# A Compaction Method for STLs for GPU in-field test

Juan-David Guerrero-Balaguera, Josie E. Rodriguez Condia, Matteo Sonza Reorda
Department of Control and Computer Engineering
Politecnico di Torino, Torino, Italy
{juan.guerrero, josie.rodriguez, matteo.sonzareorda}@polito.it

***Abstract* — Nowadays, Graphics Processing Units (GPUs) are effective platforms for implementing complex algorithms (e.g., for Artificial Intelligence) in different domains (e.g., automotive and robotics), where massive parallelism and high computational effort are required. In some domains, strict safety-critical requirements exist, mandating the adoption of mechanisms to detect faults during the operational phases of a device. An effective test solution is based on Self-Test Libraries (STLs) aiming at testing devices functionally. This solution is frequently adopted for CPUs, but can also be used with GPUs. Nevertheless, the in-field constraints restrict the size and duration of acceptable STLs. This work proposes a method to automatically compact the test programs of a given STL targeting GPUs. The proposed method combines a multi-level abstraction analysis resorting to logic simulation to extract the microarchitectural operations triggered by the test program and the information about the thread-level activity of each instruction and to fault simulation to know its ability to propagate faults to an observable point. The main advantage of the proposed method is that it requires a single fault simulation to perform the compaction. The effectiveness of the proposed approach was evaluated, resorting to several test programs developed for an open-source GPU model (FlexGripPlus) compatible with NVIDIA GPUs. The results show that the method can compact test programs by up to 98.64% in code size and by up to 98.42% in terms of duration, with minimum effects on the achieved fault coverage.**

**Keywords—Functional Testing, Graphics Processing Units (GPUs), Self-Test Libraries (STLs), Test Compaction**

## I. Introduction

Currently, Graphics Processing Units (GPUs) are effective platforms used in several data-intensive applications, sometimes in safety-critical domains. Safety-critical applications increasingly employ GPUs as the main workhorse to perform complex operations and process large amounts of information (e.g., for Artificial Intelligence and sensor fusion operations). However, in this domain, effective methods to identify possible faults arising in a device and to face their effects are crucial goals set by the functional safety standards.

One feasible test solution corresponds to the Software-Based Self-Test (SBST) strategy, which is based on developing special test programs (TPs) able to detect faults. These TPs comprise Self-Test Libraries (STLs) [1].

STLs implement a functional at-speed test strategy to detect faults with a functional test run at the maximum operating clock frequency and in operational conditions. Furthermore, STLs are suitable for in-field testing (during the operative life of a device) and allow, in particular, the periodic testing of most internal modules. Currently, manufacturers, such as STMicroelectronics, Arm, NXP, Infineon, Renesas, Cypress, and Microchip (among the others), provide their customers with STLs, thus offering in-field test capabilities for their processor-based products targeting several domains (industrial, medical, aerospace, and automotive) [2]–[5].

An STL for processors and microcontrollers includes one or more TPs developed with different approaches and programming styles to achieve a given structural fault coverage (FC), e.g., in terms of stuck-at faults. Similarly, STLs for GPUs are composed of several 'Parallel Test Programs' (PTPs), which provide the same features as TPs in processors. These PTPs are designed to exploit the intrinsic parallelism of GPUs to perform the test.

In the past, several works demonstrated the feasibility of developing PTPs for control units [6], [7] memory modules [6], [8], [9], and functional units [10]–[12] in GPUs, achieving a good FC. Each PTP requires a given amount of time to be executed. However, it is possible that several application's constraints might limit the available execution time. In this scenario, short and fast PTPs are desired. Thus, compaction methods can support the optimization of PTPs and simplify their adoption for in-field test. PTPs in STLs are generated using different approaches (e.g., custom, ATPG-based, pseudorandom, deterministic), so the compaction of a PTP can be a challenging task. Moreover, parallelism features and constraints of a given PTP must be considered when compacting their size and duration.

Previously, several works proposed methods to compact TPs for processor-based systems. These methods effectively reduce the size and duration of TPs while maintaining the same FC [13]–[15]. In [16], the authors split TPs into sub-routines and remove individual instructions after analyzing the FC contribution of each sub-routine. Authors in [17] exploited reordering techniques among different pieces of a TP to maintain the FC and reduce the length of the TP. In both cases, a high computational effort is required to analyze and compact a given TP. In fact, the compaction process is based on the production of compacted TP candidates from the original TP, which are then fault simulated to assess the new FC. However, the required time and computational costs for the compaction of an individual TP are exceptionally high. It is worth noticing that none of the reported techniques in the literature face the compaction of PTPs and STLs for GPUs, and some of them can hardly be extended from CPUs to GPUs.

