About the functional test of the GPGPU scheduler

B. Du*, Josie E. Rodriguez Condia†, M. Sonza Reorda‡, L. Sterpone§
Politecnico di Torino
Torino, Italy
{*boyang.du, †josie.rodriguez, ‡matteo.sonzareorda, §luca.sterpone}@polito.it

Abstract[1]—**General Purpose Graphical Processing Units (GPGPUs) are increasingly used in safety critical applications such as the automotive ones. Hence, techniques are required to test them during the operational phase with respect to possible permanent faults arising when the device is already deployed in the field. Functional tests adopting Software-based Self-test (SBST) are an effective solution since they provide benefits in terms of intrusiveness, flexibility and test duration. While the development of the functional test code addressing the several computational cores composing a GPGPU can be done resorting to known methods developed for CPUs, for other modules which are typical of a GPGPU we still miss effective solutions. This paper focuses on one of the most relevant module consists on the scheduler core which is in charge of managing different scalar computational cores and the different executed threads. At first, we propose a method for evaluating the fault coverage that can be achieved using an application program. Then, we provide some guidelines for improving the achieved fault coverage. Experimental results are provided on an open-source VHDL model of a GPGPU.**

Keywords—***GPGPU Scheduler, SBST, functional testing.***

I. Introduction

General Purpose Graphical Processing Units (GPGPUs) were introduced in the last decades to accelerate graphic tasks in different kinds of computer systems, with special emphasis on gaming applications. More recently, GPGPUs found wide application on mobile devices for other tasks, such as face recognition and video processing. Today, GPGPUs are entering into areas, where high performance requirements are combined with safety requirements. A major example lies in the so called *advanced driver-assistance systems* (ADAS) in the automotive area, including systems devoted to Automatic Cruise Control, Pedestrian Recognition and Protection, Forward Collision Warning and Automatic Parking. The adoption of ADAS is also considered as an intermediate step towards the development of autonomous and semi-autonomous cars. All these ADAS systems are based on sensors such as cameras, radars or lidars; producing a sustained flow of data that must be processed and produce decisions in real-time. GPGPUs are very well suited for this data fusion task and are thus being increasingly considered. However, since any failure by these systems may provoke serious consequences, effective solutions to first assess and then mitigate faults arising within GPGPUs became a hot topic, as mandated by regulations and standards, such as ISO 26262.

Several works [1][2] already investigated the sensitivity of GPGPUs to radiation effects, which are a major cause for transient faults. These works are mainly based on exposing real GPGPU devices to accelerated radiation fluxes, and then observing the resulting effects. In this way it is possible to estimate both the probability of a single fault to arise, and the one that a fault may produce a critical failure, despite the mitigating mechanisms possibly implemented by the system. Moving to permanent faults, achieving the required safety targets clearly mandates the adoption of special techniques to minimize the chances that possible faults created by the manufacturing process or by other mechanisms (e.g., aging) escape the different test procedures applied at the device, board and system level. Moreover, given the very high safety targets required, the advanced semiconductor technology used to manufacture current GPGPU devices, and the relatively long lifetime of ADAS systems, it is mandatory to develop efficient techniques to detect permanent faults before they cause critical failures (*in-field test*).

In all cases a test must be performed, which should be able to detect a very high percentage of those faults that may cause critical failures. In both cases, the test must be relatively fast, and must be performed taking into account the environment where the target device works, with minimum impact on the rest of the system.

Due to these constraints, Software-based Self-test (SBST) has been widely adopted for in-field test [3], following the guidelines provided by research work in the area [6]. The development of SBST test procedures can sometimes be done by abstracting detailed structural information, when the function and architecture of the target module is known (e.g., for caches, or branch prediction units). In other cases, it is based on these information for both developing the SBST code, and for assessing the achieved fault coverage.

In the case of GPGPUs, the development of effective SBST test procedures can be split in two parts

- When targeting the computational cores of the GPGPU, one can use the techniques developed for traditional CPUs [5]. The work in [4] is an example of what can be done in this direction.
- More efforts are required to target those modules, which are specific of the GPGPU architecture. As an example, testing the scheduler existing in any GPGPU, as well as the interface between the scheduler and the computational cores, requires specific algorithms able to excite the different faults and to make them observable.

