On the Functional Test of Special Function Units in GPUs

On the Functional Test of Special Function Units in GPUs

Juan-David Guerrero-Balaguera, Josie E. Rodriguez Condia, Matteo Sonza Reorda
Department of Control and Computer Engineering
Politecnico di Torino, Torino, Italy
{juan.guerrero, josie.rodriguez, matteo.sonzareorda}@polito.it

***Abstract* —The Graphics Processing Units (GPUs) usage has extended from graphic applications to others where their high computational power is exploited (e.g., to implement Artificial Intelligence algorithms). These complex applications usually need highly intensive computations based on floating-point transcendental functions. GPUs may efficiently compute these functions in hardware using ad hoc Special Function Units (SFUs). However, a permanent fault in such units could be very critical (e.g., in safety-critical automotive applications). Thus, test methodologies for SFUs are strictly required to achieve the target reliability and safety levels. In this work, we present a functional test method based on a Software-Based Self-Test (SBST) approach targeting the SFUs in GPUs. This method exploits different approaches to build a test program and applies several optimization strategies to exploit the GPU parallelism to speed up the test procedure and reduce the required memory. The effectiveness of this methodology was proven by resorting to an open-source GPU model (FlexGripPlus) compatible with NVIDIA GPUs. The experimental results show that the proposed technique achieves 90.75% of fault coverage and up to 94.26% of Testable Fault Coverage, reducing the required memory and test duration with respect to pseudorandom strategies proposed by other authors.**

Keywords— Functional Testing, Graphics Processing Units, In-field test, Software-based self-test, Special Function Units

I. INTRODUCTION

GPUs are increasingly used not only for consumer applications but also in domains (e.g., automotive, high-performance computing, robotics) where dependability (and in particular safety) plays a crucial role. In the latter case, their complexity and the very advanced semiconductor technologies used for their manufacturing may increase the probability of faults well beyond the acceptable thresholds [1].

When considering permanent faults, this means that the probability of new faults arising during the operational life (due to aging or other phenomena) may be significantly higher than acceptable and induce an unacceptably high failure probability at the application level.

To face this situation, a possible solution lies in adopting devices (e.g., NVIDIA Xavier), where hardware-based hardening solutions (e.g., [2]) can guarantee the ability to detect and possibly detect a very high percentage of faults. However, the usage of different ad hoc devices for safety-critical applications may involve a significant increase in their cost with respect to the "normal" ones. It may also impact their performance, power consumption, and ease of use.

Another solution may be represented by effective in-field test solutions. In principle, they may detect faults before they produce critical failures, thus lowering the failure probability to acceptable values.

Clearly, the ability to effectively detect permanent faults possibly affecting a GPU while the same is already in the operational phase is a real challenge. Design for Testability (DfT) solutions (such as Logic and Memory BIST) can be effectively exploited if the in-field test is performed at the power-on, when timing constraints are more relaxed. However, in some cases, the hardware cost of DfT solutions is too high, and/or the application safety constraints ask for a more frequent in-field test, for example, using the time slots left idle by the application. In such a scenario, software-based self-test (SBST) [3] may represent a promising solution for GPUs, following what is already widely done for MCUs and SoCs used for safety-critical applications, especially in the automotive domain[2]. An important advantage of such solution lies in the fact that the semiconductor company (owning full details about the product) can develop suitable Self-Test Libraries (STLs) able to achieve a given Fault Coverage, which has been computed via Fault Simulation. The STLs are then provided to the system company using the device, which is in charge of integrating them in the application code and activating their execution at the due time and with the required frequency.

An STL is basically a set of software procedures that can activate and make visible the effects of possible permanent faults possibly affecting a module or device. At the end of its execution, the test code itself is able to check the results produced by each procedure and discriminate between good and faulty devices, possibly returning error information.

In past works [4][5][6][7][8], several techniques have been proposed for developing effective STLs for GPUs, showing that they can achieve good fault coverage figures with respect to stuck-at faults, as mandated by standards (e.g., ISO 26262).

