Revealing GPUs Vulnerabilities by Combining Register-Transfer and Software-Level Fault Injection

Revealing GPUs Vulnerabilities by Combining Register-Transfer and Software-Level Fault Injection

Fernando F. dos Santos[†] Josie E. Rodriguez Condia[*] Luigi Carro[†] Matteo Sonza Reorda[*] Paolo Rech[*]
[†]PPGC, Institute of Informatics, Universidade Federal do Rio Grande do Sul, Porto Alegre, Brasil
{ffsantos | carro}@inf.ufrgs.br
[*]Department of Control and Computer Engineering, Politecnico di Torino, Torino, Italy
{josie.rodriguez | matteo.sonzareorda | paolo.rech}@polito.it

***Abstract*—The complexity of both hardware and software makes GPUs reliability evaluation extremely challenging. A low level fault injection on a GPU model, despite being accurate, would take a prohibitively long time (months to years), while software fault injection, despite being quick, cannot access critical resources for GPUs and typically uses synthetic fault models (e.g., single bit-flips) that could result in unrealistic evaluations.**

This paper proposes to combine the accuracy of Register-Transfer Level (RTL) fault injection with the efficiency of software fault injection. First, on an RTL GPU model (FlexGripPlus), we inject over 1.5 million faults in low-level resources that are unprotected and hidden to the programmer, and characterize their effects on the output of common instructions. We create a pool of possible fault effects on the operation output based on the instruction opcode and input characteristics. We then inject these fault effects, at the application level, using an updated version of a software framework (NVBitFI).

Our strategy reduces the fault injection time from the tens of years an RTL evaluation would need to tens of hours, thus allowing, for the first time on GPUs, to track the fault propagation from the hardware to the output of complex applications. Additionally, we provide a more realistic fault model and show that single bit-flip injection would underestimate the error rate of six HPC applications and two convolutional neural networks by up to 48% (18% on average). The RTL fault models and the injection framework we developed are made available in a public repository to enable third-party evaluations and ease results reproducibility.

***Index Terms*—Fault injection, Graphics Processing Unit (GPU), Reliability**

I. Introduction

The computational characteristics, efficiency, and flexibility of modern Graphics Processing Units (GPUs) have pushed their adoption in High Performance Computing (HPC) and safety-critical applications, such as autonomous vehicles for the automotive and aerospace markets. This market shift from consumer applications has suddenly pushed the interest, and posed questions, about GPUs reliability.

GPU vendors have worked to improve their devices' reliability by designing more robust memory cells [1] and in the qualification of platforms compliant with strict automotive reliability standards as the ISO26262 [2]. In the meanwhile, the research community has been carefully studying GPUs reliability through fault injection/simulation [3]–[8] or beam experiments [9], [10].

One of the main issues related to the characterization of complex devices, such as GPUs, lies in the conflicting limitations imposed by the available reliability evaluation methodologies. These methodologies are either realistic and exhaustive but offer limited visibility (beam experiments), have full visibility but are extremely time-consuming (circuit or gate-level fault simulations), or are fast and cheap, but inject synthetic fault models in a limited set of programmer accessible resources (software fault injection). The single/double bit-flip model, adopted in most software fault injectors, accurately represents only faults in the memory resources, which are the ones that can be easily protected with ECC. Unfortunately, a fault in unprotected and hidden to software resources, such as pipeline registers, ALU, and peculiar GPU modules (scheduler or control units), might have a not-obvious impact on the operation(s) output that we intend to characterize.

Inspired by previous works that define the concept of multi-level or hybrid fault injection [11]–[18], we propose to combine Register-Transfer Level (RTL) evaluation on a GPU model (FlexGripPlus [19]) with software fault injection in a real GPU (NVIDIA Volta). The time required to have an RTL reliability evaluation of highly complex codes is exacerbated by the number of available resources in modern GPUs. For instance, an RTL fault injection, limited to a relatively small module as the GPU scheduler, would require more than 720 hours using a 12 nodes server to characterize LeNET, the simplest Convolutional Neural Network (CNN). Our idea is to characterize, with the GPU RTL model, the effect of a transient fault in unprotected resources (we do not consider the ECC protected memories) in the execution of the most common *SASS instructions* (i.e., machine operations that are effectively executed in the NVIDIA GPU hardware) rather than of a whole code. Additionally, we have characterized a *tiled* Matrix Multiplication (t-MxM) mini-app for its importance in CNN's execution [20], [21].

We measure the impact of the generic RTL fault, that we call *fault syndrome*, on the output value of 12 instructions (and the mini-app) executed with three input ranges. Then, we use the efficiency of a specially crafted version of NVBitFI [3] software fault injector to inject the most suitable fault syndrome in real-world applications (we consider 6 HPC applications and 2 CNNs). This strategy reduces the time required to have a detailed reliability evaluation of complex applications from the tens of years an RTL fault injection would need to just tens of hours.

Thanks to our framework, we are able, for the first time, not only to unveil the effects of faults on otherwise hidden GPU resources, but also to present a more detailed fault model to be used in the reliability evaluation of complex codes. Additionally, we can identify the hardware source of those faults that are more likely to propagate to software visible states and to the application output inducing, for instance, misdetections in CNNs. This information is precious, as it helps researchers to focus the design of a hardening solution to a subset of critical resources. The RTL fault injection highlights that functional units are the most probable source for data errors, identifies a subset of 16% of pipeline registers as responsible for the vast majority of Detected Unrecoverable Errors (DUEs), and confirms previous assertions about the scheduler corruption leading to multiple corrupted threads [9], [22]. We found that the fault syndrome at the instruction output does not follow a uniform distribution, but rather a power law (few effects are extremely probable). Finally, we see that scheduler faults are extremely critical for t-MxM. They can cause multiple errors distributed in a row or block of the output matrix that, if injected in CNNs, can cause misdetection.

The contributions in this paper are:

- A thorough analysis, based on over 1.5 million RTL injections, of the effects of transient faults in the GPU scheduler, pipeline registers, control units, functional units, and special units on 12 instructions;
- A detailed characterization of RTL faults distribution at the output of tiled MxM (used in CNNs);
- The description of a fault model (available on a public repository [23]) to be adopted instead of single bit-flip;
- The adaptation of a software framework (NVBitFI) to inject the RTL fault model in real GPUs;
- A method allowing the reliability evaluation of real-world applications, from HPC and safety-critical domains.
- The identification of the GPU resources whose corruption is more likely to generate errors.

The reminder of the paper is structured as follows: after summarizing background and related work, highlighting the contributions and limitations of our strategy (Section II), we give an overview of the proposed idea (Section III). In Section IV, we detail the two-levels fault injection frameworks (RTL and software) and how we combine them. Section V presents the results of the RTL-level fault injection and describes the fault model we propose. Section VI shows the effects of the fault model in real world codes and compares our data with naive fault-injection. Finally, Section VII concludes the paper and paves a path for future work.

II. Background and Related Work

A. Radiation Effects in Computing Devices

There are several sources of transient faults that can reduce the reliability of a computing device, including environmental perturbations, software errors, process/temperature/voltage variations, and radiation-induced events. The latter are particularly critical, as they dominate error rates in commercial devices [24]. A transient fault leads to one of the following outcomes: (1) no effect on the program output (i.e., the fault is masked), (2) a Silent Data Corruption (SDC) (i.e., an incorrect program output), or (3) a Detected Unrecoverable Error (DUE) (i.e., a program crash or device hang/reboot).

Recent results, based on field data from HPC servers, have highlighted that parallel architectures, particularly GPUs, have a high fault rate because of the high amount of available resources [25]–[27]. Additionally, recent works have identified some peculiar reliability weaknesses of GPUs architecture, suspecting that the corruption of the GPU hardware scheduler or shared memories can severely impact the computation of several parallel threads [9], [10], [22], [25], [26]. As a result, multiple GPU output elements can potentially be corrupted, effectively undermining the reliability of several applications, including CNNs [28], [29]. Unfortunately, as it is not possible to inject faults in software directly on the scheduler, all previous findings are based only on experimental observations and speculations that still need to be confirmed. One of the goals of our paper is to understand if and how faults in characteristic resources of GPUs affect HPC and safety-critical applications correctness.

B. Reliability Evaluation Methodologies

The effects of faults in computing devices can be evaluated at different levels of abstractions, from gate level to architectural level and system level, as illustrated in Figure 1. Each evaluation methodology has some benefits and limitations, which we summarize next. We also discuss why the complexity of GPUs exacerbates the limitations associated with the available methodologies. In general, methodologies that act closer to the fault physical source (i.e., the silicon implementation) are more realistic (and costly in terms of processing time) while methodologies closer to the output manifestation of the fault are more efficient (but less realistic in terms of the fault effect in real applications).

