Check for updates

IET Computers & Digital Techniques

Special Issue: Defect and Fault Tolerance in VLSI and Nanotechnology Systems

IET Journals
The Institution of Engineering and Technology

Kernel and layer vulnerability factor to evaluate object detection reliability in GPUs

ISSN 1751-8601
Received on 5th April 2018
Revised 7th July 2018
Accepted on 28th August 2018
E-First on 4th December 2018
doi: 10.1049/iet-cdt.2018.5026
www.ietdl.org

Fernando Fernandes dos Santos[1] ✉, Luigi Carro[1], Paolo Rech[1]
[1]*Instituto de Informática, Universidade Federal do Rio Grande do Sul, Bento Gonçalves Avenue, Porto Algre, Brazil*
✉ *E-mail: ffsantos@inf.ufrgs.br*

Abstract: Video recognition applications running on Graphics Processing Unit are composed of heterogeneous software portions, such as kernels or layers for neural networks. The authors propose the concepts of kernel vulnerability factor (KVF) and layer vulnerability factor (LVF), which indicate the probability of faults in a kernel or layer to affect the computation. KVF and LVF indicate the high-level portions of code that are more likely, if corrupted, to impact the application's output. KVF and LVF restrict the architecture/program vulnerability factor analysis to specific portions of the algorithm, easing the criticality analysis and the implementation of selective hardening. We apply the proposed metrics to two Histogram of Oriented Gradients (HOG), and You Only Look Once (YOLO) benchmarks. We measure the KVF for HOG by using fault-injection at both the architectural level and high level. We propose for HOG an efficient selective hardening technique able to detect 85% of critical errors with an overhead in performance as low as 11.8%. For YOLO, we study the LVF with architectural-level fault-injection. We qualify the observed corrupted outputs, distinguishing between tolerable and critical errors. Then, we proposed a smart layer duplication that detects more than 90% of errors, with an overhead lower than 60%.

1 Introduction

Today image processing algorithms are fundamental for applications that go beyond the multimedia or gaming ones. In fact, modern and next generation driverless cars rely on image processing to detect objects, obstacles, and pedestrian on frames received from cameras or radars. While the performances of image processing algorithms are a key feature, their reliability needs to be paramount when applied to self-driving cars.

Object detection is a resource hungry application. Nowadays, graphics processing units (GPUs) are the most adapt devices to execute object detection applications as they have a massive amount of resources and are extraordinarily efficient to parallelise execution. GPUs are adopted to accelerate object detection applications, disclosing the use of embedded GPUs in self-driven cars and robotics applications [1, 2]. GPUs are also part of projects implementing the advanced driver assistance systems, which rely on camera feeds and radar signals to detect obstacles, such as pedestrians, and activate the car brakes to prevent collisions [3].

Assisted and especially autonomous driving systems, which are the new trends in the automotive market, rely on computer-aided detection to function correctly and safely. Unfortunately, while GPUs are extremely efficient for parallel processing in terms of both floating-point operations per seconds (FLOPs) and FLOPs per watt, their architecture has been proved to have some intrinsic weaknesses [4–7]. The corruption of shared resources (such as the caches) or critical resources (such as the hardware scheduler), in fact, is likely to affect the correctness of several threads, leading to multiple corrupted elements in the output. Additionally, in parallel applications, a single thread data coming from a single thread can be used to feed several downstream threads. A single corruption on the upstream thread is then likely to affect several other threads.

GPU integrated into a vehicle guidance system must be compliant with the strict ISO26262 constraints [8]. Object detection is categorised as a safety-critical feature, and so it must respect the automotive safety integrity level D (ASIL-D). ASIL-D requires any component in the system to be able to detect 99% of the permanent and transient faults that might occur. Failure tolerance of this system is limited to ten failure-in-time (FIT) (errors in 10^9 hours of operation). Defective behaviours could be the outcome of faults from various sources that undermine the system reliability. Those sources include software errors, environmental perturbation, and process, temperature and voltage variations [9–11]. The generated error may corrupt logic operations or data values, leading to a single data corruption (SDC), cause the system to crash or hang or be masked and cause no noticeable error. Radiation-induced soft errors are a significant concern in modern computing devices because, if uncorrected, produce a failure rate that is higher than all the other sources combined [11]. The reliability characterisation of algorithms implementing object detection executed on GPUs is then mandatory to ensure human life and vehicle safety.

In this study, we evaluate the reliability of object detection algorithms as executed on modern GPUs. We especially consider two algorithms, histogram of oriented gradients (HOG) [12] and you only look once (YOLO) [13], which are the core of several object-detection applications [14–16]. Currently, the most accurate object detection methods are based on feature extraction such as HOG and YOLO [12, 13, 15]. HOG and YOLO, then, share the same computational characteristics with most of the modern objects detection frameworks based on feature extraction algorithms. The analysis we present and the hardening efficiency we achieve is then likely to be valid also for other object detection methods.

Then, based on the proposed reliability evaluation, we design and validate efficient solutions to improve the reliability of HOG and YOLO. Nowadays, the most common approach to improve the reliability of applications is through software or hardware redundancy [17, 18]. Double modular redundancy (DMR) and triple modular redundancy (TMR) consist of the duplication or triplication of a system module. DMR or TMR can be done on hardware or software level. Full DMR and TMR are not ideal for image processing applications, as some portions of the code could be intrinsically fault-tolerant and their duplication is likely to introduce unnecessary overhead. In this study, we show that not all application modules (i.e. kernels or layers) need to be duplicated to achieve high reliability since they do not share the same criticality. Then, evaluating application-specific portions, we are able to find a trade-off between performance and hardening efficacy, without adding unnecessary overhead.

Our idea arises from the observation that to efficiently extract different features from a frame and perform the correct detection,

© The Institution of Engineering and Technology 2018

modern algorithms are divided into several kernels (HOG) or layers (YOLO). Kernels or layers are typically heterogeneous in terms of operations performed, complexity, memory requirement, and/or execution time. Additionally, each kernel or layer is likely to have a proper transient error sensitivity and the corruption of a kernel or layer may or may not be critical for the final object detection feature. As a result, the protection of a kernel or layer may or may not significantly improve the object detection framework reliability. To efficiently improve the reliability of modern image processing algorithms we should, first, identify the kernels or layers whose corruption is more likely to impact the object detection correctness. We define the kernel vulnerability factor (KVF) as the probability of a fault in a kernel to affect the application output and layer vulnerability factor (LVF) as the probability for a fault in a layer to affect the application output. Additionally, we measure both the KVF and LVF for major and minor impact output errors to identify those portions of the code that are likely to generate errors that significantly affected the performed object detection. KVF and LVF are powerful tools to distinguish kernels and layers that should be hardened from kernels whose corruption is unlikely to affect the application.

Thanks to the fault-injection campaigns we identify the KVFs of HOG and LVF for YOLO. All collected errors are available in a public repository, allowing third-party analysis, and enabling reproductivity of our results [19]. We show that not all the kernels (layers for YOLO) are critical, and hardening techniques should only focus on kernels with a high KVF/LVF. With CUDA-GDB injections, we also correlate the KVF with high-level code portions, showing which variables are more likely to be responsible for the observed errors. CUDA-GDB [20] combined with an NVIDIA SASSIFI [21] fault injector provides complete information about how faults on each kernel propagate until the final output and help in designing efficient hardening solutions. As shown in the study, by duplicating only the kernel or layer with higher KVFs or LVF, we can detect more than 80% of errors with a performance overhead of only 11.8%.

