Analyzing the Impact of Soft Errors in Deep Neural Networks on GPUs from Instruction Level

YOUNIS IBRAHIM
College of Internet of Things
Hohai University
Changzhou, China
Younis@hhu.edu.cn

JUNYANG LIU
College of Internet of Things
Hohai University
Changzhou, China
2418959620@qq.com

XUANXUAN YANG
College of Internet of Things
Hohai University
Changzhou, China
1862910201@hhu.edu.cn

HONGWEI SHA
College of Internet of Things
Hohai University
Changzhou, China
1602236595@qq.com

Peng Li
Shanghai Xieji Technology Company
Shanghai, China
lipeng0105@126.com

HAIBIN WANG
College of Internet of Things
Hohai University
Changzhou, China
wanghaibin@hhuc.edu.cn

Abstract: - Deep Neural Networks (DNNs) used in safety-critical systems cannot compromise their performance due to reliability issues. In particular, soft errors are the worst. Selective software-based protection solutions are among the best techniques to improve the reliability of DNNs efficiently. However, their most significant challenge is precisely hardening portions of the DNN model to avoid performance degradation. In this work, we propose a comprehensive methodology to analyze the reliability of object detection and classification algorithms run on GPUs from the lowest (instruction) evaluation level. The ultimate goal is to avoid the performance penalty of full instruction duplication by confidently identifying the vulnerable instructions. For this purpose, we propose a technique, Instruction Vulnerability Factor (IVF). By applying our methodology on ResNet and YOLO models, we demonstrate that both models' most vulnerable instructions can be precisely determined. Moreover, we show that YOLO is more sensitive to the changes caused by soft errors than ResNet. Also, ResNet depends on the input image in its reliability, while YOLO tends to be independent.

Key-Words: - Reliability, neural networks, GPUs, soft errors, instruction-level vulnerability, selective hardening

Received: June 3, 2020. Revised: October 20, 2020. Accepted: November 18, 2020. Published: December 8, 2020.

1 Introduction

The scope of what computers can learn has significantly been increased by Deep Neural Networks (DNNs). In particular, the field of computer vision is empowered by image classification [1] and object detection [2] techniques. The state-of-the-art You Only Look Once (YOLO) and Residual Networks (ResNet) have shown impressive performances for object detection and classification systems, respectively [2]. Moreover, YOLO is one of the fastest accurate detectors, while ResNet has been used in nineteen modern detectors as their backbone feature extractor [3]. Consequently, the past decade has witnessed the widespread adoption of DNNs in various domains. Numerous of these applications have been utilized in safety-critical areas, such as self-driving cars [4], [5], healthcare [6], [7], and in space [8]. The performance of the DNN models in such environments is no longer the only requirement, but also the reliability [9].

As the latency plays an essential role in safety-critical applications for real-time responses, it is a risk to run compute-intensive algorithms, such as DNNs, on regular CPUs. Therefore, massively parallel accelerators, such as Graphics Processing Unit (GPUs), should be used to tackle this bottleneck [9]. In fact, GPUs provide the lowest latency compared to other accelerators [10].

A member of the US Technical Advisory Group (TAG) to the ISO 26262, Kurt Shuler, has stated the importance of reliability in self-driving cars "*If we want to be serious about autonomous vehicles, the safety standard landscape needs to evolve beyond pass/fail checklists*" [11]. This statement indicates that, in critical domains, the model's performance cannot be compromised due to its reliability.

Many sources of faults cause unreliability for these DNN systems. However, transient hardware faults (i.e., soft errors) cause the majority of the system failures [12]. Transient faults are originated from various sources, such as high-energy particles, temperature, voltage variation, malicious attack, and clock skews [13]. The biggest concern about transient faults for DNNs is that they lead to incorrect predictions at the model's output. This type of error is called silent data corruptions (SDCs), which are usually not detectable by regular DNN algorithms, leading to misclassification or misdetection [14], [15].

A common approach to address soft error issues in GPUs is utilizing software-based solutions, especially selective software-based techniques. They allow hardening parts of the source code instead of full duplication of the executing programs. These strategies can tolerate transient faults at relatively lower costs. However, they incur excessive overheads that can severely harm the DNN model's performance. This is because prior selective hardening approaches have been conducted at different levels of abstraction, including the DNN layer's level [15], [16] and DNN kernel's level [14], [16]. To our best knowledge, DNN resilience from the assembly-language level has not been investigated on GPUs.

A first-class downside of the prior protection strategies is the degradation of performance that stems from the solutions themselves. For instance, self-driving cars must demonstrate the ability to respond within a few milliseconds (per iteration) before they are deployed on the road [10], [17]. Applying traditional hardening techniques to such systems is likely to impact the latency by delaying the response time, which is a risk. Therefore, our work aims to identify the portions of the DNN models that are the most sensitive to soft errors from the lowest evaluation level of software-based solutions, instead of high-level evaluations.

