Effective Fault Effects Evaluation for Permanent Faults in GPUs executing DNNs

Effective Fault Effects Evaluation for Permanent Faults in GPUs executing DNNs

JUAN DAVID GUERRERO BALAGUERA, Control and Computer Engineering (DAUIN), Politecnico di Torino, Torino, Italy
JOSIE ESTEBAN RODRIGUEZ CONDIA, Control and Computer Engineering (DAUIN), Politecnico di Torino, Torino, Italy
MATTEO SONZA REORDA, Control and Computer Engineering (DAUIN), Politecnico di Torino, Torino, Italy

Deep Neural Networks (DNNs) have permeated multiple applications, including cutting-edge safety-critical domains, which require relevant computational power, often provided by Graphic Processing Units (GPUs). GPUs are manufactured with advanced semiconductor technologies that can be affected by faults during the operational phase (e.g., due to wear-out, aging, or environmental harshness), whose effects possibly reach the DNN outputs, in some cases leading to catastrophic consequences. Hence, hardware-aware reliability assessments of DNNs are crucial to be considered in the context of safety-critical systems (following regulations/standards of specific application domains). Application-level fault injection (FI) techniques (i.e., DNN parameter corruption) are often adopted for the reliability evaluation of DNNs; unfortunately, these approaches hardly represent fault effects from GPU hardware. This work proposes an FI strategy based on Hardware-Injection-Through-Program-Transformation (HITPT) to mimic the effect of permanent faults (PFs) at the GPU instruction level, enabling effective assessment of PFs on DNN's reliability. Our approach provides a good trade-off between the fault effect evaluation's accuracy and the required computational time. Using the proposed approach, for the first time, we systematically assessed the effects of PF in GPUs executing some DNN sample cases. The results indicate that the faults injected closer to the hardware, using our evaluation strategy, can produce a higher accuracy degradation than the evaluations performed by the typical application-level FI that modify only the DNN parameters. Furthermore, the proposed FI methodology provides insightful results to identify the most suitable fault-tolerance solutions (e.g., selective hardening or design diversity) for their application at thread levels inside GPU's kernels.

CCS Concepts: • **Computer systems organization** → **Reliability**; *Multiple instruction, multiple data*; • **Computing methodologies** → **Neural networks;**

Additional Key Words and Phrases: Artificial neural networks, deep neural networks (DNNs), graphic processing units (GPUs), fault simulations, reliability evaluation, permanent faults

This work extends and unifies preliminary publications in [22, 23].

Authors' Contact Information: Juan David Guerrero Balaguera, Control and Computer Engineering (DAUIN), Politecnico di Torino, Torino, Piemonte, Italy; e-mail: juan.guerrero@polito.it; Josie Esteban Rodriguez Condia, Control and Computer Engineering (DAUIN), Politecnico di Torino, Torino, Piemonte, Italy; e-mail: josie.rodriguez@polito.it; Matteo Sonza Reorda, Control and Computer Engineering (DAUIN), Politecnico di Torino, Torino, Piemonte, Italy; e-mail: matteo.sonzareorda@polito.it.

ACM 1084-4309/2025/02-ART33
https://doi.org/10.1145/3715327

ACM Reference Format:
Juan David Guerrero Balaguera, Josie Esteban Rodriguez Condia, and Matteo Sonza Reorda. 2025. Effective Fault Effects Evaluation for Permanent Faults in GPUs executing DNNs. *ACM Trans. Des. Autom. Electron. Syst.* 30, 2, Article 33 (February 2025), 33 pages. https://doi.org/10.1145/3715327

1 Introduction

Current advances in **Deep Neural Networks (DNNs)** technologies allow them to exhibit noticeable accuracy capabilities, attaining human-level performance for solving complex tasks, e.g., for image and speech recognition, natural language processing, and decision-making [34]. These outstanding capabilities of DNNs have permeated a broad spectrum of application domains, including self-driving vehicles [26, 34, 37], **unmanned aerial vehicles (UAVs)** [33], and medical equipment [26, 59]. However, the remarkable accuracy and performance of the DNNs in solving challenging problems also demand high computational capabilities, which can only be granted by specialized hardware accelerators.

Today, a wide variety of hardware accelerators are available for speeding up DNN execution, including **Tensor Processing Units (TPUs)**, **Field Programmable Gate Arrays (FPGAs)**, and **Graphics Processing Units (GPUs)** [8, 26, 35]. Despite the existence of all these options, in many cases, GPUs have become the preferred choice for **artificial intelligence (AI)** applications due to their outstanding performance and programming flexibility [36, 52]. GPUs possess specialized programming tools that facilitate the deployment of DNNs and optimize the device's performance [3, 16].

The widespread use of DNNs in safety-critical domains is, unfortunately, being partly overshadowed by several reliability concerns, hardware failures being a major concern [5, 54]. Indeed, several studies have demonstrated that GPU devices can be highly susceptible to transient faults that impact the operation of DNNs [47–49]. Furthermore, GPUs are now widely used in applications that require long lifespans (e.g., automotive, aerospace, or military) and often operate under harsh conditions with extreme temperatures [21]. These conditions, combined with semiconductor miniaturization (e.g., 5nm for the NVIDIA GPU Hoper architecture), exacerbate the degradation of the device due to possible test escapes, aging, premature wear-out, and even terrestrial radiation phenomena [21, 24], leading to an increased occurrence of permanent faults [28]. Recent studies have demonstrated that hardware faults in AI accelerators are often challenging to detect and counteract since they do not always produce visible effects, such as a complete system crash. Instead, they may silently propagate through the application execution as **Silent Data Errors (SDEs)** [4, 20, 53]. These reliability concerns can apply to GPU devices, which are pervasive in multiple application domains nowadays. Moreover, over-stressing specific units within a GPU (e.g., the register files or the logic arithmetic resources) can significantly increase the likelihood of permanent fault occurrence and negatively impact the system's overall reliability. In addition, permanent hardware faults may drastically reduce the effectiveness of error protection mechanisms (e.g., ECCs), increasing the device's vulnerability even during its operational life [31]. Finally, permanent faults cannot be considered stemming from manufacturing defects only; they may also arise during the in-field operational phase, seriously threatening the dependability of modern applications, including DNN-based systems.

In this regard, most safety standards (i.e., ISO 26262) require identifying faults that can cause critical failures and demand more precise reliability evaluations, which are essential for creating effective hardening solutions. Moreover, heightened attention to the reliability of DNNs in safety-critical contexts has intensified efforts to accurately assess the impact of permanent faults. Unfortunately, identifying GPU permanent faults that can produce critical failures in DNNs, depending

on the application, remains challenging using current methods and tools. Currently, predominant fault evaluation methods for DNNs rely on **Fault Injection (FI)** at the application level. However, these FI approaches are based on agnostic fault models that target *synapses* (i.e., using dropout, stuck-at, or single bit-flip models in DNNs' weights [9, 32, 38]), or *neurons* (i.e., using dropout or Byzantine neuron fault models in DNN's feature maps [32, 43]). These FI approaches overlook the low-level hardware details, making it difficult to estimate the real impact of permanent faults affecting any GPU component and their effects on the DNN's performance. Thus, adopting FI methodologies supporting hardware-aware fault models is crucial for obtaining more realistic reliability evaluations.

Gate- or RT-level fault simulations offer high accuracy but become impractical due to the massive size of GPU accelerators (billions of gates) and the complexity of DNN architectures (tens of layers and hundreds of millions or billions of parameters), resulting in prohibitively long simulation times [11, 40]. For example, simulating a thousand faults on an RT-level GPU model running a small DNN (e.g., LeNet) would exceed 10,000 days. Therefore, exploring alternative solutions, such as different abstraction levels, is necessary to balance simulation time and fault evaluation accuracy [11]. Alternatively, the **Software Implemented Fault Injection (SWIFI)** strategies model hardware faults in the software, propagating them through code execution at device speed. SWIFI induces software errors during application execution or by modifying source code beforehand [7]. Recently, it has been adapted for GPUs using the **Hardware Injection Through Program Transformation (HITPT)** method. HITPT inserts corruption routines at the GPU's instruction level by instrumenting GPU kernels before execution, offering precise control by modeling faults on specific GPU structures like registers and core units using software-level hardware identifiers.

Recent works have successfully conducted reliability evaluations of DNNs concerning transient faults on GPU devices using the HITPT approach [27, 47–49]. However, this FI approach has yet to be fully explored to evaluate the effects of *permanent faults*, making it an attractive strategy to model and evaluate the impact of these faults on DNNs.

Despite the significant benefits of HITPT, the practical support for analyzing permanent faults is still challenging since, in this case, the fault (inside a given hardware structure) must persist during the entire application's execution. In this regard, former works have attempted to propose strategies to effectively adopt HITPT and enable the evaluation of permanent faults in GPUs executing DNNs. The authors in [11] proposed evaluating stuck-at faults in GPUs and their impact on DNN workloads, combining gate-level FI simulations with HITPT. Nevertheless, their approach still demands a significant computational effort, leading to long fault evaluation times ($\approx$ 1,180 hours for only one tiny DNN model). On the other hand, authors in [22] and [23] propose a HITPT mechanism for assessing the resilience of DNNs concerning permanent faults on register files and functional units of a GPU, respectively. Their preliminary evaluations indicate that FIs require 10 to 90 seconds per fault on small DNN models. Their results suggest that DNNs are highly sensitive to faults affecting the first set of registers per thread [22] and the **Fused-Multiply-Add (FMA)** units [23]. Unfortunately, their evaluations are restricted to a few small DNN models for image classification on just one GPU device. However, DNN implementations can vary across GPU systems, leading to different fault behaviors at both the layer and overall execution levels. Thus, additional evaluations are necessary, encompassing diverse DNN applications and detailed characterization of DNN layers to address permanent faults that could correspond to errors at the application level.

In this work, we unified and extended the capabilities of our prior works [22, 23] regarding the fault injection strategies based on HITPT to evaluate the resilience of DNN models against permanent faults in several hardware structures of GPUs. This paper presents some general mechanisms to model permanent faults to **general-purpose register files (GPRFs)**, **scalar processors (SPs)**, and **special function units (SFUs)** and extends the modeling of permanent faults on **predicate**

registers (PRs) and **Tensor Core Units (TCUs)** in GPU devices. Our approach offers several advantages over existing techniques for assessing DNN reliability: *i*) it employs a more realistic fault model by injecting and propagating permanent faults at the device instruction level, as opposed to application-level fault injections targeting weights or intermediate activations; *ii*) it enables faster evaluations compared to simulation-based methods; *iii*) it provides a comprehensive solution for identifying sensitive hardware structures during full DNN workload execution; and *iv*) it facilitates the characterization of individual layers to create more realistic fault/error models at the application level.

We applied the proposed fault injection technique to assess the effect of over 160,000 permanent faults in the GPRFs, PRs, SPs, SFUs, and TCUs of a GPU across 10 different DNNs. The first set of seven models corresponds to image classification applications (namely LeNet5, AlexNet, DarkNet19, VGG-16, MobileNetv3, ResNet50, and ViT); the second group comprises three variations of Yolo-v3 for object detection. Additionally, we injected more than 240,000 permanent faults into individual convolutional layers to analyze their propagation effects across various GPU devices, including a Jetson Nano board and an Ampere GPU architecture. The experimental evaluation required approximately 500 hours of computational time to complete.

The following are the major contributions of this work:

- We introduce a unified Fault Injection strategy utilizing HITPT to evaluate the impact of **permanent faults (PFs)** affecting GPU devices, evaluating their effects on DNN workloads. Our FI technique is designed to target PFs within various hardware components of a GPU, including GPRFs, PRs, SPs, SFUs, and TCUs.
- We developed a prototypical fault injection tool based on the binary instrumentation tool NvBit.[1] In addition, we conceived a fault injection flow and framework that facilitates the evaluation of the impact of permanent faults on different DNN workloads[2] (i.e., full DNN inference or individual DNN layers characterizations).
- Employing the proposed solutions and tools, we assessed the fault sensitivity of the DNN at both the register and bit-wise levels. The results show that faults affecting the first 10 registers per thread lead to a notable degradation in DNN accuracy, exceeding 68%. Furthermore, we observed that faults affecting the most significant bits of the GPRFs can result in a degradation of up to 70%. On the other hand, faults in the PRs mainly cause the GPU to crash due to memory access violations; in only a few cases, some faults ($< 10\%$) were able to affect the performance of the DNNs significantly.
- We evaluated at the operation and bit-wise levels the DNN's fault sensitivity. The evaluation results show that the DNNs are particularly sensitive to permanent faults affecting the **Integer-Multiply-ADD (IMAD)**, **Floating-Point-Fused-Multiply-ADD (FFMA)**, and the TCUs, producing a DNN accuracy degradation of 80%, 20%, and 20%, respectively.
- We demonstrated the existence of notable differences in fault rates and DNN accuracy degradation between conventional application-level fault injections and our proposed approach, which models permanent faults closer to real hardware.
- We evaluated the effects of permanent faults on individual DNN layers executed on different GPU devices, providing an overall analysis of the sensitivity of several layers.

The paper is structured as follows: Section 2 outlines the motivations for our work and overviews the previous related works. Section 3 covers essential background information. In Section 4, we detail the proposed fault injection methodology. The experimental setup is described

[1] https://github.com/divadnauj-GB/nvbitPERfi/tree/Dev
[2] https://github.com/divadnauj-GB/pytorch-DNNs

in Section 5. Section 6 presents the experimental results and discusses the evaluation of DNN reliability using both application and architectural fault injection levels. Finally, Section 7 concludes the paper and outlines future research directions.

2 Motivations and Related Work

Fault Injection (FI) is a widely accepted approach used to assess the resilience of electronic systems regarding hardware faults. However, the complexity of modern hardware devices (such as Graphic Processing Units or GPUs) and the sophistication of DNNs limit the effectiveness of the FIs' evaluations, exacerbating the evaluation time or making evaluations less realistic. Thus, it is crucial to adopt FI approaches that can handle the hardware and software complexity of DNNs while accurately evaluating GPU hardware faults within a reasonable experimental time frame.

Nowadays, the resilience evaluation of DNNs w.r.t. hardware faults mainly relies on application-level FI approaches based on corrupting either the parameters of the DNN (i.e., weights or bias) or its intermediate feature maps. In fact, there exist multiple available frameworks that can perform such assessments (e.g., *PytorchFi* [32], *ARES* [38], and *TensorFI* [9] among others). Although this is the most popular approach for fast and cheap reliability evaluations of DNNs, there are still concerns about its capabilities to effectively model faults in the underlying hardware (i.e., GPU devices).

Indeed, when conducting fault injection (FI) in DNN parameters (i.e., *synapse corruption*), some studies suggest that stuck-at or bit-flips affecting the weights of the DNN could represent permanent faults impacting system memory [6, 42]. However, many of these evaluations fail to differentiate between detailed system implementations, treating CPU-based and GPU-based systems similarly. Moreover, when applied to GPU-based systems, such error models overlook insights into GPUs and their internal memory units (such as shared memory, constant memory, and register files), which can also play a critical role in the DNN's reliability.

