Evaluating the impact of Permanent Faults in a GPU running a Deep Neural Network

Evaluating the impact of Permanent Faults in a GPU running a Deep Neural Network

Juan-David Guerrero-Balaguera*, Luigi Galasso*, Robert Limas Sierra†, Ernesto Sanchez* Matteo Sonza Reorda*
*Politecnico di Torino - Department of Control and Computer Engineering (DAUIN)
†Universidad Pedagogica y Tecnologica de Colombia (UPTC) - Electronic Engineering School

***Abstract*—Currently, Deep Neural Networks (DNNs) are fundamental computational structures deployed in a wide range of modern application domains (e.g., data analysis, healthcare, automotive, robotics). The computational complexity is inherent in these cognitive models, which demand high-performance devices like Graphics Processing Units (GPUs). Therefore, the implementation of DNNs on GPU devices is becoming increasingly frequent, even for cutting-edge safety-critical applications (e.g., autonomous and semi-autonomous cars). Thus, the reliability evaluation of these applications is mandatory because several phenomena (including aging) may produce permanent defects in the GPU, thus inducing the DNN to produce wrong results. Until now, the effects of permanent faults on DNNs have been mainly investigated at the application level, only, e.g., acting on the parameters of the network. This paper presents an environment allowing for the first time a more detailed experimental evaluation of the impact of permanent faults in a GPU on the reliability of a DNN running on it, based on considering faults at the architectural level. The results of the fault injection campaigns we performed on the GPU register files are compared with those at the application level, proving that the latter ones are generally optimistic.**

***Index Terms*—Artificial Neural Networks, Deep Neural Networks, Graphics Processing Units (GPUs), Reliability evaluation.**

I. Introduction

Nowadays, Deep Neural Networks (DNNs) are widely used in numerous application areas such as multimedia, healthcare, robotics, and automotive [1]. Image and video processing are the primary fields where the DNNs support intelligent systems that can recognize objects and make decisions based on the environment's information. Lately, Convolutional Neural Networks (CNNs), which are a type of DNNs, have gained importance in safety-critical systems, such as self-driving vehicles [1]–[3]. The reliability estimation of these intelligent systems is crucial since they must meet the requirements set by safety standards (e.g., ISO26262 for the automotive domain).

The inherent computational complexity of modern DNNs demands high-performance hardware accelerators to fulfill the application's constraints. Among the wide variety of hardware accelerators [2], [4], [5], Graphics Processing Units (GPUs) have become the dominant devices to support the implementation and acceleration of DNNs-based applications.

The GPU popularity is mainly due to their flexible architecture combined with their computational power and scalability. Companies have also developed specialized ecosystems that provide hardware and software solutions increasingly adopted by cutting-edge safety-critical applications such as autonomous driving systems [6]. Consequently, the reliability evaluation of the DNNs executed on GPUs warrants special attention considering their growing popularity in safety-critical applications domains.

Historically, the DNNs are known to be resilient to errors mainly due to their redundant architecture and connections structure, which allow them to tolerate some level of neuron faults or noise corrupting the input data during the computational process [7], [8]. However, the operational conditions of the application may induce aging of the GPU device (e.g., due to the wear-out produced by long operational times), which can produce permanent defects forcing the DNN model to produce wrong inference results [9]. Thus, Permanent Faults (PFs) in the GPU device may endanger the DNN's proper operation affecting its reliability and possibly producing catastrophic results during its operational life.

The resiliency assessment of neural networks to PFs mainly resorts to Fault Injection (FI) campaigns at different abstraction levels, from the application level down to the hardware one [10], [11]. So far, the application-level FI campaign is the primary methodology employed to investigate the impact of PFs on DNNs. This approach only targets the Neural network parameters (e.g., weights or activation functions) [8], [9], [12], [13]. Unfortunately, the degree of accuracy at the application level differs by far from a real scenario considering faults in the physical hardware employed to execute the Neural Network. In fact, the application level can only mimic PFs in the system memory but it does not allow to consider faults in hardware elements such as register files, functional units, and control units.