Beam experiments induce faults directly in the transistors by the interaction of accelerated particles with the Silicon lattice, providing highly realistic error rates [24]. Beam experiments are not included in Figure 1 because, as errors are observed only when they appear at the output, generally they do not allow to track faults propagation. This prevents one to associate observed behaviors with the fault source and, thus, to identify the most vulnerable device resources.

Software fault injection is performed at the highest level of abstraction and, on GPUs, it was proved efficient in identifying those code portions that, once corrupted, are more likely to affect computation [3]–[5], [7], [8], [30]. However, the analysis is limited as faults can be injected only on that subset of resources which is visible to the programmer. Unfortunately, critical resources for highly parallel devices (i.e., hardware scheduler, threads control units, etc.) are not accessible to the programmer and, thus, cannot be characterized via high level fault injection. Additionally, the adopted fault model (typically single/double bit-flip) might be accurate for the main memory structures (register files, caches) but risks to be unrealistic

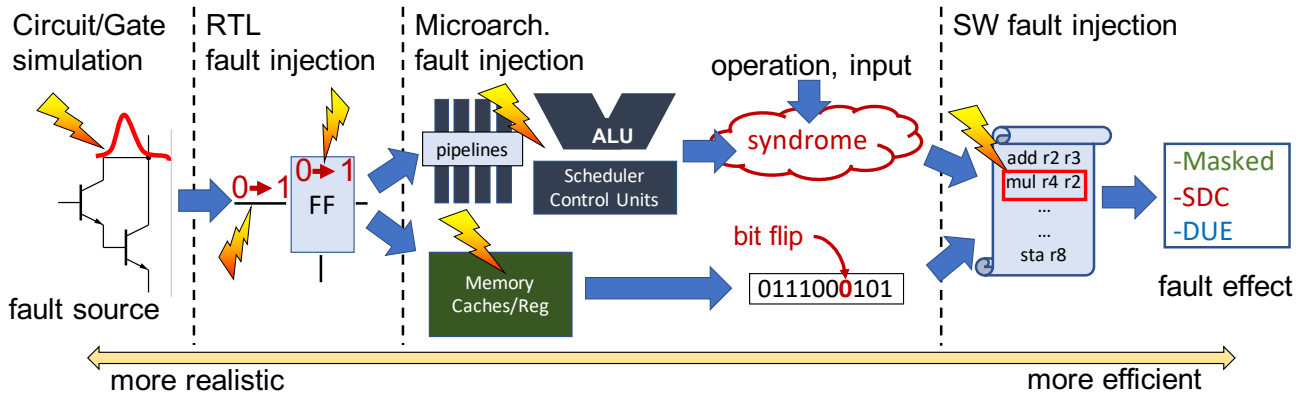

Fig. 1. Abstract view of fault injection and propagation. Reliability evaluations closer to the fault source (i.e., the silicon implementation) are more realistic by extremely costly. Evaluations closer to the fault manifestation at the output are more efficient but risk to be unrealistic. Single or double bit-flips injections at software level, for instance, accurately simulate memory faults, only. Faults in other resources have a syndrome that depends on operation and input value.

when considering faults in the computing cores or control logic, as also shown in [18]. In fact, as shown in Figure 1, while a fault in the memory array directly translates into a corrupted value, the single transient fault in a resource used for the execution of an operation (pipelines, ALU, scheduler, etc..) can have not-obvious effects on the operation output. We call this not-obvious effect *syndrome*. The syndrome induced in the instruction output by faults in the computing core depends on the operation, on its input, and on the corrupted resource. The only possible way to find this syndrome, as we do in our paper, is to perform lower level fault injection.

Micro-architecture fault injection provides a higher fault coverage than software fault injection as faults can, in principle, be injected in most modules. A preliminary work, based on Multi2Sim, presented micro-architectural fault injection data on GPUs, but the analysis is limited to just memories [6]. One of the issues of micro-architectural fault injection in GPUs is that the description of some modules (including the scheduler and pipelines) is behavioural and their implementation is not necessarily similar to the realistic one. A recent work has demonstrated that micro-architectural fault injection provides a sufficiently accurate reliability evaluation on ARM embedded CPUs [31]. On GPUs such a demonstration is still missing, and is likely to be more challenging due to the complexity of the hardware underneath the micro-architecture.

Register-Transfer Level (RTL) fault injection accesses all resources (flip flops and signals) and provides a more realistic fault model, given the proximity of the RTL description with the actual implementation of the final hardware [15], [18], [19]. However, the time required to inject a statistically significant number of faults makes RTL injections impractical. The huge amount of modules and units in a GPU and the complexity of modern HPC and safety-critical applications exacerbate the time needed to have an exhaustive RTL fault injection (hundreds of hours for small codes), making it unfeasible. Previous work that evaluates GPUs reliability through RTL fault injection is limited to naive benchmarks [19].

Circuit or Gate Level Simulations induce analog current spikes or digital faults in the lowest abstraction level that still allows to track fault propagation (not available with beam tests). There are two main issues with the level of details required to perform this analysis on GPUs: (1) a circuit or gate level description of GPUs is not publicly available and, even if it was, (2) the time required to evaluate the whole circuit would definitely be excessive (the characterization of a small circuit takes weeks [14]).

Hybrid or combined fault injections at different levels of abstraction have been adopted to increase the reliability evaluation efficiency without jeopardizing its accuracy. Some works have proposed to use a detailed RTL fault injection in specific portions of the circuit and a fast fault simulation in others [15], [16]. Recent works combined an extremely detailed gate level fault injection in tandem with a faster (but still impracticable for complex devices) RTL evaluation [12], [14]. Cho et al. used high level simulation (not using real hardware) triggering a RTL model when the fault needs to be injected [13]. Subasi et. al focuses on RTL injection to provide a more detailed fault model, but limited to embedded processors ALU [18]. While our paper takes inspiration from the two level fault injection concept, none of these works address GPUs (nor parallel devices in general), but mainly embedded processors, with a completely different complexity scenario. CPUs, in fact, have just one or few pipelines and faults are unlikely to affect multiple threads while, on GPUs, several pipelines and computing units need to be characterized and it is fundamental to consider also multiple threads corruption. Previous solutions, then, do not scale to GPUs complexity. Additionally none of previous works provide, as we do, a fault model database that could be used in future evaluations.

C. Contributions and Limitations

In this paper we propose to combine, for the first time for GPUs, the fine grain evaluation of *RTL fault injection* with the flexibility and efficiency of *software fault injection in real GPUs*. As characterizing realistic codes with RTL fault injection is unfeasible, we limit the RTL analysis to

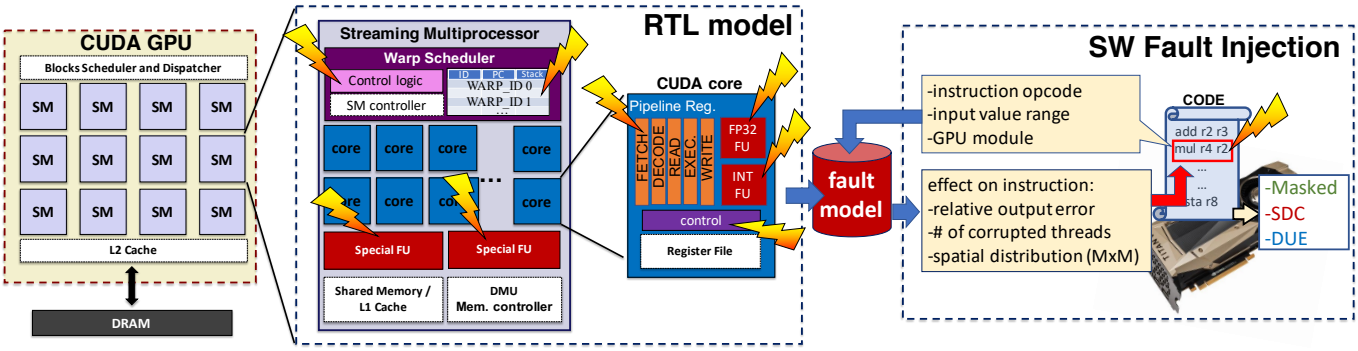

Fig. 2. Scheme of the proposed two-level fault injection framework. Using the RTL model we characterize the effects that faults in GPU modules (we do not inject in the modules depicted as white boxes) have on the SASS instructions output. Based on the instruction opcode, its input, and the module of interest we pick the fault model (syndrome) to inject in software on a real GPU that executes a code.

common GPU *SASS instructions* (or simply *instructions*), gathering the syndrome induced by faults in the instruction output value, i.e., we produce an accurate fault model for the most common machine operations. As all GPU modules are accessible in the RTL model, we can provide deeper insights on GPU faults source and characterize also the faults effects on multiple threads. Then, using an updated software framework (NVBitFI), we inject the syndrome that comes from our RTL analysis (rather than a simplistic fault model as all previous works on GPU software fault injection do). The high speed of software fault injection allows us to observe the effect of fault syndromes in the execution of real-world applications, while the few previous works on GPU RTL fault injection are limited to naive workloads.