In other works [7][8][9] data and control modules are tested and analyzed by the presence of faults. These experiments included reliability characterization by statistical fault injection methods and checking the effect of software errors in the system. In [10] it is presented a methodology to mitigate hardware failures in GPUs. Moreover, a rescheduling strategy is introduced to ensure execution under hardware malfunction.

This paper focuses on the test of the GPGPU runtime scheduler and aims at investigating the fault coverage that can be

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

achieved resorting to a functional approach. The GPGPU runtime scheduler is probably the most critical unit within a GPGPU since it manages the parallel task execution and it partitions a complete job into tasks considering the device memory size and by dynamically scheduling the available GPGPU cores.

The implementation of this module is not known in detail for commercial GPGPU devices, thus we resort to the FlexGrip open-source model developed by the University of Massachusetts [11]. We first evaluated the stuck-at fault coverage that can be reached on some specific parts of the scheduler and on the scheduler/core interface by simple programs running on the GPGPU (identifying those faults, that cannot produce any failure, given a specific scenario), and then propose some techniques to improve such a figure.

The paper is organized as follows. In Section II we summarize the key characteristics of the FlexGrip model, with special emphasis on the scheduler module we target in our work. In Section III we propose the techniques for generating suitable test programs for such a module. Section IV reports some experimental results, and Section V finally draws some conclusions.

II. AN OVERVIEW OF FLEXGRIP

FlexGrip is a VHDL model for a GPGPU module developed by the University of Massachusetts and targeting a Xilinx FPGA [11]. The module is based on the Nvidia G80 Tesla architecture and is compatible with the Nvidia CUDA Compilation Toolkit. It accepts, as input application code, CUDA-binary code (.SASS) compiled with SM 1.0 compatibility level. 27 instructions are supported by FlexGrip.

The CUDA programing model parameters, employed in FlexGrip, and the kernel parameters, such as Grid dimension, Block dimension and Blocks per core, can be configured before simulation. Meanwhile, kernel application parameters could be configured to pass the global memory address of input and output data into the kernel application execution.

A. The Overall Architecture

The FlexGrip architecture is based on the SIMT (*Single Instruction Multiple Thread*) paradigm and exploits a custom Streaming Multiprocessor (SM) core with five stages pipeline (fetch, decode, read, execution/control and write). Moreover, each SM (see Fig. 1) includes a warp unit as thread controller. In the SIMT architecture, a single instruction is fetched, decoded and then distributed to be executed on every scalar processor (SP) in the SM. Read and write stages load and store data operands from and to registers files (FRs), shared, global or constant memories.

The level of instructions parallelism in the SM core is customizable by user or host. It allows the selection of one of three possible configurations corresponding to 8, 16 or 32 SPs, respectively. This parameter will affect the total number of logic elements, the total energy consumption and the performance of the whole GPGPU.

The number of SPs allows the execution of the same parallel instruction and it implies some changes in the warp distribution of threads. In the 8-SP core configuration, a 32-threads warp is divided into four lanes and every SP must execute 4 threads per warp in sequence. For the 16-SP configuration, the warp threads are organized in two lanes per SP. Finally, in the 32-SP configuration, the maximum instruction parallel capability is obtained with 32 threads executed in parallel one by each SP. It is worth highlighting that the SP hardware only exists in the execution/control pipeline stage. The other pipeline stages are shared inside the SM and one instruction is fetched, decoded, read and written per SM. Only integer operations are supported by FlexGrip.

A major advantage in FlexGrip is the use of an additional module to support thread-level branching at hardware level (branch unit). This module is not commonly supported by SMs in G80 architecture and manages control flow instructions and synchronization points in order to create or return from conditional branches with multiples paths. A conditional branch instruction causes *divergence paths* when a set of threads make different decisions in terms of branch taken or not taken, while a synchronization instruction can cause previous paths to *converge*. Once path divergence occurs, all the threads will continue until they reach the synchronization point where the path convergence occurs, so the threads could be executed in parallel again. At hardware level, the branch unit includes a warp stack memory which is employed to store thread information such as actual mask, program counter and state for each individual thread.