On the other hand, Fault Injection of PFs at the hardware abstraction level involves injecting permanent defects (e.g., stuck-at-0/1) in the gate-level circuit description. Then, in theory, the faulty hardware simulates the execution of the DNN on the GPU in order to find the final effect of each evaluated fault. Unfortunately, the lower the hardware abstraction level, the higher the simulation time required by the FI campaign. Although the gate-level FI is more accurate, the size of the hardware accelerator (million of gates for GPUs) and of the DNN architecture (tens of layers and hundreds of millions of parameters) generate an excessive simulation time requirement (e.g., $>$ 10,000 days for thousand faults on an RT-level GPU model running LeNet). These off-limits simulation times

prevent the possibility of performing reliability evaluation at this or lower hardware levels. [10]. Consequently, we need to explore further alternatives (e.g., combining different abstraction levels) to find a trade-off between the simulation time and the accuracy of the analysis of the impact of the faults.

An alternative solution to be considered relies on architectural-level fault injections using the Hardware Injection Through Program Transformation (HIPT) technique [11], [14], [15]. This strategy modifies the application's source code at the ISA level to mimic the fault effect. Then, the modified program is executed on a real GPU, and the possible fault effects are propagated at hardware device speed.

In the reported literature, the HIPT method has been investigated for estimating the reliability of the GPUs with respect to transient faults, only [16]–[19]. However, modeling a PF at the architectural level through HIPT for GPUs implies a challenging endeavor because the fault effect must be described in such a way that it persists during the application's execution, affecting only some of the parallel threads that share the faulty GPU's resources.

This work presents for the first time a solution allowing the reliability evaluation of DNNs with respect to PFs at the architectural level and compares its results against the results of the FIs performed at the application level. The architectural fault injector is a customized binary instrumentation tool designed to conduct permanent FIs on the register files of the GPUs. The tool bases its operation on the mechanism implemented by the NVBitFI tool [14], which was initially developed for transient faults injections. Our Fault Injection environment permits the evaluation of the register sensitivity to faults and the assessment of their impact on the reliability of a DNN within reasonable simulation times (e.g., $\approx$ 12 hours for seven thousand faults evaluating the LeNet model). Additionally, the environment allows a bit-oriented analysis to identify the group of bits in the registers which are most prone to threaten the reliability of the DNN.

The experimental results indicate that FI at the application level is not enough to assess the reliability of DNNs, since this type of FI does not reveal the real impact of faults at the hardware level, thus producing incorrect and optimistic results.

The rest of the paper is organized as follows. Section II introduces the essential background. Section III describes the fault injection methodology. Section IV provides the fault injection results and presents a brief discussion about the reliability evaluation of DNNs using application and architectural fault injection levels. Finally, Section V draws some conclusions and outlines future research work.

II. Background

A. Deep Neural Networks

A Deep Neural Network is an Artificial Neural Network composed of multiple layers between the input and output layers. Convolutional Neural Networks (CNNs) are one class of DNNs. CNN's architecture mimics the pattern of neuronal connections in the human brain's visual cortex. A CNN comprises an input layer, several hidden layers, and an output layer. Unlike other DNNs, the hidden layer of a CNN performs a convolution operation between the filter elements and the inputs. For this purpose, the input corresponds to a two-dimensional matrix (i.e., an image). The filter or kernel 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. The most common pooling operation is the max-pooling operation. Additionally, each layer of the CNN contains a non-linear activation function to limit the output value of neurons. Finally, the output layers of the CNN employ a fully connected layer to perform the classification task [20] [7]. Well-known CNN architectures include LeNet, AlexNet, VGGNet, GoogLeNet, ResNet, and DenseNet.

B. Graphics Processing Units (GPUs)

Graphic 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 in a hierarchical structure. The SM is the primary execution unit in the GPU, which encompasses several independent sub-cores (up to four for modern GPU devices). Each SM sub-core comprises several parallel processing cores known as Stream Processors (SP), Special Function Units (SFU), and Tensor Cores Units (TCU). The SP supports integer and floating-point operations, 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, a SM includes local memories and register file banks to support the parallel execution of several threads. The SM core in a GPU performs Single-Instruction Multiple-Tread (SIMT) scheduling of a *Warp* (i.e., one SIMT group of 32 threads). Each SM sub-core scheduler issues one Warp instruction per clock.

C. Hardware Injection Through Program Transformation

Hardware Injection Through Program Transformation (HIPT) is a fault injection technique that reproduces at software level errors produced by a fault at the hardware level. This approach uses a software-level abstraction of fault models to inject software errors while it runs on the device or by modifying programs before their execution. This fault injection method does not need any hardware modification, and the fault propagation is performed at device speed. The implementation of the HIPT technique resorts to specialized tools that automatically allow modifying the application's source code or the insertion of instrumentation functions to reproduce the fault behavior [11].