In this work, we present a method to perform the compaction of PTPs for GPUs by combining the information about the microarchitectural operation performed by a PTP, the individual thread operation, and the fault propagation abilities of each test pattern to any visible point. This method follows and extends the basic idea of the time-efficient compaction

approach presented in [18], which was developed targeting STLs for CPUs only, and employs different abstraction levels (software, RT level, gate level) to perform only ONE logic simulation and ONE fault simulation, thus significantly reducing the required compaction time and with minimum impact on the FC.

In both logic and fault simulations, several parameters are collected and extracted to support the compaction of STLs. Firstly, a logic simulation using the RT-level model of a GPU is used to gather detailed tracing information about the executed PTP on every clock cycle. Secondly, a fault simulation is performed using the gate-level version of the circuit. This fault simulation also records the number of faults detected at each clock cycle. The results obtained in both simulations are used to identify those instructions in the PTP unable to stimulate or propagate fault effects in a target module, so listing them as candidates for elimination. To the best of our knowledge, the proposed compaction method is the first attempt to address the compaction of STLs in GPUs.

For the purpose of this work, an open-source GPU model (FlexGripPlus) was used to quantitatively evaluate and validate the proposed compaction method. FlexGripPlus is compatible with the NVIDIA GPU architecture and programming flow. The experimental results show that the compaction method is highly effective in compacting PTPs with continuous and regular structural descriptions. The reduction in the size of PTPs reached up to 98.64% and up to 98.42% in the duration while minimally affecting the fault detection capabilities in the STL.

The paper is organized as follows: Section II introduces the background. Then, Section III describes the proposed compaction method for STLs in GPUs. Section IV reports the experimental results. Finally, Section V presents the main conclusions and future works.

## II. Background

### A. GPU organization

The architecture of a GPU is based on arrays of parallel execution units (also called *Streaming Multiprocessors* or SMs) in the NVIDIA's terminology. An SM is the main operative core inside a GPU, and it implements the *Single-Instruction Multiple-Data* (SIMD) paradigm or variations, such as the Single-Instruction Multiple-Thread (SIMT). More in detail, each SM includes several functional units (*Streaming Processors* or SPs), which are used to execute the same instruction in parallel for several threads. The number of SPs (from 8 to 128) directly depends on the GPU architecture and the number of parallel threads to be processed simultaneously. Moreover, the SM also includes several functional units, such as Special Function Units (SFUs) and Tensor Core Units (TCUs), to perform specific operations and support multimedia and artificial intelligence applications. The GPU architecture also includes a memory hierarchy mainly used to reduce latency during the kernel execution. The memory resources include a '*General Purpose Register File*' (GPRF), a shared memory, a local memory, a constant memory, and an external global/main memory.

A parallel program (*kernel*), executed by the GPU and called by the Host, is divided into parts by a general controller and assigned to the available SMs. Then, each SM loads one instruction from the code and processes it in parallel through the available SPs. During the execution, the SM processes in parallel a set of threads (or *Warps*).

### B. FlexGrip GPU Architecture

FlexGripPlus [19] is an open-source GPU model based on the description of one NVIDIA's microarchitecture (G80) [20]. This GPU model supports up to 52 assembly instructions and is compliant with the programming flow of NVIDIA. FlexGripPlus is organized as a set of arrays of SMs. One general controller controls the tasks submitted to every SM. In each SM, a local controller manages the task by dispatching a warp into the available SPs. The SM is divided into five pipeline stages and executes one instruction following the Single-Instruction Multiple-Thread (SIMT) paradigm. More in detail, the SM includes 8 SPs, 8 Floating Point Units (FP32), and two SFUs. The flexibility of the GPU model allows the selection of the number of execution units (8,16, or 32) in the SM.

### C. Software-Based Self-Test

SBST [1] is a flexible and noninvasive strategy aiming at detecting faults in internal modules of a processor-based system. SBST can be used at the end of the production phase and is also widely employed for in-field test. This strategy is based on executing specially crafted TPs using selected instructions at maximum operational clock speed.

In parallel architectures, such as GPUs, the SBST strategy can also be adopted to develop PTPs. Each PTP is built employing the available Instruction-Set Architecture (ISA) of a target GPU. Each instruction in the PTP is intended to apply one or more test patterns to one or several target modules in parallel. These instructions compose routines aiming at exciting, propagating, and detecting faults when operating warps in an SM.

