Using STLs for Effective In-field Test of GPUs

Using STLs for Effective In-field Test of GPUs

Josie E. Rodriguez Condia[1], Felipe Augusto da Silva[2 3], Ahmet Çağrı Bağbaba[2 4], Juan-David Guerrero-Balaguera[1] Said Hamdioui [3], Christian Sauer[2], and Matteo Sonza Reorda [1]

[1]Politecnico di Torino, Turin, Italy
[2]Cadence Design Systems, Munich, Germany
[3]Delft University of Technology, Delft, The Netherlands
[4]Tallinn University of Technology, Tallinn, Estonia

Abstract—Reliability and functional safety are of primary importance for Graphics Processing Units (GPUs) widely used in safety-critical fields such as automotive and robotics. Moreover, cutting-edge technologies used in modern GPUs make these even more susceptible to in-field permanent faults due to aging and degradation; hence, the rising need for effective in-field test solutions to ensure the deployment of appropriate countermeasures. This work demonstrates that Self-Test Libraries (STLs) can be written and effectively applied for the in-field detection of permanent faults in a GPU. Simulations using an open-source GPU model (FlexGripPlus) confirm the validity and effectiveness of STLs. The results show that a fault coverage of 92.6% can be achieved, allowing at least ASIL B level for functional safety inside a GPU core.

Index Terms— Software-based Self-test (SBST), Graphics Processing Units (GPUs), Functional-safety, Reliability.

I. INTRODUCTION

Modern Graphics Processing Units (GPUs) are manufactured using cutting-edge technologies but are prone to suffer from in-field errors and reliability issues [1]. The flexibility and computational power of GPUs push their adoption in developing Advanced Driver-Assistance Systems (ADAS) and Sensor Fusion solutions in the automotive and autonomous systems domains. However, the premature aging and wear-out features in new transistor technologies promote the rising of permanent faults during the in-field operation. In safety-critical applications, unaffordable failures caused by faults can induce the entire system to fail or even result in catastrophic consequences if no appropriate measures are taken promptly. Hence, the development of countermeasures for the in-field detection of faults is of great importance in GPUs.

Published works, addressing in-field fault detection for GPUs, can be classified into three classes: *1)* Design for Testability (DfT) methods, which are purely hardware-oriented, *2)* hybrid approaches, which combine hardware structures with reconfigurable capabilities at the software level, and *3)* Software-Based Self-Test (SBST) solutions. DfT schemes are widely used for the end-of-production test in current devices. However, they are not always available for in-field operation and may not satisfy time constraints in many applications. Furthermore, hybrid solutions, based on the addition or use of available structures (i.e., performance counters) to extend the fault observability of a module, must be included in the design phases by modifying the hardware-software interface to provide instruction-based control of the included structures. Authors in [2] proposed an In-System-Test architecture based on the combination of DfT schemes and hybrid structures to detect faults and provide diagnosis features during the in-field operation of System-on-Chips (SoCs) and GPUs. However, a massive effort is required to develop and integrate a coordinated ecosystem to design and verify the device. On the other hand, the SBST strategy is a noninvasive and flexible approach to perform functional in-field tests of processor-based systems, which has been widely adopted in processor testing [3]. Nowadays, semiconductor companies and IP providers give SBST support for their safety-critical products (e.g., automotive). In detail, the SBST strategy resorts to specially written Software-Test Libraries (STLs) composed of suitably developed Test Programs (TPs) able to achieve a given structural fault coverage when run by the CPU with limited or null external support. A TP is a suitable sequence of selected instructions applying test patterns to a given unit and propagating fault effects up to some observation points. These are typically developed starting from high-level abstractions of a design (RT-level) and then progressively reaching and refined at lower levels (Gate-level). Moreover, TPs can often be split into small chunks of code fitting in the idle times of an application and thus more easily matching time constraints. In the past, numerous works developed effective STLs for CPUs. However, only a few works used SBST strategies for in-field tests in GPUs. Clearly, some of the techniques used for CPUs can be extended to GPUs as well. Nevertheless, GPUs have some specific features and characteristics (e.g., implicit parallelism, parallel scheduling, and shared memory management), which demand special strategies to test the corresponding hardware modules. In [4], the authors adopted several processor-based techniques into TPs for the execution units, register files, and main memories in a GPU. Nevertheless, observability issues restricted the assessment of the Fault Coverage (FC). Other work [5] addressed the test of control units (scheduling controller). However, the development of customized approaches was required. In conclusion, prior works on in-field tests are unaffordable due to huge complexity and intrusiveness (DfT and hybrid cases) or suffer from generality (SBST case), making them not fully suitable for GPUs. Hence, there is the need of providing a complete solution for in-field test.