The main contributions of our work are: (i) the KVF and LVF formalisation, and including realistic application case-study; (ii) an extensive analysis of HOG and YOLO behaviour under fault-injection; (iii) efficient and robust hardening techniques for HOG and convolutional neural network (CNN) running on GPUs.

The remainder of the paper is organised as follows: Section 2 gives insights on GPU reliability, and explain how HOG and YOLO perform object detection task. Section 5 presents the methodology for this study and presents the obtained results. In Section 7, we conclude the paper.

2 Related work and background

This section serves as a background on both reliability and object detection algorithms. We first summarise previous findings on GPUs reliability and then give background and present a reliability viewpoint of HOG and CNN, which are the case-study applications to demonstrate how the KVF and LVF can be successfully applied to design efficient hardening for GPU.

2.1 GPU reliability

A transistor's state could be perturbed by a terrestrial neutron strike, generating bit-flips in memory or spikes in logic circuits that, if latched, lead to an error. Transient errors induced by radiation leads to three possible effects. (i) The output is the same as expected (i.e. the fault is masked or the corrupted data is not used). (ii) An SDC (i.e. an incorrect program output). (iii) A program crash or device reboot.

Some GPUs' architectures have their main storage structures with single error correction double error detection (SECDED) error correction code (ECC). ECC has been shown to reduce the error rate by one order of magnitude for high-performance computing applications running on a GPU but increases the number of crashes [7].

Several ways to mitigate SDCs in software, such as code replications, are available [22]. However, their overhead may be unacceptable for a real-time safety-critical system. Previous works have been focusing on selective hardening for central processing units (CPUs), seeking the best point on performance and reliability trade-off with interesting and promising results [23–25]. In this study, we propose an efficient software hardening strategy for HOG and YOLO. Our idea is to ease the implementation of the hardening solution by identifying the most critical high-level portions of the code and to prove that even a high-level selective-hardening could be extremely efficient, specifically for feature-extraction algorithms. In Section 6.3, we suggest a smart DMR for object detection based on our experimental data.

2.2 Histogram of oriented gradients

HOG is one of the most common feature detectors for a pattern or object detection [26], particularly in automotive applications [14]. While HOG can be combined with a variety of classifiers to detect pedestrians, in this work, we choose to apply a support vector machine (SVM) as the classifier. The reliability discussions and fault-injection results presented in this work are directly extendable to other classifiers. Detection in each frame is performed by creating a set of *bounding boxes* (*BBs*), which are pedestrian candidates. Within these BBs, the features are extracted and classified using the SVM, yielding a set of validated BBs. HOG consists of many kernels with different computing characteristics executed in the GPU. Our goal is to measure the KVF of HOG, which is the probability of corruptions in each kernel to impact the final object detection.

HOG's first kernel is *resize*, which is a preprocessing on the image for colour correction and resizing, enhancing detection capabilities. Then, *compute gradients* is executed, in which a simple derivative mask filter is applied to the image to detect gradients (i.e. edges). Then, HOG performs *compute histograms*, dividing the image into 8×8 pixel regions called *cells*. Within each cell, a histogram of the pixels' gradient orientations is computed. *Normalise histograms* is the next phase, grouping adjacent cells as spatial regions, called *blocks*, based on their gradient orientation. Each block is represented by a block descriptor, which is a vector that considers the contributions of all the normalised cells in the block. Finally, the *classifier* (i.e. the SVM that performs classification) is fed with the block descriptor.

The probability of a fault to affect the kernels and the probability for a kernel corruption to affect the HOG output depend on the kernel itself are evaluated in the following sections. Being an image processing algorithm acting as a filter, HOG is intrinsically resilient to SDCs. Nevertheless, we have observed some very critical errors that corrupt the final detection entirely. Those errors are very troublesome if we consider the reliability characterisation of HOG applied to fields such as pedestrian detection, for which reliability is mandatory. For an in-depth discussion and analysis of the observed results, please refer to [27].

2.3 Convolutional neural networks

Deep neural networks (DNNs) are one of the most efficient ways to perform image classification [28], segmentation [29], and object detection [13, 30]. Prior work has tried to adopt DNNs for real-time object detection, showing promising results [31, 32]. One of the critical steps when using DNNs for object detection is convolution. A kernel filter is convolved with a matrix to extract specific features of the image. The kernel filter slides over the input matrix, multiplying and accumulating products at every position of the input with every position of the kernel. This process can be mapped to a matrix multiplication operation. Each block is reorganised as a row of matrix $\boldsymbol{A}$, and the filter kernel is replicated as columns of a matrix $\boldsymbol{B}$. Convolution is then computed as $\boldsymbol{A} \times \boldsymbol{B}$.

Artificial neural networks (ANNs) are composed of a pipeline of *layers*. On a traditional ANN, a layer is a group of nodes, called *neurons*, with a specific activation function or operation to perform. Layers are executed one after the other in a sequential way. The first layer will receive the input and the last layer will produce the classification results. A CNN is a deep network, with several layers that perform *convolution* on the raw image or a feature map (i.e. the output of an upstream convolutional layer). These convolution layers extract the features from the image,

© The Institution of Engineering and Technology 2018

1751861x, 2019, 3, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/iet-cdt.2018.5026 by Cochraneltalia, Wiley Online Library on [01/03/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Table 1 HOG and YOLO separated by kernels/layers

Kernel/layer	Main characteristics	Execution time percentage
HOG		
resize	it resizes the input image and applies colour correction	5.56%
compute gradients	a derivative mask filter is applied to the image, to detect gradients	9.78%
compute histograms	the image is divided into a group of cells, then a histogram of gradient orientation is computed for each cell	47.81%
Normalise histograms	it groups the adjacent cells into blocks, then each block is represented by a block descriptor that will be used by the classification	14.43%
classify	in this implementation of HOG, an SVM is fed with the block descriptors to make the object classification	22.42%
YOLO		
convolution layer	it makes the feature extraction, each layer will extract different features from the input frame	91%
maxpooling	it does the down sampling on the convolution layer output, removing the network unnecessary information	7%
fully connected	it is in charge of making the network final classification based on the features extracted by the layers before	2%

which are used to detect objects. Additionally, the CNN includes also rectified linear units (ReLU), Maxpool, and fully connected layers, which are the primary structures present in most modern CNNs. ReLU layers apply an activation function (removing all negative values) on the feature map to eliminate the linearity of the input. Maxpool splits the matrix into n block-based regions and propagates only the largest value in each block, reducing the dimension of the processed data by n times. Maxpool layers mask a significant portion of SDCs, while fully-connected layers tend to spread them. The fully connected layers are conventional ANNs in charge of classifying objects based on the extracted features.

In this study, we consider the YOLO [13] framework. *YOLO* is based on *Darknet*, which is an open source CNN written in C and CUDA [13]. YOLO performs detection by dividing the input image into $S \times S$ grids. For each grid, YOLO calculates the class probability, the BBs (i.e. potential objects), coordinates and confidence. Performing processing of each frame ``only once' delivers detection and classification in real time. YOLO is composed of 24 convolutional layers interleaved with maxpooling layers to reduce the dimension of the processed data.

3 Fault-injection methodology

In this work, we aim to evaluate the reliability of HOG and YOLO by performing extensive fault-injection campaigns. Our goal is to identify critical portions (i.e. kernels or layers) of the algorithm. We perform fault-injection with three different levels of abstraction: (i) register file (RF) injections, (ii) instruction output (INST) injections, and (iii) high-level fault-injection. (i) and (ii) are performed using SASSIFI [21], while for (iii) we have used CUDA-GDB [20] for the HOG algorithm.