While the proposed strategy can effectively allow a more detailed and accurate GPU reliability analysis, we acknowledge some intrinsic limitations. (1) RTL is not the lowest possible abstraction layer (see Figure 1). We choose RTL fault injection because the circuit or gate models are not available for GPUs and their characterization would, in any case, take too long. As shown in previous work, though, RTL evaluation accuracy is very close to gate level simulation [14]. (2) Our evaluation shares with any other research work based on open source models the limitation of being based on mature architectures. FlexGripPlus, which is the only RTL open-source GPU model currently available, is based on NVIDIA G80 architecture. While we cannot guarantee that the observed fault syndrome is representative of cutting-edge GPUs, the G80 is still CUDA compliant and is based on the same Instruction Set Architecture (ISA) of modern NVIDIA GPUs such as Kepler, Volta, and Turing (with the exception of tensor core and few other instructions based on updated modules). As the hidden structures of a GPU, such as the scheduler and the pipeline registers, are also supposed to be present even in modern architectures, given the CUDA compatibility we made the decision to use the RTL description we have available, even if from a different generation. Also, while probably the FlexGripPlus intrinsic limitation might impact the precision of our evaluation, it does not undermine the impact of the proposed strategy, that is directly adaptable to other GPU RTL models, as they become available. (3) The syndrome imposed by a fault could depend on the operation input. Testing all inputs combination is obviously impossible. We decided to limit the characterization to three input ranges.

III. Overview of the Idea

The proposed reliability evaluation framework for GPUs is divided in two main steps: RTL fault simulation and software fault injection, as depicted in Figure 2.

Using a GPU RTL model (details in Section IV-A), we inject faults in the GPU main computing modules. We consider Pipeline Registers, Warp Scheduler, FP32 and INT functional units, Special Function Units (SFUs), and control signals. We *do not* inject errors in memories (caches and register file) as we assume that GPUs employed in applications with strict reliability requirements feature ECC. Moreover, as a fault in a memory cell(s) affects a software visible state directly (it translates into a corrupted value with no further operations), its syndrome is already well known (single/double bit-flip) and depends just on the memory technology [24]. On the contrary, a fault inside a computing resource during an operation's execution has a not-obvious impact on the output (*syndrome*) [18], which we intend to characterize.

Rather than executing an application in the RTL model, we characterize the effect of faults in a subset of GPU ISA *SASS instructions*. A SASS instructions is the simplest, atomic, two-inputs machine operation that form the NVIDIA GPU compiled code, and is directly translated into hardware signals inside the device. We choose to characterize the instructions that, based on GPU code profiling, are more common in applications taken from universally adopted benchmark suites for HPC and safety-critical applications (Rodinia [32], NVIDIA SDK [33], CNNs [20], [21]). The chosen instructions, described in details in [34], are: Floating point operations (FADD, FMUL, FFMA - Fused Mul and Add), Integer operations (IADD, IMUL, IMAD - Mul and Add), Trascendental functions (SIN, EXP), Load/Store (GLD, GST), Branch (BRA), and Integer set predicate/register (ISET). While these instructions represent only a small part of all the ≈200 different opcodes in a GPU ISA [34], they account for more

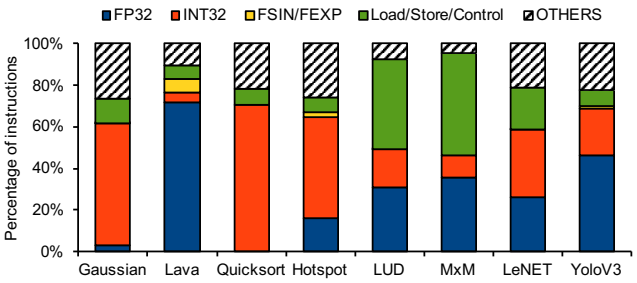

Fig. 3. Applications instruction profile. We characterize, with RTL injections, FP32 (FADD, FMUL, FFMA), INT32 (IADD, IMUL, IMAD), Special Funtions (FSIN, FEXP), Control (GLD, GST, Branch, ISET) SASS instructions. "Others" are the instructions we do not characterize.

than 70% of the executed instructions in common codes for GPUs, as shown in Figure 3. Our framework allows future updates, to add additional instructions that are of interest. We also characterize a *mini-app (tiled MxM)* to highlight possible scheduler corruption effects that could be hidden in the SASS instructions characterization (details in Section V-A).

A perfect RTL fault injection would require one to test each instruction with the exact input values it receives when executed in the code being characterized, which is clearly unfeasible. We decide to limit the analysis to three input ranges (Small, Medium, Large, as detailed in Section IV). Previous work has shown that software fault injection results for GPUs do not depend on the input value (with unbiased values) [35]. Part of our contribution is to understand if this result still holds for RTL fault injection and how much the fault effect on the instruction output depends on the input value.

With the RTL fault injection we have measured both the probability for the fault to reach a software visible state (i.e., the Architectural Vulnerability Factor, AVF [36]) and the fault impact on the instruction output value. We built a database of possible fault syndromes based on: the instruction opcode, the input range, and the injection site (the corrupted module). To quantify the syndrome we built a statistical distribution of the *relative difference* (i.e., absolute difference between the expected and the faulty instruction output, divided by the expected value). In other words, we track how much, in percentage, the fault has modified the instruction(s) output. The syndromes, the number of corrupted threads, and the spatial distribution of wrong elements (for tiled-MxM), populate a database used for the software fault injection [23].

To inject the RTL fault syndromes in software we update the already developed NVBitFI framework [3] (details in Section IV-B). NVBitFI profiles the compiled code to be evaluated, listing all the executed SASS instructions. The fault (or *error* as it has reached a visible state, according to Avizienis definition [37]) is injected at the output of a randomly selected instruction while the code is being executed on the real GPU. The updated version of NVBitFI extracts, from the RTL fault injection database, the most suitable fault syndrome to apply, considering the opcode and input range. Once the instruction output is corrupted, the code execution continues and the effect on the output is characterized as SDC, DUE, or Masked. With NVBitFI, then, we measure the probability for the faults that reached a software visible state to propagate further, till the application output (i.e., the Program Vulnerability Factor, PVF [38]).

TABLE I
EVALUATED MODULES, SIZES AND INSTRUCTIONS USED PER MODULE

Module	RTL Size (Flip-Flops)	Type	Instructions
FP32	4,451	Execution/Data	FADD, FMUL, FFMA
INT	1,542	Execution/Data	IADD, IMUL, IMAD
SFU	3,231	Execution/Data	FSIN, FEXP
SFU controller	190	Control	FSIN, FEXP
Scheduler controller	3,358	Control	ALL
Pipeline Registers	10,949	Control/Data	ALL

The benefit of our strategy relies on the fact that the detailed and time consuming RTL evaluation on the SASS instructions is done only ones, to populate the syndromes database. The software fault injection maintains its efficiency (thus allowing the evaluation of complex applications) but provides both extra accuracy, by using the RTL syndromes, and impact, as we can correlate the observed SDCs with their hardware source.

IV. Evaluation Methodologies

In this section we detail the two-levels fault injection frameworks and how they are combined.

A. RTL Fault Injection Framework

We use FlexGripPlus [19] GPU model to perform the RTL fault injection. FlexGripPlus is an open-source VHDL-based GPU model, which implements the Nvidia G80 architecture [39], with details on the most representative modules, and compatible with the commercial CUDA programming environment. This model can use three different configurations (8, 16, or 32) per Streaming Multiprocessor, selected before simulation or synthesis.

