STLs for GPUs: Using High-Level Language Approaches

This article has been accepted for publication in IEEE Design & Test. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/MDAT.2023.3267601

STLs for GPUs: Using High-Level Language Approaches

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

***Abstract*—Self-Test Libraries (STLs) are widely used for in-field fault detection in processor-based systems. Currently, their adoption is being extended to Graphics Processing Units (GPUs), due to their increasing usage in the safety-critical domain, and the demand for effective in-field functional safety mechanisms mandated by the functional safety standards (e.g., ISO 26262 for automotive). This work describes a method to develop suitable STLs resorting to High-Level Languages (HLLs) (e.g., CUDA) for some modules in GPUs, thus reducing the complexity of the STLs development process. We also discuss the method's main advantages/limitations, as well as the challenges and constraints when developing HLL STLs for GPUs. Furthermore, we outline those cases demanding the usage of Low-Level Languages (LLLs) for the implementation of STLs. The evaluation and validation of the method resort to the FlexGripPlus GPU model, implementing one micro-architecture of NVIDIA. The experimental results show that HLL STLs can be effectively developed for regular modules in the GPU.**

***Index Terms*—In-field Testing, Graphics Processing Units (GPUs), High-Level Programming Languages, Self-Test Libraries (STLs), Software-Based Self-Test (SBST).**

I. Introduction

The programming flexibility and the performance of Graphic Processing Units (GPUs) promote their adoption in the systems' domain (e.g., self-driving and robotics), where reliability and safety are sensitive issues. Both factors are fundamental when dealing with GPUs implemented using cutting-edge technology-scaling nodes. In fact, these semiconductor technologies might be affected (among the others) by permanent faults arising during the operational phase due to premature aging or wear-out [1]. Hence, suitable safety mechanisms are required to detect faults in GPUs and prevent catastrophic effects.

Among the available mechanisms, this paper focuses on functional testing solutions based on software-based techniques (Software-Based Self-Test, or SBST). This strategy is flexible and non-invasive and offers in-field testing capabilities while demanding zero hardware costs [2]. The SBST strategy is based on the development of specialized routines (*Test Programs*, or TPs) resorting to the programming capabilities (i.e., Instruction Set Architecture, or ISA) and the architecture of the device. These TPs can excite the internal faults and propagate their effects to visible locations for detection purposes. Groups of TPs compose a Self-Test Library (STL) [2]. Currently, several Intellectual Property (IP) core vendors and device developers/manufacturers offer STL solutions for their processor-based products and support their usage in safety-critical applications [3]. These STLs are integrated by the system company in the application code and activated with the required frequency. Moving to GPUs, several works already demonstrated that effective STLs can be developed, targeting functional units, memory modules, and scheduler controllers [4]–[6].

Until now, STLs for GPUs have been developed at machine level (e.g., using assembly languages, *Low-Level Languages*, or LLLs) [3]. This approach requires significant engineering effort, development time, as well as deep architecture knowledge of the target device, and is characterized by a limited re-usability. The development of STLs using LLLs (LLSTLs) is particularly demanding for GPUs, due to the difficulties in handling implicit parallelism at the fine-grain level. As a result, the design, implementation, and validation of TPs for GPUs is highly critical, due to the lack of detailed information about the ISA, so involving long development times, even when assisted by specialized tools. The structural and ISA differences between GPU products also significantly reduce the portability of LLSTLs. In contrast, HLLs for GPUs (such as *CUDA* or *OpenCL*) simplify the programming effort and are the best (*sometimes the only!*) method to develop and encode applications. In fact, adopting HLLs (instead of LLLs) may increase the programmer's productivity (from 3 to 10 times depending on the target platform [7], [8]). Moreover, the increasing complexity of modern GPUs can be handled straightforwardly with HLLs, allowing an easier maintenance process. Additionally, some features of HLLs, such as the programming scalability and flexibility, encourage test specialists to adopt HLLs for STL's development. Unfortunately, there are still several challenges and open questions when resorting to HLLs for the development of TPs and STLs (leading to HLTPs and HLSTLs).

In the literature, some authors explored the development of HLSTLs for GPUs. In [6], several OpenCL microbenchmarks were discussed to test intermittent faults produced by the device's stress or temperature changes. On the other hand, authors in [5] combined CUDA with an Intermediate Language (IL) ISA (PTX) to test permanent faults in functional units and register files. Unfortunately, none of the mentioned works addresses other sensible modules inside the GPU, such as controllers, memories, and pipeline registers, nor discusses the compiler intervention during the test.

