2019 22nd International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)

On the in-field test of the GPGPU scheduler memory

Stefano di Carlo*, Josie E. Rodriguez Condia†, Matteo Sonza Reorda‡,
Politecnico di Torino, Dept. of Control and Computer Engineering, Torino, Italy
{*stefano.dicarlo, †josie.rodriguez, ‡matteo.sonzareorda}@polito.it

Abstract[1]—**GPGPUs have been increasingly successful in the past years in many application domains, due to their high parallel processing capabilities and energy performance. More recently, they started to be used in areas (such as automotive) where safety is also an important parameter. However, their architectural complexity and advanced technology level create challenges when matching the required reliability targets. This requires devising solutions to perform in-field test, thus allowing the systematic detection of possible permanent faults. These faults are caused by aging or external factors that affect the application execution and potentially generate critical misbehaviors. Moreover, effective in-field test techniques oriented to verify the integrity of GPGPU modules during in-field operation are still missed. In this work, we propose a method to generate self-test procedures able to detect all static faults affecting the scheduler memory existing in each streaming multiprocessor (SM) of a GPGPU. NVIDIA CUDA-C is selected as high-level programing language. The experimental results are obtained employing the NVIDIA Nsight Debugger on a NVIDIA-GEFORCE GTX GPU and a memory fault simulator.**

Keywords—***GPGPUs, SBST, memory testing.***

I. INTRODUCTION

Some safety-critical applications in the automotive domain (e.g., Advanced Driver Assistance Systems, or ADAS) require performing real-time processing on massive amount of data coming from sensors (e.g., cameras and radars). For these applications, General Purpose Graphics Processing Units (GPGPUs) are very suitable solutions, due to their computational power and reduced power consumption. Unfortunately, this kind of applications is also highly safety critical, since the effects of any fault affecting the hardware may have severe consequences. Hence, the standards and regulations in the area (e.g., ISO 26262) require very strict constraints in terms of reliability. In order to match these requirements, no matter the complexity of the underlying devices and the advanced semiconductor technology used to manufacture them, it is mandatory to adopt some suitable form of in-field test, able to timely detect any permanent faults arising in the GPGPU. Among the possible solutions, the usage of self-test procedures is increasingly common and already widely adopted by companies in the automotive domain. The idea is to provide the system company using a certain device with a library of software procedures (known as self-test procedures) that are provided by the device producer and then integrated in the application code. Each of these procedures can be activated when required (e.g., at the power-on, or periodically), properly excite the module under test, look at the produced results and return a flag stating whether a fault has been detected or not. Since these procedures are developed by the semiconductor company delivering the device, the exact figure about the achieved Fault Coverage (FC) with respect to structural faults (e.g., stuck-at faults) can be computed. This approach is often known as *Software-based Self-test* (SBST) [1] and is today widely supported by many semiconductor and IP companies, such as Infineon [2], STMicroelectronics [3], Cypress [4], Renesas [5], Microchip [6] and ARM [7].

When considering ADAS devices, a similar approach can be followed. In this case, GPGPUs (or similar modules in terms of architecture and characteristics) are quite common. Since GPGPUs integrate many processing units (PUs), which share several microarchitectural modules with traditional CPUs (e.g., the register file and the ALU), the development of self-test procedures for these modules can leverage the techniques developed for CPUs. On the other side, special techniques are required to test some modules which are specific of GPGPUs, such as the thread scheduler. Such a module is in charge of storing and processing the information about the status of each thread, e.g., to trigger at each clock cycle the PUs associated to the active threads. Looking at the microarchitecture of this component, the most significant portion of this block is a memory array where this information is stored and continuously updated.

In some works, the authors introduced methods to increase the robustness [8] and some mitigation strategies [9] for some modules in GPGPU-based systems using a combination of high-level instrument programs and inline-assembly code. In other works [10], the authors proposed some first techniques to test the scheduler and the related memory. In this paper, we remove a major limitation of the work in [10], where single-cell faults in the memory were targeted, only. In particular, we propose a set of techniques which target a wide range of faults including both single-cell faults and coupling-faults involving multiple cells. The proposed techniques allow developing the high-level code performing the operations required to implement a desired March memory test algorithm. Although this paper focuses on the NVIDIA GPGPU architecture, the proposed techniques can be easily adapted to work on other architectures as well.

The paper is organized as follows: Section II introduces the background about the behavior, structure and operation of the scheduler's memory. Moreover, this section summarizes the test primitives (TPs) and lists the operational restrictions in the usage of the memory. Section III presents the proposed approach, the patterns for each targeted field, the test case algorithm and the general implementation of test primitives resorting to a high-level programing interface (CUDA-C). Section IV reports some experimental results, and Section V finally draws some conclusions.

[1] This work has been partially supported by the European Commission through the Horizon 2020 RESCUE-ETN project under grant 72232.

II. BACKGROUND

A. The SM scheduler warp controller

Following the NVIDIA terminology, the basic GPGPU architecture employs groups of identical PUs, called *Streaming Multiprocessors* (*SMs*), to compute instructions with high throughput. These modules are organized in groups called *Warps* [11] (32-48 consecutive threads) by the scheduler controllers (*SCs*), which are able to manage and control the GPGPU tasks. A warp SC is present inside each SM and manages the thread group distribution and execution [12]. This module includes one or more memories to store the information about the warp execution status in the SM.

In a real GPGPU, the SC memory is organized into addressable units called *Line-Entries* (LEs). Each LE stores the status of a warp. The SC assigns the total number of line-entries to be used according to the kernel configuration of the application. This controller employs a static organization to store the information of a warp in each LE.

The parameters stored in each LE include a warp ID field, a warp actual Program-Counter field (WPC), and a thread Active-Mask (TAM) field. The TAM field is composed of 32 or 48 bits, each bit represents the active (1) or inactive (0) state of the associated thread. The WPC field is composed of 32 or 35 bits. Recent implementations of the SC increase the number of stored parameters in order to store the information status of each thread [13]. During the initialization process, the scheduler configures each LE with the information concerning the threads active in each warp.