This work, for the first time, evaluates the overall effectiveness of employing the SBST strategy for the in-field test of all logic modules of a GPU core. Moreover, this work experimentally quantifies the fault coverage achievable on the logic modules in a GPU core. Finally, it evaluates how suitable STLs can support the failure modes and effects analysis

(FMEA) required in all safety-critical domains. The main contributions of this work can be summarized as:

- A general overview of the characteristics and strategies to develop STLs for GPUs;
- An evaluation (the first publicly available, to the best of our knowledge) of the overall fault coverage obtained on a GPU core with the STL execution;
- A report about the FMEDA process on a GPU core using STLs as the only fault-tolerance mechanism.

This work resorts to the FlexGripPlus model, describing one low-level microarchitecture of NVIDIA, to evaluate and validate the development of STLs for GPUs. The experimental results show that up to 92.6% of the stuck-at faults in the logic blocks of a GPU core can be covered using the STLs we developed. The FMEDA analysis shows that these results enable to qualify the considered modules inside a GPU core via STLs at least for the ASIL B level. Higher levels can be achieved by combining the STLs with other Safety Mechanisms.

II. Architectural Organization of GPUs

A. General overview

This section employs NVIDIA's terminology to describe the architectural organization of a GPU.

GPUs are special-purpose processors organized as arrays of parallel cores (*Streaming Multiprocessors* or *SMs*). Each SM adopts the Single-Instruction Multiple-Data (*SIMD*) paradigm or variations, such as Single-Instruction Multiple-Thread (*SIMT*) by NVIDIA. Internally, each SM comprises several pipeline stages and uses a specific Instruction Set Architecture partly resembling RISC ones with extensions to support parallelism.

A host controller (CPU) submits a parallel program to the GPU for processing. Then, the program is distributed among the available SMs by the schedulers. Internally, the scheduler controllers manage and trace the operation of a group of threads (*warp*), which are operated in parallel on individual execution units (*Scalar/Streaming Processors* or *SPs*). Each SP is composed of an Integer (INT) and a Floating-point core (FP). Moreover, the SM includes other hardware accelerators (SFUs).

Each SM has access to several levels of the memory hierarchy (Register File, Shared, Local, Constant, and Main Memory). The Register File and the Shared Memory are organized in banks for parallel access and are store the individual and shared operands and results for each thread, respectively. Both resources and the first levels of cache are located inside the SM. The second-level caches, the constant, and the main memories are located outside as a shared resource among the SMs.

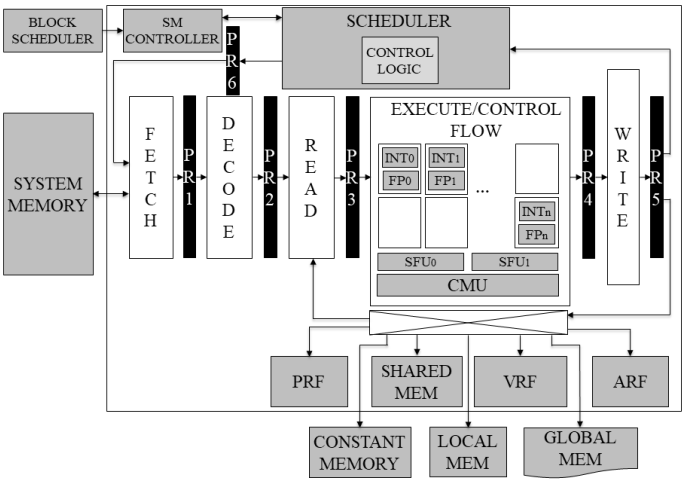

Fig. 1. A general scheme of an SM in FlexGripPlus

B. The FlexGripPlus model

FlexGripPlus is an open-source soft-GPU model based on the NVIDIA G80 microarchitecture and fully described in VHDL [6]. FlexGripPlus is compatible with the CUDA programming environment (SM 1.0) and is based on a set of SMs supporting up to 52 assembly (SASS) instructions.