Current GPUs are frequently used to implement Machine Learning applications. In such a case, they are often equipped

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

[2] The list of semiconductor and IP companies providing their customers with SBST-based test procedures allowing to in-field test their devices includes ARM, STMicroelectronics, Infineon, Cypress, Renesas, Microchip.

with *Special Function Units* (SFUs) able to efficiently perform transcendental computations (in particular, exponential, logarithmic, and trigonometric functions).

The authors in [7] proposed several custom test strategies, one for each component inside the GPU architecture, and until now, it is the only work that considered functional testing on SFUs. However, the lack of architectural details of the SFUs and observability forced them to develop test programs using random values and evaluate the fault detection capabilities through the input and output interfaces of some GPU internal module, only. Then, the achieved fault coverage could only be estimated using an error threshold parameter obtained from the tolerance characterization of the SFU.

In this paper, we extend our previous work on the SBST-based in-field test of GPUs and propose a solution to develop effective STLs for SFUs. The method mimics the basic idea reported in [9] for a simple pipelined CPU to leverage the test patterns generated at the module level by a combinational ATPG and then transform them into a parallel program (CUDA kernel). To the best of our knowledge, this is the first work providing a specific method to generate a functional in-field test for a SFU in an GPU and provide quantitative results of functional test methods, using a microarchitectural model of a GPU.

In order to validate the effectiveness of the proposed method, we used an open-source GPU model we developed (named *FlexGripPlus*) [10], enriching it with some SFUs [11]. FlexGripPlus is compatible with the NVIDIA GPU architecture and development flow. In the rest of the paper, we describe the techniques we propose to develop effective STLs for the SFU and report experimental results proving their effectiveness. In particular, we show that they can achieve 90.75% of stuck-at Fault Coverage and demonstrate that some of the undetected faults are untestable under operating conditions, so obtaining a 94.26% of Testable Fault Coverage. These results significantly outperform those obtainable with the method in [7].

Although this paper mainly refers to the architecture on an NVIDIA GPU, most of the ideas and techniques we describe can be extended to other GPU architectures as well.

The paper is organized as follows. Section II provides some background about GPU organization and overviews the main features of the micro-architecture of the FlexGripPlus model and its SFU specifications. Section III describes the methodology to develop the STL for the SFU. Section IV reports the experimental results and their analysis, and Section V draws some conclusions.

II. Background

A. GPU oganization

The Graphics Processing Units (GPUs) are organized based on arrays of parallel execution units (also known as *Streaming Multiprocessors* or SMs). The SM is the main execution core inside a GPU, and it implements the *Single-Instruction Multiple-Data* (SIMD) paradigm or a variation, such as the Single-Instruction Multiple-Thread (SIMT). In this way, each SM includes several functional units (also known as Streaming Processors or SPs), which are used to process instructions in several threads. The number of SPs per SM may vary in a range from 8 to 128 and depends on the architecture and the number of parallel threads to be processed simultaneously. Furthermore, other functional units, such as Special Function Units (SFUs) and, more recently, tensor cores are also included in a minor number into the SMs to perform specific operations and support multimedia and artificial intelligence based on 'Convolutional Neural Netwoks' (CNN) applications.

More in detail, a program kernel (parallel program executed in the GPU and called by the Host) is divided into parts by a general controller and assigned to the available SMs. Then, each SM loads one instruction from the program code and processes it in parallel through the available SPs. The program is divided into thread-groups (or *Warps*), and one new instruction is processed when all warps finished the execution of the previous one. Some modern GPU architectures may include additional local controllers (i.e., two controllers) in the SM to execute multiple instructions at the same time by dividing the SPs, so executing a limited number of threads per instruction in parallel, but improving performance.