A custom RT-level framework [40] performs the fault injection through a general controller that manages the *ModelSim* environment, which hosts FlexGripPlus. The controller injects one fault (as a single transient) in the targeted GPU module, according to a faults list. For the analysis presented in this paper, we inject errors in the warps scheduler, the pipeline registers, the Integer Functional Units (INT FUs), the Single Precision Floating Point FUs (FP32 FUs), the Special FUs (SFUs) used for transcendental functions, and the control logic (see Figure 2), but we expressly do not consider faults in the main memory structures (caches, register file, shared memory). Table I lists the characterized modules, their size, and the instructions that use each module. Overall, our characterization covers ≈84% of the resources (flip flops) involved in the computation of the characterized instructions, excluding memories (≈23% if considering ECC-protected memories).

Once the fault is propagated to any of the available outputs (instruction output register, memories, or control signals), its effect is classified, by comparing the output values and signals with the golden ones obtained in a fault-free simulation, as SDC (output values mismatch), DUE (hang), or Masked (no effect).

We host the RTL model and fault injection framework on a server built with 12 Intel Xeon CPUs running at 2.5 GHz and 256 GB of RAM. The duration of the fault campaign depends directly on the program's length and the number of locations to inject, which is proportional to the target module's size. For instance, one micro-benchmark requires 8 and 5 hours, in our server, to perform the fault injections on the scheduler controller and the Floating-Point Unit, respectively.

We generate a **general report** per fault campaign, which includes the effect (SDC, DUE, Masked) of each injected fault based on (1) the characterized instruction, (2) the input value range (we test three input ranges per instruction, details in Section V-C), and (3) the target module (where the fault is injected). We also classify the fault effect as *individual* (one single thread affected) or *multiple* (more threads affected). The general report allows to measure the AVF for each module and instruction as the ration between observed errors (SDCs/DUEs) and injected faults.

To characterize the syndrome at the output, we generate, for each observed SDC, a **detailed report** that contains the location of the injected fault, the golden value, the faulty value, the number of affected bits, the number of affected threads, the eventual spatial distribution of erroneous values in the warp output, and the memory address.

B. Software Fault Injection Framework

To inject, in software, the fault syndrome obtained with the RTL evaluation, we have updated the already available NVBitFI framework [3]. NVBitFI is the most suitable fault injector for this work since it allows to instruct the kernels at SASS level (the machine code executed in the GPU hardware). Other fault injectors such as GPUQin, CAROL-FI, Kayotee, GPGPU-SIM, SASSIFI [4], [7], [41], [42] do not inject at SASS level (but SASSIFI), do not offer support for Volta and newer architectures, or do not inject in CUDA libraries.

NVBitFI can inject transient errors in the GPU's ISA visible states, modifying the SASS instructions output of a code being exectuted on a real GPU. NVBitFI allows the user to select the fault model to inject (single, double bit-flip). We modify the injection procedure to inject the syndrome obtained with the RTL evaluation (see Figure 2). When NVBitFI picks an instruction to be corrupted, our framework identifies its opcode. From the RTL fault database, we select the most suitable syndrome to apply based on the source of the fault, the opcode, and the input range (according to a statistical distribution, as discussed in Section V-C). The syndrome, as mentioned, is described as a relative error. The updated NVBitFI, then, modifies the instruction output value of a relative amount (e.g., if the syndrome is 100%, NVBitFI multiplies by two the instruction output value).

Additionally, we have included a dedicated procedure to corrupt the output of tiled-MxM inside CNNs. The fault injector picks a random tile during the execution of a random CNN layer and modifies its output elements according to the syndrome (relative error and spatial distribution) defined with the RTL fault injection (details in Section V-D).

V. RTL Reliability Evaluation

In this Section we detail the results of the RTL characterization of faults effect in the considered GPU modules. We first describe the tested micro-benchmarks and mini-app, then we present the AVF and the fault syndrome description.

A. Micro-benchmarks and mini-app description

We design several micro-benchmarks to characterize the effects of RTL faults in integer, floating point, special functions, memory movements, and control-flow instructions. To observe the possible propagation of the single injected fault to individual or groups of threads, each micro-benchmark instantiates 64 threads (2 warps) executing the same instruction. We also test a mini-app (i.e., tile-based matrix multiplication) to better observer scheduler faults impact. It is worth noting that we inject only one fault in one targeted injection site per micro-benchmark execution. Possible multiple threads corruptions are caused by that single fault propagation and not by multiple fault injections.

With the goal of characterizing the syndrome at the output of **arithmetic instructions** we have designed 8 specific CUDA micro-benchmarks to run on FlexGripPlus, one for each targeted floating point (FMUL, FADD, and FFMA), integer (IADD, IMUL, and IMAD), and special (FSIN, FEXP) instructions. Each of the 64 threads executes the same instruction without interactions between threads. We test the floating point and integer opcodes with three different pre-defined input ranges: *Small* (S, both inputs in the range $6.8\text{x}10^{-6}$ to $7.3\text{x}10^{-6}$), *Medium* (M, in the range 1.8 to 59.4), and *Large* (L, in the range $3.8\text{x}10^{9}$ to $12.5\text{x}10^{9}$). In the software fault injection any instruction with an input smaller than S (bigger than L) receives the S (L) syndrome, values in between receive the M syndrome. The range selection is heuristic, based on observed common values for SASS instructions inputs on the considered HPC and CNNs applications. We also test a combinations of input ranges (first input S, second input L), obtaining very similar results than S and M ranges (not shown in the paper). For the special functions (FSIN, FEXP), we select three inputs according to the operational constraints in the SFU (in the range 0 to $\pi/2$), avoiding range reduction procedures. To avoid the bias of our results we perform a fault injection campaign on 4 different randomly selected values for each input range.

We also consider **memory movements** (GLD and GST) and **control-flow instructions** (BRA, ISET). The load and store micro-benchmark performs a load operation followed by a store operation. For the control-flow operation, we allocate a limited number of set-register instructions before the branch operation. A fault is detected when a set register is not correctly assigned or when the branch condition fails. We anticipate that, not surprisingly, in most cases faults in control-flow instructions collapse the execution leading to a DUE.

Scheduler corruptions (and multiple threads corruptions in general) may have specific effects on the execution of codes in which threads interact with each other that may not be detected with the micro-benchmarks we have designed. As a specific

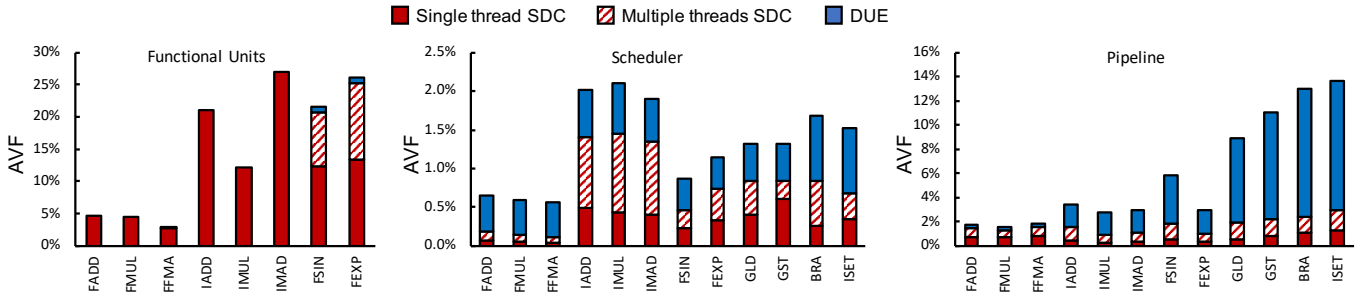

Fig. 4. AVF of the injections at RTL level on the functional units (FP32, INT, SFU), the scheduler, and pipeline registers for the different instructions. We plot the average AVF measured with the S, M, L input ranges.

and extremely important case study, we also characterize with RTL fault injection a **tile-based matrix multiplication (t-MxM)** mini-app. The choice of the mini-app is dictated by the observation that more than 70% of operations inside a CNN is MxM related [21]. To avoid memory latencies and, thus, improve matrix multiplication efficiency, large matrix multiplications are split into *tiles* (smaller MxM). The tile size is set to maximize performances without saturating caches and registers. In our framework the optimal tile size is of 8x8. Each tile is assigned to a Streaming Multiprocessor and, then, all tiles are combined to form the output of MxM. In a CNN, the MxM output forms the layer output (feature map).

To select the input for t-MxM, we execute LeNET and YOLOv3 with the MNIST [20] and VOC2012 [21] datasets, and observed that most tiles involved in convolution process have similar values, while the tiles at the edge of the feature map have a higher amount of zero operands, because of padding [20], [21]. We then characterize three inputs for the tiles with the RTL fault injection: (Max) Max tile (the tile with the highest sum of elements values), (Z) Zero tile (the tile with the highest number of zeros), and (R) Random tile (a tile selected among the ones without significantly biased values). We test 4 different values per tile type (Max, Z, R).