In this study, we present and evaluate a methodology to analyze the reliability of DNNs on GPUs at the assembly-language instruction level. The ultimate objective is to avoid unnecessary overheads that would be produced by various source-code level approaches. We consider the injection of transient fault to characterize its impact on two DNN techniques, image classification (ResNet) and object detection (YOLO). Adopting our methodology, we are able to precisely identify the vulnerable instructions of the ResNet and YOLO models that cause unreliability to the system. The main contributions of this paper are:

- A thorough methodology to analyze the probability of faults in the low-level instructions that probable to cause a failure in the model's prediction.
- Instruction Vulnerability Factor (IVF) and applying it to two case studies (ResNet and YOLO models) to calculate IVF according to the corrupted predictions.
- Conducting an extensive analysis of the two models through SASSIFI fault injection to characterize the error resilience behaviors of image classification and object detection systems against soft errors.
- Error criticality measurement to classify instructions that tend to produce critical SDCs and determine whether the critical error rate depends on the model's input.

The remainder of the paper is organized as follows. Section 2 serves as a background on the two models, transient errors, and the GPU architectures considered in our work; and reviews related work. Section 3 demonstrates our analysis methodology. Experimental results are presented and discussed in Section 4. Section 5 concludes the paper.

2 Background and Related Work

In this section, we first present a brief background on object recognition benchmarks (i.e., object classification and object). Then, an overview of transient errors in GPUs is presented. Also, GPU architectures used are presented. Last, we review related works.

2.1 Object Classification and Detection

DNNs are widely adopted in computer vision tasks. Object classification and detection are the two most efficient techniques of DNNs to extract meanings from images and videos for computer vision tasks [2]. Image classification is the process of taking the entire image and predicting which category this image belongs to. Object detection is the process of recognizing all the objects alongside their locations in a given image [3]. The significant difference is that the later technique deals with distinguishing between objects within the same image. This means that it is tasked to categorize and locate all known content in the scene. Thus, detection means classification plus localization.

In this study, we consider ResNet50 [18] as a classifier, and YOLO3 [19], as an object detector. These models were pre-trained on the ImageNet dataset and available on Darknet, an open-source DNN framework [20]. After executing ResNet and YOLO, they both predict a probability vector as a final output. Thus, if the probability exceeds the predetermined threshold (i.e., a precision value), objects are identified and classified. However, the fundamental difference between the two is that YOLO (for being a detector) goes beyond just classifying objects. It takes the

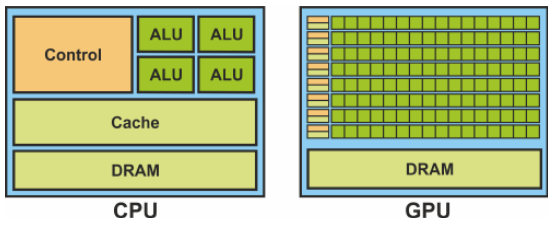


Fig. 1: Main differences between CPU and GPU architectures.

entire image as an input and learns the bounding box coordinates and their class probabilities. This means YOLO divides each image into a grid of S x S, and each grid predicts N bounding boxes and their confidence. The confidence reflects the accuracy (i.e., precision) of the bounding box and whether it contains an object (regardless of its class). Therefore, these models have different workflows to achieve their tasks.

When a transient fault modifies one of the probability values in that vector (in ResNet case), coordinates, or confidences (in YOLO case), the model may significantly suffer from precision degradation impacting the rank of the classified or detected object. Thus, it can lead to a wrong prediction. Hence, the reliability evaluation and behavior characterization for these DNNs are of importance.

2.2 Transient Fault

A transient fault is a temporary disturbance in low-level components of a device (e.g., flip-flop or SRAM) such as a GPU. As we demonstrated in our previous work [14], a transient fault can produce three types of errors (1) Silent Data Corruptions (SDCs), severely undermining system's reliability (i.e., wrong prediction); (2) Detectable Uncorrectable Error (DUE), leading to program's exit or hang without completing the current execution; (3) the error is masked without having any effect (Masked).

SDC errors are our primary concern in this study because any original DNN algorithm cannot detect them [9]. Hence, they can cause misclassification or misdetection. Also, prior studies have shown that transient faults can lead to significant performance losses or accuracy drops [13]-[16]. SDCs are evaluated by comparing the injected output with an error-free output version [12], [14].

2.3 Tested GPU Architectures

Since this paper focuses on investigating the resilience of low-level instructions, the specific architectures are necessary to be specified. NVIDIA GPUs are programmed by Compute Unified Device Architecture (CUDA), which is a programming model and extended version of the C language that contains both the host (i.e., CPU) code and device (i.e., GPU) code [21]. NVIDIA GPUs, in general, have significantly different architectures compared to CPU architectures, as illustrated in Fig. 1

As we are only concerned about the GPU part, it is worth mentioning that the CUDA source code is compiled with the NVCC, NVIDIA's front-end compiler [22]. Then, NVCC translates code written in CUDA into PTX (Parallel Thread Execution). Finally, the back-end compiler, PTX assembler (ptxas), translates the PTX into machine code, which can then be run on the CUDA processing cores [23]. PTX is NVIDIA's assembly and intermediate-level Instruction Set Architecture (ISA) that highly depends on the GPU's microarchitecture. Several microarchitecture generations are available, such as Tesla, Fermi, Kepler, Maxwell, Pascal, Volta, and Turing [24]. Each of which has different compute capabilities and static assembly instructions. At this point, the required notice is that the PTX code is required to be translated into a specific native-target hardware ISA before the actual execution. Also, the propagation of low-level faults is affected by microarchitectural divergences [15].