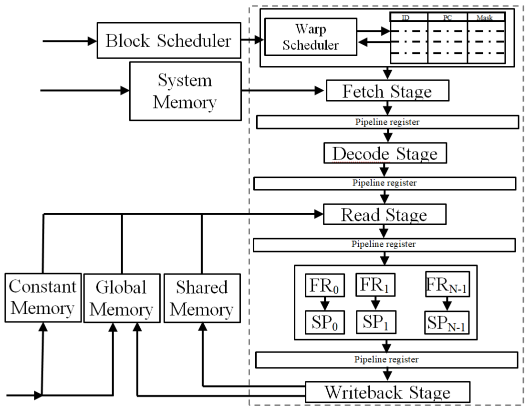

Fig. 1. The FlexGrip architecture: a focus on the SM.

B. Warp Unit

Nvidia defines a warp as a set of 32 or 64 threads which execute the same set of instructions in parallel with data independency. This implies that every thread assigned to a SP can execute parallel operations employing different data resources. In the FlexGrip architecture, the thread execution is managed by a warp unit in each SM with a limit of 32 threads per warp.

The warp unit is mainly composed of warp generator, warp scheduler, warp checker, and two memories to store information about the warp threads, namely the *warp state memory* and the *warp pool memory*. The warp pool memory stores the ID, the execution program counter, and the thread mask for each warp. The warp state memory stores the actual state of the warp (*active*, *wait*, *stop* and *finish*). Each entry line in both memories corresponds directly to a warp that is being or will be executed in the SM. For every thread in a warp, the fence registers store in a bit the state of thread, indicating whether the thread is in use or in waiting state. The warp scheduler reads and checks the fence registers in order to identify when all warp threads are in a waiting state. If this condition is true, it means that all previous divergence paths converged and the synchronization barrier is released, so that warps can return to normal operation.

The warp generator defines the total number of warp pool lines to be employed by the application and assigns the entry lines in the pool memory needed to manage them. Moreover, this unit writes the initial value of every warp entry line in the pool memory.

The warp checker modifies the fields in the warp lines after a comparison between the real state values, coming from the SM, and the predefined values in the warp pool memory. Moreover, this module writes and reads the state memory. The warp scheduler is composed of a state machine and coordinates the execution of every warp in the SM.

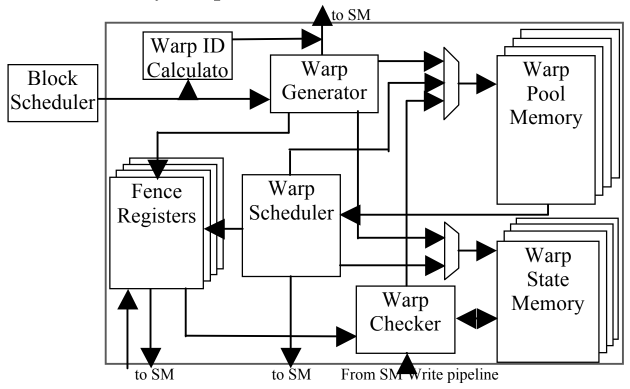

Fig. 2. FlexGrip Warp Unit General Architecture.

The configuration of FlexGrip starts in the block scheduler. This block defines and controls (among other configuration parameters useful for the SM execution) the dimension of blocks and warps for the application. The block scheduler controls and manages the execution of block threads in the SM, adopting a round robin scheme, and it sends the configuration parameters to the warp unit in order to control the execution of multiple warps.

For a correct execution of the branch unit, the G80 architecture supports synchronization barriers which allow the generation of convergence points. These points are employed to establish a memory address, generally after conditional paths ends, where threads wait until all divergence path threads reach to the same address.

C. *Interface between scheduler and the SM*

The interface is formed by a set of connections that interconnects the warp unit and the SM controller. Inside the SM, some connections are used to notify and modify the thread execution according to the executed instructions. The branch unit, in the execution/control stage, is able to generate divergence paths in active threads. The interface is used to modify the content of the thread state in the warp unit.

The interface connections carry information about control signals, number of actual active threads, content of initial mask state, address of general purposes registers per thread and shared memory, size of shared and register used by thread, thread program counter and warp identifier.