On the other hand, some studies employ *neuron-level* fault injection (FI) strategies to model faults in the data-path units of a specific accelerator by corrupting the output feature maps of the DNN's layers [1, 51, 58]. These strategies are commonly used to assess transient or permanent fault effects on small AI accelerators. Typically, this type of evaluation involves FI campaigns where permanent faults are assumed to either completely disable a **Multiply-Accumulate (MAC)** unit (i.e., forcing MAC results to zero at the software level [1, 51]) or partially corrupt some bits of MAC unit results (i.e., injecting stuck-at/bit-flips into the feature maps of the DNN [46]).

Unfortunately, *neuron-level* fault injection (FI) fails to account for the fact that a computational core may be reused multiple times and perform various operations before propagating fault effects to the final feature map results. In GPU devices, arithmetic cores are utilized for both DNN computations (e.g., convolutions, activation functions) and application parallelism control (e.g., thread management, memory access). Therefore, modeling a random fault affecting any internal unit of a GPU (e.g., storage element, functional unit) at the application level using standard FI methodologies is challenging. Additionally, relying solely on application-level FI campaigns can overlook hidden GPU vulnerabilities. Consequently, protection strategies developed at the application level based on such evaluations may prove ineffective, potentially compromising the functionality of the DNN in the presence of actual hardware defects. Thus, it is imperative to adopt hardware-aware FI techniques that enable deeper evaluations, considering architectural or micro-architectural details, to provide more precise information about the interaction between faults, GPU hardware units, and the target DNN.

There are various FI methods that can assess the DNN reliability with respect to GPU faults, broadly categorized into *simulation-based*, *emulation-based*, *hardware-based*, and *software-implemented* [2, 5, 44]. However, not all of them are suitable for evaluating GPU faults on DNNs.

Table 1. Qualitative Comparison of Fault Injection Techniques for Assessing DNN Resilience to GPU Hardware Faults

Parameter	Simulation-based		Emulation-based	Hardware-based	Software-implemented	
	uArch-level	*Cycle-level*			*Application-level*	*HITPT*
Cost	High	Medium	Medium-High	High	Low	Low
Development effort	High	Medium	Medium-High	Low	Low-Medium	Medium-High
Accuracy	High	Low	Medium-High	Low-Medium	Low	Medium-high
Flexibility	Low	Medium	Medium	High	High	Medium
Fault injection time	High	Medium-High	Low-Medium	Low	Low-Medium	Low-Medium
Main advantages	• High Accuracy	• Flexibility	• High Accuracy • Fast	• Realistic Evaluations • Fast	• Cheap • Fast	• Cheap • Good Accuracy • Moderated Sim. time
Main drawbacks	• Expensive • Slow	• Slow • Low accuracy • Representative model required	• High development efforts • FPGA devices required • HDL sinthesizeble	• Expensive • Low fault controllability • Not for static faults	• Low Accuracy • Hardware Agnostic	• GPU HW required

Factors like cost, development effort, accuracy, flexibility, and evaluation time must be considered to determine the optimal tradeoff between accuracy and evaluation time for each method.

Table 1 presents a comparison of various fault injection approaches based on the aforementioned factors, categorized into three levels: Low, Medium, and High. These factors include **Cost**, which encompasses the total costs for reliability evaluation, such as time, resources (e.g., hardware platforms, computational equipment, specialized software), and engineering effort; **Development effort**, which assesses the complexity involved in creating and deploying the fault injection setup; **Accuracy**, which measures how closely each FI method describes the effects of real faults in the hardware; **Flexibility**, which evaluates the adaptability and portability of a given fault injection technique across different evaluation scenarios (e.g., various GPU architectures, simulation complexities, or DNN models and frameworks); and **Fault Injection time**, which denotes the time required to inject and observe the effects of each fault.

2.0.1 Simulation-based FI. uses simulation models to study the behavior of a fault while the hardware computes a given task. Nonetheless, this evaluation strategy is limited by the availability of simulation models and tools. This reliability evaluation approach has mainly two hardware abstraction levels: Micro-architectural simulation (*uArch-level*) and architectural level mainly using *cycle-level* simulation models and tools.

The first approach uses the RTL or gate-level descriptions of the hardware to simulate faults while executing small benchmarks [10, 40]. The *uArch-level* simulation strategies provide accurate results when evaluating an individual or a few units. Unfortunately, the main drawback of this approach lies in the huge simulation time required when considering neural network workloads, which might lead to unfeasible evaluation times (e.g., >10,000 days to evaluate a tiny DNN running on top of an RTL GPU's model [11]). On the other hand, this approach can be costly since it requires specialized simulation tools, extensive development expertise to set up the simulation frameworks, and high computational power to execute the experiments.

The second simulation abstraction (*cycle-level*) simulates the behavior of functional components (e.g., Adders, controllers, multipliers, or registers) inside the system. In this case, faults can be only simulated at the outputs of such blocks, or in the interconnections between blocks [50, 57]. Although this fault simulation approach can provide acceptable fault evaluation results, it requires a non-negligible effort to model the faults and perform the simulations since it requires to development of custom simulation tools that are not always freely available. Moreover, the accuracy of the evaluation is limited by the fault model used at these levels and the simulation time requires a significant amount of time in function of the target workload.

2.0.2 Emulation-based FI. reduces the evaluation time using FPGAs to implement the target circuits of the DNN with a corruption mechanism either inserting saboteurs inside the device's

RTL model or corrupting the FPGA bit-stream [19]. Although the evaluations at this level provide accurate evaluations close to reality, this FI technique requires synthesizable GPU models in HDLs, costly FPGA devices or clusters of FPGAs, and non-negligible development time.

2.0.3 Hardware-based FI. (also known as *physical* FI) induces faults in real hardware platforms while executing a DNN application. This approach is mainly used to assess the impact of transient fault effects in the form of **Single Events Upsets (SEUs)** [48, 49]. This strategy requires special facilities with radiation equipment capable of performing particle strikes into the device. Unfortunately, such evaluations are expensive and limited to transient fault effects, leaving static fault models out of scope.

2.0.4 Application-level FI. is the preferred approach to assess the reliability of DNNs by corrupting the weights or the feature maps of the model. In recent works, this fault evaluation strategy has become popular to assess the reliability of such applications with respect to hardware faults[41, 42, 45]. However, there are serious concerns about the level of realism this fault evaluation level can provide due to its hardware-agnostic features since CPUs, GPUs, or any accelerators are indistinguishable at this level [5, 54].

2.0.5 ***Hardware Injection Through Program Transformation (HITPT)*** *FI.* This software-based FI strategy mimics the behavior of hardware faults by inserting saboteur routines on the original application software at the assembly level, such that the faults are activated during the program execution [56]. While HITPT is limited to injecting faults on visible hardware structures at the software level (e.g., register files, arithmetic cores, and memory resources), injecting faults at the assembly level enables more realistic evaluations for complex DNNs. This approach offers faster evaluation times compared to simulation-based methods, and its speed is on par with or slightly slower than emulation-based or application-level fault injection strategies. HITPT has primarily been utilized to assess transient fault effects on GPU workloads, including DNNs, using instrumentation tools like SASSFI [25] and NvBitFI [56]. However, the resilience of DNN workloads concerning GPU's permanent faults has still not been fully explored through this method.

It is essential to mention that recent advancements have explored cross-layer FI approaches combining RT/gate-level simulations with software-based fault propagation using HITPT. Such evaluations combine the accuracy provided by simulation-based approaches on the target core of the GPU and the fast fault propagation of the software-based FI. Several works have successfully explored this approach [11, 21]. Indeed, in [11], the authors devise a cross-layer FI mechanism to study the resilience of Neural Networks w.r.t. permanent faults on Integer and Floating-point cores in GPU devices. Such a work used simulation-based FI to create syndrome tables, which were later used by a software-based approach to propagate the errors at the ISA level. Although this method significantly reduces the evaluation time compared to simulation-based strategies, it still requires a significant amount of time, (around 1,180 hours), to evaluate a tiny DNN model. In this regard, the fault injection method we propose works at the GPU's instruction level and effectively evaluates the reliability of DNN workloads concerning permanent faults on several hardware structures, such as register files, functional units, and tensor core units. Moreover, our proposed method enables fast evaluations that allow managing even large DNN models.

3 Background

3.1 Graphics Processing Units

Graphics Processing Units (GPUs) are hardware accelerators specially designed to provide a high throughput during the execution of high-performance applications, such as machine learning using DNNs. Modern GPUs are composed of **Streaming Multiprocessors (SMs)** organized

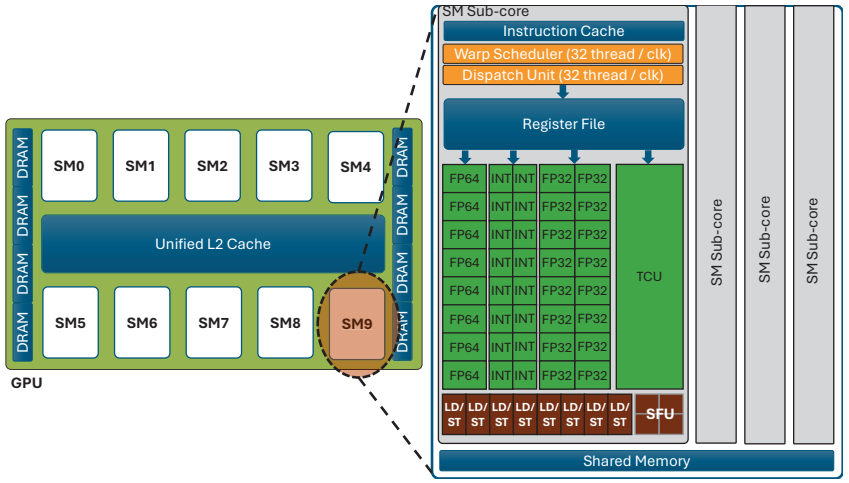


Fig. 1. GPU architecture.

hierarchically. The SM is the primary execution unit in the GPU, which encompasses several independent sub-cores (up to four for modern GPUs). Figure 1 depicts the general structure of a GPU device. Each SM sub-core comprises several parallel processing cores known as **Stream Processors (SPs)**, **Special Function Units (SFUs)**, and **Tensor Cores Units (TCUs)**. The SP supports integer (INT) and floating-point operations (FP32), the SFU executes transcendental functions, and the TCU performs parallel matrix multiplications usually employed to deploy DNNs. Typically, one SM sub-core contains up to 32 SPs, 4 SFUs, and 2 TCUs. Additionally, an SM includes load/store units (LD/ST) to access local memories and register file banks, supporting the parallel execution of several threads [10, 14].

The GPU device computes a parallel program called *kernel* that comprises multiple parallel *threads* distributed among the available resources of the device. The threads in a kernel are organized in groups called *Thread-Blocks*. The block scheduler assigns to each *SM* several blocks scheduled in a queue, maximizing the occupancy and the GPU's performance. When one Thread-Block releases the SM resources, another Thread-Block initiates the operation. The SM core in a GPU adopts the **Single-Instruction Multiple-Tread (SIMT)** to schedule and execute on Thread-Block in smaller groups of threads called *Warps* (i.e., one SIMT group of 32 threads) [14].

3.2 Deep Neural Networks

A Deep Neural Network is an Artificial Neural Network composed of many computational layers that process a given input and produce an output prediction. A DNN comprises an input layer, several hidden layers, and an output layer. **Convolutional Neural Networks (CNNs)** are DNNs whose hidden layers perform convolution operations between filter elements and input data. The inputs typically correspond to a two-dimensional matrix (i.e., an image). The filter is also a two-dimensional matrix but smaller in size. There are weight filters and bias filters. The weight filter element is multiplied by the input node, and the bias filter element is added. Pooling operations are performed to reduce the dimensions of the output. Additionally, each layer of the DNN contains a non-linear activation function to limit the output value of neurons. Finally, the output layers of the DNN employ a fully connected layer to perform the classification task [18] [55]. Well-known DNN architectures include LeNet, AlexNet, ResNet, and DenseNet.

Nowadays, multiple frameworks exist for developing and deploying DNNs using GPU acceleration, such as Darknet, Pytorch, and TensorFlow. Darknet corresponds to a C-language DNN

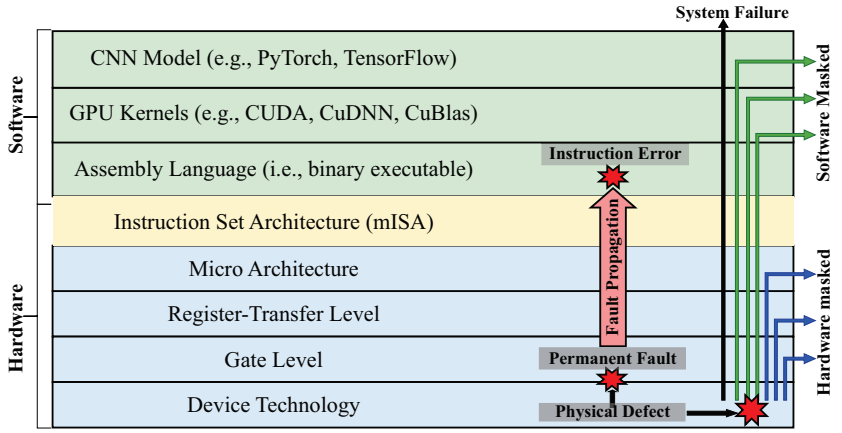


Fig. 2. System abstraction layers and fault propagation effects from the GPU device to the DNN execution.

implementation that uses Cublas libraries to run on GPU devices [39], while PyTorch and Tensorflow are Python-based frameworks that provide flexible development and deployment of DNNs on GPU devices using cuDNN libraries [16].

3.3 Fault Propagation through the GPU's System Abstraction Layers

Contemporary electronic systems, like GPUs, consist of layered hardware and multiple software abstraction levels. Particularly, the hardware layers physically execute the functionalities outlined in an **Instruction Set Architecture (ISA)**. This execution is dictated by the microarchitecture, which delineates the components of the device from a high-level to a low-level perspective. Meanwhile, on the software front, applications utilize assembly language to enact algorithms and interface with the hardware in accordance with the specifications of the ISA [10, 14]. Indeed, the GPU acceleration of DNNs involves several software layers between the ISA and the application level (refer to Figure 2). The DNN uses high-level operations that describe layer computations (e.g., convolutions) at the application level. Every layer operator wraps libraries that provide GPU kernels, maximizing the performance of a DNN computation (e.g., Cublas, Cutlas, or CuDNN). Moreover, several GPU kernel functions may be used by the same type of layer, considering the size of the layer, the GPU architecture, and the computation algorithms that leverage the maximum performance of the GPU device.

Table 2 showcases example kernels utilized across various GPU devices for convolutional layers sourced from different DNN models in PyTorch. The GPU kernels employed in DNNs exhibit diversity both across different GPU devices and within layers on the same device. Thus, a hardware-level defect inside the GPU may result in faults at higher abstraction levels, potentially going unnoticed and causing a nonobservable impact. If a fault arises, it could spread through the hardware structures, affecting the functionality of software layers and leading to significant system malfunctions. While certain fault effects may be suppressed within various system layers, others may penetrate the software layer, resulting in system failures [29]. Consequently, only those faults that reach the software level can seriously affect the application's operation. In essence, the intricate software architecture of DNNs dictates the propagation effects of those faults that become visible at the GPU instruction level. Therefore, adopting fault injection mechanisms targeting DNN software layers at the assembly level, such as using HITPT, enables a more realistic evaluation of hardware faults compared to injections at the application level, where hardware faults are challenging to represent.