This paper extends our preliminary work [9], in which we introduced a method to develop HLSTLs for GPUs. We now extend and generalize the methodology and analyze the

main benefits and constraints when using ILs (i.e., virtual assembly languages, such as PTX or AMD IL) to support the development of HLSTLs. In detail, the method pursues the complexity reduction in the development flow of STLs using LLLs. Furthermore, we describe and analyze the main advantages and constraints when developing TPs adopting HLLs, LLLs, ILs, or a combination of them when required. The main contribution of this work can be summarized as follows:

- The description of a method to develop suitable STLs targeting the detection of permanent faults and resorting to HLLs and ILs in selected GPU units.
- The identification of challenges and constraints when developing HLTPs and HLSTLs for GPUs.
- The coding guidelines to suppress the main limitations when adopting HLLs and ILs to develop TPs and STLs for GPUs.

The experimental validation resorts to one microarchitectural GPU model (FlexGripPlus [10]) and two GPU devices (NVIDIA's Jetson Nano, and GeForce GTX 960M). The results show that HLTPs and HLSTLs are suitable when targeting regular units (e.g., those composing the datapath of a GPU core, such as the functional units and the register file). On the other hand, a combination of different abstraction levels (HLLs, ILs, and LLLs) is necessary to develop STLs for other modules, such as controllers and programmer's hidden units, due to the compiler features, observability constraints, and architectural characteristics of such units.

In this work, we used NVIDIA's concepts and tools to develop and validate the proposed method. However, the proposed techniques and results can be adapted and extended into other GPU architectures.

II. Organization of GPUs

A. GPU Architecture

In NVIDIA's terminology, GPUs are massively parallel processors organized as cluster arrays of units called 'Streaming Multiprocessors' (SMs). An SM implements the Single-Instruction Multiple-Thread (SIMT) paradigm (an extension of 'Single-Instruction Multiple-Data', or SIMD). Each SM includes several functional units (*'Streaming Processors'* or SPs) to perform the same operation in parallel on several threads, as well as supplementary units, such as Special Function Units (SFUs) and Tensor Core Units supporting multimedia and artificial intelligence applications.

Moreover, the GPU's memory hierarchy is organized nto several levels, such as a General-Purpose Register File (GPRF), a shared memory, a local memory, a constant memory, and an external global/main memory. Additionally, the GPU includes several hidden units, such as the pipeline registers, located across the architecture and storing sensitive information for its operation.

The execution of a parallel program (*kernel*) in a GPU is orchestrated by hierarchical hardware-based schedulers handling the thread execution and managing divergent paths among the threads inside each SM. The schedulers fetch, decode, and execute kernel instructions in parallel using the available cores. This execution flow is conducted in small parallel units (e.g., 32 or 48 threads) called *warps*.

B. GPU programming model

The GPU's programming model is based on environments (such as the 'Compute Unified Device Architecture' or CUDA) abstracting and hiding the compilation and programming complexity, and providing code flexibility and portability (compatibility across generations), so simplifying the programmer's task. Other abstraction levels, such as ILs and virtual Instruction Set Architectures ('vISAs'), offer the flexibility and portability benefits of HLLs, and the fine-grain control of a LLL. Thus, a kernel requires minimal description changes to be used in different GPU architectures, keeping the same programming model of previous hardware versions. Clearly, an optimized backend compiler is used.

In detail, the program's compilation requires two phases: *i)* from HLL to IL (also known as Parallel Thread Execution languages or *'vISA'*), and *ii)* from IL to LLL (machine level or *'mISA'*). In the first phase, most compiler settings (e.g., optimizations flags, maximum register usage, granted fast math features, etc.) take effect to increase the performance in the parallel execution, so creating a compressed version of the HL program at the IL-level (e.g., PTX). The second phase uses the IL program to produce a device-specific program at the mISA level and its binary executable, which may change among GPU generations and devices. In the second phase, additional compiler optimizations include: *i)* implicit management of memory resources, *ii)* software-based out-of-order organization of mISA instructions, and *iii)* selection of suitable instructions to perform an intended functionality. In the first case, the compiler evaluates the performance trade-off and selects the best memory resource for operand placement (e.g., small arrays are stored in the local memory). Similarly, the compiler organizes unrelated instructions, removing or modifying the sequence of instructions in a program but preserving the intended functionality. Unfortunately, these compiler's optimizations are restrictions when focusing on testing objectives.

III. A Method for Developing Parallel HLSTLs

The proposed method exploits the divide-and-conquer strategy by splitting the architecture of a GPU into modules developing individual STLs by resorting to HLLs and ILs. Fig.1 depicts the flow of the proposed method, which includes three main steps: *1)* Modular test generation, *2)* Programming language mapping, and *3)* Test program evaluation.

