Modular Functional Testing: Targeting the Small Embedded Memories in GPUs

Matteo Sonza Reorda, Josie Condia

Modular Functional Testing: Targeting the Small Embedded Memories in GPUs

Josie E. Rodriguez Condia and Matteo Sonza Reorda

Dip. di Automatica e Informatica (DAUIN), Politecnico di Torino, Torino, Italy,
josie.rodriguez@polito.it, matteo.sonzareorda@polito.it,
http://www.cad.polito.it

Abstract. Graphic Processing Units (GPUs) are promising solutions in safety-critical applications, e.g., in the automotive domain. In these applications, reliability and functional safety are relevant factors. Nowadays, many challenges are impacting the implementation of high-performance devices, including GPUs. Moreover, there is the need for effective fault detection solutions to guarantee the correct in-field operation. This work describes a modular approach to develop functional testing solutions based on the non-invasive Software-Based Self-Test (SBST) strategy. We propose a scalar and modular mechanism to develop test programs based on schematic organizations of functions allowing the exploration of different solutions using software functions. The FlexGripPlus model was employed to evaluate experimentally the proposed strategies, targeting the embedded memories in the GPU. Results show that the proposed strategies are effective to test the target structures and detect from 98% up to 100% of permanent stuck-at faults.

Keywords: Graphics Processing Units (GPUs) Software-Based Self-Test (SBST), Modular Test program, In-field Testing

1 Introduction

Graphics Processing Units (GPUs) are powerful devices devoted to processing high-demand data-intensive applications, such as multimedia, multi-signal analysis, and High-Performance Computing (HPC). Moreover, GPUs' flexibility and programming capabilities have boosted the operational scope of these technologies into new domains, so these devices are now also used for safety-critical applications with substantial requirements in terms of reliability and functional safety.

In the automotive field, safety-critical applications, such as Advanced Driver Assistance Systems (ADAS) [1] and sensor-fusion systems, usually require huge computational power and real-time capabilities. For this purpose, cutting-edge technologies are used to implement modern GPU platforms to maximize performance and reduce power consumption. Nevertheless, some studies [2][3] have proven that devices with the latest transistor technologies are prone to be affected by faults during the device's operative life. One of the most critical challenges arises when permanent faults (for example, caused by wear-out or aging

[4]) affect a module during the in-field operation, potentially altering the functionality and the reliability of a device. Thus, end-of-manufacturing testing is no longer sufficient. These recent challenges require additional testing procedures executed during the in-field operation of a device.

In practice, test engineers employ three main methods to perform testing during the operational phase of digital devices. These mechanisms are based on *i)* hardware, *ii)* software and *iii)* hybrid approaches and can be used to solve the new reliability and technology challenges in GPUs. The hardware mechanisms include solutions based on the addition of Design for testability (DfT) structures, such as *Logic* and *Memory Built-in Self-Test* [5], which can be activated at the Power-on or during idle times of the operation and stimulate/observe the internal modules of a device detecting possible faults [6]. Furthermore, 'Error Correcting Codes' (ECC) structures can detect errors and also provide mitigation features into memory modules or communications peripherals.

On the other hand, software solutions are based on designing special programs using appropriate combinations of instructions to test a target functionally. These non-invasive and flexible mechanisms are formally called *Software-Based Self-Test* (SBST)[7]. Finally, the last approach is based on hybrid mechanisms, combining hardware structures and software programs to detect [8] or mitigate [9] faults located in the different modules of a device. Both (hardware and hybrid) solutions are costly when targeting small modules in a GPU and should be developed and included in a design before the production phase. Moreover, both methods cannot be used in already existing hardware platforms.

The testing procedures of GPUs must consider that these special-purpose parallel architectures are particularly efficient when executing embarrassingly parallel programs, as a result of two main factors: the parallel operation of threads and the efficient procedures of loading and storing operands from/to memory. However, real applications are far from this behavior, and most of them are composed of non-easily-parallelizable algorithms. Thus, these applications usually include intra-warp divergence, which is produced when a group of threads (also known as *Warp*) follows different execution paths with different instructions. In [10], the authors analyzed the applications in the CUDA Software Development Kit (SDK) and concluded that approximately 33% of the total execution on them is devoted to process intra-warp divergence. Furthermore, in [11], the authors profile a divergence map of typical programs and workloads in GPUs. Results show that most applications might produce thousands or millions of divergence conditions during the operation of the applications.

The GPUs include several structures and features to manage issues related with the intra-warp divergence. A special structure called Convergence Management Unit (CMU) (also known as Branch Convergence/Divergence Controller, Branch Controller, or Divergence Controller) is employed to manage the intra-warp divergence. The CMU controls the operation of multiple paths in the same group of threads. Internally, the CMU evaluates control-flow instructions and uses a stack memory to store relevant information concerning the execution paths. Thus, the CMU is crucial for the correct operation of an application in

the GPU, and a fault affecting this unit can propagate through the modules and collapse the entire operation of the device and the application.

On the other hand, the use of several levels of memory supports an efficient management of operands and reduces latencies generated by the inactive threads, allowing high-performance operation in GPUs [12]. In the GPU parallel architecture, several in-chip and out-chip memories cooperate on hiding stall conditions during the operation of programs that become more critical when a few threads access dispersed memory locations. Faults in these structures might compromise the operation of a thread. Some hardware solutions, such as ECC structures are now common. However, these solutions are not always acceptable and the best tradeoffs are adopted (i.e., in some cases the ECCs are limited to massive memory structures, such as the cache, the shared memory and the register file), leaving other structures unprotected.

Several works demonstrated that SBST solutions [7] could be successfully integrated into safety-critical applications, such as the automotive ones [13]. Most previous works on GPUs proposed SBST strategies targeting some data-path modules [14], including the execution units [15] [16], the register file [17], the pipeline registers [18] and some embedded memories [19]. Moreover, other solutions targeted critical modules in the control-path (i.e., the warp scheduler [20], their internal memories [21][22], and parts of the convergence management unit [23]). Nevertheless, to the best of our knowledge, most of the proposed strategies were designed after relevant programming efforts and analyses, considering the specific micro-architectural details of the targeted structures, complicating portability and generalization. Thus, practical strategies to provide convenient procedures in developing SBST mechanisms are still missing in parallel architectures, including GPUs.

In the present work, we go beyond traditional approaches of developing test programs using assembly instructions only and we propose a modular approach to develop functional test procedures using the SBST strategy. The validation of this proposed approach employs as targets some relatively small but critical memory modules in the GPU.

The proposed modular approach exploits a key feature of most SBST strategies in which a test program corresponds to a combination of several routines, which are linked together and integrated into the test program. Thus, each routine's intended functionality can be seen as a 'modular' and independent block. This abstraction level (routines as blocks) can be used to explore alternative descriptions and observe the advantages and limitations of diverse topologies for a given target. Moreover, this method allows to port test routines between different targets, simplifying the development of functional test programs.

This manuscript is an extension of a previous work [23], which introduced a modular approach to develop parametric test programs. The procedures were experimentally validated using the memory in the convergence management module. The main novelties of this work with respect to [23] are: *i)* a detailed description of the proposed modular strategy to generate test programs, *ii)* the exploration of different test-program topologies for a given module, *iii)* the im-

plementation of different test routines (in a test program), considering operational constraints, and *iv)* the validation through experiments using the small embedded memories in the GPU core.

The proposed modular approach, and the developed SBST strategies are implemented and evaluated resorting to the FlexGripPlus model, an open-source version of the NVIDIA GPU architecture. Results show that the flexibility of the proposed modular test programs do not compromise the fault detection capabilities (from 98% to 100%) for the evaluated modules.

This work is organized as follows. Section II introduces a basic overview of the GPU architecture, with special emphasis on the FlexGripPlus and the memory modules targeted to validate the proposed strategies. Section III describes the modular SBST strategies to test permanent faults. Sections IV, V, and VI describe the procedures to develop test programs in the stack memory, the Predicate register file, and in the address register and vector register files, respectively. Section VII reports the main constraints and limitations during the development of the test programs. Section VIII reports the experimental results, and Section IX draws some conclusions and outlines future works.

2 Background

2.1 General Organization of GPUs

GPUs are special-purpose parallel processing devices composed of arrays of execution units (also called 'Streaming Multiprocessors', or SMs), implementing the Single-Instruction Multiple-Data (SIMD) paradigm [24] or variations, such as the Single-Instructions Multiple-Thread (SIMT). An SM is the main operative core inside a GPU and is organized as a few pipeline stages, including various execution units (also known as Stream/Scalar Processors, or SPs), some cache memories, a Register File (RF), one or more scheduler controllers, and one or more dispatchers. More in detail, the operation of an SM starts when the scheduler controller submits an available group of threads (also called Warp or Work-group) and one instruction is fetched, decoded and executed in the available SPs. Then, a new instruction is loaded and executed.