By correlating the injected fault location with the executed kernel or layer, we can evaluate the KVF or LVF, which is an indication of the most critical parts of the algorithm. Then, with high-level fault-injection, we can further understand the vulnerability of the code, correlating the high-level code portion with their corruption probability to propagate to the output. This complete analysis allows us to confidently detect which parts of the algorithm tend to produce output errors, and propose dedicated mitigation solutions.

We perform all fault-injection on HOG and YOLO running on NVIDIA K20 on selected complex frames from the UrbanStreet dataset [33]. Such a decision is based on the desire to analyse object detection performing on complex frames with multiple pedestrians in different positions and situations, while also using inputs from a universally recognised dataset. The complexity of the chosen frames allows us to evaluate how transient errors impact the application performing detection on a wide set of situations, such as clear pedestrians, clusters of pedestrians, and pedestrians partially covered by other objects of the scene. It is worth noting that our objective is not to assess the quality of the performed detection, but rather to understand how HOG and YOLO react to faults (Table 1).

It is also important to note that while the primary purpose of our fault-injection procedures is to simulate and better understand radiation-induced failures, the provided insights are directly extendible to other sources of transient errors, environmental perturbation, and process, temperature and voltage variations.

3.1 SASSIFI fault-injection

SASSIFI injects transient errors in instruction set architecture (ISA) visible states, such as general purpose registers, stored values, predicate registers, and condition registers [21]. In this work, SASSIFI is used to inject faults into two injection sites: the INST and in the RF. To restrict the RF architecture vulnerability factor (AVF) evaluation to a specific code portion, we did not inject faults in the whole RF but only on those registers that are being used in the specific code portions, we are analysing. As such, our result for RF is to be intended as the probability for a fault in a RF used in a specific layer or kernel to impact the overall application output.

RF injections are to be considered the lowest level of injection, while INST injections are somehow performed at a higher level. In fact, with INST we simulate all the faults occurring during the execution of an instruction, which appears at the INST. RF injections, on the contrary, are bit-flips in low-level resources that, once digested by instructions, modify the output. Using SASSIFI, we injected at least 10,000 faults per benchmark (i.e. 5000 for each error site). At 5000 faults the statistical significance of the calculated AVF and program vulnerability factor (PVF) gets saturated, being sufficient to guarantee the worst case statistical error bars at 95% confidence level to be at most 1.96% [21, 34].

All fault injections were performed on a Tesla K20. The K20 supports the *Kepler* ISA and is fabricated in 28 nm standard complementary metal oxide semiconductor technology. This model has 2688 CUDA cores divided across 14 streaming multiprocessors.

3.2 CUDA-GDB fault-injection

With CUDA-GDB, we perform a fault-injection routine on every kernel executed on the GPU, using an approach similar to the one discussed in [35]. Using a python script and GDB, we can freeze HOG at runtime and change local variable values on any of the HOG kernels. To do this, before fault-injection starts we perform a profiling routine, which maps all accessible local kernel variables into a list. We then randomly select a kernel to place a breakpoint. When such a breakpoint is reached at runtime, the execution is frozen, and a random local variable from the kernel is selected from the list. The context is then switched to the host, which performs fault-injection by assigning a random value to the selected variable. Execution is then resumed.

We choose to inject random values (i.e. load the correct value and then generate a random value based on the original value.) and no single bit-flips because, from a high-level view, the random value assigned simulates all kinds of errors, such as single and multiple bit-flips. When radiation or other sources of SDCs generate bit-flips on low-level resources, the error may propagate, resulting in an unpredictable and wrong value written to memory.

From a high-level view, the wrong variable value is not necessarily limited to a single bit of difference from the correct

© The Institution of Engineering and Technology 2018

1751861x, 2019, 3, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/iet-cdt.2018.5026 by CochraneItalia, Wiley Online Library on [01/03/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

value. As we inject faults in high-level variables, injecting random values (which includes but is not limited to single bit flips) is the fault model that better fits our purposes. Thus, we have preferred to use a random value generator and inject random values rather than inject only single bit flips.

4 Proposed reliability evaluation metrics

In this section, we formalise the KVF and LVF metrics and propose a strategy to distinguish between critical and non-critical errors for object detection.

4.1 KVF and LVF

AVF and PVF are the standard metrics to evaluate fault injection results at the hardware level or at the software level, respectively [36, 37]. AVF is the probability for a fault in an architectural state to propagate to the executed program output. PVF is the probability for an error in a program variable or instruction to affect the code output. AVF and PVF are compelling tools to evaluate the overall program or architecture vulnerabilities. However, specific applications such as image processing frameworks are composed of a set of completely different high-level portions. We intend to measure the vulnerability and the criticality of each high-level portion to identify those portions to be hardened.

For this purpose, we specify AVF/PVF metrics to cover modular vulnerability factors of HOG and YOLO. KVF is used to evaluate HOG kernels vulnerability and LVF is used to evaluate CNN layer vulnerability.

KVF is the probability of a fault injected in a GPU kernel affecting the application (HOG in our case study) output; it is obtained by dividing the number of observed SDC by the total amount of injected faults.

LVF is the probability of a fault injected at CNN layer affecting the CNN output; it is obtained by dividing the number of SDCs by the total number of injected faults per layer.

It is worth noting that KVF and LVF gain importance as CNN layers and HOG kernels are highly heterogeneous and will have different LVF and KVF, respectively. LVF and KVF could be less interesting for more homogeneous or well-structured algorithms such as arithmetic intense applications or physical simulations.

Each kernel or layer reliability could be further analysed to identify those portions of the kernel or layer that are more vulnerable. However, the protection of kernel or layer is an incredibly trivial task as algorithms are modular. The duplication of a layer inside a CNN, for instance, can be done with little effort. On the contrary, the duplication of variables or instructions inside the layer requires internal synchronisation and the re-allocation of processor resources. Moreover, the duplication of a single instruction could result in a doubled execution time if parallel resources are saturated. As we show in Section 6, by duplication kernels or layers, we can achieve excellent reliability with little overhead.

4.2 Error criticality evaluation

Some errors affecting the output of an object detection framework could still be acceptable. In fact, output errors could be tolerated as long as the predicted object's position is sufficiently close to the expected position. To distinguish between critical and tolerable errors we use *precision* and *recall*, which are metrics introduced to access the quality of a given classifier.

Precision is given by

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}, \quad (1)$$

where TP is the number of true positives (pedestrians that were correctly detected) and FP is the number of false positives (objects incorrectly detected as pedestrians). Precision measures the fraction of the detections produced by the classifier that relate to a pedestrian so that a value of 100% means that all detections produced by the classifier are actual pedestrians.

On the other hand, *recall* is given by

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}, \quad (2)$$

where FN is the number of false negatives (a pedestrian that was not detected). Recall indicates the fraction of existing pedestrians that were detected by the classifier, even in the event of an error. Hence, a value of 100% means that all pedestrians were successfully detected.

To obtain the TP, FP and FN values, we analyse the relationship between the BB on the erroneous output and the golden one. Individually, we consider a BB n in the corrupted output a TP if, for any BB m of the golden output, their Jaccard similarity is ≥ 0.5 (i.e. their areas overlap by at least 50%). Otherwise, we mark n as a FP. Similarly, if for a given BB m of the golden output there is no BB n on the corrupted output which satisfies this condition, a FN is detected. The chosen threshold of 0.5 was identified as a good trade-off to evaluate detection quality [38].

