HAL open science

Understanding the Effects of Permanent Faults in GPU's Parallelism Management and Control Units

Juan-David Guerrero-Balaguera, Josie E. Rodriguez Condia, Fernando F. dos Santos, Matteo Sonza, Paolo Rech

To cite this version:

Juan-David Guerrero-Balaguera, Josie E. Rodriguez Condia, Fernando F. dos Santos, Matteo Sonza, Paolo Rech. Understanding the Effects of Permanent Faults in GPU's Parallelism Management and Control Units. SC 2023 - ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis, ACM; IEEE, Nov 2023, Denver, United States. pp.1-12, 10.1145/3581784.3607086 . hal-04132590

HAL Id: hal-04132590

https://inria.hal.science/hal-04132590v1

Submitted on 19 Jun 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Understanding the Effects of Permanent Faults in GPU's Parallelism Management and Control Units

Juan-David Guerrero-Balaguera
Politecnico di Torino – Department of Control and Computer Engineering
Torino, Italy

Josie E. Rodriguez Condia
Politecnico di Torino – Department of Control and Computer Engineering
Torino, Italy

Fernando F. dos Santos
Univ Rennes INRIA – Taran Group
Rennes, France

Matteo Sonza Reorda
Politecnico di Torino – Department of Control and Computer Engineering
Torino, Italy

Paolo Rech
University of Trento – Department of Industrial Engineering
Trento, Italy

ABSTRACT

Graphics Processing Units (GPUs) are over-stressed to accelerate High-Performance Computing applications and are used to accelerate Deep Neural Networks in several domains where they have a life expectancy of many years. These conditions expose the GPUs hardware to (premature) aging, causing permanent faults to arise after the usual end-of-manufacturing test. Techniques to assess the impact of permanent faults in GPUs are then strongly required, thus allowing to estimate the reliability risk and to possibly mitigate it.

In this paper, we present a method to evaluate the effects of permanent faults affecting the GPU scheduler and control units, which are the most peculiar and stressed resources, along with the first figures that allow quantifying these effects. We characterize over 5.83×10^5 permanent fault effects in the scheduler and controllers of a gate-level GPU model. Then, we map the observed error categories in software by instrumenting the code of 13 applications and two convolutional neural networks, injecting more than 1.65×10^5 permanent errors. Our two-level fault injection strategy reduces the evaluation time from hundreds of years of gate-level evaluation to hundreds of hours. We found that faults in the GPU parallelism management units can modify the opcode, the addresses, and the status of thread(s) and warp(s). The large majority (up to 99%) of these hardware permanent errors impacts the running software execution. Errors affecting the instruction operation or resource management hang the code, while 45% of errors in the parallelism management or control-flow induce silent data corruptions.

KEYWORDS

Error models, GPUs, Permanent faults, Reliability, fault injection

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SC 2023, Denver, CO,
© 2023 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00

ACM Reference Format:
Juan-David Guerrero-Balaguera, Josie E. Rodriguez Condia, Fernando F. dos Santos, Matteo Sonza Reorda, and Paolo Rech. 2023. Understanding the Effects of Permanent Faults in GPU's Parallelism Management and Control Units. In *Proceedings of International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 2023).* ACM, New York, NY, USA, 12 pages.

1 INTRODUCTION

Graphics Processing Units (GPUs) are increasingly adopted in several fields, including High-Performance Computing (HPC), autonomous robots, automotive, and aerospace applications. The use of GPUs in applications that wander off their traditional fields (gaming, multimedia, and consumer market) has suddenly pushed the interest and posed questions, about their reliability [4].

Currently, active GPU research targets the evaluation of reliability and the identification of feasible improvements. Most studies highlight a high sensitivity of GPUs to transient faults [12, 14, 17, 24, 27, 32, 44, 47, 50, 51], caused by the high number of available resources and the advanced semiconductor technology they adopt. Additionally, the parallelism management and control units of GPUs have been shown to be particularly critical since their corruption affects multiple threads [24, 38]. The parallelism of GPUs, which provides unquestionable benefit in terms of performance, is, then, one of the most vulnerable characteristics of the device. GPU manufacturers have provided effective reliability countermeasures by improving the memory cells design [39], adding error-correcting codes [16], hardware structures for fault testing [25], and proposing software checksum [21] or multi-threading redundancy [49].

Most of the available research on GPU's reliability targets *transient* faults, leaving *permanent* faults largely unexplored. This was justified since, in most applications the GPU's life expectancy does not exceed two years. However, GPUs employed in automotive, aerospace, and military applications are expected to be operative for many years. Additionally, typical operative conditions of HPC grade GPUs, such as over-stress, high temperature, high frequency of operation, and technology node shrinking, are shown to accelerate aging [23] and even to expose the device to terrestrial radiation-induced permanent faults [20]. The extended utilization and premature aging suddenly raise questions about how GPUs behave in the presence of permanent faults. Crucially, only a few

preliminary works target permanent faults in GPUs [18, 26, 46] and none focus on the parallelism management units.

In this paper, we aim to significantly advance the understanding of GPUs reliability by proposing a method to target a totally unexplored aspect: the effect of permanent faults in the GPU circuitry in charge of parallelism management. We decided to focus on the scheduler, the fetch, and decoder units since (a) they are the most peculiar GPU resources (no equivalent modules exist in CPUs), (b) a permanent fault affecting them will have non-trivial effects on the code execution, (c) they cannot be easily protected with error correcting codes or hardware redundancy, (d) they are likely to age faster than other units since they are always active during the execution of any software code.

The challenge our methodology addresses is to provide an accurate, yet efficient, permanent faults impact evaluation. Injecting permanent faults directly in software would be fast [46], but is not a viable option for faults affecting the scheduler and control units, since a suitable fault model is missing. In fact, to accurately simulate the effect of permanent faults in the software, it is first necessary to track how *each* machine instruction that uses the malfunctioning resource behaves. A detailed gate-level analysis is not a viable option either because of the extremely long simulation time. To characterize a simple CNN as LeNet on a gate-level GPU simulator would take more than 10,000 days!

Inspired by other works in the field [6, 8, 33, 40, 41], we adopt a hybrid approach that combines the accuracy of gate-level simulations with the flexibility of software-level fault injections. The behavior of a permanent fault is evaluated, at the gate level, using an open-source model of an NVIDIA GPU (FlexGripPlus [9]). The impact of the permanent faults on the execution of each machine instruction is evaluated and classified, identifying error models to propagated in a real GPU device at the software level. Then, to propagate these errors in software, we have crafted a dedicated software-based error injector framework (NVBitPERfi) able to automatically handle the error instrumentation in the kernel, to apply the corruption observed in the low-level evaluation, and to properly corrupt multiple threads and/or warps. With this approach, we characterize and discuss the reliability with respect to permanent faults of 15 real workloads, including two CNNs (LeNet and YOLOv3). The proposed frameworks, the gate-level analyses, the software-level reports, and the new NVBitPERfi tool are available in a public repository [1].

The main contributions of this work are:

- A method to identify the effects of permanent faults in terms of errors at the instruction level;
- The formalization of 13 categories of instruction errors *(error models)* based on the effects of the permanent faults in the GPU's warp scheduler controller, fetch, and decoder unit;
- A new fault injector *"NVBitPERfi"* built on top of NVBit, to map the 13 error models in software, instrument the code, and evaluate the permanent error effects in applications;
- An accurate understanding of why and how permanent faults in the GPU parallelism management units affect the execution of 15 real workloads.

The remainder of the paper is structured as follows: Section 2 provides the background and related work. Section 3 describes the proposed multi-level fault injection evaluation. Section 4 reports the gate-level fault injection results and identifies the error categories. Section 5 presents the software-level implementation of the error categories and evaluates their propagation effects in real applications. Section 6 concludes the paper.

2 BACKGROUND AND RELATED WORK

In this section, we provide background and related works about GPUs reliability. Then, we discuss permanent faults and the challenges in their evaluation. Finally, we highlight the contributions and limitations of our study.

2.1 GPUs architecture and reliability

Modern GPUs are designed as arrays of parallel cores organized in processing clusters or *Streaming Multiprocessors* (SMs). SMs include 2 to 4 *Parallel Processing Block* (PPBs). One scheduler inside each SM statically distributes the tasks (thread-groups or *Warps*) among the PPB cores. To manage parallelism and submit, distribute, and track warps into the available cores, each PPB includes a Warp Scheduler Controller (WSC), a fetch unit, and an instructions decoder unit [31, 34, 45]. These are the units we characterize in this paper.

