A Multi-level Approach to Evaluate the Impact of GPU Permanent Faults on CNN's Reliability

A Multi-level approach to evaluate the impact of GPU Permanent Faults on CNN's Reliability

Josie E. Rodriguez Condia*, Juan-David Guerrero-Balaguera*, Fernando F. dos Santos†,
Matteo Sonza Reorda*, Paolo Rech‡
*Politecnico di Torino - Department of Control and Computer Engineering DAUIN
{josie.rodriguez, juan.guerrero, matteo.sonzareorda}@polito.it
†Institut National de Recherche en Sciences et Technologies du Numerique (INRIA)
fernando.fernandes-dos-santos@inria.fr
‡University of Trento - Department of Industrial Engineering
paolo.rech@unitn.it

***Abstract*—Graphics processing units (GPUs) are widely used to accelerate Artificial Intelligence applications, such as those based on Convolutional Neural Networks (CNNs). In some domains in which CNNs are heavily employed (e.g., automotive and robotics), the expected life-time of GPUs is over 10 years. As in these domains reliability is a major concern, the analysis of the impact of permanent faults (e.g., due to aging) needs to be paramount. Crucially, while the impact of transient faults on GPUs running CNNs has been widely studied, an accurate evaluation of the impact of permanent faults is still lacking. Performing this evaluation is a challenge, due to the complexity of GPU devices and of the software implementing a CNN. In this work, we propose a methodology that combines the accuracy of gate-level fault simulation with the speed and flexibility of software fault injection to evaluate the effects of hardware permanent faults affecting a GPU. First, we profile the executed low-level GPU instructions during the CNN execution. Then, using extensive gate-level fault injection campaigns, we provide an accurate analysis of the effects of permanent faults on the internal modules executing the targeted instructions. Finally, these effects are propagated in the application using a fast software-based fault injection. The method allows for the first time to estimate the percentage of permanent faults leading the CNN to produce wrong results (i.e., changing the result of its work). The feasibility of the method, which allows to flexibly trade-off accuracy with the required computational effort, is shown using LeNet running on a Ampere Nvidia GPU as a case-study. The method reduces the computational effort for the evaluation in several orders of magnitude.**

***Index Terms*—Convolutional Neural Networks (CNNs), Graphics Processing Units (GPUs), Permanent faults**

I. Introduction

Modern GPUs are known to be highly susceptible to *transient* faults. The sensitivity of GPUs and parallel applications, including Convolutional Neural Networks (CNNs), to radiation and other sources of transient faults has been widely studied through beam experiments [1], software fault injection [2], [3], and microarchitectural fault simulation [4], [5]. These faults can cause the output corruption of a running application and lead to catastrophic consequences in safety-critical domains.

Recently, GPUs have been adopted in applications (such as automotive, robotics, aerospace, and health care) in which the device life expectation is in the order of 5 to 10 years. This life expectation is much longer than the typical 1-2 years for GPUs used in gaming, mining, or high performance computing applications and poses novel challenges in the GPUs reliability evaluation. In fact aging, degradation, and wear-out effects must now be considered, as they can lead to *permanent* hardware faults in the GPUs, which may produce unacceptable critical effects when the GPU is used in a safety-critical domain.

Unfortunately, there are very limited research results available to estimate how frequently a permanent fault affecting a GPU which runs a CNN may produce a critical failure (e.g., a failure that produces a misclassification if the CNN is used as a classifier). The probability of a (permanent) fault to produce a critical failure depends not only on the architecture of the underlying GPU, but also on the characteristics of the software implementing the CNN. An exhaustive gate-level fault simulation is impractical for this purpose due to the unacceptably high computational requirements. In fact, the parallel nature of GPU architecture, the huge number of possible faults due to the number of gates (millions of them for a GPU core), and the complexity of the software implementation and architecture of the CNN ($\approx$3M weights and $\approx$9K neurons for LeNet5) make the permanent fault effect evaluation extremely challenging. Therefore, a novel accurate and efficient approach is required. What we propose is to combine the speed of high-level software fault injection with the detailed analysis supported by gate-level fault simulation.

In this paper, we propose a combined fault injection framework that enables a detailed and efficient evaluation of the effects of permanent faults in GPUs executing CNNs. To track the effect of a permanent fault, our framework combines the fault injection in two different abstraction levels (hardware and software). The low-level microarchitectural hardware effect of a permanent fault is determined using an open-source model of the hardware of an Nvidia GPU (FlexGripPlus [6]) and then propagated in a real GPU device at the software level. This approach reduces the prohibitively high execution times of low-level microarchitectural hardware simulations for complex algorithms, such as those for managing CNNs by several orders of magnitude, while still allowing an accurate and detailed permanent fault analysis. For example, considering an

RT-level GPU model, one fault campaign requires about eight days to evaluate a 32X32 matrix multiplication (MxM). Thus, the complete evaluation of a CNN, such as LeNet5, which incorporates hundreds of MxM operations plus further max-pooling and activation operations, would require unfeasible simulation times ($>$ 10,000 days!).

In principle, the proposed framework can be used to perform a permanent fault analysis in all GPU resources. In this paper we show the feasibility of the method by focusing on the effects of faults affecting the two main functional units, i.e., the Floating Point Unit (FPU) and the Integer core (INT). Our approach is based on four main steps: *i)* software profiling, *ii)* gate-level fault injection, *iii)* software-level fault propagation, and *iv)* fault classification.

We first profile the CNN application in software, identifying the information about each instruction that uses a target functional unit (i.e., input values, output result, and opcode). This software profiling step resorts to a binary instrumentation tool based on NVBit [7], which traces (at run time) also the parallel operational configurations for software (*thread_ID*, *warp_ID*, *Block_ID*) and hardware (*lane_ID*, and *SM_ID*) parameters.

In the second step, we simulate each permanent fault in a functional unit (FPU or INT) resorting to a low-level microarchitectural fault simulator, tracking its effects on the outputs of the unit during the execution of each single instruction.

The third step propagates the permanent fault effects through the CNN application. For this purpose, we run the application in a real GPU, injecting the effects of the fault at the end of the execution of each instruction using a specific hardware core (i.e., CUDA core). Finally, the error effect is classified and the critical effects are traced-back to identify the source structure that has produced them.