By using Precision and Recall, we can then go a step beyond the traditional SDC detection by considering error criticality, making it possible to identify errors that could prevent detection or erroneously lead the vehicle to a sudden stop.

Based on the Precision and Recall values of the corrupted output, we divide errors into three categories:

- *No impact:* The output is affected by a silent data corruption that still maintained both values of Precision and Recall at 100%. The output error does not impact object detection.
- *Minor impact:* The produced output had a Recall value of 100% (i.e. all pedestrians are correctly detected), but had a Precision value between 90 and 100% (some objects are incorrectly identified as pedestrians).
- *Major impact:* The produced output either had a Recall value <100% (missed a pedestrian) or had a Precision value <90% (a considerable amount of objects are incorrectly identified as pedestrians).

We choose to be considerably strict on Precision, but extremely strict on Recall. This is based on the fact that incorrectly detecting an object as a pedestrian may lead to an unnecessary stop, causing inconveniences more often than accidents. Missing actual pedestrians, on the contrary, may easily lead to accidents, injuries and even deaths, as the vehicle does not stop when it should. We will compare the KVF of HOG considering the three error criticality categories.

5 Fault-injection results

5.1 HOG KVF evaluation

Following the methodology discussed in Section 3, we use SASSIFI to inject low-level faults and CUDA-GDB to inject high-level faults on every portion of the algorithm executed on the GPU. Fig. 1 shows the obtained KVF for each error criticality category defined in Section 4.2.

We measure the KVF with CUDA-GDB fault-injection and with SASSIFI fault-injection using both injection sites. The KVF value is obtained by dividing the number of observed SDCs by the total amount of injected faults and represents the probability of an injected fault in the kernel cause an error at the HOG output. The KVF is, as to be expected, significantly dependent on the injection framework. As shown in Fig. 1, the KVF increases as injections are made at a higher level. This is not surprising, as the high-level faults affect resources which are much more likely to be used and, thus, to impact the final output. Low-level faults injected with SASSIFI on RFs are far less likely to generate SDCs than faults injected on the INST. This is because, while instruction results are bound to be used by the following instructions, the corrupted RF data could be obsolete or unused. In other words, INST injections are meant to measure the PVF, while RF injections provide the AVF.

Fig. 1*c* shows that 20% of high-level injected faults on *resize* produced an SDC, all of which majorly impacted the final detection. However, Figs. 1*a* and *b* show that no produced SDCs

© The Institution of Engineering and Technology 2018

1751861x, 2019, 3, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/iet-cdt.2018.5026 by CochraneItalia, Wiley Online Library on [01/03/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

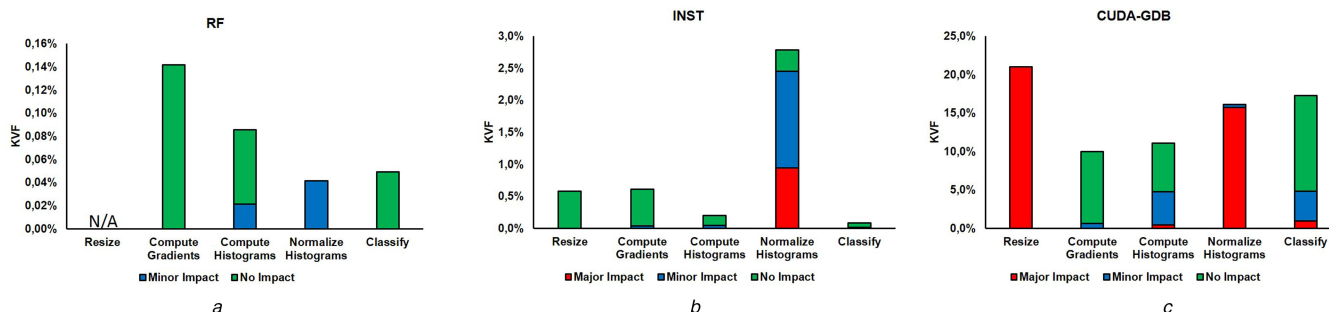

Fig. 1 *KVF for each HOG kernel obtained using*
(a) SASSIFI fault-injection on the RF injection site, ***(b)*** SASSIFI fault-injection on the INST injection site, ***(c)*** CUDA-GDB fault-injection. N/A indicates that no errors were observed

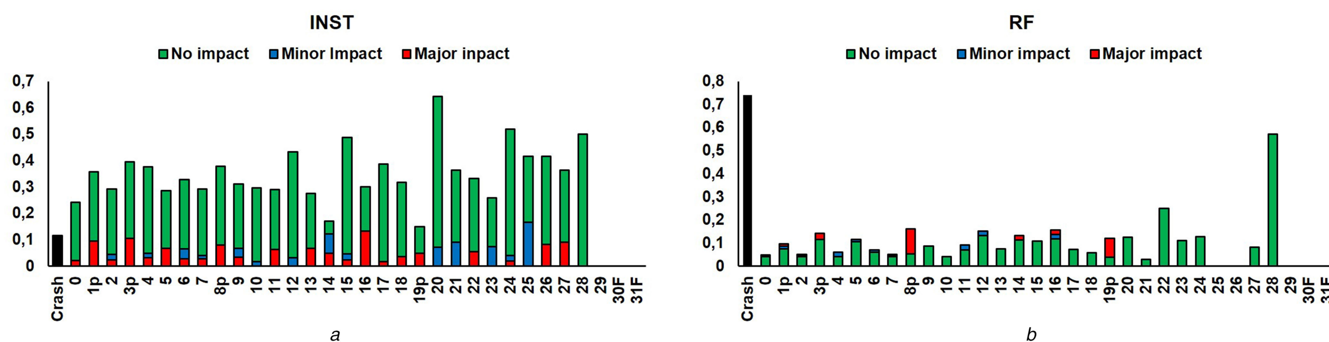

Fig. 2 *LVF for each YOLO layer obtained using*
(a) SASSIFI fault-injection on the INST injection site, ***(b)*** SASSIFI fault-injection on the RF injection site

from our SASSIFI fault-injection campaign impacted detection quality. While being pretty robust against low-level faults, resize is very susceptible to high-level corruption. This kernel is responsible for resizing the input image, followed by a texture lookup. This lookup is responsible for colour correction and colour bias elimination, which improves detection capabilities. We observed that all SDCs on resize were caused by corruption on a red, green and blue value passed to the function responsible for this texture lookup. Since CUDA-GDB directly writes a random value to a variable during fault-injection, it is reasonable that high-level injections on this specific value significantly impacted the final detection, as the texture lookup is performed on a wrong input. Low-level faults, on the other hand, do not directly alter this variable, but only the results of intermediate computations, which could be filtered and do not impact this variable as significantly, or might not even relate to it.

Normalise histograms also seems to be a critical kernel. 15.6% of high-level and 0.94% of faults injected with SASSIFI on the INST produced a major impact SDC. Also, despite rare, all faults on the RF injection site had an impact on the final output. Normalise histograms needs to be hardened, as it is vulnerable to both high- and low-level faults.

Compute gradients, on the other hand, seems to be a robust kernel. Only 0.64% of faults injected by GDB and 0.03% of faults injected with SASSIFI on the INST impacted detection, and only in minor ways.

Compute histograms seems slightly vulnerable, as 0.39% of high-level faults had a major impact on the produced detection. Also, 4.36% of high-level faults, 0.04% of faults injected by SASSIFI on the INST and 0.021% of faults injected by SASSIFI on the RFs had a minor impact.