Table 2. GPU Kernel Execution for the First Five PyTorch Convolutional Layers of Several DNN Architectures (LeNet5 Only has Two)

DNN	Layer	RTX 3060TI	Jetson Nano
Lenet	Conv1	fusedConvolutionReluKernel	trt_maxwell_scudnn_128x32_relu_interior_nn_v0
	Conv2	fusedConvolutionReluKernel	trt_maxwell_scudnn_128x32_relu_small_nn_v0
AlexNet	Conv1	voidimplicit_convolve_sgemm voidop_generic_tensor_kernel	trt_maxwell_scudnn_128x64_relu_large_nn_v0
	Conv2	fusedConvolutionReluKernel	trt_maxwell_scudnn_128x64_relu_small_nn_v1
	Conv3	fusedConvolutionReluKernel	trt_maxwell_scudnn_winograd_128x128
	Conv4	fusedConvolutionReluKernel	trt_maxwell_scudnn_winograd_128x128
	Conv5	fusedConvolutionReluKernel	trt_maxwell_scudnn_winograd_128x128
MobileNetv3	Conv1	voidimplicit_convolve_sgemm	trt_maxwell_scudnn_128x32_relu_small_nn_v0
	Conv2	voidcudnn::cnn::conv2d_grouped_direct_kernel	voidcuDepthwise::depthwiseConvFP32Kernel
	Conv3	voidimplicit_convolve_sgemm	trt_maxwell_scudnn_128x32_relu_interior_nn_v0
	Conv4	voidcask_trt::computeOffsetsKernel trt_ampere_scudnn_128x32_relu_interior_nn_v1	trt_maxwell_scudnn_128x64_relu_interior_nn_v1
	Conv5	voidcudnn::cnn::conv2d_grouped_direct_kernel	voidcuDepthwise::depthwiseConvFP32Kernel
ResNet50	Conv1	voidcask_trt::computeOffsetsKernel trt_ampere_scudnn_128x64_relu_xregs_large_nn_v1	trt_maxwell_scudnn_128x64_relu_medium_nn_v1
	Conv2	sm80_xmma_fprop_implicit_gemm_f32f32_f32f32_f32_	trt_maxwell_scudnn_128x64_relu_interior_nn_v1
	Conv3	trt_ampere_scudnn_winograd_128x128_ldg1_ldg4_relu_tile148t_nt_v1	trt_maxwell_scudnn_winograd_128x128
	Conv4	voidcask_trt::computeOffsetsKernel trt_ampere_scudnn_128x64_relu_interior_nn_v1	trt_maxwell_scudnn_128x64_relu_interior_nn_v1
	Conv5	voidcask_trt::computeOffsetsKernel trt_ampere_scudnn_128x64_relu_interior_nn_v2	trt_maxwell_scudnn_128x64_relu_interior_nn_v1

4 Software-based Fault Injection Methodology

This work introduces a method to assess DNN resilience with respect to permanent faults in GPUs. Although this work focuses on DNN workloads only, the method can also be extended to other GPU-based applications (e.g., LLMs used in safety-critical systems or linear algebra applications). Our strategy can assess the impact of permanent faults on either the full DNN inference or individual layers.The proposed method employs the HITPT technique to mimic the effect of hardware faults, within the GPU, using injection procedures at the instruction level. It focuses on efficient fault injection mechanisms for modeling permanent faults on different hardware structures inside the SMs of GPUs. More in detail, we develop a hardware-aware instruction level fault injection model for emulating permanent faults on the *General Purpose Register Files* (GPRFs), *Predicate Registers* (PR), *Scalar Processors* (SPs), *Special Function Units* (SFUs), and *Tensor Core Units* (TCUs) in GPUs.

4.1 Fault Injection Flow for Reliability Evaluation of DNNs

We devised a fault injection flow that incorporates each fault injection mechanism specifically devised for each type of unit (i.e., GPRFs, PRs, SPs, SFUs, and TCUs) to evaluate the resilience of DNN workloads concerning PFs on GPU devices. The flow allows to assess the impact of PFs either in **full DNN** models or in **individual layers** extracted from a DNN. In the first case, the evaluations permit quantifying the impact of faults on the performance of the DNN (e.g., the accuracy of the model). The second evaluation type seeks to investigate the fault propagation effects when considering different GPU devices and kernel implementations for basic DNN operations, as introduced in Table 2. This last evaluation can be used to effectively produce layer-level error models with respect to permanent faults to enable the hardware-aware evaluation of faults in large DNN workloads resorting to application-level error injections.

Our proposed fault injection flow comprises four main stages as depicted in Figure 3: ❶ DNN model setup, ❷ fault list generation, ❸ fault injection campaign, and ❹ fault impact evaluation.

In the first stage (model setup), we select and configure the DNN model according to the validation dataset, the number of images to be evaluated, the DNN framework for GPU implementations, and the model type. The model type defines the evaluation of either *(i)* a full inference of the DNN

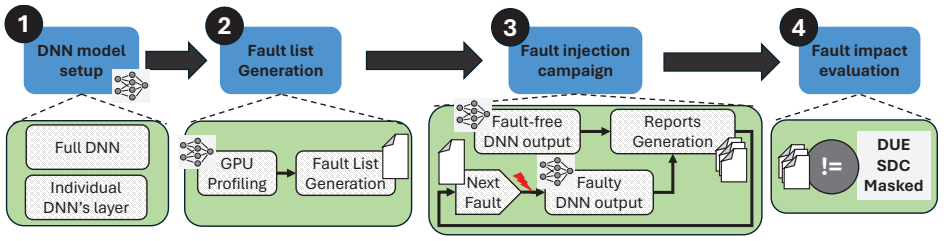


Fig. 3. Fault injection flow for reliability evaluation of DNNs.

or *(ii)* the execution of a single layer. When evaluating single layers, we devise a DNN instrumentation mechanism that creates a custom dataset and extracts the target DNN layer from the original model. The dataset is generated by performing a full inference of the DNN model on the initial dataset (e.g., ImageNet). At the same time, the inputs and output tensors of the target layer within the DNN are captured, compressed, and stored. The layer extraction isolates the target layer from the original DNN model, creating a new single-layer model with the same parameters and configurations as the original one. Then, during the evaluation process, the individual layer processes the custom dataset, storing the outputs for later evaluations.

In the second stage (fault list generation), we used a similar philosophy of the HITPT approach introduced by [56] that comprises two steps: *(i)* DNN profiling and *(ii)* faults list generation. The profiling gathers all the information about the DNN execution on the GPU device, such as disassembled kernels, number of Threads per Kernel, and registers per Kernel. The extracted information during the profiling is used later by the fault list generation process to create a set of fault descriptors that indicate the target hardware structure (i.e., GPRFs, PRs, SPs, SFUs, or TCUs), the type of fault (i.e., stuck-at 0/1), among other additional parameters associated to each target structure. In fact, these additional parameters are specifically generated to describe the persistent effects of faults and may change according to the target hardware structure during the fault injections. In the following subsections, we provide more specific details about the addition fault descriptors and how they differ from the state of the art for transient faults introduced in [56].

The third stage (fault injection campaign) first executes the DNN workload in a fault-free scenario. Then, it takes the list of faults and orchestrates an entire fault injection campaign by executing the DNN workload with one fault at a time. During the fault injection, the following reports are generated: *(i)* a fault injection profiling report, *(ii)* an output data corruption report, and *(iii)* a GPU error log report.

In the fourth stage, the reports of the fault injections are analyzed to evaluate the impact of the injected fault by comparing the fault-free execution results against the faulty scenarios. This evaluation process considers two different evaluation approaches. The first one evaluates the impact of the faults on the whole performance of the DNN (e.g., accuracy degradation). On the other hand, we evaluate the impact of faults on the individual layer by evaluating spatial and scalar effects at the output tensors.

4.1.1 Fault Classification for Full DNN Inference. In the case of the full DNN evaluation, it is worth noting that the impact of faults manifests as degradation in the operational capabilities of the targeted DNN. Consequently, in this paper, we evaluated such DNN degradation induced by hardware faults by evaluating specific DNN metrics. In particular, we used the *Accuracy* for image classification models and ***Intersection over Union*** **(IoU)** for object detection models.

In the first class of DNNs, we define the **Relative Accuracy Degradation (RAD)** as the relative difference between the DNN accuracy of the fault-free and faulty DNN executions. Equation (1)

describe the MRAD calculation, where ACC_{gold} and ACC_{faulty} indicate the classification accuracy of the *fault-free* and the *faulty* models, respectively.

$$RAD = \frac{ACC_{gold} - ACC_{faulty}}{ACC_{gold}} \tag{1}$$

On the other hand, for object detection DNNs, we used the Jaccard index or Intersection over Union (IoU) between the bounding boxes of the fault-free and faulty DNN executions. Equation (2) describes the calculation of the IoU per fault (IoU_{PF}), where BBx_{gold} and BBx_{faulty} refer to the bounding boxes of the fault-free and faulty detected objects within the image, respectively.

$$IoU_{PF} = \frac{\mid BBx_{gold} \cap BBx_{faulty} \mid}{\mid BBx_{gold} \cup BBx_{faulty} \mid} \tag{2}$$

These evaluation metrics can be used to classify the severity of every fault considering four main categories as listed in the following:

- **Masked:** $RAD = 0.0$ or $IoU_{PF} = 1$. No difference is observed between the faulty scenario and the golden one.
- **Safe-SDC: Safe Silent Data Corruption** describes the $RAD = 0.0$, meaning that the confidence prediction values for at least one image differ from the fault-free scenario, but the classification is still correct. In the case of object detection, an IoU_{PF} greater or equal to 0.9 represents the cases where the detected objects are slightly shifted w.r.t. the fault-free scenario, but the bounding boxes still localize the object correctly.
- **Critical-SDC: Critical Silent Data Corruption** described as $RAD < 0.0$ for image classification. At least one image was wrongly classified in comparison with the fault-free scenario. In the case of object detection, an IoU_{PF} lower than 0.9, indicates that at least one detection bounding box is significantly shifted or resized w.r.t. to the fault-free scenario.
- **DUE: Detected Unrecovered Error** describes the fault effect of producing a system hang or crash. This error interrupts the execution of the DNN at any time. The causes of this behavior can be memory access violation, memory misalignment violation, or timeout; the last one makes the DNN model enter an infinite loop.

4.1.2 Fault Classification for Individual Layers Execution. Regarding the evaluation of single layers, we classify the faults according to the effects they produce at the output tensor of the layer as follows.

- **Masked:** There is no difference between the output tensor in the fault-free and faulty modes of the evaluated layer.
- **SDC:** The fault produces an error in the output tensor compared to the *fault-free* tensor.
- **DUE:** A sudden crash or hand of the GPU while executing the layer. The causes of such fault effects are typically memory access violation, memory misalignment violation, or timeout.

In addition, we can further evaluate the SDC faults using the *fault injection profiling report* in combination with the *Output data corruption report* to obtain the corruption pattern (i.e., spatial error distribution) associated with the fault effects on the evaluated hardware structures of the GPU as well as the magnitudes of error induced at the output of the layers.

4.2 Fault Injection in General Purpose Register Files (GPRFs)

Modeling Permanent Faults in general-purpose register files using HITPT strategies leverages the GPU execution model, where each active thread within a single *Thread-Block* executed in a given SM has access to a private set of registers to support the *SIMT* parallel execution model. In addition, the same register can be used by other threads of different thread blocks executed in the same SM.

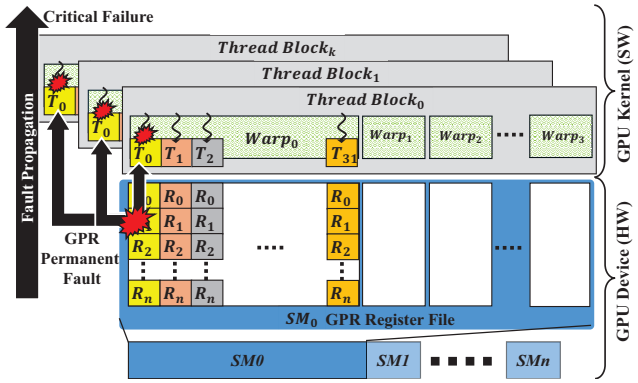


Fig. 4. Propagation of PFs from GPU register files to the application level.

Consequently, the PF implementation requires ensuring that the fault effect persists during the application's execution, corrupting every computation on the targeted defective register across the associated threads in multiple thread blocks. Figure 4 illustrates the propagation mechanism of a PF affecting the register files and its interaction with the GPU kernel execution. This means that one faulty register can impact more than one thread, mainly when they belong to different *Thread-Blocks* executed by the same SM. For example, a faulty register (e.g., *R1*) used by one of the SM0's active threads (e.g., T_0 in $Warp_0$) will induce errors on equivalent threads for all *Thread-Blocks* executed in the same faulty SM.

This fault behavior can be modeled at the software level by modifying the instruction-level program, where the fault effect must be refreshed every time the faulty register is used as a destination. This register corruption is conducted by inserting special routines (i.e., software-level *saboteurs*) after all Kernel's instructions using the selected register as the destination. The saboteur routines use hardware and software identifiers to identify the specific register where the fault has to be injected. More in detail, to ensure the fault controllability, we used the following fault identifiers: <*SMID*, *threadID*, *RegisterID*, *Mask*, *stuck-at*>. *SMID* represents the *SM* where the fault is injected; *threadID* is the resident or active thread inside the *SM*; this identifier allows identifying the set of registers for a unique *WarpID* and *LaneID*; *RegisterID* is the faulty target register; *Mask* is the bit location inside the target register; *stuck-at* represents the type of the fault (0 or 1) according to the stuck-at fault model.

4.3 Fault Injection in Predicate Registers (PRs)

The predicate registers in GPUs are special hardware units that support the control flow execution of the GPU's kernel execution, also known as thread/warp divergence [12, 17]. In this case some threads in a warp may be required to execute different algorithm paths. Consequently, GPUs incorporate a set of private registers (up to eight different predicate registers) per active thread in an SM that support the control flow of a parallel application execution. Given the complexity of the GPU kernel, the compiler determines the usage and the number of predicate registers during the compilation stages.

During the normal kernel execution of the thread divergence, the control flow instructions (e.g., comparisons) set the value of a selected predicate register for all active threads. Then, the subsequent instructions using predicated conditions are executed only by those threads with valid values in the associated predicate register. Finally, when all predicated and not predicated instructions are executed, a warp synchronization instruction reestablishes the lockstep execution for all threads, continuing the program execution in parallel for all threads.

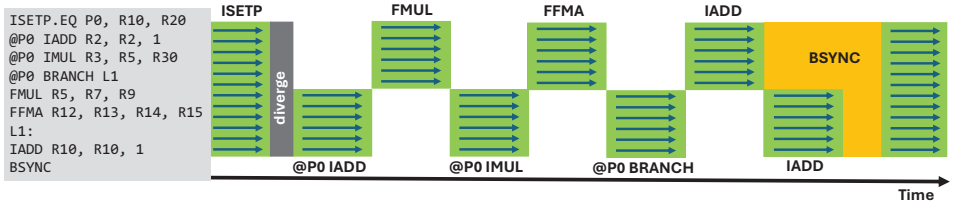


Fig. 5. Illustration of the GPU's divergence operation using the predicate register P0. In the example, a group of threads with a true value in P0 takes the path of the predicted instructions (i.e., instructions with @P0 modifier), and a second group of threads with a false P0 value executes the non-predicted instructions.