Therefore, specific microarchitectures should be used for a particular study. In this study, we target Maxwell and Pascal microarchitectures for two reasons: (1) because they share the same ISA [24]; and (2) they have been intensively used to accelerate deep learning models [3], [12], [25].

2.4 Related Work

The reliability of DNN algorithms on GPUs against soft errors has been widely studied. Unfortunately, the existing selective software-based approaches have been conducted at source code levels.

Santos *et al.* intensively analyzed and evaluated the reliability of different DNN algorithms on different GPUs [15]. Based on their analysis, the author found that on an average of 76% of the GPU operations for a DNN model are spent in matrix multiplications (MxM). Accordingly, they proposed a solution based on Algorithm-Based Fault-Tolerance (ABFT) to exploit the substantial utilization of MxM. However, this solution is applied at a higher level than the instruction level since it is an architectural-independent technique. Further, this study evaluated the vulnerability of Maxpooling layers and proposed a solution to be applied at the layer's level.

Hong *et al.* studied how soft errors in model parameters could affect the performance of the DNN model [13]. They conduct this study by changing the values of the parameters stored in a memory. Their proposed methodology has been applied in six DNN architectures. The authors demonstrate that the DNNs have limitations against parameter perturbations. They show that accuracy can drop by 99%.

Table 1: Low-level static instructions used by both models, grouped depending on their purposes.

LD_ST	FP	FxP		Conv	PR_CC	Ctlr	Misc
LDG	FADD	IADD	LOP32I	F2F	PSETP	BRA	MOV
STG	FADD32I	IADD3	SHL	F2I	ISETP	EXIT	SEL
	FFMA	IADD32I	SHR	I2F	FSETP	JCAL	S2R
	FMUL	ISCADD	SHF	I2I		SSY	
	FMUL32I	ISCADD32	XMAD			SYNC	
	FCMP	ISET				RET	
	FCHK	ICMP					
	MUFU	LOP					
	RRO	LOP3					

In our previous works [14], [16], [26], we have analyzed and evaluated DNN models' reliability, from two perspectives DNN kernel's and layer's levels, to identify the vulnerable kernels and layers of the DNN's source code. By implementing our strategy, we were able to determine the vulnerable portions confidently. Accordingly, selectively hardened the top critical portions (kernels or layers) to reduce potential overheads.

From the above-reviewed works, we notice that the fundamental difference between our work and its prior counterparts is the evaluation level. All prior studies apply their evaluation approaches "before" the compilation of the source code, while we target the PTX instructions, which "after" the compilation of the front-end compiler (NVCC). Therefore, to the best of our knowledge, our study is the first to characterize the reliability of DNNs on GPUs from the instruction level. We focus on low-level instructions to propose a selection-based strategy that can guide designers and developers to select the only vulnerable instructions for protection effortlessly. Hence, our strategy would lead to even lower overhead. We validate our methodology on image classifier (ResNet) and detector (YOLO) as our case studies.

3 Analysis Methodology

In this study, we aim to investigate the error resilience of ResNet and YOLO by performing a comprehensive fault-injection campaign into GPU's low-level instructions that execute these models. Hence, the main objective of this study is to identify the vulnerable instructions of each model.

3.1 Fault-injection Setup

To inject faults into NVIDIA's GPUs (i.e., Maxwell and Pascal architectures), we use a fault injector introduced by NVIDIA, called SASSIFI [27]. It is worth noting that SASSIFI allows researchers to choose where to inject faults and what type of faults to be injected, depending on the study the researchers will conduct. These two parameters (i.e., where and what) will form the fault model of our study.

For *where* to inject, two injection modes are available: Register file (RF) mode and Instruction Output

Table 2: The occurrence rate of the instruction groups.

Group	LD_ST	FxP	FP	PR_CC	Ctrl	Conv	Misc
ResNet	3.56	56.08	4.52	8.85	3.50	7.11	16.38
YOLO	3.72	56.27	4.40	9.28	3.45	6.88	16.00

(IO) mode. In our study, as we plan to analyze individual instructions' resilience, we only need IO mode to perform instruction output-level injections. Thus, faults are injected in the output of the currently executing instruction, which is randomly selected. The purpose is to examine the probability that the destination register's value or address (for the current instruction) is subjected to errors [27].

For *what* to be injected, SASSIFI provides several bit-flip models (BFMs). We choose random-value BFM because it represent both single and double BFMs.

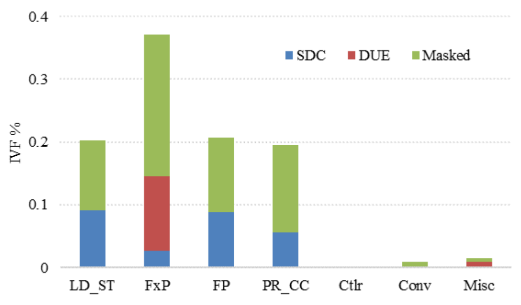
Once the above setup is finalized, we perform a fault-injection campaign of 1000 injections at each injection site with the given BFM. Then, we collect all the experimentally-obtained results and compare each with the model's golden (original) result and record the difference.