III. The Proposed Evaluation

The evaluation we performed consists of a preliminary permanent fault coverage analysis of a simple benchmark applications using the whole set of GPGPU assembly instruction, described in the subsection III.A. Secondly, we developed three incremental methods to enhance the fault testing capabilities of any application implemented on GPGPU. The three methods are accurately described in the subsection III.B showing the interdependence between the application execution flow (e.g., divergence and convergence paths), the thread mapping and the shared memory allocation.

Please note that for the purpose of this paper, a fault can be labeled as detected at the end of the execution of a given piece of SBST code when one or both of the following condition hold (called detection by memory content and by performance evaluation, respectively).

In the first situation, the results, produced in memory by the fault-free and each faulty system, are compared. A fault is detected when the comparison produces a mismatch. In the second, Performance evaluation is based on two checks. Firstly, we check if the kernel execution was correctly completed and the simulator generated the results in memory. If the fault caused the system to hang and kernel execution did not finish, the fault is labeled as detected. In the case of execution completed, we check and compare the kernel execution time (e.g., the number of clock cycles) with the fault-free one. In case of time mismatch, the fault is labeled as detected *by performance degradation*. In these evaluation, it is possible that a fault could be detected by performance evaluation and memory mismatch; however, the checker priories a fault detected by wrong results in memory instead of performance error.

A. *Basic program behavior*

We first selected a simple benchmark (`VectorAdd`) to evaluate the effects of permanent faults in the warp unit (i.e., in the warp pool memory and in the interconnections with the SM). This sample program performs an add operation between two vectors and generates a result vector. The benchmark is composed of 18 SASS assembly instructions.

This simple program corresponds to a one dimension data intensive application. Despite its simplicity, it suitably stimulates the warp unit. The application is configured to use one grid and 256 threads per block. This implies that the size of the input vectors is limited to 256 operands.

With the application configuration selected above, the number of warp pool line entries used is 8 out of the 32 available lines. Hence, we evaluated the effects of the program execution on the testable fields, i.e., the mask field and the program counter in the used warp pool line entries and in the interconnections interface.

A fault affecting the controls signals or the warp base parameters between the warp unit and the SM could stop or hang the system. As a consequence, the faulty SM hardware uses other system configuration parameters than the correct ones and the application never finishes or generates a valid result. A wrong base address for either the shared memory or the file registers could overlap operands from different threads and thus compromise the application execution. Furthermore, a fault in the actual thread state connections can generate performance issues, such as unexpected latency, or a mismatch in the final results in memory.

Permanent faults in the warp pool memory may also generate some issues. In the actual mask field can produce two possible scenarios. In the first scenario, some threads preserve a permanent active state and never finish the kernel execution. In the second scenario, some threads preserve a permanent inactive state and are not allowed to execute instructions and write results in the global memory. This implies that some results will not be written to memory. In this case, a memory mismatch is produced at end of the program execution.

A fault in the execution program counter field can generate a hang in the thread execution and access to invalid locations in the program memory. Finally, a fault in other warp pool fields, which correspond to memory and register file base addresses configuration can cause the kernel to crash. However, some of these faults are observable but not controllable. These permanent faults belong to block scheduler configuration settings and a GPGPU functional testing approach cannot access to block settings, so those faults could be observed by proposed observation mechanism but it is not possible to stimuli additional faults.

In Section IV we report some experimental results supporting the analysis above and provide a quantification of the different fault effects on a sample test case.

B. Enhanced versions

Taking into account the previous analysis, we propose three approaches to enhance an existing program (such as the one introduced in the previous sub-section) to increase its ability to test the faults affecting the fields of a warp pool line entry and the connections between the warp unit and the SM.

These approaches heavily rely on the thread ID, or index, which uniquely identifies a thread during its execution which, in turn, depends on the kernel dimension and application complexity. Moreover, it defines the distribution of blocks and threads in a grid and could be used by multiple threads to access multiple data locations (Multiple Threads Multiple Data or MTMD). This parameter is also used as a base address to load or store operands from different memory locations.

The proposed SBST approaches employ the thread ID to identify the number of a thread executed in a SP and to change the thread execution order. The change in execution generates divergence paths and the expected stimuli to check the actual mask field. The reader is invited to refer to Fig. 3 for a pseudo-code for the different methods.

1) Method M1

This method uses the thread ID during the application execution without compromising the application order. M1 is based on introducing a set of comparisons between the thread ID of each thread and a set of constant values.