For the purpose of this work, we focus on detecting permanent faults in the TAM and WPC fields of the scheduler memory. This memory is written and read by the SC based on the control-flow instructions executed by the warp. At each instruction cycle, the memory is read in order to define the active threads in the warp and the instruction to be executed. The writing procedure is carried out once a new instruction is ready to be executed or the number of active threads is modified by divergence generation [14].

The SC memory has some operational restrictions, detailed in section C, and the access to the LEs cannot be performed employing the conventional methods for accessing data memories in processor-based systems. This means that special techniques are required to implement the required test primitives.

B. Test Primitives for scheduler memory access

From the very rich literature on memory testing we can easily derive the set of operations (denoted as *Fault Primitives*, or FPs) required to test different sets of functional faults that can affect both a single cell or couples of interfering cells in a memory. FPs have been used to define the most common class of memory test algorithms, i.e., March algorithms, able to test the memory by applying proper sequences of read and write operations with a complexity that grows linearly with the size of the memory. Interested readers may refer to [15] for a complete theoretical description of memory functional fault models. In this paper, we focus on how FPs and March algorithms can be translated into test programs for the scheduler memory.

1) Single cell static faults

Based on the set of primitives described in [15], Table 1 reports the full set of single-cell static fault primitives. The term "static" refers to the fact that they represent faults sensitized by a single memory operation. On each row, the Addressable Functional Fault Primitive for the scheduler, denoted as AFFP(SCH), includes the sequence of operations required to generate the input stimuli.

"A" denotes a test pattern writing in the target bit of the LE, and "$\overline{A}$" represents the complementary pattern value. In the stimulus, the initial conditions are in bold. The AFFP is organized as follow:

AFFP = <*Initial_Conditions, (Stimuli), Output_Fault_Value, Output_Fault-free_Value*>

Analyzing the AFFPs in Table 1, one can note that RDF and DRDF fault primitives share the same stimuli pattern and the total number of patterns is reduced by collapsing the similar stimuli patterns with the expected output value.

2) Coupling cell permanent faults

The coupling fault primitives are related to the interaction between two different cells; an aggressor cell *(a)* and a victim cell *(v)*. The considered fault primitives are presented in Table 2, where, X, Y and Z are logic values.

TABLE 1. STATIC FAULT PRIMITIVES FOR A SINGLE CELL

Fault	Fault model	FP	AFFP(SCH)
TF	Transition fault	$<\bar{A}\ W_{\bar{A}}/\bar{A}/->$ $<AW_{\bar{A}}/A/->$	$<\bar{A}, (W_{\bar{A}}, r_{\bar{A}}, W_A, r_A), \bar{A}/A>$ $<A, (W_A, r_A, W_{\bar{A}}, r_{\bar{A}}), A/\bar{A}>$
WDF	Write destructive fault	$<\bar{A}\ W_{\bar{A}}/A/->$ $<AW_A/\bar{A}/->$	$<\bar{A}, (W_{\bar{A}}, r_{\bar{A}}, W_{\bar{A}}, r_{\bar{A}}), A/\bar{A}>$ $<A, (W_A, r_A, W_A, r_A), \bar{A}/A>$
RDF	Read destructive fault	$<\bar{A}\ r_{\bar{A}}/A/A>$ $<Ar_A/\bar{A}/\bar{A}>$	$<\bar{A}, (W_{\bar{A}}, r_{\bar{A}}, r_{\bar{A}}), A/\bar{A}>$ $<A, (W_A, r_A, r_A), \bar{A}/A>$
DRDF	Deceptive RDF	$<\bar{A}\ r_{\bar{A}}/A/\bar{A}>$ $<Ar_A/\bar{A}/A>$	$<\bar{A}, (W_{\bar{A}}, r_{\bar{A}}, r_{\bar{A}}), A/\bar{A}>$ $<A, (W_A, r_A, r_A), \bar{A}/A>$