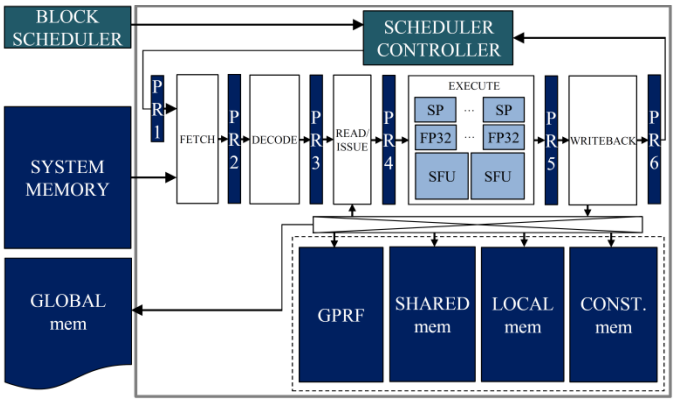

Fig. 1 A general scheme of the SM in FlexGripPlus

The GPU architecture also includes a memory hierarchy mainly used to reduce latency during the kernel execution. The memory resources include a '*General Purpose Register File'* (GPRF), a shared memory, a local memory, a constant memory, and a global/main memory. The GPRF is organized in banks and is associated with each SP. Furthermore, the shared memory is also organized in banks and can be addressed by any thread. GPRF and shared memory are the in-core structures of the SM. The other memories are external resources.

B. FlexGrip GPU Architecture

The FlexGripPlus model [10] is an open-source soft-core GPU implementing the NVIDIA's G80 architecture [12]. This model is an improved and extended version of the FlexGrip model [13]. FlexGripPlus supports up to 52 assembly instructions and is compliant with the CUDA programming environment. The organization of FlexGripPlus is based on an array of SMs. One general controller (*block scheduler controller*) commands the tasks submitted to the SMs. In each SM, a local controller (*warp scheduler*) manages the task by dispatching a warp into the available SPs. The SM is divided into five pipeline stages, and some pipeline registers (PRx) dividing the stages (see Figure 1) and executes one instruction following the Single-Instruction Multiple-Thread (SIMT) paradigm. More in detail, the SM includes 8 SPs, 8 Floating Point Units (FP32), and two

Special Function Units (SFUs). Furthermore, the flexibility of the model allows the selection of the number of execution units (SPs, FP32) per SM (8, 16, or 32). Similarly, the number of SFUs corresponds to 2, 2, or 4 per SM.

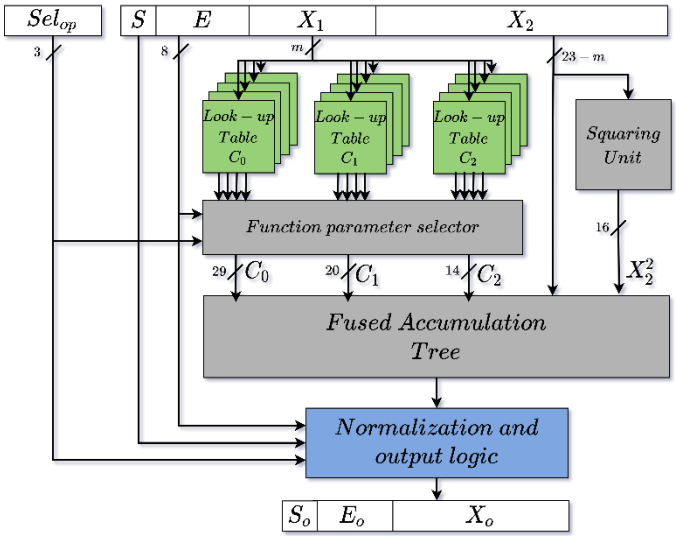

Fig. 2 Block diagram of the SFU architecture.

C. The SFU

The available SFU module in FlexGripPlus is an improved version of the first open-source SFU presented in [11]. This SFU can perform six transcendental functions (sin(x), cos(x), 1/sqrt(x), log2(x), 2^x, 1/x and sqrt(x)) using the IEEE 754 floating-point standard with single-precision format. The SFU implements a fast function approximation using a Minimax Quadratic Interpolator [14]. The component is described in VHDL, and it is similar to what is specified by the NVIDIA G80 architecture [12].

The SFU module evaluates transcendental functions employing a set of lookup tables (LUTs) used to retrieve the coefficients for the quadratic polynomial approximation followed by a fast evaluation of the polynomial. The approximation's coefficients are obtained according to the enhanced minimax approximation algorithm described in [14].

