Using Formal Methods to Support the Development of STLs for GPUs

Using Formal Methods to Support the Development of STLs for GPUs

Nikolaos I. Deligiannis†, Tobias Faller*, Josie E. Rodriguez Condia†, Riccardo Cantoro†,
Bernd Becker*, Matteo Sonza Reorda†
† Politecnico di Torino, Department of Control and Computer Engineering (DAUIN) - Turin, Italy
* University of Freiburg, Department of Computer Science - Freiburg, Germany

***Abstract*—Graphics Processing Units (GPUs) boost the development of high-performance safety-critical applications. The reliability of such systems is of utmost importance since faults affecting the hardware may occur at any time during the systems' operational life. Thus, methods to effectively test these devices during their in-field operation are necessary. One popular solution relies on Software Test Libraries (STLs), which recently have been started being used for GPUs as well, since they are effective in terms of fault detection capabilities, intrusiveness, flexibility, and test duration. A drawback of the STL approach for GPUs is the extensive effort used to develop effective test routines for complex structures, e.g., controllers, due to the complicated constraints stemming from the ISA, the available compilation flows and parallelism constraints. We propose a novel technique based on formal methods to support the generation of stimuli and enhance the quality of pre-existing STLs for GPUs. To validate the proposed method, we resort to an open-source GPU model (*FlexGripPlus*). Experimental results show that the method can effectively generate complementary code fragments to be added to existing STLs and increase their fault coverage. In the case of the GPU's decoding unit, the stuck-at fault coverage was increased by nearly 10%.**

***Index Terms*—Formal Methods, Graphics Processing Units (GPUs), Software Test Libraries, Test Quality**

I. Introduction

Nowadays, Graphics Processing Units (GPUs) represent widely used platforms to implement high-performance applications, using complex and dense algorithms even in the safety-critical domain (i.e., self-driving cars and autonomous machines) [1], [2]. Applications in this domain, such as sensor fusion and Neural Networks (NNs), require major reliability features, as mandated by the regulations and standards in the field (e.g., ISO 26262). Modern GPU devices employ the latest integration transistor technologies, which, from the reliability viewpoint, are more prone to the rising of faults during the operative period of the device (e.g., by aging or wear-out) than mature technologies. Thus, effective periodical in-field mechanisms and techniques are needed to detect and mitigate faults in GPUs devoted to safety-critical applications.

One in-field functional test approach relies on the Software-Based Self-Test (SBST) strategy, successfully employed in processor-based systems [3]. The SBST strategy is a non-intrusive and flexible method to develop test programs (TPs), resorting to the Instruction Set Architecture to apply test patterns, which excite and propagate faults inside a targeted unit in a device. In practice, a Software Test Library (STL) is a collection of TPs. These TPs are mainly designed to target individual units and allow the singular identification of faults in a device. Several works [4], [5] have proposed strategies to develop TPs for GPUs, targeting the functional units, the register files, and the internal controllers. One of the strategies is based on automatic mechanisms by analyzing and exploiting Automatic Test Pattern Generator (ATPG) algorithms to provide reasonable test patterns, which are later translated into equivalent instructions. Similarly, TPs composed of pseudo-random instructions can be developed [6]. Both ATPG-based and pseudo-random strategies are mainly effective on combinational units. Other strategies analyze the structure of a target unit and use custom algorithms to develop affordable test routines [7], [8]. Unfortunately, two main factors restrict the development of effective TPs for GPUs: *i)* the deep knowledge required to understand the structural features of any target unit, and *ii)* the parallelism constraints when addressing a given unit. In all the aforementioned approaches, huge manual efforts and long development times are required when developing specific TPs on particular units of a GPU (e.g., controllers). Moreover, the TP quality, in terms of fault coverage and program size, could be negatively affected by operational constraints and structural restrictions, leading to insufficient fault coverage of functional ATPG-based TPs. Thus, complementary methods are required to improve the quality of the TPs.