TABLE 2. STATIC FAULT PRIMITIVES FOR COUPLING CELLS

Fault	Fault model	FP	AFFP(SCH)
CFds	Disturb Coupling fault	$<X^a Z^v, w_y{}^a/\bar{Z}^v/->$	$<\bar{X}^a\bar{Z}^v, (\boldsymbol{W_{\bar{X}}{}^a, r_{\bar{X}}{}^a, W_{\bar{Z}}{}^v, r_{\bar{Z}}{}^v}, W_{\bar{X}}{}^a, r_{\bar{X}}{}^a, r_Z{}^v), Z^v, \bar{Z}^v>$ $<\bar{X}^a Z^v, (\boldsymbol{W_{\bar{X}}{}^a, r_{\bar{X}}{}^a, W_Z{}^v, r_Z{}^v}, W_{\bar{X}}{}^a, r_{\bar{X}}{}^a, r_Z{}^v), \bar{Z}^v, Z^v>$ $<X^a\bar{Z}^v, (\boldsymbol{W_X{}^a, r_X{}^a, W_{\bar{Z}}{}^v, r_{\bar{Z}}{}^v}, W_{\bar{X}}{}^a, r_{\bar{X}}{}^a, r_Z{}^v), Z^v, \bar{Z}^v>$ $<X^a Z^v, (\boldsymbol{W_X{}^a, r_X{}^a, W_Z{}^v, r_Z{}^v}, W_{\bar{X}}{}^a, r_{\bar{X}}{}^a, r_Z{}^v), \bar{Z}^v, Z^v>$ $<\bar{X}^a\bar{Z}^v, (\boldsymbol{W_{\bar{X}}{}^a, r_{\bar{X}}{}^a, W_{\bar{Z}}{}^v, r_{\bar{Z}}{}^v}, W_{\bar{X}}{}^a, r_{\bar{X}}{}^a, r_Z{}^v), Z^v, \bar{Z}^v>$ $<X^a Z^v, (\boldsymbol{W_X{}^a, r_X{}^a, W_Z{}^v, r_Z{}^v}, W_X{}^a, r_X{}^a, r_Z{}^v), \bar{Z}^v, Z^v>$
		$<X^a Y^v, r_x{}^a/\bar{Y}^v/->$	$<\bar{X}^a\bar{Y}^v, (\boldsymbol{W_{\bar{X}}{}^a, r_{\bar{X}}{}^a, W_{\bar{Y}}{}^v, r_{\bar{Y}}{}^v}, r_{\bar{X}}{}^a, r_Y{}^v), Y^v, \bar{Y}^v>$ $<\bar{X}^a Y^v, (\boldsymbol{W_{\bar{X}}{}^a, r_{\bar{X}}{}^a, W_Y{}^v, r_Y{}^v}, r_{\bar{X}}{}^a, r_Y{}^v), \bar{Y}^v, Y^v>$ $<X^a\bar{Y}^v, (\boldsymbol{W_X{}^a, r_X{}^a, W_{\bar{Y}}{}^v, r_{\bar{Y}}{}^v}, r_X{}^a, r_Y{}^v), Y^v, \bar{Y}^v>$ $<X^a Y^v, (\boldsymbol{W_X{}^a, r_X{}^a, W_Y{}^v, r_Y{}^v}, r_X{}^a, r_Y{}^v), \bar{Y}^v, Y^v>$
CFtr	Transition coupling fault	$<X^a 0, w_1{}^v/0^v/->$	$<\bar{X}^a 0^v, (\boldsymbol{W_{\bar{X}}{}^a, r_{\bar{X}}{}^a, W_0{}^v, r_0{}^v}, r_{\bar{X}}{}^v, r_0{}^v), 0^v, X^v>$ $<X^a 0^v, (\boldsymbol{W_X{}^a, r_X{}^a, W_0{}^v, r_0{}^v}, r_X{}^v, r_1{}^v), 0^v, X^v>$
		$<X^a 1, w_0{}^v/1^v/->$	$<\bar{X}^a 1^v, (\boldsymbol{W_{\bar{X}}{}^a, r_{\bar{X}}{}^a, W_1{}^v, r_1{}^v}, r_{\bar{X}}{}^v, r_0{}^v), 1^v, \bar{X}^v>$ $<X^a 1, (\boldsymbol{W_X{}^a, r_X{}^a, W_1{}^v, r_1{}^v}, r_{\bar{X}}{}^v, r_1{}^v), 1^v, \bar{X}^v>$
CFwd	Write destructive coupling fault	$<X^a Y^v, w_y{}^v/\bar{Y}^v/->$	$<\bar{X}^a\bar{Y}^v, (\boldsymbol{W_{\bar{X}}{}^a, r_{\bar{X}}{}^a, W_{\bar{Y}}{}^v, r_{\bar{Y}}{}^v}, W_{\bar{Y}}{}^v, r_{\bar{Y}}{}^v), Y^v, \bar{Y}^v>$ $<X^a\bar{Y}^v, (\boldsymbol{W_X{}^a, r_X{}^a, W_{\bar{Y}}{}^v, r_{\bar{Y}}{}^v}, W_{\bar{Y}}{}^v, r_{\bar{X}}{}^v), Y^v, \bar{Y}^v>$ $<\bar{X}^a Y^v, (\boldsymbol{W_{\bar{X}}{}^a, r_{\bar{X}}{}^a, W_Y{}^v, r_Y{}^v}, W_{\bar{Y}}{}^v, r_Y{}^v), \bar{Y}^v, X^v>$ $<X^a Y^v, (\boldsymbol{W_X{}^a, r_X{}^a, W_Y{}^v, r_Y{}^v}, W_{\bar{Y}}{}^v, r_Y{}^v), \bar{Y}^v, X^v>$
CFrd	Read destructive coupling fault	$<X^a Y^v, r_y{}^v/\bar{Y}^v/\bar{Y}^v>$	$<\bar{X}^a\bar{Y}^v, (\boldsymbol{W_{\bar{X}}{}^a, r_{\bar{X}}{}^a, W_{\bar{Y}}{}^v, r_{\bar{Y}}{}^v}, r_{\bar{Y}}{}^v, r_{\bar{Y}}{}^v), Y^v, \bar{Y}^v>$ $<X^a\bar{Y}^v, (\boldsymbol{W_X{}^a, r_X{}^a, W_{\bar{Y}}{}^v, r_{\bar{Y}}{}^v}, r_{\bar{Y}}{}^v, r_{\bar{Y}}{}^v), Y^v, \bar{Y}^v>$ $<\bar{X}^a Y^v, (\boldsymbol{W_{\bar{X}}{}^a, r_{\bar{X}}{}^a, W_Y{}^v, r_Y{}^v}, r_Y{}^v, r_Y{}^v), \bar{Y}^v, Y^v>$ $<X^a Y^v, (\boldsymbol{W_X{}^a, r_X{}^a, W_Y{}^v, r_Y{}^v}, r_Y{}^v, r_Y{}^v), \bar{Y}^v, Y^v>$
CFir	Incorrect read coupling fault	$<X^a Y^v, r_y{}^v/Y^v/\bar{Y}^v>$	$<\bar{X}^a\bar{Y}^v, (\boldsymbol{W_{\bar{X}}{}^a, r_{\bar{X}}{}^a, W_{\bar{Y}}{}^v, r_{\bar{Y}}{}^v}, r_{\bar{Y}}{}^v), Y^v, \bar{Y}^v>$ $<X^a\bar{Y}^v, (\boldsymbol{W_X{}^a, r_X{}^a, W_{\bar{Y}}{}^v, r_{\bar{Y}}{}^v}, r_{\bar{Y}}{}^v), Y^v, \bar{Y}^v>$ $<\bar{X}^a Y^v, (\boldsymbol{W_{\bar{X}}{}^a, r_{\bar{X}}{}^a, W_Y{}^v, r_Y{}^v}, r_Y{}^v), \bar{Y}^v, Y^v>$ $<X^a Y^v, (\boldsymbol{W_X{}^a, r_X{}^a, W_Y{}^v, r_Y{}^v}, r_Y{}^v), \bar{Y}^v, Y^v>$
CFdrd	Deceptive read destructive CF	$<X^a Y^v, r_y{}^v/\bar{Y}^v/Y^v>$	$<\bar{X}^a\bar{Y}^v, (\boldsymbol{W_{\bar{X}}{}^a, r_{\bar{X}}{}^a, W_{\bar{Y}}{}^v, r_{\bar{Y}}{}^v}, r_{\bar{Y}}{}^v, r_{\bar{Y}}{}^v), Y^v, \bar{Y}^v>$ $<X^a\bar{Y}^v, (\boldsymbol{W_X{}^a, r_X{}^a, W_{\bar{Y}}{}^v, r_{\bar{Y}}{}^v}, r_{\bar{Y}}{}^v, r_{\bar{Y}}{}^v), Y^v, \bar{Y}^v>$ $<\bar{X}^a Y^v, (\boldsymbol{W_{\bar{X}}{}^a, r_{\bar{X}}{}^a, W_Y{}^v, r_Y{}^v}, r_Y{}^v, r_Y{}^v), \bar{Y}^v, Y^v>$ $<X^a Y^v, (\boldsymbol{W_X{}^a, r_X{}^a, W_Y{}^v, r_Y{}^v}, r_Y{}^v, r_Y{}^v), \bar{Y}^v, Y^v>$