A. Modular test generation

This step evaluates *i)* the functional characteristics of the unit (e.g., arithmetic, control, or storage), *ii)* the architecture specifications inside the GPU (e.g., data path or control path units), and *iii)* the controllability and observability constraints (i.e., how the program instructions can activate the unit). These attributes are crucial to identify proper test methods (e.g., automatic, deterministic, or custom) [4] as well as their

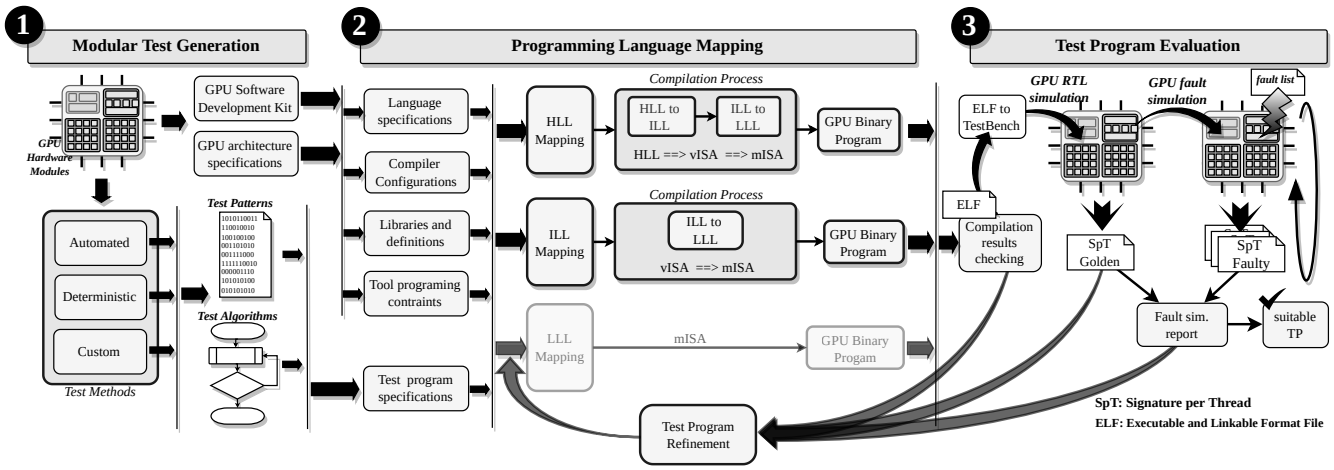

Fig. 1. A general scheme of the proposed methodology to generate HLL TPs and to translate LLL into equivalent HLL STLs

implementation feasibility at HLTPs or ILTP. For testing purposes, each module can be classified into one of the following three types: *1)* Regular, *2)* GPU's distinctive, and *3)* Hidden structures.

Regular modules correspond to replicated units inside the architecture, which perform the same functionality in parallel (i.e., functional units and register files). These structures (belonging to the data-path) are visible resources of the GPU, allowing the application of any test data at the input of the unit (i.e., high controllability) and then observing its results after issuing an operation (i.e., high observability). Automated or deterministic testing methods are suitable approaches for them.

GPU's distinctive modules correspond to special modules (e.g., schedulers, divergence controllers, and decoders) located both in the data path and control path of the GPUs. Their functional features and complex organization impose testing restrictions (i.e., low controllability or observability), demanding additional efforts to generate and apply the test data as well as to guarantee the propagation of the fault effects up to an observable point. Combinations of automated, deterministic, and custom methods are suitable testing solutions for these units.

Finally, the third unit type corresponds to hidden modules for the programmer, which refer to hardware structures invisible to the programmer (e.g., pipeline registers and embedded controllers). The inherent complexity of these units demands customized testing algorithms and especially elaborated test methods.

B. Programming Language Mapping

Mapping TP specifications into HLLs can be easier and faster since high-level constructs (e.g., nested *for* loops) can describe complex specifications with a reduced engineering effort. Unfortunately, mapping effective test algorithms into HLLs or ILs is not a straightforward task since compilation stages apply optimizations seeking the maximum execution performance and security of the device. In fact, HL optimizations are in opposition to testing objectives by limiting the programmer's control. Nevertheless, these compiler constraints can be avoided by adopting adequate coding styles and compilation settings for test purposes.