In general, a PTP is composed of three main parts: *i)* thread registers load, *ii)* parallel operation execution, and *iii)* propagation of the result to an observable point.

In principle, these steps are repeated for each thread in the program. However, it is also possible that divergences could be present, so only a portion of the threads executes a given operation, meanwhile missing threads skip or perform different procedures. This divergence behavior is commonly used to excite control modules but may affect the test quality on functional units and regular structures in the GPU. In these parallel architectures, the fault detection of a PTP is commonly performed using exceptions and thread signatures [21] out of the values on any observation point or memory output of the GPU. A comprehensive overview of the main issues (and possible solutions) to be faced when generating STLs in an industrial environment can be found in [22].

Although several works have been published regarding the compaction of TPs for processors, to the best of our knowledge, there are no published works facing the compaction of PTPs for GPUs.

## III. Proposed compaction approach

The proposed compaction method assumes the availability of a Self-Test Library (STL) for a particular module in the GPU or for the complete GPU. The STL can be split into PTPs. Each PTP is composed of a given number of instructions, using the GPU's assembly language, and targets a given fault model with a required execution time (clock cycles or ccs) and FC per target module in the GPU.

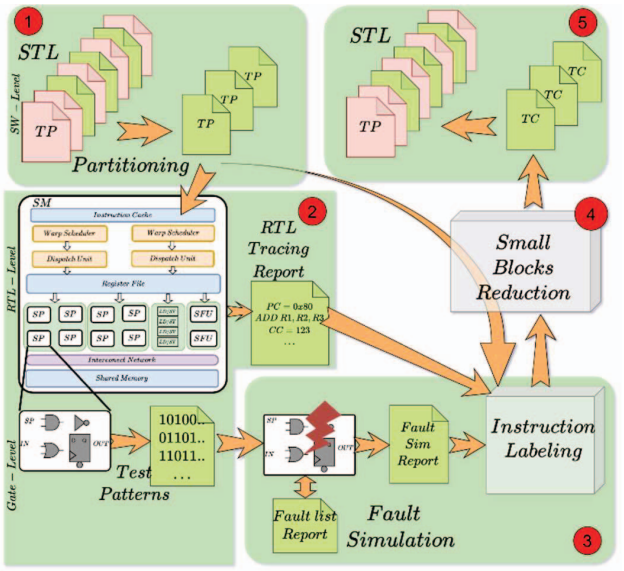


**Fig. 1.** A general scheme of the proposed compaction approach for functional TPs in GPUs.

The compaction approach proposed here works on PTPs targeting the stuck-at fault model. However, the same compaction approach can be adapted considering other fault models as well.

The compaction approach is divided into five stages. *i)* PTP partitioning, *ii)* Logic tracing, *iii)* Fault detection analysis and labeling, *iv)* PTP reduction, and *v)* PTP reassembling.

In the program partitioning stage, see (1) in Figure 1, the target PTPs are extracted and analyzed individually to perform the compaction procedure. This analysis consists in the identification of the portions of the PTP which are suitable for compaction: we called these portions of code *Admissible Regions for Compaction* (ARCs). The identification of the ARC follows three steps. The first step defines and finds the Basic Blocks (BBs) of each PTP. One BB is a group of instructions that are always executed in sequence (no in/out jumps or loops in the BB) [23]. In the GPU case, a BB can be defined as a group of embarrassingly parallel plain sequence of SIMD or SIMT instructions [24]. The second step analyzes the control flow graph of the PTP and incorporates in the ARC all BBs in the PTP except those BBs involved in parametric loops whose iterative parameter is calculated by any BB inside or outside the loop. Once the ARCs are identified and chosen, the third step of the first stage of the compaction method, extracts these regions from the PTPs. In contrast, other regions of the PTPs are discarded as candidates for compaction and remain unaffected during the compaction process.

The logic tracing stage, see (2) in Figure 1, performs two logic simulations (one RTL and one GL) with the PTPs in the microarchitectural description of the GPU and extracts information about the execution with the purpose of identifying the relationship between each instruction in the BBs and its effects in terms of fault detection per warp.