The State coupling faults (CFst) FP $<\overline{A}; \overline{V}/V/-, \overline{A}; V/\overline{V}/-, A; V/\overline{V}/-, A; \overline{V}/V->$ in Table 1, and the state faults (SF) FP ($<\overline{A}/A/-, A/\overline{A}/->$) and the incorrect read faults primitive (IRF) FP ($<\overline{A}, r_{\overline{A}}/\overline{A}/A, A, r_A/A/\overline{A}>$) in Table 2, are neglected by the proposed approach, due to lack of GPGPU instructions, available functions or software-based methods to verify the behavior of the corresponding faults.

Although the AFFP(SCH)s for the coupling faults are different, some associated Sensitizing Operation Sequences (SOSs) are similar and it is possible to collapse identical patterns. In this approach, the total number of patterns to cover the FPs is reduced to 30. Some coupling faults (CFrd, CFir and CFdrd) can be grouped with the same SOS, since the only difference among them is one additional read operation. Therefore, the SOS with the lowest number of reading operations can be neglected and the CFdrd SOS is employed to sensitize those coupling faults.

Each AFFP(SCH) pattern should be adapted to include the scheduler memory operational restrictions.

C. Scheduler Memory operational restrictions

The SC memory cannot undergo any possible operation or sequence of operations. In particular, the following operational restrictions exist:

1. During the device configuration and program starting phase, it is not possible to write whichever initial state in the TAM field. By default, all threads start in the active state (*value 1 for all bits*).
2. Once a warp ends the execution of one instruction, the LE is actualized and one read operation is implicitly performed. On the other hand, a write operation is performed when a new instruction starts its execution or when the number of active threads is modified.
3. Any pattern applied to the TAM field can generate thread divergence. This divergence causes the execution of two paths (*Taken* and N*ot-Taken*). Moreover, these paths are consecutively executed.
4. Once a warp is dispatched to the SM, the execution of the warp path cannot be stopped. Moreover, if thread synchronization mechanisms are employed (i.e., *__syncthreads()*), one or more threads must be maintained in active state in the TAM field in order to keep the warp active. In contrast, a TAM field with all inactive threads represents a terminated warp.
5. The scheduler includes two warp dispatchers, which manage the scheduling of the warps in the SM. These units submit the warps to the SM based on performance considerations, by means of a pseudo-random algorithm and complex data-hazards control methods. However, the warp submission is not executed in order. For the purpose of this work, this behavior may compromise the execution of fault primitives, which require operations on consecutive LEs.

III. PROPOSED METHOD

We propose a method to detect permanent and coupling faults in the line-entries of the SM scheduler memory based on a functional approach and employing high-level programing language functions (CUDA-C). The method employs a mapping between a set of fault primitives and GPGPU functions to generate a test program. This approach is able to provide the rules to transform any March algorithm into a GPGPU test program generating the same sequence of operations on the target memory and thus detecting the same defects.

The proposed approach divides each March operation in one or a set of kernels. This division allows executing the test program during the idle intervals of in-field operations including the set of restrictions presented by the scheduler memory operation. The kernel test program is stored in the system memory aside from the application kernels. However, the host must activate the test sequences. In contrast, test results are stored in selected free locations in the global memory.

The first and second restrictions cannot be avoided. This means that all test patterns applied to the TAM field start with an initial state of all threads active (*all bits in 1*) and an initialization pattern is required.

According to the third restriction, the path divergence generation by a control-flow instruction cannot be stopped once it is executed. The selection of a low number of control-flow functions and operations on each path can guarantee a low execution time. A predefined set of external parameters (see Table 3) is used to divide the operations of a test program into small parts. These parameters increase the detection of coupling faults among and inside line entries. Concerning the fourth restriction, those parameters must guarantee that at least one bit (*thread*) remains at the value 1 (*active*). This is the reason for the missing pattern with all 0s.

TABLE 3. PATTERNS TO DETECT A COUPLING FAULT IN AN LINE-ENTRY

Pattern	Description
11111111…00000000… / 00000000…11111111…	First half X, second half $\bar{X}$
00001111...00001111… / 11110000...11110000…	First four bits X, second four $\bar{X}$
00110011…00110011… / 11001100…11001100…	First two bits X, second four $\bar{X}$
10101010…10101010… / 01010101…01010101…	Alternated X and $\bar{X}$
11111111111111111…	All in ones

During the evaluation of the divergence not-taken path, the third restriction arises. Once the threads finish the taken path, the SC changes the status of the inactive threads and these become active. An inverse write operation is performed in this field at the start of the not-taken path execution as an effect of the previous change. This behavior can be skipped and does not generate issues in the adaptation of March operations.