The SM also includes a CMU able to control different execution path in a warp, which are produced when conditional assessments are present in a parallel program. Modern SM implementations also include a memory hierarchy composed of several levels of memories, aiming at the reduction of latency and race conditions when load and store operations are performed.

2.2 FlexGripPlus

FlexGripPlus is a microarchitectural RT-level GPU model described in VHDL [25]. This model is a new version built on top of the original FlexGrip model [26]. FlexGripPlus implements the Nvidia G80 microarchitecture, supporting up to 52 assembly instructions, and is also compatible with the CUDA programming

environment. The latest version of the GPU model has the flexibility to select among 8, 16, or 32 SPs and may be configured to include both Floating-Point Unit (FP32) and Special Function Unit cores.

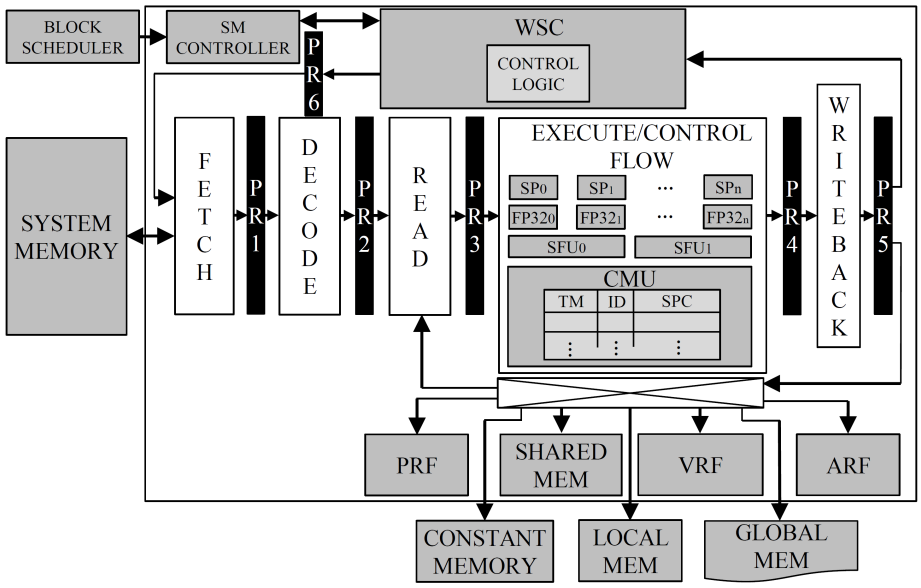

Fig. 1. A general scheme of the SM architecture in FlexGripPlus.

The FlexGripPlus architecture is based on the SIMT paradigm and exploits a SM core with five stages of pipeline (Fetch, Decode, Read/Issue, Execute/Control-flow and Write-back), as shown in Fig. 1. In the SIMT paradigm, the controller (WSC) submits a warp (32 threads) and one instruction is fetched, decoded, and distributed to be processed on an independent SP. The Read/Issue and Write-back stages load and store data operands from/to the Register File (RF), shared, global, local or constant memories. Moreover, the Address Register File (ARF) and a Predicate Register File (PRF) are used to perform the indirect addressing of memory resources and to store conditional flags, respectively.

The Execute/Control-flow stage is composed of the SPs, FP32s and other accelerators. Moreover, this stage contains one CMU used for intra-warp branching and also controls and traces the intra-warp divergence (caused when threads of a warp follow different execution paths, so executing different instructions). The CMU handle two paths in the same level of divergence by executing every path in a serial manner until both paths return to convergence (all threads in a warp execute the same instruction). The number of supported nesting divergence is proportional to the number of threads in a warp. Furthermore, the CMU also

stores the information related to perform conditional branches with several execution paths.

The following subsection describes the purpose of the embedded memories in the GPU architecture and in the FlexGripPlus model.

2.3 Embedded memories

Inside the SM core, several embedded memories are used to indirectly access to memory resources, to store the predicate flags, after the execution of conditional instructions, and to store information for divergence management. These embedded memories are limited in size, in some cases lie inside controllers, making hard and expensive to add fault detection or mitigation structures, such as ECC or BIST.

Stack Memory: This special-purpose embedded memory is located inside the CMU and stores the starting (divergence point) and ending (convergence point) addresses when a conditional assessment instruction is executed by a warp. More in detail, the memory contains a set of 32 Line Entries (LEs). The number of LEs is directly related with the number of threads in a warp and the maximum number of nested divergences per warp. A divergence point can be defined as the address, in a parallel program, where two paths (Taken and No-Taken) are produced by effect of a conditional operation, so causing intra-warp divergence (threads in a warp execute different paths with different instructions). Furthermore, a convergence point is the location in the parallel program where the intra-warp divergence ends, so the threads in a warp execute one path again.

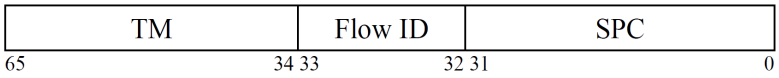

Fig. 2. Organization of one LE in the stack memory of FlexGripPlus.

Each LE in the stack is composed of three fields, (see Fig. 2). These fields are the 'thread mask' (TM), the flow ID, and the 'program counter of a warp' (SPC). The TM stores the status of the active threads in a warp and an active logic state represents the number of active threads executing a path (Taken or No-Taken). The flow ID represents the execution state of the intra-warp divergence. This field can be "01" (for a branch condition) or "00" (for a convergence point or embarrassingly parallel condition). The SPC can store the starting address of the paths or the convergence point address after both paths are executed.

The CMU employs two LEs to manage the intra-warp divergence. The first LE stores the convergence point (also known as synchronization point) and the number of active threads at the moment of starting the divergence. The second LE stores the starting address for the No-taken path and the threads to execute

this path. It is worth noting that the CMU uses a new set of LEs to store the status once nesting divergence is produced.

Predicate Register File (PRF): This module stores the predicate flags after the execution of conditional assessments, by each thread, in a warp. These conditional assessments are the product of logic-arithmetical operations or explicit setting operations. When the GPU model is configured with 8 SPs, 2,048 one-bit size locations are assigned per SP. These locations are divided in groups of 4-bits registers (C0, C1, C2 and C3) and distributed among the available threads. Each predicate register Cx stores the logical state of the zero (Z), the sign (S), the carry (C), and the overflow (O) flags for each thread. The flags remain constant in the subsequent clock cycles until the execution of a new instruction affects their state. Furthermore, these predicate flags are also used as conditions for the executions of instructions, so these are commonly read before the execution if required. Recent implementations of the PRF provide support for up to 8 predicate registers per thread.

Address Register File (ARF): This module is a structure of registers devoted to perform indirect indexing for external memories to the SM, including the shared and constant memories. These additional registers are mainly used in case of performance optimization for the several threads in a program and are mainly focused on the efficient access of memory sectors organized as arrays or matrices. Furthermore, the ARF reduces latency of accessing frequently used data by a kernel.

Each one of the eight SPs has an associated ARF module composed of 512 registers of 32 bit-size holding up to 128 threads. Each ARF module is distributed among the threads, so four registers (A0, A1, A2, and A3) can be employed per thread.

Vector Register File (VRF): This is a massive structure composed of 16KB general-purpose registers of 32 bit-size and located inside of an SM. This structure is the fastest element in the memory hierarchy of the SM and is one of the most critical units in the operation of a thread, since most instructions store or load operands from this structure. The VRF is divided among the eight cores and it is distributed among the threads in a program during the configuration phase.

Since recent GPU architectures protect the VRF against fault effects through ECC structures, this module is not considered as the main target for the development of SBST programs. However, we employ this module to validate and also explore different options of implementing test programs.

3 Modular Functional Testing Approach

The modular approach for testing is a generic strategy to develop functional test programs taking into account the microarchitectural composition of a target

module, the interaction with the parallel architecture of the GPU, its functional operation, its constraints, and the fault model. This modular approach is based on the development of a group of generic procedures, which are represented as a set of interconnected blocks, that once translated, compose a test program.