In the case of NVIDIA GPUs, NVBitFI is the state-of-the-art error injecting tool, based on the HIPT approach, that instruments the target program to inject errors and propagate

them using a real GPU device. Additionally, this tool can instrument unknown libraries during compilation since the instrumentation process is performed directly on the CUDA executable at the SASS abstraction level. NVBitFI mainly provides support to perform transient fault injections using different fault models. Despite the fact that this tool includes a simple implementation of a Permanent Faults injector, the tool can be improved to incorporate other fault models, such as the popular stuck-at model [14].

III. Fault Injection Methodology

This work presents an environment based on a binary instrumentation tool able to perform FI campaign of PFs on a GPU device during the inference of DNNs. The environment allows evaluating the reliability of any DNN architecture considering the presence of faults at the hardware level rather than the usual high-level approaches that only consider faults affecting the parameters of the neural network.

The framework includes four building blocks: (1) profiler, (2) fault list generator, (3) fault injector, and (4) fault classifier. Additionally, a global controller manages each building block to control the fault injection process considering PFs that occur during the inference phase of the DNNs. The FI requires three steps: *i)* the fault list generation, *ii)* the golden model generation, and *iii)* the fault injection process.

The first step (fault list generation) performs profiling of the DNN to gather the execution information, such as the number of SMs used, of Threads per Kernel, of Registers per Kernels, as well as the opcodes of the instructions used to deploy the Neural Network on the GPU. Thenceforth, the collected information allows the generation of valid faults. In the second step, the global controller initiates the inference of the DNN considering the fault-free scenario, which serves as a reference to evaluate the results produced by faults on the DNN model. Finally, in the last step, each fault (identified in the first step) is injected and propagated through all executed kernels mimicking its effects in the GPU. The results of the faulty inference of the DNN are collected and compared against the reference model. This comparison allows the classification of the fault according to its severity impact on the DNN results.

A. Fault classification

The accuracy is a usual metric employed to measure the neural network generalization capabilities. This metric can be calculated as the number of correct predictions divided by the total number of input evaluated images. A Permanent Fault at the hardware level may produce different possible effects during the inference of a DNN affecting its accuracy in the most critical cases. To quantify the damage produced by a Permanent Fault, we define the Relative Accuracy Degradation as $RAD = (ACC_{gold} - ACC_{faulty})/ACC_{gold}$, where ACC_{gold} and ACC_{faulty} indicate the classification accuracies of the *fault-free* and the *faulty* models, respectively. Usually, the PFs classification falls into three main categories: Silent Data Corruption (SDC), Detected Unrecoverable Error

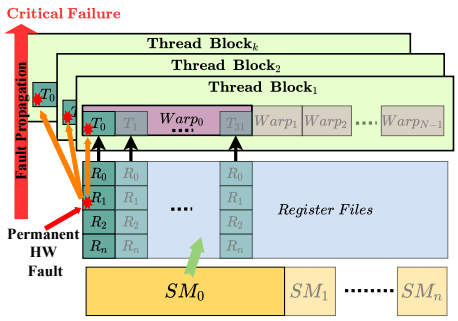


Fig. 1. Propagation of PFs from GPU register files to the application level.

(DUE), Masked. In this work, we consider four possible fault categories as follows:

- **Masked**: $RAD = 0.0$. No difference is observed between the faulty scenario and the golden one.
- **SDC-safe**: $RAD = 0.0$. The confidence prediction values for at least one image differ from the fault-free scenario but the classification is still correct
- **SDC-Critical**: $RAD > 0.0$. At least one image was wrongly classified with respect to the fault-free scenario
- **DUE**: The fault produces a system hang or crash. This error interrupts the execution of the CNN at any time. The causes of this behavior can be memory access violation, memory misalignment violation, or timeout (the fault stucks the CNN model in an infinite loop).

B. Permanent fault injection

The fault injector module resorts to the HIPT technique to mimic the Permanent Fault presence in the register files of the GPU. Nonetheless, the same concepts presented here can be expanded to other internal modules of the GPU. The modeling of a PF in the register files considers the parallel execution model used by GPU and the possible effect that the fault might produce when propagates through the application.