B. Architectural Vulnerability Factor

We perform fault injection in 6 GPU modules characterizing, for most modules, 12 SASS instructions and t-MxM, with different input sets (3 ranges and 4 values per range for arithmetic operations and t-MxM). In total, we perform 144 RTL fault-injection campaigns and, for each campaign, we inject more than 12,000 faults. That is, we present data from more than $1{,}72 \times 10^6$ fault injections. This guarantees a statistical margin error lower than 3%.

Figure 4 depicts, for injections in Functional Units (FP32, INT, SFU), Warp Scheduler, and Pipeline Registers, the AVF of each instruction. We have not considered injections in functional units for GLD, GST, BRA, and ISET as the FUs are idle when executing those instructions. In Figure 4 we distinguish between SDCs affecting a single or multiple threads. As we have observed, in accordance with [35], that the AVF does not significantly depend on the input range (the AVF difference between S, M, and L inputs is always lower than 5%), in Figure 4 we show the average AVF measured with the three input ranges.

Figure 4 shows that faults in the scheduler are less likely to impact the computation than faults in the functional units or pipeline (the y axes are on different scales). We recall that in our micro-benchmarks threads do not interact with each others, reducing the scheduling strain. In Section V-D we show that the scheduler AVF increases significantly in more complex codes. Moreover, the functional units corruptions are much more likely to generate SDCs than DUEs while the outcome of injections in the pipeline is mainly dominated by DUEs. We further investigate the observed behaviours next.

More than 60% of the SDCs caused by scheduler corruptions affect more than one thread for the INT and FP32 micro-benchmarks while injections in the functional units cause multiple threads corruption only for FSIN and FEXP. This is because the GPU has a dedicated ADD, MUL, and MAD unit for each thread while the few (two) available special function units (SFUs) need to be shared among different threads (see Figure 2). A deeper analysis of the multiple SDCs source revealed that the multiple corrupted threads observed with functional units corruptions in FSIN and FEXP are actually caused by faults in the control units of the SFUs. Interestingly, also pipeline injections cause multiple threads corruption. Investigating the causes for those multiple threads we found that, while most of pipeline registers ($\approx$84%) store operands for each parallel core, there is also a small portion of registers ($\approx$16%) devoted to control signals. The corruption of these latter registers caused the observed multiple threads SDCs.

On the average, the number of corrupted parallel threads per warp is 1 for INT and FP32 functional units, 8 for the SFUs, 28 for the scheduler, and 18 for the pipeline. These averages show that the parallel operation's modules in the GPU, such as the scheduler and the pipeline, are more prone to corrupt a high number of multiple threads in a warp than others. A fault in the control structures and signals of the pipeline and, mainly, of the scheduler (which manages the warp operation), affects multiple threads. The lower number of threads corrupted in the pipeline is related to the number of available FUs and active threads at a given time (8 in our case). As some signals are not updated until a new warp is dispatched, their corruption affects, on the average, two of the four groups of 8 threads in

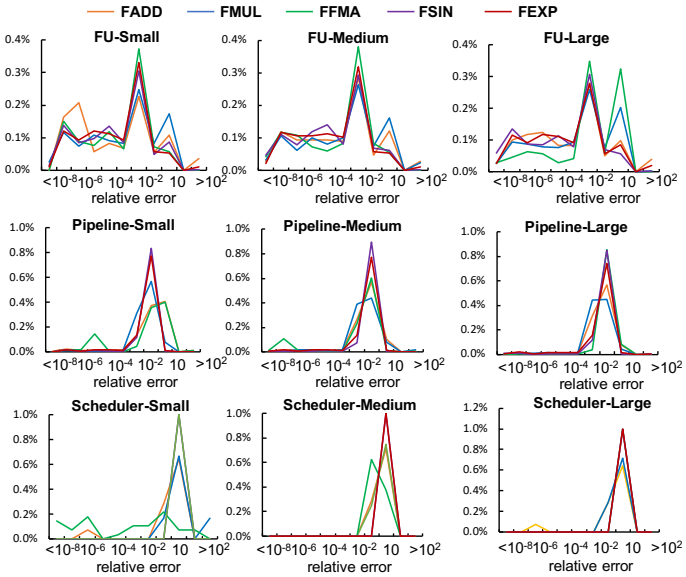

Fig. 5. Distribution of the fault syndrome (relative error) from the RTL fault injection in the Functional Units (top), Pipeline (middle), and Scheduler (bottom) for the floating point instructions executed with S, M, L inputs.

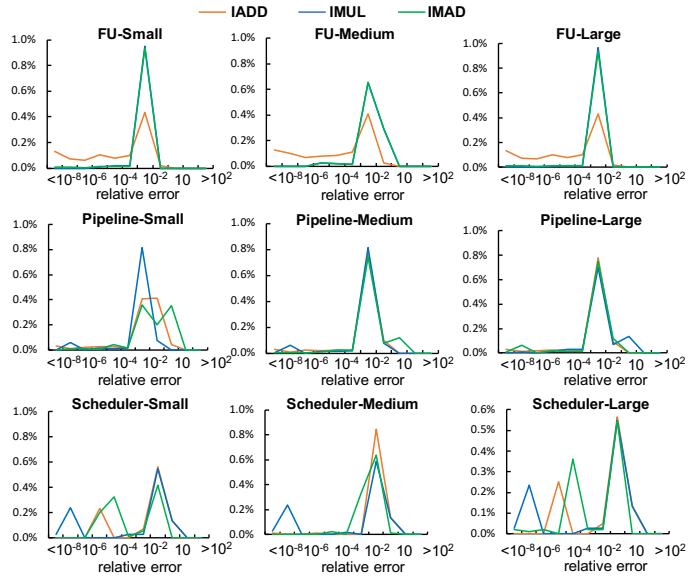

Fig. 6. Distribution of the fault syndrome (relative error) from the RTL fault injection in the Functional Units (top), Pipeline (middle), and Scheduler (bottom) for the integer instructions executed with S, M, L inputs.

a warp (32 threads).

The high DUE AVF for the pipeline (0.3% for floating point, 2% for integer, 4% for special units, 10% for control operations) is caused by corruptions of pipeline control registers that, despite being few (≈16%), are confirmed to be highly critical. The DUE AVF is exacerbated for special function units, because of the additional control signals required to share the use of the few available SFUs, and for control flow operations (GLD, GST, BRA, ISET) for which faults in the data path are likely to corrupt the control flow.

The scheduler DUE AVF is almost constant (between 0.5% and 0.6%) for all instructions but BRA and ISET, for which the DUE AVF is about 0.8%. Tracking the fault propagation we found that scheduler DUEs are caused by faults affecting structures in the controller devoted to store the state of the warp or memory addresses. In contrast, the scheduler SDCs are mostly caused by faults affecting warp state bits, so disabling active or enabling inactive threads.

As observed in Figure 4, the functional units AVF for the floating point instructions (FADD, FMUL, FMAD) is much smaller than for the integer instructions (IADD, IMUL, IMAD). This is caused by the higher complexity and area of the floating point units, that are more than 3x larger than the integer units (see Table I). A larger area increases the number of injection sites, thus reducing the probability to hit a critical resource for computation.

It is worth noting that the AVF assumes that a fault has occurred and does not include any information about the probability for the fault to be generated. In order to consider also the fault generation probability, the modules AVF should be weighted with the module relative size (shown in Table I). A more accurate evaluation would consider the fault rate of the different modules, for which either proprietary information on the technology sensitivity or beam experiments would be necessary. As a first evaluation, combining Table I and Figure 4, we can say that functional units, having a huge size and high AVF, are likely to be the source of most SDCs, while pipelines are likely to be the cause of most DUEs.

C. Fault Syndrome

For each SDC observed at the instructions output we keep a *detailed report* (described in Section IV-A), that includes also the corrupted output value. We characterize the fault syndrome by measuring the *relative error* induced by the fault. We also perform an additional analysis of the binary distribution of errors at the output (how many bits of the output are corrupted and in which position). Nevertheless, there was not a clear pattern for the bits corruption (in the large majority of cases, ≈24 bits are wrong, randomly distributed). This is in accordance with the fact that computational faults induce a not-obvious syndrome, as also observed in [18].