The scheduler modifies the full content of the TAM and the WPC fields. Nevertheless, the writing process is different for each field. This writing is based on conditional and unconditional control-flow operations on each case. In most GPGPUs, the divergence paths, taken and not-taken, must be executed in different operation cycles. The proposed approach employs only the taken path, neglecting the inverse writing operation described below.

The fifth restriction relates to the dispatcher units execution. Considering that details of the dispatchers operation and the way to predict the warps emission during the execution are not provided in detail by the manufacturer [11, 13], the proposed method employs thread synchronization functions (*barrier instructions*) and semaphore variables to detect coupling faults between consecutive cells. Moreover, this technique is able to skip the dispatchers operation controlling the injection order. This method stops temporarily the path execution of a warp and launches a desired and ready warp. It is worth noting that there is a possibility of non-expected consecutive read procedures on the same LE. Nevertheless, these additional reads do not generate issues in the March algorithm adaptation and execution.

The next sections describe the steps to generate write and read operations in the TAM and WPC fields.

A. Patterns for the TAM field

The following sequence of operations is required to generate a write operation in the TAM field in a LE. This approach is also effective for some coupling faults between two consecutive cells (*CFrd*, *CFir* and *CFdrd*).

1) Patterns generation in the TAM field for coupling fault in a single cell

1. Execute an embarrassingly parallel function **F** (*Initial condition*).
2. Execute a divergence generation function (*writing operation in the target bit(s) in the TAM field*).
3. Execute the taken path (*read operation in the full TAM field*).
4. Execute the not-taken path (*generate a write operation and read operations with the inverted value on selected bit(s) field*).
5. Convergence point execution **CP** (*parallel execution of instructions with implicitly read operations*).

The step 4 is skipped and is considered as an interval condition to start other March operations. This is the basic writing procedure to the LE. The *CFrd*, *CFir* and *CFdrd* coupling faults can be satisfactorily evaluated using both divergence paths, due to the total number of write and read

operations involved. However, additional elements must be included in order to detect a large number of coupling faults.

2) Patterns generation in the TAM field for coupling faults in multiple cells

The detection of coupling faults between consecutive cells (LEs) requires additional steps including thread synchronization, warp selection and nesting divergence in order to assure correct evaluation of each potential fault cell.

The following steps describe the sequence of operations to generate the coupling faults detection:

1. **F** (*Initial condition*).
2. Select a target warp or LE ((*a) cell*).
3. Execute a first divergence generation function (*write operation in the target bit(s) field of an (a) cell*).
4. Execute the taken path of divergence (*read the full TAM field for (a) cell*).
5. Execute a barrier function in the *(a)* cell path, which stops the warp execution and launches a different warp.
6. Select a new target warp (*(v) cell*).
7. Execute a second divergence generation function (*write in the target bit(s) of a (v) cell*).
8. Execute the taken path for the second divergence (*It generates reading operations of the full TAM field in the (v) cell*).
9. Execute barrier function in the *(v)* cell path, launching a new warp.
10. Execution of the not-taken path for the *(a)* cell (*write the opposite value in the target bit(s) in TAM field, followed by reading operations*).
11. Execute a barrier function in the *(a)* cell path.
12. Execution of the not-taken path for the *(v)* cell (*write with the opposed value in the target bit(s) in the TAM field, followed by read operations*).
13. Execute a barrier function in the *(v)* cell path.
14. **CP**

The steps 9-13 are employed to evaluate more coupling faults in the TAM field (*CFds*, *CFtr*). However, these steps can be skipped with a correct selection of the input stimuli patterns.

A nested divergence is included to successfully keep the warp execution and the need of at least one active thread in the TAM field. This behavior is required to detect coupling faults of the group (*CFrd*, *CFir*). This additional divergence can be placed between steps 4 and 5 or between the steps 10 and 11 for the *(a)* cell. The same approach is used between steps 8 and 9 for the victim cell.

The *CFwd* faults are detected by the introduction of a third nested divergence. This function is placed in the *(a)* cell path after the second divergence in the not-taken path. This is used to generate the evaluation of missing threads and guarantee the writing procedure in the scheduler memory during the execution.

This divergence is designed in a way that the first thread or the last thread is selected on each path, to generate a required stimulus. Additional barrier instructions are added in order to keep the synchronization of the warp execution. Moreover, the step 14 is removed and replaced by the steps 4, 5, 10 and 11 to launch a new aggressor operation.

An external pattern is applied to the *(a)* cell in the first divergence to select the number of threads active during the victim evaluation. The divergence evaluation in the *(v)* cell generates the equal division in two groups of threads. However, in order to cover all the coupling faults, both division cases should be evaluated.

In this approach, the reading procedures are implicitly integrated in the writing procedure. The simplest case of a reading procedure can be performed by the execution of non-control flow instructions by the warp. The reading procedure is applied for both fields on each instruction cycle and cannot be avoided or stopped. The SC reads an active LE continuously at the starting and ending points of each instruction cycle in order to preserve system coherency in thread execution. The inactive or halted line-entries are read when those become active.

The previously proposed steps can be applied for any consecutive configuration of *(a)* and *(v)* cell. In order to generate displacements on any direction (*increment or decrement*) across the memory LE in a desired order, the dispatcher unit is skipped employing the barrier instructions and shared variables described below.

B. Patterns for the WPC field

The evaluation of WPC field instead requires a sequence of steps to access the system memory in the GPGPU. The proposed method considers GPGPU architectures with a shared WPC field among the threads in a warp. For the purpose of this work, the kernel design and execution must be executed as parallel as possible. In order to generate the required stimulus for the field, some subroutines (*functions*) are placed in strategic locations in the memory. The main kernel accesses each subroutine employing unconditional control-flow operations. Inside the subroutine, a set of barrier functions are employed to halt the warp execution and start a different warp. The approach, employed to generate the warp selection in the TAM field, is also used for this field.