Firstly, the application is executed. Secondly, the thread ID and the constant value *j* are compared, and depending on the comparison, two possible (*divergence*) paths are generated in the SM. With different instructions executed in the paths, a difference (of timing) in thread execution is produced. Then, the divergence is repeated to consider the mark fields of all the warp pool lines. The total number of such repeated evaluations depends directly on the number of warps-per-block and threads-per-warp configured by the host according to the application.

A permanent fault in the mask field generates wrong divergence paths, leading to performance variation which can be used for fault detection. This technique only targets the permanent faults affecting the ID field of the warp pool line. However, some comparison routines can be suitably placed in the program memory in order to also detect faults affecting the bits belonging to the warp program counter.

2) Method M2

With this method, the divergence paths generated by the comparison are divided into two groups, one of which will execute a global memory store instruction to extend the performance variation and introduce the possibility of checking global memory for fault detection. The other group will bypass this instruction.

In the presence of a permanent fault in the mask field, unauthorized and permanently active threads will store final memory results more than once. This will not affect the final result of the application in global memory but will cause a performance variation due to extra global memory accesses. On the other hand, a permanently inactive thread will never write to global memory, leading to some missing data in the final results in the global memory.

The same approach employed in M1 method is used to check the warp program counter fields in the warp pool line. Additional branches are also inserted to detect faults affecting the warp program counter. With this approach some interface interconnection faults can also be detected.

3) Method M3

The methods M1 and M2 depend on performance variation triggered by divergence paths in the test program to detect the permanent faults in the mark field. However, in practical GPGPUs, these methods would require performance counters for detection, which are not necessarily implemented or easily accessible. To solve this issue, a variation of M2 is presented. In this approach, a basic memory thread signature is introduced.

Firstly, a signature initialization with zero is carried out. For every thread executed by the application a particular memory location is required to store the signature. This means one exclusive memory location for each thread.

One of the two divergence paths groups (convergence path) is composed of the following procedures: load thread signature, increase it, store of the signature in memory and the thread instruction for global memory access as in M2.

```
j ← 0                                              ► Clear constant
…                                                  ► Normal app. Execution
Sig_per_thread[] ← 0                               ► Initialize signature      (M3)
for i ϵ {set of ThreadId in SM} do                 ► Evaluate for every ThreadID
    if i == j then                                 ► If ThreadID Matches
        Divergence_path_GroupA();                  ► Divergence path Group A
        Thread_Store_in_memory();                  ► Memory results store     (M2)
        Sig_per_thread[i] ← Sig_per_thread[i]+1    ► Set signature             (M3)
        Sig_store_in_memory();                     ► Store  signature (M3)
    else
        Divergence_path_GroupB ();                 ► Divergence path Group B
    j ←j+1                                         ► Change constant value
```

Fig. 3. Pseudocode for method M1 (white), M2 (white and light gray), and M3 (white and dark gray).

In this method, the thread signature is essentially a counter which is incremented each time the thread writes into memory: the expected value for every thread is one. A higher value implies that the thread has written more than once in memory. A signature with value zero implies that the thread never accessed the global memory. The detection of permanent faults in mark field can be performed by checking the signatures in the final results.

IV. Experimental Analysis

A. The FlexGrip simulation setup

In order to assess the effectiveness of the proposed techniques we performed some simulation-based fault injection campaigns using the FlexGrip model [11].

A tool has been developed to automate the fault injection using ModelSim, which is able to start the simulation, load the test program, generate the ModelSim command to inject a permanent fault in the desired signal in the design and collect the log file generated by FlexGrip after each fault simulation. The log file contains the final results of the test program in the global memory, which is then compared with the golden results from fault-free simulation for fault detection purposes. In case the log file is not generated at all, it means that the kernel never finished the test program execution, and such fault is labeled as detected by system hang.

Besides checking the final results in global memory, performance evaluation is also performed by comparing the actual and golden kernel execution time. A fault is labeled as detected by "performance degradation" when the execution time, in the fault simulation, is longer than the expected value.

At the end of the fault injection campaign, the list of faults injected, labeled according to the two checks mentioned above, is gathered for coverage analysis as presented later in this section. The fault injector reads a fault list file which includes the fault type, the logic value and the signal name.