3.2 Instruction Groups

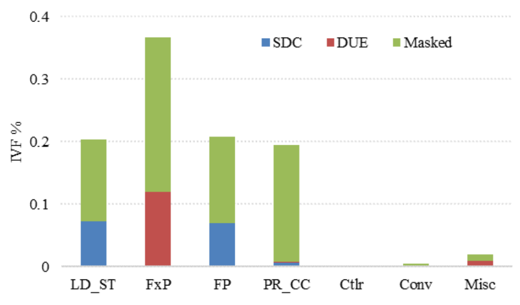
By implementing and profiling our two models (ResNet and YOLO) on the Darknet framework with SASSIFI, we found that each of them requires tens of low-level static instructions to be executed. Each of these instructions is used by thousands of threads to become a dynamic instruction per run. Hence, it is impractical to analyze every static instruction individually. Instead, it is more practical to organize these instructions into groups, then analyze and interpret the common and anomalies.

It should be mentioned that SASSIFI provides fixed instruction groups for evaluation, which are common to every parallel application run on it. However, those groups do not provide any exclusive insights about DNN algorithms, which have unique characteristics distinct from traditional applications [9], [10]. Therefore, we create our customized instruction groups based on the GPU microarchitectures and models considered in our study. The grouping is based on the purpose, where static instructions in the same group achieve one larger purpose [24]. As listed in Table 1, we distribute all the instructions used in our models into seven groups (the first row of Table 1):

- Load and Store (*LD_ST*) instructions: Are used to access memory elements for loading data from or storing data to.
- Floating-point (*FP*) instructions: Are used to execute Add and Multiply (MAD) operations, specifically for operating on single-precision



(a)



(b)

Fig. 2: The vulnerability analysis of instruction group for (a) ResNet model and (b) YOLO model.

(32-bit) and fused or non-fused floating-point numbers.

- Integer/Fixed-point (*FxP*) instructions: Are used to execute Add and Multiply (MAD) operations, specifically for operating on single-precision and fused or non-fused integer numbers.
- Conversion (*Conv*) instructions: Are used for special registers to convert a value with a particular type (i.e., *FP* or *FxP*) and length to another value with a different type and/or length.
- Predicate Registers and Condition Code (*PR_CC*) instructions: Are used to set special-purpose registers. A predicate register (PR) works as a mask to select active threads for operations. That means instructions use PR to determine whether the thread takes a branch or not (i.e., conditional branch), and true or false.
- Control (*Ctrl*) instructions: Are used to constitute the control flow of the given program (i.e., model), such as instruction dispatch units and block/warp schedulers of the GPU.
- Miscellaneous (*Misc*) instructions: Are used for special registers to achieve a particular task, such as data movement instruction. They hold a few special values, such as ThreadIdx.x and BlockIdx.

3.3 Instruction Vulnerability

To implement our methodology, we propose a technique, Instruction Vulnerability Factor (IVF), to quantify the impact of a fault in the instructions that are used by each model. IVF is the probability of a fault in an instruction to affect the model's computations. The average IVF value is calculated for every static instruction used in the DNN algorithm.

As shown by [28], regardless of the GPU architecture used, the error rate always depends on the executed code. Therefore, it is reasonable to investigate the error resilience of image classification (ResNet) and object detection (YOLO) models from an instruction-level perspective.

4 Results and Analysis

Following the methodology demonstrated in Section 3, we report and analyze the experimentally obtained results in this section. To present a detailed analysis of the targeted models, we separately analyze the instruction groups' vulnerability, SDC criticality, and input dependency.

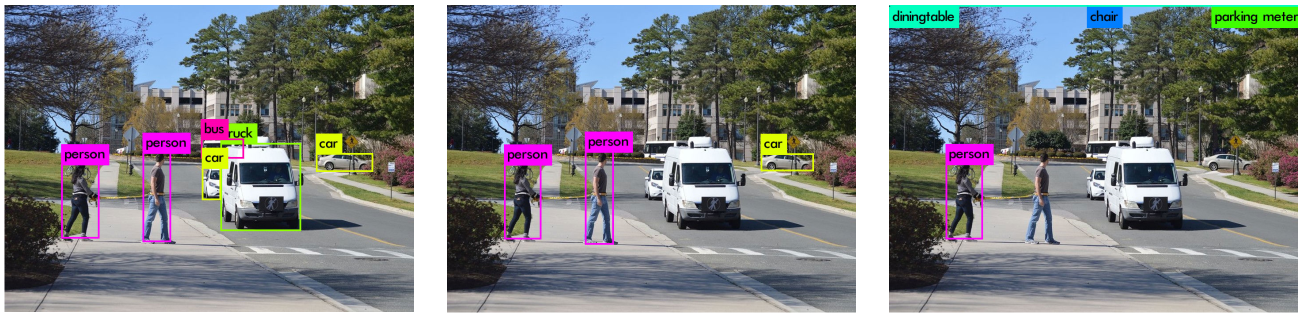
4.1 Instruction Groups Analysis

As introduced in Section 3.3, IVF can be used to calculate the overall error probability of a given model from an instruction perspective. First, we show each instruction group's occurrence rate, the percentage that the group contributes to the entire model's execution. Second, we present the error probability of each group.