Since GPUs have a large area with high availability of computing resources, they are particularly susceptible to experience hardware faults. A fault in a GPU hardware can have one of the following effects on the executed software. (1) **Masked**: no effect on the program output. The corrupted data is not used, or the circuit functionality is not affected. (2) **Silent Data Corruption (SDC)**: undetected output corruption, that is, the application finishes, but the output is not correct. (3) **Detected Unrecoverable Error (DUE)**: program or system crash. The GPU reliability to *transient* faults has already been evaluated through microarchitectural and low-level fault simulation [3, 9, 42], software-based fault injection [14, 32, 47, 50, 51], and beam experiments [12, 17, 24, 27, 44]. Since GPUs execute several processes in parallel, it has been shown that a transient corruption in the WSC or a single error in shared resources affects various output elements [17, 21, 27, 38, 44].

2.2 Permanent faults on GPUs

Permanent fault represent a new sever risk for GPUs reliability. In fact, while GPUs used in multimedia are changed every few years, GPUs employed in various domains such as the automotive, military, and aerospace ones, have an extended life expectancy. Additionally, the *"International Roadmap for Devices and Systems - 2022"* (IRDS) and independent studies [19, 43] state that modern digital devices implemented with cutting-edge technologies (beyond 7nm) are highly susceptible to *Electromigration* and *Time-Dependent-Dielectric-Breakdown*, both major sources of accelerated aging and permanent faults [23, 28]. Crucially, IRDS emphasizes that the device's lifetime decreases by half at each new manufacturing process generation[23]. This, in combination with over-stress operative conditions (typical for GPUs), high-temperature, and radiation-induced latch-ups, exacerbate the probability of having permanent faults [2, 11, 20, 35, 37].

A permanent *fault* is a physical defect in a circuit, unit, or device. When the applied stimuli activate the permanent fault, it can propagate to a visible software state, corrupting data or operation output,

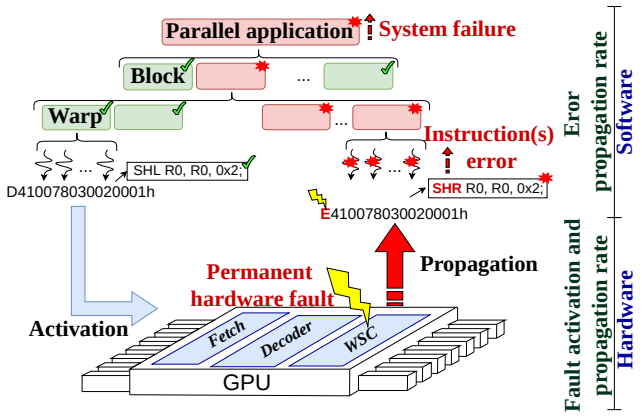

Figure 1: Propagation of permanent faults in GPUs. A hardware permanent *fault* in the scheduler, fetch, or decoder unit produces nontrivial *error* models at the instruction level and affects multiple threads/warps, causing a system *failure*.

thus becoming an *error*. If the error propagates and affects the outputs produced by the system, it becomes a *failure* causing a crash, hang, or silent data corruption. As shown in Figure 1, we track permanent faults propagation from the hardware to the software output using two metrics: the *Fault Activation and Propagation Rate* (FAPR), that measures the probability of a permanent hardware fault to be activated by stimuli (e.g., program's instructions) and then propagate reaching a software visible state (thus becoming an error). Then, the *Error Propagation Rate* (EPR) measures the probability for errors (caused by a permanent fault) to propagate till the output, becoming a failure.

Several works performed extensive analyses of possible sources of permanent faults in processor-based systems [5, 22]. Other studies wisely focused on identifying error models at higher levels to simplify the analysis [15]. In [36], the authors emphasize the importance of fine-grain low-level and cross-layer resilience evaluations, highlighting the weakness of purely software error propagation. Inspired by these insights, we combine low-level and software-level fault injections, to achieve high-reliability evaluation accuracy.

Only a few preliminary studies evaluate the incidence of permanent fault effects in GPUs. In [13], the authors investigate the effect of permanent faults in GPUs by increasing the temperature and accelerating the aging process. Other works evaluate GPU memory permanent faults affecting the weights of a CNN [29]. Some permanent fault injectors have also been proposed. In [18], the authors proposed a customized software-based error injector to evaluate the effect of permanent faults on the register file and functional units. Additionally, NVBitFI [46] allows the user to inject permanent faults. Unfortunately, in all the available permanent fault injectors, the proposed error models consider only limited hardware units (mostly memory resources and functional units).

None of the studies on GPUs have focused their evaluations on the parallelism management units, as we do in this paper. Previous works tools can hardly be extended to evaluate more complex and critical units, such as the parallelism management units.

2.3 Contributions and potential limitations

This is the first paper proposing a methodology and showing results about the propagation of permanent faults in the GPU parallel management and control units (i.e., WSC, decoder, and fetch units). The criticality of these units and the possible impact of their failures in the threads/warps execution require a dedicated and accurate study. The complexity of the evaluation we propose is totally different from previous works that focus on GPU memory or functional units failures [18, 46]. In fact, besides the effect on the instruction execution, depending on the fetched/decoded/scheduled instruction, a permanent corruption in the GPU parallel management units can modify (a) the opcode or operands, (b) the control-flow of the code, (c) the threads/warps status (enable/disable), (d) the assignation or enabling of a GPU resource. Additionally, as depicted in Figure 1, all these corruptions can affect one or multiple threads or warps.

We adopt a multi-level fault injection approach, separating the accurate low-level simulation from the software-level propagation. This methodology has been shown effective in reducing the simulation time in other works in the field [8, 10, 33, 40], but has never been applied to GPU's parallel management units. Thanks to our multi-level analysis, we can track permanent faults propagation in real-world applications.

The proposed methodology, thanks to the documentation included in the public repository [1], can be easily extended to other units in the GPU, to other devices, and to other fault models (delay, intermittent, or transient faults). Despite the generality of our methodology, we acknowledge that our evaluation has some intrinsic limitations. In fact, the low-level characterization uses one of the few available open-source GPUs (FlexGripPlus). The implemented GPU's Instruction Set Architecture (ISA) in the available model is old (G80), which is a common constraint for research works targeting commercial devices [7, 10]. Thus, the distribution of error models might be biased by the specific GPU unit implementation. However, since FlexGripPlus is CUDA compliant, the behavior of the faults is expected to be similar also in modern GPUs, and, as mentioned, our methodology can be adapted to more advanced models as they become available. Furthermore, since our method is focused on units inside SMs, novel GPU features (such as concurrent streams and multi-kernels, that act at a higher architectural level) will not undermine the accuracy of our evaluation. In fact, modern features are handled by task schedulers or additional hardware that then end up assigning the blocks to the internal units in the SM that we are evaluating.

3 PROPOSED METHODOLOGY

This section describes the main idea and the steps of the proposed method, described in Figure 2. The *low-level* evaluation exploits the accuracy of gate-level simulation to classify fault effects in terms of instruction errors. The *high-level* part employs a time-efficient software-based fault injector on real GPUs to assess the effect of permanent faults on complete applications. The method comprises five steps: **1.** hardware unit profiling, **2.** gate-level fault injection, **3.** error identification, and classification, **4.** code instrumentation and instruction-level error propagation, and **5.** application evaluation and failure classification. The next subsections detail each step.

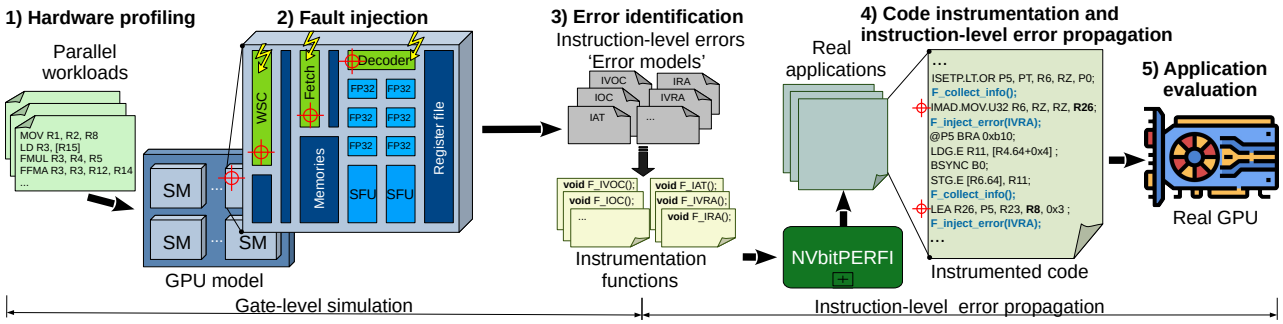

Figure 2: A general scheme of the method to characterize fault effects in parallelism management units of GPUs.

