Enhancing the Effectiveness of STLs for GPUs via Bounded Model Checking

Enhancing the Effectiveness of STLs for GPUs via Bounded Model Checking

NIKOLAOS DELIGIANNIS, Control and Computer Engineering (DAUIN), Politecnico di Torino, Torino, Italy
TOBIAS FALLER, University of Freiburg, Freiburg im Breisgau, Germany
JOSIE ESTEBAN RODRIGUEZ CONDIA, Control and Computer Engineering (DAUIN), Politecnico di Torino, Torino, Italy
RICCARDO CANTORO, Control and Computer Engineering (DAUIN), Politecnico di Torino, Torino, Italy
BERND BECKER, Albert-Ludwigs-Universitat Freiburg, Freiburg, Germany
MATTEO SONZA REORDA, Control and Computer Engineering (DAUIN), Politecnico di Torino, Torino, Italy

Graphics Processing Units (GPUs) are becoming widespread, even in safety-critical applications. In that case, it is imperative to guarantee that the probability of producing critical failures due to hardware faults is lower than a given threshold. To detect possible permanent hardware faults as soon as they appear during the operational phase (e.g., due to aging), Software Test Libraries (STLs) have gained significant traction as a widely adopted test solution due to their effectiveness in terms of fault detection capabilities, test application time, and flexibility. However, a major drawback of this solution is the lack of automation in the STL generation phase. As a result, high manual labor is required for their generation. This becomes even more arduous in complex architectures that require in-depth knowledge to cover hard-to-test faults. In this article, we introduce a methodology based on Bounded Model Checking to support the generation and improvement of stuck-at-oriented STLs for hard-to-test units in GPUs, showing that we can enhance the test coverage achieved by pre-existing STLs while also identifying a set of functionally untestable faults. To experimentally validate the proposed method's effectiveness, we use the `FlexGripPlus` GPU model to target two hard-to-test units, one medium to low complexity sub-unit and one high complexity sub-unit, as study cases. For both units, we had pre-existing STLs written for the stuck-at model. Resorting to the proposed method, the STLs' test coverage was increased by 9.57% and 2.19%, respectively. In addition, the method also identified a significant number of functionally untestable faults.

Authors' Contact Information: Nikolaos Deligiannis, Control and Computer Engineering (DAUIN), Politecnico di Torino, Torino, Italy; e-mail: nikolaos.deligiannis@polito.it; Tobias Faller, University of Freiburg, Freiburg im Breisgau, Baden-Württemberg, Germany; e-mail: fallert@tf.uni-freiburg.de; Josie Esteban Rodriguez Condia, Control and Computer Engineering (DAUIN), Politecnico di Torino, Torino, Italy; e-mail: josie.rodriguez@polito.it; Riccardo Cantoro, Control and Computer Engineering (DAUIN), Politecnico di Torino, Torino, Italy; e-mail: riccardo.cantoro@polito.it; Bernd Becker, Albert-Ludwigs-Universitat Freiburg, Freiburg, Germany; e-mail: becker@informatik.uni-freiburg.de; Matteo Sonza Reorda, Control and Computer Engineering (DAUIN), Politecnico di Torino, Torino, Italy; e-mail: matteo.sonzareorda@polito.it.

ACM 1084-4309/2025/01-ART23
https://doi.org/10.1145/3706635

CCS Concepts: • **Hardware → Test-pattern generation and fault simulation**; **Safety critical systems**;

Additional Key Words and Phrases: Graphics processing units, functional safety, software test libraries, test quality, formal methods, reliability, bounded model checking

ACM Reference Format:
Nikolaos Deligiannis, Tobias Faller, Josie Esteban Rodriguez Condia, Riccardo Cantoro, Bernd Becker, and Matteo Sonza Reorda. 2025. Enhancing the Effectiveness of STLs for GPUs via Bounded Model Checking. *ACM Trans. Des. Autom. Electron. Syst.* 30, 2, Article 23 (January 2025), 24 pages. https://doi.org/10.1145/3706635

1 Introduction

In the last years, **Graphics Processing Units** (**GPUs**) have been adopted in a wide variety of applications. This is due to the remarkable parallelization capabilities and computational power GPUs provide. Among the application areas where GPUs are used, some fall in the safety-critical domain [27]. For example, in the automotive industry, GPUs have found widespread utilization in various applications such as autonomous guided vehicles [2, 3] and advanced driver assistance systems [26]. GPUs are particularly important for computationally intensive tasks involving **machine learning** (**ML**) applications and computer vision algorithms. In the avionics and space industry, GPUs are utilized as underlying engines to support vision-based navigation and mid-air object detection [5]. Additionally, GPUs aid in computationally intensive tasks, including flight management and data processing, playing a vital role in the industry [1, 22]. Furthermore, there are plans to employ GPUs in railway systems as a powerhouse to enable trains' safe and dynamic management based on environmental and geometrical parameters [4]. Lastly, GPUs are utilized in various industrial applications, including industrial control robots and predictive equipment maintenance [25]. It is worth noting that GPUs are not the exclusive hardware options for data accumulation and processing tasks (e.g., silicon health and operational metrics [34]). In many cases, **Tensor Processing Units** (**TPUs**), ASICs, and FPGAs are used alongside GPUs for these purposes.

In such safety-critical scenarios, manufacturers must provide additional assurances that failures will not occur during the device's mission cycle. Furthermore, in the rare event that a fault activates an error, it must not propagate to a system failure that risks human life or endangers the environment. The functional safety standards (e.g., DO-254 for avionics, EN-50129 for railways, IEC-62061 for industrial machinery, and ISO-26262 for automotive) mandate high test coverage thresholds to be reached for an electronic device mounted on a safety-critical system to be considered safe. As an example, in automotive ISO-26262 requires 98% functional fault coverage for every highly critical environment in the car (e.g., airbags). This challenging target is normally achieved using a mix of different solutions, including redundancy, **Design for Testability** (**DfT**), and functional self test. When it comes to the latter, which involves conducting tests by manipulating functional inputs and observing corresponding outputs without relying on DfT, test engineers frequently turn to ***Software-Based Self-Testing*** (**SBST**) [28]. This approach helps them generate ***Software Test Libraries*** (**STLs**) tailored to specific fault models and widely used to detect permanent hardware faults (e.g., stuck-at and transition delay faults) occurring during the operational phase (*in-field test*).

The SBST strategy is a flexible and non-intrusive method that relies on carefully crafted ***Test Programs*** (**TPs**) and utilizes the **Instruction Set Architecture** (**ISA**) to apply test patterns. These patterns are designed to activate potential faults within a targeted device unit and propagate their effects to some visible location(s). TPs are primarily developed at the assembly level, and in practice, an STL is a collection of TPs. These TPs effectively allow the detection of hardware faults in a device. Their development is up to the manufacturing company, which then passes them to

the system company, which integrates them into the application software. STLs can be activated during the operational phase, e.g., during the application idle times. The dominant fault models currently supported by the safety standards are the stuck-at and the transition delay models. STLs have been used extensively in the past for end-of-manufacturing [28] and in-field test in the case where the **circuit-under-test** (**CUT**) is a processor or controller [17, 36] and are now used for in-field test of GPUs as well [30].

1.1 Previous Works

In the literature, several works addressed the development and adaptation of SBST strategies for GPUs [15, 20, 30]. Some authors demonstrated the feasibility of adopting pseudo-random test strategies (used initially in CPUs) for functional units in GPUs [15]. Other strategies targeted internal registers, some simple controllers, and functional units by exploiting **Automatic Test Pattern Generation** (**ATPG**) algorithms to provide reasonable test patterns later translated into equivalent machine instructions and assembled as TPs [19, 20]. Nevertheless, both (pseudo-random and ATPG-based strategies) are mostly effective on combinational units.

Other more elaborated structures, such as schedulers and in-chip accelerators, require costly custom testing strategies and involve considerable engineering effort, commonly combining architectural analyses and ad-hoc algorithms to develop acceptable test routines [16].

Authors in [9] proposed custom fault primitives to develop TPs addressing memories inside controllers. In [11], the authors proposed sets of custom TPs, sequentially executed, addressing the functional test of registers in the GPU's core pipeline. Similarly, in [12], custom testing strategies are introduced to test some embedded memories in the GPU architecture functionally. In [21], the authors face the testing of **Tensor Core Units** (**TCUs**) by resorting to ***Universal Test Patterns*** (**UTPs**), which are derived from the unit's functionality and their primitive computations. While custom test routines might be effective locally, these strategies can hardly be applied to more complex structures in GPUs, such as large controllers. In fact, the results in [30] indicate that the combination of several testing strategies is mostly effective in functional units and memories within GPUs. Unfortunately, those results also show limitations in test effectiveness and quality on complex units, such as controllers, so indicating that clever algorithms and strategies are required to test those complex units effectively.

Furthermore, robust formal analyses and algorithms, such as ***Bounded Model Checking*** (**BMC**) [29] have been proposed and adapted to evaluate the reliability of digital systems through building and solving weighted maximum-satisfiability problem models. Unfortunately, the main target of the evaluations has been mostly focused on processors, neglecting the parallelism and the complex structures in GPUs. A preliminary work [14] performed a first attempt to employ formal methods to identify functional stimuli and enable the control/sensitization of permanent faults in one GPU unit. However, the functional stimuli were mostly evaluated for fault activation (i.e., controllability) purposes without complete testability verdicts. Thus, developing effective and quality TPs and STLs through formal methods remains unexplored for most GPU structures.

In particular, two main factors limit the development of effective TPs for complex units in 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., control units). Moreover, in terms of fault coverage and program size, the TP quality could be negatively affected by operational constraints and structural restrictions, leading to insufficient fault coverage of functional ATPG-based TPs, so exacerbating the need for complementary methods or alternative mechanisms to improve the quality of TPs for GPUs.