In this work, we propose a novel technique based on formal methods to generate effective test patterns and enhance the quality of pre-existing STLs for GPUs. Our approach takes into account the structural constraints of a GPU module and the parallel programming constraints of the device to generate test patterns that can be easily translated into equivalent parallel instructions for the GPU. To the best of our knowledge, this is the first work proposing techniques to enhance the quality (in terms of fault coverage) of STLs by employing formal methods for GPUs. Due to the highly complex nature of GPU designs, several architectural and structural constraints are required in the TP generation, such as the correct specification of the targeted unit's environment, including control signals and protocols. In fact, compared to the test program generation for CPU cores, additional parallel constraints for correctly modeling warps, parallel thread branching, convergences, and scheduling operations have to be considered to generate effective test patterns for GPUs

and enhance the quality of TPs. This additional complexity requires a powerful and dynamic approach that scales with the constraints, which in our technique we achieve with an elaborate constraint specification mechanism.

More specifically, given the gate-level description of a functional unit of the GPU and a list of functional constraints, our goal is to generate functional stimuli that enable the control/sensitization of stuck-at faults. If this is not possible for a certain fault, then this fault is untestable. Otherwise, we generate a functional test pattern, which is further used to check whether it is possible to propagate the fault up to the unit observation/test points i.e., the unit's primary outputs which is experimentally likely. This goal is pursued without resorting to any kind of Design-for-Testability infrastructure, as it is common for in-field test. Moreover, given an STL for a specific functional unit, we can check which faults are functionally undetectable and thus, target them i.e., generate patterns for them (if possible) to increase the STL's overall fault coverage. The proposed method enables the test or functional safety engineer to generate valid instructions that enable the detection even of the most difficult to detect faults, or label them as untestable. Basically, we achieve this by first reducing the pattern generation problem to a bounded model checking (BMC) problem and then by effectively solving it resorting to an appropriate solver [9]. In the experiments, we used an open-source GPU model (*FlexGripPlus*) [10], and we targeted one key control unit (the decoding unit) to evaluate and validate the proposed method. According to the results, the proposed technique can identify up to 10.09% new effective test patterns and increase the fault coverage by up to 9.57% with minimal computational effort (< 2 minutes). Although we employ the FlexGripPlus GPU model, the proposed technique can be adapted to other units and GPU architectures as well.

The remainder of the paper is organized as follows. Section II provides a background of the GPU organization and overviews the formal methods. Section III introduces the proposed approach to analyze and enhance the quality of STLs in GPUs by combining structural features and the expression of the complex GPU constraints via formal methods. Section IV reports the experiments performed to evaluate and validate the proposed method on an open-source GPU core (FlexGripPlus). Finally, Section V draws some conclusions and outlines some possible future works.

II. Background

This section summarizes the organization of a GPU and the main concepts for the formal methods used in this work.

A. GPU organization

GPUs are special-purpose accelerators specially conceived to exploit hardware parallelism and process extensive amounts of data with high throughput. Modern general-purpose GPU architectures are organized as arrays or clusters of execution cores (also known as Shader cores, *Streaming Multiprocessors* or SMs), which execute hybrid parallel assembly instructions (i.e., Shader Assembly by Nvidia) and also address in parallel several memory levels in a complete memory hierarchy (see Figure 1).

In general, each SM is organized as a set of pipeline stages, controlled by one or more schedulers and dispatchers units. In each pipeline stage, the SM executes one parallel instruction, internally divided into the procedures of fetching, decoding, execution, reading from memory, and writing to memory. Moreover, the SMs include several execution units (CUDA cores or *Streaming Processors* or SPs) and other accelerators (i.e., Special Function Units or SFUs). In detail, one parallel program is divided as a set of blocks (Cooperative Thread Arrays, short CTAs) and distributed among the available SMs. The internal controllers submit one instruction from the parallel program for processing, each for a group of threads *(Warps)*. The submitted instruction is initially decoded and then processed in parallel by the available SPs, as one SP per lane. In fact, the same instruction is processed in parallel by several threads using different operands per thread. Then, a new instruction (from the same or another thread group) is submitted and processed.

Modern GPU designs allow the decoding and execution of several instructions in parallel and divide the distribution of the available SPs per SM among the instructions to process, so more than one instruction per thread group can be executed simultaneously [11]. More in detail, the decoding unit plays an important role — as a control unit inside the SMs — in identifying incoming instructions and assigning hardware units and operand sources for the parallel processing among the different parallel threads.

