A Methodology for Selective Protection of Matrix Multiplications: a Diagnostic Coverage and Performance Trade-off for CNNs Executed on GPUs

Javier Fernández, Irune Agirre, Jon Perez-Cerrolaza
Ikerlan Technological Research Center, Basque Research and Technology Alliance (BRTA), Spain
javier.fernandez,iagirre,jmperez@ikerlan.es

Jaume Abella, Francisco J. Cazorla
Computer Architecture-Operating Systems department
Barcelona Supercomputing Center, Spain
jaume.abella, francisco.cazorla@bsc.es

***Abstract*—The ability of CNNs to efficiently and accurately perform complex functions, such as object detection, has fostered their adoption in safety-related autonomous systems. These algorithms require high computational performance platforms that exploit high levels of parallelism. The detection, control and mitigation of random errors in these underlying high computational platforms become a must according to functional safety standards. In this paper, we propose protecting, with a catalog of diagnostic techniques, the most computationally expensive operation of the CNNs, the matrix multiplication. However, this protection entails a performance penalty, and the complete CNN protection may be unaffordable for those systems operating with strict real-time constraints. This paper proposes a three-stage methodology to selectively protect CNN layers to achieve the required diagnostic coverage and performance trade-off: i) sensitivity analysis to misclassification per CNN layers using a statistical fault injection campaign, ii) layer-by-layer performance impact and diagnostic coverage analysis, and iii) selective layer protection. Furthermore, we propose a strategy to effectively compute the achievable diagnostic coverage of large matrices implemented on GPUs. Finally, we apply the proposed methodology and strategy in Tiny YOLO-v3, an object detector based on CNNs.**

***Index Terms*—Safety, GPU, CNN, protection, matrix multiplication, fault detection**

I. Introduction

Techniques based on deep neural networks (DNNs), and more particularly on convolutional neural networks (CNNs), are extensively used to deploy complex functionalities such as those required by autonomous systems, e.g., object detection tasks [1]. These highly parallelized algorithms handle massive volumes of data and impose high-performance demands on the underlying hardware where they are deployed. CNNs are commonly executed on graphics processing units (GPUs) that provide the required data parallelism through their thread-level parallelism. However, their high parallelism and memory hierarchy can spread a single fault to multiple, jeopardizing the proper reliable execution [2]. For instance, a single bit-error can lead to a dangerous misclassification of objects (e.g., not detecting pedestrians and traffic lights) [3].

Depending on the application criticality, these errors can lead to catastrophic consequences. Specially in safety-related systems, where random errors must be detected, controlled and mitigated according to functional safety standards such as IEC 61508 [4].

This paper proposes a step-by-step methodology for selective protection of matrix-matrix multiplications (MMMs) to achieve the required diagnostic coverage (DC) with a performance trade-off for CNN deployed on GPUs. These arithmetic operations are the backbone of CNNs and take most of their execution time [5]. A recent paper proposes creating an array of execution signatures using a catalog of diagnostic techniques implemented at the arithmetic operation level to protect the MMM according to functional safety standards [3]. This catalog provides different degrees of DC incurring an execution time penalty (performance penalty) according to the implementation and the dimensions of the MMM. Therefore, a trade-off between performance and DC shall be analyzed to apply this safety diagnostics catalog. However, these safety-related systems commonly operate in real-time, having to cope with stringent timing constraints. The resulting slowdown for the highest DCs may be unaffordable when applied to the complete CNN. In those cases, we propose to reduce the performance impact by considering two new factors in the selective protection of each CNN layer: i) its sensitivity to misclassification and ii) its performance penalty for the complete CNN. We enumerate the main contributions of this paper as follows:

- We define a step-by-step methodology to selectively protect CNNs deployed on GPUs by implementing diagnostics in the MMM [3]. This methodology has three stages: i) CNN's sensitivity to misclassification analysis, ii) layer-by-layer performance impact and DC analysis, and iii) selective CNN layer protection.
- We present a strategy to efficiently determine the achievable DC of large matrices implemented on GPU. This strategy is based on the DC analysis of the grid of thread blocks in which they are launched on the GPU.
- We apply the proposed methodology and strategy in an object detection application based on Tiny YOLO-v3.

The rest of the paper is structured as follows: Section II outlines the necessary background. Section III describes our methodology for protecting CNNs and proposes a strategy to evaluate the achievable DC of large matrices. Then, Section IV evaluates our methodology in Tiny YOLO-v3 and Section V presents an analysis of the related work. Finally, in Section VI we draw conclusions and provide future research lines.

II. Background

This section explains basic concepts such as safety certification, the object detector Tiny YOLO-v3 and the proposed CUTLASS MMM to be included in the Tiny YOLO-v3 CNN. Finally, we outline a solution to provide the MMM with diagnostic capabilities.