B. Changes to FlexGrip

The original FlexGrip has been designed using the Xilinx System Generator to take advantage of the libraries and cores provided by Xilinx for better resources utilization in the target FPGA (Virtex6). For the scope of our work, Xilinx libraries cores have been changed with synthesizable VHDL modules exactly replicating the functionalities of the original FlexGrip Xilinx Core Generator modules.

These modules include different register files previously implemented using Xilinx Block Memory IP Core replaced by behavioral level dual-port memory implementation, scalar processor for different arithmetic and logic operators, and so on. After the modifications, it is possible to simulate the resulting model with the standard library using ModelSim and perform the fault injection analysis.

C. Methods implementation

The three SBST methods are applied to the original application code using the FlexGrip model.

1) Method M1

In FlexGrip, the SM initialization loads the thread ID to a specific location in the shared memory. To use the thread ID in the test program, a move from the shared memory to an un-occupied register needs to be inserted.

The total number of constants to compare depends on the blocks and threads per block configuration. For the present application, eight comparison routines are required, according to the total number of SPs selected in SM configuration for FlexGrip. The implementation of the M1 method code is composed of 91 assembly instructions, including the original application.

The routines employed are aimed to perform the following operations:

1. Selection of a convergence address point. An instruction memory address is defined as a convergence point for thread divergence.
2. Comparison between the thread ID register and a constant value. Threads with different thread ID values will enter in waiting state before the convergence point.
3. Conditional branch execution. This generates the divergence paths and the stimuli required for pool line test.
4. Execution of divergence paths. Threads which match the constant value will execute two extra NOP instructions. All threads will converge again at the convergence point determined in operation 1 and continue to execute in parallel.

These operations are repeated for every constant value predefined in the test code, as illustrated in Fig. 4.

```
...                       ► Application code
GLD Rx, g[0x06]           ► Move of threadIdx.x (stored in shared memory)
MVI Ry, Z                 ► Move constant parameter per SP (from 0 to (Z-1))
...                       ► Application code
                          --------------- M1 code ---------------
AND Rx, Ry                ► Comparison between constant value and threadIdx.x
SSY Dir_1                 ► Convergence point definition
BRANCH Dir_2              ► Conditional evaluation
NOP                       ► Divergence Path
NOP
Dir_2:GST M[Ra],Rb        ► Convergence Path, Storage thread results
Dir_1:  NOP.S             ► Warp branch stack release (Convergence point)
--- Repeat Z-1 times according to the number of threads per block.
--- End of M1 code.
```

Fig. 4. General Pseudocode for a routine implementing Method M1. *Z* is the number of threads per block or threads defined for the application.

The number of conditional branch routines is directly related with the maximum number of threads-per-block and the number of threads defined in the application. These routines are placed at the end of the application code.

2) Method M2

This method employs the same thread ID index. However, the application execution order is modified and the final global memory store instruction is replaced with a series of conditional routines. In our example, 90 assembly instructions are employed.

In this approach, the first three operations are equal to the method described above. In operation 4 the path includes the final memory result storage instruction. In this case, the execution of the store instruction generates an additional latency due to global memory access. The operations are repeated the same number of times as M1.

3) Method M3

In this approach, the use of the thread signature requires some additional space in the global memory. In this case, a base address is selected for this purpose. The following operations are carried out replacing the final storage operation:

1. Thread signature initialization. For every active thread, the signature in the global memory is initialized with zero.
2. Selection of a convergence point.
3. Comparison between thread ID and a constant value, generating divergence paths of thread execution.
4. Threads whose thread ID matches the constant value will perform final memory results store operation, load the thread signature, increase once, and store back into global memory.
5. Other threads execute three NOP instructions and an unconditional branch instruction to reach the convergence point.

In FlexGrip, the code is implemented in .SASS assembly language with 152 instructions. However, the proposed approaches can also be developed at a higher level (e.g., using CUDA C) with the use of a switch statement and the kernel (threadIdx.x) variable. An example of routine implementation of the method M2 at high level is presented in Fig 5.

```
...                         ► Normal application code
switch(threadIdx.x)         ► Comparison of threadIdx.x
{
  case Z:                   ► Thread execution for threadIdx.x with Z value
  Thread_final_Store();     ► Store of results in global memory
  break;
  ...                       ► Comparison with other Z-1 value
}                           ► End of M2 code
```

Fig. 5. CUDA code for method M2. *Z* is the number of threads per block or thread in the application.