In the proposed method, a bottom-up approach evaluates and translates TP specifications as one or a set of test routines in the adopted programming language (e.g., HLL, IL, or LLL). For this purpose, several iterations might be needed to verify the testing features of the HLTPs. Table I summarizes the mapping strategies that can be adopted to implement test methods into HLLs and ILLs for a given GPU module. The reported information covers diverse modules and test methods organized by unit type, GPU module, test method, detailed test strategy, HLL mapping strategy, and ILL mapping strategy. For example, a GPU's divergence stack memory module (DSm) handles the thread divergence operations in the SM. The test of this unit resorts to several deterministic algorithms implementing control-flow operations addressing each location in the stack. There are two specific tests strategies adopted for this module: *i)* pyramidal nested divergency management and *ii)* synchronism management (SyncTrick) [11]. The first test strategy requires indirect usage of conditional branches, controlled divergences, nested divergences, and function calling, forcing the device to address a new line in the stack.

These TP specifications are mapped into an HLL or ILL, taking care of the following considerations: *i)* definition of operands location (i.e., main, shared, or constant memory). It is worth noting that some of the operands can be defined as literals embedded in the code for the HLL mapping or immediate operands in the case of ILL mapping. *ii)* Repeat a consecutive evaluation of nested conditional statements for each thread in a warp and for each available stack entry. For the HLL mapping case, the STL developer can resort to "if" statements. On the other hand, the ILL mapping implements a combination of control-flow vISA instructions (i.e., comparison instructions, branch instructions, and predicated instructions). *iii)* For each conditional path, a Signature-per-Thread (SpT) mechanism [12] is adopted to enhance test observability and efficiently report the test results.

Fig. 2 shows a snippet sample code describing the sequence of conditional statements implemented in CUDA and PTX, respectively. The mapping of the second test solution (syncTrick)

```
1   __global__ void Warp_Scheduler_T(int* SpT, int* vars){
2   ...
3   int Tid = blockDim.x * blockIdx.x + threadIdx.x;
4   ...
5   if (Tid == vars[0]) {
6   ... }                    ⇐ SpT_update(fault-free value);
7   else{
8   ... }                    ⇐ SpT_update(fault value);
9   ...
10  if (Tid == vars[n]) {
11  ... }                    ⇐ SpT_update(fault-free value);
12  else{
13  ... }                    ⇐ SpT_update(fault value);
14  }
```

(a)

```
1   .entry Warp_Scheduler_T(.param, .u64 SpT .param .u64 vars){
2   ...
3   cvt.u32.u16 %r1, %tid.x;
4   mov.u16 %rh1, %ctaid.x;
5   mov.u16 %rh2, %ntid.x;
6   mul.wide.u16 %r2, %rh1, %rh2;
7   add.u32 %r3, %r1, %r2;
8   mul.wide.s32 %rd2, %r3, 4;
9   ...
10  ld.param.u64 %rd3, [vars];
11  add.u64 %rd4, %rd3, %rd2;
12  ld.global.u32 %r5, [%rd4+0];
13  setp.eq.u32 p, %r3, %r5;
14  @p bra SpT_update_FF;
15  <SpT_update_faulty_path_ops>
16  bra continue;
17  SpT_update_FF:
18  <SpT_update_fault_free_path_ops>
19  continue:
20  ...
21  add.u64 %rd4, %rd4, N;
22  ld.global.u32 %r5, [%rd4+0];
23  setp.eq.u32 p, %r3, %r5;
24  @p bra SpT_update_FF;
25  <SpT_update_faulty_path_ops>
26  bra continue;
27  SpT_update_FF:
28  <SpT_update_fault_free_path_ops>
29  continue:
30  ...
31  }
```

(b)

Fig. 2. An example of mapping the test strategies for the Divergence Stack into HLLs and ILLs. (a) CUDA implementation (b) PTX implementation

is not feasible at HLL (even using ILs) since the explicit use of some control-flow mISA instructions to optimize the stack addressing is not allowed at these programming levels. After compilation, the binary executable object is used to validate and evaluate the test capabilities of each TP.

C. Test Program Evaluation

This step validates the functionality of a TP or STL (as a binary executable). This validation is divided in three stages: *i)* Compilation results checking, *ii)* Test Program validation, and *iii)* Test program refinement, as depicted in Fig. 1. The first stage verifies the compilation results by checking the content of an Executable and Linkable Format (ELF) file. This file contains information about the device's resources usage that allows identifying significant compiler optimizations for a TP or STL, which may lead to removing, compaction, or replacing testing features (i.e., conditional statements generating TPats or different data allocation).

In addition, this step considers further checks to programming structures and mISA instruction formats (i.e., call to routines, miscellaneous instructions, etc). When the initial checking does not succeed, the TP requires improvements through a refinement process.

