimized to execute a minimum number of instructions. In the end, each path has the same amount of instructions to maintain an equivalent execution time. Moreover, the same predicate register is reused to reduce the resource overhead. One comparison instruction affects the predicate register cleaning the flag before each comparison. A test pattern is applied via the following steps:

1) Movement of a test pattern from one General Purpose Register (GPR) into one of the address registers in the ARF
2) Copy of the test pattern to the address registers
3) Retrieve information from ARF into a set of four GPRs
4) Compare GPR 1 with the golden value
5) Classify the possible fault (or fault-free)
6) Repeat steps 4 and 5 for the missing address registers
7) Change the test pattern and restart again.

V. Fault Injection Environment

The valiadation of the developed SBST test programs was performed in a custom fault injection environment based on the *ModelSim* framework. The fault injector follows the guidelines introduced in [12, 19], and for the purpose of this work, injects permanent faults in the target memories. The injector is composed of the fault injector controller (FIC), a fault injector decoder (FID), and a fault injector checker and classifier (FICC). The FIC manages the configuration of the GPGPU model and the simulation framework. Moreover, it begins and finishes the fault simulation. The FID translates, from an input fault list, one fault into an equivalent sequence of commands managing a fault. These commands are applied to the model before the simulation starts. The FICC checks the fault effect in the model and the method used for finishing the simulation. Finally, the FICC classifies the fault effect. The faults are classified as *i) Silent Data Corruption fault* (SDC), when a fault generates mismatches in the memory results, allowing its detection. *ii) Hanging or Crash fault*, when a fault stops the program execution or avoids the correct termination. *iii) Timeout fault*, when a fault affects the system and produces a change in the execution time of the program. In this case, the memory results are not affected, and *iv) Masked* when a fault does not affect the system execution and the results.

A fault injection campaign starts by defining and sending an input fault list to the FIC. The fault list is composed of the target fault model (*Stuck-at*) and the location in the target module. Each line in the fault list includes a target location for fault injection. Then, the FIC starts a fault-free (*golden*) simulation to store the memory results and the simulation time as reference parameters during the fault campaign. The fault simulation time is fixed as twice the golden simulation time to detect potential *timeout* fault effects. Then, the FICC compares the results in memory and the execution time to classify a fault.

A new fault simulation then starts again by reading another line from the input fault list and finishes when there are no more lines in the fault list. At the end of the fault campaign, one fault report file is created that describes the effect of every fault in the system. A second fault report includes a quantitative classification of the faults.

```
1  …                                          1  …
2  MVI R8, 0x0;                               2  MVI R2, 0x55555555;
3  MVI R7, 0xaaaaaaaa;                        3  R2A A1, R2;
4  SSY 0x8c;          // Divergence Point     4  ADA A2, A1, 0x00;
5  ISET.S32.C0 o [0x7f], R7, R8, NE;          5  …
6  BRA (C0.NE), 0x7c;                         6  A2R R5, A3;
7  GLD.U32 R8, global14 [R0]; // faulty p.    7  MVI R8, 0x55555555;
8  IADD32I R8, R8, 0x6;                       8  ISET.S32.C0 o [0x7f], R5, R8, EQ;
9  BRA 0x8c;                                  9  GLD.U32 R8, global14 [R0];
10 GLD.U32 R8, global14 [R0]; // fault-free p. 10 SSY 0x190;          // Divergence point
11 IADD32I R8, R8, 0x1;                       11 BRA (C0.NE), 0x188;
12 NOP.S;          // Convergence Point       12 IADD32I R8, R8, 0x10000;//fault-free p.
13 GST.U32 global14 [R0], R8;                 13 BRA 0x190;
14 …                                          14 IADD32I R8, R8, 0x1;     // faulty p.
15                                            15 NOP.S;          // Convergence point
16                                            16 GST.U32 global14 [R0], R8;
17                                            17 …
```

Fig 3. Fragments of the implemented SBST to test the PRF (left) and ARF (right) memories

VI. Experimental Results

The FlexGripPlus model was programmed with 8 SPs, and each test program was configured with the maximum number of threads per block affordable for an SM. The fault simulation experiments were performed on a workstation with an Intel Xeon CPU running at 2.5 GHz, equipped with 12 cores, and 256 GB of RAM. During the fault injection campaign, a set of representative benchmarks and the developed test programs were evaluated, targeting the PRT, RF, and ARF modules.