3.1 Hardware Unit Profiling

The first step collects the golden (fault-free) operational information from a targeted hardware unit in the GPU (WSC, fetch, and decoder unit). To efficiently collect accurate functional information, we use the fine-grain implementation (gate-level netlist) of the unit to test, while the rest of the GPU is simulated at the Register Transfer Level (RTL). The GPU mixed implementation (gate-level for the units of interest, RTL for the rest) allows to collect and trace per-cycle information on the tested unit at the gate level and keep the interaction with the other units at RTL. This step provides the golden copy of all the unit signals, including the input patterns.

We developed a *profiling hardware mechanism* tool to instrument the GPU model and profile the hardware unit utilization. For each instruction, our tool collects structural and operational information from the unit, including (i) the status of primary inputs and outputs, (ii) the timing information for the instruction, (iii) the instruction's type, and (iv) the time intervals of each performed operation. The units profiling resorts to the characterization of each instruction from several representative parallel workloads.

3.2 Gate-Level Fault Injection

We characterize permanent *stuck-at* faults on each possible fault site. Exhaustive gate-level fault injection campaigns are essential to determine if a fault is activated, propagated, and how it possibly manifests as an error at the output of the evaluated component. The simulation complexity would explode if we consider all possible stimuli combinations for each fault characterization. Therefore, as mentioned in the previous step, we select as stimuli the individual instructions *(patterns)* extracted from representative workloads.

The effect of a permanent fault can manifest in any point in time, depending on the stimuli. Thus, we exhaustively evaluate the activation and propagation of every individual hardware fault till the complete instruction execution for then changing the input stimuli. To track the propagation of any possible effect on the outputs of the unit, we compare the golden instruction profile with the current faulty operation. Then, we collect the observed effects and the exciting pattern (instruction information), which later serve to identify and correlate the fault model with the corrupted instruction. It is worth noting that we track the execution of the complete instruction across the GPU architecture to guarantee the identification of any possible fault propagation, so allowing the characterization of masking, hanging, and latent (inactive) fault effects.

3.3 Error Identification and Classification

In this step, we correlate the hardware profiling and the fault injection results to identify hardware fault effects at the gate level in terms of instruction-level errors (hence, modelled in software). We build a list of possible instruction-level errors caused by the injected permanent hardware faults. The error models from the low-level fault injections are classified according to the corrupted functionality and effects on the software's visible state of every instruction. Thus, the error models represent the mapping of hardware faults as instruction errors. Given the parallel architecture of GPUs, a fault might corrupt the instruction execution in one or multiple threads, and in one or multiple warps. We identify and classify errors considering both cases.

3.4 Instruction-level Error Propagation

Once the permanent hardware fault effect has been characterized as a visible software effect, we can propagate it through representative applications using fast software-based fault injection in real GPUs.

To do so, we implemented one software *error function* for each permanent error model obtained in the gate-level error identification and classification experiment (step 3). These error functions are inserted in the application's SASS code to implement the permanent error effect during the application execution, mimicking at the software level the equivalent effect of a permanent hardware fault in the GPU unit.

We crafted a customized binary instrumentation tool (*NVBitPERfi*, that will be open-source after paper publication) built over the NVBit framework [48] to propagate the instruction-level error through the application software. To implement the software-level error propagation at the ISA-level, our tool adopts the Hardware-Injection through Program Transformation (HIPT) technique [46]. NVBitPERfi mimics the behavior of a permanent error during the execution of an application considering: *i)* the error model specifications, *ii)* the device GPU architecture details, and *iii)* the instrumentation mechanisms offered by the injection framework.

We addressed two main challenges in the implementation of NVBitPERfi. (1) The target hardware units corruption can impact one or multiple threads in one or multiple warps. (2) Since we are addressing permanent faults, each instruction mapped to the corrupted hardware unit must be corrupted. Thus, we need to identify all the instructions activating the fault.

It is crucial to have detailed hardware error specifications to identify how many threads/warps need to be affected by the permanent hardware fault. An error, for instance, may disable/enable one or multiple threads by interchanging their execution with a set of threads from the same warp or different warps (e.g., $< Thread_0, Warp_0 >$ issues the $< Thread_{17}, Warp_8 >$, and this produces the skipping execution of the $< Thread_0\ Warp_0 >$). To identify the instructions mapped to the corrupted hardware, we consider GPU's architectural details, that denote the parallelism specifications, such as the maximum number of resident warps/threads per Streaming Multiprocessor (SM), and the number of sub-partitions that every SM contains. We use these architectural functionalities to define an error descriptor that links the physical defects of hardware units under analysis and the portions of the parallel application where the error will take effect. We consider the following fields: *i)* the SM identifier number, *ii)* the sub-partition identifier (PPB), *iii)* the set of warps associated with the sub-partition, *iv)* the target threads inside the selected warps, and *v)* additional parameters related to the specific error model, such as targeted operands, opcodes, error bit-masks, etc.

3.5 Applications Evaluation

Once the permanent fault effect has been characterized as a software error model and the procedures to corrupt thread(s)/warps(s) in a real GPU have been implemented, we can effectively evaluate the impact of permanent faults on real workloads. This methodology reduces the simulation times by several orders of magnitude compared to the classical logic simulation approach. For example, an entire fault injection campaign for all the error models using our methodology for the GEMM code can be performed in less than 24h, while using only low-level fault injections, the same campaign would take 60,000 hours (i.e., more than 6 *years*).

We select 15 realistic workloads (listed in Table 1) to evaluate the error models implemented on NVBitPERfi. To demonstrate how the methodology can be applied to any application, we select workloads from various domains, including Deep Learning, Linear algebra, N-body simulation, and Graphs. The selected codes are instrumented and executed inside NVBitPERfi and the permanent fault outcome is characterized as masked, SDC, or DUE..

4 LOW-LEVEL FAULT CHARACTERIZATION

In this Section, we present the results of the gate-level permanent fault injection experiments performed in GPU parallelism management units using the FlexGripPlus GPU model. We configure FlexGripPlus with one PPB per SM cluster, and 32 SP cores per PPB. The gate-level implementations of the units are obtained using a 15nms Open Cell Library [30]. Table 2 shows the percentage of area occupied by each unit, compared to one FP32 functional unit core, and their utilization percentage, taken from profiling 14 typical workloads (described below). Despite the relatively low area of the fetch and decoder units, these units are of paramount importance in the execution of instructions since they are continuously stimulated by every instruction (while the FP32 unit is stressed, on average, only by 10% to 40% of instructions), thus accelerating aging. Despite the relatively small area of the units we target, their continuous operation and their failure criticality motivate our study.

Table 1: Codes used for the software-level error injections.

	Data type	Domain	Suite
vectoradd	FP32	Linear algebra	CUDA SDK
lava	FP32	N-body	Rodinia
mxm	FP32	Linear algebra	CUDA SDK
gemm	FP32	Linear algebra	CUDA SDK
hotspot	FP32	Structured Grid	Rodinia
gaussian	FP32	Linear algebra	Rodinia
bfs	INT32	Graphs	Rodinia
lud	FP32	Linear algebra	Rodinia
accl	INT32	Graphs	NUPAR
nw	INT32	Dyn. Programming	Rodinia
cfd	FP32	Unstructured Grid	Rodinia
quicksort	INT32	Sorting	CUDA SDK
mergesort	INT32	Sorting	CUDA SDK
lenet	FP32	Deep Learning	Darknet
yolov3	FP32	Deep Learning	Darknet

Table 2: Tested units area and utilization percentage w.r.t. a FP32 functional unit.