For the instruction occurrence analysis, Table 2 lists the occurrence rate of all instruction groups for ResNet and YOLO models. By looking at Table 2, we observe that the two models have roughly the same distribution rates for all instruction groups. The reason is that even though the two models have different computing characteristics (classification vs. object detection), and also YOLO has more layers (107) than ResNet (69). However, there is no significant difference in their central structure. They follow the same structure, which is a series of convolution operations interleaved with Residual operations. Therefore, it is reasonable for the two models to share almost (99%) the same static instructions. Nevertheless, this does not mean injecting faults into these models would also lead to the same trends because the sensitivity of each lies behind their prediction semantics. This is explained in the next subsection.

Major observations for the occurrence rates are reported as follows:

FxP instructions contribute to the most significant percentage of the occurrence rate (at least 56%).



(a) Error-free prediction (b) Condition one (c) Condition two

Fig. 3: YOLO meeting different conditions for error criticality: (a) YOLO met no condition (predicted normally), (b) YOLO met condition one (the number of the objects unmatched), (c) YOLO met condition two (the labels of the detected objects are different).

This is due to the parallelism in the GPU architecture, which has been built around the concept of Single Instruction Multiple Thread (SIMT) fashions (see Fig. 1). Many ALUs are utilized per fetching [29]. While *FP* instructions also fall under the same parallelism rule that *FxP* follows, they show one of the lowest occurrence rates (no more than 4.52%). This ultimately depends on the program (i.e., model) being executed. ResNet and YOLO use integer operations more often than floating-point ones.

Ctrl and *LD_ST* instruction groups have the smallest percentage of occurrence, respectively, at most 3.49% and 3.72%. For *Ctrl*, by looking at Fig. 1, we find that GPU's control unit occupies only a small portion of the hardware area. Thus, GPUs inherently have a tiny number of control instructions. For the *LD_ST* group, GPUs have relatively less memory access in order to reduce memory transactions and, thus, hide memory latency [29].

PR_CC instruction group occurs more than twice of *LD_ST* and *Ctrl* groups (at least 8.85%). Each GPR instruction can be predicated if it has active threads. This means instructions use PR to determine whether a thread takes the branch or not (i.e., conditional branch), and true or false. Moreover, *PR_CC* can be an integer, floating-point, or condition code. Therefore, their rates are relatively higher than in *LD_ST* and *Ctrl*.

For the error analysis, we injected a total of 10,000 faults into each model, observing the average of 2,687 and 1,543 SDCs from ResNet and YOLO, respectively. The disruption of various low-level instructions causes these SDCs. Therefore, we use IVF (our measurement unit) to measure the error percentage of each instruction group. As shown in Fig. 2, IVF values are presented for DUE, SDC, and Masked for both models.

By looking at Fig. 2, for both models, SDC errors are more likely to occur than DUE errors. For ResNet, 26% of the total injected faults result in SDCs, while YOLO produces 15% SDCs of total injected faults. An interesting observation is that *LD_ST* and *FP* are among the lowest occurrence rates, respectively, no more than 3.72% and 4.52%. But, they show the highest rates of SDC in both models, where *LD_ST* and *FP* groups produce about 9% IVF in ResNet and 7% IVF in YOLO.

These values make these two instruction groups are the most vulnerable ones amongst all groups. For *LD_ST*, this occurs due to the working mechanism of DNNs, in which data is massively loaded from the previous layer to the current layer, and also the processed data is stored again in the memory (per layer). Considering the number of layers per model and the number of filters (32 up to 256) per layer, we note that an enormous amount of Load and Store instructions are required. For *FP*, to maintain the model's precision, data stored in the image (matrix) and filters (also matrices) are typically represented in floating-point formats. This means that many floating-point (FP) operations are required.

The most noticeable difference between the two models is that when faults are injected into *PR_CC* instructions. ResNet is likely to generate a considerable amount (0.06% IVF) of SDCs with this instruction (see Fig. 2a). YOLO, on the other hand, tends to mask the vast majority (0.17% IVF) of errors injected into this instruction group, where only significantly less (0.01% IVF) SDCs are produced (see Fig. 2b). The reason why YOLO masks errors with *PR_CC* injection is as follows: although YOLO takes the entire image as an input (the same as ResNet), it divides the image into grids. Then, image classification and localization are applied to each grid. Last, it predicts the bounding box coordinates and their corresponding class probabilities for objects (if there is any) [19]. Thus, there will be regions (i.e., grids) in the image which do not contain any objects. Therefore, faults injected in *PR_CC* are mostly masked. This is not the case with ResNet since it considers the whole image