1.2 This Work

In this article, we present a novel approach leveraging formal methods to improve the quality of pre-existing STLs targeting stuck-at faults in complex units within GPUs.

Our approach cleverly combines the structural constraints of a GPU unit and their parallel programming constraints (i.e., parallel thread branching, divergences, convergences, and scheduling operations) to generate effective test patterns that can then be translated into equivalent parallel instructions and TPs for GPUs. In detail, we propose a *BMC*-based method to identify candidate test patterns while considering functional constraints applied to a targeted GPU sub-unit (e.g., a controller) and address hard-to-test faults. The generated patterns are then analyzed, translated into TPs, and validated via focused fault simulation campaigns to provide a complete testability verdict.

In particular, given the gate-level description of a GPU unit and a list of its functional constraints, our approach generates functional stimuli that enable the control/sensitization of stuck-at faults. If this is not possible for a certain fault, then this fault is marked as functionally untestable. Otherwise, we generate a functional test pattern, which is further used to (and possibly optimized to) propagate the fault up to the unit observation/test points (i.e., the unit's primary outputs). This goal is pursued without resorting to any DfT infrastructure, as it is common for other in-field test solutions. Finally, the generated functional test patterns are grouped and transformed into TPs. Moreover, given an STL for a specific unit, we can identify the untested faults and, thus, systematically target them, i.e., generate patterns for them (if possible) and then convert them to TPs to enhance the STL's overall fault coverage. Basically, we achieve this by first reducing the pattern generation problem to a BMC problem and then by effectively solving it using an appropriate solver [23].

When applied to an existing STL, our method offers two enhancements. Firstly, performing BMC-based pattern generation on a standalone sub-unit of a GPU enables the exploration of focused test patterns that resort to functional constraints, which increase the fault coverage of a pre-existing STL by specifically targeting hard-to-test faults, which is unfeasible in other SBST approaches. This enables a more thorough assessment of the system's robustness. Secondly, the evaluation of functional constraints in our approach supports the identification of some functionally untestable faults within a targeted unit. By identifying and addressing these faults, we can enhance the overall effectiveness and reliability of the STL by marking them as safe and by removing them from the computation of the STL's fault coverage. Lastly, we demonstrate our method starting from the basic GPU behavior without making assumptions about the application environment. That is, no mission profile was enforced in the form of functional constraints during our BMC pattern generation. Since adding new constraints will make the task of the BMC easier, the effectiveness of the proposed method is expected to increase if a mission profile is considered.

To the best of our knowledge, this is the first work proposing an approach to enhance TPs' quality (in terms of fault coverage) by employing formal methods on STLs for GPUs.

In particular, our approach employs a new elaborated, powerful, scalable, and dynamic mechanism for constraint specification (functional constraint formulation mechanism) to face the highly complex nature of the structures in GPU designs (i.e., constraints of architectural and functional parallelism, including correctly modeling warps, parallel thread branching, convergences, scheduling operations, control signals and protocols) and allow the effective generation of test patterns. Moreover, the fault propagation evaluation is embedded as part of the BMC problem. Furthermore, no extra computational effort is spent on generating an extraction sequence.

We use the (`FlexGripPlus`) [31] GPU model as the instrument to validate the effectiveness of the proposed approach on the STLs. In particular, the (`FlexGripPlus`) model is the only available low-level micro-architecture open-source GPU model supporting the CUDA programming model and with several available reference STLs. We targeted, through application-independent functional

constraints, two key units for the GPU operation (the *decoding unit* and the *divergence management unit*) for which it is hard to develop effective TPs and STLs due to their operation and functional constraints [30]. Both units are strategically targeted since they are functionally different or do not even exist in CPUs, emphasizing the peculiarities of GPUs. Furthermore, these units differ in architectural complexity and size (i.e., total number of gates). Namely, the decoding unit is a medium to low-complexity circuit; instead, the divergence management unit is a high-complexity circuit.

To summarize, the main contributions of this work are

- —The development of a novel approach leveraging BMC to improve the fault detection capabilities of pre-existing STLs for GPUs, specifically targeting stuck-at faults.
- —The introduction of an elaborate constraint specification mechanism to address hard-to-test units in GPU. This mechanism combines a GPU's unit structure and the parallel programming features as valid constraints for effectively generating candidate test patterns, which are translatable into parallel assembly instructions.
- —The definition of a constraint specification mechanism that embeds and combines fault propagation within the BMC problem, eliminating extra computational effort for generating extraction sequences and feasible candidate test patterns.
- —The extensive evaluation and validation of the proposed formal approach for enhancing STLs in GPUs by resorting to the open-source GPU model `FlexGripPlus`, targeting the decoding and divergence management units. The quantitative results indicate an increase in the amount of identified new test patterns for both units (1,172 and 2,421, respectively) and a significant improvement in the fault coverage for both units (around 9.57% and 2.19%, respectively).

It is important to note that for the purpose of the experiments reported in this article, we did not apply any system or application-specific functional constraints, which could significantly increase the achieved fault coverage by increasing the number of safe (i.e., functionally untestable) faults [8].

It is also worth noting that our approach is intended to support the development and enhancements of effective STLs when the micro-architecture of a targeted device is available. Although we refer to the architecture and the technical vocabulary of NVIDIA in the text, the proposed STL enhancement methodology is flexible (non-architecture-specific), and it can be used to target other GPU units and other architectures. Adapting further units requires a clear definition of the functional constraints for a targeted unit for their later assembly in the pattern generation flow.

The rest of the article is organized as follows. In Section 2, we provide extensive background information on the architecture of the GPUs and on the functional units that we will be targeting as CUTs. In Section 3, we present in detail our method and further explain the differences and novelties with respect to the method presented in [14]; and in Section 4, we explain the flow of our experiments. In Section 5, we present our experimental results on the two functional units (decoder and divergence management unit) while elaborating on the differences in terms of constraint formulation and runtime of the method. Lastly, in Section 6, we draw some conclusions.

2 Background

2.1 Organization of GPUs

GPUs are special-purpose hardware accelerators exploiting a mix of ***Multiple-Instructions*** *Multiple-Data* (**MIMD**) and ***Single-Instruction Multiple-Data*** (**SIMD**) paradigms to efficiently process large amounts of data in a parallel fashion [10]. In particular, modern GPUs comprise cluster arrays of parallel hardware processors (also known as *Streaming Multiprocessors* or SMs), see

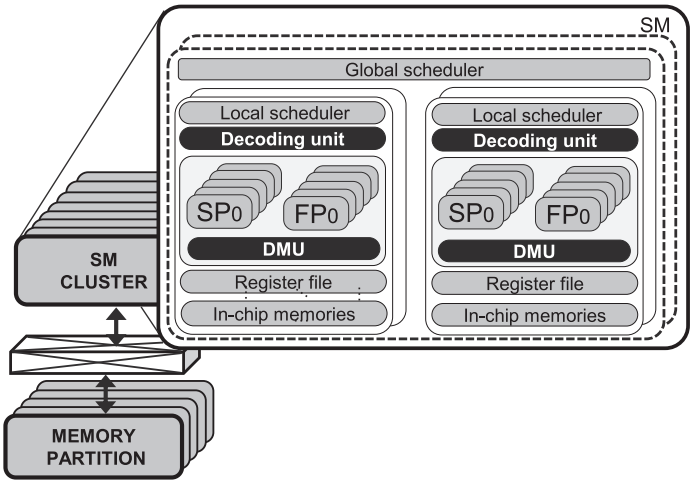

Fig. 1. A general scheme of the GPU's organization.

Figure 1. Each SM includes a set of functional hardware units (e.g., Floating-Point units "*FPs*" and special co-processors "*SPs*") to operate many threads (e.g., 1,024 threads) per task. These threads are organized in sets of warps (e.g., 32 threads per warp) for their submission and execution inside the SM's units [24].

Current GPU generations improve the application's performance and provide extensive flexibility (for their adoption in several domains) by including hardware units devoted to controlling and handling the operation of the threads configurable per application. This control involves the management and tracing of the control flow per thread, as well as the management of the flow per task. Those hardware units include schedulers, dispatcher controllers, and control-flow management sub-units [13]. In particular, the schedulers and control-flow management units play a crucial role in the execution of the parallel tasks of an application. The schedulers distribute the task workloads among the available SMs. At the fine-grain level (i.e., thread), the control-flow management units handle the running thread execution paths that allow coherency and correct task completion.

Other critical units in GPUs, such as decoder controllers, efficiently process the different micro-instructions per SM according to the submitted tasks. These decoders resort to the GPU's ISA (e.g., SASS [33]) to effectively process and collect the required operands for the execution of each machine instruction. Interestingly, both control units (control-flow management units and decoder controllers) are so critical that faults inside them might corrupt an application. Thus, efficiently identifying faults in such units is vital to correctly operating a GPU system.