As a case study, we choose the LeNet CNN, composed of 32 parallel kernels, to evaluate the impact of permanent hardware faults in the functional units at the application level. The experimental results show that hardware faults in the FPU cause a significant percentage of critical effects in the CNN ($\approx$10% for the FADD, $\approx$5.4% for FMUL, and $\approx$15.5% for FFMA instructions). Moreover, results allow us to state that hardware permanent faults in the Integer core of a GPU can collapse the operation of the hardware accelerator (this happens in $\approx$97.5% of the faults related to the IADD3 instruction). The proposed framework required about 54 hours for the error propagation experiments targeting one instruction with a set of fault error syndromes, thus reducing by several orders of magnitude the required time for the evaluation of a CNN, when compared with a fully microarchitectural characterization.

To the best of our knowledge, our method is the first allowing to evaluate the impact on the application level of low-level microarchitectural permanent hardware faults in a GPU when running a CNN.

The reminder of the paper is organized as follows: Section II presents background and related work on GPUs reliability analysis. Section III introduces the proposed two-levels method. Section IV describes the implementation of the proposed approach for CNNs. Section V presents and discusses the experimental results, while Section VI draws conclusions and introduces future works.

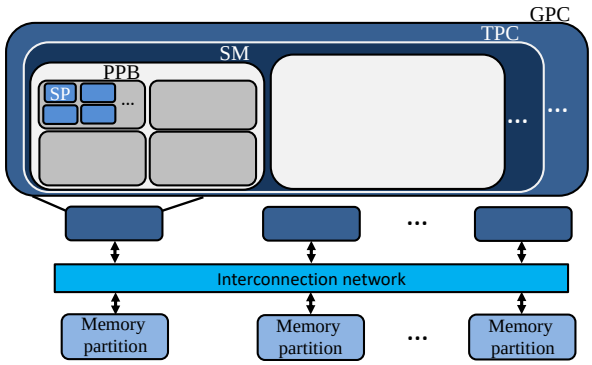

Fig. 1. A general scheme of the internal organization of a GPU.

II. Background and Related Work

A. Organization of a GPU

GPUs are special-purpose processors designed to exploit hardware parallelism and provide a high throughput in the execution of applications. Currently, modern GPU designs are mainly composed with a set of homogeneous cores organized in a hierarchical fashion, including *Graphics Processing Clusters* (GPCs), *Texture Processing Clusters* (TPCs), and *Streaming Multiprocessors* (SMs), or *SIMD Engines*, see Figure 1.

The SMs are the principal execution unit in a modern GPU and are partitioned into 2 to 4 Parallel Processing Blocks (PPBs). Each SM includes one task scheduler controller to distribute the task *(Blocks)* among the PPBs cores. In detail, each PPB handles a set of parallel functional units (known as *Streaming Processors* or SPs, or 'CUDA cores'), and a register file. Internal schedulers, in the PPB, distribute groups of threads *(warps)* among the available SPs. Each SP includes Floating-Point Units (FP32/FP64) and Integer cores (INT).

B. System abstraction layers and fault propagation

Modern electronic systems, such as GPUs, are composed of stacked hardware and several software abstraction layers. In detail, the hardware layers implement the functionalities of an Instruction Set Architecture (ISA) physically. This implementation is defined by the microarchitecture that describes the building blocks of the device from high to low level. On the software side, the applications implement algorithms through assembly language to interact with the hardware according to the ISA's specifications.

A fault at the hardware level might produce issues in upper layers or be masked without producing any observable effect. When the fault occurs it may be propagate through the hardware structures affecting also the operation of software layers and generate critical system failures (see Figure 2). Although some faults might be masked in the different layers of the system, other faults can reach the system's software layer and produce system failures [8].

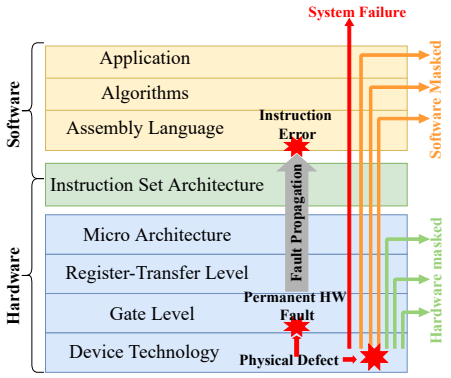

Fig. 2. System abstraction layers and fault propagation effects.

C. Dynamic binary instrumentation tool

To profile the execution of the software in the GPU hardware and to inject faults in the code execution we take advantage of NVIDIA's NVBit and NVBitFI.

NVBit is a dynamic binary instrumentation tool framework developed for NVIDIA GPUs [7]. This framework offers APIs that cooperate with the CUDA driver APIs to perform instruction's inspection and inject arbitrary CUDA functions to kernels before launching them. A new instrumentation tool can exploit the use of customized CUDA/C/C++ functions (controlled by the NVBit APIs) to incorporate special functionalities into a pre-compiled binary application regardless of libraries integrated in the application (e.g., cuBlas or cuDNN).

NVBitFI is an instrumentation tool based on the NVBit framework. This tool has been specially designed to perform reliability evaluation of any application developed for NVIDIA GPUs [3]. NVBitFI employs the Hardware Injection through Program Transformation (HIPT) technique to mimic hardware faults using error injection through instrumentation functions designed to modify the results of the issued SASS instructions. The tool operates in different phases. First, the target kernel is intercepted by the callback APIs before the GPU issues it. Then, the inspection and instrumentation APIs are used to insert the error function after the target instructions. Finally, the new version of the kernel is executed in the GPU device to propagate the desired error to the application's output. NVBitFI mainly offers support to evaluate the reliability of GPUs under transient faults model with a minimal and *simplistic approach* for modeling permanent faults. Therefore, the adoption of multi-abstraction level methodologies offers a solid way to create a more realistic scenario by combining the accuracy of low-level evaluations with the speed of software error propagation. Despite the fact that NVBitFI mechanism targets only NVIDIA GPUs, it could be scaled into other GPU vendors considering the guidelines and equivalent tools described in [3].

D. Related works

The implicit parallelism and programming flexibility of GPUs make them the preferred platforms to accelerate various codes, especially CNN, in safety-critical applications. In this domain, reliability is one of the most important constraints. It has already been shown that complex and big devices as GPUs suffer from a very high transient fault error rate, that could jeopardize the device reliability [1]–[4], [9].

The GPUs employed in gaming, high performance computing, and mining systems are typically replaced after a few years. As GPUs are included in automotive or robotic applications, their operative life is expected to be much longer, in the range from 5 to 10 years. Unfortunately, after the certified life period, a GPU may start suffering from fatigue or degradation (aging or wear-out) of its internal components, such as internal cores (SMs) and memories. Permanent hardware faults could then arise and affect the operation of a running application.