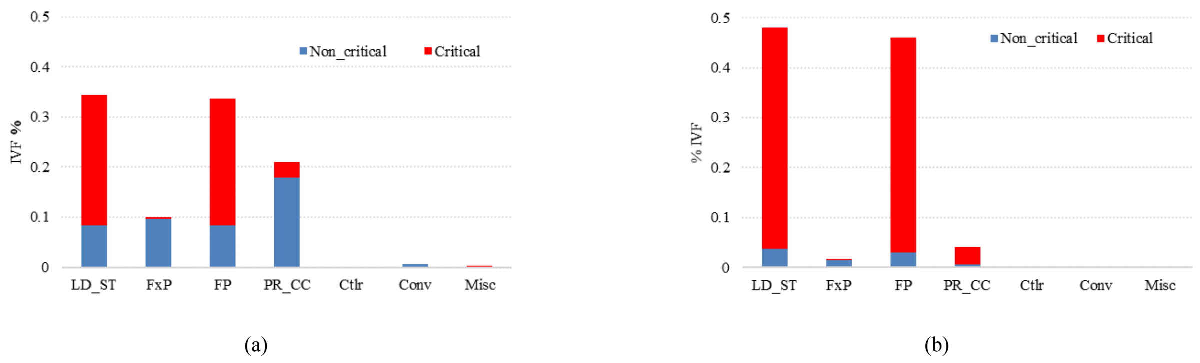


Fig. 4: SDC criticality analysis for (a) ResNet model and (b) YOLO model.

in its predictions.

The two models show the same trends in the DUE errors, where 12% of the injected faults produced DUEs and, more importantly, about 92% of the DUEs in the entire model caused by *FxP*. One instruction group responsible for 92% of the overall IVF is worth looking at. We believe that this finding can help GPU designers to consider putting a watchdog for *FxP* units to resolve DUE's problem.

In conclusion, *LD_ST* and *FP* instructions are the most vulnerable ones against SDCs. For *LD_ST*, due to the vast data transmission among layers, *FP* instructions are susceptible due to the sensitivity in the model's precision. Accordingly, only instructions with the highest IVF values are nominated for protection to avoid overheads that stem from mitigation techniques.

4.2 SDC Criticality Analysis

As demonstrated in the previous subsection, only *LD_ST* and *FP* were found to be the most vulnerable instruction groups to SDCs in both models. SDC errors can cause incorrect predictions. However, not all SDCs lead to failure in classification or object detection systems. Therefore, it is necessary to quantify whether the obtained SDC is critical (i.e., causes misclassification or misdetection). In this subsection, we analyze the criticality of SDCs and classify them into critical and non-critical SDCs.

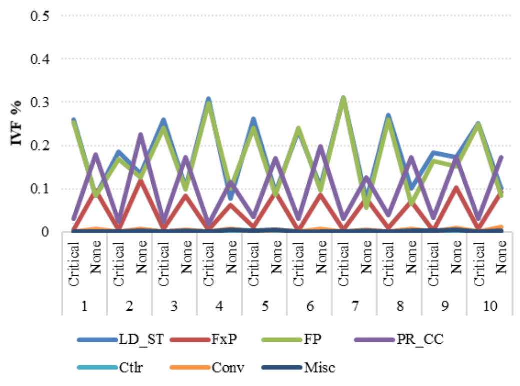
For ResNet, to differentiate between critical and non-critical SDCs, we consider only one condition. If and only if the error propagated, reached the model output, and altered the probabilities vector. Thus, it changed the class's rank. Then, it is categorized as a critical SDC. As a result, it incorrectly predicts a different object rather than the one that appears in the image (e.g., a bird instead of a truck).

For YOLO, we defined three conditions for critical SDC: (1) If the number of objects in the error-free and injected versions is different; (2) condition one is false, but the name of the objects are different; (3) conditions one and two are false, but the top-ranked object's coordinates have been altered by +/-5% compared to the error-free version. These conditions are demonstrated in Fig. 3. In the error-free prediction, YOLO detected six objects (two persons, two cars, a bus, and a truck). Thus, the model correctly predicted all objects (Fig. 3a). When it met condition one, it only detected three objects while missing three significant objects (Fig. 3b). When YOLO met condition two, five of the labels attached to the objects are different (Fig. 3c). Indeed, it "incorrectly" detected objects do not even exist in the image. We did not show condition three's image output due to the paper's space limit. This condition adjusts the location of "the colored rectangle surrounding an object," the top-ranked object (the green rectangle in this case) by +/-5%.

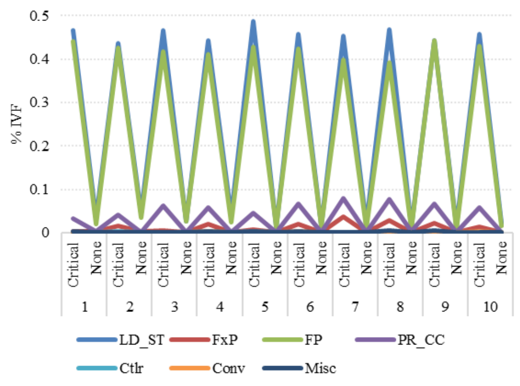
Fig. 4 shows the IVF values of the critical and non-critical SDCs for the instruction groups, both models. For ResNet (Fig. 4a), we observe that 55% of the SDCs (for the whole model) were critical, and 51% of them caused be *LD_ST* and *FP* groups. For YOLO (Fig. 4b), 91% of the SDCs (for the whole model) were Critical, and 87.3% of them caused be *LD_ST* and *FP*. We can conclude that YOLO is much more sensitive to SDC changes than in the ResNet case. The reason behind these occurrences is the number of metrics (conditions) used to identify whether the error is critical or not. For ResNet case, only one metric is required to cause misclassification, while for YOLO, three metrics are required. Thus, having three choices to decide is not the same as having only one choice.