Each kernel executed by the GPU is composed of several threads 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. Each thread block is then issued in the *SM* when the previous block finishes and releases its resources. Each thread inside each warp has access to a private set of registers to support the *SIMT* parallel execution model. Clearly, any PF affecting the register files may induce errors during the execution of threads that share the same faulty hardware.

Fig. 1 illustrates the relationship between a fault in the registers of the GPU and its propagation through the application. It is worth noting that one fault can affect more than one thread, mainly when they belong to different blocks executed by the same SM. Therefore, the fault injector tool requires an appropriate definition of fault to meet the behavior mentioned above. We define the fault location as the quintuple <*SMID*, *threadID*, *RegisterID*, *Mask*, *stuck-at*>. *SMID* represents the *SM* where the fault should be injected; *threadID* is the resident thread in the *SM*: this allows to identify a unique *WarpID* and *LaneID*; *RegisterID* is the faulty target register; *Mask* is the

Algorithm 1 Permanent fault injection adopting the HIPT FI campaign technique in GPU register files

Input: Fault F_i defined by: $< SM_{ID}, Thrd_{ID}, Reg_{ID}, Mask, ST@ >$
Output: Application output affected by the fault F_i

```
for each kernel K_i in the DNN model do
    for each instruction I_j in K_i do
        Inspection(I_j)
        if R_d in I_j matches the target Reg_ID then
            Insert injection function after I_j
        end if
    end for
    Just In Time compilation
    Instrumented kernel execution
end for
```

bit location inside the target register; *stuck-at* represents the type of the fault (0 or 1) according to the stuck-at fault model.

Algorithm. 1 describes the mechanism we devised to mimic the permanent effect of the fault during the DNN's execution. First, the tool works on each kernel and modifies its assembly source code, inserting an instrumentation function right after the instructions whose destination register corresponds to the fault location. The instrumentation function consists of reading the content of the target register (in a Warp and Thread) and then modifying one of its bits by forcing it to 0 or 1 to emulate the permanent characteristics of the fault. Finally, when the kernel is totally instrumented, the tool performs the Just-in-Time compilation to update the binary representation of the new kernel version. After that, the tool resumes the execution of the application and submits the instrumented kernel on the GPU device.

IV. Experimental Results

In this section we present the results obtained by the proposed fault injection approach in order to evaluate the reliability of different DNN architectures. Four pre-trained DNN models were employed for the experiments: LeNet, AlexNet, Darknet19, and VGG-16. The LeNet model can classify images of handwritten digits (0 to 9) using the MNIST dataset, AlexNet and Darknet19 classify images from 1,000 categories from the ImageNet dataset, and VGG-16 classifies images from 10 different classes defined by the CIFAR-10 dataset. The implementation of each DNN resorts to the darknet [21] environment which support GPUs acceleration. These fault injection campaigns were performed on a workstation HP Z2 G5 with an Intel Core i9-10800 CPU with 20 cores, 32 GB of RAM memory, and equipped with an RTX 3060TI GPU platform including an NVIDIA Ampere architecture with compute capability (CC) 8.6.

The Fault Injections are performed at two different levels of abstractions: the architectural level and the application level. The fault injection at the architectural level implements the methodology proposed in this work by modifying the NVBitFI tool in order to support the modeling of PFs on the register files of the GPU using the stuck-at fault model.

The universe of permanent faults to be considered during the fault injection campaign may be excessive due to the complexity of the DNN and the number of fault locations at the application and architectural level. In the case of a PF in the weights, the number of faults can exceed 3,000 million for AlexNet. Similarly, the number of faults in the register files of a single SM of the GPU can exceed 5 million for the Ampere architecture. Thus, exhaustive fault simulations in all possible fault locations are impractical due to the time and complexity required to evaluate the entire set of possible faults.

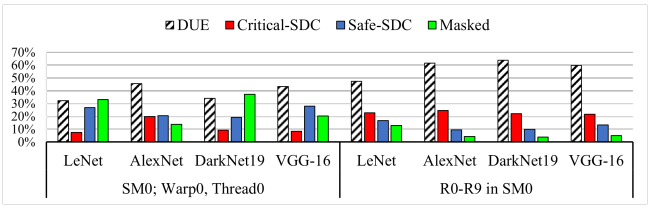


Fig. 2. Fault classification results: architectural FIs.