Unit	Area (nm^2)	FP32 core (%)	Utilization (%)
WSC	11,854.4	114.3	100.0
Decoder	760.8	7.3	100.0
Fetch	708.2	6.8	100.0
FP32 unit	10,367.8	100.0	~(10.0 - 40.0)

The low-level evaluation starts with the hardware profiling of each targeted unit, to identify the signals of interest and the golden (fault-free) unit output. We use *all* the dynamic instructions (more than 25,200) from 14 representative parallel workloads from Rodinia and NVIDIA SDK benchmarks (*Sort*, *Vector_Add*, *FFT*, *Tiled Matrix Multiplication*, *Naïve Matrix Multiplication*, *Reduction*, *Gray_Filter*, *Sobel*, *Scalar Vector Multiply*, *Nn*, *Scan_3D*, *Transpose*, *Euler_3D*, and *Back Propagation*). Then, the fault evaluation resorts to 42 localized fault injection campaigns (one per benchmark for each of the three units) to evaluate the execution of every individual dynamic instruction from the workloads (i.e., the equivalent *exciting pattern* activating a unit) and identify the fault propagation effects. This procedure evaluates 426,092, 130,480, and 152,236 permanent faults (i.e., the whole stuck-at-fault list) for the exciting patterns in the WSC, fetch, and decoder units, respectively. The hardware profiling and the fault injection campaigns are performed on a server machine including 12 Intel Xeon CPUs running at 2.5 GHz and with 256 GB of RAM. It is worth noting that extensive multi-threading schemes (from 10 up to 40 parallel processes) are used to speed up the fault evaluation campaigns.

Table 3 first reports the total number of considered stuck-at faults for each unit and classifies faults in the following categories:

- *Uncontrollable* faults, i.e., those permanent faults that are never activated or propagated by any input stimuli.
- *Hardware Masked* faults, i.e., faults that are activated by the input stimuli but whose effect never reaches the unit outputs (30.0% in the WSC, 24.5% in the fetch, and 22.2% in

Table 3: Percentage of faults that are uncontrollable, masked, cause hangs or instruction-level errors.

Unit	Total	Uncontrol-lable	HW Masked	HW Hang	SW errors
WSC	29,850	35.9%	30.0%	3.6%	30.5%
Fetch	9,320	26.9%	24.5%	1.2%	47.4%
Decoder	10,874	26.0%	22.2%	2.5%	49.3%

the decoder) in any of the executed instructions. These faults are thus innocuous and can be discarded from our analysis.

- Permanent faults that cause a *hardware hang* (i.e., the GPU stops responding). Only 1.2% to 3.5% of the injected permanent faults cause a hang. These faults induce the unit execution to stop or corrupt the unit's ports (e.g., high-impedance). A detailed analysis shows that most hang source sites handle control signals in the units (e.g., state machine control signals) or synchronization signals among the units (e.g., pipeline).
- *Software errors:* faults that reach one or more unit's outputs and can thus produce an impact on the software. These faults are highly likely, being 30.5% of injections for the WSC, 47.39% for the fetch, and 49.29% for the decoder unit. These faults corrupt the output of the unit handling or selecting instruction parameters, such as the memory type or the thread(s)/warps(s) status. We specifically focus on this group in the following analysis.

To further categorize the faults in the last category, i.e., those that produce instruction-level errors, we combine the information from the hardware profiles, the fault injection campaign result, and the structural information. We have identified four main error groups (*i)* operation, *ii)* control-flow, *iii)* parallel management, and *iv)* resource management errors), which are further divided into 13 types of errors affecting the software instructions, as follows.

4.0.1 *Operation errors.*

- **Incorrect Operation Code Error (IOC):** the operational code of an instruction is modified and still valid, but the executed instruction type (or its parameters) is different from the original code.
- **Invalid Operation Code Error (IVOC):** the opcode of the instruction is modified and not valid.
- **Incorrect Register Addressed Error (IRA):** an incorrect (yet valid) register is addressed, affecting the instruction.
- **Invalid Register Addressed Error (IVRA):** an incorrect and not valid register is addressed (i.e., a register outside the limit of registers per thread).
- **Incorrect Immediate Operand Error (IIO):** the immediate operand is corrupted.

4.0.2 *Control-flow errors.*

- **Work-flow Violation Error (WV):** the workflow of an instruction is modified by corrupting the predicate conditions.

4.0.3 *Parallel management errors.*

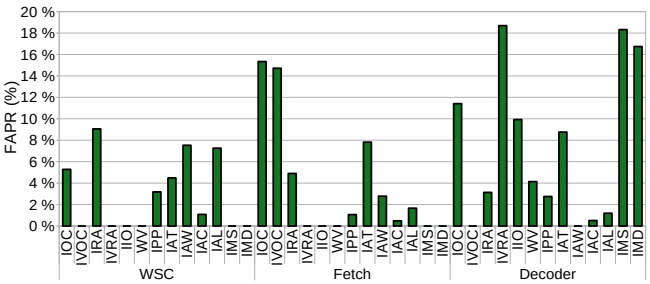

Figure 3: Fault Activation and Propagation Rate (FAPR) for permanent faults in the WSC, fetch, and decoder units. Faults are grouped by error types.

- **Incorrect Parallel Parameter Error (IPP):** incorrect addressing of resources shared among the warp, such as the shared memory and register files regions.
- **Incorrect Active Thread Error (IAT):** unauthorized enable or disable of threads in a warp.
- **Incorrect Active Warp Error (IAW):** incorrect detention, assignation, or unauthorized submission of a warp.
- **Incorrect Active CTA Error (IAC):** incorrect detention, assignation, or unauthorized submission of a CTA (cooperative thread array) in the GPU core.

4.0.4 *Resource management errors.*

- **Incorrect Active Lane Error (IAL):** unauthorized enable or disable of lanes in a GPU core.
- **Incorrect Memory Source Error (IMS):** incorrect assignation of a memory resource for operand loading.
- **Incorrect Memory Destination Error (IMD):** incorrect assignation of a memory resource for result's storing.

We differentiate errors that cause an *incorrect* operand or action from those that cause an *invalid* operation or action. While both types of errors modify the same instruction field (e.g., both IOC and IVOC modify the opcode), the former is likely to induce a data error since a (wrong) instruction is executed or a (wrong) memory value is read/written, while the latter blocks the execution. It is worth noting that most errors affect the thread management units and the parallelism in the GPU. Thus, most error types (IOC, IVOC, IRA, IVRA, IPP, IAW) affect all threads in a warp, while others (IIO, WV, IAT, IAC, IMS, and IMD) mainly corrupt one or a few threads per warp. The information about multiple threads/warps corruption is used in Section 5.1 to map the effects into instruction-level errors.

Figure 3 shows the FAPR, i.e., the probability for a hardware permanent fault to be activated and to propagate to a software visible state. We separate the FAPR of permanent faults injected in each of the three units to cause one or more of the identified error types. The most common instruction error models are IVRA, IMS, and IMD in the decoder unit, IVOC in the fetch unit, and IOC for all units. On the contrary, some instruction error classes are highly unlikely to occur (e.g., from 0.48% of IAC in WSC, to 7.53% of IAW in the fetch unit). The low percentage of IAC errors (i.e., wrong block scheduling) is explained by noting that the considered WSC, the fetch, and the decoder units handle finer-grain parallel management (operation of threads and warps), instead of coarse-grain (CTAs)

```
...
/** Error function: Part I*/
M^1 ⇐ Rd[T_x, W_x]^2
/**Target SASS instruction*/
IMAD Rd, Rsx, Rsy, Rsz
/** Error function: Part II*/
R_IR^5 ← Rd^4 ⊕ bitErrMask^3
R_IR[T_x, W_x] ⇐ Rd[T_x, W_x]
Rd[T_x, W_x] ⇐ M
...
```

a) Destination operand field case

```
...
/** Error function: Part I*/
R_IR ← R<sx,sy,sz>^6 ⊕ bitErrMask
M ⇐ R<sx,sy,sz>[T_x, W_x]
R<sx,sy,sz>[T_x, W_x] ⇐ R_IR[T_x,W_x]
/**Target SASS instruction*/
IMAD Rd, Rsx, Rsy, Rsz
/** Error function: Part II*/
R<sx,sy,sz>[T_x, W_x] ⇐ M
...
```

b) Source operand field case

Figure 4: Description of IRA/IVRA error models.

parallel management. Interestingly, faults in the decoder unit cause a wider spectrum of possible instruction effects (11 out of 13 error categories). This is due to the fact that the decoder directly interacts with the machine code of the instructions.

We also identified some single permanent faults causing more than one error type. This is unsurprising since we are considering permanent faults that can be activated differently based on the stimuli. We have seen that (a) the same permanent fault may produce different types of software errors (from 1.28% to 14.9% for the WSC, about 1.98% in the fetch, and less than 0.25% in the decoder unit, depending on the executed instruction), and (b) the same permanent fault may simultaneously produce two or more types of software errors during the operation of a single instruction (less than 18.4% of faults). Since we keep track of the instruction opcode and input stimuli that activated the permanent fault, we can correlate the error model to inject in software (Section 5.2) with the instruction being executed. This information allows to propagate the hardware fault in software and to understand the probability for a permanent fault to be activated in a realistic application.