A general scheme of the SFU architecture is presented in Fig. 2. It is composed of: *i)* lookup tables that contain the coefficients of each function that will be evaluated. $X1$ addresses a set of three LUTs to get three finite-word coefficients $C0$, $C1$and $C2$. *ii)* a special parameter selector, which has the function of selecting the parameter to be evaluated, taking into account the selector (Sel_{op}), sign and exponent value. *iii)* a squaring unit, which is in charge of calculating the fast square operation of X2. *iv)* The Fused Accumulation Tree (FAT), which is a systolic array of Carry-Save Adders (CSAs) that performs a fast evaluation of the polynomial approximation of a function $f(X)$, in the range $X_1 \leq X < X_1 + 2^{-m}$ using the expression presented in equation (1) Finally, *v)* the normalization and output logic adjusts the exponent and evaluates special cases according to the IEEE 754 representation. The m parameter, in $X1$ and $X2$, determines the number of bits used to address the LUTs. This parameter depends on the function to be evaluated and the target precision. In this work, the SFUs modules use m = {6,7} selected according to [14].

$$f(X) \approx C_0 + C_1 X_2 + C_2 X_2^2 \quad (1)$$

Each function to be approximated uses one or two sets of LUTs [14]. The set of tables is selected according to the exponent (E) of the input operand. For example, the $log2(x)$ function uses one set of tables when $E = 0$ and another when $E \neq 0$. In contrast, for the $rsqrt(x)$ and $sqrt(x)$ functions the correct set of tables depends on whether the exponent is even or odd. The functions $sin(x)$, $cos(x)$, 2^x and $1/x$ require only one set of LUTs for each one.

The GPU has several SFU components (named *SFU cores*) and one special dispatcher that controls the threads distribution to each SFU core. We called *SFU processor* this set of components.

III. Proposed Approach for the Functional Test of SFUs

The proposed methodology aims to define a logic flow for the generation of in-field test programs to be executed by the GPU and targeting faults in the SFU.

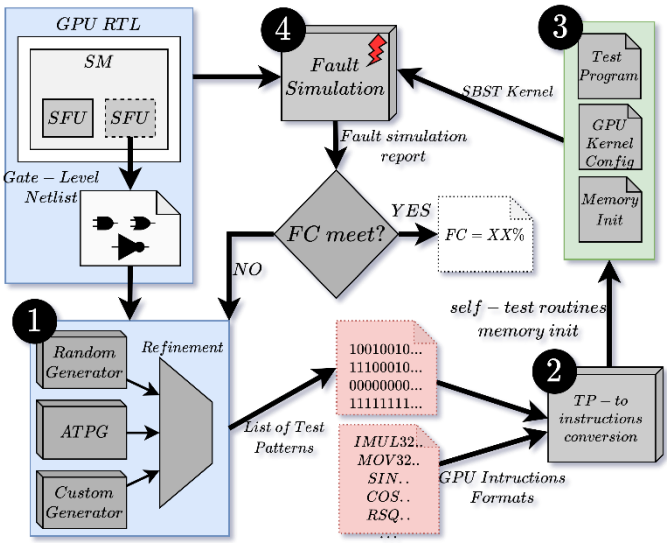

Fig. 3. A general scheme of the method for SBST generation

A. Basic flow

The proposed methodology is composed of a sequence of four stages: *i)* Test Pattern (TP) generation, *ii)* TP conversion into test routines, *iii)* SBST kernel construction, and *iv)* fault simulation of the SBST kernel. Fig. 3 shows a general scheme of the proposed method highlighting each stage.

In the first stage (*TP generation*), three different sets of test patterns are generated starting from the gate-level netlist of the SFU core. The first set of TPs is generated using a random approach without restrictions; the second set is generated resorting to an ATPG. The third set is created using an ad-hoc generation method that performs a customized TPs generation targeting the most critical components in the SFU. The three sets of TPs are reduced to one set of TPs through the elimination of redundant TPs.