The second stage (*Test program validation*) is divided into two sub-stages: logic simulation and fault simulation. The logic simulation verifies the correct functional execution of a TP using an RTL GPU model. Firstly, the ELF file is transformed into a GPU model-compliant Test-Bench containing the TP's information. Secondly, the GPU model executes the TP and captures the Signature per Thread (SpT) that indicates the fault-free status of the GPU and, in turn, serves to verify the program's correct operation. The fault simulation resorts to a customized simulation environment that takes the GPU's RT-level model and evaluates the Test Programs, targeting one GPU module and injecting stuck-at-faults (SAFs), one at a time. The fault simulation considers a fault as detected when at least one mismatch exists between the fault-free and the faulty SpT.

Finally, a given TP is considered valid if it is compliant with the TP specifications and fulfills minimum fault detection capabilities. Otherwise, the refinement step is used to provide changes to the algorithm, described functions, or compiler settings to improve the fault coverage.

D. Defeating compiler and architectural constraints for testing purposes

Developing STLs using HLLs or ILs presents challenges related to the constraints imposed by the compiler and the structural features of certain GPU modules. In order to solve such mapping constraints, several techniques can be applied to reduce or bypass the compiler optimization's effects and preserve testing capabilities in a TP. A first strategy requires the adoption of adequate coding styles to force the compiler to preserve test functionalities. These techniques include the efficient use of *'device intrinsic functions'* from libraries (e.g., *math.h*). For example, HLTPs for SFUs require intrinsic functions to guarantee the generation of mISA instructions addressing the modules and applying the desired Test-Patterns (TPats). Fortunately, in the case of IL instructions for the SFU, these are directly mapped as vISA ones.

Other techniques manipulate the arguments (inputs/outputs of a kernel) to preserve instructions or routines targeting GPU module. e.g., testing the GPRF requires several individual arguments (from 3 to 127) to force the compiler to allocate and address all possible registers per thread. Then, these arguments are loaded with external patterns. Similarly, testing the Scalar-Proessors (SPs) include additional arguments to generate most mISA formats and preserve TPats. Moreover, reducing the use of local variables and augmenting the input arguments in a kernel, help to suppress the instruction's replacement and the out-of-order organization on HLTPs. Alternatively, including explicit references to memory (e.g., global or shared) in at least one of the HLL's or ILL's operands, prevents the compiler to optimize the usage of immediate operands. In addition, the manipulation of variables in the shared memory reduces the compression, replacement, and reordering of instructions during compilation.

Other strategies require the creation of data dependencies between consecutive operations/instructions, the usage of the global memory to store partial results, and barrier synchronizations force the compiler to remove instructions to assure that processed values are available for the next instructions. It is worth noting that these strategies can be used individually or in combination with mapping test specifications as HLTPs and ILTPs.

TABLE I
SUMMARY OF THE MAPPING STRATEGIES FROM THE TEST ALGORITHM TO HIGH-LEVEL LANGUAGES (HLLS) AND INTERMEDIATE LEVEL LANGUAGES (ILLS)

Unit Type	Module	Test Method	Characteristics of the Test Strategy	HLL mapping(Note)	IL mapping(Note)
Regular	FU	A	• Tpats generation (ATPG/Pseudorandom) • Tpats processing as operations and operands • Operations grouping by type (logic, arithmetic) [4]	• The mapping is data-size aware, (8bit, 16bit, or 32bit) • Sequence of (+,-,*,/,%) operations, or math.h functions • Insert intrinsic functions (e.g., SFU operations) • Update an SpT status	• The mapping is data-size aware, (8bit, 16bit, or 32bit) • Explicit operands load from memory to registers • Consecutive vISA instructions (logic, arithmetic) • Insert dedicated SFU vISA instructions • Update an SpT status
	GPRF	A/D	• MARCH algorithm/ Custom method for SAFs as embarrassingly parallel function	• Declaration of local variables as many as HW registers • Consecutive R/W operation on local variables • Update an SpT status	• Declaration of virtual registers, as many as HW registers • Explicit operands load from memory to registers • Consecutive R/W on the virtual registers • Update an SpT status
Specific	PRF	D	• Controlled divergence management based on Tpats (e.g., checkerboard) as an embarrassingly parallel function [4] • Tpats transformation: operand values and relational operators	• Consecutive evaluation of conditional 'if' statements • Update an SpT status on each divergence path	• Explicit operands load from memory to registers • Consecutive control-flow vISA instructions • Update an SpT status on each divergence path
	ARF	D	• MARCH algorithm	Due to functional and language abstraction constraints, it's not possible to map the method at this level.	
	WSm	D	• Warp divergency management and routine placement [4]	• Local and shared variables for enabling/disabling warps • Loop over all warps and threads identifiers • Evaluate nested 'if' statements for warps and threads • Update the SpT on each divergence path • Update local and shared variables (software barrier) • Ending loop	• Explicit operands load from memory to registers • Virtual and private registers for enabling/disabling warps • vISA based loop implementation over warps and threads • Nested control-flow vISA instructions per warp per thread • Update the SpT on each divergence path • Insert barrier vISA instructions • Update virtual and private registers
	DSm	D	• Pyramidal nested divergency management: Consecutive nested divergences for stack addressing [4] • Synchronism management (SyncTrick): exploit synchronism for stack addressing [11]	• Repeat for threads in a warp and stack entries: • Consecutive conditional statements ('if') for operands • Update the SpT on each divergence path • SyncTrick mechanism can't be mapped	• Repeat for threads in a warp and stack entries: • Consecutive control-flow vISA instructions for operands • Update the SpT on each divergence path • SyncTrick mechanism can't be mapped
	DU	A/D	• Pseudorandom approaches for logic-arithmetic and memory-based operations. • Deterministic control-flow operations in combination with miscellaneous operations to build embarrassingly parallel functions [4]	• Various thread indexing implementations • Consecutive operations with different data types • Consecutive data type conversion and casting • R/W operations in shared. main and constant memories • Consecutive conditional 'if' statements • Unconditional branching by nested function invocations • Update the SpT after each operation • Miscellaneous operations cannot be mapped at this level	• Explicit memory address calculations • Load operands from memory to local registers • Consecutive vISA instructions with different data types • Interleave control-flow vISA instructions • Implement nested subroutine calls (unconditional branching) • Update the SpT after each vISA instruction • Include miscellaneous vISA instructions if available
Hidden Units	PR	C	• Multi Kernel approach: particular test methods addresing different hidden elements of the targeted structures [12]	• Several cooperative thread array configuration per function • The function descriptions follow the methods previously described for the other units	