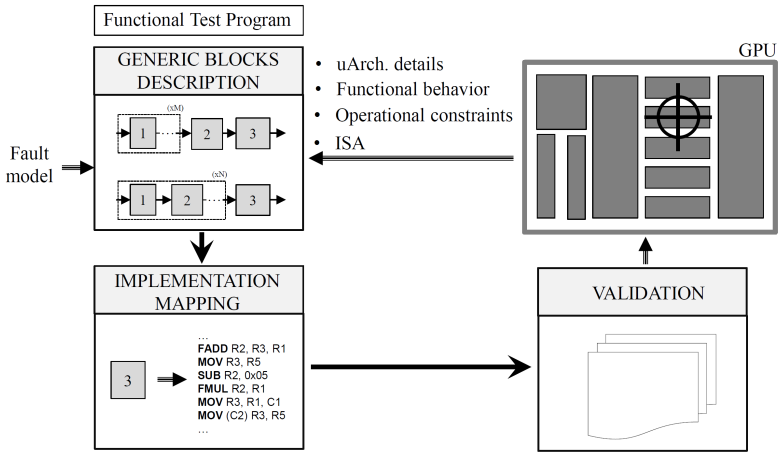

Fig. 3. A general scheme of the proposed modular approach to develop functional test programs.

The approach for modular testing considers three steps: *i)* Generic blocks description, *ii)* Implementation or mapping, and *iii)* validation, see Figure 3.

In the beginning, the organization of the test program is initially defined as a set of generic high-level blocks, which are then divided into a group of interconnected procedures to generate the intended test functionality. This modular abstraction provides flexibility that can be used to explore and address different approaches of functional test in any module.

The Generic blocks description is a strategy to represent the behavior of the interconnected procedures aiming the test a given target module. This representation considers the operation of the module and its interaction with the system, the operational constraints, and the features of the target fault model.

In this stage, the most relevant functionalities of each hardware module are employed to define a sequence of generic procedures (blocks) that, once combined, allows the functional test of the module. Each procedure is intended to aim one of this three functionalities: 1) *Fault Controllability*, 2) *Fault Observability* or 3) *Program Monitoring*. *Fault Controllability* procedures are directly related to the ability to inject test patterns through the available instructions and structural resources. *Fault Observability* procedures propagate the effect of a fault in the module into one of the available outputs of the GPU, such as control signals, buses, or external memories.

In principle, a *Fault Controllability* procedure injects test patterns in the target module. However, the feasibility of applying those patterns must be evaluated for each target module. It is worth noting that SBST is a non-invasive test strategy, so it is possible that some modules have not controllability support (i.e., instructions able to activate faults in the module) to apply a test pattern. On the other hand, *Fault Observability* procedures describe the feasible methods to propagate the fault effect to any visible output. This feature becomes important in modules that are operated by different threads. In a parallel architecture, such as the GPU, the observability of faults might implicitly include parallel fault propagation. The micro-architectural features of a module provide the composition and contribute to identifying the *Fault Controllability* and *Fault Observability* procedures for a module. Finally, *Program Monitoring* procedures introduce optional management, tracing, or check-pointing in the test program. They can be used to increase the observability of faults or other purposes, such as the test program's division into parts. In this stage, a general analysis of the module's observability and controllability allows the definition of the procedures to integrate the functional test program.

The operational constraints and the target fault model provide the relevant limitations regarding the controllability or the fault's observability. At this level, the constraints are used to propose alternative procedures aiming at the management or even removing these limitations. The features of the fault model are used as complementary information to verify that each procedure and the combination of them allow the test of faults.

For illustration purposes, the method is supported in a scheme describing the procedures (software functions) of the modular test program, see Figure 4. This scheme is composed of *blocks* (1, 2 and 3), representing modular procedures, and a set of *interconnections* (Arrows), indicating the serial sequence of operations during the test. Finally, dotted modules in a scheme represent loop functions as the repetition of one or several blocks. It is worth noting that in parallel architectures, and depending on the structural location of the target module, several threads might execute the same test program in parallel.

The flexibility of the modular approach allows the exploration of test programs by composing different block interconnections and different routines, so potentially allowing test designers to explore different benefits or constraints in the development of a test program.

The second step (Implementation or mapping) builds a test program by translating the blocks in the modular scheme into equivalent software routines. In fact, the modular schemes only consider the main functional features of a test program and include all microarchitectural constraints. Thus, the implementation stage is based on the translation or mapping of each procedure (block or function) into the equivalent software routines using the available instructions of the GPU's microarchitecture. In this step, the interconnections and internal loop are also considered and included in the mapping process.

The identification of the specific set of instructions allows the operation of any intended functionality from the modular organization, so aiming the test of

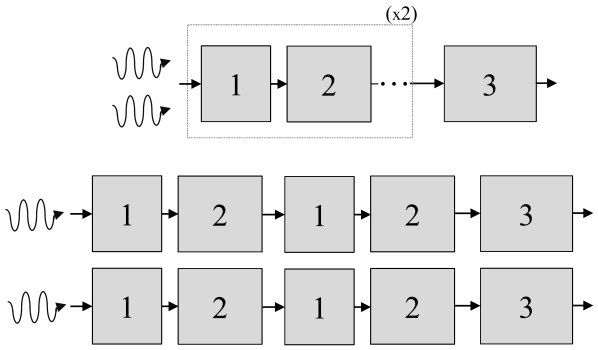

Fig. 4. An example of the modular organization of a test program. General organization and test program description (Top), Equivalent organization per thread (Bottom).

the targeted modules. More in detail, the implementation of the modular test program requires the use of an incremental approach. Initially, each block (procedure) is analyzed and translated individually. Then, a preliminary evaluation is performed to verify the functional test operation. This process is repeated for each procedure in the modular scheme. Then, the main interconnections among the blocks are mapped and checked. Finally, the internal loop is automated to provide portability according to the number of resources per thread and warps in the test program.

Each procedure (block) can be composed of a number of instructions ranging from simple instruction up to complex program procedures. Similarly, the internal loop requires the addition of several instructions at the beginning or at the end of the routines, which are commonly included to manage the control flow and the sequence of the program. In the end, the blocks of all proposed SBST strategies are described as a combination of a high-level programming language, such as CUDA (when possible), and instructions at the assembly-level (SASS for the used GPU). The main advantage is that minimal changes in a block (procedure) are required to change the functionality of a test program (i.e., the test can be focused on fault detection or diagnosis).

Finally, in the third step (validation), the self-test routines are verified using fault injection campaigns into the target modules. The output reports can be employed to improve the quality of a test program.

The following sections describe the development of the functional blocks and schemes used to develop SBST programs for the divergence stack memory, PRF, ARF, and the VRF in a GPU core.

4 Stack Memory

As introduced above, the stack memory is a particular module in a parallel architecture. This module is part of the control-flow management, so one warp may access this memory to push or pop information.

4.1 Controllability

The controllability (and the injection of test patterns) of this module is achieved by forcing the execution of controlled divergences for each thread in a warp. The generation of divergences forces the stack to store the information of the number of active threads in the TM field and the starting instruction address in the SPC field, so both fields are excited each time a divergence is produced. When more than one divergence is produce, two possible effects in the stack are observed. In the first case, a serial divergence only access the same LE in the stack and changes the values of TM and SPC. On the other hand, a nesting divergence changes the target LE and both values (TM and SPC) are stored in a new addressed LE.

The detection of permanent faults in the stack is reduced to generate and perform a sequence of divergence paths as a method to excite the TM field of each LE in the Stack memory.

Using the previous information, we propose two possible methods to control the address pointer of the LEs and inject test patterns in both fields (TM and SPC) of the LEs in the stack.

The first method (*Nesting*), see Fig. 5 (Top), generates test patterns by using a sequence of recursive intra-warp divergence routines, so nesting functions cause the movement of the address in the stack pointer into a deeper LE. The divergence is produced by successive conditional assessments between the thread identified of each thread in a warp and constant values, so generating an ordered number of comparisons (following a specific path, grey path in Fig. 3) and producing the required test pattern in the TM field of each LE.

On each comparison, one or a group of threads is disabled, so defining a pattern to be stored into the deeper LE and generating two execution paths. This method is useful in managing the addressing of the LEs and injecting patterns into the TM field. The routines on each path (Taken and No-taken) expose the presence of a permanent fault in the TM. The previous process is repeated for half the number of threads in a warp, hence two LEs are required during nesting divergence management. Once the Taken routine finishes, the DMU submits the No-taken path routine when a fault is present.

A fault-free divergence stack always executes the routine in the Taken path, which generates new divergence paths and forces the test of other levels of LEs. Moreover, once a divergence is generated, two LEs store the synchronization point and the address to start the not-taken path (which can be used as test patterns). Thus, a fault can be detected when retrieving the stored values, or when the number of threads executing a path is different from the expected one, so making the fault effects visible in the outputs.