Finally, 0.96% of faults injected by CUDA-GDB on *classify* produced critical SDCs, and 6.79% had a minor impact. However, no impactful SDCs from our low-level fault-injection campaign were observed. This indicates that similarly to resize, classify is very robust against low-level faults, but somewhat critical when high-level faults are concerned. Being the classifier, this kernel is responsible for the final output of the algorithm (i.e. BBs which indicate the detected pedestrians). As discussed in Section 4.2, we allow for some imperfections when measuring the error criticality. Precisely, to measure Recall, in order to consider a pedestrian as correctly detected, we require the BB which represents this pedestrian on the golden output to overlap at least 50% with any BB on the corrupted output. Similarly, to measure Precision, we consider a BB on the produced output to represent an actual pedestrian if its area overlaps at least 50% with a BB from the golden output. As such, errors originated from low-level faults should only impact the produced BB in minor ways, which, despite slightly dislocated, are still considered to be detecting the pedestrian. High-level faults, on the other hand, could significantly alter a BB's position such that it is not considered to be correctly detecting a pedestrian, hindering detection quality.

5.2 YOLO LVF evaluation

Figs. 2*a* and *b* show the LVF values obtained injecting faults with SASSIFI in the RF and INST sites, respectively. The LVF is shown for all the 32 layers of *Darknet*, which is the CNN on which YOLO is based. The CNN is composed of 24 convolutional layers interleaved with four maxpooling layers (1p, 3p, 8p, and 19 in Figs. 2*a* and *b*), one local layer, one dropout layer, and two fully-connected layers (layers 29 and 30).

For each of the 32 layers, we measure the probability for an injection to (i) crashes the application, (ii) propagates to an output (SDC), and (iii) be masked by the application. When a fault propagates to the output and generates an SDC we also classify the criticality using the same approach used for HOG kernels and described in Section 4.2, i.e. SDCs are classified into no impact, minor impact and major impact.

In Figs. 2*a* and *b*, we do not distinguish the LVF for crashes as it does not differ from layer to layer. As shown in Figs. 2*a* and *b*, RF injections are much more likely to result in crashes than INST injections. The crash LVF for RF injection is about 0.74 per layer and for INST is about 0.12 per layer. This difference is justified considering that most crashes occurred when RF injections affect index registers and addresses for memory stores. It is worth noting that SECDED ECC in caches and registers is likely to mask most of those crashes. On the contrary, ECC will not guarantee crash protection for INST faults, as those faults affect unprotected structures (control logic, GPU–CPU synchronisation etc.).

INST injections are much more likely to generate SDCs (either of minor or major impact) than RF injections. INST injections

© The Institution of Engineering and Technology 2018

1751861x, 2019, 3, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/iet-cdt.2018.5026 by CochraneItalia, Wiley Online Library on [01/03/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

affect GPU ISA instruction set (e.g. arithmetic operations, branch instructions etc.). As INST generally will be used by subsequent operations, injections culminate mostly in SDCs.

As already discussed in Section 2.3 YOLO's convolutional layers are based on matrix multiplication algorithm. Nevertheless, each one of 24 convolutional layers extracts specific features from the input frame. So, each layer has different sizes and performs convolutions using different kernels. Therefore, the SDC LVF varies between the layers, i.e. their impact on the final algorithm vulnerability is different. Even if the convolutional process is based on the same operation (matrix multiplication), each LVF will be different since the layers have distinct purposes.

As shown in Figs. 2*a* and *b*, the probability of an injection to affect the network output depends on the position of the layer. LVF decreases before the maxpooling layers (1P, 3P, 8P, 19P in the figure) and then increase in downstream layers. Maxpooling discards three-fourth of the data from the previous layer, eventually masking SDCs.

Injections at the input of the two fully-connected layers (30F and 31F) do not produce SDCs, confirming the intrinsic fault tolerance of the ANN [39–42]. Most of the SDCs at the CNN output come from faults impacting a convolution layer. As YOLO performs 3D convolutions on 3D matrices on layers 0 to 29, any corruption, while propagating, potentially affects a 3D area. Recall that Figs. 2*a* and *b* showed how vulnerable are the layers. Most of those errors generated in a previous layer once digested by the downstream layers will affect the object detection output. Obviously, the higher the number of corrupted elements generated by the fault, the higher the number of items affected by propagation. Then, the higher the percentage of corrupted elements, the lower the probability for maxpooling to mask errors. Thus, more efficient hardening methodology is necessary, not allowing the errors that appear in a previous layer to be propagated by subsequent layers. In the following section, we discuss how to better explore the GPUs' architecture to make efficient DMR for CNNs.

6 Selective hardening

In this section, we propose and validate efficient and smart hardening techniques designed for HOG, based on the analysis presented in the previous sections. Moreover, we designed a smart layer DMR based on a metric which evaluates the best trade-off between vulnerability and performance.

6.1 HOG selective hardening implementation

Since we explicitly consider HOG being used for pedestrian detection, our goal is to present a selective hardening methodology that makes the algorithm robust against SDCs, especially those which tend to affect detection. As for a real-time embedded system, both execution time and power consumption need to be optimised. Our goal is to improve reliability without unnecessary overhead. We select duplication with the comparison as a baseline hardening solution as it has been demonstrated to detect more than 90% of SDCs in GPUs [43]. Unfortunately, a full duplication of HOG will introduce a 2.5× execution time overhead, which is unacceptable for a real-time system. Thanks to our KVF analysis we can select which kernels should be duplicated or protected with other strategies.

The available embedded GPUs do not have ECC implemented. The first hardening solution we propose, then, duplicates and checks not just operations and computations but also memory values.

Duplicating memory operations and memory addresses will increase the execution time of the proposed techniques. The resulting overhead is then expected to be high. If we assume that the memories are protected by ECC, i.e. no memory check is done at any level (registers, addresses or caches) we can duplicate only logic and math operations, reducing significantly the overhead.

We observed *resize* to be a critical kernel when high-level faults are considered. However, all errors were caused by injections on the colour parameters. We can then duplicate only these variables introducing a negligible overhead and significantly improving resize's reliability.

Compute gradients has moderate KVF values for both fault-injection campaigns. Compute gradients takes about 10% of HOG's total execution time. We believe compute gradients not to be worth the duplication overhead as it is responsible for a significant portion of HOG execution time while being very rarely responsible for an erroneous detection.

Compute histograms had a low major impact and minor impact KVF in both fault-injection campaigns. Only a few injections with CUDA-GDB cause a major impact. Duplicating compute histograms will only slightly increase HOG reliability and cause a significant overhead, as this kernel takes about 50% of the algorithm's total execution time. To preserve the necessary performances, we believe that compute histograms should not be hardened.

Normalise histograms proved to be an extremely critical kernel on both fault-injection procedures and needs to be thoroughly hardened. As normalise histograms takes only about 15% of HOG's total execution time, duplication on this kernel significantly increases HOG's reliability while only slightly impacting its performance.

Lastly, *classify* was shown to be robust against low-level faults but presented some critical SDCs in our high-level analysis. About one-third of the produced SDCs during our CUDA-GDB campaign impacted the detection, some in major ways. This indicates that classify should be hardened, despite taking about 22% of HOG's total execution time.