On the one hand, the RTL logic simulation generates fine-grain information for each clock cycle (cc) about the functional execution of a PTP in the GPU. This simulation produces one tracing report that collects the crucial details about the PTP execution and the interaction at the HW-SW level to identify the sequence of executed instructions and the correlation with functional effects on a target module. One hardware monitor is incorporated for tracing purposes in one SM of the GPU without any effect on the functional operation of the PTP. This monitor captures the instruction opcodes coming from the fetch stage and traces the execution of the instructions in the GPU, generating a report for the hardware module under analysis. The tracing report contains the following information for each cc: the decoded instruction, the program counter value, the executed instruction per warp, the warp identifier, and the cc value.

On the other hand, the GL logic simulation executes the PTP in the GPU to extract the sequence of test patterns per clock cycle applied to the target module. These test patterns (binary values) are implicitly generated by each instruction of the PTP targeting a specific module in the GPU. The sequence of test patterns is extracted by observing the I/O switching activity in the target module under analysis. In the end, one test pattern report is generated and used in the subsequent stage.

The third stage, (3) in Figure 1, performs two steps: *i)* the fault simulation and *ii)* the instruction labeling. Firstly, one optimized GL fault simulation per PTP is performed to analyze the fault detection effectiveness of each instruction in the target module. The proposed optimized fault simulation reduces the unmanageable fault simulation effort required by big and complex designs, such as GPUs, by only selecting a target module instead of fault simulating the complete GPU.

This optimized approach takes advantage of the fact that test patterns unable to propagate fault effects to the outputs of a module are also unable to propagate these effects to the output of the complete GPU or a selected observation point of a PTP (i.e., the memory bus system in a GPU). Thus, the fault observability resorts to the outputs of the module (module-level fault observability [25]). The optimized fault simulation uses the test patterns report (generated in the previous step) as input. Moreover, one fault is detected when there is a discrepancy in the execution between the fault-free and the faulty versions of the module, since the selected observability point allows the trace of each propagated fault per cc.

The output of the optimized fault simulation is a detailed report (Fault Sim Report), which contains a list of each test pattern injected, the number of activated faults, and the number of detected faults per pattern. In this stage, a fault dropping mechanism is employed to increase the compaction rate for several PTPs devoted to detecting faults in the same module of a GPU. In this case, one fault list report is employed as a supporting mechanism to perform the compaction. This fault list report initially includes all faults of a target module. Then,

after each fault simulation (one per PTP), the fault list is updated, and those detected faults are removed from the report, so subsequent fault simulations and PTPs applied to the same module of the GPU only target those missing undetected faults.

***Instruction labeling algorithm***

```
Input: PTP composed of N instructions, Tracing clock cycle report QQ, Tracing
       program counter-report PC, tracing decoded instruction report DI, Warp
       identifier W, Fault sim test patterns report FSR
Output: Labeled parallel test program (LPTP)
for each instruction I in TP do
| LPTP{I}:=('unessential')
| CC=match (I, PC, DI)
| for each warp Wj executed by I in a block, do
| | for each clock cycle kth in Wjth do
| | | if FSRCCk detects faults, then
| | | | LPTP{I}:=('essential')
| | | | go to next instruction
| | | end if
| | loop
| loop
Loop
```

**Fig. 2**. Pseudocode of the instruction labeling algorithm.

The second step (instructions labeling) highlights the instructions of a PTP according to the observed fault detection capabilities reported during the fault simulation step. This labeling procedure tags each instruction as "*essential*" or "*unessential*" according to the analysis of the *Tracing* and *Fault Sim* reports generated in the previous stages.

Fig. 2 presents the algorithm describing the labeling procedure for a PTP targeting a hardware module in a GPU. In this procedure, each Instruction **I** in the PTP is analyzed and matched by looking at the tracing report and matching each instruction with their program counter value. This matching procedure identifies the temporal life (start/end in ccs) of each instruction.

As introduced in section II, one instruction in the PTP is executed by several groups of threads (*warps* in a *block*), Then for the $\mathbf{j_{th}}$ warp in a block ($Wj_{th}$), there is a test pattern correspondence between the $\mathbf{k_{th}}$ clock cycle (*CCk*) and the test pattern stored in the fault simulation report ($FSR_{CCk}$). Therefore, Instruction **I** is *"essential"* when its execution and its associated test pattern can detect faults in at least one of the executed warps. Otherwise, Instruction **I** is labeled with an *"unessential,"* tag as a candidate to be removed in the next stage of the compaction method. The previous analysis produces a Labeled Parallel Test Program (LPTP).