Aging-related reliability issues have already been widely studied in CPUs [10], [11], and more recently in hardware accelerators [12], [13]. The related research focused on identifying methods to evaluate the most sensitive locations in the processors affected by permanent faults, in order to develop mitigation or fault handling solutions. In CPUs, the approaches to evaluate the effect of permanent faults consider architectural simulation, software fault injection [14], and microarchitectural simulation and emulation. These methods are applicable only for small and medium designs and could require the combination of different strategies when dealing with large CPU designs [15]. In most cases, combining one or two strategies is enough to identify the main fault effects.

An evaluation of permanent fault effects in GPUs is still missing and a framework to track permanent fault propagation in a parallel code is still lacking. Unlike traditional CPUs, GPUs are complex accelerators; they include hardware support for thousands (or even millions) of threads, need complex programming environments, and the amount of possible fault sites is huge. The permanent fault effects evaluation for GPUs is, then, particularly challenging, and the evaluation approaches already available for CPUs are impractical for GPUs. The technical complexity of controlling several parallel threads, executed on different SMs alone, makes the propagation of permanent faults on a given unit much more complex than in a normal CPU.

Previous works [16], [17] have shown that architectural fault simulators are necessary to evaluate the effect of permanent faults effects in a computing device. Unlike transient faults, a permanent fault evaluation needs to identify the input vectors that activate the fault. A software fault injector that modifies the output of an operation cannot perform this task. Unfortunately, technical limitations in the simulators (poor representation of fault models and fault effects) and the required computational effort limit the exploration and evaluation of complex applications, such as CNNs.

For our paper we took inspiration from the use of multi-level frameworks that combines microarchitectural simulation and physical emulation [18], or software fault injection [19]. The latter approach has been successfully used in CPUs. In [20], the authors exploit context switching between RT-level and Gate-level abstractions to combine high fault simulation speed and accuracy in CPUs. However, these techniques can hardly

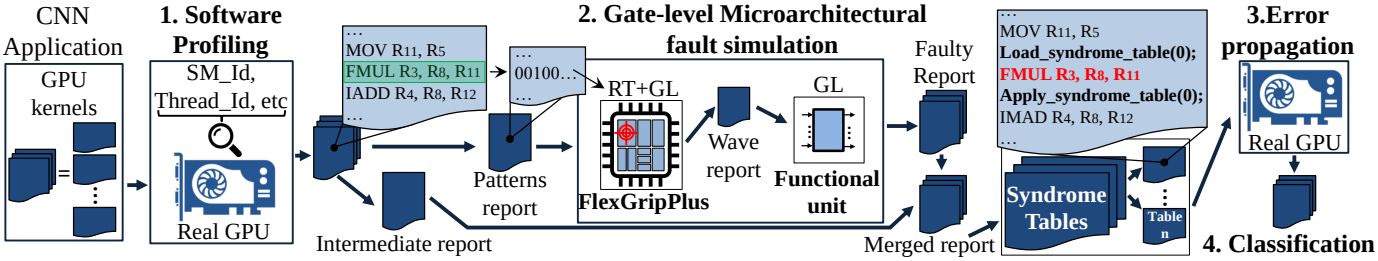

Fig. 3. A general scheme of the proposed multi-level method to evaluate permanent faults in GPUs. The application profiling identifies the instructions (and their inputs) that are mapped to a target hardware module. The gate-level simulation performs the permanent fault injection, identifying the inputs that activate each fault and reporting its effects on the output. Then, a software fault injection propagates fault effects for all the instructions executed on the target unit.

used in dense NN applications. Other work [21] proposed an hybrid fault injector approach integrating software-application and RT-level abstractions boosted with pipeline-type stages to accelerate the evaluation of NNs in CPUs. For GPUs, some works [2], [22] proposed multi-level approaches combining high-level architectural simulation and error propagation in real GPUs with the purpose of evaluating transient fault effects [23]. Nevertheless, the adoption of this multi-level philosophy for the evaluation of permanent faults has not fully explored in GPUs. It is worth noting that, even in CPUs, the permanent fault evaluation can be so complex that most studies limit the evaluation to memories [24]. We aim at going a step further for GPUs and focus on faults in the functional units.

While our work takes inspiration from the two-level fault injection concept, none of the previous works address the evaluation of permanent fault effects in GPUs. The proposed framework selects the software instructions mapped on a specific hardware unit and identifies the inputs that activate the fault. Then, the permanent fault effects are injected and propagated, in software, in a real GPU executing a code, e.g., a CNN. To the best of our knowledge this is the first work proposing a multi-level framework to evaluate permanent fault effects in GPUs.

III. Proposed Methodology

In this section, we describe the proposed methodology to evaluate the effects of hardware permanent faults in the functional units of a GPU when executing parallel applications. The proposed method is based on the combination of a two-levels abstraction approach (gate-level microarchitectural fault simulation and software-based fault injection). The method allows to evaluate the effects of permanent faults even in complex application such as CNNs.

A. Main Idea

The evaluation of permanent hardware faults effects is a highly time consuming task. This is because the effect of the fault can be measured only through low-level simulations (i.e., at gate-level) that, normally, require a huge amount of time to be completed. This is exacerbated in parallel and dense devices such as GPUs. Running a whole application in low-level simulators for GPUs is actually unfeasible.

Our idea to overcome these limitations is to first identify, through software profiling, the machine instructions (and their inputs/output) that compose the kernels of a CNN. Then, this information is used to perform gate-level micro-architectural simulations (exclusively on the functional unit of the GPU performing a targeted instruction) with the purpose of evaluating the impact of permanent faults and their propagation to the output of the operation. We inject a permanent fault (Stuck-at 0/1) in each site of a functional unit and execute an individual instruction (i.e., FFMA) to operate all the inputs provided by the software profile. This procedure allows the identification of all inputs that activate a fault and the effect in the outputs of an instruction. Finally, we propagate, in software, the corrupted outputs during the execution of the code in a real GPU (targeting the analyzed instructions, only) to represent the propagation of permanent faults.

The proposed method, as depicted in Figure 3 and detailed in the following, is composed of four steps: *i)* Software profiling and target identification, *ii)* Gate-level micro-architectural fault injection, *iii)* Software-level fault propagation, and *iv)* Classification.