4.3 Input-dependency Analysis

The model's input (i.e., image) can have various characteristics (i.e., single or multiple objects within the same image) and different qualities (i.e., color, brightness, focus, sharpness, and contrast). Besides, ResNet and YOLO handle images differently. Therefore, in this subsection, we investigate whether the



(a)



(b)

Fig. 5: SDC input-dependency analysis for different (ten) images to compare IVF values of the instruction group for (a) ResNet model and (b) YOLO model.

model's reliability has a dependency on the input sample. In other words, how these models react when fed with different inputs in the presence of soft errors? For this investigation, we performed fault injection for ten different images with each model. Fig. 5 depicts the input-dependency analysis for both models.

As a confirmation of the previous findings in Section 4.1 and 4.2, even with different images, *LD_ST* and *FP* instruction groups produce the highest critical SDC rate for both models. However, we note a variance regarding the input's dependency. By taking a close look at Fig. 5a, we can find that *FP* produces 1.6% IVF "critical" with image number 2, while it produces 3.11% IVF with image number 7 (doubled the amount of image number 2). Despite we only focus on critical SDCs, even the percentage of non-critical errors considerably varies. This means that ResNet depends on the input image in its reliability. On the other hand, YOLO seems very consistent in its error rate with all different inputs (see Fig. 5b). We believe that the same causes that led to critical SDCs are applied to this analysis. Therefore, even if the input is different for the YOLO case, the number of critical errors would remain significant because the probability of meeting one of the three conditions is high. We can conclude that YOLO maintains its error criticality rate with different inputs due to the detector's increased number of metrics. Whereas ResNet depends heavily on the input image since only one metric is measured.

5 Conclusion

In critical domains, the DNN model's performance cannot be compromised due to its reliability. Selective software-based mitigation techniques are commonly used to mitigate the impact of soft errors in DNNs. However, these methods still introduce undesirable overheads since the evaluation is performed at higher levels, which is more harmful if the latency is a determining factor in these systems.

In this study, we proposed a methodology to analyze the reliability of DNN algorithms run on GPUs from the lowest (instruction) level. For this methodology, we proposed a technique, Instruction Vulnerability Factor (IVF). We validated our methodology on image classification (ResNet) and object detection (YOLO) models. We have conducted an in-depth analysis of the given models to characterize their behaviors against soft errors. Our strategy demonstrates that only *LD_ST* and *FP* are the most vulnerable instructions. Further, YOLO is much more sensitive to SDC changes than in the ResNet case. Also, ResNet depends on the input image in its reliability, while YOLO is likely independent.

This methodology provides an opportunity for the designers and developers (who concern about DNN's reliability) to identify the vulnerable instructions. Consequently, the trade-off can be balanced between the use of mitigation solutions and model performance. The proposed methodology is promising to be valid for other DNNs algorithms on NVIDIA GPUs.

As future work, we plan to (1) extend our methodology to other DNN frameworks than Darknet; (2) consider other architectures than Maxwell and Pascal; and (3) discover selective hardening techniques based on our methodology.

References:

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, ImageNet classification with deep convolutional neural networks, *Communications of the ACM*, vol. 60, no. 6, 2017, pp. 84-90.

[2] Z. Zhao, P. Zheng, S. Xu, and X. Wu, Object Detection With Deep Learning: A Review, *IEEE Transactions on Neural Networks and Learning Systems*, vol. 30, no. 11, 2019, pp. 3212-3232.

[3] L. Jiao *et al.*, A Survey of Deep Learning-Based Object Detection, *IEEE Access*, vol. 7, 2019, pp. 128837-128868.

[4] L. Fridman *et al.*, MIT Advanced Vehicle Technology Study: Large-Scale Naturalistic Driving Study of Driver Behavior and Interaction With Automation, *MIT press*, vol. 7, 2019, pp. 102021-102038.

[5] T. A. Litman, Autonomous Vehicle Implementation Predictions: Implications for Transport Planning, *Engineering*, Vol.X, No.X, 2019.

[6] T. Ozturk, M. Talo, E. A. Yildirim, U. B. Baloglu, O. Yildirim, and U. Rajendra Acharya, Automated detection of COVID-19 cases using deep neural networks with X-ray images, *Computers in Biology and Medicine*, vol. 121, 2020, pp. 103792.

[7] A. Y. Hannun *et al.*, Cardiologist-level arrhythmia detection and classification in ambulatory electrocardiograms using a deep neural network, *Nature Medicine*, vol. 25, no. 1, 2019, pp. 65-69.

[8] V. Kothari, E. Liberis, and N. D. Lane, The Final Frontier: Deep Learning in Space*arXiv e-prints*, 2020, pp. arXiv:2001.10362.