Figure 5 illustrates an example of the usage and operation of predicate registers in a divergence execution of a GPU. First, the ISETP instruction compares the content of two registers and, according to the comparison operation (e.g., equals (EQ)), sets the result of the comparison into the selected predicate registers (e.g., P0). It is worth noting that this comparison and PR assignment occur in parallel for all the active threads at the time the comparison is issued. Then, the stored value in the predicate registers is used to decide whether a thread executes a predicted instruction or not. In a GPU kernel, the predicated instructions use the predicate condition (e.g., @P0), indicating that a given thread executes the instruction if its associated predicate register (P0) contains a true condition. In addition, the threads that do not execute the predicated instruction take a different path, executing instructions in an alternative path. The divergence mechanism finishes with a thread synchronization instruction such as BSYNC.

However, when a fault affects any PR of a given active thread, it potentially will cause changes in the GPU's divergence operation, causing the affected thread to follow a different execution path with respect to the one that is supposed to take in a fault-free situation. Figure 6 depicts the case where faults on the PR of two different threads (highlighter in red color) change their execution flow following the wrong path of the algorithm. This behavior may produce critical effects on the execution of the application, and in the case of a DNN execution, it can produce either wrong prediction outputs or a crash of the whole system.

In this regard, we implemented a fault injection mechanism that targets the PR of the GPU's SMs by corrupting a specific PR of a selected active tread. The fault injection resorts to a saboteur routine allocated right after the instructions that use the target PR as the destination. This saboteur routine maintains the permanent effect of the fault during the complete kernel execution. In order to ensure the fault controllability on a single thread and PR, we used similar fault identifiers as the case of GPRFs as follows: <*SMID*, *threadID*, *PRID*, *stuck-at*>. *PRID* corresponds to the specific PR target of the fault injection, and *stuck-at* represents the type of fault (0 or 1) according to the stuck-at fault model.

4.4 Fault Injection in SP and SFU Cores

The fault modeling of a PF in SP and SFU cores of the GPU follows a strategy similar to the one introduced for the GPRFs. Nonetheless, in this case, the HITPT strategy must corrupt all instructions associated with the target core inside an SM. Figure 7 depicts an illustrative example of the propagation effects of a faulty GPU's core through the GPU's kernel execution. Specifically, a defective core (e.g., the FP32 core of the SM sub-core 0 inside the SM0) produces incorrect computations in several threads of the parallel application. Nonetheless, only the warps assigned to the faulty SM's sub-core are exposed to data corruption. Likewise, the permanent effect of a fault will affect all *Thread-Blocks* assigned to the SM where the faulty core resides.

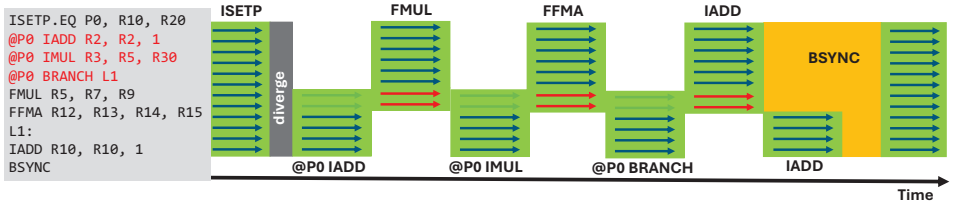


Fig. 6. Effects of permanent faults on the predicate registers P0 of two threads that caused those threads (highlighted in red) to take a different path in comparison to the fault-free scenario.

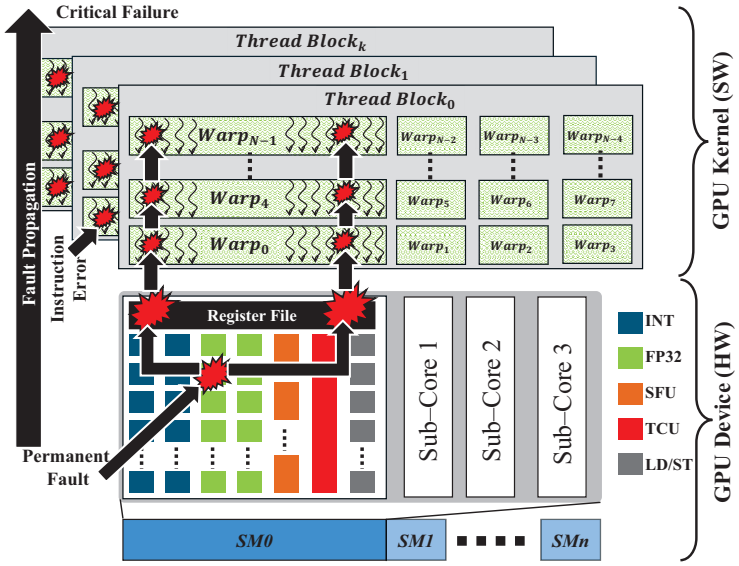


Fig. 7. Propagation of PFs from functional units to the application level.

It is worth noting that the hardware defects in the arithmetic cores of the GPU are visible at the software level on the register files associated with the threads computed by the defective core. Consequently, we devise a HITPT-based fault injection mechanism corrupting the registers (i.e., Destination or Source) used by the instructions issued on the targeted defective functional unit, mimicking faults on the interfaces (i.e., inputs and outputs) of the target GPU core. We used such hardware/software interaction to model single stuck-at faults in each data-path core's input/output interfaces affecting one arithmetic core at a time. Hence, the fault injection modifies **all instances of the same instruction associated with the selected core**, transforming the values of the registers (i.e., forcing stuck-at 1/0). In order to guarantee fault controllability, we selected several hardware and software identifiers as follows <*SMID*, *SubCoreID*, *CoreID*, *OpcodeID*, *Mask*, *IOPortID*, *stuck-at*>. *SMID* represents the *SM* where the fault should be injected; *SubCoreID* is the number of the SM sub-core targeted for the injection; *CoreID* defines the target core number inside the SM sub-core; *OpcodeID* defines the faulty operation affected by the injected fault. *IOPortID* defines the input/output ports interface where the fault will be allocated (e.g., 0: output port, 1, 2, or 3 input ports index); *Mask* is the bit location inside the target port; *stuck-at* represents the type of the fault (0 or 1) according to the stuck-at fault model.

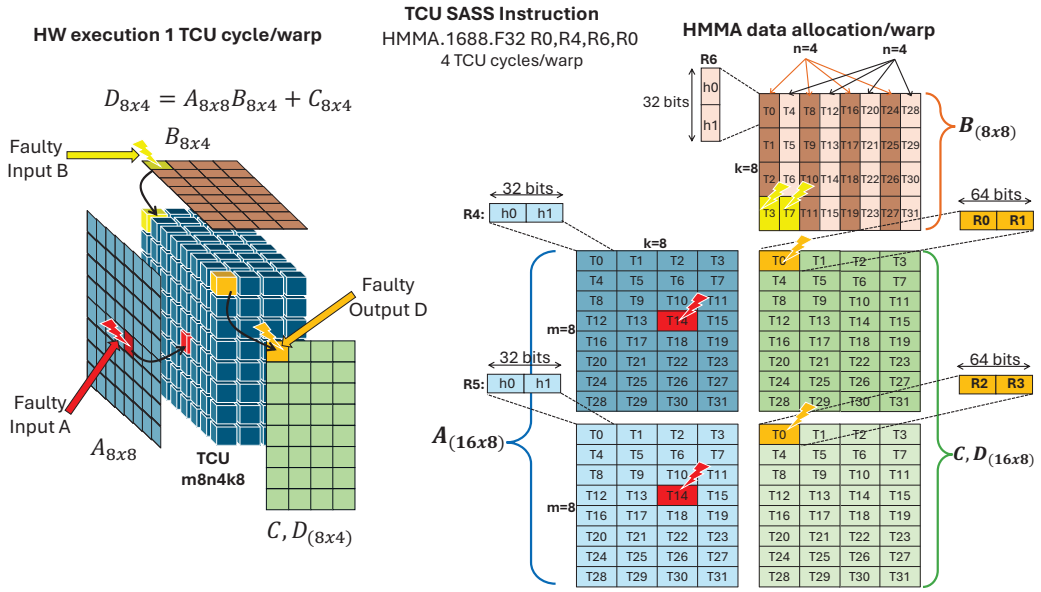


Fig. 8. Modeling the effect of permanent faults on the Ampere's TCU while executing the `HMMA.1688` instruction. According to the A100 Ampere architecture, the TCU hardware performs one 8x4x8 Matrix Multiplication in one clock cycle per warp (i.e., 32 threads) [13]. In addition, the `HMMA.1688` instruction spends four clock cycles to compute a Matrix Multiplication (MM) of shape 16x8x8 [13]. Therefore, one fault in the TCU hardware can be modeled by corrupting four values in the input and output matrices of the executed instruction (i.e., A, B, C, or D).

4.5 Fault Injection in Tensor Core Units (TCUs)

The fault injection targeting the TCU cores resorts to a modeling strategy that mimics the effect of permanent faults at the inputs/outputs of the TCU unit, following a similar strategy used for the case of SPs and SFU cores. Figure 8 illustrates the TCU operation and the fault effects propagation for an Ampere GPU architecture when executing the instruction `HMMA.1688.F32 R0,R4,R6,R0`. Unlike the SPs or SFU cores, the operation of the TCU requires that all threads in a warp (i.e., 32 threads) collectively hold the A, B, C, and D operands matrices. The size of such matrices is determined by the instruction specifier $m \times n \times k$, meaning that A, B, and C/D have shapes of $m \times k$, $k \times n$, and $m \times n$, respectively [15].

In the example depicted in the Figure 8, the instruction specifier `1688` corresponds to m=16, n=8, and k=8 matrices dimensions computed by one warp, which also determines how the operands matrices are loaded and distributed in the registers across all threads within the warp. In detail, the matrix A (light blue) uses two 32bit registers per thread (e.g., R4 and R5), and every register allocates two different float16 values (named as $h0$ and $h1$), allowing to have a matrix of size 16×8. Similarly, matrix B (light red) uses one 32-bit register per thread (e.g., R6) that fits two float16 values, creating a matrix of size 8×8. Finally, the accumulation and result matrices C/D (light green) use four different 32-bit registers (e.g., R0, R1, R2 and R3) producing a result matrix of size 16×8 in float32 representation.

It is worth noting that the TCU core in the Ampere architecture executes a matrix multiplication of shape 8x4x8 in 1 clock cycle. Therefore, the execution of the `HMMA.1688` instruction requires four clock cycles to generate the final result [13]. This behavior indicates that the `HMMA.1688` instruction incorporates an inner tiling execution that splits the original shapes into four independent

matrix multiplications, one per output register in the warp. For example, the first TCU computation executes $R0 = R4 \times R6^{even} + R0$ for all the threads in the warp, where $R0$ corresponds to the first segment results of the multiplication, $R4$ holds the first half of the matrix A of shape [8x8], and $R6^{even}$ indicates the elements in the even columns of matrix B forming the shape [8x4]. Similarly, the second TCU computation executes $R1 = R4 \times R6^{odd} + R1$, where $R6^{odd}$ indicates the elements in the odd columns of matrix B forming the shape [8x4] and $R1$ corresponds to the second segment result.

According to the assembly-level execution of TCU instructions, any fault in the inputs/outputs of the TCU core can be effectively modeled directly in the registers that contain the HMMA operands. For example, a fault in one of the outputs of the TCU core (orange) is equivalent to corrupting all output registers for a single thread within a warp (e.g., R0, R1, R2, and R3 for T0). On the other hand, any fault at the inputs of the TCU can be modeled by corrupting the registers associated with the inputs matrices A and B, for instance, corrupting R4 and R5 in T14 for operand A, or corrupting R6 in T3 and T7 for operand B. In order to guarantee the persistence of the permanent fault during the execution of the application, it is necessary to insert an instrumentation function in all HMMA instructions. The instrumentation functions are placed before or after the target instruction according to the location of the fault.

In order to guarantee fault controllability for the TCU, we selected several hardware and software identifiers as follows <*SMID*, *SubCoreID*, $IOTCU_{ID}$, *ThreadID*, *Mask*, and *stuck-at*>. *SMID* represents the *SM* where the fault should be injected; *SubCoreID* indicates the target TCU inside the SM; in Ampere GPUs architectures, there are four TCU units; $IOTCU_{ID}$ defines the input/output ports interfaces of the TCU where the fault will be allocated (e.g., 0: output matrix, 1, 2, or 3 are the indexes of the input matrices A, B, and C respectively). *ThreadID* indicates the specific DPU inside the TCU targeted for fault corruption. *Mask* is the bit location inside the target register associated with the selected matrix operand; *stuck-at* represents the type of the fault (0 or 1) according to the stuck-at fault model.

4.6 Unified Fault Injection Procedure

Algorithm 1 describes the unified mechanism we devised to mimic the permanent effect of a fault on a GPU device during the DNN's execution. This process works per Kernel issued to the GPU by following two main steps: *(i)* kernel instrumentation and *(ii)* kernel compilation and execution. In the first step, every GPU's Kernel is intercepted to insert the fault injection mechanism at the assembly source code. Such kernel transformation first inspects every assembly instruction to obtain the instruction *Opcode* and the destination and source operands (i.e., R_{dest}, Src_1, Src_2 and Src_3), and then insert specialized saboteur routines according to the specific faulty hardware structure of the GPU. When considering faults on the GPRFs or PRs, the same saboteur routine is inserted after all instructions containing the target register Reg_{ID} or PR_{ID} as the destination register. On the other hand, when targeting SP, SFU, or TCUs, there are two types of saboteur routines. The first one models PFs in the input interfaces of the target core, and it must be allocated before each instance of the target *OpcodeID* instruction. The second type of saboteur routine models PFs at the output interface of the core, and it is placed after the target *OpcodeID* instruction. The *IOPortID* or $IOTCU_{ID}$ identifier defines the saboteur routine to be inserted.

Once the Kernel is instrumented, the next step performs the Just-in-Time compilation to create the binary representation of the new Kernel's version [56]. Also, the GPU resumes the execution of the application and submits the faulty Kernel instead of the original one. The fault is injected and propagated during the execution of the instrumented Kernel (i.e., the faulty Kernel). This propagation is ruled by the saboteur routines that select a specific *SM*, *Warp*, and *thread* according to the runtime identifiers selected for each specific faulty hardware.