Based on this analysis, we have developed two hardened versions of HOG. If no ECC is present in the main memory, we need to duplicate the whole critical kernels, including memory spaces and memory operations. The introduced overhead is 84.8%, most of this overhead comes from duplications and checks of memory values, which are very time-consuming. If ECC is present, there is no need to duplicate and check memory values, since we are assuming that ECC would protect the registers, caches and the main memory. We then only need to duplicate and check operations and computations, and the introduced overhead is about 11.8%. It is worth noting that both versions produce a much smaller overhead than the average overhead imposed by a full algorithm duplication, which is about 150% [7].

6.2 HOG selective hardening validation

In this section, we validate the proposed hardening technique through an extensive fault-injection campaign using NVIDIA's SASSIFI. Fig. 3 shows the percentage of detected SDCs of every criticality category on each hardened HOG kernel. This figure only shows results from the hardening version which assumes that ECC is disabled, allowing us to analyse the effectiveness of both our operations and memory values duplication.

On resize, every single produced SDC was detected, albeit not influence the final output. On Normalise Histograms, 90% of major impact and 75% of minor impact SDCs were detected. Finally, on classify, we detect all errors that impact detection.

Similarly, Fig. 4 shows the percentage of detected SDCs for each hardened version (ECC disabled or enabled). Please note that SDCs caused by kernels which were not hardened are included, hence the lower detection values compared to Fig. 3. However, the vast majority of these errors fall into the no impact category, which are tolerable errors as detection is not affected in any way. In fact, this shows that the unhardened kernels, as expected, produced mostly errors which do not need to be detected, as they do not compromise the final detection. Such a result validates the opinion that these kernels were not worth the overhead caused by future hardening efforts.

Concerning errors which had an impact on the final detection, both versions maintained high-detection rates, detecting more than 80% of major impact SDCs, and more than 65% of minor impact SDCs. Considering the efforts in maintaining the real-time detection capabilities, we believe these results to be a good compromise between SDCs detection capabilities and performance overheads.

© The Institution of Engineering and Technology 2018

1751861x, 2019, 3, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/iet-cdt.2018.5026 by CochraneItalia, Wiley Online Library on [01/03/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

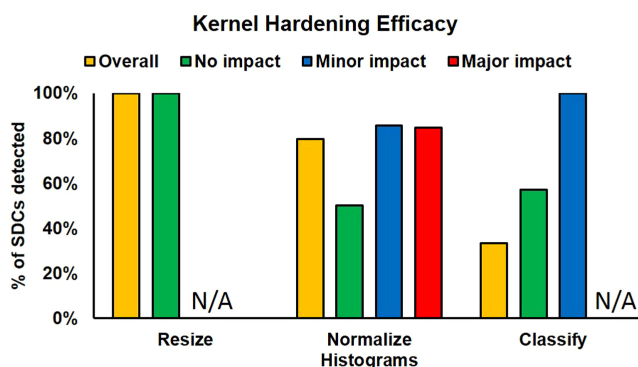

Fig. 3 *Percentage of detected SDCs on the hardened kernels when ECC is enabled or disabled. N/A indicates that no errors were observed*

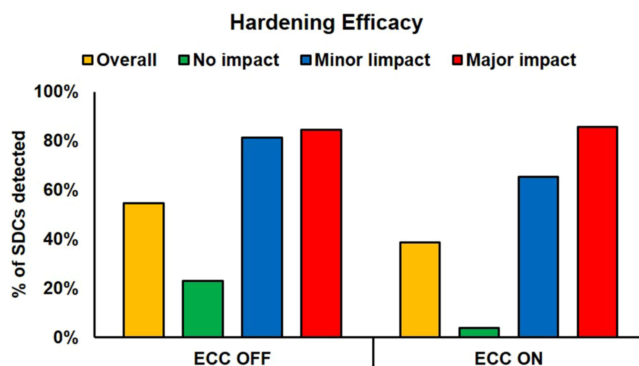

Fig. 4 *Percentage of detected SDCs in the HOG hardened with ECC enabled or disabled*

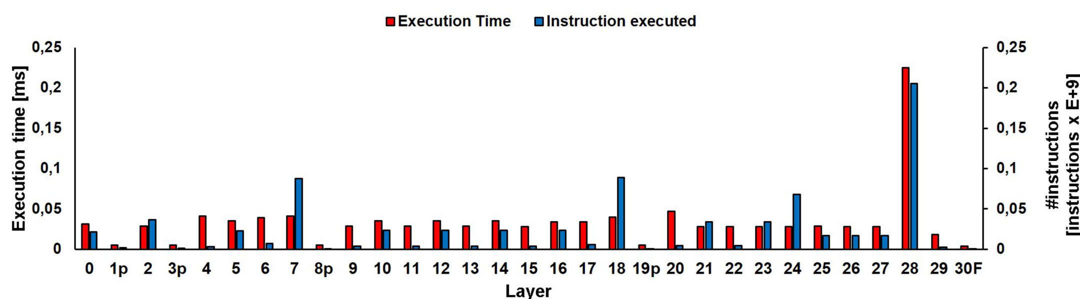

Fig. 5 *Execution time and instruction executed per layer. Execution time is shown in milliseconds, and the instruction executed per layer are shown in E^{+09}*

6.3 CNN layers smart DMR

Nowadays, DNNs are state-of-the-art for object detection applications. The size of a deep neural network can vary from only five layers [44] to a thousand layers [45]. This specific characteristic makes the design of a selective hardening more challenging for CNN compared to other applications, such as HOG.

Following an approach similar to HOG DMR, we intend to design a smart DMR for YOLO by identifying the layers that are worth duplicating (i.e. once duplicated, significantly increase YOLO reliability). To select the layers to duplicate, we use a heuristic technique that contemplates not only the LVF (i.e. the probability for an injection in the layer to propagate to the output generating a minor or major impact error), but also the number of executed instructions in the layer (*#instructions*), the layer execution time, and LVF. We consider the execution time and the number of instructions to evaluate the probability for faults to be generated (the higher the execution time, the higher the fault probability) and the overhead potentially introduced when duplicating the layer (the higher the number of instructions, the higher the overhead). It is worth noting that we consider the number of instructions executed in the layer and not the layer execution time for the overhead as for small layers duplication does not significantly increase execution time. It may then be unfair to consider the overhead for duplicating a (small) layer as zero as double instructions are executed (which increases energy consumption). Moreover, on smaller GPUs in which even a small layer saturates the parallel resources the duplication of a small layer could result in a time overhead.

Fig. 5 shows the layer execution time and the number of operations executed on each layer of YOLO. On a sequential processor, execution time and the number of instructions executed are directly correlated. On a GPU and parallel processors, in general, the relation between execution time and the number of instruction executed depends on the available parallel resources and eventual dependencies. For the specific case of YOLO, as shown in Fig. 5, the execution time for most layers is similar while the number of instructions varies significantly. Layer 28 is the only outlier, as the number of instructions is so high to saturate the available resources.

To identify the layers that are more likely, once duplicated, to improve CNN reliability, we calculate, for each layer, the following ratio:

$$\text{Layer_duplication_impact} = \frac{\#\text{instructions}}{\text{execution_time} * \text{LVF}} \quad (3)$$

the higher the number of instruction, the higher the overhead introduced when duplicating the layer. The higher the denominator, the higher the benefit brought by the layer duplication. Layers with a high *layer duplication impact* may not be right candidates for being duplicated.