[9] Y. Ibrahim, *et al.*, Soft Errors in DNN Accelerators: A Comprehensive Review, *Microelectronics Reliability*, Vol. 115, pp. 113969, 2020.

[10] Z. Chen, G. Li, K. Pattabiraman, and N. DeBardeleben, BinFI: an efficient fault injector for safety-critical machine learning systems, *International Conference for High Performance Computing, Networking, Storage and Analysis*, Denver, Colorado, 2019.

[11] K. Shuler, Taking Self-Driving Safety Standards Beyond ISO 2626*Semiengineering*, https://semiengineering.com/taking-self-driving-safety-standards-beyond-iso-26262, accessed on June 19, 2020.

[12] F. F. d. Santos and P. Rech, Analyzing the criticality of transient faults-induced SDCS on GPU applications, *ACM Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, Denver, Colorado, 2017.

[13] S. Hong, P. Frigo, Y. Kaya, C. Giuffrida, and T. Dumitraş, Terminal brain damage: exposing the graceless degradation in deep neural networks under hardware fault attacks, *Proceedings of the 28th USENIX Conference on Security Symposium*, Santa Clara, CA, USA, 2019.

[14] Y. Ibrahim, *et al.*, Soft Error Resilience of Deep Residual Networks for Object Recognition, *IEEE Access*, , vol. 8, 2020, pp. 19490-19503.

[15] 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, 2019, pp. 663-677.

[16] J. Wei *et al.*, Analyzing the impact of soft errors in VGG networks implemented on GPUs, *Microelectronics Reliability*, vol. 110, pp. 113648, 2020.

[17] A. Grzywaczewski, Training AI for Self-Driving Vehicles: the Challenge of Scale, *NVIDIA*, Oct. 2017, Available: https://devblogs.nvidia.com/training-self-driving-vehicles-challenge-scale/

[18] C. Szegedy *et al.*, Going deeper with convolutions, *IEEE Conference (CVPR)*, 2015, pp. 1-9.

[19] J. Redmon and A. Farhadi, YOLOv3: An Incremental Improvement, *arXiv e-prints*, 2018, p. arXiv:1804.02767.

[20] J. Redmon, Darknet: Open Source Neural Networks in C, *Darknet*, [online]. Available: http://pjreddie.com/darknet/, 2016.

[21] S. Cook, CUDA programming: a developer's guide to parallel computing with GPUs, *Newnes*, 2012.

[22] CUDA Toolkit Documentation, *NVIDIA*, https://docs.nvidia.com/cuda/cudacompiler-driver-nvcc/index.html. Accessed: 2020-06-14.

[23] NVIDIA Parallel Thread Execution ISA, *NVIDIA*, https://docs.nvidia.com/cuda/parallel-thread-execution/index.html. Accessed: 2020-06-21

[24] NVIDIA Corporation, CUDA_Binary_Utilities, Nov. 2019, *NVIDIA*, https://docs.nvidia.com/cuda/archive/9.1/pdf /CUDA_Binary_Utilities.pdf.

[25] J. Lew *et al.*, Analyzing Machine Learning Workloads Using a Detailed GPU Simulator, *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2019, pp. 151-152.

[26] Y. Ibrahim, H. Wang, and K. Adam, Analyzing the Reliability of Convolutional Neural Networks on GPUs: GoogLeNet as a Case Study, *2020-Intern. Conf. Comp. Info. Tech. (ICCIT-1441)*, University of Tabuk, Tabuk, KSA, Sep 2020.

[27] S. K. S. Hari, T. Tsai, M. Stephenson, S. W. Keckler, and J. Emer, Sassifi: An architecture-level fault injection tool for gpu application resilience evaluation, *Intern. Sym Performance Analysis of Sys and Software (ISPASS)*, California, USA, Oct 2017.

[28] V. Fratin, D. Oliveira, C. Lunardi, F. Santos, G. Rodrigues, and P. Rech, Code-Dependent and Architecture-Dependent Reliability Behaviors, *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, , 2018, pp. 13-26.

[29] M. Stephenson *et al.*, Flexible software profiling of GPU architectures, *ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, 2015, pp. 185-197.

Contribution of individual authors to the creation of a scientific article (ghostwriting policy)

Younis Ibrahim: Has conceived the presented idea, methodology, implementation, writing, reviewing, and editing.

Creative Commons Attribution License 4.0 (Attribution 4.0 International , CC BY 4.0)

George Smith has implemented the Algorithm 1.1 and 1.2 in C++.
Junyang Liu: Responsible for the fault-injection setup and its campaigns, also writing the original draft.
Xuanxuan Yang: Has implemented the analysis algorithm in Python.
Hongwei Sha: Has Carried out the analysis and wrote its corresponding section.
Peng Li: Has modified and added to the methodology and also revised and edited the manuscript.
Haibin Wang: Is the direct supervisor of the whole project.

Sources of funding for research presented in a scientific article or scientific article itself

This work is supported through Innovation Foundation of Radiation Application, China Institute of Atomic Energy (KFZC2018040205). It is also supported by Changzhou Sci & Tech Program (Grant no. CZ20180003).