In the second stage, the TPs are automatically converted into GPU instructions. These instructions are selected so that they apply the target TPs to the SFU. Then, a self-test routine is developed using the selected instructions for each TP plus

additional instructions employed to implement a Signature-per-Thread (SpT) mechanism [6]. This mechanism resorts to logic and shifts instructions to update the signature value after the application in each TP. At the end of the self-test routine, the computed and expected SpTs are compared. If they do not match, then a fault is detected.

The overall self-test library includes several independent routines (e.g., one per each SFU operation). In this way, we can split the test process for the SFU, reducing their duration and more easily matching the time constraints for in-field test.

After the generation of the self-test routines, the third stage turns them into an SBST kernel. For this purpose, the SBST kernel requires the specifications of TP allocation (as operands of instructions or as elements of the global memory). Moreover, other configuration parameters, including the number of Treads-per-Block (TpB), the number of Blocks-per-Core (BpC), and pointers initialization are defined.

In the last stage, a fault simulation is performed to evaluate the fault coverage achieved by the developed SBST kernel. If the results are too low, the process could be restarted from the first stage increasing the effort or the amount of the generated TPs, targeting the still undetected faults.

B. Stage 1: Test Pattern Generation

In this stage, we perform the generation of a compact set of TPs starting from a combination of three independent sets of TPs. The first set of TPs is based on the generation of random input patterns. This set of test patterns also includes edge conditions, such as NaN, $\pm\infty$, ± 0, and subnormal's representation of the IEEE 754 values for each operation in the SFU. The second set of TPs is generated using an ATPG targeting the combinational part of the SFU, only. However, the complexity of the SFU structure does not allow the ATPG to generate TPs for all possible faults. For this reason, a further set of TPs is required.

This third set of TPs is created resorting to a custom hand-made approach, targeting the interconnections among components in the SFU and the faults in the combinational part of the SFU that were not yet detected.

Finally, the three sets of TPs are joined into a single set of TPs. However, we can reduce the size of this set of TPs by removing replicated or useless TPs. This optimization can be performed by fault simulation of the combinational part of the SFU.

C. Stage 2: Self-Test routine development

The construction of the self-test routine is developed in three main steps: *i)* TPs classification, using the constraints given by the ISA, *ii)* mapping of TPs into equivalent instructions, and *iii)* addition of signature mechanisms to detect the possible fault presence.

The TPs classification step identifies those TPs that cannot be transformed into GPU instructions due to ISA restrictions, limiting the full control of the primary inputs of the SFU. This situation is present, for example, when one TP requires a specific combination of values in the selection inputs of the SFU, but the existing instructions of the GPU do not allow that combination. In this case, the TP cannot be applied, it is classified as "invalid TP", and it is discarded.

In the second step, the TPs classified as "valid" are converted into instructions. Each TP has two main fields: the operand data field, and the function field. The function field of the TP selects the matching instruction that executes the desired function in the SFU. This instruction refers to a source register (containing the operand data of TP) and destination register where the fault effect will be observed through the SpT mechanism.

The allocation of the data operand in the GPU registers can be managed using two main approaches. The first consist of loading the value of the data operand directly on a register using the Register Immediate Addressing (IMM – routines) in this case the immediate values are part of the opcode of the instruction. In the second approach, the data operands are read from memory to a register using the Memory Addressing (MEM – routines). Any of those techniques guarantee the injection of the TPs to the SFUs during the runtime execution.

1) IMM routines

This approach employs the register immediate-addressing mode defined by the GPU ISA. This operation moves the operand data and the expected SpTs (*immediate*) directly to a general-purpose register for every thread in the GPU. It is possible to employ one or a combination of instructions for this purpose, depending on the ISA specifications and the available instructions.

The main advantage of the proposed SBST approach is that it only uses registers and the target SFUs to perform its job. Any other memory resource in the GPU core is not employed during the execution of the SBST program. Furthermore, the SBST program can be implemented by organizing the TPs in any order. However, we anticipate that the execution time may require a considerable number of clock cycles, because the GPU core executes the test program in a purely sequential manner, so neglecting the implicit parallel capabilities of the GPU architecture.