5 SOFTWARE-BASED PERMANENT ERROR PROPAGATION

This section describes the environment we developed to analyze the propagation of errors at the software level and discusses results.

5.1 Error model implementation/propagation

We implement, as software procedures, the permanent error models derived from the fine-grain circuit-level analysis in a customized binary instrumentation tool (NVBitPERfi) to mimic in detail each error according to their specifications.

We use the same philosophy of NVBitFI: we devised dedicated instrumentation functions inserted in the assembly source code of the GPU kernels during the instrumentation stage [48]. Then, the error is injected and propagated once the *faulty kernel* is issued on the device at speed. Some error models require only one

```
...
/**Target SASS instruction*/
S2R Rd, SpecialRegisterID<x,y,z>^7
/** Error function */
Rd[T_x, W_x] ⇐ Rd[T_x, W_x]^2 ⊕ bitErrMask[T_x, W_x]
...
```

Figure 5: Description of IAT/IAW/IAC error models.

instrumentation function right before or after the targeted SASS instruction. Other error models are more challenging, since they require modifying an operand before the actual instruction execution and then restoring its content after the execution. These models are implemented with two instrumentation functions, plus a global memory storage mechanism to keep temporary data and communicate the functions during the runtime error propagation.

The software-level implementation details for each error model are described in the following, grouping the descriptions based on the technical similarities and highlighting their peculiarities.

IRA and IVRA: Incorrect/Invalid Register Addressed Error models select a wrong register address in one of the operands fields for all instructions issued by the GPU. IRA selects an address that points to a valid wrong register (i.e., within the maximum number of registers per thread). IVRA selects registers outside of these boundaries as one of the operands. To implement IRA and IVRA, we use two different approaches; one is used when the corrupted register address represents the source operand and the other when the destination address is corrupted. The error descriptor for IRA and IVRA includes the parameters introduced in the section 3.4 (instruction, thread(s), warp(s) affected) plus additional parameters: *bitErrMask*, and *errOperloc*. The *bitErrMask* is the bit level mask that modifies the target operand register number, and *errOperLoc* is the operand position inside the instruction (0 means destination operand *Rd*, and 1, 2, or 3 one of the source operands $R_{<sx,sy,sz>}$).

Fig. 4 shows the implementation of the two operation modes of IRA/IVRA. The first mode refers to the error that targets the destination operands, thus the error function stores the content of the destination register *Rd* into *M* before the instruction is executed. Then, after launching the target instruction, the second instrumentation function copies into the target error register (R_{IR}) the result of the operation stored in *Rd*, Finally, *Rd* content is restored. In the case of an error affecting the source operands, a function (issued before the instruction's execution) uses a memory location *M* to store the content of the source register $R_{<sx,sy,sz>}$ before performing any data modifications. Then, the targeted register operand $R_{<sx,sy,sz>}$ takes the content of the error-accessed register I_{IR}. A second function, executed after the execution of the target instruction, restores the original source register $R_{<sx,sy,sz>}$ content.

[1] M indicates a global memory location used for temporary data storage.
[2] The $[T_x, W_x]$ notation indicates the set of threads T_x on selected warps W_x where the error takes effect.
[3] $bitErrMask$ denotes the bits mask used to induce the index error to overwrite value.
[4] Rd corresponds to the destination register.
[5] R_{IR} represents the incorrect or invalid register to be accessed obtained from applying the $bitErrMask$ field to the original register number.
[6] $R < sx, sy, sz >$ denotes one of the source registers $R_s x$, $R_s y$, or $R_s z$.
[7] $SpecialRegisterID_{<x,y,z>}$ refers to the special register SR_TID or SR_CTAID in any of the dimensions x, y or z

IAT, IAW, and IAC: Incorrect Active Thread/Warp/CTA error models disable/enable or wrongly assign threads, warps, or CTA. To implement this behavior at the software level, we disable the execution of a set of threads on selected warp(s) by replacing their identifiers with different (wrong) ones, pointing threads to the same or different warps. For example, for disabling thread0 in warp0, the index associated with the thread changes to the index of another thread (e.g., thread8 in warp0). Thus, the register that contains indexes for all threads will not contain the index of the disabled thread, producing the error effect during the execution by skipping the execution of thread0 in warp0.

Fig 5 presents the modeling concept of IAT, IAW, and IAC. This procedure is applied to the desired number of threads on selected target warp(s) $[T_x, W_x]$ issued on a specific SM sub-partition. It implements one instrumentation function after the instructions that copy the content of a special register $SpecialRegisterID_{<x,y,z>}$ into a destination register *Rd*. In the case of IAT or IAW, the instrumentation function affects only the instructions that take the content of SR_TID for one of the x, y, or z dimensions of the parallel thread indexing of the application. The IAT (thread) error model keeps at least one thread active in the warp for its execution, whereas the IAW (warp) error model forces all indexes inside a warp to change, producing a full substitution of a particular warp for another. For IAC (CTA) error, the instrumentation function modifies the destination register *Rd* of the instructions reading the SR_CTAID special register of one of the thread's three dimensions x, y, or z indexing registers. In this case, when the index of the block changes, the obtained effect leads to incorrect block thread execution.

IAL: The software-level implementation of Incorrect Active Lane error requires two different approaches. The first one, the unauthorized inactive lane (Fig. 6.a), ignores the result of all instructions executed on a specific functional unit in one or several lanes (e.g., Integer or floating point cores). This functionality can be achieved by replacing the result of such instructions with the content of the destination register captured before executing the instructions. The second approach (Fig. 6.b) forces the execution of all predicated instructions associated with the Integer or Floating point Lane where the error is injected. An instrumentation function is inserted before the target instruction to check the predicate register status. Hence, if the predicate register *disables* an instruction's execution, then the function changes its status to *enabled*, forcing the execution of an instruction that was not supposed to be executed.

IIO, IMS, IMD, WV, and IOC: All these errors modify a field in the executed instruction(s). These errors can be implemented by modifying the destination register of a selected group of instructions either with a random value or with a different operation, using the same instruction operands (see Fig. 7). The instruction's group subject of the instrumentation and/or error injection is determined by the error type. Incorrect Immediate Operand (IIO) applies an error mask in the destination register for all instructions containing at least one reference to immediate operands. Incorrect Memory Source (IMS) inserts an error mask in all instructions containing at least one operand reference to constant or shared memory. Work-flow Violation (WV) selects and inserts an error mask in all instructions that write to a selected predicate register, affecting the application's control flow. Incorrect Memory Destination (IMD) targets all the instructions with shared memory as a

```
...
/** Error function: Part I*/
M ⇐ Rd[Lane, W_x]
/**Target SASS instruction*/
IMAD Rd, Rx, Ry, Rz
/** Error function: Part II*/
Rd[Lane, W_x] ⇐ M
...
```

a) Disable lane execution

```
...
/** Error function */
if Pr[Lane, W_x] == disabled then
    Pr[Lane, W_x] ← enable
end if
/**Target SASS instruction*/
< Pr > IMAD Rd, SrcOp_x, SrcOp_y, SrcOp_z
...
```

b) Enable lane execution

Figure 6: Description of IAL error models.

```
...
/**Target SASS instruction*/
IADD Rd, SrcOp_x, SrcOp_y
/** Error function */
Rd[T_x, W_x] ⇐ SrcOp_x[T_x,W_x] ReplOp SrcOp_y[T_x,W_x]
...
```

a) IOC

```
...
/**Target SASS instruction*/
IMAD Rd, SrcOp_x, SrcOp_y, SrcOp_z
/** Error function */
Rd[T_x, W_x] ⇐ Rd[T_x, W_x] ⊕ bitErrMask[T_x, W_x]
...
```

b) IIO/IMS

```
...
/** Error function */
R_<s,a>[T_x, W_x] ⇐ R_<s,a>[T_x, W_x] ⊕ bitErrMask[T_x, W_x]
/**Target SASS instruction*/
STS [Ra], Rs
...
```

c) IMD

```
...
/**Target SASS instruction*/
ISETP Pr, Rx, Ry, Rz
/** Error function */
Pr[T_x, W_x] ⇐ Pr[T_x, W_x] ⊕ bitErrMask[T_x, W_x]
...
```

d) WV

Figure 7: Description of IOC/IIO/IMS/IMD/WV error models.

destination reference by inserting an error bitErrMask either into the data register to be stored or in the register that addresses the shared memory. Finally, Incorrect Operation Code (IOC) targets all instructions issued by the integer or floating point cores by taking the input operands and replacing them with any other operation.