The Nesting strategy can inject test patterns on the even LEs of the stack memory. However, the odd ones are missing. The generation of test patterns for these fields requires the explicit addition of one synchronization function (SSY) before start the comparisons causing the divergence. The effect of SSY is the movement of the address pointer to the next or deeper LE in the stack memory. Then, the same previous procedure can be applied again, so testing the odd LEs.

On the other hand, the main issue of this strategy is the procedure to manage disabled threads. When a thread is disabled, this cannot be turned active again until the divergence paths are executed, and a convergence point is reached. Thus, it is not possible to test or detect a permanent fault in a deeper LE location. This restriction implies that the comparisons should be performed multiple times, targeting different threads in the TM field. We anticipate that this strategy may suffer from considerable code length and excessive execution times.

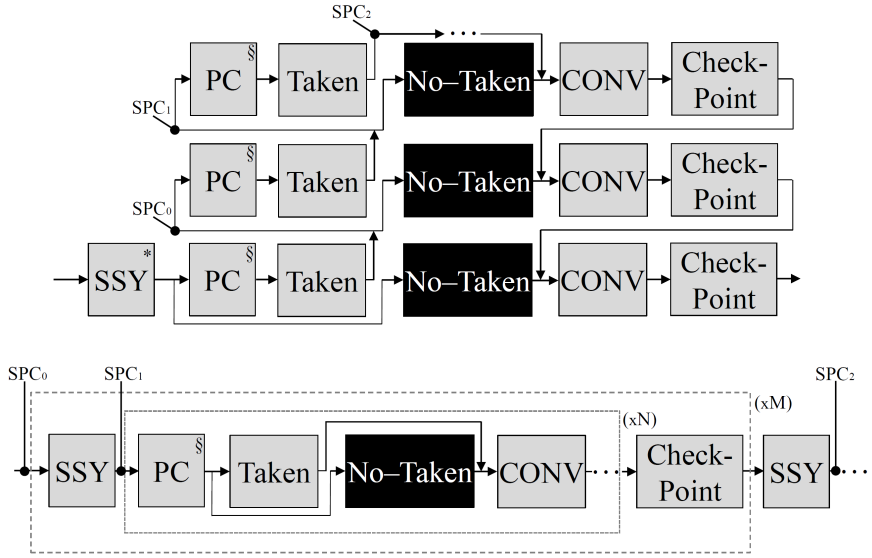

Fig. 5. A general scheme of the proposed modular SBST strategies Nesting (Top) and Sync-Trick (Bottom) to test the stack memory. (*) Optional function to test the odd LEs. (§) Optional functions to distribute the test functions in the system memory.

The second method (called *Sync-Trick*), see Fig. 5 (Bottom), exploits the functionality of synchronization functions (SSY) to deceive the CMU when testing the stack memory. This method allocates SSY functions in strategically selected locations in the test program to generate the movement in the stack pointer.

More in detail, one SSY is explicitly located before each sequence of controlled divergence functions to test the TM of a LE. Hence, this function forces the

controller to allocate a new level of LE in the memory without the need to generate an intra-warp divergence explicitly. The advantage of this method is that each LE can be addressed without the need of disabling specific threads to create nesting addressing of the memory. Thus, this strategy replaces the generation of nesting divergence by the management of the stack pointer and the execution of sequential controlled divergences.

The sequences of intra-warp divergence operations, generating the Taken and No-taken paths, inject the test patterns into the target LE. This process can be repeated N times (number of threads in a warp) to use different active threads and memory addresses as test patterns. Then, a new SSY addresses a deeper LE and the test procedure is restarted. It is worth noting that this mechanism is effective to move across one direction and reach deeper LEs in the memory. However, the returning phase (to a previous LE) requires the achievement of the convergence point address, which is initially stored in the stack by the execution of the SSY instruction.

4.2 Observability

The fault effect propagation is achieved using the Signature per Thread (SpT) strategy [18][21][27]. This mechanism assigns one signature to each thread to map and to propagate the effect of a permanent fault into the global memory. Each SpT is updated, taking advantage of both paths (Taken and No-taken) produced during an intra-warp divergence. Thus, the same mechanism used to test faults is used to increase the observability of the structure under test. Each SpT computes and accumulates intermediate results for each verified LE. The SpTs are finally grouped and stored in global memory for later analyses.

4.3 Test Program Organization

The interconnections and the main architecture of each proposed test approach are defined knowing the stack memory's observability and controllability methods.

Figure 5 presents the basic schemes of the modular composition of the Nesting and Sync-Trick test mechanisms.

In the first case (Nesting), the test generation is based on nesting divergences, so the general test strategy starts with selecting the target LE in the stack (optional use of the SSY function). Then, one divergence (two paths) divides the number of active threads, followed by the execution of a taken routine. This routine is in charge of update the SPT in the active threads. Then, a new divergence is produced (two new paths). Similarly, the same procedure restarts again, and a new taken routine is operated. The previous procedure continues until all LEs in the stack are addressed. Finally, the No-Taken paths are operated before reach the routine of convergence (CONV).

In the second approach (Sync-Trick), the operation starts selecting a target LE in the stack (using the SSY function). Then, the divergence is generated, and the two paths are created. The taken (function) path updates the signature

per thread, and, finally, the Not-Taken path is executed, and both paths reach the convergence function (CONV). The previous procedure is repeated as the number of threads in a warp (N). Then, a new target LE in the stack is addressed, and the procedure is restarted again until the total number of LEs in the stack (M) is tested.

In both schemes, the address pointers SPC0, SPC1, and SPC2 represent each block function's effect on the stack address pointer and the values stored in the SPC field of the stack memory.

As depicted in both schemes, the PC and Check-point procedures are also included in the test strategy. These complementary functions are introduced to increase observability or to allow the division into parts when possible.

A control-flow routine (PC) can be included before or in one of the divergence paths to test the high bits in the SPC field. In fact, a detailed overview of the SPC field revealed that this field is partially tested. This issue is mainly caused by the short length of the test program for both strategies. In order to complete the test of the SPC field, the test routines are redistributed across the system memory, so generating the missing test patterns and the PC routine is used to address those test routines distributed in the memory.

The check-point routines are included to verify the testing of the SPC field of the LEs in the stack. These routines are located after the convergence point. In this way, any permanent fault in the SPC is detected when the convergence point or the starting address of the No-Taken path are incorrectly read from the LEs by the effect of any permanent fault. A fault in the SPC field generates an unexpected addressing in the system memory. The permanent fault is detected by mismatches in time execution and through the signatures stored in the global memory.

The check-point routines verify, through a check-point signature, the correct flow execution of a program. Moreover, this function compares an expected check signature value with the actual accumulated value during the test program's execution. When the comparison matches, the accumulated signature is updated, otherwise the test program finishes propagating in memory the error in the SPC field of the evaluated LE. The same strategy can be applied to any of the two controllability methods (Nesting or Sync-Trick).

The use of these additional functions (Check-Point and PC) is optional, considering that these strategies are costly in memory overhead for an in-field execution. It is worth noting that the proposed technique takes into account the operational restrictions to develop the test programs using the Stuck-at fault model. Other fault models would require the adaptation of the Sync-trick mechanism. However, it would be hard or impossible to follow the Nesting strategy. The convergence function (CONV) synchronizes both paths' operation and restart (from that point) the embarrassingly parallel operation of all threads in a warp.

4.4 Implementation

The synch-trick strategy cannot be directly described in CUDA, and explicit assembly level descriptions are required. In contrast, the Nesting mechanism can

be directly mapped into the CUDA without modifications. The implemented code for both test methods is composed of the following functions: *i)* Initialization function, *ii)* synchronization function (SSY), *iii)* flow control function (PC), *iv)* intra-warp divergence function and SpT update functions (Taken and No-Taken), and *v)* check-point function (Check-point).

Each function is described independently and can be attached depending on the target of a test program. The initialization function defines and initializes the registers for each thread. Moreover, this function initializes the addresses to store the SpTs and check-point signatures. The functionality of other functions was introduced in the previous section.

Two main operations can be employed to manage the addressing of LEs in the stack memory. Initially, the convergence function is implemented using one synchronization instruction (SSY), which affects the stack pointer in the memory, and moves it to the next LE. When the program reaches the convergence point, the pointer returns to the previously addressed LE. During the execution, the first LE is used only for storing purposes. In contrast, the second LE is employed during the management of the divergence, and control-flow instructions can affect this LE with writing or reading operations. Thus, when the operation of the first path ends, the information in the second LE is used to start the not-taken path until the convergence point is reached. The CONV function, which is interpreted as the return from an addressed LE to the previous one, is described using exit control-flow instructions, such as (NOP.S).