ALGORITHM 1: GPU kernel Instrumentation flow for fault injection on the GPRFs, PRs, SPs, SFUs, and TCUs

Inputs:

GPRFs fault descriptor: Fault F_i defined by: $< SM_{ID}, Thrd_{ID}, Reg_{ID}, Mask, ST@ >$

PRs fault descriptor: Fault F_i defined by: $< SM_{ID}, Thrd_{ID}, PR_{ID}, ST@ >$

SPs & SFU fault descriptor: Fault F_i defined by: $< SM_{ID}, SubCore_{ID}, Core_{ID}, Opcode_{ID}, Mask, IOPort_{ID}, ST@ >$

TCUs fault descriptor: Fault F_i defined by: $< SM_{ID}, SubCore_{ID}, IOTCU_{ID}, Thread_{ID}, Mask, ST@ >$

Output: Application output affected by the fault F_i

```
1:  for each kernel K_i in the DNN model do
2:      for each instruction I_j in K_i do
3:          if FI-Mode == GPR then                                   ▷ Registers FI Mode
4:              Inspection(I_j: Opcode R_dest, Src_1, Src_2, Src_3)
5:              if R_dest in I_j matches the target Reg_ID then
6:                  Insert injection function after I_j
7:              end if
8:          end if
9:          if FI-Mode == PR then                                    ▷ Predicate registers FI Mode
10:             Inspection(I_j: Opcode R_dest, Src_1, Src_2, Src_3)
11:             if R_dest in I_j matches the target PR_ID then
12:                 Insert injection function after I_j
13:             end if
14:         end if
15:         if FI-Mode in [SP, SFU]  then                            ▷ Functional Units in SPs and SFU FI Mode
16:             Inspection(I_j: Opcode R_dest, Src_1, Src_2, Src_3)
17:             if Opcode matches the target Opcode_ID then
18:                 if IOPort_ID in [Src_1, Src_2, Src_3] then
19:                     Insert injection function before I_j
20:                 else
21:                     Insert injection function after I_j
22:                 end if
23:             end if
24:         end if
25:         if FI-Mode == TCU then                                   ▷ Tensor Core Units FI Mode
26:             Inspection(I_j: Opcode R_dest, Src_1, Src_2, Src_3)
27:             if Opcode matches the target HMMA then
28:                 if IOTCU_ID in [Src_1, Src_2, Src_3] then
29:                     Insert injection function before I_j
30:                 else
31:                     Insert injection function after I_j
32:                 end if
33:             end if
34:         end if
35:     end for
36:     Just In Time compilation                                     ▷ Faulty Kernel Execution
37:     Instrumented kernel execution
38: end for
```

5 Experimental Setup

The proposed fault injection strategy for assessing the effects of permanent faults at the architectural level of the GPU was implemented as a binary instrumentation tool by customizing *NVBitFI*. This new fault evaluation strategy was applied to a set of representative DNN models, also focusing on a selection of individual layers to evaluate the effects of permanent faults in different hardware structures of GPUs. More in detail, we evaluate permanent fault effects in general-purpose register files, predicate registers, scalar processors, special function units, and tensor core units in GPUs.

The experimental evaluation features a selection of representative DNN model architectures that use the most common GPU operations during a DNN inference, ranging from small DNNs (e.g., LeNet) with few compact GPU kernels to large DNN models (e.g., transformers) that use complex GPU kernels. This spectrum of applications is representative of different scenarios in terms of possible GPU usage, and enables effective evaluation of the impact of permanent faults. Specifically,we used 10 pre-trained non-quantized DNN models: LeNet, AlexNet, Darknet19, VGG-16, MobileNetV3, ResNet50, Vision-Transformer (ViT), Yolo-V3-tiny, Yolo-V3-MobileNetv1 (yolo-V3-MNV1), and Yolo-V3-MobileNetv3 (yolo-V3-MNV2). The LeNet model can classify images of handwritten digits (0 to 9) using the MNIST dataset. AlexNet, Darknet19, MobileNetV3, and ResNet50 classify images from 1,000 categories from the ImageNet dataset. VGG-16 and ViT classify images from 10 classes defined by the CIFAR-10 dataset. The Yolo models perform object detection capabilities that can detect 23 different objects from images of the COCO dataset.

It is important to underline that the models AlexNet, MobileNetv3, and ResNet50 were obtained from the available pre-train models in Torchvision. The DarkNet19 model was obtained from the darknet environment [39]. The VGG-16 and ViT models were fine-tuned using the cifar10 dataset using PyTorch, and the Yolo-v3 models were obtained from the pre-trained framework available in [30]. In addition, 1,000 images were used from the validation datasets to perform the inference of the image classification models. On the other hand, for the object detection models, we used 100 images from the validation datasets containing three to five different objects.

In this work, we adopted Pytorch+TensorRT as the DNN evaluation framework for all DNN models, except for Darknet19 implemented using the darknet framework [39]. All DNN models were configured to use the TCU core of the GPU. In addition, we developed a PyTorch-based DNN setup to evaluate 17 individual convolutional layers taken from the most representative DNN architectures (i.e., two layers from LeNet and the first five layers of AlexNet, MobileNetv3, and ResNet50, respectively).

The universe of permanent faults to be considered during the FI campaign may be excessive due to the complexity of the DNN and the number of fault locations, which might be proportional to the size of the GPU device. For example, the number of faults in the GPU's GPRFs of a single SM can exceed 5 million for the Ampere architecture. In this regard, leveraging the regular structure of a GPU, we target a single hardware structure on a single SM core (e.g., GPRFs, SPs, or TCUs) to perform FI campaigns. As the GPU has multiple SMs, the FI targeting only one of them is expected to show similar effects when targeting the same hardware component in other SMs, as it was already demonstrated in [11].

In order to demonstrate the flexibility of the FI framework, we performed several fault injection campaigns. In the first place, when considering full DNN workloads, we conducted six FI campaigns per DNN model targeting the SM0 of the GPU (i.e., one FI campaign targeting the GPRF, PR, INT, FP32, SFU, and TCU structures). We selected the SM0 since, according to the profiling information obtained for different DNNs, we observed that such an SM core has higher thread block assignation than the others. Therefore, the higher the usage of a given SM, the higher the probability a fault will significantly impact the application, and consequently, the same faults in other SM cores will have an equivalent or slightly lower impact as demonstrated in [11].

The experiments related to GPRF involve the evaluation of around 112,000 faults corresponding to all the registers for one thread of a unique Warp (i.e., ≈ 16,000 faults per DNN). Likewise, we evaluated 1,280 faults on the PR of one thread in one warp for all DNN models. Regarding the FI campaigns targeting the computational cores of the GPU, we conducted four different FI campaigns considering all possible faults for only one *INT*, *FP32*, *TCU*, and *SFU* operation at a time in the $SubCore_0$ of the SM_0 of the GPU. For this fault simulation, the list of faults includes all instructions' opcodes related to the target functional units. This set of experiments resorted to injecting around 48,000 faults in total (i.e., ≈ 4,800 per DNN).

Additionally, we executed a further FI campaign at the DNN's application level, targeting the permanent faults on the static parameters of the DNN (i.e., weights). This fault injection follows the state-of-the-art approach presented by [41], injecting stuck-at faults on the weights of the DNN, adopting a statistical fault sampling providing a confidence level of 99% and an error margin of 1%. This FI campaign allows us to quantitatively compare the differences between the results produced by high-level and instruction-level FI approaches. This experimental evaluation involved the evaluation of around 3 million faults (i.e., ≈ 14,000 faults per layer on convolutional or fully connected layers for all DNNs).

Finally, we performed FI on individual convolutional layers taken from different DNN models. The experiments allow us to assess the impact of faults in the GPU while executing single DNN operations. The experiments consist of performing 34 FI campaigns targeting different GPU architectures. Every layer was assessed using two different FI campaigns: (1) statistical fault injections targeting the register files of one SM adopting a statistical fault sampling based on a confidence level of 99% and an error margin of 1% (≈ 5,000 faults per layer), and (2) an exhaustive FI campaign targeting the outputs of three individual floating point operations (i.e., FMAD, FMUL, FADD) in all the Scalar Processors (SPs) of the SM0 (≈ 2,048 faults per layer).

The experiments were conducted on a workstation HP Z2 G5 with an Intel Core i9-10800 CPU with 20 cores and 32 GB of RAM. It was equipped with an RTX 3060TI GPU platform incorporating an NVIDIA Ampere architecture. In addition, the experiments related to FI campaigns on individual convolutional layers were also carried out on the embedded GPU Jetson Nano.

6 Experimental Results

This section presents the experimental results regarding the reliability evaluation of DNNs with respect to permanent faults in GPUs. It is worth noting that the complete experimental setup exercised the target GPU by executing more than 40,000 GPU kernels for every evaluated fault. This significant number of kernels is due to the fact that some layers at the DNN architectural level (e.g., PyTorch) are mapped into more than one GPU kernel. Hence, our experimental evaluation encompasses about 6.5×10^9 kernel executions for all DNNs and evaluated faults. The experiments concerning the full DNN workloads required around 309 hours on the RTX 3060TI GPU. The individual layer evaluation experiments lasted for 168 hours using the Jetson Nano and 84 hours using the RTX 3060TI GPU. The application-level fault injections lasted for around 108 hours.

6.1 FI Results for full DNNs: Register Files

Figure 9 depicts the fault classification results of FI on the GPRF and the PR of the GPU device, respectively. When the FI campaign considers all registers in one resident thread ($SM_0, Warp_0, Thread_0$), the results show that between 32% to 55% of faults crash the GPU device (*DUE*), preventing the DNN's complete execution. Furthermore, the number of faults inducing wrong results (*Critical-SDC*) for LeNet, Darknet19, VGG-16, MobileNetv3, and ResNet50 does not exceed 10%; only for AlexNet and Yolo-V3-MNV3 the number of these faults reaches almost 20%. Between the 20% and 37% of the faults corrupt the generated DNN outputs without changing the

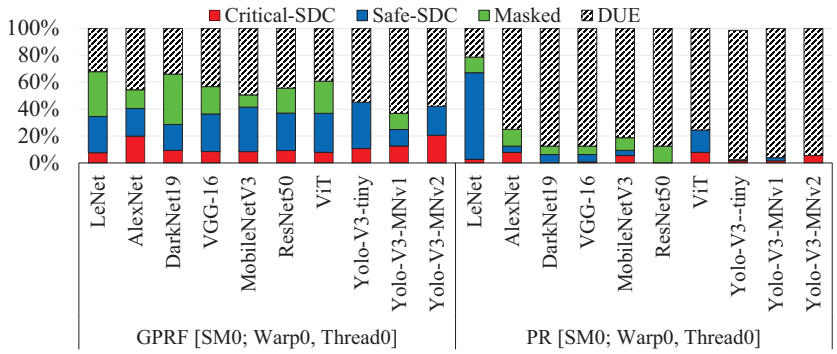
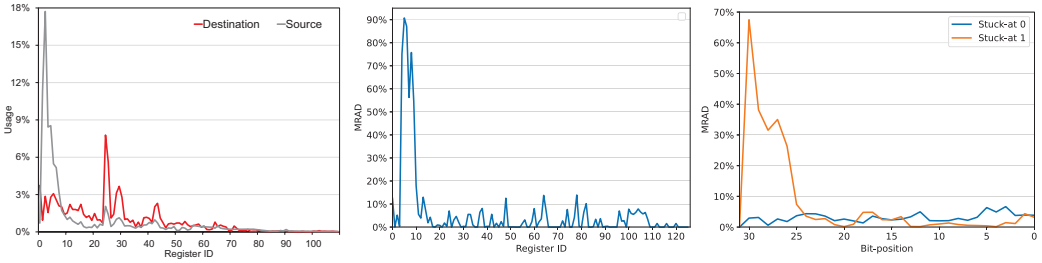


Fig. 9. Fault classification results for GPRF and PR fault injections.



(a) Register Usage (b) Register sensitivity (c) Bit sensitivity

Fig. 10. Usage of GPU's registers and sensitivity of GPU's registers to permanent faults.

classification results (*Safe-SDC*), and less than 40% of faults do not have any impact on the DNN's inference (*Masked*).

On the other hand, the FI on the predicate registers indicated that faults affecting this hardware structure mostly produce a device crash on hang (i.e., DUEs). In fact, for all DNN models, more than 60% of the faults led to DUEs caused mainly by illegal memory access. Interestingly, the faults on PR for the LeNet Model only produced DUEs in only 20% of the cases, while other faults did not produce any effect or only caused critical SDC in a few cases. After studying the differences between LeNet and the other models, we found out that in the case of LeNet, the GPU's kernels only used four PR out of eight possible ones. In addition, the number of kernels and the workload size (e.g., number of Thread Blocks) is significantly lower for LeNet than other DNNs such as ResNet or Yolo-V3 models. It is important to note that up to 5% of the faults in the PR can induce wrong predictions of the DNN, especially for AlexNet, MobileNetV3, and ViT; in the case of yolo-v3 models, more than 95% of the faults cause DUEs, and just around 4% produce wrong object detection or significant deviations for yolo-v3-MNV2.

6.1.1 Register-wise Sensitivity to PFs. A register-wise evaluation indicates that some registers are more sensitive to faults than others due to their usage inside the application. Such register usage is defined by the compiler considering aspects such as the performance and occupancy configurations of the GPU kernels. In fact, as depicted in Figure 10(a), the set of the first 10 registers is the most used one for the evaluated DNNs, which, at the same time, cause faults in those registers to produce significant DNN degradation.

Figure 10(b) depicts the impact of PFs arising in the GPRF (faults inducing DUE effects are not included in the analysis). The figure represents the **Mean Relative Accuracy Degradation**

(MRAD) per register. This MRAD metric measures the average degree of misclassification produced by faults in the registers used by one resident thread in the SM0. From the plot, we can observe that the registers from R3 to R11 are the most critical ones, producing more than 50% MRAD and reaching up to 90% MRAD for R8. The accuracy degradation in the other registers is uniform and does not exceed 15% MRAD. Although PFs in many registers produce less than 20% MRAD, this is still a high percentage of critical effects that create risky results for any application relying on DNN models executed by GPUs.

A deep inspection of the disassembled kernels indicates that the first 10 registers of each thread in a GPU are used as an indexing mechanism to manage the parallel execution on the GPU. These registers contain the *threadID* and *blockID* parameters that individualize every thread execution and their memory accessing. Therefore, a fault affecting those registers can propagate by corrupting the parallelism of the GPU, forcing some threads to compute wrong data, which in turn corrupts the complete kernel computation.

6.1.2 Bit-wise Sensitivity to PFs. We also evaluated the registers' bit-wise sensitivity to PFs considering the MRAD metric. Figure 10(c) illustrates the bit-oriented accuracy degradation for the AlexNet model produced by stuck-at-0/1 faults. Although we introduced the results for AlexNet in order to simplify the presentation of the results, it is important to highlight that we found a similar behavior for all other evaluated DNNs. The results consider only the effect of the faults silently propagated to the DNN's output (SDCs).

The results show that the propagation effect of stuck-at-0 faults does not exceed 9% of MRAD. However, stuck-at-1 faults significantly impact the classification result of the DNN, especially for the **most significant bits (MSBs)** of the registers (25th to 30th, but especially the 30th bit), which generate an accuracy degradation up to 68%. Interestingly, the MSBs causing the higher accuracy degradation correspond to the exponent bits used by the IEEE754 standard for the floating-point representation. The results prove that stuck-at-1 faults located in the 25th bit and beyond produce a significant MRAD. Consequently, such bit locations on the register files are highly sensitive to faults, corresponding to a high probability that a fault generates wrong DNN outcomes.

In the end, the observed results point out new challenges regarding the development of fault tolerance mechanisms able to counteract the fault effects of permanent faults on GPUs's registers. Although there are multiple attempts to enhance the reliability of DNNs by resorting to high-level hardening solutions (e.g., clipping the activation functions at the DNN architecture level), their scope is restricted to mitigate fault effects after the execution of one or several GPU operations (i.e., kernels). This means that high-level hardening solutions do not have control over the GPU execution itself, which constrains their effectiveness in mitigating complex effects of permanent faults (e.g., broken thread executions caused by predicate or register files). Hence, it is necessary to devise or adopt more elaborate solutions that are able to act within the GPU device. In this regard, the fault effects generating wrong thread indexing or changes in the execution flow inside the GPU can only be addressed by adopting fault-tolerant mechanisms at the level of the GPU kernels or threads. This unveils opportunities for adopting selective hardening solutions at the thread level by recomputing the thread index values and extra control flow checking strategies able to identify possible wrong thread execution in case of divergencies (e.g., **Triple modular redundancy (TMR)** or **duplication with comparison (DWC)** strategies).