IPP and IVOC: Incorrect Parallel Parameter (IPP) error has several ways of affecting the GPU operation, but most of them lay into two main categories *i)* the wrong resource addressing hardware resources (i.e., registers or shared memory modeled by IRA, IVRA, IMS, and IMD), and *ii)* by generating an incorrect threat execution modeled by IAT or IAW. On the contrary, Invalid Operation Code (IVOC) represents an invalid opcode operation that generates an

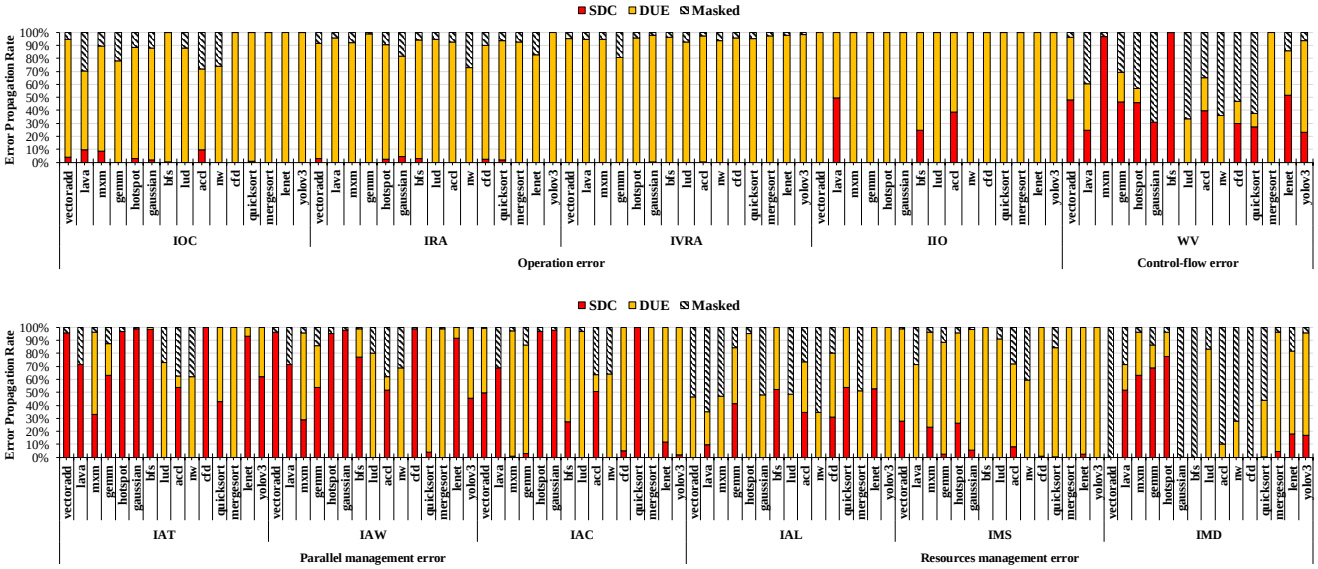

Figure 8: Error Propagation Rate results of each error model propagated on 15 applications. IPP and IVOC are not shown since IPP is similar to other models and IVOC induces only DUEs.

invalid instruction exception at the software level, leading to a Device Unrecoverable Error in all cases the error is injected.

5.2 Error propagation results

The software-based injection experiments have been performed on a workstation with an Intel i9-10900 CPU with 10 Cores, 32 GB of RAM, and one NVIDIA Ampere 3070ti GPU. We evaluated 15 real applications injecting 1,000 errors per application per error model. We target one sub-partition (PPB) of SM0. Overall, we inject more than 165,000 errors that took 300h of real GPU simulation.

In Figure 8 we report, for the 15 applications, the *Error Propagation Rate* (EPR), i.e., the probability for an error (produced by a fault that was activated and corrupted one or more of the unit outputs) to propagate to the software output. We plot the EPR for SDC, DUEs, and Masked. We group the results per error model. As discussed in Section 4, we show the 11 error models grouped by the four main error groups (i.e., *Operation Errors*, *Control-flow Errors*, *Parallel Management Errors*, and *Resource Management Errors*). We do not show IPP or IVOC error models since IPP can be implemented by any of the other error representations (IRA, IVRA, IAT, IAW, IMS, or IMD) and IVOC always generates DUEs at the low-level injections.

An interesting result from Figure 8 is the very high EPR for all error models and applications (the average EPR is 84.2%). The applications that are either compute-intensive (i.g., yolov3, lava, or LeNet) or instance many kernels during the execution (i.g., bfs, mergesort, and quicksort) present, for most of the error models, an EPR equal or close to 100%. It is worth noting that permanent faults, by definition, are less likely to be masked compared to transient faults, as the resources are permanently damaged.

We can also see that the code's characteristics can significantly impact the EPR. This is particularly evident in two error models, WV (work-flow) and IMD (incorrect memory destination). For the WV error model, codes with many control flow blocks or thread indexing limitations that, once modified, can impact a significant amount of data (i.e., vectoradd, mxm, gemm, hotspot, bfs, and gaussian) show a high SDC EPR. Additionally, applications that can impact the memory addressing or block synchronization (i.e., lud, nw, and mergesort) show a high DUE EPR. The EPR is changed similarly for the IMD error model. For many codes, the error model IMD has no impact on the execution (i.e., vectoradd, gaussian, bfs, and cfd). The IMD error model affects instructions that operate on shared memories by changing the register that is the source or destination of an instruction that loads or stores on shared memory. Consequently, codes that do not use shared memories will have 100% of the injected faults masked.

Figure 9 summarizes the main findings for the 11 evaluated error models by showing the *Average* EPR between all the codes. Interestingly, the group of **Operation Errors** shows a predominance of DUEs for all error models. In particular, on average, the percentage of IOC, IRA, IVRA, and IIO injections that generate a DUE is 87%, 90%, 95%, and 92%, respectively. The Operation Errors, as discussed in Section 5.1, have a particular characteristic of modifying the behavior of all or many instructions in one or all threads within a warp or multiple warps. When many instructions are modified due to a permanent fault, the expected outcome is to have at least one thread, or many threads, performing illegal instructions, accessing incorrect memory accesses, or operating with registers outside the thread register bounds, which leads to a DUE. In fact, the percentage of incorrect memory addresses and illegal instructions generated by IOC, IRA, IVRA, and IIO error models are, on average, 99.05%, 99.76%, 100%, and 98.29% of the total DUEs.

On the contrary, most of the error models that belong to the **Control-flow** and **Parallel Management** groups (WV, IAT, and

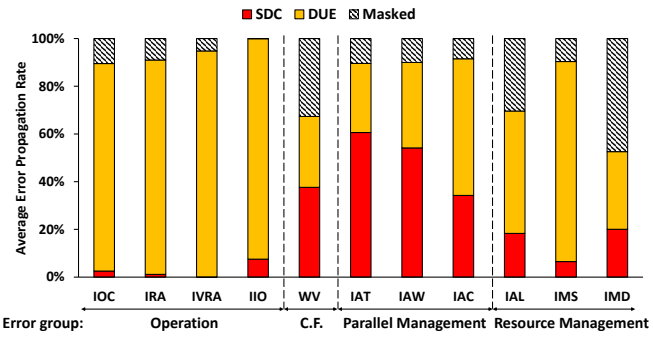

Figure 9: Average Error Propagation Rate among the 15 tested applications.

IAW) have a high SDC EPR which is, on average, 38%, 61%, and 54%, respectively. The combination of the error model and the executing code significantly changes these injections' outcomes. For example, when single or multiple threads are disabled due to the IAT error model, the output that would be expected from that thread will not be produced, generating an SDC. This is the case of codes like vectoradd, gaussian, cfd, and bfs, where the IAT error model enables/disables threads on the execution, and the code is able to finish (i.e., due to low interdependencies of the threads) but generates, most of the time, SDCs. Similar behavior is observed for IAW and WV error models, but in these cases, with a slightly higher incidence of DUEs than for IAT. In fact, these error models affect multiple threads or warps simultaneously, which can lead to the corruption of multiple output elements.