2.2 Control-flow Management in GPUs

Commonly, parallel programs comprise several fragments of parallel execution combined with sequential and conditional operations. GPUs resort to infrastructure composed of crucial hardware units inside the SMs, to control the execution of parallel tasks involving conditional branches, conditional breaks, calls or jumps to subroutines, return from subroutines, and especially managing divergence and convergence features among the threads [13]. In parallel tasks, thread divergence appears when one or a subset of threads executes one or more alternative paths (operating different instructions). Thus, sophisticated mechanisms are involved in controlling and storing the starting and ending points for each path, as well as in the correct execution of each path. Thread

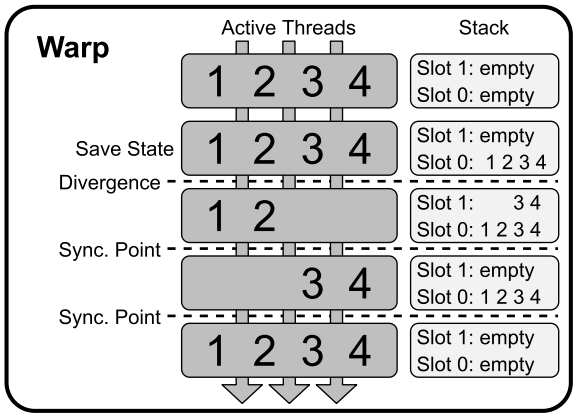

Fig. 2. Thread divergence handling of a task using a stack and explicit synchronization points in GPUs.

divergence in a task is directly associated with conditional branches or conditional breaks inside one or more loops in a program [35].

Figure 2 shows a representative example of how a task (with four threads) is handled when thread divergence appears. First, a post-divergence state (*convergence point*) is explicitly saved by a hardware controller by pushing it onto the divergence stack to be restored later. Then, a divergence occurs, resulting in some threads executing operations (*active*) while others are set as inactive and pushed onto the stack for a later run. Once a synchronization (*convergence*) point is reached, those idle threads are now active for operation, while the previously active threads are disabled. Once all threads reach the convergence point, the topmost thread state is restored from the stack to continue executing the task in parallel.

To utilize the full performance of the parallelism in GPU architectures, *control-flow hardware units* (i.e., schedulers and dispatchers) are specially co-designed (combining software and hardware) to effectively manage the hardware parallelism and optimize the software implementation as a set of the different programming control-paths. Each control path is represented as a sequence of instructions triggered by control-flow instructions according to the programming flow.

The *instruction decoding unit* is used on each SM to configure the hardware structures for the execution of parallel instructions. Similarly, the *Divergence Management Unit* or DMU (also known as *Branch unit* or *Convergence Management Unit*) is a fundamental structure handling and managing the control-flow operation of each thread executed on the SM.

In both cases, the complexity of the structures in the units is proportional to their intended functionality in the system. In the first case, the decoding process mainly uses direct mapping between the input mnemonic and the machine language. Thus, this unit is mainly composed of highly dense combinational circuits. In contrast, DMUs comprise several sub-units (e.g., optimized state machines, stacks, and glue logic) to manage and interleave the execution of different threads on multiple programming flows for a given task in the SM. The internal sub-units and their elaborate interaction inside each unit increase the complexity of effectively developing functional testing solutions (e.g., resorting to testing instructions) since the implicit operational features of a unit might restrict the fault controllability, observability, or both.

For the decoding unit, effective functional testing solutions require the evaluation of most instruction formats, as well as their operand variations, which directly depend on the GPU's ISA. Similarly, functional testing for the DMU imposes several constraints since this unit handles the parallel execution of threads and warps, and its operation directly depends on the structure of

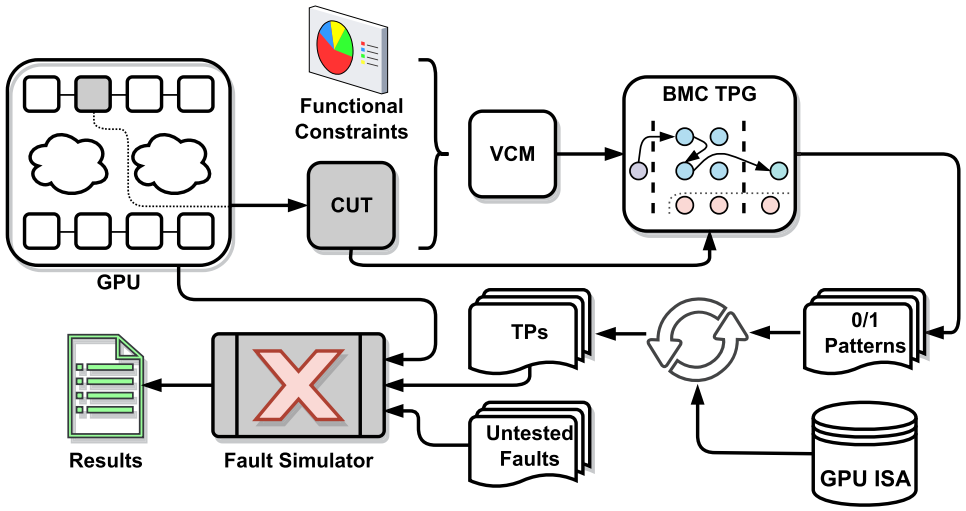

Fig. 3. A general scheme of the proposed method using BMC for test pattern generation followed by SASS transformation and fault simulation.

the task's program (e.g., the presence of divergence and convergence cases, as well as the type of instructions involved).

3 Proposed Approach

In our method, the problem of generating TPs for a given GPU unit is solved via ATPG using BMC [7]. The BMC technique [17, 29] is well known for STL generation targeting processor circuits. However, in complex designs, such as GPUs, special steps and additional parameters are required (in greater detail) to face the GPU hardware's structural parallelism and configuration features (e.g., the complex starting state of a GPU hardware circuit).

In the aforementioned articles, a synchronization sequence is typically generated that drives the circuit in a well-defined initial state where there are no Don't Care values in the flip-flops of the CUT. Whereas, if an initial state is *a priori* known and extracted (e.g., via logic simulation of the STL), in our method, it can be directly applied on every memory cell, saving valuable computation time considering the size of the GPU. Furthermore, certain GPU units depend on architecture-specific memory models (e.g., the DMU interacts with an auxiliary stack). We anticipate that our method is scalable to represent and analyze abstraction from memory models with the so-called free literals, thus making the method suitable for tackling such cases, see Section 4.2.

Our method is illustrated in Figure 3. First, we identify the target unit inside the GPU as our CUT and synthesize it into a gate-level representation by using a user-given technology library to target relevant stuck-at faults accurately. We assume that a pre-existing STL is available as a baseline (i.e., achieving a certain percentage of coverage on the CUT under the stuck-at-fault model). A preliminary evaluation of the original STL determines the faults left untested, which are the main focus of our method. Then, the expertise of GPU researchers and users allows the identification of the textual functional constraints of the CUT, which are employed to manually generate a ***Validity Checker Module*** (**VCM**), dictating the functional constraints to accurately circumscribe the search space of our **BMC-based test pattern generation** (**BMC TPG**). Then, the CUT's gate-level netlists and the VCM are passed to the BMC engine to generate 0/1 test patterns, which are then transformed to GPU assembly instructions which are then used to form

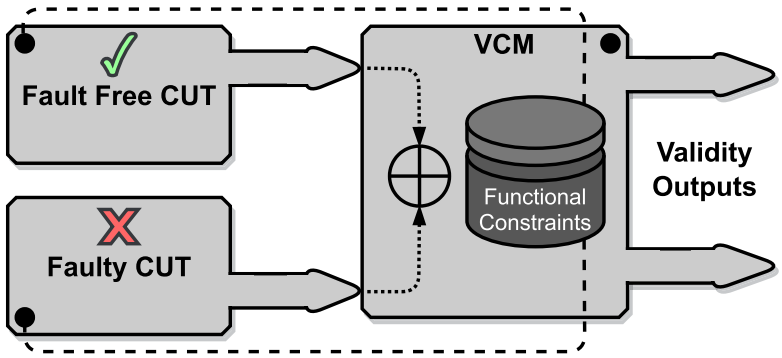

Fig. 4. VCM interaction with the miter circuit.

TPs. The generated test patterns are then transformed into GPU assembly instructions (i.e., SASS in the glossary of NVIDIA), grouped with complementary instructions (when required) as STLs, and validated with the assistance of a commercial fault simulator. In this process, we check whether the previously untested faults (from the original STL) have been detected with the newly generated instructions.

It is worth noting that the only step in the proposed method that requires manual input from the test engineer's perspective is the generation of the VCM circuit. All other steps, are fully automated.

The following subsections describe in detail the steps of the approach.

3.1 Validity Checking and Functional Constraints

The next critical step in ensuring the method's success lies in formulating and applying constraints to the CUT. A functional constraint set has to be devised to replicate the operating conditions of the CUT. This constraint set ensures that the behavior of the CUT can later be mapped to a functional STL. This constraint set generally includes enforcing valid states and control inputs (limiting the behavior of component interfaces to adhere to the present bus protocols, limiting memory address ranges, etc.). This crucial task is carried out by the VCM adapted to the special needs in the context of GPUs [18]. These constraints are derived from various sources, including (i) the complex architectural characteristics and the valid architecture states of the CUT, (ii) the interactions between the CUT and the entire GPU; for example, certain CUT input signals may have unique values enforced by neighboring functional blocks, (iii) the dependencies of the CUT on specific configurations; for instance, if a particular input is influenced by the program counter, it must not exceed a certain value. The functional constraint formulation is a vital step for the success and accuracy of the BMC TPG that follows. If wrong or incorrect constraints are applied, the resulting patterns may not be functionally valid, as the underlying SAT engine would make incorrect assumptions when trying to identify a model for the CNF.