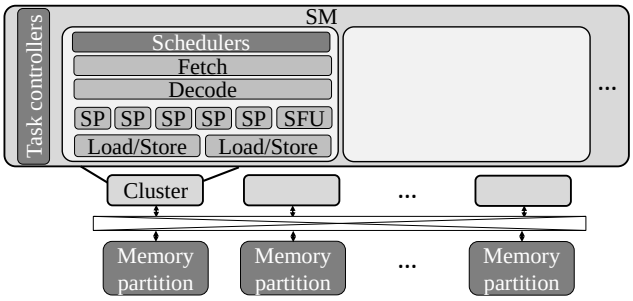

Figure 1. A general scheme of the internal organization of a GPU.

B. Formal Methods & Test Program Generation

It has been long known that an ATPG problem can be reduced to a Boolean satisfiability instance and solved using a SAT solver [12]. However, this approach was not widely implemented in commercial EDA tools, as the so-called structural ATPG approaches generally provided better scalability. More recently, significant improvements in the underlined SAT solvers in conjunction with extended solving capabilities specifically developed and tailored to ATPG led to an increased interest in such techniques [13], [14].

Formal methods have been adopted in the past for generating TPs while targeting processor circuits. In [15], the authors propose a generic methodology based on formal methods for

generating instruction sequences that test structural faults in a processor circuit. While using the OpenRISC 1200 processor as a use case, they elaborate on the formulation of constraints for the expression of Boolean differences while relying on an off-the-shelf Bounded Model Checker to determine the testability of these faults. In [16], the authors elaborate on a Bounded Model Checking (BMC) technique for generating TPs for a group of RISC-V processors destined for IoT applications. In [17], the authors propose a methodology for the generation of TPs for testing sequential control units in functional mode on superscalar processors. Lastly, in the context of stimuli generation for Burn-In testing, in [18], while targeting processor circuits, we propose a methodology able to generate functional stimuli to stress units within the core based on formal methods effectively.

Overall, formal methods have been widely adopted for attacking the problem of TP generation for processor circuits. On the other hand, functional test stimuli generation for GPU circuits is an open problem and challenging task due to the vast majority of operational constraints that must be considered, as mentioned earlier, and up to our knowledge, there are no other works in that area.

III. Formal Methods Perspective

The automatic generation of functional stimuli for a unit belonging to a GPU is an arduous task. To overcome this difficulty, we rely on formal methods as an underlying engine since they empower the test engineer to write expressive and detailed constraint formulations to support the functional correctness of the stimuli generation. Furthermore, via the use of k-induction and Craig interpolation [19], [20], to check for unreachability, we are able to prove the absence of a functional test pattern in case of untestable faults. Without a systematic definition of functional constraints, it is possible to generate test patterns for the targeted unit, but some of those patterns would not be translatable to proper instructions to compose a TP. Furthermore, the identification of untestable faults would not be possible. However, care must be taken because in a strict, over-constrained scenario there is the risk of miss-classification of untestable faults. That is, to identify a certain category of faults as untestable, while in fact they are testable ones.

The generation of valid TPs requires the formulation of complex, module-specific constraints that correctly model the functional environment of the GPU's module and the STL, including control inputs and status signals of the unit, the set of valid operations and operands, similar to the case of a CPU. The complexity of the TP generation is further extended due to the highly parallel architecture with complex scheduling and highly optimized execution units. The proposed approach enables us to specify all the constraints for the GPU's module, without requiring explicit enumeration of all valid states.

The constraint set is implemented via a so-called *Validity Checker Module* (VCM) [21], which is a circuit written in a *Hardware Description Language* (HDL), circumventing a tedious, manual specification of low-level ATPG constraints by using a high-level language. The VCM is synthesized into a gate-level description using the synthesis tool of choice and used by the BMC process later. Both gate-level circuits (the target GPU module, and the VCM) are connected into a single circuit and converted to a CNF by applying a symbolic simulation performing the Tseitin transformation of the circuit's logic formula. By imposing constraints on the VCM's output pins and encoding arbitrary circuits as VCM, the constraints are propagated to the GPU module, excluding non-functional states from the BMC problem.