B. Software profiling and target identification

This step aims at collecting and tracing all the executed instructions from the kernels of a CNN in the GPU. The main target is to use the collected information to identify the relation between the executed instructions and the functional units of the GPU (e.g., FADD and FFMA in the FPU, and IADD in the INT core), so identifying the set of instruction in the CNN to evaluate. Moreover, the software profiling provides fine-grain and detailed information concerning the execution of each instruction. This information includes the distribution of each instruction among the resources of the GPU. These resources are described in terms of software parameters (*Blocks*, *Warps*, and *Threads*) and hardware (*Streaming multiprocessor cores* or SMs, and CUDA cores or *Lanes*), so providing virtual *(software)* and physical *(hardware)* identification of each executed instruction in the system. Both sets of parameters (software and hardware) are later use to propagate any fault effect in the software-level fault propagation.

The software profiling step provides a complete report (*Golden profile report*) of the application executed on a real device and includes all executed instructions in the GPU. The

report incorporates (for each instruction), the input operands (from the register file or memories), the software configuration parameters (*thread_ID*, *warp_ID*, and *Block_ID*), and the hardware core used (*lane_ID*, and *SM_ID*).

From the Golden profile report, we extract a sub-set of instructions and their information (e.g., IADD, or FFMA) as main candidates for evaluation in the gate-level fault injection campaigns. Then, the profile report (containing only the sub-set of instruction for evaluation) is divided into two reports: *1)* the 'Patterns Report' and *2)* the 'Intermediate Report', as depicted in Figure 3. The Patterns Report contains only the input operands and the mnemonic of the instruction for the gate-level microarchitectural fault simulation. In contrast, the intermediate report holds the complementary information for each instruction, which is later used in the software-level fault propagation step to localize targets for error injection. It is worth noting that redundant inputs from the same instruction are not included in the patterns report.

C. Gate-level microarchitectural fault simulation

The accurate study of hardware fault effects on the output of an instruction (i.e., IADD or FFMA) is the main targets of the gate-level fault simulation step. We focus on atomic instructions as we will then propagate in software (at application level) the observed effects. For this purpose, rather than simulating a fault in a given instruction in the complete micro-architecture of a GPU, the proposed method only focuses on the gate-level microarchitectural fault simulation of the specific functional unit. Since these units are part of the data-path and are directly connected to the memory hierarchy in the GPU (register file and memories), the specific gate-level fault simulation propagates any permanent fault impact from the functional units exactly as a complete micro-architectural simulation. In other words, emulating the whole GPU would just increase the simulation time while not improving accuracy.

A fault-free execution of the targeted instruction is performed at the microarchitectural level. We have crafted the GPU model with a complete RT-level description of just the target functional unit of the GPU to provide the instruction golden operation (i.e., fault free data). This mixed description of the GPU model (RT and gate-level) provides the accuracy of the gate-level execution inside the functional unit with a reduced simulation time removing the details of the other units, unused in the evaluation. In this step, the input operands from the patterns report are used to obtain the golden outputs of the instruction, which are then stored as a complete report ('wave report') and later used in the fault simulation campaigns. Moreover, this report is also employed in the procedures of comparison and classification of fault effects.

A set of gate-level fault injection campaigns are performed exclusively at the functional unit, using the wave report. This campaigns provide a fine-grain control of all possible faults in the unit, providing exhaustive or focused fault injection configurations (i.e., subsets of faults in flip-flops, or ports).

The fault injection campaign starts with the placement of one permanent hardware fault (stuck-at) inside the functional unit. In addition, one group of input operands from the wave report is applied in sequence for simulation (i.e. the values of R_1 and R_2 in IADD R_3, R_1, R_2). Then, the obtained outputs are stored for later analysis. Finally, the fault simulation restarts with the placement of a new permanent fault and the injection of all groups of input operands. This procedure is repeated for all targeted faults in a functional unit.

A post-processing step identifies the propagated fault effects from the collected results by comparing the collected results against the golden ones from the wave report. In case of a mismatch, the result is stored as propagated faulty results. Otherwise, the results, free of fault effects, are discarded.

When the fault injection is completed, the identified faulty results (those containing any fault effect from a permanent fault) are merged with the information stored in the intermediate report. This 'merged report' (see Figure 3) is useful to identify at the software level the locations and the instruction effect by the propagation of a permanent hardware fault, since this report contains the identification of the permanent fault in the unit, input operands, the golden output result, the faulty result, the mnemonic of the instruction and the parallel configuration parameters.

D. Software-based fault propagation

In this step the collected hardware fault effects are propagated as instruction errors in the software application. For this purpose, the code of the application is instrumented (off-line) with flexible functions to inject and propagate the effects of permanent hardware faults across a CNN application as errors in the instructions (fault syndrome). We use a real GPU to reduce the evaluation time of the complete CNN application. Algorithm 1 outlines the proposed method to propagate error effects at the instruction-level through the CNN.

As first step, the merged report is divided according to the identified hardware faults during the gate-level experiments. Thus, the software-based fault injection propagates (each time) the equivalent micro-architectural fault effects in a given instruction (as an instruction error) affected by a hardware fault. The propagation (of instruction errors as an equivalent fault effect) is based on 'Fault Syndromes', which are build from the bit-wise comparison between the golden and faulty values from the gate-level fault simulation. Each Fault Syndrome contains the output effect of a hardware fault affecting an instruction, so the error effects are propagated across the application during the software-based fault injection.

Since a given instruction (i.e., FADD) can be used several times in the CNN, we propagate the fault effect each time the instruction is executed on an specific functional unit in the GPU (combination of *SM*, *PPB* and *Lane*) resorting to *'Syndrome Tables'*, which are a collection of fault syndromes and allow the injection of specific error effects. Each input pattern applied to the evaluated functional unit can excite one permanent fault differently; thus, one fault may produce multiple error syndromes during the execution of a CNN.

One syndromes' table is available during the execution of the CNN to support the code instrumentation and to allow the

propagation of error effects. Then, the selection and propagation of a syndrome (from the table) follows three steps during the run-time application on the GPU: *i)* retrieval of the fault syndrome (*Load_syndrome*), *ii)* execution of the instruction, *iii)* propagation of the error syndrome (*Apply_syndrome*), see Figure 3. First, in the retrieval step, a matching procedure searches the specific instruction and identifies, from the syndrome table, the feasible syndrome representing the fault effect. Two parameters (inputs and instruction-type) are used to search inside the syndrome table. Then, during the propagation step, the output value of the instruction is processed with the fault syndrome to produce the error effect. Finally, the error is injected by replacing the original output value with the affected one and the CNN resumes.