Some locations, in the highest part of the system memory, present difficulties for function placement. The solution is to include additional GPGPU kernels in the main memory. Those kernels and functions are placed in the GPGPU system memory during the compilation process. The location placement uses the same patterns presented in Table 3.

The test program can be divided in pieces employing independent test kernels to apply specific patterns. Mainly, each subroutine is composed of embarrassingly parallel functions and thread barriers.

1) Patterns generation in the WPC field for coupling faults in a single cell

The following list presents the basic steps to generate a single writing procedure, and implicit reading procedures, on the actual WPC field.

1. Execution of an embarrassingly parallel instruction **P** (*reading ops, initial condition*).
2. Execution of an unconditional flow-control function (calling a function stored in the GPGPU system memory at a predefined location) (*Writing on the target bit field and reading procedures*).
3. Return from the subroutine, and then compare the signature. Start a new call for another subroutine in other memory placement (*Writing the bit field and reading procedures*).
4. (*Repetition of the step 2 and 3 if required*).

As discussed for the TAM field, the reading procedures are integrated in the writing sequence. For each method, a thread per signature (*d_signature*) is included as observation mechanism to detect any misbehavior in the memory cells. Thus, a mismatch in the final signature will indicate a fault present in the memory cell. The signature is evaluated at the end or in the middle of each pattern evaluation.

2) Patterns generation in the WPC field for coupling faults in multiple cells

The detection of coupling faults between consecutive LEs requires the use of additional steps. The aggressor and the victim are carefully chosen by a warp selection process that generates divergence. The following steps are used to generate the coupling fault evaluation.

1. **P** (*initial condition*).
2. Selection of an *(a)* cell and execution of an unconditional flow-control function (*write procedure in the target LE, (a) cell*).
3. Execution of the subroutine path and generation of a barrier function in the *(a)* cell path, launching a new warp (*reading procedure in the target LE, (a) cell*).
4. Selection of *(v)* cell and execution of the unconditional flow-control function (*write procedure in the target LE, (v) cell*).
5. Execution of the subroutine path and generation of a barrier function in the *(v)* cell path, launching a new warp.
6. Return to the main kernel path and execution of embarrassingly parallel operations (*writing operation in the returning stage, reads in the embarrassing execution*).

The implementation of the previous steps allows detection of coupling faults of the group (*CFds*, *CFtr*). Other steps must be included in order to detect *CFir* coupling faults. These faults require an additional step after step 5. This step requires the execution of more functions inside the subroutine in order to generate the implicit readings in the memory LE. An additional barrier function is added to generate the stimulus in the aggressor and the victim. Testing *CFwd* coupling faults requires a new aggressor subroutine or writing procedure, which is included after step 6.

C. Test cases

We selected two March algorithms (MATS+ and MATS++) as test cases to demonstrate the characteristics of the proposed approach and generated a set of test programs for the two fields for each LE in the scheduler memory.

1) MATS+ and MATS++ algorithm.

The MATS+ algorithm is composed of the operations presented in Table 4 (the reader may refer to [9] for details on the March test notation). To apply the March test in the scheduler memory some implicit read and initialization steps are added, see Table 4.

TABLE 4. TEST PROGRAM FOR MATS+ AND (*) MATS++

Original Ops.	Adapted Ops. to scheduler memory
M1: ↕ ($W_{(0)}$)	***Init. Steps*** **(**$W_{(x)}$, $R_{(x)}$**);** **M1:** ↕ ($W_{(0)}$, $R_{(0)}$**);**
M2: ⇑ ($R_{(0)}$, $W_{(1)}$)	***Init. Steps*** **(**$W_{(x)}$, $R_{(x)}$**);** **M2:** ⇑ ($R_{(0)}$, $W_{(1)}$, $R_{(1)}$**);**
M3: ⇓ ($R_{(1)}$, $W_{(0)}$, $R_{(0)}$*)	***Init. Steps*** **(**$W_{(x)}$, $R_{(x)}$**);** **M3:** ⇓ ($R_{(1)}$, $W_{(0)}$, $R_{(0)}$**);**

In Table 4, in bold we reported the additional steps required to generate the expected March operations. The initialization steps, listed below, are required to select a specific warp (LE) and skip the dispatchers operation. This process is applied to each evaluated field in the LE.

For the implementation, a kernel is designed for each operation (Basic Block Kernel or BBK). The BBKs require, as input parameters, the signature location and the external stimuli. The external stimuli is only required for TAM field evaluation. Then, it is divided in independent test program chunks. This is important during in-field test, since the test must often be executed during the idle slots of the system. Table 5 presents the adaptation of the original MATS+ algorithm to a set of kernels for one pattern evaluation.

From Table 5 we can observe that the external parameter applied on each kernel is not the same. In the first and third kernels, the same pattern is applied. However, the second kernel requires the inverted external pattern to stimulate the target bit fields with the operation. The previous kernel sequence should be applied to each pattern in the Table 1. At the end, the same approach is applied eight times.

TABLE 5. MARCH OPERATION AS A SEQUENCE OF CUDA KERNEL EXECUTIONS (ADAPTATION PRESENTED FOR ONE PARAMETER FOR THE TAM FIELD)

Original March operations (*MATS+*)	Adapted March Operations	Equivalent CUDA kernel
↕ ($W_{(0)}$)	↕ ($W_{(0)}$,$R_{(0)}$)	**Test_kernel_decrement**<<<*TOTAL_BLOCKS*, *TOTAL_THREADS*>>> (TOTAL_THREADS, vector_params[0], d_signature);
⇑ ($R_{(0)}$, $W_{(1)}$)	⇑ ($W_{(0)}$, $R_{(0)}$, $W_{(1)}$, $R_{(1)}$)	**Test_kernel_decrement_x**<<<*TOTAL_BLOCKS*, *TOTAL_THREADS*>>> (TOTAL_THREADS, vector_params[1], d_signature);
⇓ ($R_{(1)}$, $W_{(0)}$)	⇓ ($R_{(1)}$, $W_{(0)}$, $R_{(0)}$)	**Test_kernel_increment**<<<*TOTAL_BLOCKS*, *TOTAL_THREADS*>>> (TOTAL_THREADS), vector_params[0], d_signature);

2) Algorithm implementation for coupling faults in CUDA.