Three representative applications are selected to compare the capabilities of fault detection of typical workloads with those of the proposed test strategies (*PRF_T*, *ARF_T,* and *VRF_T*). These applications are the matrix multiplication (*MxM*), the Bitonic Sort (*Sort*), and the Fast Fourier Transform (*FFT*). The *MxM* application uses the shared memory to load parameters and employing a tiling approach to operate them efficiently. In contrast, the *Sort* application is based on thread divergence and use of the RF to perform the sorting of data operands. Finally, the *FFT* implements a butterfly structure that operates the Fourier transformation in one dimension. Additional details regarding the selected benchmarks can be found in [12, 18]. Table 1 reports the key parameters about the representative benchmarks and the implemented test programs for each module under test.

As explained previously, the same test program is executed in parallel by each thread targeting different locations in the tested structures. In each case, 4,096 memory locations (bytes) are required to store the detection results as SpTs. It is worth noting that the reported version of the PRF test program uses sequential reading and writing operations to update the SpT of each thread. This condition causes an additional latency (observed in total execution time) by the continuous operations in G_mem. On the other hand, the representative benchmarks were configured with 1,024 threads for the MxM application and 64 threads for the Sort and FFT applications.

An initial fault campaign injected faults in the entire structures of the PRF and the ARF. In this case, the fault list was divided into 8 and 32 pieces for the parallel fault campaigns. For the PRF and the ARF, a total of 32,768 and 262,144 faults were injected, respectively.

The target memories are regular structures in the GPGPU design, which are distributed equally among the available SPs. Moreover, test programs are designed to access data operands independently. Thus, it is possible to perform the fault injection in one structure belonging to any SP and determine the complete fault coverage (FC) of all similar structures in the SM. The FC is computed using all faults detected and classified in

one of the possible classifications different from *masked* as described previously in section V.

TABLE 1. PERFORMANCE PARAMETERS OF THE IMPLEMENTED TEST PROGRAMS

Benchmarks or SBST kernels	Execution time (Clock Cycles)	Number of Instructions
MxM	774,437	294
Sort	233,720	26
FFT	96,373	168
PRF_T	1,890,106	434
ARF_T	338,240	122
VRF_T	108,958	82

TABLE 2. FC FOR THE TWO VERSIONS OF THE FAULT CAMPAIGN

Fault campaign		PRF	ARF
Complete	**Fault list size**	32,768	262,144
	SDC (%)	100.0	100.0
Reduced	**Fault list size**	4,096	32,768
	SDC (%)	100.0	100.0

TABLE 3. FC FOR BENCHMARKS AND TEST PROGRAMS

Benchmark or SBST kernel	Target module	Total faults	SDC (%)	Halt (%)	Total FC (%)
MxM	**PRF**	32,768	0	0.38	0.38
	ARF	262,144	25.07	0.0	25.07
	RF	262,144	18.26	8.24	26.5
Sort	**PRF**	32,768	0.16	0.04	0.20
	ARF	262,144	0.0	0.0	0.0
	RF	262,144	0.18	0.07	0.25
FFT	**PRF**	32,768	0.15	0.19	0.34
	ARF	262,144	0.0	0.0	0.0
	RF	262,144	0.19	0.21	0.4
PRF_T	**PRF**	32,768	100.0	0.0	100.0
ARF_T	**ARF**	262,144	100.0	0.0	100.0
VRF_T	**RF**	262,144	100.0	0.0	100.0

We performed individual fault campaigns targeting only one memory structure of a particular SP at a time. Then, we repeated the experiments, focusing on all memories belonging to all SPs in the SM. In the end, the fault lists were reduced on each case, and 4,096 and 32,768 faults were injected for the individual fault campaigns targeting the PRF and ARF modules, respectively. Table 2 presents the FC results for both targets in the fault campaign (*complete* and *reduced*). Detection results of *Timeout* and *Halt* are zero (0%) for both versions. Results allow us to affirm that in case of massive fault injection campaigns, the performance increases significantly (by reducing the execution time) when the target of the fault injection includes identical modules in structure and function. Both reduced versions of the fault campaign compressed the entire fault simulation in a proportion of 8 (as the number of SPs). It is worth noting that the target structures are evaluated by the test programs using embarrassingly parallel instructions. Moreover, there is no interaction among the threads and their operands.