Each SM is divided into five pipeline stages (*Fetch*, *Decode*, *Read*, *Execute/Control-Flow*, and *Write*), as shown in Fig. 1. The number of SPs in the *Execute* stage is configurable among 8, 16, or 32. Moreover, pipeline registers ('PRx') are located between the pipeline's stages. Each SM also includes three register files (*Vector Register File* 'VRF'), *Address Register File* 'ARF'), and *Predicate Register File* 'PRF'), devoted to storing operands, addresses, and predicate flags of each thread, respectively.

Each SM includes one scheduler controller and a Divergence Management Unit (DMU) for intra-warp divergence control and execution.

In general, the FlexGripPlus model holds the same basic functional modules of a commercial GPU, including scheduler controllers, parallel execution units, file registers, and pipeline stages. Nevertheless, the current memory hierarchy in FlexGripPlus differs from the included in commercial devices by missing the cache memories.

Despite the few structural limitations, the FlexGripPlus model includes a low-level detailed microarchitectural description of an NVIDIA GPU and is employed as a tool to evaluate the effectiveness of STLs for GPUs developed using the SBST strategy.

III. SBST strategies for GPUs

STLs developed with the SBST strategy can be deployed as complementary mechanisms to monitor the status of a GPU during its operative life and contribute to identifying possible fault effects. In fact, the main advantage of STLs is the ability to detect faults with zero hardware costs. Moreover, STLs test a device at the operational speed and normal conditions, thus also addressing delay faults and avoiding overtesting.

In the functional-safety domain, the identification, and management of faults in a device are mandatory. Some faults can be classified as *safe*, when they are proven not being able to produce any failure in the considered operational scenario. Safe faults are not considered when computing the achieved FC.

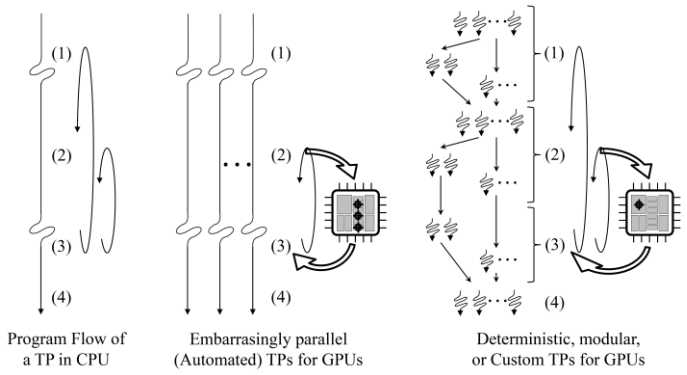

Fig. 2. A general scheme of the execution flow of TPs for CPUs and GPUs

In this domain, STLs can be used as safety mechanisms and increase reliability by guaranteeing the in-field detection of a sufficient percentage of faults, thus matching the requirements of the functional safety standards, possibly in combination with other mechanisms (e.g., ECC for memories, and watchdogs). STLs are widely used for CPUs but they can be adopted for accelerators, such as GPUs, which demand periodic testing solutions when used in safety-critical applications. In this case, we must consider two main features: *i)* most in-field faults in GPUs can only be observed by looking at results they produce in memory (as main observation point), and *ii)* the development of TPs requires architectural details from a targeted unit. In general, any TP is mainly executed following four steps: *(1)* Initialization, *(2)* test pattern's injection, *(3)* fault effect's propagation to any observation point, and *(4)* identification, see Figure 2. In the execution of a TP, several loops can apply different test patterns or propagate their effects. However, TPs for GPUs must face the addressing of each module exploiting the implicit parallelism and operational constraints (e.g., divergence and thread-synchronization). For this purpose, these TPs must exploit three main characteristics from the parallel operation of GPUs:

- Instruction Parallelism;
- Distributed scheduling;
- Management of functional units and memory resources.

The following subsections summarize some specific strategies and algorithms used in the development of TPs for STLs targeting GPUs. It should be noted that each GPU module may require a combination of different approaches. Fortunately, one TP may target the test of several modules in parallel.

A. Extending functional test techniques from CPUs to GPUs

Two approaches originally developed for CPUs can be adopted and extended to the GPU domain: automated and deterministic [3, 7].