The VCM is a small circuit, with respect to the CUT, manually written in a **Hardware Description Language** (**HDL**) like `SystemVerilog` and later synthesized into a gate-level representation using the same technology library as the CUT. The CUT and VCM are encoded into a single *miter circuit*. The VCM's circuit inputs are connected to the miter circuit (faulty and fault-free CUT) as depicted in Figure 4. Also, the VCM has full access to every internal part of the CUT (indicated by the dashed lines in Figure 4), making the application of functional constraints not limited to the CUT's primary inputs and outputs feasible. VCM's logic computes if the state and behavior of the CUT match the constraints that have been encoded inside the VCM. The VCM's validation

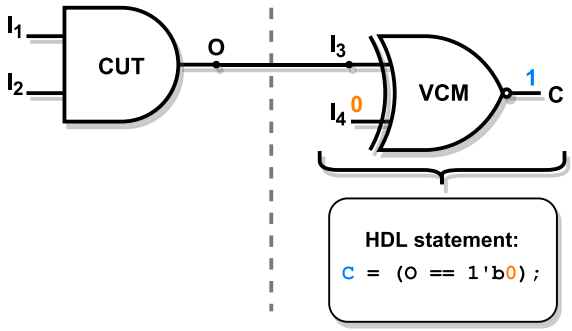

Fig. 5. Constraint application through the VCM.

result is indicated via its outputs. A Boolean value of 1 indicates that the constraint is held, while a 0 indicates that the constraint is not held and the miter circuit shows an invalid behavior. Since the VCM is generated according to the CUT specifications, we can say that it is CUT-specific. For example, if the CUT is a unit that performs mathematical operations, which are a subset of the supported instructions, we would need to include a logic block that accepts the appropriate input for the unit and enforces only the supported opcodes, i.e., the instructions utilizing the CUT, within the corresponding bit segments.

For each functional unit used as a CUT, the corresponding VCM must be developed to model its functional constraints. If adequate specification information is available for the CUT, the development of the VCM is straightforward. However, if the specification information is insufficient, the task can become more time-consuming, as the test engineer may need to infer the functional constraints of the CUT, for example, by examining its RTL.

By performing a symbolic simulation on the CUT and the VCM gate-level descriptions, we generate the Boolean formula of the combined circuit in a **Conjunctive Normal Form** (**CNF**). In this way, we embed the constraints in the CNF, hence making the underlying BMC engine aware of the desired functional scenario. Thus, we can formulate complex functional constraints via circuit logic inside the VCM and apply those constraints to the CUT.

A very simplified example of this interaction is shown in Figure 5. As the concept of the VCM applies to constraining any generic circuit, for brevity, we replace the miter circuit with a single AND gate (called CUT in the following) in this example. We wish to constrain the AND gate's output always to be 0. Note, however, that the constraints encoded in the VCM are typically of complex nature instead. The final CNF of the circuit is the conjunction of the two sub-CNFs, for the CUT and the VCM, respectively, i.e., $\text{CNF} = \text{CNF}_{\text{cut}} \wedge \text{CNF}_{\text{vcm}}$. The two independent CNFs CNF_{cut} and CNF_{vcm} are constructed using the Tseitin transformation of the logic AND and XOR gates, respectively. For the VCM, we add a unit clause for each VCM output, i.e., C, to enforce the constraint always to hold.

$$
\begin{aligned}
\text{CNF}_{\text{cut}} &:= (\neg \text{I}_1 \vee \neg \text{I}_2 \vee \text{O}) \wedge (\text{I}_1 \vee \neg \text{O}) \wedge (\text{I}_2 \vee \neg \text{O}) \\
\text{CNF}_{\text{vcm}} &:= (\neg \text{I}_3 \vee \neg \text{I}_4 \vee \ \ \text{C}) \wedge (\ \ \text{I}_3 \vee \ \ \text{I}_4 \vee \ \ \text{C}) \wedge \\
&\quad\ (\ \ \text{I}_3 \vee \neg \text{I}_4 \vee \neg \text{C}) \wedge (\neg \text{I}_3 \vee \ \ \text{I}_4 \vee \neg \text{C}) \wedge \\
&\quad\ (\neg \text{I}_4) \wedge (\text{C})
\end{aligned}
$$

Additionally, a unit clause that forces the VCM's output to 1 is created as a constraint. Since this unit clause must be evaluated as true, it simplifies the CNF_{vcm} to the unit clause $(\neg \text{I}_3)$ (step i.) which is equal to the unit clause $(\neg \text{O})$ since the XNOR gate's input is driven by the AND gate's

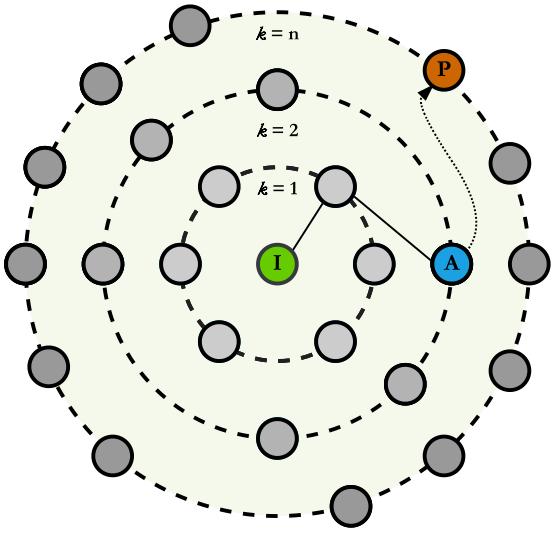

Fig. 6. BMC-based pattern generation flow for stuck-at faults.

output (step ii.). Given that the unit clause must be positively assigned in order for the CNF to be satisfied too, two extra clauses are eliminated (step iii). This sequence of events is shown below:

$$
\begin{aligned}
\text{CNF} &= \text{CNF}_{\text{cut}} \wedge \text{CNF}_{\text{vcm}} \\
&\equiv (\neg \text{I}_1 \vee \neg \text{I}_2 \vee \text{O}) \wedge (\text{I}_1 \vee \neg \text{O}) \wedge (\text{I}_2 \vee \neg \text{O}) \wedge (\neg \text{I}_3) && \text{(i).} \\
&\equiv (\neg \text{I}_1 \vee \neg \text{I}_2 \vee \text{O}) \wedge (\text{I}_1 \vee \neg \text{O}) \wedge (\text{I}_2 \vee \neg \text{O}) \wedge (\neg \text{O}) && \text{(ii).} \\
&\equiv (\neg \text{I}_1 \vee \neg \text{I}_2 \vee \text{O}) \wedge (\neg \text{O}) && \text{(iii).} \\
&\equiv (\neg \text{I}_1 \vee \neg \text{I}_2) \wedge (\neg \text{O}) && \text{(iv).}
\end{aligned}
$$

3.2 Automated Test Pattern Generation (ATPG) via BMC

The next step in our method is the BMC-based ATPG. We start with a well-defined initial state for the CUT. That is, there are no Don't Care values in the circuit and all signal assignments abide by the functional constraints. The search or transition space is circumscribed implicitly by the constraints imposed through the VCM. Remember that, during BMC, the VCM's validation outputs are constrained via an invariant to always have a value of 1. This enforces the VCM's constraints as they are propagated through the VCM's circuit logic to the VCM's inputs and with that to the miter circuit itself.

For each stuck-at fault of the CUT, an invariant is generated and added as a target state of the BMC problem, responsible for activating and propagating the fault to an observable point. For example, for a generic stuck-at fault X, a textual definition of this invariant is "Can the fault site X be set to the opposite logic value and propagate the fault-effect to a primary output of the CUT?". When this iterative process finishes, we obtain solutions (models) for each BMC problem solved for every stuck-at fault.

This process is depicted in Figure 6. Starting from the initial state `I` for our miter circuit, we try first to reach the state `A`, corresponding to activating the considered stuck-at fault. After reaching state `A`, a difference is generated within the miter circuit, and the goal now is to reach the final state `P` corresponding to the propagation of the difference to an *observation point*, i.e., a designated part of the circuit from which we can observe a signal's value and compare it with a known fault-free value to determine whether a fault is detected or not.

If the target state P is reached for a fault, then this fault is marked as potentially testable, and decoding of the CUT's input signals' literals to 0/1 logic represents a potential test pattern for the fault. Note that the verdict is potentially testable due to the fact that the propagation points are the primary outputs of the GPU unit and not the GPU itself. Hence, although it may be the case that a fault is identified as testable in the context of the GPU unit, it is not propagatable to a GPU output or observation point.

However, in the case that the target state cannot be reached, then the fault is marked as untestable under the functional constraints imposed during the BMC process. It is safe to deduce that a fault classified as functionally untestable within the boundaries of the GPU's sub-unit will be also functionally untestable for the whole GPU since if the fault cannot be controlled within the sub-unit then it will also not be controllable when considering the whole circuit as well. Furthermore, considering that the observation points of the unit are the primary and pseudo-primary outputs of the sub-module, if the fault is controlled but not observed then it will be also not propagatable to the rest of the GPU as well. Lastly, if for a given fault during the BMC process, the solver exceeds a specific limit such as maximum unrolling depth or maximum solve time, then the fault is marked as aborted and no verdict about its testability is generated.

3.3 Conversion to Assembly and Fault Simulation

Once the candidate patterns are generated, an automatic process of conversion and mapping translates the patterns into valid assembly instructions, considering the supported ISA for the target device (e.g., SASS ISA in NVIDIA GPUs).

A preliminary step identifies the device's supported ISA and analyzes two main features: (i) the instruction's opcode and (ii) the data operand formats, which are directly involved in the test pattern mapping (e.g., one or a set of instructions). We also analyze the GPU microarchitecture and the execution of assembly instructions to observe the interaction between each instruction and the input ports for the module under test, thus allowing the identification of possible patterns to be used for a given unit. In detail, the ISA identification serves as a source database employed during mapping each candidate test pattern by comparing a pattern with the database in search of possible matches. First, we identify valid opcodes, considering that an instruction opcode determines its operation type (e.g., data movement or arithmetic operation). When a mapping match is found, the candidate test pattern can be translated into an assembly instruction, and different parts of the instruction code can be created starting from the generated pattern.