It is worth noting that propagating permanent fault effects as instruction errors is a challenging task, since the effect of a permanent fault must remain active across the application, but only for those instructions executed on an affected hardware unit (i.e., combination of a given *SM*, *PPB*, and *Lane*). For this purpose, the syndrome tables are used as a highly flexible mechanisms to inject error effects from the gate-level results. These tables contain all possible fault effects from an instruction, so it is possible to use the two conditional functions (*load_syndrome* and *apply_syndrome*) to inject and evaluate fault effects (on the GPU's structures) when a target instruction is found. Moreover, the mechanism of tables of syndromes can be used for any number of instructions and it is independent of the application, so it is possible to scale their use in the evaluation of other applications.

Algorithm 1 Propagation algorithm of instruction error effects

Input: Syndrome table for fault F_i; Faulty location in the GPU (*SM*, *PPB*, and *Lane*); Target opcode instruction OP_t
Output: Fault classification for F_i (DUE, SDC, Masked)

```
load syndrome table for F_i
for each kernel K_i in the CNN model do
    for each instruction I_j in K_i do
        Inspection(I_j)
        if I_j matches the target OP_t then
            Insert instrumentation function before I_j
            Insert instrumentation function after I_j
        end if
    end for
    Just-In-Time compilation
    Launch the GPU execution of the instrumented kernel
end for
Assign failure classification category to F_i
```

E. Classification

The final step of our methodology is the identification of the effects of the propagated errors in the application output.

The output reports from the software-based propagation experiments are analyzed in search of critical effects in the CNN (i.e., misclassification). For this purpose, the output results are classified as: *Detected Unrecoverable Error* (or DUE) and *Silent Data Corruption* (or SDC) errors. DUEs are produced when the operation of the GPU is hanged and the CNN does not produce any output value or classification. SDCs are determined when at least one mismatch is found in the outputs. In detail, there are two types of SDCs: *i)* safe SDCs and *ii)* critical SDCs. In the first case (*safe SDC*), the propagation effect of the permanent hardware faults causes a mismatch in the output results of the CNN. However, the error effect is not enough to produce an error in the classification. In the second case, a *critical SDC* is identified when the propagation of the fault effects produces changes in the classification by the CNN.

IV. Implementation

In this Section we described the environment we crafted to implement the proposed methodology.

A. Environment

A new environment integrates one software profiling tool based on the *(NVbit)* mechanism, two commercial simulators (*ModelSim* and *Tetramax*), and a new software-based fault injector for Syndromes Propagation of Permanent Faults *(NVSPPF)* based on *(NVbit)* operation to implement and validate the proposed method.

A profiling tool extracts the input and output values, and the parallel operational configuration parameters from all instructions in the kernels of a CNN. Then, a general controller, described in a scripting language, manages the interaction between the two simulators. The first logic framework (*ModelSim*) handles the mixed description (RT and gate-level) of FlexGripPlus [6] and it is used for the fault-free simulation of the functional units (at gate level) with the identified operands of a given instruction. The second simulator framework (*Tetramax*) performs fault injection campaigns on the functional unit (at gate level). In this framework, two external memories [25] speedup the fault simulation by linking them to the functional units and evaluating several input patterns with a single permanent hardware fault (stuck-at) at a time. The 15nm open-cell technology library [26] was used to synthesize the FlexGripPlus model for the gate-level fault injection campaigns on the functional units.

The results of the gate-level simulation provide syndrome tables after a transformation process, one per propagated hardware fault. Each entry of the table contains: *i)* the input operand values of the target instruction, *ii)* the error syndrome produced by the fault in the hardware unit, and *iii)* the mnemonic of the instruction.

Finally, *NVSPPF* handles the syndrome tables and propagate errors from specific hardware location in the GPU (based on algorithm 1). In practice, the framework applies suitable error syndromes to each instance of the target instruction in the application's source code, so propagating the effect of a permanent fault on the CNN. In this case, the framework considers the propagation of one hardware fault at a time (one syndromes' table file) per CNN inference and then generates its classification as DUE, SDC or Masked.

The framework's operation is divided into two main parts: *i)* kernel inspection and instrumentation, and *ii)* at-speed error

propagation. In the first step, the syndrome table for a fault is loaded into the GPU's global memory, so all error syndromes are available during the run-time operation of a CNN. Then, the framework identifies in each kernel the targeted instructions and modifies (off-line) its source code as stated before in section III-D. The inspection and instrumentation stage (highlighted in gray in the Algorithm 1) is issued by the host (CPU) before the kernel submission in the GPU.

The at-speed error propagation flow is performed in the GPU by executing the instrumented kernels, which can only affect targeted instructions from a specific core. Then, the final inference result is used to classify a given fault according to its failure severity impact in the CNN outcome.

B. Limitations

The proposed method owns three major limitations in the characterization of permanent fault effects on a CNN.

The first limitation relies on the fact that we employ an open-source low-level (RT- and gate-level) GPU (*FlexGripPlus*) to characterize the impact of hardware faults. Unfortunately, this model may include simplistic hardware descriptions compared to modern GPUs. However, FlexGripPlus is one of the few low-level micro-architectural models, including open and detailed descriptions of the functional units.

The dynamic operation of the scheduling controllers in a GPU, which are in charge of managing and dispatching the blocks of a program among the cores, may also limit the accuracy of our technique. The dynamic operation of the controllers complicates the effective propagation of error effects across the instructions of a CNN. Moreover, the scheduling policies adopted by such controllers in GPUs are not fully detailed and released. Several experiments confirmed that, each time the CNN is executed, the scheduler controllers dispatch the blocks of a program on the available SMs, but not always on the same cores. Thus, some instructions are not permanently assigned to fixed hardware *Lanes* and *SMs*. Our approach uses syndromes' tables (with the error effects of all instructions) to face the limitation and allow for any instruction's searching and matching process regardless the scheduler intervention. Hence, it is possible to inject and propagate accurate errors on different units without changes in the framework. Unfortunately, the addition of a complete syndromes' table also increases the simulation time of the error propagation campaigns. However, the adopted syndromes' table strategy bypasses the dynamic scheduling operation in the controllers. In fact, the syndromes' table strategy can be used as a mechanism to allow the selection of the best trade-off between accuracy and simulation time.

Finally, the cumulative error propagation effect, only present during the software-based fault propagation, is observed when two or more errors are injected into the software (using the syndromes' tables), and the error effects corrupt other instructions. Thus, corrupted values in the instructions might produce different error effects not found in the table. In this case, we face this limitation by skipping the injection of the error in the program.