When implement the test code in CUDA C level, PTX-CUDA C mixing programming is required to allow finer control of branch instructions and convergence, though instrumentations are still needed after PTX-SASS translation as control flow instructions are not explicitly allowed in higher level.

D. Experimental Results

Two fault simulation campaigns have been performed on the GPGPU model, configured with a clock of 100 Mhz, using the fault list of the warp pool memory lines (2,048 elements) and the interface connection (478 elements). FlexGrip has been configured with one grid, 256 blocks and 24 threads per warp. Table 1 presents the results of the fault simulation campaign for the warp pool memory lines. The original application (VectorAdd) requires 142.005μs. of simulation time. Algorithms M1, M2 and M3 require 142.005 μs, 415.215 μs and 1.122 ms respectively.

There are some permanent faults in the warp pool memory that are functionally untestable , such as the permanent bits in the higher part of the warp ID and some bits from shared memory and general-purpose registers base addresses. In fact, these fields are kept constant during the kernel execution. There are also fields defined as constant values which are not accessible. Taking into account those restrictions, the total number of untestable faults per entry line is equal to 156 permanent faults. Furthermore, 23 faults are observable but not controllable. These faults correspond to the base addresses of the general purposes registers file and shared memory space of every thread, so for the application with eight warps entry lines, the total amount of untestable faults is 1,064.

From Table I, method M1 increases the number of detected faults comparing to the original application (VectorAdd), while method M2 is capable to detect all the testable permanent faults in the warp pool memory. The 256 permanent faults detected by memory mismatch in M2 are all related to the actual mask field for the eight lines used by application. It means that all permanent faults in the actual field are detected. The same numbers of permanent faults in program counter, detected by method M1, are detected also by M2.

TABLE 1. FAULT DETECTION RESULTS IN WARP POOL MEMORY.

Application Code	**VectorAdd**	**M1**	**M2**	**M3**
Total Faults	2,048	2,048	2,048	2,048
Testable Faults	984	984	984	984
Detected Faults	624	728	984	984
Hang	440	613	616	616
Memory Mismatch	184	115	112	368
Performance degradation	0	0	256	0
Testable Fault coverage (%)	63.41	73.98	100	100
Fault coverage (%)	30.46	35.54	48.04	48.04

The method M3 aims at a different purpose. This approach allows fault detection in the actual mask field employing a different observation mechanism, i.e. checking the final results in the global memory only. Hence, it does not require the use of additional or complex hardware for performance or timing measurement during test.

An analysis of the interconnection signals shows that a total of 201 connections are not relevant for kernel execution and thread control in the GPGPU and are classified as untestable faults for the proposed methods. In Table 2, the results of the fault injection campaign regarding the interconnections between the warp unit and the SM is reported. M1 increases the fault detection coverage by 7.94% as it is able to detect those faults affecting the bits carrying the thread state information between the warp unit and the branch unit to control the branch execution. M2 further increases the fault coverage to 85.92% instead of 100%, as connections regarding shared memory size, general-purpose registers size or number of warps do not generate misbehavior in FlexGrip operation.

TABLE 2. RESULTS FOR FAULT DETECTION IN WARP UNIT CONNECTIONS.

Application Code	**VectorAdd**	**M1**	**M2**	**M3**
Total Fault	478	478	478	478
Testable Faults	277	277	277	277
Detected Faults	155	177	238	236
Hang	105	157	154	161
Memory Mismatch	50	20	20	75
Performance degradation	0	0	64	0
Testable Fault Coverage (%)	55.95	63.89	85.92	85.20
Fault Coverage (%)	32.42	37.02	49.79	49.37

Faults detected by performance degradation correspond to the signals between the warp unit and the Execution/Control stage in SM which are in charge of preserving execution coherency between the two modules.

The method M3, without checking Performance degradation, is still able to achieve very close fault coverage by only checking the final results in global memory. Fault coverage is increased by 29.25% comparing to the original application, which means that 81 additional permanent faults are detected: 64 of them belong to the actual thread state and the other faults belong to the thread program counter.