A: Automated; D: Deterministic; C: Custom
FU: Functional Units; GPRF: General-Purpose Reg. File; PRF: Predicate Reg. File; ARF: Address Reg. File; Wsm: Warp Scheduler Memory; DSm: Divergence Stack; DU: Decoder Unit; PR: Pipeline Reg.
Note: Kernel function must have input arguments; Operands allocated in memory (main, shared, constant); Immediate or literal operands are allowed; At least one operand must be a memory reference

TABLE II
GPU MODULES FEATURES AND STL DEVELOPMENT APPROACH

Unit Type	Module	Num of cells (*)	Test Method	STL Mapping		
				CUDA	PTX	SASS
Regular	Scalar Processor (SP)	206,824	A	F	F	F
	Special Function Units (SFU)	90,982	A	F	F	F
	General-Purpose Reg File (GPRF)	524,288	D	F	F	F
	Predicate Reg File (PRF)	16,384	D	P	P	F
	Address Reg File (ARF)	131,072	D	-	-	F
Specific	Warp Scheduler mem (WSm)	5,118	D	P	P	F
	Divergency Stack mem (DSM)	273,600	D	P	P	F
	Decoder Unit (DU)	1,896	A D	P	P	F
Others	Pipeline Regs (PRs)	2,382	C	P	P	F

(*) Combinational and sequential cells using the synthesis library 15nm NanGate OCL
(F) Test algorithm fully mapped into the target programming language
(P) Test algorithm Partially mapped into the target programming language
(-) Test algorithm not mapped into the target programming language
A: Automated; D: Deterministic; C: Custom

TABLE III
MAIN FEATURES OF THE IMPLEMENTED STLS FOR REGULAR UNITS OF THE GPU

GPU Module	HLSTLs (CUDA)			ILSTLs (PTX)			LLSTLs (SASS)		
	Duration (cc)	Size (instr)	FC (%)	Duration (cc)	Size (instr)	FC (%)	Duration (cc)	Size (instr)	FC (%)
SP	5,366,208	76,513	86.95	5,922,414	1079	81.94	4,881,855	74,604	87.20
SFU	1,331,200	16,856	94.30	212,914	117	94.30	212,914	117	94.30
GPRF	3,256,058	698	100.00	-	-	-	108,958	82	100.00
ARF	-	-	-	-	-	-	338,240	122	100.00

IV. EXPERIMENTAL RESULTS

The FlexGripPlus GPU model was used to validate the proposed approach, and several *'High-Level-Test-Programs'* HLTPs and *'Intermediate-Level-Test-Programs'* ILTPs were developed targeting different modules inside it. The effectiveness of the developed TPs has been evaluated through fault simulation experiments resorting to commercial EDA tools and considering the stuck-at-fault (SAF) model. These fault simulation campaigns 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. The FlexGripPlus GPU was configured with one SM, 8 SPs, and 2 SFUs.