The pseudo-code reported in Fig 1 represents the general CUDA implementation of the kernel to evaluate coupling faults among consecutive cells. This kernel is launched concurrently by the aggressor and victim warps (*line-entries*). A warp selection function is used to divide the *(a)* and *(v)* cells and to execute the kernel in a sequential fashion.

__global__ void Test_kernel_decrement_x (int* *divergence_parameters*, **int*** *signature* …)	
{	
Parameter_initialization();	►Initialization of Local and Shared variables.
Thread_warp_size_verification_correction(); (‡)	►Warp number resize (*Total Thread number is not multiple of 32*).
For warp **in** kernel **do:** (‡)	►Search each Warp ID
If *Warp_Selected*() **then:** (‡)	►Select a Warp ID in order (*Increment* / *decrement*)
Load_divergence_parameters(); (‡)	►External pattern to be used in *(a)*.
If warp **is** *Aggressor* **then:** (‡)	►Check if warp ID is *(a)*.
Aggressor_warp_enabled(); **	►Check if *(a)* cell has associated *(v)* cell.
If divergence_parameter **is** '0' **then:** (†)	►First divergence function (Not-Taken Path)
Signature_evaluation();	►Signature evaluation and $R_{(x)}$ operation.
Barrier_operation();	►Warp execution Halt.
Else: (†)	►First divergence function (Taken Path)
Signature_evaluation();	► Signature evaluation and $R_{(x)}$ operation.
For Warp_Id > 0 **do:**	►Check if *(v)* cell has been executed
If *divergence_parameter*(0) **is** '1' **then:** (†)	►Nesting (Second) divergence function
Signature_evaluation();*Barrier_operation*();	► Signature evaluation and $R_{(x)}$ operation.
Else if *divergence_parameter*(31) **is** '1' **then:** (†)	►Nesting (Second) divergence function
Signature_evaluation();*Barrier_operation*();	► Signature evaluation and $R_{(x)}$ operation.
Else: (†)	►Nesting (Second) divergence function
Signature_evaluation();*Barrier_operation*();	► Signature evaluation and $R_{(x)}$ operation.
Signature_evaluation();	►Implicit Read in one instruction cycle.
Warp_ID --;	►Decrement in Warp ID value.
Else if warp **is** *Victim* **then:** (‡)	►Check if warp ID is *(v)*.
victim_warp_enabled(); **	►Check if *(v)* has associated *(a)* cell.
If threads **in** warp ('<16' / ***'>15'***) **then:** (†)	► Divide the threads in lower or **higher** part
Signature_evaluation(); *Barrier_operation*();	► Signature evaluation and $R_{(x)}$ operation.
Else: (†)	►
Signature_evaluation();	► Signature evaluation and $R_{(x)}$ operation.
For Warp_Id > 0 **do:**	► Check if *(a)* cell has been executed
Barrier_operation();	► Warp execution Halt.
Barrier_operation();	► Warp execution Halt.
Else: (‡)	►Warp is not selected to be launched.
Signature_evaluation(); *Barrier_operation*();	►Implicit $R_{(x)}$, Warp execution stops.
Warp_synchronization(); (‡)	►Final warp synchronization.
Clear_Parameters(); (‡)	►Clearing of parameters in shared memory.
}	

FIG 1. CUDA PSEUDO-CODE OF THE KERNEL TEST IMPLEMENTATION FOR COUPLING FAULTS IN CONSECUTIVE CELLS. (‡) FUNCTIONS TO SKIP DISPATCHERS. (**) OPTIONAL FUNCTIONS FOR EDGE LINE-ENTRIES EVALUATION. (†) DIVERGENCE-GENERATION FUNCTIONS FOR THE *(a)* AND *(v)* LES.

According to warp selection, each *(a)* or *(v)* cell executes different paths and internally uses divergence functions to stimulate the field. In the aggressor path, a second divergence path is used to evaluate additional coupling faults that require a *W(0)* starting condition. The barrier operations are used to stop the warp (*cell*) execution and generate the launch of a new warp. The previous kernel algorithm can be simplified, avoiding the second divergence in order to test permanent, static and some coupling faults. As explained below, a set of variables are added to each path in order to skip the execution of the dispatchers. Each variable was attached to a warp execution.

IV. EXPERIMENTAL RESULTS

The experimental results are obtained employing a NVIDIA© GeForce GTX 960M GPGPU with 1.176 GHz of clock rate and 32 threads per warp. In order to check the kernel execution and establish the performance metrics, such as

execution time, and GPGPU resource overhead by the test programs we used the NVIDIA© Nsight™ 5.6 tool. Moreover, we employed the NVIDIA© Visual profiler to determine the total number of instructions executed by the test kernel. Table 6 presents the performance characteristics of the proposed test programs for different line entries sizes in the memory. Additionally, the table reports the required idle times to apply the test sequences. Each test program is designed to use one block over one SM in the GPGPU.

TABLE 6. TEST PROGRAM CHARACTERISTICS FOR THE PROPOSED APPROACH TO EVALUATE DIFFERENT LINE-ENTRY SIZES. (*)ACTIVE KERNEL FUNCTIONS ONLY.