In M2 some permanent faults are detected with different observation mechanisms. Some faults affecting the thread program counter and base address of the general-purpose register file previously detected by memory mismatch become detected by Hang or Performance observation. Faults related to bus control signals, memory and general-purpose register file base addresses are detected by Performance degradation in M2.

M1 is able to detect some faults in the warp pool memory and in the interconnections; however, the percentage of fault coverage is low. On the other hand, M2 achieves higher fault coverage by introducing store instruction to access global memory to increase performance variation among different divergence paths. Meanwhile, M3 achieves similar high fault coverage by only checking the final results in global memory, taking advantage of a signature variable for each thread.

V. CONCLUSIONS AND FUTURE WORKS

In this work three SBST incremental methods are proposed to detect faults inside the GPGPU warp scheduler. These methods can be applied to any suitable existing GPGPU application. The key idea is their capability to generate divergence paths of thread execution and use performance variation among the threads and/or final results in global memory to detect permanent faults.

Fault injection campaigns have been carried out using the FlexGrip GPGPU VHDL model. Results indicate that both method M2 and M3 are promising SBST methods able to achieve high fault coverage. Especially, the M3 method requires only to check the final results in global memory after test program execution, which is a typical mechanism used in processor SBST techniques. As future works we plan to extend the characterization to further GPGPU modules and to compare the fault coverage results with extended Instruction Set Architecture (ISA) fault simulators. Besides, we plan to use the proposed techniques on gate-level netlist models and real platforms.

REFERENCES

[1] D. Sabena; L. Sterpone; L. Carro; P. Rech, "Reliability Evaluation of Embedded GPGPUs for Safety Critical Applications", IEEE Transactions on Nuclear Science, Year: 2014, Volume: 61, Issue: 6, Pages: 3123 – 3129

[2] L. Bautista Gomez; F. Cappello; L. Carro; N. DeBardeleben; B. Fang; S. Gurumurthi; K. Pattabiraman; P. Rech; M. Sonza Reorda, "GPGPUs: How to combine high computational power with high reliability", 2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)

[3] Paolo Bernardi; Riccardo Cantoro; Sergio De Luca; Ernesto Sánchez; Alessandro Sansonetti, "Development Flow for On-Line Core Self-Test of Automotive Microcontrollers", IEEE Transactions on Computers, Year: 2016, Volume: 65, Issue: 3, pp. 744 – 754

[4] Stefano Di Carlo; Giulio Gambardella; Marco Indaco; Ippazio Martella; Paolo Prinetto; Daniele Rolfo; Pascal Trotta, "A software-based self test of CUDA Fermi GPUs", 2013 18th IEEE European Test Symposium (ETS)

[5] Andreas Riefert; Riccardo Cantoro; Matthias Sauer; Matteo Sonza Reorda; Bernd Becker, "On the automatic generation of SBST test programs for in-field test", 2015 Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 1186 – 1191

[6] A. Paschalis; D. Gizopoulos, "Effective software-based self-test strategies for on-line periodic testing of embedded processors", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Year: 2005, Volume: 24, Issue: 1, Pages: 88 – 99

[7] Sparsh Mittal, "A Survey of Techniques for Modeling and Improving Reliability of Computing Systems", IEEE transactions on parallel and distributed systems, Year: 2016, Vol. 27, no. 4

[8] N. Farazmand, R. Ubal, and D. Kaeli, "Statistical fault injectionbased AVF analysis of a GPU architecture," in Proc. 8th IEEE Workshop Silicon Errors Logic Syst. Effects, Year: 2012.

[9] J. Tan, Y. Yi, F. Shen, and X. Fu, "Modeling and characterizing GPGPU reliability in the presence of soft errors," Parallel Comput., Year: 2013, vol. 39, no. 9, Pages 520–532

[10] David Defour, Eric Petit, "A software scheduling solution to avoid corrupted units on GPUs", Journal of Parallel and Distributed Computing, Year: 2016, Volumes 90–91, Pages 1-8

[11] Kevin Andryc; Murtaza Merchant; Russell Tessier, "FlexGrip: A soft GPGPU for FPGAs", 2013 International Conference on Field-Programmable Technology (FPT), pp. 230 – 237