Nevertheless, the previous procedure is not entirely valid when a program includes intra-warp divergence, warp barriers, or when the execution depends on data operands. Table 3 reports the results of the experiments for the representative benchmarks and the test program for the different modules under test. Finally, no faults causing timeout conditions were detected for each benchmark or implemented SBST strategies.

As observed in the results of Table 3, the proposed methods are useful in testing the PRF, the ARF, and the RF modules independently. The propagation of the fault effect into global memory is the main advantage of the proposed technique, and 100% of detections are mapped to the G_mem using the SpT mechanism. In contrast, the representative benchmarks are ineffective in detecting faults from the modules under test. The previous behavior can be explained considering that all the applications employ only parts of the RF, ARF, and PRF during the execution of the program, so some faults are not propagated or affect the functionality of the program. However, it shows that an elaborated test program is required to test special structures such as the ARF and PRF. Although we used the FlexGripPlus model to develop and validate the proposed SBST strategies. We claim that the proposed methodology can be adapted and used for the most recent GPGPU architectures, such as Maxwell and Pascal, which include similar structures.

VII. CONCLUSIONS

We introduced a methodology to develop self-test routines targeting some of the memories composing the memory hierarchy in GPGPU devices. The proposed solutions take advantage of the regularity of these structures. We proved that SBST methods can be effectively developed when targeting regular structures in GPGPUs.

We adopted the proposed methodology to test the targeted structures in a sample GPGPU. In each case, the use of different test patterns was critical for accessing the structures. The use of the Signature per Thread (SpT) mechanism, to support the observation of fault effects, is particularly effective when addressing the test of complex structures in parallel architectures. The gathered results show that 100% of stuck-at faults affecting single cells of each memory can be detected. In contrast, a very low fault coverage figure can be obtained, resorting to usual application codes.

REFERENCES

[1] W. Shi, M. B. Alawieh, X. Li, and H. Yu, "Algorithm and hardware implementation for visual perception system in autonomous vehicle: A survey," *Integration,* vol. 59, pp. 148-156, 2017/09/01/ 2017.

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

[3] P. Rech, C. Aguiar, R. Ferreira, C. Frost, and L. Carro, "Neutron radiation test of graphic processing units," in *2012 IEEE 18th International On-Line Testing Symposium (IOLTS)*, 2012, pp. 55-60.

[4] Infineon Technologies. (2020). *https://www.hitex.com/tools-components/software-components/selftest-libraries-safety-libs/*

[5] ST Microelecronics, "AN3307 Application note Guidelines for obtaining IEC 60335 Class B certification for any STM32 application," , 2016.

[6] Cypress, "AN204377 FM3 and FM4 Family, IEC61508 SIL2 Self-Test Library," ed, 2017.

[7] Renesas Electronics. (2020). *https://www.renesas.com/eu/en/products/synergy/software/add-ons.html*.

[8] Microchip Inc., "DS52076A 16-bit CPU Self-Test Library User's Guide," ed, 2012, p. 52.

[9] ARM. (2020). *https://developer.arm.com/technologies/functional-safety*.

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

[11] 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.

[12] J. E. R. Condia and R. Sonza Reorda, "Testing permanent faults in pipeline registers of GPGPUs: A multi-kernel approach," in *2019 IEEE 25th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, 2019, pp. 97-102.

[13] 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, pp. 1-6.

[14] S. Di Carlo, G. Gambardella, I. Martella, P. Prinetto, D. Rolfo, and P. Trotta, "Fault mitigation strategies for CUDA GPUs," in *Test Conference (ITC), 2013 IEEE International*, 2013, pp. 1-8.

[15] 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.

[16] K. Andryc, M. Merchant, and R. Tessier, "FlexGrip: A soft GPGPU for FPGAs," in *2013 International Conference on Field-Programmable Technology (FPT)*, 2013, pp. 230-237.

[17] M. J. Flynn, "Some Computer Organizations and Their Effectiveness," *IEEE Transactions on Computers,* vol. C-21, pp. 948-960, 1972.

[18] 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.

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