The complexity of the mapping operation (from pattern to instruction) varies according to the location of the targeted unit in the GPU's structure and the unit's correlation with the assembly instructions. The mapping complexity is low when the targeted unit is part of the data path or directly interacts with the assembly's instruction execution. In contrast, the complexity is high when the mapping involves other units interacting with the instructions before the values provided by the pattern reach the targeted unit.

As an example, let us consider the case of an ALU's adder unit. In this scenario, the pattern mapping is relatively simple since a pattern might indicate an appropriate opcode for ALU operation and its operands. Another example of medium mapping complexity can be considered the decoding unit since we must identify valid instruction opcodes and operands (simultaneously) from the candidate test pattern to create the equivalent assembly instruction. Lastly, high-complexity pattern mapping scenarios, such as a DMU, require indirect or complementary steps to achieve specific unit states. For example, a test pattern may indicate that certain divergencies/convergences must first precede the targeted assembly instruction, requiring additional code to recreate the initial conditions and build a valid STL.

Regarding STL development, low mapping complexity cases do not introduce additional costs and allow a simplified TP development since simple complementary test mechanisms can be included for fault detection (e.g., memory mismatches/comparisons and signatures). In contrast, a high mapping complexity requires the identification of those units involved in the execution of an assembly test instruction. In this case, we determine those possible complementary operations (instructions) to provide initialization conditions to correctly apply instructions on a targeted unit (e.g., when a test instruction requires parallel data-movement operations from memory, we must initialize the involved registers with valid memory addresses or values for each thread). At this point, the candidate test patterns with any match as valid instruction's opcode are mapped. However, we discard those patterns without any mapping match.

Finally, we resort to focused fault simulations (through commercial-grade frameworks) to validate and confirm the effectiveness of the new TPs and STLs to detect a targeted fault inside a unit. We combine RTL and Gate-level descriptions of the device to evaluate the stimuli effects from the assembly test instructions on a targeted unit and only observe any possible corruption effect on the GPU memory as the main observation points for fault propagation.

To reduce the validation time, we only execute the candidate TPs on their targeted fault. When the candidate TP correctly activates and propagates the effects into an observable point, we classify the TP as effective and include them as part of an STL. Instead, when the TP validation fails, the fault propagation does not reach an observable point, so we discard it and classify the associated fault as potentially untestable.

4 Experimental Setup

To experimentally assess the effectiveness of the proposed method, this work targets the decoding unit and the DMU of the `FlexGripPlus` GPU model [31]. Both units, which are of sequential nature, have been synthesized using the Nangate 45nm technology library [32] via `Synopsys Design Compiler`. Then, we identified each unit's operational, structural, and functional constraints through a careful overview of their operation in the GPU. those constraints are summarized in Tables 1 and 2 and explained in the following sub-section.

4.1 Functional Constraints

Constraints for the decoding unit are applied to the control inputs (instruction opcodes), the program flow (program counter), as well as selected warp and thread states (warp IDs, lane IDs). These constraints are mostly of a combinational nature and only include restrictions that limit the opcodes to the ISA and operands to valid ranges. Regarding the decoding unit, no further assumptions need to be taken into account since the constraints enlisted in Table 1 emulate accurately an operational scenario of the unit by forcing it to execute supported assembly instructions.

For the DMU the derived constraints have a greater complexity (with respect to the decoding unit) and include constraints with sequential behavior on the control inputs (reset, instruction opcodes), the program flow (program counter, jump addresses), and warp and thread states (warp IDs, lane IDs, thread execution masks). Constraints on address ranges, masks, and program flow are necessary to enable mapping the TPG results into STLs. Thus, we further enforce via the VCM the following functional constraints:

—Disable thread divergences (thread_mask = FFFFFFFFh).
—Only short jumps ($pc_{n+1} \leq pc_n + 100h$).
—Warp ID is constant ($warp\_id_{n+1} = warp\_id_n$).
—Warp lane ID is constant ($lane\_id_{n+1} = lane\_id_n$).

Table 1. Operational Constraints of the Decoding Unit

Operational feature	Context and Operational constraint
Instruction set	All valid and 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	Value 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 (e.g., 0.32–1.53 GB)
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

On one side, a strict constraint set helps make the transformation from TPG results into an STL easier. On the other side, it leads to some faults being classified as untestable by the TPG even though with a less restrictive constraint set those faults may be testable and can still be transformed into an STL perhaps with extended effort. The reported constraints are mainly intended to represent an example used to show the feasibility of the method, to be further refined based on the knowledge of the specific scenario and application constraints. Thus, the VCM is extended with the aforementioned special-purpose configuration inputs.

These constraints were derived through an analysis of the GPU and the programming style specifications. For example, one critical constraint pertains to the behavior of the program counter, which must consistently adhere to the memory mapping of the device and should not reach extreme values. For instance, a fault scenario could necessitate a condition for activation and propagation that forces the program counter to assume a value beyond the memory region allocated to the GPU. Consequently, we categorize such faults as safe and actively prevent scenarios in which the program counter strays beyond its allocated boundaries.

Additionally, it is noteworthy that the program counter's increment following a control transfer instruction is typically modest. To address this, we consider the effective addresses within the TPs for the DMU and establish a representative value for the effective address that follows a control transfer instruction.

Regarding the thread mask, we ensure the seamless placement of consecutive test patterns or instructions in sequence by ensuring an identical number of threads in execution. Any variance in this parameter implies the implicit inclusion of additional instructions targeting the DMU, leading to changes in the number of active threads. This, in turn, may impact the activation and propagation of specific faults.

As for the lane warp parameters, they are intimately linked to the configuration of the total number of threads per kernel. When these values remain constant, it becomes feasible to execute consecutive test patterns and instructions within the same program. However, if the warp id varies

Table 2. Operational Constraints of the Divergence Management Unit

Operational feature	Context and Operational constraint
Host reset	Disabled
Unit enable	Enabled
Flow opcode	From 0 to 8, All valid opcodes for the unit
Next program counter	All values from `0x00000000` to `0xffffffff` in multiples of 8 (64-bit)
Target address	Target address of a control transfer instruction. Constrained to respect the pc bounds and format (multiples of 8)
Warp id	Current warp executing an instruction. From `0x00` up to `0x1F`. Monotonically increasing one-hot values in round-robin fashion.
Warp lane id	Current lane of a warp dispatched to execute an instruction. If all 32 cores are selected its value is `0x0`. Otherwise, it increases in a monotonic, round-robin fashion from `0x0` to `0x3`
Initial mask	Indicates the active status of the GPU. Fixed to `0xffffffff` to indicate the active status of a kernel
Current mask	Status of a warp submitted for execution in the GPU. For instance, a value of `0xffffffff` indicates a fully parallel operation of the warp
Instruction mask	Equivalent to current mask
Stack-Related Signals	
Stack data	Groups of 66-bit entries stemming from the stack that track previously occurred divergences. Each 66-bit entry has the format {active mask|opcode|pc} and each field is constrained to respect the boundaries mentioned earlier (i.e., valid opcode and pc within configuration boundaries)
Stack full	32-bit register that tracks the current state of the stack. Constrained to either hold the same value (no divergence) or to monotonically increase (divergence occurred) or decrease (reconvergence)
Stack empty	Complementary signal of the stack full

across different test patterns, there is no guarantee of executing two or more consecutive patterns. In such cases, multiple programs with distinct configurations might be necessary. Additionally, it is important to note that controlling each warp may become challenging, as their ids would be subject to the scheduling policy of the GPU core's global scheduler, which is not directly manageable.

4.2 Stack Model

In the analysis of complex CUTs, such as the DMU, the interaction between units is crucial to determine and produce feasible and valid TPs. This means that the CUT's interfaces are constrained to ensure that communication protocols are correctly handled. In particular, the DMU unit has a memory stack interface, which is vital for storing its current operative thread states. Therefore, our analysis requires correctly handling its memory protocol and active usage. In detail, the DMU is connected to a stack model for every warp, resulting in 32 stacks in total.

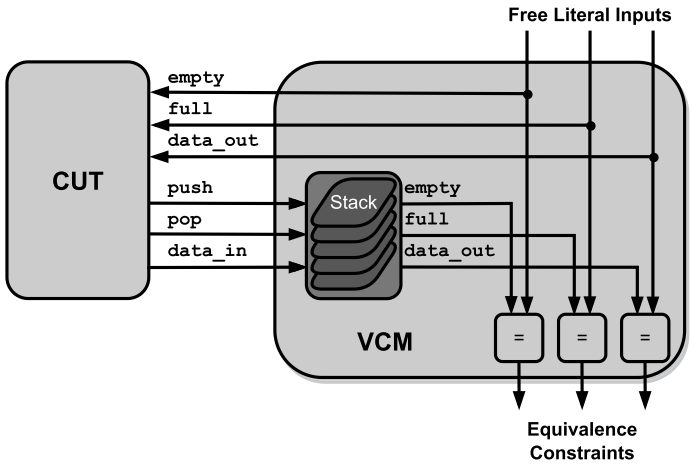

Fig. 7. VCM adapted with an integrated stack for the DMU unit.

Figure 7 depicts a scheme of the adaptation performed on the VCM to embed and hold a stack model during the DMU analysis. To speed up the unit analysis, the stack model is limited to a reasonable size (i.e., depth of two entries), enabling the generation of patterns (and STLs) with up to two divergence points and describing the five minimum operational behaviors: linear program flows, divergence from a non-diverged state, divergence from a diverged state, joining from two divergences, and joining after a single divergence and all combinations of repeated consecutive divergences and joins.