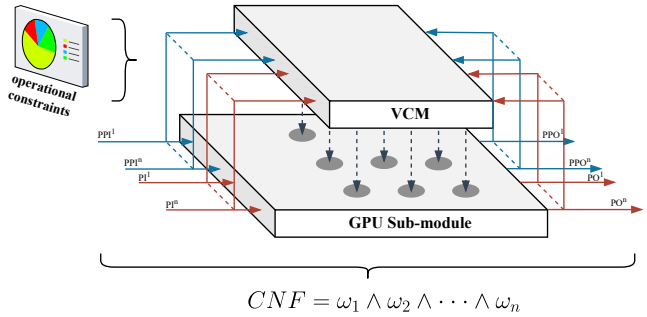

Figure 2. CNF generation steps of the VCM & GPU sub-module

Figure 2 depicts the VCM concept, where the VCM acts as an entity that controls the BMC process and is attached to the GPU module. It is attached to selected inputs, outputs, and internal signals of the GPU module, through which the targeted module's state and behavior is observed. The VCM uses those observations to compute if the GPU module's behavior is valid and signals this as a Boolean signal on one of it's output ports. By adding an assumption to the BMC solver forcing all of the VCM outputs to always signal a valid behavior, only solutions that model the VCM-defined behavior are generated. Overall, via the VCM concept we avoid non-functional states and the occurrence of an unwanted behavior on the targeted module during the BMC process.

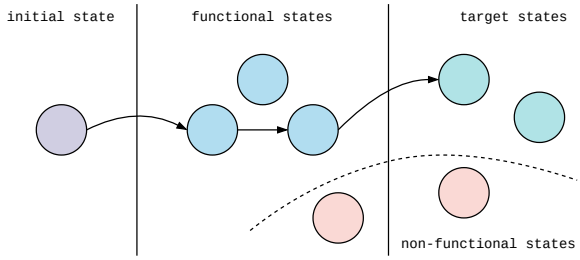

Figure 3. Abstract bounded model checking scheme

Figure 3 shows an abstract view of the bounded model checking problem, where the GPU's module is initialized with a fixed, valid, and well-defined initial state. The GPU transitions from the initial state over possibly multiple functional states that sensitize and propagate the fault, finally reaching the target state, which propagates the fault effect to at least one of the primary outputs of the module. If no path exists that reaches the target, then the BMC solver can

prove, given enough time and computational power, that this fault is untestable, assuming only functional behavior. This proof is either accomplished by finding a fix-point in the set of reachable states after unrolling for some time-frames and proving via Craig interpolation that the target is not reachable, or by showing via k-induction that no initial state exists that reaches the target in k time-frames.

In Figure 3, the non-functional states shown separated by the dotted boundary are excluded by the VCM. Those states are GPU module-specific and are either excluded due to a violation of functional constraints on the module's inputs, an invalid state that does not occur in a functional scenario, or a combination thereof.

Input : A tuple *(G^α, G^β)* where
G^α is the gate-level description of the GPU sub-module
G^β is the gate-level description of the VCM
Output: A tuple of testability verdicts and patterns for every stuck-at fault of G^α

```
1  faults ⟵ ∅; verdicts ⟵ ∅; patterns ⟵ ∅;
2  foreach port p of cell c in G^a do // FAULT LIST GENERATION
3      r_p^sa0 ⟵ GenerateSensitizationReq(p, 1)
4      r_p^sa1 ⟵ GenerateSensitizationReq(p, 0)
5      faults = faults ∪ {r_p^sa0, r_p^sa1}
6  end
7  foreach requirement set r in faults do // BMC
8      foreach stuck-at fault f in r do
9          CNF ⟵ GenerateCNF(G^α, G^β, f)
10         v ⟵ SolveBMC(CNF)
11         switch verdict v do
12             case reachable do
13                 verdicts = verdicts ∪ {(f, controllable)}
14                 patterns = patterns ∪
15                 {(f, ExtractPattern(CNF, v))}
16             end
17             case unreachable do
18                 verdicts = verdicts ∪ {(f, untestable)}
19             end
20         end
21     end
22 end
23 return (verdicts, patterns)
```

Figure 4. BMC-based functional stimuli generation routine

The BMC-based stimuli generation routine is given in pseudo-code format in Figure 4. As input it requires the two gate-level descriptions of the GPU module and the respective VCM that enforces the complex set of functional constraints for the target module. Initially, for every port of each gate-level cell of the targeted unit, we generate a pair of sensitization requirements under the stuck-at-fault model. To elaborate on the testability of the stuck-at zero fault of a port p, the target requirement for the BMC problem is to sensitize p to the logic value of 1. On the other hand, to elaborate on the testability of the stuck-at one fault of p the target requirement would be to sensitize p to the logic value of 0. Lastly, for every testability requirement generated, a BMC problem is created, having as a target the aforementioned requirement. If the solver determines that the target state is reachable, then the respective fault is a controllable one and a test pattern can be extracted from the solution of the CNF formula. On the other hand, if the solver responds with an unreachable status, then this means that the corresponding fault is an uncontrollable (and thus, untestable) one. Note that the BMC solving step of the algorithm can be fully parallelized since the corresponding BMC problems for the testability verdicts of the stuck-at faults are unrelated.