The fourth stage (test program reduction), see (4) in Fig. 1, processes and reduces the LPTP. Fig. 3 shows the reduction algorithm employed to remove unessential instructions from a LPTP. Firstly, the LPTP is divided into BBs. Each BB is divided in Small Blocks (SBs) of a sequence of instructions that comprises the load of test operands in the registers, execute an operation, and propagate the result to an observable point. Then each SB is analyzed Instruction by Instruction. One SB is removed from the LPTP when all instructions inside the SB are labeled as "*unessential*". On the other hand, those SBs containing at least one "*essential*" instruction remain unchanged for the final Compacted PTP (CPTP). It is worth noting that removing an SB may also imply the additional removal and relocation of associated input data from the main memory, which depends on the parallel kernel parameters and the location of the SB in the PTP.

Finally, the reassembling step (5) in Fig. 1, replaces the original PTPs using the generated CPTPs, so reassembling the STL. In this stage, a final fault simulation is employed to evaluate the FC features of the CPTPs in the new STL.

***Reduction Algorithm***

```
Input: Labeled Test program LPTP with N instructions divided into M consecutive BBs,
       and each BB is segmented in SBs (SB1, SB2, SB3, . . ., SBm);
Output: Compacted Parallel Test Program CPTP
for each SB in LPTP, do
| for each instruction, I in SB do
| | if the label of I is 'essential,' then
| | | append SB to CPTP
| loop
Loop
```

**Fig. 3.** Pseudocode of the reduction algorithm to remove SBs from an LPTP.

## IV. Experimental results

The proposed compaction approach was implemented as a tool written in *Python* language. This tool interacts with one logic simulator and one fault injector simulator, composing an environment to analyze and compact the GPU's STLs. Both simulators (logic and fault injector) can handle the RTL and GL description of the FlexGripPlus model, which was configured with one SM and 8 SP cores.

The simulations reports, employed during the compaction process, are generated as text files. The test patterns employed in the fault simulation of the target modules employ the VCDE format. The compaction procedures and experiments were performed on a workstation with two AMD EPYC 7301 16-core processors running at 2.2GHz and equipped with 128 GB of RAM memory.

In the experiments, one STL for GPUs was used to evaluate and validate the proposed compaction approach. The STL comprises several PTPs targeting diverse units inside the GPU, such as control units, memory modules, and functional units. In the STL, the PTPs devoted to testing the Decoder Unit (DU) and the parallel functional units occupy around 90.69% (157,113 instructions out of 173,241) of the size and 75.70% (12 million out of 16 million ccs) of the test duration in the STL. Thus, any compaction in those PTPs represents a meaningful reduction in the size and duration of the overall STL. Additionally, the programing structure employed to develop those PTPs fits the ARC definition explained in section III. It is worth noticing that 47.60% of overall faults of the GPU belong to the DU and the parallel functional units. Then the PTPs targeting these units in the GPU are good candidates to apply the compaction methodology devised in this work because they contain the most considerable size and duration of the whole STL and target a significant amount of faults of the GPU. The others PTPs are excluded of the compaction since they have been developed carefully to test control units and any instruction removal breaks the devised test algorithm.

The first set of PTPs devoted to testing the DU comprises three PTPs (IMM, MEM, and CNTRL). The IMM PTP targets the execution of all instruction formats using at least one immediate operand. This PTP also includes the Register-based instructions. Similarly, the MEM PTP is composed of instructions that perform memory accesses (global memory and

shared memory). Finally, The CNTRL PTP uses immediate-based instructions, memory-addressing instructions, and register-based instructions to generate special conditions to be used by the control flow instructions. The IMM and MEM PTPs are configured for parallel operation as one block and 32 threads per block. On the other hand, the CNTRL PTP is configured as one block and 1024 threads per block. The PTPs were developed by a specialized test engineer resorting to a pseudorandom approach using all instructions formats of the supported assembly language (Streaming ASSembler language or SASS) in FlexGripPlus.