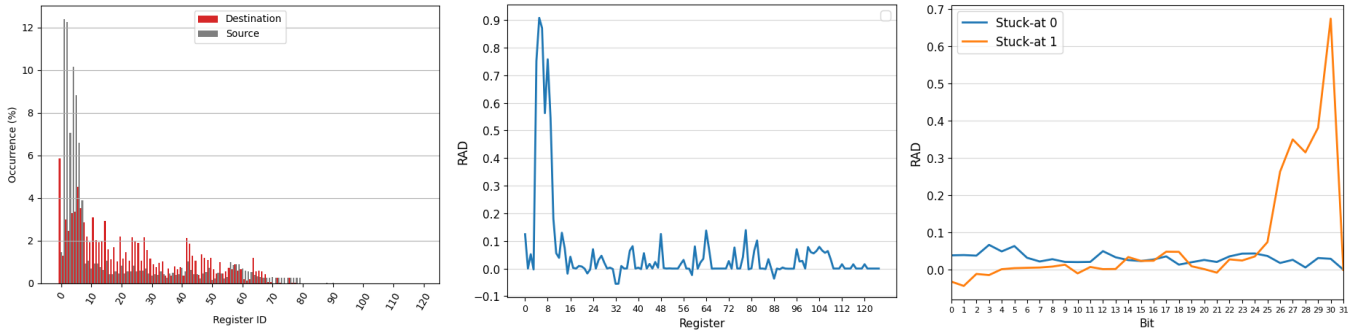
TABLE I
NUMBER OF FAULTS CONSIDERED FOR EACH FI

FI Campaign	LeNet	AlexNet	DarkNet19	VGG-16
SM0, Warp0,Thread0	7,233	7,296	10,945	7,233
R0-R9 in SM0	16,410	16,545	16,410	16,547
Weights	62,435	132,412	306,750	260,869

This work performs first an exhaustive fault injection targeting only the registers used by one of the resident threads in the SM0. After that, an extended fault simulation is performed, resorting to a fault sampling methodology exposed by [9], [22], using 99% of confidence level and 1% of margin error. This sampling approach is applied at the application level (i.e., faults in the DNN weights) and at the architectural level (i.e., targeting the first ten registers of each resident thread in the SM0). This last fault simulation considers the first ten registers per thread since the profiling tool reports that this subset of registers is the most frequently used during the CNNs inference. The number of faults considered for each FI campaign are presented in Table I. The fault simulation lasts around 107 hours for all DNN models.

Fig. 2 depicts the fault classification results of the FICs at architectural level. When the FI campaign considers all registers in one resident thread ($SM_0, Warp_0, Thread_0$), the results show that between 32% to 45% of faults hang the GPU device (*DUE*), preventing the DNN complete execution. Furthermore, the number of faults inducing wrong results (*Critical-SDC*) for LeNet, Darknet19, and VGG-16 does not exceed 10%, only for AlexNet the number of these faults reaches almost 20%. Between the 20% and 37% of the faults induce tolerable results (*Safe-SDC*), and less than 40% of faults does not have any impact in the DNN's inference (*Masked*).

When we consider faults distributed in the first ten registers of one SM, the results report up to 60% of *DUE* faults. Additionally, a significant portion of faults ($> 20\%$) are considered *critical-SDC*. Faults classified as *Masked* do not exceed the 5% for AlexNet, Darknet19, and VGG-16, and no more than 13% in the case of LeNet.



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

Fig. 3. Sensitivity of GPU's Registers to Permanent Faults in the GPU running AlexNet

A. Register sensitivity to PFs

Permanent faults in the register files produce different effects on the DNN's outputs. Some registers are more sensitive to faults than others due to their usage inside the application. Fig. 3.a depicts the register usage for AlexNet. The chart shows the occurrence percentage of each register when it is used as the source (gray color) and destination (red color). It is worth noting that the evaluated DNN models have the same registers usage characteristics where the first ten registers are the most commonly used.

Fig. 3.b depicts the impact of PFs in the registers (DUE 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 chart, we can observe that the registers from R3 to R11 are the most critical ones, producing more than 50% of accuracy degradation and reaching up to 90% for R8. The accuracy degradation in the other registers is uniform, and does not exceed 15%. Despite the fact that many registers have less than 20% of accuracy degradation due to PFs, this is still a high percentage of critical effects that create risky results for any application running these DNN models.