© The Institution of Engineering and Technology 2018

1751861x, 2019, 3, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/iet-cdt.2018.5026 by CochraneItalia, Wiley Online Library on [01/03/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

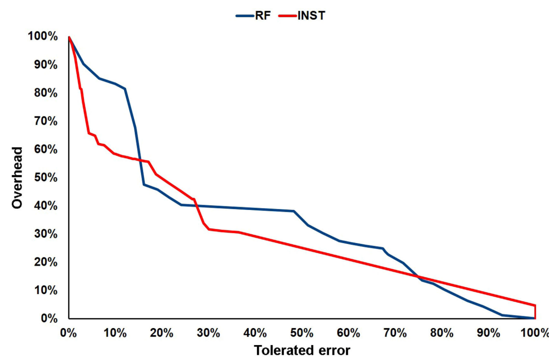

Fig. 6 *INST and RF overhead reduction as the tolerated error increase*

Fig. 6 shows the DMR overhead decreasing as the percentage of tolerated error increases. Both INST and RF injection sites are plotted in Fig. 6. We sort all YOLO layers based on the *Layer_duplication_impact* metric (see (3)). Then, to calculate the percentage of tolerated errors, the layers which have the highest *Layer_duplication_impact* values are removed from the duplication. When a layer is excluded from the DMR, the overall hardening efficacy is decreased, increasing the error. However, the overall duplication overhead is reduced, improving the overall algorithm performance.

Fig. 6 shows that the INST curve decreases faster than the RF curve. It is possible to note that for the INST injections if one tolerates 10% of total errors, the duplication overhead decreases to 57%. If one goes further and relaxes the error tolerance to 20%, the performance overhead will be 45%. While the RF Layer duplication overhead decreases slower than INST, with 16% of tolerated error the overhead will be about 47%.

Let us suppose that we want to duplicate only 30% of our entire neural network, i.e. ten layers in the YOLO case. If we choose to exclude a layer duplication without considering performance, we can remove from DMR those layers which have the smallest LVF. Analogously, if we do not want to consider LVF, we would take out the layers that have the largest execution time. The problem of selecting layers to duplicate must deal with multiple variables, include memory size, FIT, RF usage, number of float point operations, and so on. If one chooses the layers that will be removed from modular redundancy based on a single criterion, the final DMR can neither have optimal overhead nor hardening (i.e. the same for TMR). Selective hardening for DNNs is much harder than HOG, as the problem introduces multiple variables to consider. Our heuristic provides a simple tool to simplify the hardening process for DNN since it considers execution time, instructions executed and LVF.

7 Conclusions

In this study, we proposed the concept of KVF and LVF to describe the reliability of GPUs' applications. We apply KVF to the HOG algorithm and LVF to the YOLO neural network. Using metrics derived from the image processing community, we access how detection is affected by faults injected both at high- and low-architectural levels. Based on the KVFs evaluated with our fault-injection campaigns, we propose an efficient hardening technique for HOG, duplicating only the kernels with high KVF. We then validate the proposed techniques for HOG, through an extensive fault-injection campaign, using SASSIFI.

For YOLO, we proposed a strategy that compromises the reliability and performance characterisations of the algorithm when applied to real-time pedestrian detection on GPUs. We show that hardening real-time object detection applications have a trade-off when selecting which layers will be hardened.

In the future, we plan to extend the LVF study to other DNNs. Our goal is to try to find the optimal hardening for deep learning applications that use CNN. We want to design an algorithm to find optimal layer selection, seeking the best trade-off among accuracy, performance, code size, and LVF.

8 References

[1] Van Essen, B., Kim, H., Pearce, R., *et al.*: 'LBANN: livermore big artificial neural network HPC toolkit'. Proc. Workshop on Machine Learning in High-Performance Computing Environments, ser. MLHPC'15, Oak Ridge, TN, USA, 2015, pp. 5:1–5:6. Available at http://doi.acm.org/10.1145/2834892.2834897

[2] Amin, S.U., Agarwal, K., Beg, R.: 'Genetic neural network based data mining in prediction of heart disease using risk factors'. 2013 IEEE Conf. on Information Communication Technologies, Thuckalay, Tamil Nadu, India, April 2013, pp. 1227–1231

[3] European New Car Assessment Programme: 'Euro NCAP rating review, report from the ratings group', June 2012. Available at http://www.euroncap.com

[4] DeBardeleben, N., Blanchard, S., Monroe, L., *et al.*: 'GPU behavior on a large HPC cluster'. 6th Workshop on Resiliency in High Performance Computing (Resilience) in Clusters, Clouds, and Grids in conjunction with the 19th Int. European Conf. on Parallel and Distributed Computing (Euro-Par 2013), Aachen, Germany, 26–30 August 2013

[5] Wunderlich, H.J., Braun, C., Halder, S.: 'Efficacy and efficiency of algorithm-based fault-tolerance on GPUs'. 2013 IEEE 19th Int. On-Line Testing Symp. (IOLTS), Chania, Greece, July 2013, pp. 240–243

[6] Oliveira, D., Rech, P., Quinn, H., *et al.*: 'Modern GPUs radiation sensitivity evaluation and mitigation through duplication with comparison', *IEEE Trans. Nucl. Sci.*, 2014, **61**, (6), pp. 3115–3122

[7] de Oliveira, D.A.G., Pilla, L.L., Santini, T., *et al.*: 'Evaluation and mitigation of radiation-induced soft errors in graphics processing units', *IEEE Trans. Comput.*, 2016, **65**, (3), pp. 791–804

[8] Dongarra, J., Meuer, H., Strohmaier, E.: 'ISO26262 standard', 2015. Available at https://www.iso.org/obp/ui/\#iso:std:iso:26262:-1:ed-1:v1:en

[9] Laprie, J.C.: 'Dependable computing and fault tolerance: concepts and terminology'. Twenty-Fifth Int. Symp. on Fault-Tolerant Computing, 1995, Highlights from Twenty-Five Years, Pasadena, CA, USA, June 1995, p. 2

[10] Nicolaidis, M.: 'Time redundancy based soft-error tolerance to rescue nanometer technologies'. Proc. 17th IEEE VLSI Test Symp., 1999, Dana Point, CA, USA, 1999, pp. 86–94

[11] Baumann, R.: 'Radiation-induced soft errors in advanced semiconductor technologies', *IEEE Trans. Device Mater. Reliab.*, 2005, **5**, (3), pp. 305–316

[12] Dalal, N., Triggs, B.: 'Histograms of oriented gradients for human detection'. IEEE Computer Society Conf. on Computer Vision and Pattern Recognition, 2005. CVPR 2005, San Diego, CA, USA, June 2005, vol. 1, pp. 886–893

[13] Redmon, J., Divvala, S.K., Girshick, R.B., *et al.*: 'You only look once: unified, real-time object detection', CoRR, vol. abs/1506.02640, 2015. Available at http://arxiv.org/abs/1506.02640

[14] Sivaraman, S., Trivedi, M.M.: 'Looking at vehicles on the road: a survey of vision-based vehicle detection, tracking, and behavior analysis', *IEEE Trans. Intell. Transp. Syst.*, 2013, **14**, (4), pp. 1773–1795

[15] Jia, Y., Shelhamer, E., Donahue, J., *et al.*: 'Caffe: convolutional architecture for fast feature embedding'. Proc. 22nd ACM Int. Conf. on Multimedia, ser. MM'14, Orlando, FL, USA, 2014, pp. 675–678. Available at http://doi.acm.org/10.1145/2647868.2654889

[16] Neagoe, V.E., Ciotec, A.D., Bărar, A.P.: 'A concurrent neural network approach to pedestrian detection in thermal imagery'. 2012 9th Int. Conf. on Communications (COMM), Bucharest, Romania, June 2012, pp. 133–136