The HLTPs and ILTPs were written in CUDA and PTX and compiled using CUDA toolkit SDK 5.0 with a Compute Capability 1.0. Additionally, two GPUs (NVIDIA Jetson Nano and GeForce GTX 960M) were employed to evaluate the TP's execution and observe the compilation impact of various coding styles in different environments (CUDA SDK 11.2 and CC 5.3, and CUDA SDK 5.0 and CC 5.1).

Table II reports the evaluated modules inside one SM of the GPU, their size, the used test method for TP development, and the mapping features as HLLs, ILs and LLLs. Table III reports the implementation details and the Fault Coverage (FC) results of the developed and implemented STLs in regular units. The results show the effectiveness of STLs using HLL and IL for units that are fully controllable and observable (e.g., SPs and GPRF).

The duration of the HLSTLs (CUDA) is longer than its equivalent LLSTL (SASS) version (from 1.1 to 6 times for SPs and SFUs). This cost can be explained by the required memory operations (reading and writing) in the HLTPs to prevent compiler optimizations. Moreover, the TPats are replicated inside each block of threads to produce redundancy in the operations inside the SM, avoiding the scheduling intervention during the test of these units. Similarly, the ILSTL duration is longer than the equivalent LLSTL for 'Scalar-Processors' SPs.

TABLE IV
MAIN FEATURES OF THE IMPLEMENTED STLS FOR THE SPECIAL UNITS OF A GPU

GPU Module	CUDA			PTX			SASS		
	Duration (cc)	Size (instr)	FC (%)	Duration (cc)	Size (instr)	FC (%)	Duration (cc)	Size (instr)	FC (%)
WSm	98,480	276	38.20	-	-	-	112,200	392	100.00
DU	2,150,612	12,354	68.75	1,589,678	12,116	73.74	6,125,561	65,653	80.10
DSM	987,526	875,422	35.10	-	-	-	1,030,473	12,524*	98.40
PRF	1,750,023	392	28.00	-	-	-	1,890,106	434	100.00
PRs	649,400	22,292	80.20	-	-	-	1,204,097	27,492	95.10

(*) The use of SASS instructions allowed a significant reduction in the total size of a TP

However, the required number of instructions in ILSTLs is significantly reduced, since the TP description in PTX allows more friendly fine-grain management than mISA descriptions, allowing the selection of memory and immediates for operand storage. This memory management also increases the degree of parallelism in PTX, helping the TPs' size reduction. For SFUs, ILTPs description is simple, generating a program remarkably similar to the corresponding LLTP. Notably, the achieved FC in HLTPs and ILTPs for SP testing can sometimes be moderately lower than that achieved by LLTPs. The main responsible for the slight FC reduction is the SpT computation, which uses logic/arithmetic instructions in the SPs. Although the SpT algorithms encoded in HLTPs and ILTPs are functionally equivalent, the compiler produces different SpT versions in comparison to those in LLTPs. Hence, the TP's FC capabilities vary (the patterns produced by the SpT on SPs strongly depend on the mISA instructions used for that purpose).

Table IV reports the results for the distinctive units and the pipeline registers in a GPU. As the reader can notice, the HLTPs have limited testing capabilities (about 28% to 68.75% FC) due to the partial mapping of the test algorithms in HLLs, along with the compilation impact. Indeed, the test of these modules resorts to specific algorithms, which are functionally characterized by low performance. Thus, the compiler modifies the code, following unavoidable optimization philosophies. In fact, the compiler produces a functional HLTP, but removes or changes the execution order of operations, so affecting the FC and producing a negative impact on the testing capabilities of HLTPs. Nevertheless, a hybrid approach (combining HLTPs with manually added mISA instructions) increases their FC to acceptable values (100% for PRFs and up to 98.41% for the DSM). These instructions are required to inject patterns by addressing a module in specific conditions (i.e., addressing memory locations or addressing different stack lines in the DSM). It is important to mention that for the DSM, the final insertion of mISA instructions also reduced the routine complexity, so compacting the TP. Nonetheless, the test engineer decides when such manual addition of instructions is justified considering the tradeoff between productivity, the effort for the test program generation, and the test coverage improvement.

On the other hand, the ILTPs to test the WSm, DSM, PRF, and PRs produced identical results in terms of duration, size, and FC to the ones by HLTPs. This characteristic behavior obeys mainly to the deterministic nature of the test methods used to test such modules, (e.g., based on specific operations, such as conditional statements to induce controlled divergence). However, the ILTPs for the DU reach higher coverage (73.74%) than equivalent HLTPs (68.75%). In fact, the direct usage of some miscellaneous and control-flow instructions at PTX level provides fine-grain control to produce TPats which

cannot be generated using CUDA.