2) MEM routines

The limitations of the immediate-addressing strategy can be avoided by storing the TP values (for each SFU function) in any memory resource of the GPU. This method permits the execution of parallel threads in the GPU core, so exploiting the execution of the same functions in the SFUs to inject different TPs (as operands) per thread and allowing a faster test program execution. Hence, the test program can use the TPs to exploit the SIMT organization in the GPU core (the same operation is performed with different values in the different threads).

In this memory-based approach, the TPs are grouped according to the function field that they have in common. Then, the number of TpB is defined to configure the SBST kernel. This parameter is specified relying on the average number of TPs to be applied, since the number of TPs per SFU's operation can be different. Thus, suitably selecting the value for this parameter allows trading-off between the length of the test program and the number of TpB executed per instruction.

It is important to note that for the self-test routines development, some SFU operations could include TPs that

require executing more threads than the number of TpB for the SBST kernel. Thus, threads of operations are executed several times as a consecutive procedure of reading data from memory followed by the same SFU instruction, so executing a fixed number of TpB.

From the hardware side, a TpB is made up of Warps with size Wz, with each thread occupying one *lane*. Each Warp runs sequentially in *Parallel Groups of Warp Lanes* (PgWL). The SFU processor dispatcher mechanism symmetrically distributes each PgWL to each SFU core to guarantee the same number of Threads-per-SFU (TpS) processed by each execution module, applying the same operation on them. Therefore, the data for the TP operands and the SpT, stored in memory, must be properly organized to test all '*SFUs per SM'* (SpSM). In that case, each TP operand value for the same SFU function must be replicated (within one PgWL) as many times as the number of available SpSM, to ensure the injection of all TPs into each SFU core. This data replication can be performed using two techniques: *i)* data operands replicated directly from memory or *ii)* replication managed through the pointers of each tread.idx:

a. *Replication allocated in memory:* In memory, the TPs' operand values are replicated and allocated consecutively. In this way, after reading data from memory, each PgWL will have the data correctly organized to apply the same TPs to each SFU without adding additional instructions in the program.
b. *Replication from tread pointer index:* This approach only stores in memory the data operands of the TPs for one SFU. Then, the replication process is performed by efficiently managing the address pointers for each thread within each PgWL, so accessing to same data operands stored in memory and replicating the data for each SFU in the reading process. This technique requires SpSM times less memory than the in-memory replication technique. Also, the performance can be affected because the reading process requieres to access several times in memory for the same operand data for each PgWL.

IV. Experimental results

For the experimental campaign to validate the proposed SBST approach, the FlexGripPlus model was configured with 1 SM, 8 SP-cores, and 2 SFUs per SM. All fault simulation experiments were performed on a workstation with two AMD EPYC 7301 16-core processors running at 2.2GHz and equipped with 127 GB of RAM memory.

The SFU module at gate-level is composed of 23,380 combinational cells, 516 sequential cells, 519 In/out pins and accounts for 180,540 stuck-at faults. The results were obtained using a commercial synthesis tool and the Nangate 15nm OCL technology library [15].

The evaluation and validation were performed in two steps: (1) a RT-level simulation of the GPU running the SBST program to extract the reference values in the input and output ports of the SFU, and (2) the gate-level fault simulation.

For the experiments, we developed a parser tool in *Python* to automatically generate the self-test routines. This tool takes as input the generated TPs and the Instruction Set Architecture of the GPU (SASS)[16]. The format of the assembly instructions is used to match the input TPs and produce the self-test routines. Moreover, the parser tool produces a Test Program, the configuration parameters for the GPU and the data memory content to be used during the execution of the test program. The total computational cost required to build a SBST kernel since the TP generation to the fault simulation is about 8 hours.