[17] Engelmann, C., Ong, H., Scott, S.L.: 'The case for modular redundancy in large-scale high performance computing systems'. Proc. of the IASTED Int. Conf. on Parallel and Distributed Computing and Networks, Honolulu, HI, USA, January 2009

[18] Teifel, J.: 'Self-voting dual-modular-redundancy circuits for single-event-transient mitigation', *IEEE Trans. Nucl. Sci.*, 2008, **55**, (6), pp. 3435–3439

[19] dos Santos, F.F., Rech, P., Oliveira, D., *et al.*: 'taco2016-log-data', 2016. Available at https://github.com/UFRGS-CAROL/taco2016-log-data, 1 June 2016

[20] Nvidia: 'CUDA-GDB: CUDA toolkit documentation'. Available at http://docs.nvidia.com/cuda/cuda-gdb/index.html

[21] Hari, S.K.S., Tsai, T., Stephenson, M., *et al.*: 'SASSIFI: an architecture-level fault injection tool for GPU application resilience evaluation'. Int. Symp. on Performance Analysis of Systems and Software, Santa Rosa, CA, USA, October 2017

[22] Mitra, S.: 'System-level single-event effects'. IEEE Nuclear and Space Radiation Effects Conf., NSREC 2012 Short Course, Miami, Florida, USA, 2010

[23] Rehman, S., Shafique, M., Kriebel, F., *et al.*: 'Reliable software for unreliable hardware: embedded code generation aiming at reliability'. 2011 Proc. Ninth IEEE/ACM/IFIP Int. Conf. on Hardware/Software Codesign and System Synthesis (CODES+ISSS), October 2012, pp. 237–246

[24] Shafique, M., Rehman, S., Aceituno, P.V., *et al.*: 'Exploiting program-level masking and error propagation for constrained reliability optimization'. 2013 50th ACM/EDAC/IEEE Design Automation Conf. (DAC), Austin, TX, USA, May 2013, pp. 1–9

[25] Vallero, A., Savino, A., Politano, G., *et al.*: 'Cross-layer system reliability assessment framework for hardware faults'. 2016 IEEE Int. Test Conf. (ITC), November 2016, pp. 1–10

[26] Ren, X., Ramanan, D.: 'Histograms of sparse codes for object detection'. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR), Portland, OR, USA, June 2013

[27] Fernandes, F., Weigel, L., Jung, C., *et al.*: 'Evaluation of histogram of oriented gradients soft errors criticality for automotive applications', *ACM Trans. Archit. Code Optim.*, 2016, **13**, (4), pp. 38:1–38:25. Available at http://doi.acm.org/10.1145/2998573

© The Institution of Engineering and Technology 2018

1751861x, 2019, 3, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/iet-cdt.2018.5026 by CochraneItalia, Wiley Online Library on [01/03/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

[28] Ciregan, D., Meier, U., Schmidhuber, J.: ‘Multi-column deep neural networks for image classification’. 2012 IEEE Conf. on Computer Vision and Pattern Recognition, Providence, RI, USA, June 2012, pp. 3642–3649

[29] Cireşan, D.C., Giusti, A., Gambardella, L.M., *et al.*: ‘Deep neural networks segment neuronal membranes in electron microscopy images’. Proc. 25th Int. Conf. on Neural Information Processing Systems, ser. NIPS'12, Lake Tahoe, NV, USA, 2012, pp. 2843–2851. Available at http://dl.acm.org/citation.cfm?id=2999325.2999452

[30] Szegedy, C., Toshev, A., Erhan, D.: ‘Deep neural networks for object detection’, in Burges, C.J.C., Bottou, L., Welling, M., *et al.* (Eds.): ‘*Advances in neural information processing systems*’, vol. **26** (Curran Associates, Inc., 2013), pp. 2553–2561. Available at http://papers.nips.cc/paper/5207-deep-neural-networks-for-object-detection.pdf

[31] Ribeiro, D., Mateus, A., Nascimento, J.C., *et al.*: ‘A real-time pedestrian detector using deep learning for human-aware navigation’, CoRR, vol. abs/1607.04441, 2016. Available at http://arxiv.org/abs/1607.04441

[32] Angelova, A., Krizhevsky, A., Vanhoucke, V., *et al.*: ‘Real-time pedestrian detection with deep network cascades’. Proc. BMVC 2015, Swansea, Wales, UK, 2015

[33] Fragkiadaki, K., Zhang, W., Zhang, G., *et al.*: ‘Two-granularity tracking: mediating trajectory and detection graphs for tracking under occlusions’. Computer Vision – ECCV 2012, Florence, Italy, 2012, pp. 552–565

[34] Leveugle, R., Calvez, A., Maistri, P., *et al.*: ‘Statistical fault injection: quantified error and confidence’. 2009 Design, Automation Test in Europe Conf. and Exhibition, Nice, France, April 2009, pp. 502–506

[35] Fang, B., Pattabiraman, K., Ripeanu, M., *et al.*: ‘GPU-Qin: a methodology for evaluating the error resilience of GPGPU applications’. 2014 IEEE Int. Symp. on Performance Analysis of Systems and Software (ISPASS), March 2014, pp. 221–230

[36] Wilkening, M., Sridharan, V., Li, S., *et al.*: ‘Calculating architectural vulnerability factors for spatial multi-bit transient faults’. 2014 47th Annual IEEE/ACM Int. Symp. on Microarchitecture (MICRO), Cambridge, UK, December 2014, pp. 293–305

[37] Sridharan, V., Kaeli, D.R.: ‘The effect of input data on program vulnerability’. Proc. 2009 Workshop on Silicon Errors in Logic and System Effects, ser. SELSE ‘09, Stanford, CA, USA, 2009

[38] Fawcett, T.: ‘An introduction to ROC analysis’, *Pattern Recognit. Lett.*, 2006, **27**, (8), pp. 861–874

[39] Alippi, C., Piuri, V., Sami, M.: ‘Sensitivity to errors in artificial neural networks: a behavioral approach’, *IEEE Trans. Circuits Syst. I, Fundam. Theory Appl.*, 1995, **42**, (6), pp. 358–361

[40] Bettola, S., Piuri, V.: ‘High performance fault-tolerant digital neural networks’, *IEEE Trans. Comput.*, 1998, **47**, (3), pp. 357–363

[41] Distante, F., Piuri, V.: ‘Anna-use of pads for artificial neural network analysis: the defect and fault tolerance issue’. 1991 Proc. Advanced Computer Technology, Reliable Systems and Applications, Bologna, Italy, May 1991, pp. 537–540

[42] Piuri, V.: ‘Analysis of fault tolerance in artificial neural networks’, *J. Parallel Distrib. Comput.*, 2001, **61**, (1), pp. 18–48

[43] Oliveira, D., Pilla, L., Hanzich, M., *et al.*: ‘Radiation-induced error criticality in modern HPC parallel accelerators’. 2017 IEEE Int. Symp. on High Performance Computer Architecture (HPCA), Austin, TX, USA, 2016

[44] Lecun, Y., Bottou, L., Bengio, Y., *et al.*: ‘Gradient-based learning applied to document recognition’, *Proc. IEEE*, 1998, **86**, (11), pp. 2278–2324

[45] He, K., Zhang, X., Ren, S., *et al.*: ‘Identity mappings in deep residual networks’, CoRR, vol. abs/1603.05027, 2016. Available at http://arxiv.org/abs/1603.05027

© The Institution of Engineering and Technology 2018

1751861x, 2019, 3, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/iet-cdt.2018.5026 by CochraneItalia, Wiley Online Library on [01/03/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License