To interconnect the stack model with the CUT unit, our approach introduces the concept of "*Free-literal*" inputs for the VCM, which consist of unconstrained inputs for the BMC engine to support the interconnection (e.g., between the stack model's outputs with the DMU's inputs) and operation (i.e., control variables) for a targeted CUT. In addition, a set of *Equivalence Constraint* checkers is used to force the DMU's stack inputs to be equivalent to the stack model's output. Since the DMU manages 32 stacks simultaneously (one stack per warp/32 threads), the extended VCM includes the stack model as a generate statement, instantiating it multiple times. The stack's model width is 66 bits, the stack pointer is 2 bits, and the stack output is registered. It must be noted that a set of complementary constraints prevents stack overflow conditions during CUT analysis.

4.3 Formal Methods Framework and ATPG

The target unit and the VCM gate-level descriptions are parsed via FreiTest. FreiTest is our ATPG framework that is derived from PHAETON [29]. Even though the core concept of the VCM originates from PHAETON, the framework has been fully rewritten and redesigned primarily for RISC-V STL generation. This includes the circuit import, VCM handling, fault simulation, data export, and visualization, as well as CNF generation and the whole ATPG process itself. For the needs of our work, necessary adaptations for handling GPU STL generation have been added.

The mapping process employs an in-house developed tool, written in `Python`, that considers the GPU's ISA and its analysis to build a database, which serves as a starting point for comparing and generating legal instructions and test routines (if possible) from the candidate test patterns. More in detail, our mapping approach automatically checks each candidate test pattern and identifies any match among the pattern, the instruction opcodes, and the unit's input ports activating any

function in the unit. For instance, when a candidate test pattern indicates a hexadecimal code to activate the inputs of a targeted unit (e.g., `0x20`), the procedure searches in the database for any match between the unit's inputs and any valid instruction opcode. If the matching procedure is successful, it is possible to map the test pattern into a valid instruction to activate the unit. Then, the instruction opcode is completed with complementary information from the pattern or reference information (e.g., a pattern indicating an immediate movement instruction `'MOV Rx <- IMM'` without information regarding its operand requires a reference value `'IMM'`). In some cases, it is possible to directly map test patterns into instructions (e.g., one valid instruction from a test pattern). However, some test patterns require more instructions to correctly represent the operative conditions for the correct activation of the unit, (e.g., one test pattern mapped as a memory-stack read implies that this memory was previously filled, indicating that complementary instructions must first fill the memory with suitable values). According to the targeted unit, a set of custom rules is included in the description of the automatic mapping process to generate the necessary instructions for correctly executing the instruction with the test pattern. In addition, a fault detection mechanism based on software signatures and conditional statements is included as part of the mapping procedure. These complementary instructions propagate any fault effect to an observable point (e.g., the GPU's main memory) and check for any possible mismatch between a fault-free operation and the fault-generated effect.

The evaluation and validation of the new test routines is based on a set of focused fault simulation campaigns resorting to two commercial-grade logic/fault simulation frameworks (`QuestaSim` by Siemens EDA and `Z01X` by Synopsis). Due to the huge simulation time to perform a complete gate-level simulation of the complete GPU, we divide the validation into three steps: (i) fault-free RTL simulation, (ii) focused gate-level fault simulation, and (iii) RTL propagation of fault effects. The first logic simulator reads an RTL description of the GPU, which is employed to execute the test routines in the complete GPU. The complete routine determines the reference golden output (in GPU's memory) and the values (waveforms) on the input ports for the targeted unit. Then, the second simulator is used to execute a gate-level fault simulation of the faults in the unit using the waveforms as inputs. The propagation of the faults is observed at the unit's output and then employed as input in the RTL simulation to observe the propagation effect from the unit's output up to the observation point of the system. It is worth noting that the collected waveforms, during the fault simulation, only represent the corrupted impact of the test instruction; all other complementary instructions on the new test routine are executed fault-free during the RTL simulation of the effects since the fault does not affect them directly. In particular, we evaluate each test routine and identify when the hardware fault is activated and propagated across the targeted unit. The RTL simulations allow the verification of the fault propagation across the GPU and the evaluation of the software-based detection mechanism. Obviously, test patterns that cannot be mapped into valid instructions (no match with the instruction's opcode database) and those unable to propagate any effect on the system's observation point are discarded.

5 Results

Our experiments were performed on a machine equipped with two Intel(R) Xeon(R) E5-2680 CPUs and 256 GB of RAM. In detail, the generation of each candidate test pattern required approximately 3.1 and 14.7 seconds for the decoding unit and the DMU, respectively. The difference between the two times can be explained by considering the notable size difference of the two units, which in turn means a difference in the size of the two search spaces. Also, the post-processing of the results and the generation of the TPs requires an effort proportional to the number of functional constraints and (most importantly) their architectural complexity.

Table 3. Main Features of the Reference STLs Regarding Fault Coverage, Memory Footprint, and Execution Performance

GPU unit	Total faults	FC(%)	Memory footprint (bytes)	Execution (cc)	Number of instructions
SMP controller	15,200	57.8	262,378	6,125,561	65,653
Warp Scheduler	109,409	58.5	262,378	6,125,561	65,653
Fetching unit	4,605	88.8	262,378	6,125,561	65,653
Decoding unit	13,218	70.4	262,378	6,125,561	65,653
Execution unit	799.657	84.5	133,348	3,729,005	36,702
DMU	110,148	56.8	50,096	1,030,473	12,524
DMU memory	4,224	98.4	50,096	1,030,473	12,524
Address Register File	131,072	100.0	0	338,240	122
Predicate Register File	32,768	100.0	16,384	1,890,106	434
Vector Register File	131,072	100.0	32,768	108,958	82
Overall GPU's SM core	1,351,373	83.1	494,974	13,222,343	115,517

Table 4. Generated Patterns Classification

Unit	Patterns type	Generated patterns	Target faults
Decoding unit	Single	535	535
	Multiple	1,274	637
	Total	1,809	1,172
DMU	Single	7,364	7,364
	Multiple	285,395	61,329
	Total	292,759	68,693

5.1 Enhancing the Fault Coverage of STLs

As pre-existing STLs, we use those proposed in [12, 20, 30], which have been developed manually by experienced test engineers and address the functional testing of the decoding unit and the internal stack memory of the DMU. Table 3 reports the main features of the reference STLs in terms of fault coverage, memory footprint, and execution performance for the different units of a GPU. It is worth noting that the STLs address some of the control units in the GPU, such as the decoding unit, in a combined manner.

From the experimental results, we classify the patterns generated by our method by distinguishing between *single* and *multiple* patterns. The *single* category corresponds to cases with one test pattern (i.e., one SASS instruction) per stuck-at fault. This means a candidate TP activates and propagates a fault to an observable point in one instruction cycle for later comparison and detection. The *multiple* category corresponds to cases with more than one test pattern per stuck-at fault (i.e., > 1 SASS instructions), meaning that the TP requires multiple cycles to propagate and activate a fault. In both cases (decoding unit and DMU), we focused on the faults left undetected by the original STLs and aimed at generating test patterns detecting them. It is worth noting that complementary instructions are used by the TPs to allow fault detection (i.e., comparison and detection).

The classification of the generated patterns is presented in Table 4. For the decoding unit, 535 single and 1,274 multiple patterns were generated, targeting 1,172 stuck-at faults, untested by the original STL. Thus, we evaluated 1,172 TPs by mapping the patterns into instructions while

considering the parallel configuration constraints determined during the formal analysis of the unit (number of active warps, thread ID, block number, etc.).

Our results show that out of the newly identified test patterns, about 25% can be directly translated into valid instructions in the GPU's ISA and executed with minimal restrictions in terms of parallel configuration, so directly enhancing the test coverage without any significant effort in the TP generation. Thus, these new instructions can be fruitfully and easily added to the existing STL. In addition, 25.34% of the new patterns required specific parallel configurations after translation into instructions, e.g., specific memory locations (storing variables) or access to particular memory resources, such as the shared memory. This means that these instructions cannot be included in previously developed test routines, and complementary ad-hoc TPs must be created. For instance, a test pattern for a fault that requires a precise data value to be written in a specific memory address indicates that if this memory slot is already allocated for the current TP program, then a new TP must be generated with a specific configuration just for this particular fault. Finally, about 49.6% of the generated test patterns could not be mapped to any valid GPU instruction and were discarded from further evaluation.

For the DMU, 292,759 patterns were generated, out of which 7,364 were single and 285,395 were multiple, targeting 68,693 stuck-at faults.

The complex functionality performed by the DMU prevents the straightforward validation of the candidate TPs, as for the decoding unit. In fact, effective test patterns require preliminary initialization phases on the DMU unit, so involving the execution of additional instructions to create the required conditions for the instructions in charge of exciting the target fault (e.g., addressing a fault in a port handling the eighth level of the stack might require addressing first the first seven levels). In particular, some initialization strategies involve explicit control-flow management instructions to force the state of the DMU (e.g., initialization of the stack states) that are mandatory before the execution of the targeted test routines. For each instruction, we devised specific strategies. In the case of control-flow management instructions, such as conditional branches and calls to subroutines (*BRA*, *BREAK*, *CAL*, and *RET*), we use instructions forcing the execution of conditional cases or forcing pre-calls to subroutines. In some cases, pre-branches or pre-calls are required to reach more specific initialization states in the DMU and its stack. Moreover, we include instructions to validate the destination of each branch or subroutine. Similarly, for *BREAK* instructions, initialization instructions force the execution of iterative scenarios. For pre-control flow instructions (e.g., setting of convergence points *SSY* and loop ending points *PREBREAK*), additional instructions are used to force the execution of each scenario (convergence or loop) and to reach each point.