The identified test patterns can be divided in two groups: *(i)* single patterns, and *(ii)* pattern pairs. In the first case, the patterns can directly sensitize the corresponding fault inside the module and possibly propagate the fault towards any primary output. In contrast, the pattern pairs correspond to sequences of two test patterns to initialize, excite, and possibly propagate a targeted stuck at fault.

For example, if for a cell port p, we wish to elaborate on the controllability of the stuck-at fault 0, thus we have to enforce the opposite logic value. If during the initialization phase, p is assigned the logic value of 1, then we need to first move p to the logic value of 0 (first pattern) and then to the logic value of 1 (second pattern). Hence, two (a pair) of test patterns are used to initialize and excite the possible faults on the target site p. Then, the patterns (single or pairs) are translated into instructions to compose TP routines. Finally, a set of fault simulations verify the new test pattern's effectiveness in exciting and propagating faults from a targeted unit in the GPU.

Case Study: The GPU Decoding Unit

Table I
OPERATIONAL CONSTRAINTS OF THE DECODING UNIT IN A GPU

Operational feature	Operational constraint
instruction set	All supported instructions from the SM 1.0 of the G80 architecture
warp processing	Dispatched and executed in increasing order (from 0 to 31)
warp lanes	Dispatched according to scheduler; Increasing sequence (from 0 to 3)
cooperative thread array (CTA) id	Dispatched according to scheduler; Round-robin approach (from 0 to 15)
thread register size	From 0 to 64
thread active state	Active threads in a warp during execution of an instruction; active (1) or inactive (0); at least one active field must be active to execute an instruction
warp instruction program counter	Within the limits of the GPU's system memory
pipeline stall	Status and control line in the SM; when active, the unit stops its execution until this line is released
pipeline done	Completion acknowledge status of a previous operation from all pipeline's stages in the SM; when active, the unit is ready for the next operation

Table I reports the main identified parallel operational constraints for the decoding unit, which are used to generate the VCM hardware module during the analysis. These constraints are determined considering the primary inputs and outputs of the unit and the interaction with the system (i.e., the parallel configuration parameters, which are commonly used for executing instructions). The first group of constraints comprise

the supported instructions from the GPU's ISA. This constraint allows the generation of test patterns later translated into valid instructions. The second group of constraints depends on the feasible and valid configurations for instructions during the execution (i.e., size of registers per thread and the number and distribution of warps, CTAs, and SP-lanes).

Another group of constraints handle the control features in the unit, including the stall and done conditions in the operation of the unit, and the instruction program counter's limit, which is defined according to the system memory. Finally, the thread's status in a warp (*active / inactive*) is listed as an additional constraint for the unit since the status affects the execution of a possible instruction by the GPU.

IV. Experimental results

This section describes the experimental setup and reports the experimental results obtained in evaluating and validating the proposed technique.

A. Experimental setup

In the experiments, we employ the FlexGripPlus GPU model [10] targeting the gate-level description of the decoding unit inside one SM. Moreover, a custom workflow was developed to include the proposed formal method analysis and the evaluation and validation steps for the TPs. The formal analysis, the evaluation, and the verification fault-injection campaigns are performed on a server of 12 Intel Xeon CPUs running at 2.5 GHz and with 256 GB of RAM. The targeted module (the decoding unit) is synthesized using the Nangate 45nm open-cell technology library [22], consisting of 987 combinational and 359 sequential cells that account for 11,610 permanent stuck-at faults.

FreiTest, originated from the path ATPG tool PHAETON [23], is used as an underlying framework to flexibly model the ATPG problem considered in this paper. In particular, we adapted and restricted to deal with the stuck-at-fault model and enabled built-in optimizations. On top of the framework, we built an application that accounts for approximately 1,000 lines of `C++` code for the implementation of the proposed method (reported in Figure 4) and approximately 200 lines of `SystemVerilog` code for specification of the VCM.