The relative error syndrome analysis highlights a very interesting trend. Figures 5 and 6 show the distribution of relative errors for the tested micro-benchmarks. We plot, in the y-axes, the percentage of observed SDCs that modify the instruction output value from less than 10^{-8} to over 10^{2}. That is, 0.2% of SDCs observed on FADD executed with the Small input range modify the output value by $10^{-6}\times$ and 0.3% of SDCs observed on FFMA executed with the Large input range modify the output value by $10\times$.

As depicted in Figures 5 for floating point and in Figure 6 for integer instructions, the relative difference between the observed corrupted values and the expected value does not follow a Gaussian distribution. For all instructions there is a clear peak that depends on the input range and on the injection

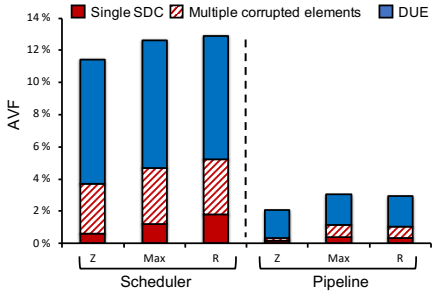

Fig. 7. AVF of the scheduler (left) and pipeline (right) for DUEs, single and multiple thread SDCs for the Max, Zero, and Random t-MxM.

TABLE II
DISTRIBUTION OF THE MULTIPLE PATTERNS (SINGLE CORRUPTED ELEMENTS ARE NOT LISTED) OBSERVED WITH T-MXM.

inj. site	row	col.	row+col.	block	rand.	all
scheduler	0.96%	0.07%	0.45%	5.77%	0.69%	54.6%
pipeline	45.4%	1.36%	1.04%	7.29%	0.42%	4.17%

site. In some configurations (FFMA and FMUL FU injections with L input) two peaks are present. It is also interesting to notice that the relative difference distribution is narrow, if compared to the floating point or integer representation range. Only in few cases (less than 0.05%), we observed a syndrome with a relative error higher than 10^2 (i.e., the corrupted output value is 100x bigger/smaller than the expected one). This observation attests that even injecting a random number of bit-flips in the instruction output might not be realistic. Our results show that there is a limited difference between corrupted and correct values. Interestingly, the median of the syndrome values between S/M/L varies by just $\sim$1% in all cases but MUL and FMA, for which the median changes by up to 30% (bigger input range has higher median). Only for MUL and FMA, then, we expect a syndrome dependence on the input.

Once the opcode, the input range, and the injection site have been determined, to injecting the syndrome in software we need to select a relative error, statistically taken from the data presented in Figures 5 and 6 and available in [23]. The syndrome (as relative error) does not follow a Gaussian distribution (all distributions have a p-value smaller than 0.05 on the Shapiro-Wilk test), but rather follows a *power law* distribution [43], in which few events are predominant. We create a Pseudo-Random Number Generator (PRNG) function, based on the mathematical formalization in [43], that generates the syndrome to be injected as follows:

$$relative\_error = P^{-1}(1-r) = x_{min}(1-r)^{-1/(\alpha-1)} \quad (1)$$

Where $0 \leq r < 1$ is a uniformly distributed random value, α is the scaling factor, and x_{min} is the values lower bound. Both parameters are extracted from our data based on the methods described in [43].

D. Tiled MxM errors distribution

To better study the impact of multiple threads corruptions in codes with threads interactions we have characterized, with the RTL fault injector, the mini-app t-MxM. We test three input types (Max, Zero, and Random tiles), averaging the results obtained with four values per each input type. For t-MxM, we inject faults in the scheduler and pipeline registers but not in the functional units. Faults in this latter module, as shown in Figure 4, would not cause multiple threads corruptions in t-MxM (there is no transcendental operation). The effects of the single thread SDC would then be the same as the ones observed injecting the FU syndrome in software, without requiring a costly RTL injection.

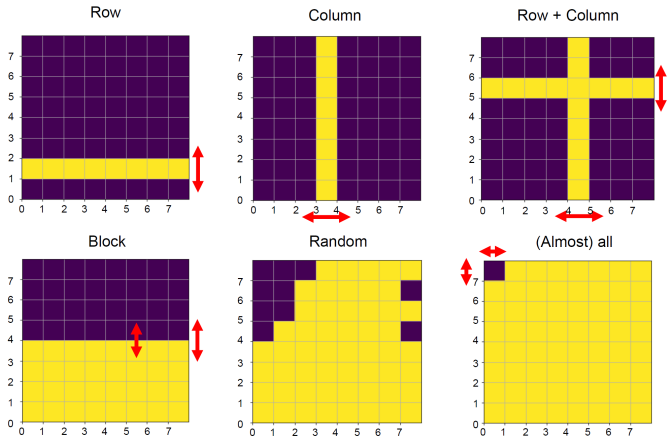

Fig. 8. Spatial distribution of the multiple corrupted elements patterns observed in the RTL injection on the mini-app t-MxM. Arrows indicate that neither the position of the observed pattern nor the block size are fixed.

Figure 7 shows the average AVF for DUEs, single and multiple corrupted elements in the t-MxM output for injections in the scheduler and pipeline. We recall that we inject one fault per execution: the multiple corrupted elements are caused by the single fault propagation.

A major difference from the micro-benchmarks AVF in Figure 4 is that, for t-MxM, the scheduler AVF is higher than the pipeline one. As mentioned, while the micro-benchmarks are very simple and do not implement threads interactions, t-MxM also includes several instructions for computing memory addresses and threads indices. The higher strain on the scheduler and the higher portion of time spent in scheduling operation increases the AVF (for both SDCs and DUEs). On the contrary, the pipeline AVF is higher in the micro-benchmarks because, when a fault appears at the first instruction output, it is marked as SDCs, without further chances to be masked (there is no other computation). In t-MxM an instruction's wrong output can be masked in a downstream operation (for instance, with a multiplication by zero). This statement is supported by our pipeline data in Figure 7, that shows a much lower SDC AVF for the Z tile.

An additional interesting result from Figure 7 is that the portion of SDCs that affect multiple elements is very high (at least 70% of scheduler induced and 50% of pipeline induced SDCs). We can further study the multiple errors at the t-MxM output by visualizing the geometrical distribution of the corrupted elements. In Figure 8 we plot the 6 different

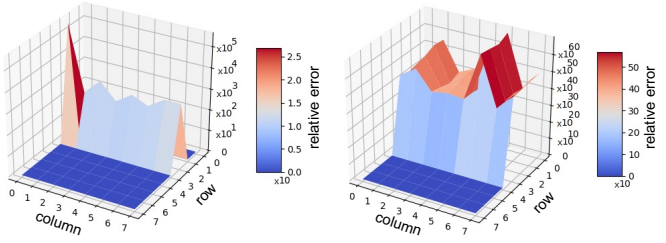

Fig. 9. Variance distribution of relative errors for an example of Row (left) and Block (Right) patterns with multiple corrupted elements.

spatial multiple corrupted elements distribution patterns that we have observed injecting faults in the scheduler and pipeline registers. We have observed corrupted elements distributed in a row, a column, a row and a column, a block of elements (varying in size), randomly, and all (or almost all) elements corrupted. Table II lists the percentage of occurrences of the different patterns (single SDCs are not listed). As the distribution of the observed patterns is very similar in the three inputs we test (M, Z, R tiles), we list the average distributions.

Interestingly, pipeline injection mostly produces corrupted rows, while scheduler injection is more likely to affect the whole output matrix. Having a whole column corrupted is very unlikely, for both injection sites. This is because t-MxM calculation is row-major and, as mentioned, the distribution of these error patterns depends not only on the way the GPU hardware reacts to the faults, but also on the software propagation. While this multiple elements distribution is not generic, the choice of t-MxM extended evaluation is dictated by its importance in CNNs execution. As shown in [28], [29], the observed multiple errors patterns (but not single element corruptions) can indeed induce misdetections in CNNs.

We have also characterized the syndrome of t-MxM. Most of the syndromes are concentrated in few values, again following a power-law distribution. In Figure 9 we plot the relative error distribution for two examples: a row and a block error. As shown, the relative error distribution varies among the corrupted elements. To inject the t-MxM syndrome in software, then, we use Equation 1 to select the range of the relative errors for all the elements to corrupt. In this range, we again select a power law distribution for the corruption of the individual output elements.

VI. Real-World Applications Evaluation

In this section we present the results obtained injecting, in software, the fault syndromes discussed in the previous section. We inject at least 6,000 syndromes per application, for a total of more than 48,000 injections that took 350 GPU hours, ensuring 95% confidence intervals to be lower than 5%.