The PC functions are relocations in the memory of the intra-warp divergence routines. These PC functions require of some instructions (in the format of 32 and 64 bits) located before each relocated function. These instructions avoid hanging conditions by permanent faults in the SPC field. In this way, when the program counter is affected by a fault, and it jumps to any unexpected memory location, it is always possible to retake control of the program and finish the execution of the GPU. Nevertheless, it is expected degradation in performance by the effect of the permanent fault. On the other hand, the intra-warp divergence routines are generated by successive comparisons between the Thread.id values, of a warp, with a constant value. The constant value is loaded using immediate instructions. The check-point signatures are predefined before execution and also loaded through immediate instructions. Then, the two paths are executed.

The functions in both paths (Taken and No-taken) update the SpT, which is firstly loaded from memory and then increased as a counter according to the path. A similar procedure is applied in the check-point routines that update check-point signatures to verify the step-by-step execution of the program, so avoiding infinite loops or unexpected branches by faults in the SPC field.

The modular description of both SBST strategies allows the exploration of multiple options for the programs. In the Nesting method, the modular approach guides the addition of functions, such as the nesting divergence, and also provides support to add or to remove optional functions targeting the SPC field (PC and Check-point). In contrast, the modularity presents considerable advantages for the Sync-Trick method. The code description of this method is scalable and

modular, so it is possible to append or remove block functions in the description of the program, targeting the individual test of LEs in the stack memory. This modularity gives us the possibility to address any or a group of LEs and to generate an independent test program. The division of the test contributes to reducing the execution time of the test program during the in-field operation of a GPU.

The Sync-Trick method can employ two approaches to evaluate LEs in memory. The first approach (Accumulative or Acc) aims the test of a consecutive group of LEs and accumulates the signatures in memory. This approach must always start from the first LE and can finish at any of the other 31 LEs in the stack.

On the other hand, the second approach (Individual or Ind) targets the testing of an individual LE and then the retrieving of signature results to the host. This approach only focuses on one of the LEs in the memory and is intended to have a reduced execution time. The performance cost (execution time (ST)) of both approaches (Acc and Ind) can be calculated using the equations (1) and (2).

$$ST(Acc) = Ts \cdot n + Ch \cdot n + SSY \cdot (n-1) \tag{1}$$

$$ST(Ind) = Ts + Ch + SSY \cdot (n-1) \tag{2}$$

where n represents the target LE in the stack memory. SSY, Ts, and Ch represent the execution time of the synchronization, test pattern injection, and check-point functions, respectively. The initialization function was not included considering that it is constant for both cases, and it is negligible in terms of duration.

From equations (1) and (2), it is clear that the cost of the Accumulative version (Acc) is higher than for the Ind version. The cost is mainly caused by the different approaches in each case. In the Acc version, the program is intended to test the number of selected LEs sequentially. In contrast, the Ind approach targets the test on one LE, so the test patterns and check-point functions are used once. The number of synchronization functions depends on the target level of LE in the stack memory.

On the other hand, the performance cost of the Nesting method is described by the expression in equation (3).

$$Ns = N \cdot Ch \cdot \sum_{i=0}^{m} (SSY + Ts) \tag{3}$$

where N represents the total number of threads in a warp, and m is the target LE to be tested. CH, SSY, and Ts have the same meaning than in equations (1) and (2). As introduced previously, the target LE could be even or odd. Thus, the starting value of i in the summation could be 0 or 1.

5 Predicate Register File

This module is a parallel structure in the GPU and it is addressed in parallel by the active threads during the execution of a program, so the maximum number of threads per core are required to perform the test of the complete module.

Controllability The PRF stores homogeneous information, so each active thread in a program have direct access to this memory, Thus, only one procedure is required to inject test patterns. The test of each register is based on the generation of load procedures (Ld), see Fig. 6. Initially, Ld targets one predicate register per thread (Cx) and assigns a value by using two possible methods: *i)* conditional assessments(PRF_T) or *ii)* direct assignments(PRF_T_R2C).

In the first case, a sequence of conditional assessments causes the activation of each predicate flag, injecting a test pattern, and propagates its effect for evaluation. On the other hand, in the direct assignment, the movement operation changes the content of the predicate register. It is worth noting that in both cases, one flag was targeted to clearly identify a fault.

5.1 Observability

A function (PROP) performs conditional evaluations to identify and classify a fault. This function propagates the effect of the target predicate register. The conditional evaluations produce two paths (Taken and No-taken). Both paths are used to update an SpT to identify if a fault was present in a given flag of a predicate register. As depicted in Fig. 6, the gray path describes the fault-free case of the test. When there are no detected faults, the test program remains convergent for all threads in a warp. In contrast, when a fault is detected, an intra-warp divergence is produced as effect of the fault and the SpT are updated indicating a detected fault. It is worth nothing that four serial procedures of conditional evaluations are required to test each register.

5.2 Test Program Organization

Two different approaches of modular test can be proposed for this module. In the first case (PRF_T), see Fig. 6 (Top), the organization of the test program is fully sequencial, so the Ld procedure injects a test pattern into a target register in the PRF. Then, the Prop routine propagates any fault and evaluates the previously register in search of faults. The taken and not-taken routines are used to update the SpT and propagate any fault effect into the available outputs. The Taken routine is intended to be operated when a faults are not present in the evaluated register of the PRF. Once, both paths reaches convergence, the previous sequence is repeated as the number of exclusive predicate registers per thread (M), the total number of flags (N) and the number of test patterns to inject per register (T).

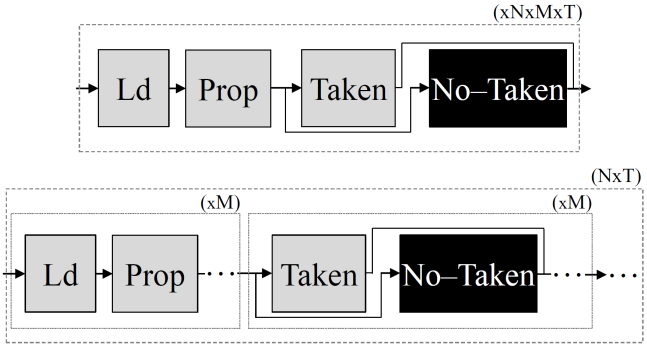

Fig. 6. General schemes of the modular approaches to test the PRF.

In the second case (PRF_T_R2C), see Fig. 6 (Bottom), the organization of the test program varies and is divided as a sequence of individual operations. First, The injection is performed using the Ld procedure and propagated with the Prop routine. Then, it is repeated M times, as the number of registers, so injecting the same test pattern to each register.

The evaluation is performed using the one divergence and the Taken procedures. Again, these procedures are repeated as the number of registers per thread, so evaluating all previously addressed flags. Finally, the complete procedure of injection, propagation and evaluation is repeated as the number of flags per register and the number of test patterns to inject. It is worth noting that the main fault model target of the proposed strategy is stuck-at faults. However, it is also possible to adapt the second case to evaluate other fault models in the PRF.

5.3 Implementation

Three versions of the Ld function can be described; one based on logic-arithmetic instructions (IOP.AND, IOP.XOR) and a second version using setting instructions (ISETP) to modify a flag in the predicate register. Furthermore, it is also possible to modify a register using a direct assignment (R2C). Thus, the main functionality (inject a test pattern) can be obtained through several descriptions. The block functions in the SBST strategy for the PRF are parametrically developed, so it is possible to easily replace one function (such as Ld) by another in the program. It is worth noting that the first two cases require several instructions before the comparison or setting.

The Prop function also supports different implementation methods. One method is based on a sequence of conditional operations, which are executed once a specific flag is active. The other method is through intra-warp divergence, so explicitly producing two execution paths corresponding to the faulty (No-Taken) and fault-free (Taken) cases. In both versions, the parametric description allows

the selection of a predicate register and a target flag, so simplifying the procedures for the generation of the test program.

The routines on each path follow a similar description with respect to those developed for the stack memory, so in principle, these functions are imported into these SBST programs.

Equations 4 and 5 represent the performance cost of both strategies for testing the PRF. As you can observe, both equations are equivalent and the performance of both strategies remains the same. The main difference between the two approaches is the order of executing the test on each register of the PRF module.

$$ST = ((Ld + Prop) \cdot M + T \cdot M) \cdot N \cdot T \tag{4}$$

$$ST = (Ld + Prop + T) \cdot N \cdot M \cdot T \tag{5}$$

The Ld implementation presents the same cost for the IOP (arithmetic and logical) and the ISETP (setting) descriptions of the functions. However, for the R2C alternative, the total description and memory footprint is reduced to a total of 36 instructions.

6 Address Register File and Vector Register File

These modules are parallel in the operation of a program in the GPU and can be accessed in parallel by the active threads.