For the fault simulation experiments devoted to verifying possible new test patterns, we employ two commercial tools handling the GPU. First, a logic simulator (*ModelSim*) traces the execution of an existing or new TP, resorting to a mixed-level description of the GPU model (RT- + gate-level). In this case, the targeted unit is the only one simulated at gate level. Moreover, in this procedure, a trace report is produced from the complete workload (TP) execution, covering the primary inputs and outputs of a targeted unit. Then, the generated trace report serves as input for a functional simulator (*Z01X*). This tool performs individual fault injection campaigns only on the targeted unit (injecting permanent stuck-at faults) to verify the effectiveness of the instructions in each TP in terms of activation and propagation of fault effects. A fault is identified as detected when at least one of the GPU primary outputs is modified by the effect of the propagation of a fault while executing one of the TPs.

B. Experimental evaluation

We employ one STL previously developed to functionally test the decoding unit of a GPU [24]. This STL is composed of three main TPs (denoted as IMM, MEM, and CNTRL) using immediate operand, memory movement, and control-flow instructions, respectively, to excite permanent faults inside the decoding units and propagate their possible effects. The main features of the TPs are reported in Table II. It is worth noting that all TPs (IMM, MEM, and CNTRL) were designed by employing a pseudo-random approach in combination with the operational constraints of the unit.

Then, we employed the set of previously described constraints to analyze the decoding unit and generate new test patterns, which are finally translated into instructions. Table III reports the results of this process. A commercial sequential ATPG tool was used for comparison purposes. In the analysis, the implemented framework identified 1,172 undetected permanent faults, which the original STLs do not cover. Moreover, the analysis provided a total of 1,809 new test patterns able to excite and propagate faults inside the decoding unit of the GPU. In detail, 535 faults are detected by an individual test pattern, while 637 faults are detected via a sequence of two patterns. The total run-time of the analysis framework was 198 seconds taking advantage of the heavy parallelization of the method to reduce its execution time.

Table II
Main Features of the original STLs for the Decoding Unit

Test Program	Duration (# of ccs)	Size (# of instructions)	FC (%)
IMM	2,229,225	32,736	71.13
MEM	3,186,236	32,581	76.59
CNTRL	710,100	366	71.18
IMM + MEM + CNTRL	6,125,561	65,653	80.15

Table III
Comparison with Commercial ATPG

	Full-Sequential ATPG	BMC-based TPG
# Generated test patterns	86	1,172
# New instructions	86	1,809
% Fully ISA-coherent instructions	100	50.34
% Increase in the STL FC	0	9.57

After translating the test patterns into equivalent instructions from the GPU's ISA, we evaluated 1,172 test routines, including only the new instructions (one or two) and the parallel configuration constraints determined during the formal analysis of the unit (i.e., number of active warps, thread ID, block number, etc.). The experimental results of the evaluation show that out of the newly identified test patterns, about 25% can be directly translated into valid instructions in the GPU's ISA and be executed with minimal restrictions of parallel configuration, so enhancing the test coverage directly

without significant effort in the design of the TPs. Thus, these new instructions can be added to the existing TPs. In contrast, a moderate percentage of new test patterns (around 25.34%) require specific parallel configurations after being translated into instructions (i.e., specific memory locations or the addressing of particular memory resources, such as the shared memory), which means that these instructions cannot be included in previously developed test routines, and ad-hoc test programs must be developed.

On the other hand, a considerable percentage of the newly identified patterns (49.66%) can only be translated into valid instructions conditioned to the activation of unfeasible predicate flags (e.g., an always 'false' execution of a global memory load). These instruction types are valid for the ISA, but are commonly avoided during the compilation procedures, so they cannot be generated by conventional GPU compilers. Hence, since for GPUs it is not possible to generate inline assembly code and embed it in any application code, these faults can never force an application to produce a failure. According to safety standards (e.g., ISO 26262) these faults can thus be labeled as *safe* and removed from the computation of the achieved Fault Coverage. At the end, 16 stuck-at faults in the unit were proven to be uncontrollable, and were labeled as untestable, during the BMC-based TPG process.