As discussed in Section IV-B, the specially crafted NVBiTFI version selects the most suitable fault syndrome to apply depending on the opcode, the input, and the module that we assume to be the cause of the fault. For this paper, we inject a cocktail of fault syndromes following the power-law

TABLE III
Sizes, domain, and PVF for all tested applications.

	Size	Domain	PVF	
			Single bit-flip	Relative error
Gaussian	512x512	Linear algebra	1.0	1.0
Lava	2 3D boxes	Particle simulation	0.69	0.91
Quicksort	4MB array	Sorting	0.94	0.95
Hotspot	1024x1024	Physics simulation	0.25	0.37
LUD	2048x2048	Linear algebra	0.82	0.99
MxM	256x256	Linear algebra	0.95	0.99
LeNET	MNIST	Classification	0.03	0.04
YoloV3	VOC2012	Object detection	0.17	0.27

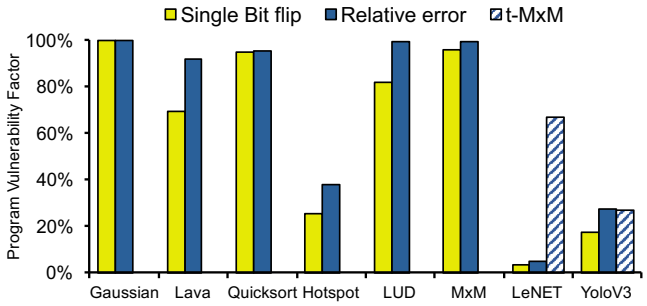

Fig. 10. SDC Program Vulnerability Factor for HPC codes.

statistical distribution described in Section VI and depicted in Figures 5 and 6. It is obviously possible to focus the software fault injection in just one module or even to tune the syndrome injection with the probabilities for the different modules and/or instructions to be corrupted. While being potentially very interesting, such an evaluation requires the area of the various modules (as reported in Table I) but also the raw probability for transient faults to be generated. Unfortunately, this information is not publicly available and could only be measured through beam experiments.

We select to characterize a set of applications, listed in Table III, that are representative of different HPC computational classes: Floating Matrix Multiplication (MxM), Lower Upper Decomposition (LUD), Quicksort, particles simulation (Lava), Gaussian elimination (Gaussian), fluid dynamics (Hotspot). We also consider CNNs for classification and object detection (LeNET and YoloV3). Each code is likely to stimulate specific GPU modules, according to the distribution of opcodes depicted in Figure 3. Hence, results obtained with the selected benchmarks could be representative also for similar applications.

While NVBitFI could inject in multiple threads and the RTL fault syndrome includes the information about the multiple SDCs, to propose a better comparison with the traditional single bit-flip evaluation, we decide to inject only single thread SDC using our fault syndrome. We want to highlight how accurate a random single bit-flip injection is compared to the RTL fault syndrome. For CNNs, we also include an RTL fault-injection on the execution of t-MxM to evaluate better the effects of scheduler faults and multiple threads corruptions in the detection and classification of objects.

Figure 10 shows and Table III lists the SDC Program

Vulnerability Factor (PVF) for the HPC codes. PVF is the probability of the faults injected in the software visible states to generate an SDC at the end of execution. In other words, when injecting in software, we assume that the fault injected in RTL has corrupted the instruction output. For the data in Figure 10 and Table III we inject only in the 12 opcodes we characterize with RTL fault injection (that represent more than 70% of all executed opcodes, as shown in Figure 3).

RTL faults that generate DUEs (shown in Figure 4) are not considered in software fault injection, as they simply hang the application. We never observed DUEs caused by the injections of the syndromes obtained from the RTL injections. This is mainly because GPU applications are highly data intensive, and avoid data driven condition statements. Thus, it is hard for a data error to affect the control flow in GPUs.

To compare our analysis with the traditional fault injection, we consider two error models for our analysis, single bit-flip (randomly injected in the 32 bits values) and the fault syndrome (injected using the power law distribution) from RTL injection. For all the codes presented in Figure 10, the fault syndrome model generates a higher or equal PVF compared with the traditional single bit-flip error model. Interestingly, we observe that the single bit-flip injection would underestimate the applications reliability of up to 30% for Lava and 48% for Hotspot, respectively. For other codes (Gaussian and Quicksort) the two fault models provide very similar results, as the PVF of the considered instructions is, by itself, extremely high (close to 1).

For CNNs, if we consider an SDC, as we do in Figure 10 and Table III, a mismatch in the application output (we will consider misdetection/classification next), the single bit-flip injection underestimates the PVF of 33% for LeNET and 50% for YoloV3. The higher reliability to transient fault of CNNs compared to HPC codes should not surprise, as it has already been observed and studied on GPUs [28], [29].

For LeNET and YoloV3, we also have measured the PVF when we inject the corrupted t-MxM, as presented earlier in this section. On LeNET, the SDC PVF when t-MxM fault model is injected is much higher than the other two fault models (12x higher), while for YoloV3 it is similar to the relative error PVF. This different behavior is because LeNET has a very small number of network parameters per layer (12,000, on average), and, thus, the corruption of a tile consists in the corruption of a considerable number of parameters. On the contrary, YoloV3 layers are very big (100,000 parameters, on average), and even fully corrupted 8x8 tile represent a small percentage of the matrix.

We can further analyze the impact of faults in CNNs by distinguishing between tolerable SDCs and critical SDCs, i.e. those that corrupt the output sufficiently to cause a network misclassification/misdetection. We found that t-MxM injection produce an unacceptable amount of critical errors. For LeNET, the number of errors that completely change the classification is 20% and, for YoloV3, it is 15%. It is worth noting that, in LeNET, none of the injected single bit-flips nor RTL single thread syndrome produce misclassifications nor misdetections. A realistic and accurate fault model, that also considers faults in GPU critical resources as the scheduler, is then necessary not to risk to underestimate the effect of transient faults in CNNs.

By investigating further the RTL fault propagation, we found that the control structures (inside the scheduler, the pipeline, and the SFU) are the primary sources of errors that corrupt multiple threads, affecting a warp or even generating the geometrical patterns of errors shown in Figure 8. As we have seen with the software fault injection, despite the limited size of these structures in a GPU core and the relative low AVF, these critical modules might produce severe consequences for an application, especially CNNs. An efficient, and effective, hardening solution for GPU should definitely target these modules.

Finally, we highlight that injecting at RTL level *one single fault* in just one of the applications listed in Figure 10 would take more than 10 hours, using our 12 CPUs server. As we inject a total of 48,000 faults, it would take 4.8×10^5 hours to produce all data in Figure 10. That is more than 54 *years*. Despite the limitations listed in Section II-C and the introduction of some simplifications on the input range, our two-level framework allows an analysis that would otherwise be impossible.

VII. Conclusions

In this paper we have applied the concept of multi-level fault injection to GPUs. Thanks to the combination of RTL and software fault injection, we reduce by several orders of magnitude the time required to have a detailed and accurate analysis of faults propagation from the hardware source to the application output. The RTL accuracy of our framework identifies the most critical GPU resources, for both SDCs and DUEs, and identifies a set of possible fault effects (syndromes) in the instructions output. The efficiency of our version of NVBitFI allows to propagate these effects in real-world applications.

The faults syndrome database we present in this paper is made publicity available with the intent of providing a more accurate fault model than the naive single bit-flip, to evaluate the reliability of applications and to validate hardening solutions. Moreover, the flexibility of our framework grants the possibility of future updates, both in terms of updated RTL model or extended instructions evaluation.

In the future we intend to include a beam experiment FIT rate evaluation of instructions, to provide also an estimation of the faults occurrence rate together with the fault propagation.

VIII. Acknowledgments

This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Sklodowska-Curie grant agreements No 886202, No 722325 (RESCUE ETN) and from The Coordenação de Aperfeiçoamento de Pessoal de Nível Superior, Brazil (Finance Code 001). We also thank the funding of CNPq, Research Productivity Scholarship grant ref. 306475/2019-7.

REFERENCES

[1] P. Rech, L. Carro, N. Wang, T. Tsai, S. K. S. Hari, and S. W. Keckler, "Measuring the Radiation Reliability of SRAM Structures in GPUS Designed for HPC," in *IEEE 10th Workshop on Silicon Errors in Logic - System Effects (SELSE)*, 2014.

[2] NVIDIA, "NVIDIA Announces World's First Functionally Safe AI Self-Driving Platform." "https://nvidianews.nvidia.com/news/nvidia-announces-worlds-first-functionally-safe-ai-self-driving-platform", 2018.