V. Experimental Results

As a case study to validate the proposed permanent fault evaluation methodology we selected the six-layers LeNet CNN [27] built over the Darknet framework [28]. To perform the evaluation of the permanent fault effects on a real GPU, we used the RTX 3060Ti GPU board equipped with an NVIDIA Ampere GPU architecture. This GPU profiles the CNN and handles the software-based fault injector *(NVSPPF)* for the error propagation. We used the RT and gate-level descriptions of the FlexGripPlus GPU model and its functional units for the gate-level fault injection campaigns.

A. Software Profiling

A preliminary software profiling of the LeNet CNN provided the total instruction count revealing that the more than 65.9% of the executed instructions (11,536,325 out of 17,505,804) use either the Floating-point unit (FP32) or the Integer core (INT). In particular, around 45.88% of the total instructions use the Floating-point unit and 20.01% uses the Integer core (INT). The remaining 34.1% instructions are memory movement, control-flow, and miscellaneous instruction. Since our goal is to understand the impact of permanent faults in the computing elements, in the experiments we target the evaluation of permanent fault effects in the FPUs and INT cores. Our methodology can be directly applied to the remaining instructions.

In the complete profiling step of the framework we extract those instructions (65.9% of the total) that employ the FP32s (FADD, FMUL, and FFMA) and INT cores (IADD3, IMAD). For each of these instructions we also kept track of its input, as described in Section V-C.

B. Gate-level micro-architecture fault injection results

Every kernel in the CNN was analyzed and a fault campaign was performed for each kernel that contains at least one of the targeted instructions. 141 fault campaigns out of 160 possible gate-level fault injections on the FlexGripPlus GPU model were performed on the 32 kernels of the CNN. For each kernel we focused on the FPU and INT units and evaluated the instructions listed in the CNN profiling (FADD, FMUL, FFMA, IADD3, and IMAD). Each fault campaign was also divided in up to 25 parts to speedup, through multi-threading, the fault simulations. We ran the experiments on a server powered by an Intel Xeon CPU running at 2.5 GHz, equipped with 12 cores and 256 GB of RAM.

We injected permanent stuck-at faults in all sites (gate-level cells and flip-flops) of the gate-level model of the FP32 and INT cores (22,044 faults). To identify the inputs that activate each fault and observe the fault effect on the instruction output, we run each instruction with all the input values identified in the profiling phase for each of these fault sites. On average, we tested 1,485,126 input vectors per instruction. That is, more than 1.54×10^{10} effects of permanent faults per input vectors are evaluated in the FP32 and INT cores per instruction.

First, we evaluated the effects of the permanent faults on each atomic instruction. Table I reports the SDC fault rate.

TABLE I
FAULT RATE AND PERCENTAGE OF INPUT PATTERNS EXCITING A PERMANENT FAULT IN THE FP32 AND INT CORES.

Instruction	SDCs fault rate (%)	Input patterns exciting permanent faults (%)
FADD	9.84	7.24
FFMA	18.72	20.0
FMUL	13.82	10.42
IADD3	4.76	27.1
IMAD	8.46	10.8

This table also reports the percentage of input patterns that excites and propagates faults (on the primary outputs of a unit) for each evaluated instruction.

From the experimental results, we can observe that each evaluated instruction is affected differently by permanent hardware faults. A small percentage of permanent faults (from 9.8% to 18.7%) is propagated to the primary outputs of the FP32 unit and corrupts the result of the floating-point (FP) instructions. A surprisingly lower percentage of permanent faults in the INT core (about 4.7% to 8.4%) were activated and propagated across the unit.

The percentage of input patterns (instruction's operands from the CNN profiling) activating at least one permanent fault is small for the INT core (IADD with 27.1%, and IMAD with 10.8%). The fault rate in both instructions (IADD and IMAD) shows that input patterns recurrently activate a limited group of faults inside the core. Thus, some internal structures of the units are not activated by the input patterns. Furthermore, the percentage of patterns activating faults inside the FP32 core (FADD with 7.24% , FMUL with 10.42%, and FFMA with the 20.0%) implies that each set of patterns per instruction excites different regions of the FP core. These relatively low fault rates (see Table I) depend on the input patterns from the CNN and the operational capabilities of a functional unit.

The input patterns include two special subsets: *i)* those identical, and *ii)* the partially identical. The identical patterns share the same values of input operands and are easily removed for the evaluation. However, the second group (partially ones) include partially shared operands (i.e., only one or two operands are identical). However, these cannot be discarded, since the missing input patterns are different among them. Thus, this group of patterns is prone to activate an exclusive set of faults inside a unit. Finally, both functional units (FP32 and INT) are used to operate several instructions, so all faults cannot be excited by a limited number of instructions.

An analysis of the output results shows that about 90% of the fault effects caused the corruption of just one bit (single bit flip) in the output results of the FADD, FFMA and FMUL instructions. In this case, bits in the exponent of the result showed a higher probability of fault propagation (>25%) in comparison to those in the mantissa. Interestingly, the lower bits in the mantissa are also prone to propagate effect of a permanent fault to the primary outputs of the unit. This is probably happening since most CNN's operands are in range from -1.0 to 1.0. For the INT core, one bit in the output was mainly affected in most of the cases (75.88%). Interestingly, there is not a clear tendency of the most commonly affected locations in the outputs. In most of the cases, any of the affected bit sites of the output is located among the least significant 23 bits of the result. This suggest that the permanent fault effects in GPUs are not trivial and should be accurately studied. Our gate-level results can be employed to model error effects into higher abstraction levels, allowing a more accurate fault injection and error propagation in the applications running o GPUs.

TABLE II
MAIN FEATURES OF THE ERROR SYNDROMES GENERATED BY PERMANENT FAULTS ON THE EVALUATED INSTRUCTIONS.

Instruction	Propagated gate-level faults	Error syndromes (size of syndrome tables)	
		Min	Max
IMAD	1,563	4	913,752
IADD3	880	4	599,355
FADD	352	1	86,583
FMUL	494	23	15,741
FFMA	637	1	143,591

C. Software-based fault error propagation results

We use the observed errors from the gate-level fault campaigns to build a set of syndromes' tables (for each hardware fault and each evaluated instruction). These syndromes are the permanent fault effects that need to be propagated by those software instructions in the kernels of a CNN. In the experiments, all kernels in the CNN were evaluated on the software injection framework using the syndromes' tables.

Each syndromes' table includes only the set of error syndromes generated by a specific hardware fault. Thus, we apply and propagate in software, in the CNN's instructions, the equivalent effect of a hardware fault. Since each hardware fault has a different effect on the executed instruction, each table is composed of a different number of error syndromes. This error propagation approach allows evaluating the individual impact of each hardware fault on the CNN's instructions and contributes to identifying the hardware faults that are most likely to modify the CNN results.