Finally, we compute the overall fault coverage as a combination of the TPs in the original STLs, and the new TP including the newly generated instructions. The overall results provide a FC of 89.72% (an increase of 9.57%, when excluding the identified safe faults), which shows the effectiveness of the proposed technique. These results support the idea that formal methods can be used as a supporting and complementary technique to enhance the development of software-based self-test routines for GPUs devoted to the safety-critical domain.

Although the experiments were performed targeting the decoding unit in a GPU, we claim that the same technique can be adapted for other controllers, functional units, and other modules in the GPU architecture.

V. Conclusions

This work introduces a novel technique to enhance the quality of Software Test Libraries for GPUs. The proposed technique exploits formal methods to analyze and identify missing test patterns for a targeted unit and increase the fault coverage of functional test routines for GPUs. The proposed technique combines the functional features of a target unit and the parallel programming restrictions to formulate the constraints for the analysis.

According to the results, the proposed technique effectively identified up to 10.09% (1,172) new effective test patterns for the decoding unit in a GPU, which were later translated into 590 equivalent test routines for the Software Test Library. Correspondingly, the fault coverage was enhanced by up to 9.57% (considering safe faults), showing the effectiveness of the proposed method in enhancing the test quality of programs for GPUs.

In future works, we plan to extend the proposed technique to other units of the GPU architecture and other hardware accelerator architectures.

References

[1] S. Alcaide *et al.*, "Safety-Related Challenges and Opportunities for GPUs in the Automotive Domain," *IEEE Micro*, vol. 38, 2018.

[2] M. Benito *et al.*, "Comparison of GPU Computing Methodologies for Safety-Critical Systems: An Avionics Case Study," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2021.

[3] N. Kranitis *et al.*, "Software-based self-testing of embedded processors," *IEEE Transactions on Computers*, vol. 54, 2005.

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

[5] S. Di Carlo et al., "A software-based self test of CUDA Fermi GPUs," in *European Test Symposium (ETS)*, 2013.

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

[7] B. Du *et al.*, "About the functional test of the GPGPU scheduler," in *International Symposium on On-Line Testing And Robust System Design (IOLTS)*, 2018.

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

[9] S. Kupferschmid *et al.*, "Incremental preprocessing methods for use in BMC," *Formal Methods in System Design*, vol. 39, 2011.

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

[11] NVIDIA, "NVIDIA Tesla V100 GPU Architecture," 2017. White paper.

[12] T. Larrabee, "Test pattern generation using Boolean satisfiability," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 11, 1992.

[13] R. Drechsler and others, "On Acceleration of SAT-based ATPG for Industrial Designs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, 2008.

[14] K. Scheibler *et al.*, "Accurate CEGAR-based ATPG in presence of unknown values for large industrial designs," in *Design, Automation Test in Europe Conference Exhibition (DATE)*, 2016.

[15] S. Gurumurthy *et al.*, "Automatic Generation of Instruction Sequences Targeting Hard-to-Detect Structural Faults in a Processor," in *International Test Conference (ITC)*, 2006.

[16] T. Faller *et al.*, "Towards SAT-Based SBST Generation for RISC-V Cores," in *Latin American Test Symposium (LATS)*, 2021.

[17] Y. Zhang *et al.*, "Software-Based Self-Testing Using Bounded Model Checking for Out-of-Order Superscalar Processors," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, 2020.

[18] N. I. Deligiannis *et al.*, "Effective SAT-based Solutions for Generating Functional Sequences Maximizing the Sustained Switching Activity in a Pipelined Processor," in *Asian Test Symposium (ATS)*, 2021.

[19] T. Wahl, "The k-Induction Principle." https://www.ccs.neu.edu/home/wahl/Publications/k-induction.pdf.

[20] K. McMillan, "Applications of Craig Interpolation to Model Checking," in *International Conference on Application and Theory of Petri Nets and Other Models of Concurrency (ICATPN)*, 2005.

[21] S. Gurumurthy *et al.*, "Automatic Generation of Instructions to Robustly Test Delay Defects in Processors," in *European Test Symposium (ETS)*, 2007.

[22] J. Knudsen, "Nangate 45nm open cell library," 2008.

[23] M. Sauer *et al.*, "PHAETON: A SAT-Based Framework for Timing-Aware Path Sensitization," *IEEE Transactions on Computers*, vol. 65, 2016.

[24] J.-D. Guerrero Balaguera *et al.*, "A Compaction Method for STLs for GPU in-field test," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2022.