		FIELD					
		TAM			WPC		
Line entry size		32	16	8	32	16	8
BBK Execution performance (uS)		778.987	359.297	168.3	573.419	187.791	91.517
Total Execution performance (mS)		18.69	8.62	4.039	13.86	4.507	2.196
Instructions executed	*BBK Kernel increment (KB)*	276.248	78.28	8.428	186.48	50.36	14.46
	BBK Kernel decrement (KB)	276.616	78.184	8.404	436.128	73.488	15.576
	Per Pattern (KB)	829.112	234.744	25.260	809.088	174.208	44.496
	Total (MB)	6.633	1.878	0.202	6.473	1.397	0.356
GPGPU overhead	*System memory (KB)*	1.84	1.84	1.84	2.424*	2.424*	2.424*
	Shared memory (B)	260.0	132.0	68.0	4.0	4.0	4.0

As shown by the results, the total time, required by test execution, remains in the range of some µs. Moreover, the system memory requirements are low for the test of one pattern for 32 line entries (1.84KB and 2.4KB for TAM and WPC fields, respectively). However, the WPC test program requires the additional placement of inactive kernels to place selected subroutines in specific memory locations. This test program requires using all the system memory. Hence, the execution of WPC test kernels during in-field test should be limited to Switch-On/Switch-Off intervals.

The number of shared elements required is low in comparison with the system memory overhead to store the kernels. The total number of registers required for each configuration remains constant to 28 in the TAM field kernels. For WPC field, this value is constant to 37.

The number of instructions to apply a test pattern (3 kernels) in the memory is relatively high (829 and 809KB). This can be explained by the use of high-level programing platforms (CUDA-C) functions and the complexity for managing and avoiding the dispatcher units. As it can be observed in Fig 1, six functions are required to control the warp execution and to skip the dispatchers. Additionally, the evaluation of coupling faults between consecutive cells requires conditional evaluations in the divergence function generation adding more instructions to the final test program. Nevertheless, the BBK size is low (<300KB) and employs less than 800µS during its execution. On the one hand, the TAM kernels employ a number of shared memory elements increasing linearly with the number of line-entries to be evaluated. On the other hand, the WPC kernels use the same number of shared variables for every memory size. These kernels are simpler than TAM kernels and the total number of memory elements required is employed in the selection and launch of warps.

An analysis with the NVIDIA Visual Profiler of the TAM test kernels with different LE size shows that these kernels employ 0% of concurrency execution. This behavior is caused by the need to apply ordered patterns in selected line-entries. Moreover, the same tool shows that these kernels spend most of the execution time (≈60%) in thread synchronization functions or halt state. Nevertheless, these functions are required to avoid the action of the dispatcher units. The WPC kernels present the same behavior; nevertheless, this value is close to 50%.

We used the memory fault simulator introduced in [16] to check the FC of the proposed test programs. Each kernel was instrumented with functions to log the operations executed at each stage of the kernel execution. This information was employed to generate the input files to the fault simulator. Table 7 presents the obtained results.

TABLE 7. FC OF THE MATS++ TEST PROGRAM FOR 32 LINE ENTRIES. (*) FAULT PRIMITIVES INITIALLY NOT CONSIDERED IN THE PROPOSED APPROACH.

Fault primitive	MATS+ Algorithm	
	TAM Field FC(%)	WPC Field FC(%)
SF_X	**100***	**100***
TF_X, WDF_X, RDF_X, DRDF_X	100	100
CFst_X	**100***	**100***
CFir_X, CFds_X, CFtr_X	100	100
CFwd_X, CFrd_X, CFdrd_X	100	100
DRF	0*	0*
CFid_X	**100***	**100***

According to the results, the proposed method is effective to test static single and coupling faults in the scheduler memory. Although the original test programs development does not consider state faults (*SF*), state coupling faults (*CFst*) and inversion coupling faults (*CFid*), the fault simulation results show that these faults are also covered in the kernel implementation.

V. CONCLUSIONS

We proposed a functional approach for developing Self-test procedures to be used for in-field test of static and coupling faults in the scheduler memory of GPGPU devices. The method was developed and implemented using high-level programing platform (CUDA-C) and microarchitectural information, only. Results on some representative test cases show that the proposed method is effective and can test all the fault primitives, thus guaranteeing a complete coverage of all static faults in the scheduler memory.

REFERENCES

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

[2] Infineon. (2018). *https://*www.hitex.com/software-components/selftest-libraries-safety-libs/pro-sil-safetcore-safetlib/.

[3] STMicroelecronics, "AN3307 Application note Guidelines for obtaining IEC 60335 Class B certification for any STM32 application," 2016.

[4] Cypress, "AN204377 FM3 and FM4 Family, IEC61508 SIL2 Self-Test Library," 2017.

[5] Renesas. (2018). *[https://*www.renesas.com/en-eu/products/synergy/software/add-ons.html#read.

[6] Microchip, "DS52076A 16-bit CPU Self-Test Library User's Guide," 2012, p. 52.

[7] ARM. (2018). *https://developer.arm.com/technologies/functional-safety*.

[8] S. Di Carlo, et al., "Increasing the robustness of CUDA Fermi GPU-based systems," *IEEE 19th International On-Line Testing Symposium (IOLTS),* 2013, pp. 234-235.

[9] S. Di Carlo et al., "Fault mitigation strategies for CUDA GPUs," *IEEE International Test Conference (ITC),* 2013, pp. 1-8.

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

[11] F. NVidia, "Nvidia's next generation CUDA compute architecture," *NVidia, USA,* 2009.

[12] J. Nickolls and W. J. Dally, "The GPU computing era," *IEEE Micro,* vol. 30, 2010.

[13] T. NVIDIA, "V100 GPU architecture. the world's most advanced data center GPU. Version WP-08608-001_v1. 1," *NVIDIA. Aug,* p. 108, 2017.

[14] W. W. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic warp formation and scheduling for efficient GPU control flow," *40th Annual IEEE/ACM International Symposium on Microarchitecture*, 2007, pp. 407-420.

[15] S. Di Carlo and P. Prinetto, "Models in Memory Testing" from *Models in Hardware Testing: Lecture Notes of the Forum in Honor of Christian Landrault*: Springer, 2009.

[16] A. Benso, S. Di Carlo, G. Di Natale, and P. Prinetto, "Specification and design of a new memory fault simulator," *11th IEEE Asian Test Symposium,* 2002, pp. 92-97.