[3] NVLABS, "Nvbitfi: An architecture-level fault injection tool for GPU application resilience evaluations." https://github.com/NVlabs/nvbitfi, 2020.

[4] S. K. S. Hari, T. Tsai, M. Stephenson, S. W. Keckler, and J. Emer, "SASSIFI: An architecture-level fault injection tool for GPU application resilience evaluation," in *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 249–258, 2017.

[5] J. Wei, A. Thomas, G. Li, and K. Pattabiraman, "Quantifying the Accuracy of High-Level Fault Injection Techniques for Hardware Faults," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pp. 375–382, 2014.

[6] A. Vallero, D. Gizopoulos, and S. Di Carlo, "SIFI: AMD southern islands GPU microarchitectural level fault injector," in *2017 IEEE 23rd International Symposium on On-Line Testing and Robust System Design (IOLTS)*, pp. 138–144, 2017.

[7] B. Fang, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi, "GPU-Qin: A methodology for evaluating the error resilience of GPGPU applications," in *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*, pp. 221–230, March 2014.

[8] B. Nie, L. Yang, A. Jog, and E. Smirni, "Fault site pruning for practical reliability analysis of GPGPU applications," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 749–761, 2018.

[9] P. Rech, L. L. Pilla, P. O. A. Navaux, and L. Carro, "Impact of GPUs parallelism management on safety-critical and HPC applications reliability," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pp. 455–466, 2014.

[10] D. A. G. Goncalves de Oliveira, L. L. Pilla, T. Santini, and P. Rech, "Evaluation and mitigation of radiation-induced soft errors in graphics processing units," *IEEE Transactions on Computers*, vol. 65, no. 3, pp. 791–804, 2016.

[11] R. Balasubramanian *et al.*, "Understanding the impact of gate-level physical reliability effects on whole program execution," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, 2014.

[12] S. Nimara, A. Amaricai, O. Boncalo, and M. Popa, "Multi-level simulated fault injection for data dependent reliability analysis of rtl circuit descriptions," *Advances in Electrical and Computer Engineering*, vol. 16, pp. 93–98, 02 2016.

[13] H. Cho, C. Cher, T. Shepherd, and S. Mitra, "Understanding soft errors in uncore components," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2015.

[14] M. A. Kochte, C. G. Zoellin, R. Baranowski, M. E. Imhof, H. Wunderlich, N. Hatami, S. D. Carlo, and P. Prinetto, "Efficient simulation of structural faults for the reliability evaluation at system-level," in *2010 19th IEEE Asian Test Symposium*, pp. 3–8, 2010.

[15] A. Ejlali, S. G. Miremadi, H. Zarandi, G. Asadi, and S. B. Sarmadi, "A hybrid fault injection approach based on simulation and emulation co-operation," in *2003 International Conference on Dependable Systems and Networks, 2003. Proceedings.*, pp. 479–488, 2003.

[16] A. L. Sartor, P. H. Becker, and A. C. Beck, "A fast and accurate hybrid fault injection platform for transient and permanent faults," *Des. Autom. Embedded Syst.*, vol. 23, p. 3–19, June 2019.

[17] E. Schneider and H. Wunderlich, "Multi-level timing and fault simulation on GPUs," *Integr.*, vol. 64, pp. 78–91, 2019.

[18] O. Subasi, C.-K. Chang, M. Erez, and S. Krishnamoorthy, "Characterizing the impact of soft errors affecting floating-point alus using rtl-ievel fault injection," in *Proceedings of the 47th International Conference on Parallel Processing*, ICPP 2018, (New York, NY, USA), Association for Computing Machinery, 2018.

[19] J. E. R. Condia *et al.*, "FlexGripPlus: An improved GPGPU model to support reliability analysis," *Microelectronics Reliability*, vol. 109, 2020.

[20] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[21] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement," *arXiv*, 2018.

[22] D. A. G. D. Oliveira, L. L. Pilla, M. Hanzich, V. Fratin, F. Fernandes, C. Lunardi, J. M. Cela, P. O. A. Navaux, L. Carro, and P. Rech, "Radiation-induced error criticality in modern hpc parallel accelerators," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 577–588, 2017.

[23] F. F. dos Santos, J. E. R. Condia, L. Carro, M. S. Reorda, and P. Rech, "DSN 2021 data repository." https://github.com/UFRGS-CAROL/dsn_2021_data, June 2021.

[24] R. Baumann, "Soft errors in advanced computer systems," *IEEE Design Test of Computers*, vol. 22, pp. 258–266, May 2005.

[25] N. DeBardeleben, S. Blanchard, L. Monroe, P. Romero, D. Grunau, C. Idler, and C. Wright, "GPU behavior on a large HPC cluster," in *Euro-Par 2013: Parallel Processing Workshops* (D. an Mey, M. Alexander, P. Bientinesi, M. Cannataro, C. Clauss, A. Costan, G. Kecskemeti, C. Morin, L. Ricci, J. Sahuquillo, M. Schulz, V. Scarano, S. L. Scott, and J. Weidendorfer, eds.), (Berlin, Heidelberg), pp. 680–689, Springer Berlin Heidelberg, 2014.

[26] D. Tiwari, S. Gupta, J. Rogers, D. Maxwell, P. Rech, S. Vazhkudai, D. Oliveira, D. Londo, N. DeBardeleben, P. Navaux, L. Carro, and A. Bland, "Understanding GPU errors on large-scale HPC systems and the implications for system design and operation," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pp. 331–342, 2015.

[27] L. B. Gomez, F. Cappello, L. Carro, N. DeBardeleben, B. Fang, S. Gurumurthi, K. Pattabiraman, P. Rech, and M. Sonza Reorda, "GPGPUs: How to combine high computational power with high reliability," in *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 1–9, 2014.

[28] F. F. d. Santos, P. F. Pimenta, C. Lunardi, L. Draghetti, L. Carro, D. Kaeli, and P. Rech, "Analyzing and increasing the reliability of convolutional neural networks on GPUs," *IEEE Transactions on Reliability*, vol. 68, no. 2, pp. 663–677, 2019.

[29] Y. Ibrahim, H. Wang, M. Bai, Z. Liu, J. Wang, Z. Yang, and Z. Chen, "Soft error resilience of deep residual networks for object recognition," *IEEE Access*, vol. 8, pp. 19490–19503, 2020.

[30] S. Tselonis and D. Gizopoulos, "GUFI: A framework for GPUs reliability assessment," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 90–100, 2016.

[31] A. Chatzidimitriou, P. Bodmann, G. Papadimitriou, D. Gizopoulos, and P. Rech, "Demystifying soft error assessment strategies on arm cpus: Microarchitectural fault injection vs. neutron beam experiments," in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 26–38, 2019.

[32] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 44–54, 2009.

[33] "Cuda code samples," Oct 2018.

[34] NVIDIA, "Cuda binary utilities."

[35] F. G. Previlon, C. Kalra, D. R. Kaeli, and P. Rech, "Evaluating the impact of execution parameters on program vulnerability in GPU applications," in *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 809–814, 2018.

[36] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin, "A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor," in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, (Washington, DC, USA), pp. 29–, IEEE Computer Society, 2003.

[37] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Trans. Dependable Secur. Comput.*, vol. 1, p. 11–33, Jan. 2004.

[38] V. Sridharan and D. R. Kaeli, "Eliminating microarchitectural dependency from architectural vulnerability," in *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, pp. 117–128, 2009.

[39] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia Tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, pp. 39–55, March 2008.

[40] B. Du, J. E. R. Condia, M. Sonza. Reorda., and L. Sterpone, "On the evaluation of SEU effects in GPGPUs," in *2019 IEEE Latin American Test Symposium (LATS)*, pp. 1–6, 2019.

[41] D. Oliveira, L. Pilla, N. DeBardeleben, S. Blanchard, H. Quinn, I. Koren, P. Navaux, and P. Rech, "Experimental and analytical study of xeon phi reliability," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, (New York, NY, USA), pp. 28:1–28:12, ACM, 2017.

[42] S. Jha, T. Tsai, S. Hari, M. Sullivan, Z. Kalbarczyk, S. W. Keckler, and R. K. Iyer, "Kayotee: A fault injection-based system to assess the safety and reliability of autonomous vehicles to faults and errors," 2019.

[43] A. Clauset, C. R. Shalizi, and M. E. J. Newman, "Power-law distributions in empirical data," *SIAM Review*, vol. 51, no. 4, pp. 661–703, 2009.