6.1 Controllability

The ARF and the VRF modules stores homogeneous information on each register, so there are not internal field divisions. This feature allows the use of only one procedure (Ld) to perform the test pattern injection. The main idea of the Ld procedure is to perform direct assignation of test patterns on each register of both modules (General Purpose Registers Rx in the VRF and address registers Ax in the ARF). The addressing routine of the registers is mainly sequential. However, the direction and the limits are defined according to the number of threads in a program (T). It is worth noting that all active threads can access the ARF or VRF during the execution of the instructions.

6.2 Observability

The propagation of a fault is performed using a function (Comp). This procedure performs conditional assessments and compare the value in any register with several predefined masks. These masks are used to identify any fault in the evaluated registers and are the base for the comparison.

There are two possible selections of the mask: *i)* Fine-grain and *ii)* Coarse-grain. A fine-grain mask allows identifying the location of the fault affecting a

register. However, several detection procedures are required. On the other hand, a coarse-grain mask allows the rapid detection of a fault, but it is not possible to identify its location.

After each comparison, a divergence is produced. This divergence is employed as mechanism to evaluate the propagation of a fault and also to update the SpT. The gray (see Fig. 7) path shows the embarrassingly parallel operations, when the test approach is used and there are not fault in the module. In contrast, the Not-taken path in black is used when a fault is detected.

6.3 Test Program Organization

The organization of the modular test programs can be defined in two methods and can be applied for both modules (ARF and VRF). It must be considered that the internal content of each routine is adapted according to the target module. In the representative schemes, we considered a test program configured with a defined number of register per thread in a warp (N), a defined number of warps (M), a predefined number of parallel threads (P) and a fixed number of test patterns to inject (T).

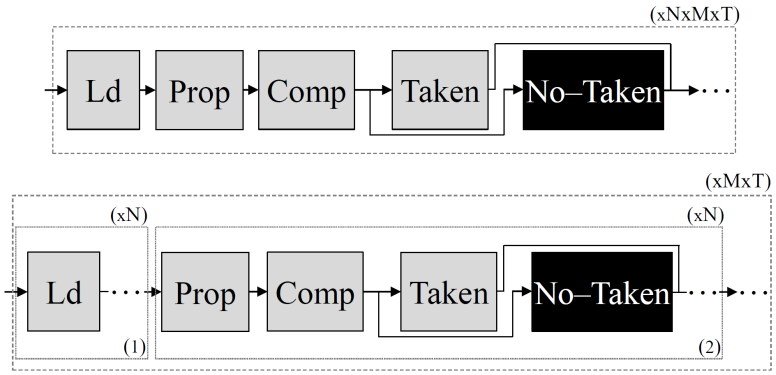

Fig. 7. General schemes of the modular approaches to test the ARF and VRF.

In the first case, see Fig. 7(Top), the procedure is performed targeting a sequential test detection. First, the Ld procedure injects test patterns in one register. Then, the Prop procedure propagates the faults effect (if present) and a comparison is performed using the Comp procedure. This comparison starts the divergence and the Taken path is evaluated to update the signature for each active thread. Finally, the previous procedure is repeated as the number of registers per thread in a warp, number of warps in the test program and the number of test patterns to inject.

The second approach, see Fig. 7(Bottom), is intended to divide the test program into small parts, so the sequential procedure of test is replaced by a two

independent stages that combined provide the same test functionality of the previously explained approach. These two independent stages are: general test patterns injection (1) and general evaluation (2). In (1), a complete sequence of one test pattern injection is performed injecting in all registers assigned to any thread. Then, in the stage (2), all propagation and comparison operations are performed to identify faults in the registers. Finally, both stages (1 and 2) are repeated as the number of warp in the program and the number of test patterns to inject.

6.4 Implementation

In the case of the ARF and the VRF memories, both approaches employ similar methods to implement the Ld function. This is based on the direct assign instructions R2A and MOV, respectively. However, the ARF structure can use an equivalent instruction to perform the assignation using another address register as source A2A.

The Comp function compares and enables a flag when mismatches are found. In both cases this function is implemented using similar mechanisms through constant values (from immediate instructions or loaded from memory). Finally, the same intra-warp divergence functions are imported and adopted for these modules. It is worth noting that the development of the blocks allows different targets of the SBST program. On the one hand, it is possible to develop the functions to only perform fault detection. Furthermore, the functions can be replaced with special versions, which provide fault diagnosis features (VRF_T_dia), so allowing the identification of the location causing the fault effect. A detailed evaluation shows that both versions of the R2A and A2A routines to implement the Ld function have the same cost in terms of execution and resource overhead.

Equation 6 describes the performance costs for both approaches, which are equivalent from the performance point of view. Nevertheless, the test can be performed in parts (ld + comp) when the modules are free and the application can be stopped for a long interval. Otherwise, the complete evaluation of each register and the splitting into parts can be employed when short interval times can be employed. It is worth noting that the performance of each approach is affected when faults are detected on each module by the execution of the missing path in order to update the detection signatures.

$$ST = ((Ld+Pr+Co+Ts)\cdot N\cdot M\cdot P = ((Ld)\cdot N+(Pr+Co+Ts)\cdot N)\cdot M\cdot P \quad (6)$$

7 Limitations and constraints

Although the proposed modular approach can be applied to any hardware module in a GPU, several programming constraints must be consider in the mapping or implementation step.

In the implementation of the SBST techniques, we explored different possible description styles for each program using high-level and middle-level abstraction

languages for programming the GPU (CUDA and PTX, respectively). However, we observed some constraints related to the implementation of the SBST technique in CUDA or PTX, due to the fact that particular combinations of instructions are required to excite any of the target modules. Furthermore, the generation of specific instructions at the two levels in some cases was not possible (e.g., the SSY instruction cannot be used in CUDA or PTX levels). Thus, we adopted the assembly language (SASS) to describe most SBST techniques that cannot be directly described at other abstraction levels.

It should be noted that compilers in the programming environment of GPUs have as main target the performance optimization of a program, so removing or reorganizing the intended test program and causing a mismatch in the intended behavior of the SBST programs. The solution to overcome this issue is based on a combination of different levels of description, when possible. This solution is affordable only when the Instruction Set Architecture (ISA) of a GPU device is well known, which unfortunately is not always the case.

8 Experimental Results

The RTL FlexGripPlus model was used in the experiments. Initially, the performance parameters of the implemented test programs are determined. Then, fault injection campaign results are reported showing the effectiveness of the proposed modular approach on the target modules.

8.1 Performance of the Test Programs

Table 1 reports the results concerning the performance parameters for all implemented test programs. It is worth noting that results reported in Table 1 were obtained by simulations performed resorting to the *ModelSim* environment.

The reported results show the performance parameters for the two possible test methods of the stack memory (Nesting and Sync-Trick methods under the accumulative and individual approaches). All versions present an overhead in the global memory of 64 locations (256 bytes) devoted to saving the SpTs and the Check-point signatures.

Regarding the performance results of both versions, it can be noted that the Sync-Trick (Ind) approach maintains an average performance cost to test any LE in the stack memory. The only difference among these programs is the number of SSY instructions included to address a selected LE. Similarly, the Sync-Trick (Acc) version can test a group of LEs consecutively. However, it requires additional execution time and cannot be stopped once the test program starts.

On the other hand, Table 1 also reports the required execution time to test the first and the second LEs in the stack using the Sync-Trick Ind (rows 2 and 3, column 4) and Sync-Trick Acc (row 5, column 4) approaches. The Individual approach requires 76 additional clock cycles to test the LEs, but it has the

Table 1. Performance parameters of the SBST programs using the two approaches to detect permanent faults in the LEs

Approach	Module	Instructions	Execution time [Clock cycles]	System memory overhead [Bytes]
Sync-Trick Ind	LE #1 Stack	403	33,449	1,612
	LE #2 Stack	404	34,211	1,616
	LE #10 Stack	412	34,589	1,648
Sync-Trick Acc	LEs 1 – 2 Stack	794	66,637	3,176
	LEs 1-10 Stack	3,922	326,423	15,688
	All Stack	12,524	1,030,473	50,096
Nesting	LE 1 Stack	683	37,986	2,732
	LEs 1-2 Stack	1,323	83,569	5,292
	LEs 1-10 Stack	6,443	528,086	25,772
	All Stack	19,883	2,567,209	79,532
PRF_T	PRF	434	1,890,106	1,736
PRF_T_R2C	PRF	398	1,795,596	1,592
ARF_T	ARF	122	338,240	488
VRF_T_det	VRF	82	108,958	368
VRF_T_dia	VRF	350	1,503,254	2,800

advantage of being able to test each LE independently. In contrast, the Accumulative method must check both LEs consecutively. Thus, the Ind approach can be adapted for in-field operation by the limited number of clock cycles required during the execution.