On the one hand, the automated approaches comprise pseudorandom-based and ATPG-based methods. The first method focuses on TPs based on a group of instructions randomly selected in combination with pseudo-random operand values. This method can exploit evolutionary algorithms to select the most suitable instructions and operands for a TP. The second method resort to Automatic Test Pattern Generation (ATPG) tools to analyze and extract test patterns from a hardware module. Then, these patterns are translated into equivalent instructions, so composing one or more TPs. However, it is possible that some test patterns cannot be translated and must be ignored (possibly resulting in safe faults). In both cases, several iterations are used in the development of each TP to improve its correct operation and reduce unnecessary overhead costs for the in-field operation.

In any case, TPs using either automated or deterministic approaches must include three strategies: *i)* Parallel Pattern Management (PPM), *ii) Signatures per Thread* (or SpT) [9], and *iii)* Parallel Injection.

The first strategy (PPM) organizes and aligns similar test patterns and expected results as consecutive memory operands, so optimizing the performance in memory management and exploiting possible execution loops. Then, each thread in the TP can address individual or shared test patterns from memory.

The SpT mechanism is based on the computation, within each thread in a TP, of a signature providing fine-grain fault-observability out of the values produced by the target module during its operation, thus propagating fault effects as errors on the computing signature and allowing fault detection. Each SpT is described and computed in software by mimicking a Multiple-Input Shift Register or a counter, which reduces the number of instructions per TP while providing extended observability. In the end, each SpT is stored in memory. The GPU itself (or the host) checks for the presence of faults by comparing a produced signature with the expected one, which is pre-calculated by the TP itself (in the development and verification phases) with minimal performance overhead (<5%) and finally stored in specific memory regions available during the operation of the TPs. Those pre-calculated golden signatures avoid latencies at the in-field operation of TPs.

The Parallel Injection techniques takes advantage of thread parallelism in *warps* or *blocks* to excite a module with different test patterns (one per thread), thus exploiting parallelism to increase the operational performance of a TP, which is effective in either individual units or regular structures.

Pseudorandom and ATPG-based approaches are effective in regular structures of a GPU, such as the functional units and the register file, since these structures are addressed (and tested) in parallel. Moreover, the static organization and the understanding of distribution policies in the schedulers allow the development of embarrassingly parallel TPs, see Figure 2, exploiting the multi-thread parallelism to inject patterns and also reducing the in-field execution of TPs [8]. On the other hand, deterministic approaches exploit the functionality and structure in a module to deploy well-defined algorithms, such as March algorithms for internal memories (e.g., within the controllers) [4].

It must be noted that, when using a deterministic approach, the adaptation of a method may require additional steps (i.e., initialization, and propagation patterns) to face the parallel operational constraints in a GPU, but additional efforts are required to control intra-warp divergences, thread synchronizations, and concurrent loops when injecting test patterns, as depicted in Figure 2.

B. Multi-kernel approach

TPs in this approach utilize a divide-and-conquer strategy to target special modules commonly managing configuration parameters devoted to controlling and configuring the parallel operation in a GPU. These configuration parameters (i.e., memory addresses, number of threads, blocks, grids, and registers per thread) come from the program and configure modules (i.e., the constant memory and the schedulers) for the operative interval of the program.

In this case, multiple TPs (*kernels*) employ the policies of scheduling and the resource's management to target a different set of faults inside a module. More in detail, each TP uses different combinations of configuration parameters, which also serve as indirect test patterns, so activating different regions (and exciting possible faults) in a module. The multi-kernel

approach is effective when testing modules managing parallel parameters, such as the scheduling controllers and the pipeline registers. Further details can be found in [9]. Finally, this approach can be extended to other modules with similar fault activation and propagation restrictions (i.e., global schedulers outside the SM).

C. Modular kernel approach

This approach exploits a top-down strategy to develop modular routines to build TPs for complex units in a GPU. The modular description of a routine starts from a high abstraction level and is then customized. In this approach, the most suitable instructions to activate and propagate faults inside the target unit are selected considering two factors: *i)* the parallel observability and controllability features and *ii)* the architectural description and operational constraints of a target unit.

Firstly, the controllability and observability features are determined for a target unit. In this case, suitable instructions (i.e., "Push" and "Pop" in a stack) are used to provide both features as initial conditions in a TP. Then, several routines to inject test patterns while exploiting parallelism are designed (in CUDA or SASS) and linked, considering the operational constraints of the unit. These routines are the basic components of a TP and describe the operation of any thread. The flexibility of the approach allows the development and exploration of several parallel routines providing the same functionality. Moreover, the execution flow in a TP can be adopted according to the selected routines. Finally, the routines are integrated as a single TP and refined for performance or fault coverage.