6.2 FI Results for Full DNNs: SPs, SFUs, and TCUs

Figure 11 shows the fault classification results for the experiments described in Section 5 regarding faults on the SPs (i.e., INT and FP32), SFU and TCU cores of a GPU. The results presented by Figure 11 show that a significant amount of faults (> 40%) injected in the integer units (INT)

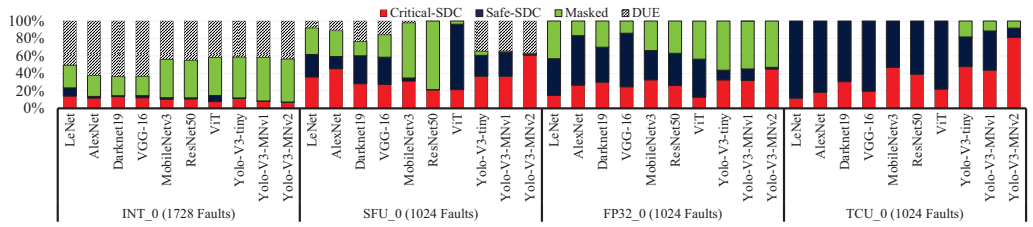


Fig. 11. Fault classification results for fault injection campaigns targeting the GPU's FUs.

produce a crash or hang in the device (DUE), preventing the DNN from completing the inference process. A deep study of the generated FI reports indicate that faults on the INT cores mainly produce memory misalignment or memory access violations. The experimental results also show that around 15% of the faults in the INT cores can induce wrong classification results (Critical-SDC) for all evaluated DNNs. Interestingly, a minimum number of faults in the INT unit (< 3%) have a tolerable impact on the DNN's output (classified as Safe-SDCs), except for the LeNet, where around 9% of the faults correspond to Safe-SDCs. We can also observe that approximately 24% and 40% of the injected faults in the integer units do not impact the DNN outputs (Masked).

Regarding faults on the SFU cores, the results show that around 25% of the faults can induce an application crash (i.e., DUEs) for LeNet, AlexNet, DarkNet19, and VGG-16; whereas for yolo-v3 models this number of faults might increase up to ≈40%. On the other hand, between 20% to 50% of the faults in the SFU core can be classified as *Critical-SDCs* depending on the evaluate DNN. LeNet, AlexNet, and the yolo-V3 models exhibit the higher percentage of faults that produce critical-SDCs, while ResNet50 and ViT have the minimum percentage of faults (≈20%) falling in the same category. When it comes to the faults classified as safe-SDCs, we observed that 30% of the injected faults produce tolerable effects for LeNet, Darkent19, Yolo-v3, and VGG16, whereas less than 10% of the faults are considered Safe-SDCs for AlexNet, MobileNetv3, and ResNet50. Interestingly, around 80% of the faults evaluated on the SFU while executing the ViT lead to safe-SDCs. Although all DNNs use the SFU mainly to compute the reciprocal $1/x$, exponential 2^x, and **reciprocal of square root (RSQ)** $1/\sqrt{(x)}$ operations, we found that on MobileNetv3 and ResNet50, the kernel implementation does not use the SFU to perform RSQ operations. Therefore, it seems to appear that removing the RSQ operation from kernels in a DNN masks the effects of the faults in around 70% of the cases, whereas, for the models that use the SFU with RSQ, less than 30% of the faults are masked.

It is worth noting that the FP32 and TCU cores are used exclusively for the DNN dot product computations. Therefore, none of the faults in those hardware units unexpectedly stops the application (DUE), but many of those faults produce SDC effects. In fact, the evaluation of faults injected in the FP32 core's interconnections shows that many of them can be labeled as SDCs. The number of faults classified as Critical-SDCs oscillates between 10% to 40% depending on the evaluated DNN. For example, LeNet and ViT have a lower percentage of critical SDCs below 15%, whereas 40% of the faults in the FP32 cores lead to critical effects in the case of Yolo-V3-MNV2. On the other hand, more than 40% of faults are categorized as Safe-SDCs for all image classification DNNs. Nonetheless, the number of safe-SDCs for the yolo-v3 models accounts for approximately 10% of faults only. Likewise, the percentage of faults considered as Masked does not represent more than 30% for the image classification models and around 40% for the object detection models.

Finally, we observed that all the faults injected in the TCU core of the GPU produce only SDCs when evaluating image classification models, whereas some small percentage can be masked in the case of object detection models (i.e., yolo-v3). More in detail, when analyzing the results for

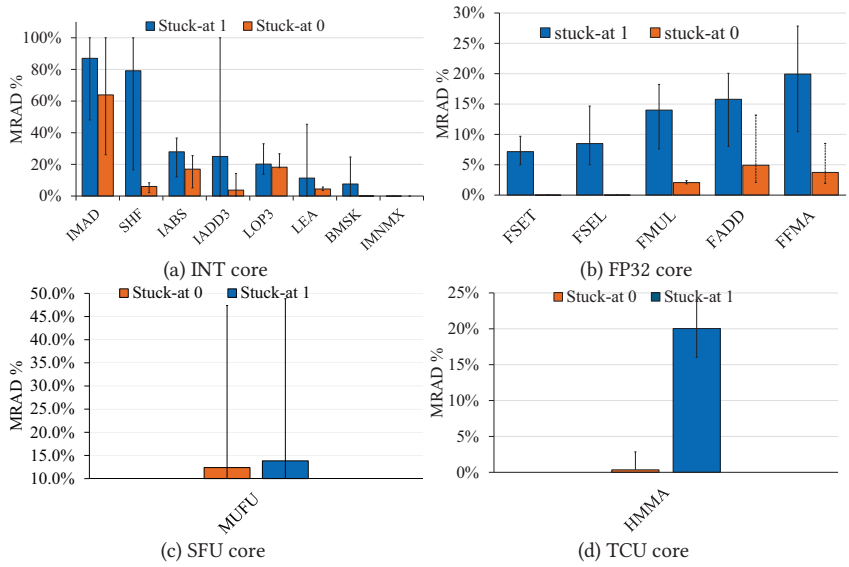


Fig. 12. Operation-wise sensitivity of DNNs to permanent faults in the GPU. The solid bars represent the average MRAD across all faults and evaluated DNNs, and the error bars indicate the maximum and the minimum MRAD across the same evaluations.

image classification, we observed that the percentage of critical faults varies depending on the DNN. For example, less than 20% of the fault in the TCU changes the expected outcome in LeNet, AlexNet, VGG-16, and ViT models. Nonetheless, this percentage of faults increases to 40% in the case of large DNN models such as ResNet50 and MobileNetv3. Interestingly, the percentage of faults on the TCU that affect the object detection models is significantly higher than in the case of image classifications. More in detail, around 80% of the faults in the TCU significantly degraded the detection capabilities of the yolo-v3-MNv2 model.

6.2.1 Operation-wise Sensitivity to PFs. Figure 12 presents the impact of PFs on the accuracy of the image classification DNNs, considering faults that affect the INT, FP32, SFU, and TCU instructions. The plots represent the Mean Relative Accuracy Degradation (MRAD) per instruction. The MRAD metric measures the average degree of misclassification of the DNN produced by faults injected in each processing unit's input/output interfaces. It is worth noting that faults classified as DUEs are not considered in the MRAD calculation because they induce a crash or a hang in the GPU, preventing the generation of a valid inference outcome.

From the charts, we can observe that faults affecting the INT core significantly reduce the accuracy of the DNN when particular instructions are executed. In the case of the **Integer-Multiply-Add (IMAD)** instruction, both stuck-at-1 and stuck-at-0 faults degrade the accuracy by around 80% and 63%, respectively. In the case of the **SHF (Funnel-Shift)** instruction, only the stuck-at-1 faults degrade an average of 80% of the DNN accuracy, and in some cases, this figure reaches 100%. Faults affecting the instructions **Integer-Absolute (IABS)**, **Integer-Add-3Inputs (IADD3)**, and **Logic-Operations-3Inputs (LOP3)** can reduce the accuracy by up to 25% for stuck-at-1 and up to 27% for stuck-at-0. The other instructions have less than 10% of DNN degradation. The significant accuracy degradation of the DNN produced by faults in IMAD, Funnel-Shift (SHF), IABS, and LOP3 can be explained by their usage in the CUDA thread indexing preamble at the beginning of each kernel. These instructions compute the linear thread identification using the ThreadID and

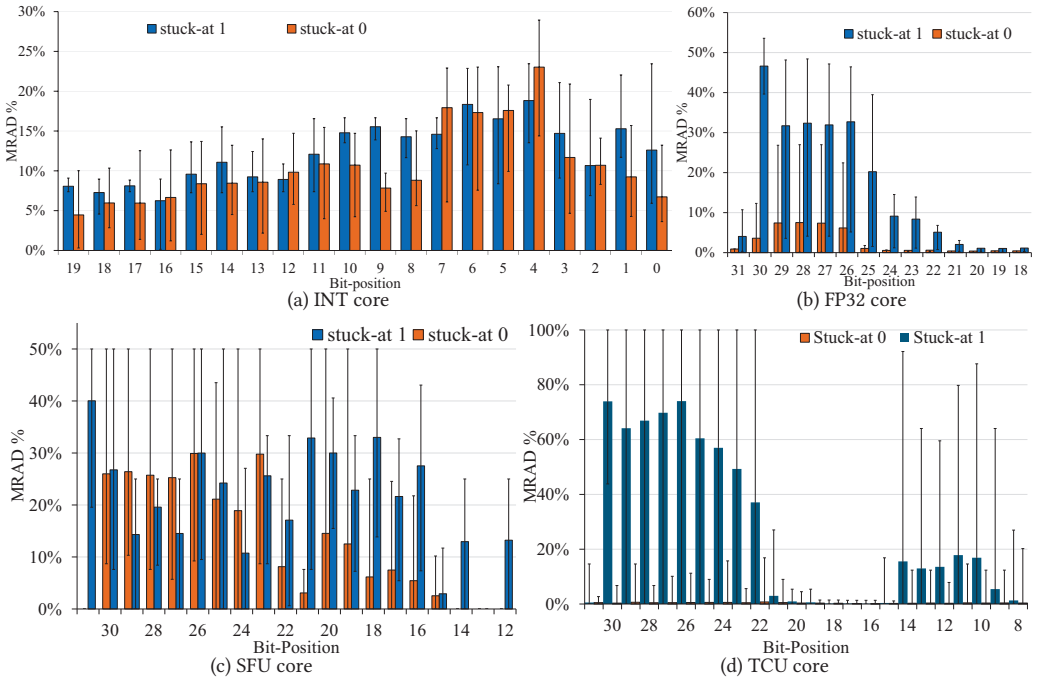


Fig. 13. Bit sensitivity to PFs in the functional units of the GPU for the evaluated DNN models.

BlockID parameters to identify the individual threads and memory addresses. Hence, wrong calculations in this crucial step of the kernel execution significantly damage the program execution in the GPU and, thereby, the DNN computation.

On the other hand, Figure 12(b) shows the accuracy degradation produced by faults on the floating-point cores. The results indicate that stuck-at-1 faults significantly impact the performance of a DNN by 7% for the **Floating-Point-Compare-Set (FSET)** operation and up to 20% for the **Floating-Point-Fused-Multiply-Add (FFMA)** operation. In contrast, the stuck-at-0 faults reduce the DNN accuracy by less than 5% for the **Floating-Point-Multiply (FMUL)**, **Floating-Point-Add (FADD)**, and FFMA hardware operations. Stuck-at-1 faults have significantly more impact than stuck-at-0 on the floating-point operations due to the data ranges in most operations (±1). In this case, the exponent (in the IEEE 754 format) contains some bits in 0 that, once forced to be 1, induce a huge error that propagates and worsens due to the fault's persistent behavior on subsequent operations.

In the case of faults corrupting the operations of the SFU core (i.e., MUFU instruction), we found that the stuck-at-1 faults induce, on average, up to 14% of the MRAD, while the stuck-at-0 faults maximum can degrade the operation of the DNNs by 12.5%.

Regarding the permanent faults on the TCU core, the results showed that faults affecting this AI accelerator in GPUs can reduce the accuracy of the evaluated DNN models by around 20% for stuck-at-1 faults and less than 1% for stuck-at-0 faults. It is worth noting that we only targeted the Matrix Multiply and Accumulate (HMMA) instructions associated with the TCU execution inside the GPU.

6.2.2 Bit-wise Sensitivity to PFs. Figure 13 shows the MRAD produced by faults injected in the input/output interfaces of the hardware cores (INT, FP32, SFU, and TCU) at the bit-level. In the case

of the INT core, only faults in the 20 **least significant bits (LSBs)** of the targeted core produce more than 5% degradation in the accuracy of the DNN. The higher impact is observed in the 4 LSBs, exhibiting between 20% and 30% of MRAD. Interestingly, both types of faults, stuck-at-0, and stuck-at-1, produce similar degradation effects for the evaluated DNNs. These results are explained by the main usage of the INT cores in the memory addressing and threading management of the GPU. Therefore, faults on the MSB mainly induced DUEs; on the contrary, faults in the LSB part of the core imply a wrong memory address but inside the valid addressing memory, which induces wrong results but without rising DUE conditions. Similarly, these cores can wrongly calculate the thread indexing, making some threads execute data from a different one.

On the other hand, for the floating-point cores FP32 and TCU, the stuck-at-1 fault seriously impacts the **most significant bits (MSBs)** of the input/output interfaces. More in detail, the 10 MSBs of the FP32 cores exhibit the highest negative impact on the accuracy of the DNN (by 5% up to 50%). We observed that those bits correspond to the exponent bits and the MSBs mantissa bits, according to the IEEE-754 representation. On the other hand, the faults on the TCU core show high MRAD in two-bit ranges, the portion of bits from 21 to 30 and the portion of bits from 8 to 14. The reason behind this difference, in comparison with the FP32 cores, is due to the TCU operation, which typically performs computations using IEEE754 FP16 format for the input operands and IEEE754 F32 for the accumulation results. Consequently, one 32-bit input is composed of two 16-bit data operands for the TCU. In this regard, we observed that the exponent and the mantissa's MSB are the most sensitive ones to stuck-at-1 faults. Interestingly, the portion of bits from 21 to 30 shows significant accuracy degradation that ranges from 30% to 70% on average, while the degradation of bits from 8 to 14 on average reached 19% of MRAD. The TCU operation can explain this significant degradation difference since the TCU results use FP32-bits such that a stuck-at-1 fault on the exponent mostly produces large-magnitude errors that propagate through the DNN execution. It is worth noting that the stuck-at-0 faults do not significantly affect the accuracy of the DNNs, showing less than 2% of MRAD.

When evaluating the faults on the SFU core at the bit level, the stuck-at-1 fault type produces around 25% of MRAD across all evaluated bits, with slightly more degradation in the MSB than in the LSB. Interestingly, the stuck-at-0 faults only influence the 16 MSBs with a degradation ratio similar to that of the stuck-at-1 faults for the same bits, reaching up to 50% of maximum MRAD.