TABLE I. COMPARISON OF DIFFERENT PROGRAMS IN TERMS OF FAULT COVERAGE, MEMORY AND EXECUTION TIME

		Data Memory size (B)	Test Duration (CCs)	Number of Instructions	FC (%)
Benchmark	SIN	4,096	40,767	11	34.41
	COS	4,096	40,767	11	34.02
	RSQ	4,096	40,767	10	37.76
	LG2	4,096	40,767	10	30.77
	EX2	4,096	40,767	11	28.99
	RCP	4,096	40,767	10	33.06
	NN	8,192	87,195	23	40.88
SBST Random method based on [7]	RND_30K	245,760	1,546,404	309	(M 82.55), (SD 0.56) **
	RND_60K	491,520	3,074,844	609	(M 77.67), (SD 0.07) **
	RND_90K	737,280	4,603,284	909	(M 89.11), (SD 0.42) **
	RND_120K	983,040	6,131,724	1,209	(M 88.26), (SD 0.36) **
SBST proposed in this work	IMM	0	1,200,034	16,856	90.75
	MEM_MR	21,312	212,914	117	
	MEM_CR	10,944	216,764	125	

** M: Mean; SD: Standard Deviation

Three different groups of program kernels were used in the fault campaigns: *i)* some representative benchmarks using the SFU, *ii)* a set of programs using pseudorandom data, based on the method proposed in [7], and *iii)* the SBST programs developed resorting to the proposed method. The comparison between the three groups allows us to better analyze the fault coverage, the test duration, the size and the memory footprint of each test program.

The benchmarks correspond to GPU programs that perform SFU's operations. The first 6 benchmarks (SIN, COS, RSQ, LG2, EX2, RCP) are embarrassingly parallel programs executing each SFU operation on an array of 1,024 elements. The last benchmark (NN) is extracted from the Rodinia benchmark suite [17] and implements a parallel version of the Nearest Neighbor algorithm, commonly used in critical applications, including pattern recognition and computer vision.

The second group is composed of SBST programs using pseudorandom test patterns following the approach presented in [7]. For the purpose of this work, we developed 4 different SBST programs. Each test program applies a different number of random values: RND_30K (30,000 random TPs), RND_60K (60,000 TPs), RND_90K (90,000 TPs) and RND_120K (120,000 TPs). Each test program was generated and executed five times to evaluate the variability in the achieved fault coverage.

Finally, the last group of test programs was generated using the proposed approach. A set of 1,815 test patterns were initially generated for the SFUs using the proposed method, leading to the development of three SBST programs (IMM, MEM_MR, and MEM_CR). The IMM program exploits immediate addressing instructions. The corresponding program size amounts to 16,856 instructions, with a relatively high total test duration time. Although IMM has the largest program size, its duration time is shorter than pseudorandom-based SBSTs. The memory-based test programs MEM_MR (input patterns

replicated in global memory), and MEM_CR (input patterns replicated by code) are SBST programs that have a similar structure. However, MEM_CR uses half of the data memory size than MEM_MR, but involves a performance penalty of around 3,900 clock cycles (CCs).

Table I presents the experimental results in terms of fault coverage, Test duration in clock cycles, program size, and data memory size. As it can be observed, the benchmark programs achieve a moderate Fault Coverage (FC) (28,99% to 40.88%) and show limited use of resources and duration time compared to the test programs. Regarding the pseudorandom SBST programs, they provide a higher FC (from 77% to 89%) but require a relevant size of memory and a longer duration time than the other test programs. The data about the FC of the SBST random-based programs also include the mean (M), and Standard Deviation (SD) computed with the five versions of each program. The highest FC reached by this method was 89.11% (using the RND_90K test-program) with an SD of 0.42%. It is important to remark that according to our results, pseudorandom test programs with a higher number of random TPs do not necessarily achieve a higher FC.

Regarding the results of the SBST programs implemented with the proposed approach, all provide a higher FC (90.75%). A considerable percentage of the undetected faults (almost 3.44% of the total) are untestable due to 130 invalid TPs, mainly caused by incompatible instructions in the ISA of the GPU and unfeasible matching between the TPs and the op-codes of the instructions (several binary combinations in the selector input of the SFU processor cannot be controlled). Marking these faults as untestable we computed a Testable Fault Coverage (TFC) equal to 94.48%, which also includes a 0.29% of untestable faults at the combinational level, according to the ATPG results.