The modular approach is effectively applied to complex units in a GPU, such as the DMU and the embedded memories [10].

D. Customs approaches

The custom approaches require the manual development of TPs following some specific algorithm which takes into account the architecture of the units, their functional operation, the expected behavior, their restrictions, and the target fault model. These TPs target particular modules in the GPU which do not exist in CPUs (such as the scheduler controllers [5] and the special-purpose memories [4]). In detail, the TPs are based on algorithms causing controlled divergence, the combination of sequences of embarrassingly parallel and serial-thread executions on a set of threads to excite and propagate fault effects.

This approach requires a deep knowledge of the GPU's low-level micro-architectural details, their parallel operations, and the use of parallelism, distributed scheduling, and available hardware resources to provide specific test solutions per module.

IV. SET UP AND PRELIMINARY GPU ANALYSIS

The evaluation of the STLs (developed using all techniques described in the previous section) resorted to a commercial fault simulation environment targeting the units in the FlexGripPlus model. This framework uses the RT-level description of the GPU and evaluates each TP by injecting stuck-at faults (SAFs) in the logic every logic module.

In the experiments, we targeted the evaluation of all logic modules and embedded memories in the GPU core. One fault is detected when at least one mismatch is found after comparing results from a golden execution and a faulty one. The flexibility of the tool allows the selection of the memory buses and the output control signals as main in-field observation points of faults. It is worth noting that the main memory and the memory controllers were not targeted, since these are not part of the GPU core.

Three preliminary architectural analyses identified safe faults in the GPU. The first analysis is based on the architectural propagation analysis, which consists of evaluating the propagation paths of each fault in the design up to the observability points. In addition, the fault activation analysis evaluates the inputs of the GPU and identifies those faults that cannot be excited. Finally, a barrier analysis provides the analysis of the structural and operational effects of removing modules in the GPU [11].

Table 1 reports the number of identified safe faults in the FlexGripPlus SM. Interestingly, the fault propagation analysis effectively identified most of the safe faults per module (>90%). The other two methods effectively identified faults in the GPU's special structures, such as inside the scheduler. A post-checking process was required to determine any detectable and dangerous fault (i.e., faults in locations which remain temporarily fixed by the effect of the kernel or host configuration, but in other conditions, these may cause misbehaviors), so removing them from the list of safe faults.

TABLE 1. UNTESTABLE FAULT IDENTIFICATION

Module	Faults	Safe faults (%)	Remaining faults
SMP controller	2,508	20.26	2,000
Warp Unit	18,804	23.44	14,396
Pipeline Fetch	737	17.77	606
Pipeline Decode	1,514	21.53	1,188
Pipeline Execute	142,124	23.88	108,185
Address Register File	32,800	0.10	32,768
Vector Register File	131,750	0.51	131,072
Predicate Register File	33,038	0.82	32,768
Divergence Stack Memory	4,568	7.53	4,224
Overall GPU's SM core	367,843	11.05	327,207

V. STL EFFECTIVENESS EVALUATION

In the experiments, FlexGripPlus was configured with one SM and 32 SPs. A set of 18 TPs were implemented using the methods outlined in Section III. Each TP is developed according to the unit's features.

Three automatic TPs targeted the functional units and the decode unit by exploiting instructions that excite as many patterns (operands) as threads on them. Moreover, three deterministic and modular TPs targeted the embedded memories, using the operational features (writing and reading methods) to excite the units. Nine TPs used the multi-kernel approach targeting programmable pipeline registers. Finally, three custom TPs focus on exciting controllers and dispatchers in the GPU.

In the end, 15 fault injection campaigns were performed on the complete GPU model, after generating the full list of SAFs, safe faults were first removed. Moreover, in the fault campaigns, the total number of faults (327,207) was reduced by injecting faults only in one module among the regular modules in the GPU (i.e., one SP and the associated register file per core, instead of the 32 execution units). As a result, in each fault simulation campaign, we injected 141,140 SAFs.

Since the current version of FlexGripPlus does not include accurate descriptions of the caches, the memory controllers were not addressed.

TABLE 2. FAULT COVERAGE RESULTS PER MODULE.