The first ten registers of each thread in the GPU are more sensitive to faults because they are commonly used for the CUDA thread indexing at the beginning of each kernel. These registers contain the *threadID* and *blockID* parameters to identify the operations of individual threads and memory addresses. Although this is the primary use of these registers, the compiler also reuses them during the execution of elaborated algorithms such as matrix and vector procedures. The majority of faults affecting these registers most likely generate *DUE* effects due to memory addressing violations. Therefore, only a few faults produce effects on the DNN's outputs, modifying the threads or block identifiers without crashing the application but generating dangerous results.

B. Bit-oriented Registers sensitivity to PFs

The computation of the mean accuracy degradation considers the bit position of the faults inside any register in order to evaluate its impact on the classification result of the DNN. This analysis resorts to the exhaustive set of faults targeting the registers of one allocated thread on the SM0.

Fig. 3.c illustrates the bit-oriented accuracy degradation for the AlexNet model produced by stuck-at-0/1 faults. Although we introduce the results for AlexNet in order to simplify the presentation of the results, it is important to highlight that we found similar results for all other evaluated DNNs. The results consider only the effect of the faults propagated to the DNN' output. The DUEs are discarded from the analysis since those faults do not generate a valid DNN inference.

The obtained results show that the propagation effect of stuck-at-0 faults does not exceed the 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 suggest that stuck-at 1 faults located in the exponent bits produce higher MRAD, because the representation of the floating-point value grows several orders of magnitude compared to the original value. This yields wrong results during the computation and surely wrong prediction results. These results are aligned with results presented by other works such [8], [9], in which the injection of PFs at the application level indicates a high sensitivity on the bit 30th. Although our results confirm the previous findings, they demonstrate that more bits further than 30th are susceptible to permanent faults at the hardware level inside the GPU device. These results also indicate that the fault injection at the application level targeting only the parameters of the DNN does not reveal the actual vulnerabilities of the hardware when executing the DNN.

In the previous subsections, we showed that many of the faults may produce a Silent Data Corruption (SDC) effect on the outputs of the DNN. These faults are dangerous for any applications running a DNN because their presence may induce wrong decisions in the final applications without any way for knowing that the system is faulty. Therefore, we compare the effect of PFs at the application and architectural level when different DNNs are executed on a GPU device. The results show that in a more realistic hardware fault scenario (architectural-level FI), the number of faults that induce wrong effects is significantly higher than when a high-level fault simulation scenario is evaluated (application-level FI).

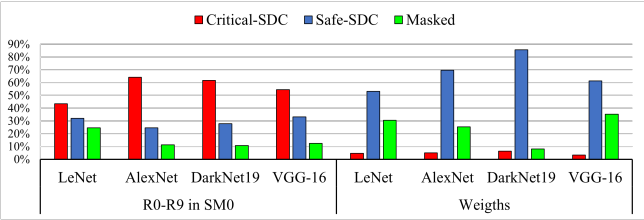


Fig. 4. Fault classification results: architectural- and application- level FI campaigns.

C. Application vs. architectural fault injection campaigns

Fig. 4 presents the classification results of faults at application level and architectural level. The fault classified as DUE are not considered in the analysis, because they do not allow to generate a valid inference result. The results demonstrate that up to 64% of the faults produce *critical-SDC* for AlexNet. The faults that generate a safe impact do not exceed 33.15% for VGG-16, and the masked ones do not exceed 12% for AlexNet, Darkenet19, and VGG-16, except for LeNet, where the percentage of masked faults reaches 24%. The results of the FI campaign at the application level show optimistic results about the reliability of the DNN with respect to PFs. No more than 7% of the faults produce a critical impact on the prediction results of any DNN model. In contrast, 85.66% of faults may produce a tolerable result (safe-SDC) in the case of Darknet19. The masked faults do not exceed 37% for all evaluated DNNs.

According to these results, we can state that FI campaigns performed at the application level are more optimistic than the FI at the architectural level in terms of reliability assessment of Neural networks. We also demonstrate that PFs in the GPU device jeopardize the DNN application by more than 40% when compared to the reliability evaluation done by injecting faults in the parameters of the neural network.