Table II reports the number of propagated faults and the number of syndromes generated per instructions. Interestingly, the number of propagated faults affecting the INT instructions is about twice the number of the FP ones. From the microarchitectural results in Table I, we observed that a percentage of hardware faults caused identical error effects in the outputs of the instructions independently of the applied input operands (0.36% for FADD, 71.65% for FMUL, 31.39% for FFMA, 45.56% for IADD, and 50.41% in IMAD). Furthermore, these identical output effects caused a few error syndromes (from 1 to 23 in Table II). However, other subsets of hardware faults, produced up to 599,355 syndromes. This variation in the number of error syndromes indicates that permanent faults in a functional units (INT or FP32) can produce either identical error effects, or specific errors (syndromes) depending on the input operands of an instruction in the CNN.

For the software error propagation experiments, we evaluated three CUDA cores (SPs) inside two representative execu-

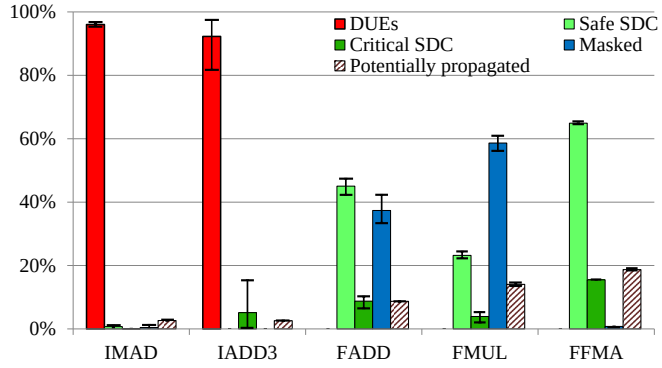

Fig. 4. Average results of the error propagation of permanent fault effects on the instructions of a CNN. Error bars show the maximum and the minimum changes in the error classification among the evaluated CUDA cores (caused by the operation of the scheduling controllers).

tion cores in the GPU (SM_0 and SM_{37}). We decided to limit the study to these two SM cores since the GPU architecture is highly homogeneous and an exhaustive evaluation of the error propagation in each CUDA core (4,864 cores in 38 SMs for Ampere) would be unfeasible due to the long simulation time to propagate error syndromes, especially for those faults producing a considerable amount of error syndromes. The two targeted SMs (0 and 37) were selected after several profiling trials on the CNN and the Ampere GPU. These trials show that the GPU's SM usage is unbalanced. In particular, SM_0 executes more threads than other SMs, whereas SM_{37} executes a minimal set of the threads. The unbalanced use of the SMs relies on two reasons: block occupancy, and block distribution, which affects the dynamic dispatching policy in the schedulers and causes the unbalanced behavior. For SM_0, we randomly choose the SP_7 and SP_8 inside PPB_1, and for SM_{37}, the SP_8 inside PPB_3. The error propagation experiments were performed on the targeted instruction (IMAD, IADD3, FADD, FFMA, and FMUL).

Figure 4 reports the average fault rate results for the evaluated instructions of the CNN in the three CUDA cores according to the proposed classification in Section III-E. Moreover, the error bars shows the maximum and minimum values obtained in the evaluated cores. Some faults are classified as 'potentially propagated', which differs from the Masked ones since, during the run-time error propagation, the input operands do not create right conditions to generate syndromes and propagate error effects. However, those faults in other cores of the GPU could potentially be activated and propagated through the CNN. The model of thread execution in the GPU elucidates this behavior since all parallel cores (CUDA cores in the SMs) work together, performing calculations related to different threads. Therefore, the thread execution model also interacts with the error propagation by distributing different threads (instructions and input operands) among the CUDA cores of an SM, which are mainly defined by a scheduling policy, so potentially missing the injection and propagation of error effects on an specific CUDA core. In any case, those 'potentially propagated' errors could generate SDC or DUE effects in the CNN when evaluated in other cores.

TABLE III
EXECUTION PERFORMANCE OF THE IMPLEMENTED METHOD FOR PERMANENT FAULT EVALUATION OF CNNS.

Step	Time (h.)
Profiling	$(0.5 - 2)$
Target Selection	$(2.7x10^{-3} - 4)$
Gate-level Microarchitectural Simulation	$(1.5 - 1,180.0)$
Software-based error propagation	$(2.1 - 207.4)$
Classification	$(0.1 - 0.5)$

The results show that INT instructions are highly vulnerable to permanent fault effects and mainly collapsed the operation of the CNN and the GPU (from 81.8% to 97.5% of faults lead to a DUE). This high sensitivity should not surprise since INT instructions are mainly used in the CNN to calculate thread identifiers and memory addresses. Thus, errors in these instructions are likely to produce failures, such as memory misalignment or illegal access, which then halt the execution of the kernel in the CNN and the GPU. This means that the implementation of a CNN (for a GPU) plays an important role in terms of CNN's vulnerability to permanent fault effects and its criticality to internal cores, such as it was observed for the instructions in the INT cores.

Most propagated faults on the FP instructions (82.4% in FADD, 81.8% in FMUL, and 65.7% in FFMA) caused minimal changes (Safe SDC or Masked) in the output of the CNN. However, some errors (from 2.0% to 15.5%) can jeopardize the application and change the output classification of the CNN. Since the FP instructions mainly process inputs, weights, and bias values from a CNN, most errors directly affect the CNN's classification. These experimental results demonstrates that permanent faults in a FPU can produce important error effects in the classification of a CNN (more than 15%), so these faults can be more critical, in comparison with other error effects, such as those produced by single bit-flip transient faults [23].

Interestingly, the dynamic dispatching policy in the scheduling controller seems to affect the error propagation results for the three evaluated CUDA cores in two SMs in the GPU (see error bars in Figure 4). These results indicate that the scheduling policy in the controllers can affect (or benefit) the propagation of permanent hardware effects from a functional units to a CNN. The results in most used (and the most affected by faults) core in the GPU (SM_0) provided higher percentage of error effects (SDCs and DUEs), in comparison to the core less used (SM_{37}), which takes advantage of the dispatching policy and propagates a lower percentage of errors to the CNN.