Finally, the TPs implemented using only HLLs or ILs can test around 50.6% of the faults, and represent 9.3% of the size of the STLs. Moreover, hybrid HLTPs (improved by additional mISA instructions) can test 45% of the faults, and occupy 90.6% of the size of the STLs. Additionally, based on our experience, the adoption of HLLs for STL's development decreases the development time by about two orders of magnitude (for the functional units), and one order of magnitude (for other modules), resorting to the combination of CUDA and SASS. Actually, this time reduction is aligned with the statement [7], [8] regarding improving programmer's productivity by adopting HLLs, so demonstrating that this productivity improvement is achievable in STL development for GPUs, too. On the other hand, the adoption of ILs for STL development offers higher flexibility, reducing the development effort by around 30% w.r.t. direct machine assembly implementations of the same TP.

V. Conclusions

This work introduces a method to develop STLs for in-field GPU testing resorting to High-level and Intermediate-level Languages. The proposed method employs a divide-and-conquer approach to target individual modules in a GPU and applies specific strategies that, in some cases, can later be mapped into high-level functions. Constraints and challenges in the development of high-level STLs require facing some implicit limitations, such as the controllability and observability of the different modules, and the optimizations of the several layers of compilation in a GPU. More in detail, the compiler constraints can be faced using strict coding styles or combining several abstraction levels to develop effective TPs and STLs. Finally, the adoption of HLLs when developing STLs for GPUs also plays an important role when finding the best trade-off between time development and execution time of the STLs.

References

[1] S. Hamdioui *et al.*, "Reliability challenges of real-time systems in forthcoming technology nodes," in *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2013, pp. 129–134.

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

[3] P. Viswanathan *et al.*, "State of the art software test libraries (stl) and asil b: Truths, myths, and guidance," ARM-Technologies, Tech. Rep., 2022, accessed: Sept. 20, 2022.

[4] J. E. Rodriguez Condia *et al.*, "Using stls for effective in-field test of gpus," *IEEE Design & Test*, pp. 1–6, 2022.

[5] S. Di Carlo *et al.*, "A software-based self test of cuda fermi gpus," in *2013 18th IEEE European Test Symposium (ETS)*, 2013, pp. 1–6.

[6] D. Defour and E. Petit, "Gpuburn: A system to test and mitigate gpu hardware failures," in *2013 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, 2013, pp. 263–270.

[7] L. Hochstein *et al.*, "Parallel programmer productivity: A case study of novice parallel programmers," in *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, 2005, pp. 35–35.

[8] M. M. Trompouki and L. Kosmidis, "Brook auto: High-level certification-friendly programming for gpu-powered automotive systems," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, 2018, pp. 1–6.

[9] J.-D. Guerrero-Balaguera *et al.*, "A new method to generate software test libraries for in-field gpu testing resorting to high-level languages," in *2022 IEEE 40th VLSI Test Symposium (VTS)*, 2022, pp. 1–7.

[10] J. E. R. Condia *et al.*, "Flexgripplus: An improved gpgpu model to support reliability analysis," *Microelectronics Reliability*, vol. 109, p. 113660, 2020.

[11] J. E. Rodriguez Condia and M. Sonza Reorda, "Testing the divergence stack memory on gpgpus: A modular in-field test strategy," in *2020 IFIP/IEEE 28th International Conference on Very Large Scale Integration (VLSI-SOC)*, 2020, pp. 153–158.

[12] Rodriguez Condia, Josie and Sonza Reorda, Matteo, "Testing permanent faults in pipeline registers of gpgpus: A multi-kernel approach," in *2019 IEEE 25th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, 2019, pp. 97–102.

Juan-David Guerrero-Balaguera is pursuing a Ph.D. in the Department of Control and Computer Engineering of Politecnico di Torino, Italy. He received the M.Sc. degree in electronic from Universidad Pedagógica y Tecnológica de Colombia (UPTC). His research interests include functional test, Artificial Intelligence, Parallel architectures.

Josie E. Rodriguez Condia received the Ph.D. degree in Computer Engineering from Politecnico di Torino, Italy in 2021, and the M.Sc. degree in electronic from Universidad Pedagógica y Tecnológica de Colombia (UPTC), Colombia in 2017. His research interests include functional test, parallel architectures, and embedded system design.

Matteo Sonza Reorda is a full professor in the Department of Control and Computer Engineering of Politecnico di Torino, Italy. He received the PhD degree in computer engineering in 1990 from the same institution. His research interests include design and test of reliable electronic circuits and systems. He is a Fellow of IEEE.