Module	Detected faults	FC (%)
SMP controller	1,156	57.8
Warp Unit	8,416	58.5
Pipeline Fetch	538	88.8
Pipeline Decode	837	70.4
Pipeline Execute	91,429	84.5
Address Register File	32,768	100.0
Vector Register File	131,072	100.0
Predicate Register File	32,768	100.0
Divergence Stack Memory	4,139	98.0
Overall GPU's SM core	303,295	92.6

Table 2 reports the obtained fault coverage (FC) figures. As observed, the developed STLs mainly focused on the largest modules in the GPU's core architecture, such as the execution units, internal logic, and embedded memories, which account for more than 90% of faults in a SM. Although some TPs provide moderate fault detection in some modules of the GPU (e.g., controllers), the small size of these structures does not significantly affect the overall FC in the GPU core. Further efforts could be made to develop suitable TPs specifically addressing these modules.

Previous results demonstrate that STLs can be effectively developed and provide a high FC. Although the obtained results were focused on one GPU core, the implemented TPs are scalable and results can be extended to multi-SM GPUs. Furthermore, the development of STLs can be applied to other GPU architectures.

VI. FUNCTIONAL-SAFETY EVALUATION

The calculation of the Fault Coverage is an indication of the design safety based on the efficiency of a given Safety Mechanism (SMech). However, it is not sufficient to assure compliance with Functional Safety standards, like ISO26262; for such a purpose, we need to determine the reduction in the probability of system failures, also known as the Failure in Time (FIT) rate. The Single Point Faults Metric (SPFM), which represents permanent faults' potential to violate safety-related functionalities, is defined by ISO26262 as evidence of Safety Integrity [12]. The SPFM considers the total FIT rate (λ), and the contribution of the fault classes:

- Single-Point Faults (λSPF): not covered by SMechs
- Residual faults (λR): undetected by SMechs.

The SPFM can be calculated according to the equation:

$$SPFM = 1 - \frac{\sum(\lambda \mathrm{SPF} + \lambda \mathrm{R})}{\sum \lambda} \qquad (1)$$

The primary methodology for determining the Safety Metrics parameters is the Failure Modes Effects and Diagnostic Analysis (FMEDA), which correlates IC components (*Gates*, *Flip-flops*, and *Memory cells*) to Failure Modes (FMs). Then, by computing the λ of individual IC components, the FC, and the Safe faults, we can determine the total λ of each FM.

First, the FMs are defined and the design components mapped. For FlexGripPlus, we considered 28 subparts (components inside the GPU core, including local controllers, functional units, embedded memories, and registers). Each subpart was analyzed to determine function-specific FMs. After mapping each FM to the appropriate design component(s), we evaluate the percentage of Safe faults and the FC. The FlexGripPlus' FMEDA comprises 92 Failure Modes mapped to 2,751,088 Gates, 1,507,085 Flops, and 784,224 Memory cells.

The analysis of FlexGripPlus, considering the 15nm FinFET-based Open Cell Library, resulted in a total λ of 10.08 FIT (based on IEC 62380 Electronic Reliability Prediction Standard), which defines a base FIT Rate for the components of a given tape-out technology. In this case, the unit's base FIT considers digital (NAND2 gate's area) and memory (cell's area) components. Then, we multiple the number of gates and cells, mapped to each FM, by the digital and memory FITs, respectively; from these, the implemented safety strategies provides the following results:

- Detected by the STL: 9.17 FIT
- Undetected (λR): 0.57 FIT
- Safe faults (λS): 0.33 FIT.

Finally, reducing λR by increasing λS and FC, directly impacts the SPFM. The proposed Safety technique based on only STLs for FlexGripPlus resulted in an SPFM of 94.27%, allowing ASIL B assessment without hardware modifications to the logic units of an SM and without any other SMech.

VII. CONCLUSIONS

This work is the first to provide a quantitative evaluation of the effectiveness of STLs for the in-field testing of GPU cores. The reported results showed that a stuck-at fault coverage of more than 92% could be obtained on the logic modules and embedded memories. The functional-safety results (SPFM of 94.27%) show the effectiveness of STLs as a safety mechanism for SMs in GPUs.

The results allow us to state that the SBST strategy can be used as an effective solution, possibly combined with other strategies, to guarantee the reliability and functional safety of GPU-based applications for safety-critical domains.

VIII. ACKNOWLEDGMENTS

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

REFERENCES

[1] D. Tiwari, et al, "Reliability lessons learned from GPU experience with the Titan supercomputer at Oak Ridge leadership computing facility," in International Conference for High Performance Computing, Networking, Storage and Analysis, 2015, pp. 1-12.