Table 5 summarizes the number of detected faults in both targeted GPU units. As reported, the generated test patterns for the decoding unit in combination with the reference TPs increased the effectiveness of the STLs by 9.57%. On the other side, the new TPs for the DMU unit increased the fault coverage by 2.19%, demonstrating the proposed approach's feasibility in generating test patterns even on complex units.

This increment was achieved despite the intricate structure of the DMU unit, the wide variety of configuration parameters (exponentially increasing the test pattern search space for any combination of operational constraints), and their structural placement (far from the execution of the assembly instructions), which increase the complexity of developing and mapping effective TPs. In fact, during the evaluation of the candidate TPs, we observed that the obtained fault coverage increment (2.19%) affected other relevant parameters and metrics in the overall STL. In particular, the associated costs in terms of test duration (around 108.14% more clock cycles), number of additional instructions (about 1,700%), and memory footprint overhead (up to 36,556%). Even when feasible according to the considered constraints, increasing further the achieved Fault Coverage

Table 5. Main Features of the Original and Enhanced STLs

	STL	Decoding unit		DMU	
		Absolute	Ratio	Absolute	Ratio
Detected faults	Original	9,305	1	62,553	1
	Enhanced	10,416	1.12	64,974	1.04
Execution time (cc)	Original	6,125,561	1	1,030,473	1
	Enhanced	6,162,959	1.01	2,144,838	2.08
Instructions	Original	65,653	1	12,524	1
	Enhanced	71,886	1.09	235,176	18.78
Memory use (bytes)	Original	262,378	1	50,096	1
	Enhanced	287,010	1.09	1,831,312	36.56

would require adding even longer (and larger) TPs, violating any reasonable threshold in terms of test time and footprint. On the other side, the faults remaining undetected after the generated TPs are highly likely not to be excitable (hence to produce any failure) in any real condition, as explained in the following sub-section.

5.2 Safe Faults and Functional Untestability

During the TPG process, there is a potential scenario in which a fault, subject to the existing functional constraints, remains untested. In the BMC context, this situation arises when the unrolling depth reaches its maximum value without successfully activating or propagating the fault or when it is conclusively demonstrated (e.g., through techniques such as Craig interpolation or k-induction) that the target state cannot be reached. In the first case, no verdict can be generated for the fault, which is then marked as *aborted*; in the second case, the fault is marked as *functionally untestable* [6]. Although these faults can be excited and propagated to an observable point in the circuit, they are incapable of causing any failure under the considered operational scenario imposed via the functional constraints through the VCM.

One of the main challenges in the development of functional test solutions is the identification of such functionally untestable faults (called *safe* by the ISO-26262 standard) on a targeted unit. For this purpose, we analyzed the complete fault list in the decoding unit through our methodology and identified 174 functionally untestable faults. Then, we correlate those faults with the state of the original STL for validation purposes. From the 174 functionally untestable faults, 123 faults are also considered as untestable by the original STLs. The missing 51 faults were classified as **not detected** (**ND**), **not controlled** (**NC**), or **not observable** (**NO**) faults, since the TPs did not include patterns to activate or propagate those faults. Thus, the results from our approach indicate that no test patterns can be used to activate and propagate effects in the unit for a certain amount of faults (which are a subset of the total functionally untestable faults), so exposing their untestable nature.

It is essential to highlight that conducting equivalent functional untestability analyses for the DMU unit is not straightforward due to the significant influence of the wide variety of operational parameters on the set of functionally untestable faults (e.g., the memory addressing capacity and the warp size can be restricted due to the application's parallel configuration). In addition, other internal components in the system (e.g., schedulers handling tasks for the GPU cores) might influence the DMU's operational parameters dynamically.

However, we consider the worst-case operational conditions in the DMU to be the main target for identifying functionally untestable faults. For example, we may consider a fault that demands

Table 6. Comparison with Commercial ATPG

	Full-Sequential ATPG	BMC-based TPG
#Generated test patterns for Decoding unit	86	1,172
#New instructions	86	1,809
%Increase in the STL FC	0.0	9.57
#Generated test patterns for DMU	419	2,421
#New instructions	419	292,759
%Increase in the STL FC	0.0	2.19

an exceedingly high number of nested divergence conditions for activation. In this scenario, the physical stack indicates the nesting limits, which might be lower than a targeted fault's needs (possibly producing a system failure). Since the maximum level of nested divergence in the DMU is much lower, the fault is classified as functionally untestable.

5.3 Comparison with a Commercial ATPG

As a complementary analysis, we compared the effectiveness of our method in generating new test patterns against a commercial-grade sequential ATPG. In this case, the sequential ATPG analyzed the DUT (e.g., decoder or DMU unit) without any operational constraint, so allowing the exploration of any possible missing candidate test pattern. After pattern identification, we employ the same translation method (from test patterns to instructions) from our proposed approach.

The results in Table 6 report the number of candidate patterns from both methods mapped into valid instructions for the GPU units. Interestingly, when using a sequential ATPG on both units, we observed that only a few new functional candidate patterns were generated, with minimal effect on increasing the STL's FC on the units. When it comes to generating functional patterns by resorting to traditional ATPG approaches, we can say that independently of the targeted unit, the results will be the same due to the nature of the full-sequential ATPG mode, which cannot handle the sequential depth of such a device effectively.

In contrast, our methodology faced the complexity of analyzing both sequential units and identified a substantial number of candidate test patterns per unit, ranging from 1,172 to 2,421. This was further reflected in the percentage of effectively mapped instructions and the increase in the STL's FC, which ranged from 2.19% to 9.57%, so demonstrating that our approach can be employed in the analysis of complex operational units. Consequently, the clear identification and formulation of the operational constraints per unit is crucial to effectively apply the proposed approach and develop/improve TPs and STLs. It is worth noting that we focused our evaluation on units of the GPU architecture. However, similar analyses can be extended to complex units in multi-threading/core processors (e.g., in-chip vector/SIMD co-processors), GPUs, and hardware accelerators.

6 Conclusions

GPUs are widely adopted in multiple domains due to the massive parallelization and computational capabilities they provide. When GPUs are used in domains that fall into the safety-critical sector, strict fault coverage targets are imposed by the safety standards. These targets are often met resorting to a number of combined techniques, including STLs. Given the lack of automation that exists in the functional test generation domain, we propose in this article a methodology that assists test engineers in enhancing the impact in terms of fault coverage of a given STL targeting specific units within a GPU. While previous works already explored the usage of formal techniques for the generation of STLs for CPUs, we focused in this article on a couple of units that are unique

to the GPU architecture, showing that the proposed method can be effectively used to enhance existing STLs targeting them.

The proposed method leverages BMC as its underlying engine to systematically target hard-to-test faults within a generic unit of a GPU, considering its operational constraints. To demonstrate the efficiency of our approach, we conducted two experiments on two units of the open-source GPU model `FlexGripPlus`: the decoding unit and the divergence management unit. Given pre-existing STLs targeting permanent hardware faults for these units, we managed to increase the achieved fault coverage by 9.57% and 2.19%, respectively.

The method is also able to detect functionally untestable faults. Most likely, the achieved fault coverage figures are very close to detecting all possible functionally detectable faults, according to the considered constraints. We experimentally assessed the effectiveness of our method considering the basic GPU behavior and without making assumptions about the application environment. That is, no mission profile-derived constraints have been applied. In such a scenario, we would expect that the total number of proven functionally untestable faults would increase since, as the search space gets restricted, more circuit locations would become untestable.

In future work, we plan to extend our method to other types of circuits, such as tensor processing units. We also plan to extend the method to face permanent delay faults.

Acknowledgments

The authors would like to thank the reviewers for their insightful comments and constructive feedback, which have greatly contributed to improving the quality and clarity of this manuscript.

References

[1] Caleb Adams, Allen Spain, Jackson Parker, Matthew Hevert, James Roach, and David Cotten. 2019. Towards an integrated GPU accelerated SoC as a flight computer for small satellites. In *IEEE Aerospace Conference (AESS'19)*. IEEE, Big Sky, MT, USA. DOI: https://doi.org/10.1109/AERO.2019.8741765

[2] Sergi Alcaide, Leonidas Kosmidis, Carles Hernandez, and Jaume Abella. 2019. High-integrity GPU designs for critical real-time automotive systems. In *Design, Automation and Test in Europe Conference and Exhibition (DATE'19)*. IEEE, Anaheim, CA, USA. DOI: https://doi.org/10.23919/DATE.2019.8715177

[3] Sergi Alcaide, Leonidas Kosmidis, Hamid Tabani, Carles Hernandez, Jaume Abella, and Francisco J. Cazorla. 2018. Safety-related challenges and opportunities for GPUs in the automotive domain. *IEEE Micro* 38 (2018), 46–55. DOI: https://doi.org/10.1109/MM.2018.2873870

[4] Jyotika Athavale, Andrea Baldovin, and Michael Paulitsch. 2020. Trends and functional safety certification strategies for advanced railway automation systems. In *IEEE International Reliability Physics Symposium (IRPS'20)*. IEEE, Dallas, TX, USA. DOI: https://doi.org/10.1109/IRPS45951.2020.9129519

[5] Marc Benito, Matina Maria Trompouki, Leonidas Kosmidis, Juan David Garcia, Sergio Carretero, and Ken Wenger. 2021. Comparison of GPU computing methodologies for safety-critical systems: An avionics case study. In *Design, Automation and Test in Europe Conference and Exhibition (DATE'21)*. IEEE, Grenoble, France. DOI: https://doi.org/10.23919/DATE51398.2021.9474060

[6] Paolo Bernardi, Michele Bonazza, Ernesto Sánchez, Matteo Sonza Reorda, and Oscar Ballan. 2013. On-line functionally untestable fault identification in embedded processor cores. In *Design, Automation and Test in Europe Conference and Exhibition (DATE'13)*. IEEE, Grenoble, France. DOI: https://doi.org/10.7873/DATE.2013.298