The remaining 5.52% of not detected faults (NDs) are faults aborted by the ATPG tool due to the complexity of the SFU core architecture, which has internal restrictions on the input ports of the FAT component. Therefore, it is challenging to find at least one TP to detect any of the faults in that component. Analyzing the SFU architecture presented in Fig. 2, the inputs of the FAT module have controllability restrictions mainly caused by the limited number of constant coefficients stored in the LUTs. In fact, they only store 1,280 values out of a total number of 2^{53} possible values for C0, C1 and C2. A similar situation happens for the X_2^2 input of the FAT module. This input depends on a squaring unit, which never produces all possible combinations.

V. Conclusions

In this paper, we propose a functional test methodology to develop self-test programs targeting the Special Function Units (SFUs) in a GPU. The proposed solution offers a TP generation strategy that allows finding an adequate set of TPs for testing the SFUs, keeping in mind the constraints coming from the SFU architecture, which prevent the generation of all TPs during the functional behavior. Despite these limitations, the generated test programs allow to reach 90.75% of stuck-at FC, corresponding to 94.48% of stuck-at TFC.

We compared the results of our method first with those produced by some application programs and then with those of pseudorandom test patterns, as proposed in [7]. In both cases, the results have been quite far from those provided by our method in terms of achieved FC, required memory, and test duration. The test program generated resorting to our method perfectly fits the requirements for on-line testing of GPUs used in safety-critical applications.

Further work is ongoing to further increase the effectiveness of the method and to extend it to other fault models.

References

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

[2] J. E. R. Condia et al, “Untestable faults identification in GPGPUs for safety-critical applications”, 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS), 2019.

[3] M. Psarakis et al. “Microprocessor Software-Based Self-Testing”, IEEE Design & Test of Computers, 2010 , Volume 27, Issue 3.

[4] B. Du et al, "About the functional test of the GPGPU scheduler," IEEE 24th International On-Line Testing Symposium (IOLTS), 2018.

[5] S. Di Carlo et al, "An On-Line Testing Technique for the Scheduler Memory of a GPGPU," IEEE Access, vol. 8, 2020.

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

[7] S. Di Carlo et al., "A software-based self test of CUDA Fermi GPUs," 18th IEEE European Test Symposium (ETS), 2013.

[8] S. Di Carlo et al, "Fault mitigation strategies for CUDA GPUs," IEEE International Test Conference (ITC), 2013.

[9] S. Guramurthy et al., “Automated mapping of pre-computed module-level test sequences to processor instructions”, IEEE International Conference on Test, 2005.

[10] J. E. R. Condia et al, "FlexGripPlus: An improved GPGPU model to support reliability analysis," Microelectronics Reliability, vol. 109, 2020.

[11] J. E. R. Condia, J. D. Guerrero-Balaguera, C. F. Moreno-Manrique and M. Sonza Reorda, “Design and Verification of an open-source SFU model for GPGPUs”, 17th Biennial Baltic Electronics Conference (BEC), 2020

[12] E. Lindholm et al, "NVIDIA Tesla: A Unified Graphics and Computing Architecture," IEEE Micro, vol. 28, no. 2, pp. 39-55, March-April 2008.

[13] K. Andryc et al, "FlexGrip: A soft GPGPU for FPGAs," 2013 International Conference on Field-Programmable Technology (FPT), 2013.

[14] J. Pineiro et al, "High-speed function approximation using a minimax quadratic interpolator," *IEEE Transactions on Computers*, vol. 54, no. 3, March 2005, doi: 10.1109/TC.2005.52.

[15] Mayler Martins, et al.. Open Cell Library in 15nm FreePDK Technology. In Proceedings of the 2015 Symposium on International Symposium on Physical Design (ISPD '15). Association for Computing Machinery, New York, NY, USA, 171–178. DOI:https://doi.org/10.1145/2717764.2717783

[16] J.E. R. Condia et al, “Programmers manual flexgripplus sass sm 1.0,” pp. 1–67, May 2020. [Online]. Available: https://doi.org/10.5281/zenodo.3819313

[17] S. Che *et al.*, "Rodinia: A benchmark suite for heterogeneous computing," 2009 IEEE International Symposium on Workload Characterization (IISWC), 2009.