The performance parameters show that for the Nesting approach, there is a proportional relation between the number of instructions and the number of LEs to test. Similarly, the relationship between the execution time and the number of LEs to test presents an increasing exponential ratio. In the end, the Nesting method requires more than twice the execution time to test the entire stack than Sync-Trick using the Acc approach. The execution time could be the relevant parameter to take into account when targeting the in-field operation.

As observed in Table 1, the implemented test programs, targeting the PRF and using two different *Fault controllability* procedures (PRF_T and PRT_T_R2C) require different execution times. Moreover, the number of instructions in both test programs is different. However, the intended test functionality of both versions is the same, so showing that the modular approach can produce different test programs for the same module and allowing the exploration of different test solutions.

Regarding the implemented test programs for the VRF, The fault diagnosis (VRF_T_dia) version requires up to 13 times the execution time of the test program targeting fault detection (VRF_T_det), only. The interesting of both test programs is that the modular program is the same, but the implemented functions produce the difference in time execution and intended functionality.

8.2 Fault injection results

The fault injection environment follows the methodology described in [18], and we injected permanent faults using the Stuck-at Fault model. On each target module, fault simulation campaigns were performed injecting faults in every location of each module, meaning 4,224, 32,768, 262,144 and 262,144 permanent faults in the stack memory, PRF, ARF and VRF, respectively. These fault simulation campaigns were performed using both representative benchmarks and the test programs implementing the proposed SBST strategies. Moreover, the SBST programs targeting the stack memory were evaluated with and without the optional PC functions.

The representative benchmarks have been carefully selected to compare the detection capabilities they can achieve with the ones provided by the proposed SBST programs. Descriptions and details regarding the chosen benchmarks can be found in [18]. For the sake of completeness and comparison, the different versions of the SBST strategy are reported in Table 2. The results are reported based on the output effect of the faults as: Faults corrupting the output results, or 'Corrupted Output Data' (Data)), faults corrupting the complete execution of the system, or 'Hang', and fault affecting the performance of a given benchmark, or 'Timeout'.

The last column of Table 2 reports the testable FC (TFC) of the benchmarks and the proposed SBST strategy. The TFC is defined as the ratio between the number of detected faults and the number of injected faults after removing the untestable faults. The untestable faults are those faults that due to structural or functional issues cannot be tested and cannot produce any failure. A detailed analysis of the stack memory revealed that a total of 192 faults are untestable. These are related to the lowest bits of the SPC field of each LE, which does not affect the execution of an instruction. Thus, these faults were removed when computing the TFC.

The Sync-Trick strategy provides a moderate FC for both cases (Ind and Acc). Moreover, the FC increases when adding the PC functions and the relocation of the test functions in the memory. These comprehensive approaches (Ind+SP and Acc+SP) obtain a high percentage of FC for the target structure.

An in-depth analysis of the results shows that the Individual approach allows detecting 100% of the faults in the TM of all LEs by looking at the results of the test procedure "Data" type faults. In contrast, the Acc version causes a small percentage (0.75%) of faults produced in the TM field and visible because they hang or crash the GPU. This behavior can be explained considering that in the Ind approach, each LE is evaluated individually, and so all detections can be labeled as Data. On the other hand, for the Acc method, a permanent fault in one LE affects the synchronization point, thus corrupting the convergence point and causing the Hang condition. More in detail, a Stuck-at-0 fault is a sensitive case during the run of the test program. A fault affecting one LE when used as synchronization causes the Hang condition.

The Nesting SBST program has a slightly lower FC than Sync-Trick with an increment of more than twice the percentage of faults causing hanging and

timeout. This fault effect is equivalent to the effect shown by in the Acc version of Sync-Trick. In this case, the Nesting method generates intra-warp divergence to move the stack pointer among the LEs (in the stack memory), testing all LEs even when a fault is detected, so other LEs are also tested. The continuous evaluation generates issues when a fault affects the LE used for synchronization purposes (when testing the even LEs). Thus, the test program may confuse the convergence point and produce the Hang or Timeout condition. According to results, the Nesting strategy seems to be more susceptible to Hang and Timeout effects than the Sync-Trick using the Acc approach.

In both approaches, the addition of the relocation in memory and the SPC functions increase the testable coverage in the stack memory. However, as explained previously, these optional functions can be employed when it is possible to use the entire system memory to relocate the test functions in specific memory locations, or the application code allows this adaptation. Similarly, both SBST approaches can detect a considerable percentage of the permanent faults in the stack memory. However, a direct comparison involving the performance parameters from Table 1 shows that the Nesting approach consumes more than twice the execution time and 37% of additional instructions. In conclusion, the Sync-Trick strategy seems to be a feasible candidate for in-field operations. Moreover, the Ind strategy can be divided into parts and adapted with the application code.

Regarding the SBST program for the PRF module, the two implemented versions obtain the 100% of fault coverage. The main advantage of both versions is the fault propagation to the global memory, so enabling the detection as Data. In fact, all faults detected are identified using this classification. Thus, the main difference between SBST approaches is the performance and overhead cost, which are mainly caused by the internal description of one modular function (Ld). The previous fault coverage results allow us to validate the exploration of different methods to implement different modular functions. In both cases, the replacement of a modular function does not affect the final fault coverage. A similar situation is observed in the SBST programs for VRF. In both approaches (Detection (Dec) and Diagnosis (Dia)), the programs reach a full fault coverage (100%). However, the performance degradation rises up to 13.8 times when employing the diagnosis version of the test program. Nevertheless, this SBST version can be affordable with mild system time constraints, such as during the Switch-on of the system.

Although Table 2 reports one SBST program for the PRF, the two versions were evaluated changing each time the Ld function. The two versions achieved the same fault coverage (100%) and the Ld functions, in both versions, have the same performance cost, so both solutions can be used identically.

A comparison of the FC obtained by the proposed SBST strategies and the representative benchmarks shows that the FC using these specialized programs is higher for the targeted modules than the FC obtained with typical applications. Thus, the FC capabilities of a representative benchmark is mostly lower. This behavior can be explained considering that most applications only use parts of the modules (e.g., only the first levels of stack memory to handle the divergence

Table 2. FC results for the representative benchmarks and the proposed SBST strategies

SBST strategy or benchmark		Module	(%)				
			Data	Hang	Timeout	FC	TFC
MxM		Stack	0.00	0.38	-	0.38	0.40
		PRF	0.00	0.38	-	0.38	0.38
		ARF	25.07	0.0	-	25.07	25.07
		VRF	18.26	8.24	-	26.5	26.5
Sort		Stack	0.15	0.04	-	0.19	0.19
		PRF	0.16	0.04	-	0.20	0.20
		ARF	0.00	0.00	-	0.00	0.00
		VRF	0.18	0.07	-	0.25	0.25
FFT		Stack	0.14	0.19	-	0.33	0.35
		PRF	0.15	0.19	-	0.34	0.34
		ARF	0.0	0.0	-	0.00	0.00
		VRF	0.19	0.21	-	0.4	0.4
Edge		Stack	0.15	0.28	-	0.43	0.47
		PRF	0.00	7.05	-	7.05	7.05
		ARF	0.00	0.00	-	0.00	0.00
		VRF	12.25	5.6	-	17.85	17.85
Sync-Trick	***Ind***	Stack	65.64	2.08	1.01	68.75	72.02
	Acc		64.89	2.84	1.01	68.75	72.02
	Ind + PC		83.00	8.49	2.44	93.93	98.41
	Acc + PC		82.24	9.25	2.44	93.93	98.41
Nesting		Stack	54.12	11.81	1.23	67.16	70.04
	+ PC		76.94	13.16	2.81	92.91	97.34
PRF_T		PRF	100.0	-	-	100.0	100.0
	R2C		100.0	-	-	100.0	100.0
VRF_T	***Det***	VRF	100.0	-	-	100.0	100.0
	Dia		100.0	-	-	100.0	100.0
ARF_T		ARF	100.0	-	-	100.0	100.0

or certain registers per thread in the ARF or VRF modules) to operate the instructions of each application.

The matrix multiplication application generates one level of divergence. Thus, other levels inside the Stack memory are not employed, and the fault effect in not detected or propagated into the application. On the other hand, the VRF and the ARF modules are excited in almost 25% and 20%, which helps to explain the fault coverage obtained for both modules (26.5% and 25.07%, respectively). In contrast, the Sort application can generate intra-warp divergence, depending on the input data operands, but it remains limited to the first LE in the stack memory. However, the percentage of detection (0.33% and 0.19%) is negligible in comparison with the proposed test strategies. A similar behavior is observed for the ARF, PRF and VRF when executing this application.