V. Conclusions and Future Work

This paper presents a framework for studying the influence of permanent faults in a GPU device and their impact on the reliability of a Deep Neural Network. The characterization resorts on fault injection campaigns using a binary instrumentation approach specially designed to mimic the PF presence on the register files of a GPU device. The experiments are performed on different DNNs implemented through the darknet environment. For the first time we are able to evaluate the impact of PFs affecting the GPU register files on the executed DNN. We also demonstrate that faults in the different registers have a different impact on the reliability of the DNNs; in particular, the first ten registers are the most sensitive to PFs. Furthermore, faults located in the most significant bits of the registers have a higher impact on the reliability of the DNNs, especially the bits 25th to 30th, generating an accuracy degradation of the DNN up to 68%. Finally, our results show that the reliability evaluation of a DNN using an application-level FI campaign (i.e., injecting faults in the DNN weights) generates optimistic results. Using application-level FI, less than 7% of faults induce wrong classification results; when the faults are considered at the hardware level in the GPU, the number of faults that create a critical effect on the prediction result of the DNN can reach up to 64%. Future activities aim to extend the set of locations where to inject faults, to evaluate more CNN architecture models, to consider aditional GPU architectures, and to propose hardening techniques to counteract the vulnerabilities caused by PFs in CNN using GPUs.

References

[1] H. Mun, *et al.*, "Recycling of adversarial attacks on the dnn of autonomous cars," in *2021 International Conference on Information Networking (ICOIN)*, 2021, pp. 814–817.

[2] W. G. Hatcher and W. Yu, "A survey of deep learning: Platforms, applications and emerging research trends," *IEEE Access*, vol. 6, pp. 24 411–24 432, 2018.

[3] R. Ravindran *et al.*, "Multi-object detection and tracking, based on dnn, for autonomous vehicles: A review," *IEEE Sensors Journal*, vol. 21, no. 5, pp. 5668–5677, 2021.

[4] Y. Chen *et al.*, "A survey of accelerator architectures for deep neural networks," *Engineering*, vol. 6, no. 3, pp. 264–274, 2020.

[5] S. Mittal *et al.*, "A survey on hardware security of dnn models and accelerators," *Journal of Systems Architecture*, vol. 117, p. 102163, 2021.

[6] NVIDIA, "NVIDIA DRIVE End-to-End Solutions for Autonomous Vehicles," https://developer.nvidia.com/drive, 2022, [Online; accessed 21-April-2022].

[7] C. Torres-Huitzil and B. Girau, "Fault and error tolerance in neural networks: A review," *IEEE Access*, vol. 5, pp. 17 322–17 341, 2017.

[8] S. Hong *et al.*, "Terminal brain damage: Exposing the graceless degradation in deep neural networks under hardware fault attacks," in *28th USENIX Conference on Security Symposium*, ser. SEC'19. USA: USENIX Association, 2019, p. 497–514.

[9] A. Ruospo *et al.*, "Investigating data representation for efficient and reliable convolutional neural networks," *Microprocessors and Microsystems*, vol. 86, p. 104318, 2021.

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

[11] S. K. Bukasa *et al.*, "When fault injection collides with hardware complexity," in *Foundations and Practice of Security*. Cham: Springer International Publishing, 2019, pp. 243–256.

[12] A. Ruospo and E. Sanchez, "On the reliability assessment of artificial neural networks running on ai-oriented mpsocs," *Applied Sciences*, vol. 11, no. 14, 2021.

[13] A. Ruospo *et al.*, "Evaluating convolutional neural networks reliability depending on their data representation," in *2020 23rd Euromicro Conference on Digital System Design (DSD)*, 2020, pp. 672–679.

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

[15] S. K. S. Hari *et al.*, "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)*, 2017, pp. 249–258.

[16] T. Garrett and A. D. George, "Improving dependability of onboard deep learning with resilient tensorflow," in *2021 IEEE Space Computing Conference (SCC)*, 2021, pp. 134–142.

[17] Y. Ibrahim *et al.*, "Soft error resilience of deep residual networks for object recognition," *IEEE Access*, vol. 8, pp. 19 490–19 503, 2020.

[18] I. Younis *et al.*, "Soft errors in dnn accelerators: A comprehensive review," *Microelectronics Reliability*, vol. 115, p. 113969, 2020.

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

[20] H. S. Das and P. Roy, "Chapter 5 - a deep dive into deep learning techniques for solving spoken language identification problems," in *Intelligent Speech Signal Processing*, N. Dey, Ed. Academic Press, 2019, pp. 81–100.

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

[22] R. Leveugle *et al.*, "Statistical fault injection: Quantified error and confidence," in *2009 Design, Automation Test in Europe Conference Exhibition*, 2009, pp. 502–506.