TABLE I. MAIN FEATURES OF THE EVALUATED PTPS

| Target Module | PTP | Size (instructions) | ARC (%) | Duration (ccs) | FC (%) |
|---|---|---|---|---|---|
| Decoder Unit | IMM | 32,736 | 100.0 | 2,229,225 | 71.13 |
| | MEM | 32,581 | 100.0 | 3,186,236 | 76.59 |
| | CNTRL | 336 | 90.0 | 710,100 | 71.18 |
| | IMM+MEM+CNTRL | 65,653 | 99.0 | 6,125,561 | 80.15 |
| SP | TPGEN | 19,604 | 100.0 | 1,447,620 | 84.07 |
| | RAND | 55,000 | 100.0 | 3,434,235 | 83.99 |
| | TPGEN+RAND | 74,604 | 100.0 | 4,881,855 | 87.22 |
| SFU | SFU_IMM | 16,856 | 100.0 | 1,200,034 | 90.75 |

The second set of PTPs targets the functional units, two PTPs (TPGEN and RAND) target the SP-Cores and one PTP (SFU_IMM) targets the SFUs. The TPGEN resorts to test patterns extracted from an ATPG. A parser tool converted the ATPG test patterns into valid instructions for the GPU. The test patterns are converted partially due to a lack of fully equivalent instructions of GPU and generated patterns. RAND is a pseudorandom-based PTP specially designed to test all SP cores of any SM in the GPU. The SFU_IMM employs an ATPG tool that generates the test patterns to test the SFU; then, a parser tool converts those test patterns into GPU instructions. The kernel configuration of each PTPs is one block and 32 threads per block. Table I reports the main features of the PTPs considered for compaction

The analyzed modules in the GPU (DU, SPs and SFUs) were synthesized using the 15nm Nangate OpenCell library [26]. In the validation fault injection campaigns, 12,834, 191,616, and 180,540 faults were injected into the decode unit and functional units, respectively.

Tables II and III report the results after applying the proposed compaction approach to the considered PTPs. In both tables, the second column reports the final size for the compacted version of each PTP. Moreover, the third column reports the compaction percentage obtained for each PTP concerning their original length. Similarly, the fourth and fifth columns show the compacted PTPs and the percentage of reduced duration. Finally, the sixth column provides the difference in the FC between the original PTP and the compacted one. The last column reports the required time to perform the compaction of each PTP.

The reported results in Table II show that the compaction approach can considerably reduce the size (up to 90.36%) and duration (up to 97.84%) for the evaluated set of PTPs targeting the DU.

A deep analysis of each PTP shows that IMM, MEM, and CNTRL are composed of a regular structure of SBs, composed of a few instructions (15 to 18). More in detail, IMM and MEM have a similar size and duration, but MEM has the highest compaction rates. This behavior can be explained when considering that the compaction of MEM is performed after IMM, so those faults already detected by the IMM are discarded from MEM. Therefore, the compaction keeps only those essential instructions that can produce suitable patterns and provoke additional fault detection in the decode unit. On the other hand, the compaction was slightly less for CNTRL than in IMM and MEM. This moderate compaction is mainly caused by the original reduced number of instructions and the inadmissible region that contains conditional and control-flow instructions.

TABLE II. THE COMPACTION RESULTS IN THE TEST PROGRAMS FOR THE DECODER UNIT

| PTP | Compaction | | | | | |
|---|---|---|---|---|---|---|
| | Size | | Duration | | Diff FC | Compaction time (hours) |
| | instr | (%) | (ccs) | (%) | (%) | |
| IMM | 884 | -97.30 | 92,423 | -95.85 | +0.06 | 2.28 |
| MEM | 442 | -98.64 | 50,144 | -98.42 | -1.79 | 2.62 |
| CNTRL | 89 | -73.51 | 447,689 | -36.95 | -0.00 | 0.91 |
| IMM+MEM+CNTRL | 1,415 | -97.84 | 590,256 | -90.36 | -0.05 | 5.81 |

TABLE III. THE COMPACTION RESULTS IN THE TEST PROGRAMS FOR THE FUNCTIONAL UNITS

| PTP | Compaction | | | | | |
|---|---|---|---|---|---|---|
| | Size | | Duration | | Diff FC | Compaction time (hours) |
| | instr | (%) | (ccs) | (%) | (%) | |
| TPGEN | 4,742 | -75.81 | 452,401 | -68.75 | -1.31 | 0.28 |
| RAND | 1,215 | -97.79 | 112,030 | -96.74 | -17.07 | 1.12 |
| TPGEN+RAND | 5,957 | -92.02 | 564,431 | -88.44 | -3.13 | 1.40 |
| SFU IMM | 9,910 | -41.20 | 662,524 | -44.79 | 0.0 | 0.31 |

Regarding the FC, the compaction approach for PTPs targeting the DU has a negligible impact (reduction of up to 0.05%). However, the FC can be improved sometimes, as observed in the IMM case with +0.06% of FC. Those effects on the FC are produced by the removal of a sequence of SBs, but producing a more favorable test sequence (in case of increasing FC) or an adverse sequence (decreasing FC) of instructions for fault detection purposes.