Although the main focus of our work is on the evaluation of the effects of GPU's permanent faults in DNN computations, we obtained results and insights that could guide in the selection of the most suitable opportunities in developing fault-tolerant mechanisms targeting permanent faults in the compute units of the GPU. In the case of faults inducing a high magnitude of errors in floating point operations (FP32 and TCUs), there are two possible fault tolerance strategies to consider: Fault aware pruning [1], and High magnitude error clipping. The former strategy can be applied by dropping to zero those operations exceeding a given threshold that can be obtained from a fault-free profiling stage. The implementation of this strategy in GPUs can be conducted by modifying the GPU' kernels by incorporating additional procedures that compare intermediate floating-point operations and correct the results when exceeding the limits. Likewise, the second strategy limits the maximum error-induced results by clipping them to a maximum threshold.

On the other hand, faults affecting the INT units in the GPU can require highly elaborated fault-tolerant solutions due to their usage across the whole kernel execution (e.g., thread indexing, memory addressing, control flow, etc.). In this regard, several strategies can be explored to mitigate the impact of faults in such units. For example, thread indexing redundancy can be adopted by using operation diversity, such as replacing INT operations with logical, SFU, or FP32 operations during thread index calculations. Alternatively, thread redundancy is a potential solution that can address fault effects on any GPU's functional units (INT, FP32, SFU, and TCUs). This strategy

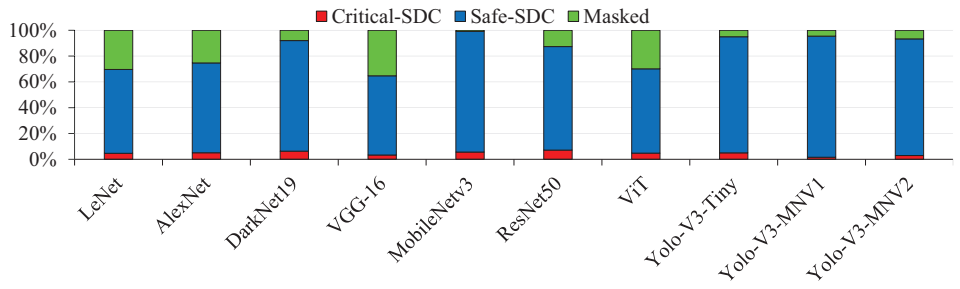


Fig. 14. Fault classification results for application-level FI campaigns.

requires the duplication of every thread of the DNNs GPU's kernels in such a way that every replica is executed on different GPU' cores. In fact, the implicit parallelism of the GPU can be used in favor of fault tolerance mechanism development (e.g., including redundancies) to tackle most of the fault effects at the lowest level (GPU execution), preventing the propagation of errors across the DNN computation.

6.3 Application-level vs. Architectural Fault Injection Campaigns

In the previous subsections, we showed that a significant amount of the evaluated faults can produce **Silent Data Corruption (SDC)** effects on the outputs of the DNN. These faults are dangerous for any DNN-based applications because their presence induces wrong decisions in the system without knowing that the hardware is faulty, potentially leading to catastrophic events. On the other hand, the reliability evaluations of DNNs w.r.t. hardware faults are typically carried out by performing data corruptions at the application level (i.e., stuck-at or bit-flips on the weights). Nonetheless, there are significant concerns about how realistic these application-level FI are with respect to actual faults on the underlying GPU device. In this regard, we conducted application-level FI campaigns to compare the results with those from the FI approach proposed in this work and determined differences concerning the severity of faults using different FI abstraction levels. The results show that the number of faults that induce wrong effects in a more realistic hardware fault scenario (i.e., the HITPT approach) is significantly higher than when using application-level FI strategies (e.g., DNN's weights corruption).

Figure 14 presents the classification results of faults at the application level, targeting the weights of all studied DNNs in this work. The results indicate that less than 10% of faults injected in the parameters degrade the performance of the DNNs by producing a critical impact on the DNN outcomes (i.e., wrong classification results or deviations of the bounding boxes in the object detection applications). Additionally, more than 65% of the evaluated faults produced safe or tolerable results for all the DNNs evaluated, meaning that such faults created some deviations in the DNN outputs, but the results are still correct. From the experiments, we show significant differences among the evaluations carried out by fault injections at the DNNs application level compared with our approach based on HITPT. In fact, the fault evaluations at the application level indicate that the injected faults on the DNN parameters do not exceed 5% of the DNN degradation. Such limited degradation is produced mainly by faults on the 30th bit of the floating-point representations of the parameters. Unfortunately, these levels of evaluation do not allow us to correlate the meaning of the injected faults on the parameters with the actual hardware defects on the GPU. On the contrary, our proposed evaluation approach is closer to the hardware level, performing fault injections at the instruction level and providing deeper insights into the possible defective hardware and the impact on the DNN workload. For this reason, we can state that our

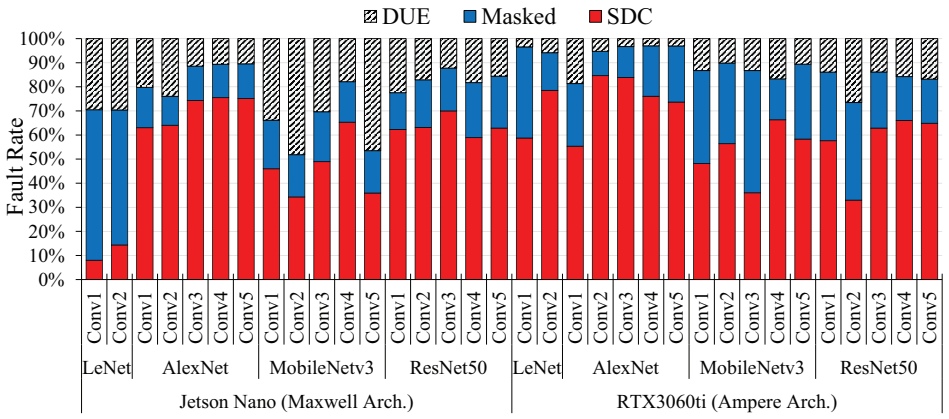


Fig. 15. Fault classification rates for the FI in the register files of individual convolutional layers on Jetson Nano and RTX 3060TI GPUs.

approach allows us to more effectively assess the impact of hardware faults on DNNs, since we inject hardware-aware (more realistic) faults than the high-level fault injections targeting DNN parameters. In fact, when evaluating faults on individual layers (see Section 6.4), we observed that the intermediate software layers between the GPU ISA-level and the DNN high-level framework play an important role in the way a fault propagates through the DNN.

6.4 FI Results for Individual Layers: Register Files

We additionally evaluate the incidence of the GPU device and the kernel implementation on the propagation effects of permanent faults at the outputs of individual convolutional layers of a DNN. We report the results of evaluating permanent faults on register files and the FFMA, FMUL, and FADD units on 17 convolutional layers of different DNN models as described in Section 5. Figure 15 reports the fault classification results of the FI experiments targeting the GPRFs. The results show that the layers of AlexNet and ResNet50 exhibit around 60% of SDC rates, whereas the layers of MobileNetV3 and LeNet have slightly lower SDC rates (by around 10%). More in detail, the fault effects have some variations according to the GPU architecture used for the experiments. In the case of Jetson Nano, the number of faults that induce DUE effects exceeded by approximately 20% the DUE cases when using the RTX 3060TI GPU. Also, when using an RTX 3060TI GPU device, there is a significant increment of the Masked cases (by around 30%) compared to the Jetson Nano GPU scenarios. These results are correlated with the usage of the register on the workload. Since JetsonNano devices have only one SM, the registers are reused during the complete workload, increasing the chance of a fault on the GPRFs to induce DUE results. On the other hand, devices with a higher degree of parallelism (i.e., 38 SMs on RTX 3060TI) distribute the workload among their SMs so that the GPR on an SM is used only on a portion of the workload, reducing the number of DUEs but generating Masked or SDCs effects.

It is worth noting that although all evaluated layers correspond to convolution operation at a high level, the final GPU computation varies by using different kernel operations as presented in Table 2. Nonetheless, when several layers of the same DNN use the same GPU kernel implementation, the fault propagation exhibits similar results of fault effect. For example, the SDCs, DUEs, and Masked figures are similar for the layers 3, 4, and 5 of AlexNet that use the kernel *trt_maxwell_scudnn_winograd_128x128* for Jetson Nano GPU. Similarly, when implemented on the

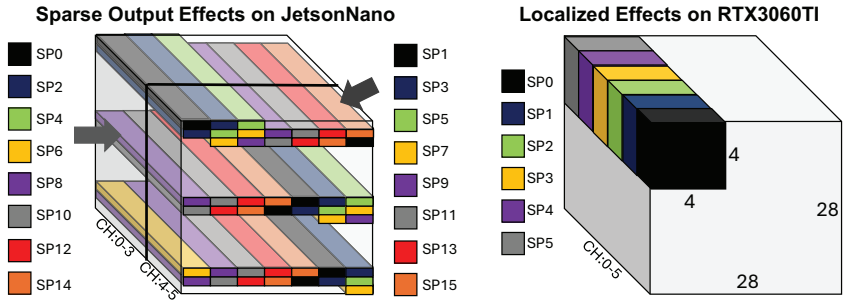


Fig. 16. Distribution of the spatial corruption effects on the LeNet's first convolutional layer (output tensor and its channels 'CH') produced by faults on FFMA units (inside different *SPs*) for Jetson Nano *(Left)* and RTX 3060TI *(Right)* GPUs.

RTX 3060TI GPU, layer 2 of ResNet50 shows fewer SDCs by approximately 30% compared to layers 1, 3, 4, and 5. These results can be explained by the fact that layer 2 for ResNet50 implements a completely different kernel operation in comparison with the other ones.

6.5 FI Results for Individual Layers: Functional Units

After performing the FIs on the GPU units executing FFMA, FMUL, and FADD, we observed that 100% of the faults generate SDC effects at the output of the convolutional layers. It is worth noting that the TCU core was disabled on the RTX 3060TI GPU in order to compare the same experiments with the Maxwell architecture, which does not include the TCU accelerator. As expected, faults on the FP cores of the GPU do not produce DUE effects since these computational units are used only during the dot-product operations of the convolutional layers of the DNN layers. In addition, we performed a spatial evaluation of the SDC effects observed at the output tensor of the computed layer, observing that there are significant differences for the same DNN layers implementing different GPU devices. The spatial evaluation of the SDC effects shows that faults behave differently according to the GPU architecture, the target GPU' core, the layer size, the size of the processed tensors, and the underlying kernel algorithm executed on the GPU. For the sake of simplicity and to demonstrate the level of detail that our proposed fault evaluation strategy can provide, we illustrate only the results for the first convolutional layer of LeNet; other layers for other DNNs show more complex and diverse fault effects. Such results require further analysis (which is out of the scope of this work) in order to generalize the propagation effects of permanent faults in GPUs and formalize error models for fast resilience evaluation at the application level.

Figure 16 depicts a representation of the observed patterns obtained after performing fault injection campaigns on the FFMA core of the GPU. We observed that for the RTX 3060TI GPU device, the PFs on the FFMA induce a localized effect on specific regions of the output tensor for all channels. There is a direct connection between the specific SP and the output channel where the effect is observed. On the contrary, when the same layer is considered on a Jetson Nano GPU board, the effects are distributed differently and in a more sparse shape. These results demonstrate that the proposed fault injection technique can be used for realistic and accurate evaluation of hardware-level faults, regardless of the software implementation employed to compute a DNN model. Furthermore, our fault evaluation strategy provides opportunities regarding the layer-wise characterization of DNNs with respect to permanent faults by generating error modeling mechanisms that describe the effect of faults in terms of magnitude and spatial data corruption. In this regard, using HITPT to evaluate permanent faults on DNN workloads highlights the inadequacy

of relying solely on fault injection at the application level (e.g., in the DNN's weights) to assess the impact of real hardware faults in the underlying GPU.

Finally, the level of detail provided by our proposed evaluation of faults on DNN architectures paves the way for the development of elaborated hardening techniques able to counteract or reduce the most critical effects produced by permanent faults on GPU devices. In fact, the experimental evaluation results provided insights regarding the effects of faults on the DNNs that can be used to explore different opportunities regarding fault tolerance mechanisms at the GPU's kernel level. To make these hardening strategies effective, it is necessary in some cases to adopt selective hardening or to implement thread-level redundancies since any hardening at the DNN application level (e.g., activation function clipping) cannot address multiple faults effects such as register files or functional units involved in the thread indexing of low-level GPU operations. On the other hand, modifying the DNN architecture can introduce additional kernels in the underlying GPU execution, which can even increase the sensitivity to faults instead of counteracting them. This makes our fault evaluation strategy essential for the evaluation and enhancement of the reliability of DNNs deployed on GPU devices.

7 Conclusions and Future Work

This paper proposes a method to investigate the impact of permanent faults (PF) inside GPUs running Deep Neural Networks. We described an instruction-level fault injection strategy to mimic permanent faults on the general-purpose register files, Predicate Registers, Scalar Processors, Special Function Units, and Tensor Core Units in GPUs. We created a prototypical fault injection tool based on the binary instrumentation tool NvBit and conceived a fault injection flow that effectively assesses the impact of permanent faults on different DNN workloads with acceptable computational requirements. To the best of our knowledge, this is the first work proposing an evaluation methodology and reporting results about the impact of PF on different DNN workloads using instruction-level fault injection campaigns. The results show that the first 10 registers are significantly more sensitive to PFs than the others, thus inducing a strong degradation in the operation of the evaluated DNNs in up to 68% of the cases for the evaluated DNNs. On the other hand, the results show that faults affecting the functional units can also induce substantial DNN degradation; indeed, the most sensitive cores are the Integer-Multiply-ADD (IMAD), the Floating-Point-Fused-Multiply-ADD (FFMA), and the Tensor Core Units (TCUs), showing average accuracy degradation figures of 80%, 20%, and 20%, respectively. The results also show that significant differences exist between the results produced by the popular application-level fault injection approaches and those produced by the approach we propose, which models permanent faults closer to the real hardware. When adopting our proposed approach, the number of faults that cause critical effects on the DNN's prediction results can reach up to 64%, whereas less than 10% of the faults at the application level can induce incorrect DNN predictions. In addition, it is not possible to associate such faults with a defective GPU structure as we do in our fault injection strategy.

Our evaluation strategy can be used to identify possible strategies and fault tolerance opportunities. In fact, the evaluation results indicate that selective hardening at GPU's kernel level, by adding thread redundancies or operation diversity, is required and can be used as a more effective fault tolerance mechanism than those developed at DNN abstraction levels. Hence, future works will focus on implementing and evaluating advanced fault-tolerance strategies against permanent faults as well as assessing the effectiveness of hardening techniques developed at the application level. Finally, we envision using the proposed fault injection method to characterize individual DNN layers to conceive more effective application- or algorithm-level fault modeling rather than the typical fault injection on the parameters.

References

[1] M. Abdullah Hanif and M. Shafique. 2020. SalvageDNN: Salvaging deep neural network accelerators with permanent faults through saliency-driven fault-aware mapping. *Philosophical Transactions of the Royal Society A* 378, 2164 (2020), 20190164.