The errors from the **Parallel Management** group mostly induce SDCs in the applications. The only error model with an average DUE EPR higher than the SDC EPR is IAC (SDC EPR is 34% and DUE EPR is 57%). This happens because IAC is an error model that causes an incorrect execution (i.e., detention, assignation, or unauthorized submission) of an entire CTA (thread block) in the kernel execution, increasing the probability of DUEs. As with the other error models from the Parallel Management group, the EPR will change according to how the code uses the GPU resources. For instance, when the IAC error model is injected, applications such as lava, hotspot, gaussian, accl, and quicksort have most of the injections leading to SDC, i.e., the SDC EPR is 69%, 97%, 98%, 51%, and 99% respectively. This happens because those applications schedule many independent parallel CTAs, then an incorrectly assigned block may still finish and produce an incorrect output.

For the error models from the **Resource Management Errors** group, the EPR shows a strong dependence between the injection outcome and the code, where, for example, in the case of IMD, the use of shared memory can determine if the error will be masked or not. Similar behavior also can be observed for the IAL and IMS error models. However, as IAL and IMS affect resources used for all the codes (by disabling GPU lanes or causing an incorrect assignation of a memory resource for the result's storing), we can see that both error models impact all codes, increasing their average EPR.

5.3 Discussion

Our method dramatically reduces the time complexity required for the evaluation of permanent faults in GPU, allowing an accurate error characterization at the gate level and a practical propagation of errors at the application level. A similar evaluation using only low-level hardware descriptions would be simply unfeasible. In fact, the simulation of a complete GPU at gate level takes, in our server, ≈14.5 h for characterizing *one* permanent fault in *one* application. If we scale the simulation time for all workloads (15 applications) and all fault locations (50,044) we tested, we would reach a theoretical simulation time of around $14.5 \times 50,044 \times 15$ hours, that is 10.8×10^6 hours: ≈ 1,242 years! In contrast, our approach required only 20.5 h for profiling, 178.1 h for low-level characterization, 4.2 h for error analysis, and ≈ 300 h of software-level error propagation for all workloads and targeted units (502.8 h in total), so speeding up the simulation of more than four orders of magnitude.

The flexibility of our method allows its adaptation/extension for the evaluation of other GPU units and fault models (e.g. delay or transient). The low-level micro-architecture characterization process just requires the adaptation of the hardware profiling tool, according to the specifications of the new target component or fault model, to identify the input stimuli.

Correlating the low-level and software analyses we can identify the most probable instruction-level errors and the hardware units responsible for the observed application failures. From the low-level analyses (Table 3 and Figure 3), around 50% of faults in *fetch* and *decoder* units produce visible instruction-level errors. These faults are mainly mapped as parallel management and operation error classes (from 1% up to 15% in the fetch, and from 12% to 19% in the decoder). Moreover, we observe that all propagated faults (30.5%) in the WSC mainly affect the parallel management parameters (from 1% to 9%). Then, from the software level error injections (Figure 9), we found that the parallel management and control flow errors are likely to induce SDCs (between 38% to 60%). The resource management errors produce mainly DUEs and 20% of SDCs, while an operation error leads to DUEs in more than 90% of the cases. Thus, permanent faults on the WSC are more likely to generate, at the application level, SDCs, whereas the permanent faults affecting the fetch unit lead, in more than 90% of the cases, to DUEs (mainly due to illegal memory access). Finally, faults in the decoder unit have a higher probability of generating DUEs (70%) and SDCs (20%).

Our method can also support the design of detection and mitigation solutions for permanent faults. Adopting software detection techniques, in combination with smart scheduling and SMs swapping, allows fast fault detection or reduces the probability of permanent faults, thus potentially extending the in-field operation of GPUs. For example, we observed that most of the faults affecting the WSC generate SDCs at the application level. Control-flow-checking strategies combined with smart thread scheduling replication can be a potential countermeasure against permanent faults in the WSC. The control-flow-checking mechanism can be used to detect malfunctions during the execution of a given application. Then, the smart scheduling policy can discard the results from a faulty thread or warp by selecting the correct data from one of the replicated results. For the case of the Fetch and Decode units, since their corruption generates DUE, a hardware-base hardening technique is to be preferred. Given the Fetch and Decode units' importance and that a fault produces a catastrophic failure for all threads, it is unfeasible to use software-based mitigation strategies.

6 CONCLUSIONS

We have proposed a method to understand hardware permanent fault activation and propagation. We exploit a multi-level approach to combine the accuracy of gate-level simulation with efficient software-based error propagation. We focus on the WSC, Fetch and Decoder units and present the first quantitative estimation of their failure in the GPU code execution. We have identify four main groups of errors: *(i)* Operation errors, *ii)* Control-flow errors, *iii)* Parallel management errors, and *iv)* Resource management errors), corresponding to 13 instruction error categories. We have implemented the instrumentation functions for the 13 error categories and used them to propagate the error effects in 15 real applications, resorting to a specially crafted instruction-level error injector (NVBitPERfi).

The experimental results show that the permanent fault effect depends on the corrupted unit and executed instruction. Faults in the fetch unit mainly (66.80%) lead to *Operation* errors, faults in the Decoder unit lead to *operation* (44.32%) and *resource management* (38.35%) errors, and faults in the scheduler lead to *parallel management* errors (54.87%). The software-level propagation of the observed error categories, shows that parallel management errors (mainly generated within the WSC) generate a high amount of SDCs (20% to 60%), whereas faults in the Fetch and Decoder units mainly lead to DUEs (> 90% and 70%, respectively) .