The results in Table III show that the reported compaction rates for the execution units are outstanding and reached a size reduction of up to 97.79% for the analyzed PTPs. Similarly, the duration reduction was up to 96.74%.

It is important to underline that the compaction method successfully reduced PTPs created resorting to test patterns generated by an ATPG tool: TPGEN (75.81% in size and 68.75% in duration) and SFU_IMM (41.20% in size and 44.79% in duration). These compaction results for SFU_IMM were obtained applying the test patterns in reverse order during the fault simulation of stage 3 in our compaction approach. The selected PTPs' compaction implies 80.71% size and 64.43% duration reduction rates for the whole STL.

Interestingly, the RAND PTP compaction produces a reduction in the FC by 17.07%. This figure is due to the fault dropping performed during the previous compaction of the TPGEN PTP. This means that several instructions in RAND PTP detect some faults that also TPGEN detects; therefore, these instructions are redundant and can be removed during the compaction of the RAND PTP. The combined FC of the complete PTPs for SP-cores was affected by only 3.13%.

It is worth notice that the compaction strategy preserves the faults detection capabilities of the PTPs for the SP cores after the compaction. The FC difference is caused by changes in the computation of the signature-per-thread (SpT). The SpT is updated by the SP-cores, applying a MISR-like algorithm, taking each test operation's result. This SpT procedure detects additional faults in the SPs. Therefore, due to the compaction, the result value of one missing SB changes the SpT calculation for subsequent SBs, limiting the ability of detecting the faults previously detected.

On the other hand, the results show that for the SFU_IMM PTP, the FC is not affected by the compaction procedure. Clearly, this happens because the SFU unit performs transcendent floating-point operations, only. Therefore, removing a non-essentials SB from the SFU_IMM does not affect the fault detection capabilities of the other SBs, since there is no data dependence among SBs.

In the case of the PTPs for the DU, the compaction of the PTPs (CNTRL, IMM and MEM) required 0.91, 2.18, and 2.62 hours, respectively. In contrast, the required time to apply the compaction method on PTPs for Functional units (TPGEN, RAND, and SFU_IMM) reached only 1.71 hours.

It is worth noting that the proposed compaction method only requires a fraction of the time to analyze and process the compaction on PTPs of an STL. In the performed experiments, the PTPs included thousands of instructions. Moreover, the target modules in the GPU (DU, and functional units) contain more than a hundred thousand faults. Our compaction method only resorts to one logic and one fault simulation to perform the compaction. Previous works [13]–[16] addressed only CPUs to compact TPs in STLs using techniques that require as many fault simulations as the number of instructions in a TP. Therefore, the computational complexity and required compaction processing time of those methods are proportional to the number of fault simulations needed, usually in the order of hundreds or thousands of them.

## V. Conclusions

We proposed a compaction method to reduce the size and duration of PTPs in STLs for GPUs. According to the results, the proposed method reaches a high compaction ratio: up to 98.64% in terms of size and up to 98.42% in terms of duration when compacting PTPs with regular structure, excluding those regions with parametric loops. The compaction method showed a minimal impact on the achieved FC for the evaluated PTPs. The compaction of the selected PTPs implies 80.71% size and 64.43% duration compaction rates for the complete STL analyzed. This compaction method is suitable to reduce the size, and the duration of the target PTP, developed via pseudorandom or ATPG-based methods. The main advantage of the proposed compaction method is the limited required computational time, due to the minimum number of logic and fault simulations required to perform the compaction. This method only uses one RTL logic simulation to trace the behavior of a target PTP and one GL fault simulation to identify helpful instructions able to detect and propagate faults.

As future works, we plan to extend the compaction capabilities to more elaborated programming styles and test programs in other accelerator's and processor's architectures, as well as targeting other fault models.

## References


[1] M. Psarakis, *et al.*, "Microprocessor software-based self-testing," *IEEE Des. Test Comput.*, vol. 27, no. 3, pp. 4–19, May 2010.

[2] ARM Technologies, "Safety – Arm." [Online]. Available: https://www.arm.com/why-arm/technologies/safety. [Accessed: 11-Jun-2021].

[3] Microchip Technology Inc, "DS52076A 16-bit CPU Self-Test Library User's Guide." Microchip Technology, 2012.

[4] ST Microelectronics, "AN3307 Application note Guidelines for obtaining IEC60335 Class B certification for any STM32 application." ST Microelectronics, 2016.