[2] Mohammad Hasan Ahmadilivani, Mahdi Taheri, Jaan Raik, Masoud Daneshtalab, and Maksim Jenihhin. 2024. A systematic literature review on hardware reliability assessment methods for deep neural networks. *ACM Comput. Surv.* 56, 6, Article 141 (Jan. 2024), 39 pages. https://doi.org/10.1145/3638242

[3] Advanced Micro Devices (AMD). 2024. Machine Learning Development on a Local Desktop with AMD Radeon™ Graphics Cards. https://www.amd.com/en/developer/resources/ml-radeon.html. [Online; accessed 7-February-2024].

[4] David F. Bacon. 2022. Detection and prevention of silent data corruption in an exabyte-scale database system. In *The 18th IEEE Workshop on Silicon Errors in Logic – System Effects.*

[5] C. Bolchini, L. Cassano, and A. Miele. 2023. Resilience of Deep Learning applications: A systematic survey of analysis and hardening techniques. https://doi.org/10.48550/arXiv.2309.16733 arXiv:arXiv:2309.16733v2

[6] Alberto Bosio, Paolo Bernardi, Annachiara Ruospo, and Ernesto Sanchez. 2019. A reliability analysis of a deep neural network. In *2019 IEEE Latin American Test Symposium (LATS).* 1–6. https://doi.org/10.1109/LATW.2019.8704548

[7] Sebanjila Kevin Bukasa, Ludovic Claudepierre, Ronan Lashermes, and Jean-Louis Lanet. 2019. When fault injection collides with hardware complexity. In *Foundations and Practice of Security,* Nur Zincir-Heywood, Guillaume Bonfante, Mourad Debbabi, and Joaquin Garcia-Alfaro (Eds.). Springer International Publishing, Cham, 243–256.

[8] Yiran Chen, Yuan Xie, Linghao Song, Fan Chen, and Tianqi Tang. 2020. A survey of accelerator architectures for deep neural networks. *Engineering* 6, 3 (2020), 264–274. https://doi.org/10.1016/j.eng.2020.01.007

[9] Zitao Chen, Niranjhana Narayanan, Bo Fang, Guanpeng Li, Karthik Pattabiraman, and Nathan DeBardeleben. 2020. TensorFI: A flexible fault injection framework for tensorflow applications. In *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE).* 426–435. https://doi.org/10.1109/ISSRE5003.2020.00047

[10] Josie E. Rodriguez Condia, Boyang Du, Matteo Sonza Reorda, and Luca Sterpone. 2020. FlexGripPlus: An improved GPGPU model to support reliability analysis. *Microelectronics Reliability* 109 (2020), 113660. https://doi.org/10.1016/j.microrel.2020.113660

[11] Josie E. Rodriguez Condia, Juan-David Guerrero-Balaguera, Fernando F. Dos Santos, Matteo Sonza Reorda, and Paolo Rech. 2022. A multi-level approach to evaluate the impact of GPU permanent faults on CNN's reliability. In *2022 IEEE International Test Conference (ITC).* 278–287. https://doi.org/10.1109/ITC50671.2022.00036

[12] NVIDIA Corporation. 2017. NVIDIA Tesla V100 GPU architecture. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf. [Online; accessed 27-August-2024].

[13] NVIDIA Corporation. 2020. NVIDIA A100 Tensor Core GPU architecture. https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf. [Online; accessed 7-August-2024].

[14] NVIDIA Corporation. 2024. CUDA C++ Programming guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. [Online; accessed 7-February-2024].

[15] NVIDIA Corporation. 2024. Multiply-and-Accumulate instruction: MMA. https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#warp-level-matrix-instructions-mma. [Online; accessed 27-August-2024].

[16] NVIDIA Corporation. 2024. NVIDIA Deep Learning TensorRT documentation. https://docs.nvidia.com/deeplearning/tensorrt/. [Online; accessed 7-February-2024].

[17] NVIDIA Corporation. 2024. Predicated esxecution. https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#predicated-execution. [Online; accessed 27-August-2024].

[18] H. S. Das and P. Roy. 2019. Chapter 5 - A deep dive into deep learning techniques for solving spoken language identification problems. In *Intelligent Speech Signal Processing*, Nilanjan Dey (Ed.). Academic Press, 81–100.

[19] Corrado De Sio, Sarah Azimi, and Luca Sterpone. 2022. FireNN: Neural networks reliability evaluation on hybrid platforms. *IEEE Transactions on Emerging Topics in Computing* 10, 2 (2022), 549–563. https://doi.org/10.1109/TETC.2022.3152668

[20] Harish Dattatraya Dixit, Sneha Pendharkar, Matt Beadon, Chris Mason, Tejasvi Chakravarthy, Bharath Muthiah, and Sriram Sankar. 2021. Silent data corruptions at scale. (2021). https://doi.org/10.48550/arXiv.2102.11245 arXiv:arXiv:2102.11245

[21] J.-D. Guerrero Balaguera, J. E. R. Condia, F. Fernandes Dos Santos, M. Sonza Reorda, and P. Rech. 2023. Understanding the effects of permanent faults in GPU's parallelism management and control units. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '23).* 14 pages. https://doi.org/10.1145/3581784.3607086

[22] Juan-David Guerrero-Balaguera, Luigi Galasso, Robert Limas Sierra, Ernesto Sanchez, and Matteo Sonza Reorda. 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

[23] Juan-David Guerrero-Balaguera, Robert Limas Sierra, and Matteo Sonza Reorda. 2022. Effective fault simulation of GPU's permanent faults for reliability estimation of CNNs. In *2022 IEEE 28th International Symposium on On-Line Testing and Robust System Design (IOLTS)*. 1–6. https://doi.org/10.1109/IOLTS56730.2022.9897823

[24] Jin-Woo Han, M. Meyyappan, and Jungsik Kim. 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

[25] Siva Kumar Sastry Hari, Timothy Tsai, Mark Stephenson, Stephen W. Keckler, and Joel Emer. 2017. SASSIFI: An architecture-level fault injection tool for GPU application resilience evaluation. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 249–258. https://doi.org/10.1109/ISPASS.2017.7975296

[26] William Grant Hatcher and Wei Yu. 2018. A survey of deep learning: Platforms, applications and emerging research trends. *IEEE Access* 6 (2018), 24411–24432. https://doi.org/10.1109/ACCESS.2018.2830661

[27] Younis Ibrahim, Haibin Wang, Man Bai, Zhi Liu, Jianan Wang, Zhiming Yang, and Zhengming Chen. 2020. Soft error resilience of deep residual networks for object recognition. *IEEE Access* 8 (2020), 19490–19503. https://doi.org/10.1109/ACCESS.2020.2968129

[28] IEEE. 2022. The International roadmap for devices and systems: 2022. In *Institute of Electrical and Electronics Engineers (IEEE)*.

[29] Maha Kooli, Firas Kaddachi, Giorgio Di Natale, and Alberto Bosio. 2016. Cache- and register-aware system reliability evaluation based on data lifetime analysis. In *2016 IEEE 34th VLSI Test Symposium (VTS)*. 1–6. https://doi.org/10.1109/VTS.2016.7477299

[30] Lornatang. 2024. YOLOv3-PyTorch. https://github.com/Lornatang/YOLOv3-PyTorch. [Online; accessed 27-August-2024].

[31] Shyue-Kung Lu, Cheng-Ju Tsai, and Masaki Hashizume. 2015. Integration of hard repair techniques with ECC for enhancing fabrication yield and reliability of embedded memories. In *2015 IEEE 24th Asian Test Symposium (ATS)*. 49–54. https://doi.org/10.1109/ATS.2015.16

[32] Abdulrahman Mahmoud, Neeraj Aggarwal, Alex Nobbe, Jose Rodrigo Sanchez Vicarte, Sarita V. Adve, Christopher W. Fletcher, Iuri Frosio, and Siva Kumar Sastry Hari. 2020. PyTorchFI: A runtime perturbation tool for DNNs. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. 25–31. https://doi.org/10.1109/DSN-W50199.2020.00014

[33] Mithun Mukherjee, Vikas Kumar, Ankit Lat, Mian Guo, Rakesh Matam, and Yunrong Lv. 2020. Distributed deep learning-based task offloading for UAV-enabled mobile edge computing. In *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. 1208–1212. https://doi.org/10.1109/INFOCOMWKSHPS50562.2020.9162899

[34] Hyunjun Mun, Seonggwan Seo, and Joobeom Yun. 2021. Recycling of adversarial attacks on the DNN of autonomous cars. In *2021 International Conference on Information Networking (ICOIN)*. 814–817. https://doi.org/10.1109/ICOIN50884.2021.9333975

[35] Anouar Nechi, Lukas Groth, Saleh Mulhem, Farhad Merchant, Rainer Buchty, and Mladen Berekovic. 2023. FPGA-based deep learning inference accelerators: Where are we standing? *ACM Trans. Reconfigurable Technol. Syst.* 16, 4, Article 60 (Oct. 2023), 32 pages. https://doi.org/10.1145/3613963

[36] Biagio Peccerillo, Mirco Mannino, Andrea Mondelli, and Sandro Bartolini. 2022. A survey on hardware accelerators: Taxonomy, trends, challenges, and perspectives. *Journal of Systems Architecture* 129 (2022), 102561. https://doi.org/10.1016/j.sysarc.2022.102561

[37] Ratheesh Ravindran, Michael J. Santora, and Mohsin M. Jamali. 2021. Multi-object detection and tracking, based on DNN, for autonomous vehicles: A review. *IEEE Sensors Journal* 21, 5 (2021), 5668–5677. https://doi.org/10.1109/JSEN.2020.3041615

[38] Brandon Reagen, Udit Gupta, Lillian Pentecost, Paul Whatmough, Sae Kyu Lee, Niamh Mulholland, David Brooks, and Gu-Yeon Wei. 2018. Ares: A framework for quantifying the resilience of deep neural networks. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1109/DAC.2018.8465834

[39] Joseph Redmon. 2013–2016. Darknet: Open Source Neural Networks in C. http://pjreddie.com/darknet/

[40] Annachiara Ruospo, Angelo Balaara, Alberto Bosio, and Ernesto Sanchez. 2020. 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)*. 1–6. https://doi.org/10.1109/DFT50435.2020.9250866

[41] Annachiara Ruospo, Alberto Bosio, Alessandro Ianne, and Ernesto Sanchez. 2020. Evaluating convolutional neural networks reliability depending on their data representation. In *2020 23rd Euromicro Conference on Digital System Design (DSD)*. 672–679. https://doi.org/10.1109/DSD51259.2020.00109

[42] A. Ruospo, G. Gavarini, C. de Sio, J. Guerrero, L. Sterpone, M. Sonza Reorda, E. Sanchez, R. Mariani, J. Aribido, and J. Athavale. 2023. Assessing convolutional neural networks reliability through statistical fault injections. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 1–6. https://doi.org/10.23919/DATE56975.2023.10136998

[43] Annachiara Ruospo and Ernesto Sanchez. 2021. On the reliability assessment of artificial neural networks running on AI-oriented MPSoCs. *Applied Sciences* 11, 14 (2021). https://doi.org/10.3390/app11146455

[44] Annachiara Ruospo, Ernesto Sanchez, Lucas Matana Luza, Luigi Dilillo, Marcello Traiola, and Alberto Bosio. 2023. A survey on deep learning resilience assessment methodologies. *Computer* 56, 2 (2023), 57–66. https://doi.org/10.1109/MC.2022.3217841

[45] Annachiara Ruospo, Ernesto Sanchez, Marcello Traiola, Ian O'Connor, and Alberto Bosio. 2021. Investigating data representation for efficient and reliable convolutional neural networks. *Microprocessors and Microsystems* 86 (2021), 104318. https://doi.org/10.1016/j.micpro.2021.104318

[46] Mehdi Sadi and Ujjwal Guin. 2022. Test and yield loss reduction of AI and deep learning accelerators. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 1 (2022), 104–115. https://doi.org/10.1109/TCAD.2021.3051841

[47] Fernando Fernandes dos Santos, Siva Kumar Sastry Hari, Pedro Martins Basso, Luigi Carro, and Paolo Rech. 2021. Demystifying GPU reliability: Comparing and combining beam experiments, fault simulation, and profiling. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 289–298. https://doi.org/10.1109/IPDPS49936.2021.00037

[48] Fernando Fernandes dos Santos, Angeliki Kritikakou, Josie E. Rodriguez Condia, Juan-David Guerrero-Balaguera, Matteo Sonza Reorda, Olivier Sentieys, and Paolo Rech. 2023. Characterizing a neutron-induced fault model for deep neural networks. *IEEE Transactions on Nuclear Science* 70, 4 (2023), 370–380. https://doi.org/10.1109/TNS.2022.3224538

[49] Fernando Fernandes dos Santos, Pedro Foletto Pimenta, Caio Lunardi, Lucas Draghetti, Luigi Carro, David Kaeli, and Paolo Rech. 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

[50] Dimitris Sartzetakis, George Papadimitriou, and Dimitris Gizopoulos. 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

[51] Christoph Schorn, Andre Guntoro, and Gerd Ascheid. 2018. Accurate neuron resilience prediction for a flexible reliability management in neural network accelerators. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 979–984. https://doi.org/10.23919/DATE.2018.8342151

[52] C. Silvano, D. Ielmini, F. Ferrandi, L. Fiorin, S. Curzel, L. Benini, F. Conti, A. Garofalo, C. Zambelli, E. Calore et al. 2023. A survey on deep learning hardware accelerators for heterogeneous HPC platforms. https://doi.org/10.48550/arXiv.2306.15552 arXiv:arXiv:2306.15552

[53] Adit Singh, Sreejit Chakravarty, George Papadimitriou, and Dimitris Gizopoulos. 2023. Silent data errors: Sources, detection, and modeling. In *2023 IEEE 41st VLSI Test Symposium (VTS)*. 1–12. https://doi.org/10.1109/VTS56346.2023.10139970

[54] Fei Su, Chunsheng Liu, and Haralampos-G. Stratigopoulos. 2023. Testability and dependability of AI hardware: Survey, trends, challenges, and perspectives. *IEEE Design & Test* 40, 2 (2023), 8–58. https://doi.org/10.1109/MDAT.2023.3241116

[55] Cesar Torres-Huitzil and Bernard Girau. 2017. Fault and error tolerance in neural networks: A review. *IEEE Access* 5 (2017), 17322–17341. https://doi.org/10.1109/ACCESS.2017.2742698

[56] Timothy Tsai, Siva Kumar Sastry Hari, Michael Sullivan, Oreste Villa, and Stephen W. Keckler. 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

[57] Sotiris Tselonis, Vasilis Dimitsas, and Dimitris Gizopoulos. 2013. The functional and performance tolerance of GPUs to permanent faults in registers. In *2013 IEEE 19th International On-Line Testing Symposium (IOLTS)*. 236–239. https://doi.org/10.1109/IOLTS.2013.6604089

[58] Jeff Jun Zhang, Tianyu Gu, Kanad Basu, and Siddharth Garg. 2018. Analyzing and mitigating the impact of permanent faults on a systolic array based neural network accelerator. In *2018 IEEE 36th VLSI Test Symposium (VTS)*. 1–6. https://doi.org/10.1109/VTS.2018.8368656

[59] Rui Zhao, Ruqiang Yan, Zhenghua Chen, Kezhi Mao, Peng Wang, and Robert X. Gao. 2019. Deep learning and its applications to machine health monitoring. *Mechanical Systems and Signal Processing* 115 (2019), 213–237. https://doi.org/10.1016/j.ymssp.2018.05.050

Received 19 September 2024; revised 12 December 2024; accepted 16 January 2025