The FFT benchmark produces two levels of intra-warp divergence, so using up to four LEs during the operation. This behavior slightly increases the percentage of faults detected. Nevertheless, the achieved percentage remains small. Finally, the Edge application causes two levels of intra-warp divergence and can detect some faults as Data and hangs. However, the total coverage of all representative kernels is minimal.

The previous scenario supports the idea that executing applications and checking their results (as it is often done when using a functional test approach) is definitely not enough to verify the functionality of crucial hardware modules in the GPU. Thus, special test programs, as those proposed in this work using the modular approach, are required to guarantee the correct operation of a module inside a device used in a safety-critical application.

The main advantage of the proposed method lies in its modularity and scalability. Scalability allows the configuration and the selection of the number of LEs to be tested in the SBST programs for the stack memory. Moreover, the test programs for the ARF, PRF and VRF can also be reduced to target only specific registers in the target modules. Finally, the scalability of their structure allows splitting the overall program in several parts, as presented for the Sync-Trick SBST program.

As introduced previously, the implementation of the test programs required the combination of high-level descriptions (about 15% of the total code in all SBST strategies), and the addition of assembly functions (about 85%). For both proposed SBST strategies targeting the memory stack, the synchronization functions (SSY) were implemented in assembly language. In this way, we could also avoid that the compiler removes or changes important parts of the test code. Similarly, in the PRF test program, the Comp modular function used specific procedures that required assembly language support. Thus, these parts and others are written at the assembly level. These limitations show that the development of test programs for these complex structures in GPUs requires access to the assembly formats to provide feasible and efficient solutions. The implementation effort could be reduced by the design of an automatic tool to include the subroutines at the assembly or binary level. Moreover, such a tool could also

be employed to develop modular approaches, targeting other modules, such as functional units in the GPU.

Although the proposed SBST strategies targeted the test of unprotected memories in a GPU model with the G80 micro-architecture, we still claim that the proposed methodology can be adapted and used for the most recent GPU architectures, such as Maxwell and Pascal that include similar structures. Moreover, other parallel architectures can also use the proposed method.

9 Conclusions

We introduced a modular and scalable method to design functional programs for testing in the field the small embedded memories in GPU cores. For this purpose, each target embedded memory was analyzed and based on controllability, observability and composition features, a set of parametric functions were developed and then combined to test each target structure in the GPU. Results show that the modular solution allows the exploration of the advantages and limitations of different routines employed in a test program. Moreover, this technique also allows the split of a test program into several parts, while still achieving the same FC, so allowing to adjust the test program to potential requirements of in-field operations.

As future works, we plan to extend the proposed method to test other functional units and critical modules in parallel architectures. Moreover, we plan to use the modular approach to target other fault models.

10 Acknowledgments

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

References

1. W. Shi, M. B. Alawieh, X. Li, and H. Yu, "Algorithm and hardware implementation for visual perception system in autonomous vehicle: A survey," *Integration*, vol. 59, pp. 148 – 156, 2017.
2. S. Hamdioui, D. Gizopoulos, G. Guido, M. Nicolaidis, A. Grasset, and P. Bonnot, "Reliability challenges of real-time systems in forthcoming technology nodes," in *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2013, pp. 129–134.
3. I. Agbo, M. Taouil, S. Hamdioui, P. Weckx, S. Cosemans, F. Catthoor, and W. Dehaene, "Read path degradation analysis in sram," in *2016 21th IEEE European Test Symposium (ETS)*, 2016, pp. 1–2.
4. X. Chen, Y. Wang, Y. Liang, Y. Xie, and H. Yang, "Run-time technique for simultaneous aging and power optimization in gpgpus," in *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2014, pp. 1–6.
5. A. J. Becker, C. A. S. Pathirane, and R. C. Aitken, "Memory built-in self-test for a data processing apparatus," Sep. 20 2016, uS Patent 9,449,717.

6. R. Gulati, A. E. Gruber, B. L. Johnson, J. C. Yun, D. Kim, A. K. H. Jong, and A. Saxena, "Self-test during idle cycles for shader core of gpu," Apr. 21 2020, uS Patent 10,628,274.
7. M. Psarakis, D. Gizopoulos, E. Sanchez, and M. Sonza Reorda, "Microprocessor software-based self-testing," *IEEE Design Test of Computers*, vol. 27, no. 3, pp. 4–19, 2010.
8. J. E. R. Condia, P. Narducci, M. Sonza Reorda, and L. Sterpone, "A dynamic hardware redundancy mechanism for the in-field fault detection in cores of gpgpus," in *2020 23rd International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, 2020, pp. 1–6.
9. ——, "A dynamic reconfiguration mechanism to increase the reliability of gpgpus," in *2020 IEEE 38th VLSI Test Symposium (VTS)*, 2020, pp. 1–6.
10. S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W.-m. W. Hwu, "An adaptive performance modeling tool for gpu architectures," *SIGPLAN Not.*, vol. 45, no. 5, p. 105–114, Jan. 2010.
11. B. Coutinho, D. Sampaio, F. M. Q. Pereira, and W. Meira Jr., "Profiling divergences in gpu applications," *Concurrency and Computation: Practice and Experience*, vol. 25, no. 6, pp. 775–789, 2013.
12. X. Mei and X. Chu, "Dissecting gpu memory hierarchy through microbenchmarking," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 1, pp. 72–86, 2017.
13. P. Bernardi, M. Grosso, E. Sanchez, and O. Ballan, "Fault grading of software-based self-test procedures for dependable automotive applications," in *2011 Design, Automation Test in Europe*, 2011, pp. 1–2.
14. M. Abdel-Majeed and W. Dweik, "Low overhead online periodic testing for gpgpus," *Integration*, vol. 62, pp. 362 – 370, 2018.
15. S. Di Carlo, G. Gambardella, M. Indaco, I. Martella, P. Prinetto, D. Rolfo, and P. Trotta, "A software-based self test of cuda fermi gpus," in *2013 18th IEEE European Test Symposium (ETS)*, 2013, pp. 1–6.
16. D. Defour and E. Petit, "A software scheduling solution to avoid corrupted units on gpus," *Journal of Parallel and Distributed Computing*, vol. 90-91, pp. 1 – 8, 2016.
17. D. Sabena, M. Sonza Reorda, L. Sterpone, P. Rech, and L. Carro, "On the evaluation of soft-errors detection techniques for gpgpus," in *2013 8th IEEE Design and Test Symposium*, 2013, pp. 1–6.
18. J. E. R. Condia and M. Sonza Reorda, "Testing permanent faults in pipeline registers of gpgpus: A multi-kernel approach," in *2019 IEEE 25th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, 2019, pp. 97–102.
19. ——, "On the testing of special memories in gpgpus," in *2020 IEEE 26th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, 2020, pp. 1–6.
20. S. Di Carlo, J. E. R. Condia, and M. Sonza Reorda, "An on-line testing technique for the scheduler memory of a gpgpu," *IEEE Access*, vol. 8, pp. 16 893–16 912, 2020.
21. B. Du, J. E. R. Condia, M. Sonza Reorda, and L. Sterpone, "About the functional test of the gpgpu scheduler," in *2018 IEEE 24th International Symposium on On-Line Testing And Robust System Design (IOLTS)*, 2018, pp. 85–90.
22. S. Di Carlo, J. E. R. Condia, and M. Sonza Reorda, "On the in-field test of the gpgpu scheduler memory," in *2019 IEEE 22nd International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, 2019, pp. 1–6.

23. J. E. R. Condia and M. Sonza Reorda, "Testing the divergence stack memory on gpgpus: A modular in-field test strategy," in *28th IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC 2020)*, 2020, pp. 1–6.
24. M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Transactions on Computers*, vol. C-21, no. 9, pp. 948–960, 1972.
25. J. E. R. Condia, B. Du, M. Sonza Reorda, and L. Sterpone, "Flexgripplus: An improved gpgpu model to support reliability analysis," *Microelectronics Reliability*, vol. 109, p. 113660, 2020.
26. K. Andryc, M. Merchant, and R. Tessier, "Flexgrip: A soft gpgpu for fpgas," in *2013 International Conference on Field-Programmable Technology (FPT)*, 2013, pp. 230–237.
27. A. Apostolakis, M. Psarakis, D. Gizopoulos, A. Paschalis, and I. Parulkar, "Exploiting thread-level parallelism in functional self-testing of cmt processors," in *2009 14th IEEE European Test Symposium*, 2009, pp. 33–38.