Finally, Table III reports the performance of the method. Two factors determine the time costs in the gate-level fault simulations (up to 1,180 hours): *i)* the number of faults (determined by the gate-level structure of the unit and the used technology library), and *ii)* the number of input patterns (directly connected with the CNN and the number of processed instructions). Furthermore, the software error propagation time (about 207 hours) depends on: *i)* the number of syndromes per table (generated during the gate-level evaluation and dependent on the number of input patterns), and *ii)* the number of

syndrome's tables (strictly related to the number of faults activated by a given input pattern in a unit during the gate-level evaluation). Since, the software error propagation step requires time for searching errors in the tables, the larger the number of syndromes per table, the longer its simulation time. Thus, the proposed method can be easily adjusted (e.g., by modifying the number of patterns per faults, and the size of the syndrome table) to trade-off accuracy with computational effort, so allowing to scale even to more complex CNNs.

Despite the long simulation times, our multi-level method of permanent faults evaluation can reduce by several orders of magnitude in comparison to fully microarchitectural approaches the time required to evaluate dense applications, such as CNNs. Our method maintains the same accuracy of the microarchitectural fault analysis at the instruction level by propagating error effects at the output of each operation.

VI. Conclusions and Future Work

This work introduced a methodology to evaluate the impact of permanent faults in CNNs running on GPUs. This method is based on the multi-level concept and combines the accuracy of gate-level microarchitectural simulation with the speed of software fault injection to characterize the effects of permanent faults in CNN applications.

Thanks to the efficient combination of gate-level and software fault injection, this is the first work to quantitatively assess the impact of permanent faults in the functional units of a GPU on the operations performed by a CNN. The proposed method can reduce by several orders of magnitude the required computational time in comparison with fully microarchitectural evaluations. Our results show that the CNN's implementation for GPUs plays an important role in the vulnerability to permanent faults affecting the functional units. Finally, results demonstrate that permanent fault effects in functional units can be critical and affect the execution of a CNN in up to 15.5% of the cases and collapse the operation of the device in up to 97.5% of the cases.

As future works, we plan to evaluate the effect of different types of permanent faults on functional units (e.g., considering delay faults as well) and in other units of the GPU. We are also working at defining other solutions able to provide different trade-off in terms of accuracy and speed, thus matching the different requirements in terms of reliability existing in the different design steps.

References

[1] F. F. d. Santos *et al.*, "Analyzing and increasing the reliability of convolutional neural networks on gpus," *IEEE Transactions on Reliability*, vol. 68, no. 2, pp. 663–677, 2019.

[2] B. Fang *et al.*, "GPU-Qin: A methodology for evaluating the error resilience of GPGPU applications," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014, pp. 221–230.

[3] T. Tsai *et al.*, "Nvbitfi: Dynamic fault injection for gpus," in *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2021, pp. 284–291.

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

[5] L. Yang *et al.*, "Practical resilience analysis of gpgpu applications in the presence of single-and multi-bit faults," *IEEE Transactions on Computers*, vol. 70, no. 1, pp. 30–44, 2021.

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

[7] O. Villa *et al.*, "Nvbit: A dynamic binary instrumentation framework for nvidia gpus," in *52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52, 2019, p. 372–383.

[8] M. Kooli *et al.*, "Cache- and register-aware system reliability evaluation based on data lifetime analysis," in *2016 IEEE 34th VLSI Test Symposium (VTS)*, 2016, pp. 1–6.

[9] M. B. Sullivan *et al.*, "Characterizing and mitigating soft errors in gpu dram," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, p. 641–653.

[10] N. Foutris *et al.*, "Assessing the impact of hard faults in performance components of modern microprocessors," in *2013 IEEE 31st International Conference on Computer Design (ICCD)*, 2013, pp. 207–214.

[11] M.-L. Li *et al.*, "Accurate microarchitecture-level fault modeling for studying hardware faults," in *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, 2009, pp. 105–116.

[12] B. Salami *et al.*, "On the resilience of rtl nn accelerators: Fault characterization and mitigation," in *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2018, pp. 322–329.

[13] J. J. Zhang *et al.*, "Analyzing and mitigating the impact of permanent faults on a systolic array based neural network accelerator," in *2018 IEEE 36th VLSI Test Symposium (VTS)*, 2018, pp. 1–6.

[14] D. A. Santos *et al.*, "Reliability analysis of a fault-tolerant risc-v system-on-chip," *Microelectronics Reliability*, vol. 125, p. 114346, 2021.

[15] D. Dutey *et al.*, "Prevention and detection methods of systematic failures in the implementation of soc safety mechanisms not covered by regular functional tests," in *2021 24th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, 2021, pp. 87–92.

[16] 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)*, 2016, pp. 90–100.

[17] A. Vallero *et al.*, "Sifi: Amd southern islands gpu microarchitectural level fault injector," in *IEEE 23rd International Symposium on On-Line Testing and Robust System Design (IOLTS)*, 2017, pp. 138–144.

[18] A. Ejlali *et al.*, "A hybrid fault injection approach based on simulation and emulation co-operation," in *International Conference on Dependable Systems and Networks*, 2003, pp. 479–488.

[19] J. Wei *et al.*, "Quantifying the accuracy of high-level fault injection techniques for hardware faults," in *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2014, pp. 375–382.

[20] A. L. Sartor *et al.*, "A fast and accurate hybrid fault injection platform for transient and permanent faults," *Design Automation for Embedded Systems*, vol. 23, no. 1–2, p. 3–19, jun 2019.

[21] A. Ruospo *et al.*, "A pipelined multi-level fault injector for deep neural networks," in *2020 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, 2020, pp. 1–6.

[22] B. Fang *et al.*, "A systematic methodology for evaluating the error resilience of gpgpu applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 12, pp. 3397–3411, 2016.

[23] F. F. d. Santos *et al.*, "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)*, 2021, pp. 292–304.

[24] R. Velazco *et al.*, "Combining results of accelerated radiation tests and fault injections to predict the error rate of an application implemented in sram-based fpgas," *IEEE Transactions on Nuclear Science*, vol. 57, no. 6, pp. 3500–3505, 2010.

[25] J. P. Acle *et al.*, "About performance faults in microprocessor core in-field testing," in *2019 IEEE 10th Latin American Symposium on Circuits Systems (LASCAS)*, 2019, pp. 229–232.

[26] M. Martins *et al.*, "Open cell library in 15nm freepdk technology," in *Proceedings of the 2015 Symposium on International Symposium on Physical Design*, ser. ISPD '15, 2015, p. 171–178.

[27] Y. LeCun *et al.*, "Lenet-5, convolutional neural networks," *URL: http://yann. lecun. com/exdb/lenet*, vol. 20, no. 5, 2015.

[28] J. Redmon, "Darknet: Open source neural networks in c," http://pjreddie.com/darknet/, 2013–2016.