REFERENCES

[1] 2023. SC 2023 Anonymous repository. Anonymous link.
[2] Jae-Gyung Ahn, Rhesa Nathanael, I-Ru Chen, Ping-Chin Yeh, and Jonathan Chang. 2021. Product Lifetime Estimation in 7nm with Large data of Failure Rate and Si-Based Thermal Coupling Model. In *2021 IEEE International Reliability Physics Symposium (IRPS)*. 1–6. https://doi.org/10.1109/IRPS46558.2021.9405193
[3] Muhammed Al Kadi et al. 2016. FGPU: An SIMT-Architecture for FPGAs. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '16)*. 254–263.
[4] Sergi Alcaide et al. 2018. Safety-Related Challenges and Opportunities for GPUs in the Automotive Domain. *IEEE Micro* 38, 6 (2018), 46–55. https://doi.org/10.1109/MM.2018.2873870
[5] Hussam Amrouch and Jorg Henkel. 2015. Reliability degradation in the scope of aging — From physical to system level. In *2015 10th International Design & Test Symposium (IDT)*. 9–12. https://doi.org/10.1109/IDT.2015.7396727
[6] Raghuraman Balasubramanian and Karthikeyan Sankaralingam. 2014. 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)*. 60–71. https://doi.org/10.1109/HPCA.2014.6835976
[7] Giani Braga et al. 2021. Evaluating softcore GPU in SRAM-based FPGA under radiation-induced effects. *Microelectronics Reliability* 126 (2021), 114348. https://doi.org/10.1016/j.microrel.2021.114348 Proceedings of ESREF 2021, 32nd European Symposium on Reliability of Electron Devices, Failure Physics and Analysis.
[8] Hyungmin Cho et al. 2015. Understanding Soft Errors in Uncore Components. In *Proceedings of the 52nd Annual Design Automation Conference (DAC '15)*. Article 89, 6 pages. https://doi.org/10.1145/2744769.2744923
[9] Josie E. Rodriguez Condia et al. 2020. FlexGripPlus: An improved GPGPU model to support reliability analysis. *Microelectronics Reliability* 109 (2020), 113660.
[10] Josie E. Rodriguez Condia et al. 2021. Combining Architectural Simulation and Software Fault Injection for a Fast and Accurate CNNs Reliability Evaluation on GPUs. In *2021 IEEE 39th VLSI Test Symposium (VTS)*. 1–7. https://doi.org/10.1109/VTS50974.2021.9441044
[11] C. Constantinescu. 2003. Trends and challenges in VLSI circuit reliability. *IEEE Micro* 23, 4 (2003), 14–19. https://doi.org/10.1109/MM.2003.1225959
[12] F. F. d. Santos et al. 2019. Analyzing and Increasing the Reliability of Convolutional Neural Networks on GPUs. *IEEE Transactions on Reliability* 68, 2 (2019), 663–677. https://doi.org/10.1109/TR.2018.2878387
[13] David Defour and Eric Petit. 2013. GPUburn: A system to test and mitigate GPU hardware failures. In *2013 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. 263–270. https://doi.org/10.1109/SAMOS.2013.6621133
[14] B. Fang et al. 2014. 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*. 221–230. https://doi.org/10.1109/ISPASS.2014.6844486
[15] Bo Fang et al. 2016. ePVF: An Enhanced Program Vulnerability Factor Methodology for Cross-Layer Resilience Analysis. In *46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 168–179. https://doi.org/10.1109/DSN.2016.24
[16] Oscar Ferraz et al. 2022. A Survey on High-Throughput Non-Binary LDPC Decoders: ASIC, FPGA, and GPU Architectures. *IEEE Communications Surveys & Tutorials* 24, 1 (2022), 524–556.
[17] D. A. G. Goncalves de Oliveira et al. 2016. Evaluation and Mitigation of Radiation-Induced Soft Errors in Graphics Processing Units. *IEEE Trans. Comput.* 65, 3 (2016), 791–804.
[18] Juan-David Guerrero-Balaguera et al. 2022. Evaluating the impact of Permanent Faults in a GPU running a Deep Neural Network. In *2022 IEEE International Test Conference in Asia (ITC-Asia)*. 96–101. https://doi.org/10.1109/ITCAsia55616.2022.00027
[19] Said Hamdioui et al. 2013. Reliability challenges of real-time systems in forthcoming technology nodes. In *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 129–134. https://doi.org/10.7873/DATE.2013.040
[20] Jin-Woo Han et al. 2021. Single Event Hard Error due to Terrestrial Radiation. In *2021 IEEE International Reliability Physics Symposium (IRPS)*. 1–6. https://doi.org/10.1109/IRPS46558.2021.9405177
[21] Siva Kumar Sastry Hari et al. 2022. Making Convolutions Resilient Via Algorithm-Based Error Detection Techniques. *IEEE Transactions on Dependable and Secure Computing* 19, 4 (2022), 2546–2558. https://doi.org/10.1109/TDSC.2021.3063083
[22] Jörg Henkel et al. 2013. Reliable on-chip systems in the nano-era: Lessons learnt and future trends. In *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1–10. https://doi.org/10.1145/2463209.2488857
[23] IEEE. 2022. THE INTERNATIONAL ROADMAP FOR DEVICES AND SYSTEMS: 2022. In *Institute of Electrical and Electronics Engineers (IEEE)*.
[24] Kojiro Ito et al. 2021. Analyzing DUE Errors on GPUs With Neutron Irradiation Test and Fault Injection to Control Flow. *IEEE Transactions on Nuclear Science* 68, 8 (2021), 1668–1674. https://doi.org/10.1109/TNS.2021.3098845
[25] Datla Jagannadha et al. 2019. Special Session: In-System-Test (IST) Architecture for NVIDIA Drive-AGX Platforms. In *2019 IEEE 37th VLSI Test Symposium (VTS)*. 1–8. https://doi.org/10.1109/VTS.2019.8758636
[26] Haeseung Lee et al. 2018. Aging-Aware Workload Management on Embedded GPU Under Process Variation. *IEEE Trans. Comput.* 67, 7 (2018), 920–933. https://doi.org/10.1109/TC.2018.2789904
[27] G. Leon et al. 2020. Evaluating the soft error sensitivity of a GPU-based SoC for matrix multiplication. *Microelectronics Reliability* 114 (2020), 113856. https://doi.org/10.1016/j.microrel.2020.113856 31st European Symposium on Reliability of Electron Devices, Failure Physics and Analysis, ESREF 2020.
[28] Wei Li and Cher Ming Tan. 2011. Black's equation for today's ULSI interconnect Electromigration reliability — A revisit. In *2011 IEEE International Conference of Electron Devices and Solid-State Circuits*. 1–2. https://doi.org/10.1109/EDSSC.2011.6117717
[29] Atieh Lotfi et al. 2019. Resiliency of automotive object detection networks on GPU architectures. In *2019 IEEE International Test Conference (ITC)*. 1–9. https://doi.org/10.1109/ITC44170.2019.9000150
[30] Mayler Martins et al. 2015. Open Cell Library in 15nm FreePDK Technology. In *Proceedings of the 2015 Symposium on International Symposium on Physical Design (ISPD '15)*. 171–178.
[31] J. R. Nickolls et al. 2005. Systems and methods for voting among parallel threads. US Patent No. 8,200,947B1.
[32] B. Nie et al. 2018. Fault Site Pruning for Practical Reliability Analysis of GPGPU Applications. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 749–761. https://doi.org/10.1109/MICRO.2018.00066
[33] Sergiu Nimara et al. 2016. Multi-level simulated fault injection for data dependent reliability analysis of rtl circuit descriptions. *Advances in Electrical and Computer Engineering* 16, 1 (2016), 93–98.
[34] C. Nvidia. 2022. Programming Guide: CUDA Toolkit Documentation. Available online on 01/12/2022.
[35] Sangwoo Pae et al. 2008. Effect of BTI Degradation on Transistor Variability in Advanced Semiconductor Technologies. *IEEE Transactions on Device and Materials Reliability* 8, 3 (2008), 519–525. https://doi.org/10.1109/TDMR.2008.2002351
[36] George Papadimitriou and Dimitris Gizopoulos. 2021. Demystifying the System Vulnerability Stack: Transient Fault Effects Across the Layers. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 902–915.
[37] C. Prasad et al. 2013. Self-heat reliability considerations on Intel's 22nm Tri-Gate technology. In *2013 IEEE International Reliability Physics Symposium (IRPS)*. 5D.1.1–5D.1.5. https://doi.org/10.1109/IRPS.2013.6532036
[38] P. Rech et al. 2014. 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*. 455–466. https://doi.org/10.1109/DSN.2014.49

[39] Paolo Rech et al. 2014. Measuring the Radiation Reliability of SRAM Structures in GPUS Designed for HPC. In *IEEE 10th Workshop on Silicon Errors in Logic - System Effects (SELSE)*.

[40] Fernando F. dos Santos et al. 2021. Revealing GPUs Vulnerabilities by Combining Register-Transfer and Software-Level Fault Injection. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 292–304. https://doi.org/10.1109/DSN48987.2021.00042

[41] Anderson L. Sartor et al. 2019. A Fast and Accurate Hybrid Fault Injection Platform for Transient and Permanent Faults. *Design Automation for Embedded Systems* 23, 1–2 (2019). https://doi.org/10.1007/s10617-018-9217-0

[42] Dimitris Sartzetakis et al. 2022. gpuFI-4: A Microarchitecture-Level Framework for Assessing the Cross-Layer Resilience of Nvidia GPUs. In *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 35–45. https://doi.org/10.1109/ISPASS55109.2022.00004

[43] Andrzej J. Strojwas, Kelvin Doong, and Dennis Ciplickas. 2019. Yield and Reliability Challenges at 7nm and Below. In *2019 Electron Devices Technology and Manufacturing Conference (EDTM)*. 179–181. https://doi.org/10.1109/EDTM.2019.8731146

[44] Michael B. Sullivan et al. 2021. *Characterizing And Mitigating Soft Errors in GPU DRAM*. Association for Computing Machinery, New York, NY, USA, 641–653. https://doi.org/10.1145/3466752.3480111

[45] A. S. TIRUMALA et al. 2017. Techniques for comprehensively synchronizing execution threads. US Patent No. 10,977,037B2.

[46] Timothy Tsai et al. 2021. NVBitFI: Dynamic Fault Injection for GPUs. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 284–291. https://doi.org/10.1109/DSN48987.2021.00041

[47] A. Vallero et al. 2017. SIFI: AMD southern islands GPU microarchitectural level fault injector. In *2017 IEEE 23rd International Symposium on On-Line Testing and Robust System Design (IOLTS)*. 138–144. https://doi.org/10.1109/IOLTS.2017.8046209

[48] Oreste Villa et al. 2019. NVBit: A Dynamic Binary Instrumentation Framework for NVIDIA GPUs. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 372–383. https://doi.org/10.1145/3352460.3358307

[49] Jack Wadden et al. 2014. Real-world design and evaluation of compiler-managed GPU redundant multithreading. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. 73–84. https://doi.org/10.1109/ISCA.2014.6853227

[50] J. Wei et al. 2014. 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*. 375–382.

[51] Lishan Yang et al. 2021. Practical Resilience Analysis of GPGPU Applications in the Presence of Single- and Multi-Bit Faults. *IEEE Trans. Comput.* 70, 1 (2021), 30–44. https://doi.org/10.1109/TC.2020.2980541

Received June 19, 2023