[2] P. K. D. Jagannadha, et al., "Special Session: In-System-Test (IST) Architecture for NVIDIA Drive-AGX Platforms," in IEEE 37th VLSI Test Symposium (VTS), 2019, pp. 1-8.

[3] M. Psarakis, et al., "Microprocessor software-based self-testing," IEEE Design & Test of Computers, vol. 27, pp. 4-19, 2010.

[4] S. Di Carlo, et al., "A software-based self test of CUDA Fermi GPUs," in 18th IEEE European Test Symposium (ETS), 2013, pp. 1-6.

[5] S. Di Carlo, et al., "An On-Line Testing Technique for the Scheduler Memory of a GPGPU," IEEE Access, vol. 8, pp. 16893-16912, 2020.

[6] J. E. R. Condia, et al., "FlexGripPlus: An improved GPGPU model to support reliability analysis," Microelectronics Reliability, vol. 109, p. 113660, 2020.

[7] P. Bernardi, et al., "Development Flow for On-Line Core Self-Test of Automotive Microcontrollers," IEEE Transactions on Computers, vol. 65, pp. 744-754, 2016.

[8] J. D. Guerrero-Balaguera, et al., "On the Functional Test of Special Function Units in GPUs," in 24th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS), 2021, pp. 81-86.

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

[10] J. E. R. Condia and M. Sonza Reorda, "Modular Functional Testing: Targeting the Small Embedded Memories in GPUs," Chapter 10, VLSI-SoC: Design trends series, ISBN: 978-3-030-81640-7, 2021.

[11] F. A. d. Silva, et al., "Determined-Safe Faults Identification: A step towards ISO26262 hardware compliant designs," in 2020 IEEE European Test Symposium (ETS), 2020, pp. 1-6.
[12] Y. Chang, et al., "Assessing automotive functional safety microprocessor with ISO 26262 hardware requirements," in Technical Papers of 2014 International Symposium on VLSI Design, Automation and Test, 2014, pp. 1-4.

Josie E. Rodriguez Condia received the Ph.D. degree in Computer Engineering from Politecnico di Torino, Italy in 2021, and the M.Sc. degree in electronic from Universidad Pedagógica y Tecnológica de Colombia (UPTC), Colombia in 2017. His research interests include functional testing, parallel architectures, and embedded system design.

Felipe Augusto da Silva is a Ph.D. Candidate on Functional Safety in Cadence and at Delft University of Technology. He holds a Computer Engineering degree from Pontifical Catholic University of Rio Grande do Sul (PUCRS) and a M.Sc. degree in Electrical and Electronics Engineering from Federal University of Santa Catarina (UFSC), Brazil. During his career as Safety-Critical Embedded Systems Engineer, he contributed to Functional Safety projects for the Automotive, Aerospace, and Defense industries.

Ahmet Cagri Bagbaba received the Ph.D degree in Computer and Systems Engineering from Tallinn University of Technology, Estonia in 2022, and the M.Sc. degrees in electronics and telecommunication engineering from the Istanbul Technical University, Turkey. He is currently working at Cadence, Germany. His research interests include Hardware Functional Safety Verification in the context of ISO26262, digital and embedded system design.

Juan-David Guerrero-Balaguera is pursuing a Ph.D. in the Department of Control and Computer Engineering of Politecnico di Torino, Italy. He received the M.Sc. degree in electronic from Universidad Pedagógica y Tecnológica de Colombia (UPTC). His research interests include functional testing, Artificial Intelligence, Parallel architectures.

Said Hamdioui is chair professor and head of the Department of Quantum and Computer Engineering at Delft University of Technology, The Netherlands. His research interests include hardware dependability and emerging computing paradigms. He is a Senior Member of the IEEE and serves on the editorial board of IEEE Design & Test.

Christian Sauer heads the European System Design Enablement team for Cadence in Munich. He works in customer-specific projects developing tailored solutions for cutting edge SoCs and systems across automotive and 5G domains. His research interests include the development of application-specific multi-processor platforms, tools and methodologies for their applications.

Matteo Sonza Reorda is a full professor in the Department of Control and Computer Engineering of Politecnico di Torino, Italy. He received the PhD degree in computer engineering in 1990 from the same institution. His research interests include design and test of reliable electronic circuits and systems. He is a Fellow of the IEEE.