[5] Infineon Technologies, "Hitex: Selftest Libraries (Safety Libs)." [Online]. Available: https://www.hitex.com/tools-components/software-components/selftest-libraries-safety-libs/. [Accessed: 11-Jun-2021].

[6] J. E. Rodriguez Condia and M. Sonza Reorda, "Testing the Divergence Stack Memory on GPGPUs: A Modular in-Field Test Strategy," *IEEE/IFIP Int. Conf. VLSI Syst. VLSI-SoC*, pp. 153–158, Oct. 2020.

[7] B. Du, *et al.*, "About the functional test of the GPGPU scheduler," *24th Int. Symp. On-Line Test. Robust Syst. Des. IOLTS 2018*, pp. 85–90, 2018.

[8] J. E. Rodriguez Condia and M. Sonza Reorda, "On the testing of special memories in GPGPUs," *26th IEEE Int. Symp. On-Line Test. Robust Syst. Des. IOLTS 2020*, Jul. 2020.

[9] S. Di Carlo, *et al.*, "An On-Line Testing Technique for the Scheduler Memory of a GPGPU," *IEEE Access*, vol. 8, pp. 16893–16912, 2020.

[10] J. D. Guerrero-Balaguera, *et al.*, "On the Functional Test of Special Function Units in GPUs," *24th Int. Symp. Des. Diagnostics Electron. Circuits Syst. DDECS 2021*, pp. 81–86, Apr. 2021.

[11] S. Di Carlo *et al.*, "A software-based self test of CUDA Fermi GPUs," *18th IEEE Eur. Test Symp. ETS 2013*, 2013.

[12] M. Abdel-Majeed and W. Dweik, "Low overhead online periodic testing for GPGPUs," *Integration*, vol. 62, pp. 362–370, Jun. 2018.

[13] M. S. Vasudevan, *et al.*, "Automated Low-Cost SBST Optimization Techniques for Processor Testing," *International Conference on VLSI Design*, 2021, vol. 2021-February, pp. 299–304.

[14] M. Gaudesi, *et al.*, "New techniques to reduce the execution time of functional test programs," *IEEE Trans. Comput.*, vol. 66, no. 7, pp. 1268–1273, Jul. 2017.

[15] A. Touati, *et al.*, "An effective approach for functional test programs compaction," *IEEE 19th International Symposium on Design and Diagnostics of Electronic Circuits and Systems, DDECS 2016*, 2016.

[16] R. Cantoro, *et al.*, "An evolutionary approach for test program compaction," *16th Latin-American Test Symposium, LATS 2015*, 2015.

[17] R. Cantoro, *et al.*, "Automated test program reordering for efficient SBST," *32nd Conference on Design of Circuits and Integrated Systems, DCIS 2017 - Proceedings*, 2018, vol. 2017-Novem, pp. 1–6.

[18] J. D. Guerrero-Balaguera, *et al.*, "A Novel Compaction Approach for SBST Test Programs," accepted in *Asian Test Symposium (ATS)*, 2021, p. 6. available in arXiv:2109.00958.

[19] J. E. Rodriguez Condia, *et al.*, "FlexGripPlus: An improved GPGPU model to support reliability analysis," *Microelectron. Reliab.*, vol. 109, p. 113660, Jun. 2020.

[20] E. Lindholm, *et al.*, "NVIDIA Tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, Mar. 2008.

[21] J. E. Rodriguez Condia and M. Sonza Reorda, "Testing permanent faults in pipeline registers of GPGPUs: A multi-kernel approach," *25th Int. Symp. On-Line Test. Robust Syst. Des. IOLTS 2019*, pp. 97–102, Jul. 2019.

[22] P. Bernardi, *et al.*, "Development flow for on-line core self-test of automotive microcontrollers," *IEEE Trans. Comput.*, vol. 65, no. 3, pp. 744–754, Mar. 2016.

[23] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.

[24] NVIDIA Corporation, "CUDA Toolkit Documentation," 2021. [Online]. Available: https://docs.nvidia.com/cuda/. [Accessed: 29-Aug-2021].

[25] J. Perez Acle, *et al.*, "Observability solutions for in-field functional test of processor-based systems: A survey and quantitative test case evaluation," *Microprocess. Microsyst.*, vol. 47, pp. 392–403, Nov. 2016.

[26] M. Martins *et al.*, "Open Cell Library in 15nm FreePDK Technology," *Symposium on International Symposium on Physical Design*, 2015.