[7] Armin Bierre. 2009. *Handbook of Satisfiability*. IOS Press. DOI: https://doi.org/10.3233/978-1-58603-929-5-457

[8] Riccardo Cantoro, Sara Carbonara, Andrea Floridia, Ernesto Sanchez, Matteo Sonza Reorda, and Jan-Gerd Mess. 2018. An analysis of test solutions for COTS-based systems in space applications. In *IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC'18)*. IEEE, Verona, Italy. DOI: https://doi.org/10.1109/VLSI-SoC.2018.8644846

[9] Stefano Di Carlo, Josie E. Rodriguez Condia, and Matteo Sonza Reorda. 2020. An on-line testing technique for the scheduler memory of a GPGPU. *IEEE Access* 8 (2020), 16893–16912. DOI: https://doi.org/10.1109/ACCESS.2020.2968139

[10] Jack Choquette. 2023. NVIDIA hopper H100 GPU: Scaling performance. *IEEE Micro* 43, 3 (2023), 9–17. DOI: https://doi.org/10.1109/MM.2023.3256796

[11] Josie E. Rodriguez Condia and Matteo Sonza Reorda. 2019. Testing permanent faults in pipeline registers of GPGPUs: A multi-kernel approach. In *International Symposium on On-Line Testing and Robust System Design (IOLTS'19)*. IEEE, Rhodes, Greece. DOI: https://doi.org/10.1109/IOLTS.2019.8854463

[12] J. E. Rodriguez Condia and M. Sonza Reorda. 2020. Testing the divergence stack memory on GPGPUs: A modular in-field test strategy. In *IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC'20)*. IEEE, Portland, OR, USA. DOI: https://doi.org/10.1109/VLSI-SOC46417.2020.9344088

[13] Brett W. Coon, John R. Nickolls, John Erik Lindholm, and D. Svetoslav Tzvetkov. 2009. Structured Programming Control Flow using a Disable Mask in a SIMD Architecture. US Patent 7,617,384.

[14] Nikolaos I. Deligiannis, Tobias Faller, Josie E. Rodriguez Condia, Riccardo Cantoro, Bernd Becker, and Matteo Sonza Reorda. 2022. Using formal methods to support the development of STLs for GPUs. In *IEEE Asian Test Symposium (ATS'22)*. IEEE, Taichung City, Taiwan. DOI: https://doi.org/10.1109/ATS56056.2022.00027

[15] Di Carlo, Stefano and Gambardella, Giulio and Indaco, Marco and Martella, Ippazio and Prinetto, Paolo and Rolfo, Daniele and Trotta, Pascal. 2013. A software-based self test of CUDA fermi GPUs. In *IEEE European Test Symposium (ETS'13)*. IEEE, Avignon, France. DOI: https://doi.org/10.1109/ETS.2013.6569353

[16] Boyang Du, Josie E. Rodriguez Condia, Matteo Sonza Reorda, and Luca Sterpone. 2018. About the functional test of the GPGPU scheduler. In *International Symposium on On-Line Testing And Robust System Design (IOLTS'18)*. IEEE, Platja d'Aro, Spain. DOI: https://doi.org/10.1109/IOLTS.2018.8474174

[17] Tobias Faller, Nikolaos I. Deligiannis, Markus Schwörer, Matteo Sonza Reorda, and Bernd Becker. 2023. Constraint-based automatic SBST generation for RISC-V processor families. In *IEEE European Test Symposium (ETS'23)*. IEEE, Venice, Italy. DOI: https://doi.org/10.1109/ETS56758.2023.10174156

[18] Tobias Faller, Phillip Scholl, Tobias Paxian, and Becker Becker. 2021. Towards SAT-Based SBST generation for RISC-V cores. In *Latin American Test Symposium (LATS'21)*. IEEE, Punta del Este, Uruguay. DOI: https://doi.org/10.1109/LATS53581.2021.9651819

[19] Juan-David Guerrero-Balaguera, Josie E. Rodriguez Condia, and Matteo Sonza Reorda. 2021. On the functional test of special function units in GPUs. In *International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS'21)*. IEEE, Vienna, Austria. DOI: https://doi.org/10.1109/DDECS52668.2021.9417025

[20] Juan-David Guerrero-Balaguera, Josie E. Rodriguez Condia, and Matteo Sonza Reorda. 2022. A compaction method for STLs for GPU in-field test. In *Design, Automation and Test in Europe Conference and Exhibition (DATE'22)*. IEEE, Antwerp, Belgium. DOI: https://doi.org/10.23919/DATE54114.2022.9774597

[21] Saurabh Hukerikar and Nirmal Saxena. 2022. Runtime fault diagnostics for GPU tensor cores. In *IEEE International Test Conference (ITC'22)*. IEEE, Anaheim, CA, USA. DOI: https://doi.org/10.1109/ITC50671.2022.00065

[22] Leonidas Kosmidis, Jérôme Lachaize, Jaume Abella, Olivier Notebaert, Francisco J. Cazorla, and David Steenari. 2019. GPU4S: Embedded GPUs in space. In *Euromicro Conference on Digital System Design (DSD'19)*. IEEE, Kallithea, Greece. DOI: https://doi.org/10.1109/DSD.2019.00064

[23] Stefan Kupferschmid, Matthew Lewis, Tobias Schubert, and Bernd Becker. 2011. Incremental preprocessing methods for use in BMC. *Formal Methods in System Design* 39 (2011), 185–204. DOI: https://doi.org/10.1007/s10703-011-0122-4

[24] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. 2008. NVIDIA tesla: A unified graphics and computing architecture. *IEEE Micro* 28, 2 (2008), 39–55. DOI: https://doi.org/10.1109/MM.2008.31

[25] NVIDIA Corporation. 2023. Industrial-Scale AI. Retrieved 12-Dec-2024 from https://www.nvidia.com/en-us/industries/industrial-sector/

[26] Ignacio Sañudo Olmedo, Nicola Capodieci, and Roberto Cavicchioli. 2018. A perspective on safety and real-time issues for GPU accelerated ADAS. In *Annual Conference of the IEEE Industrial Electronics Society (IECON'18)*. IEEE, Washington, DC, USA. DOI: https://doi.org/10.1109/IECON.2018.8591540

[27] Jon Perez-Cerrolaza, Jaume Abella, Leonidas Kosmidis, Alejandro J. Calderon, Francisco Cazorla, and Jose Luis Flores. 2022. GPU devices for safety-critical systems: A survey. *ACM Compututing Surveys* 55, 7 (2022), 37 pages. DOI: https://doi.org/10.1145/3549526

[28] Mihalis Psarakis, Dimitris Gizopoulos, Ernesto Sanchez, and Matteo Sonza Reorda. 2010. Microprocessor software-based self-testing. *IEEE Design and Test of Computers* 27, 3 (2010), 4–19. DOI: https://doi.org/10.1109/MDT.2010.5

[29] Andreas Riefert, Riccardo Cantoro, Matthias Sauer, Matteo Sonza Reorda, and Bernd Becker. 2016. A flexible framework for the automatic generation of SBST programs. *IEEE Transactions on Very Large Scale Integration Systems* 24, 10 (2016), 3055–3066. DOI: https://doi.org/10.1109/TVLSI.2016.2538800

[30] Josie E. Rodriguez Condia, Felipe Augusto da Silva, Ahmet Çağrı Bağbaga, Juan-David Guerrero-Balaguera, Said Hamdioui, Christian Sauer, and Matteo Sonza Reorda. 2023. Using STLs for effective in-field test of GPUs. *IEEE Design and Test* 40, 2 (2023), 109–117. DOI: https://doi.org/10.1109/MDAT.2022.3188573

[31] Josie E. Rodriguez Condia, Boyang Du, Matteo Sonza Reorda, and Luca Sterpone. 2020. FlexGripPlus: An improved GPGPU model to support reliability analysis. *Microelectronics Reliability* 109 (2020), 113660. DOI: https://doi.org/10.1016/j.microrel.2020.113660

[32] Silvaco. 2011. Silvaco 45nm Open Cell Library. Retrieved 12-Dec-2024 from https://si2.org/open-cell-library
[33] Mark Stephenson, Siva Kumar Sastry Hari, Yunsup Lee, Eiman Ebrahimi, Daniel R. Johnson, David Nellans, Mike O'Connor, and Stephen W. Keckler. 2015. Flexible software profiling of GPU architectures. In *ACM/IEEE Annual International Symposium on Computer Architecture (ISCA'15)*. IEEE, Portland, OR, USA. DOI: https://doi.org/10.1145/2749469.2750375
[34] Synopsys. 2023. Silicon Lifecycle Management. Retrieved 12-Dec-2024 from https://www.synopsys.com/solutions/silicon-lifecycle-management.html
[35] Ping Xiang, Yi Yang, and Huiyang Zhou. 2014. Warp-level divergence in GPUs: Characterization, impact, and mitigation. In *IEEE International Symposium on High Performance Computer Architecture (HPCA'14)*. IEEE, Orlando, FL, USA, 284–295. DOI: https://doi.org/10.1109/HPCA.2014.6835939
[36] Y. Zhang, Yi Ding, Zebo Peng, Huawei Li, Masahiro Fujita, and Jianhui Jiang. 2022. BMC-based temperature-aware SBST for worst-case delay fault testing under high temperature. *IEEE Transactions on Very Large Scale Integration Systems* 30, 11 (2022), 1677–1690. DOI: https://doi.org/10.1109/TVLSI.2022.3186946

Received 2 August 2024; revised 1 November 2024; accepted 30 November 2024