A. Safety Certification

In safety-critical systems, regulatory authorities require safety certification, which is commonly achieved by adhering to functional safety standards such as IEC 61508 [4]. These standards set out the requirements, techniques, and measures to avoid, mitigate and detect random hardware and systematic errors according to the safety integrity level (SIL). The SIL defines a safety integrity range where four is the highest and one is the lowest level (SIL1... SIL4), with the diagnostics and safety measures being more demanding at the highest levels.

"Diagnostic Coverage (DC) denotes the effectiveness of diagnosis techniques to detect dangerous errors, expressed in coverage percentage with respect to all possible dangerous errors" [6]. The required DC range is imposed by the SIL and hardware fault tolerance (HFT) (IEC 61508-2 Table 3). IEC 61508 classifies DC in three ranges: i) low ($60\,\% <$ DC $< 90\,\%$), ii) medium ($90\,\% \leq$ DC $< 99\,\%$) and iii) high ($99\,\% \leq$ DC). The implementation of DC techniques based on software becomes relevant in order to diagnose the proper operation of the hardware components periodically (e.g., against permanent errors) or to complement hardware built-in diagnosis (usually ranked as low/medium DC) [2], [6].

B. Object Detection: Tiny YOLO-v3

You only look once (YOLO) is a multi-scale object detector whose backbone network is Darknet [7], a CNN coded in C and CUDA. There are several versions of this object detector [8]–[10] and all involve the use of three types of layers: i) the convolutional layers that extract the features from the input image, ii) max-pooling layers reducing the feature map and iii) fully connected layers classifying the input. Tiny YOLO-v3 is the reduced version of YOLO-v3 with a smaller backbone that splits into a lower number of convolutional layers (13 nested layers). We depict its architecture at the center of Fig. 2.

The MMM is the most relevant and time consuming function for the CNN. It entails 67% of the CNN execution time [5]. Darknet offers several implementation options according to the deployment platform. In this paper, we focus on the CUTLASS library [11].

C. CUTLASS

CUTLASS is the open-source library provided by NVIDIA to implement high-performance matrix-multiplication in GPU-based implementations [11]. It is coded in CUDA and C++. This low-level library decomposes this algebraic operation into software modules while abstracting the designer using C++ template classes. Its hierarchical decomposition into device, blocks, warp, and threads eases the understanding of the data movement. Fig. 1 depicts the memory transfer at each stage of the matrix multiply–accumulate ($C \mathrel{+}= A \times B$).

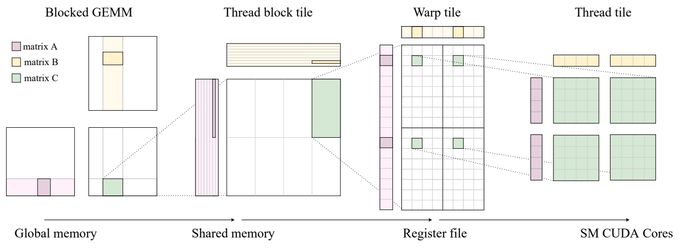


Fig. 1: CUTLASS GEMM hierarchy [11].

D. Execution Signature

Techniques based on creating an execution signature (ES) or Frame Check Sequence (FCS) are widely employed to assure integrity in data transmission. In that way, the effectiveness of diagnostics such as XOR, one's and two's complement, Fletcher, and cyclic redundancy check (CRC) have been widely studied [12], [13].

In a recent paper [3], authors provide CUTLASS with a catalog of diagnostic techniques to compute an array of ESs computed at different levels of the MMM. Algorithm 1 presents the CUTLASS MMM implementation at thread-level (t) and denotes the diagnostics as internal (I), medium (M) and external (E) depending on the loop where they are implemented. These diagnostics compute an array of ESs that must be bit-for-bit exact with the ES array of the reference execution (golden reference ES array). For that, the authors propose two architectural patterns, one based on redundancy and the other based on a periodic diagnosis of the components involving the MMM. This latter pattern compares the array of ESs against that obtained from predefined input values.

Algorithm 1 MMM loops computed at thread-level

```
1:  for $k$ from 0 to $K_t$ do
2:      External loop statements
3:      for $n$ from 0 to $N_t$ do
4:          Intermediate loop statements
5:          for $m$ from 0 to $M_t$ do
6:              Internal loop statements (MMM computation)
7:              [Internal diagnostic (I)]
8:          end for
9:          [Medium diagnostic (M)]
10:     end for
11:     [External diagnostic (E)]
12: end for
```

III. Methodology to selectively protect CNNs

In this section, we present a methodology for the selective DC-level protection of MMM-based CNNs applied in safety-related applications. The methodology, shown in Fig.2, seeks to find a trade-off between DC and the performance impact that diagnostic techniques incur. The layered architecture of CNNs is the main foundation of the methodology (see Fig.2). In this layered CNN architecture, the impact of a error is highly dependant on the layer where the error occurs and its propagation through the CNN. The proposed methodology seeks to identify the most misclassification-prone layers to provide selective CNN protection through three stages.

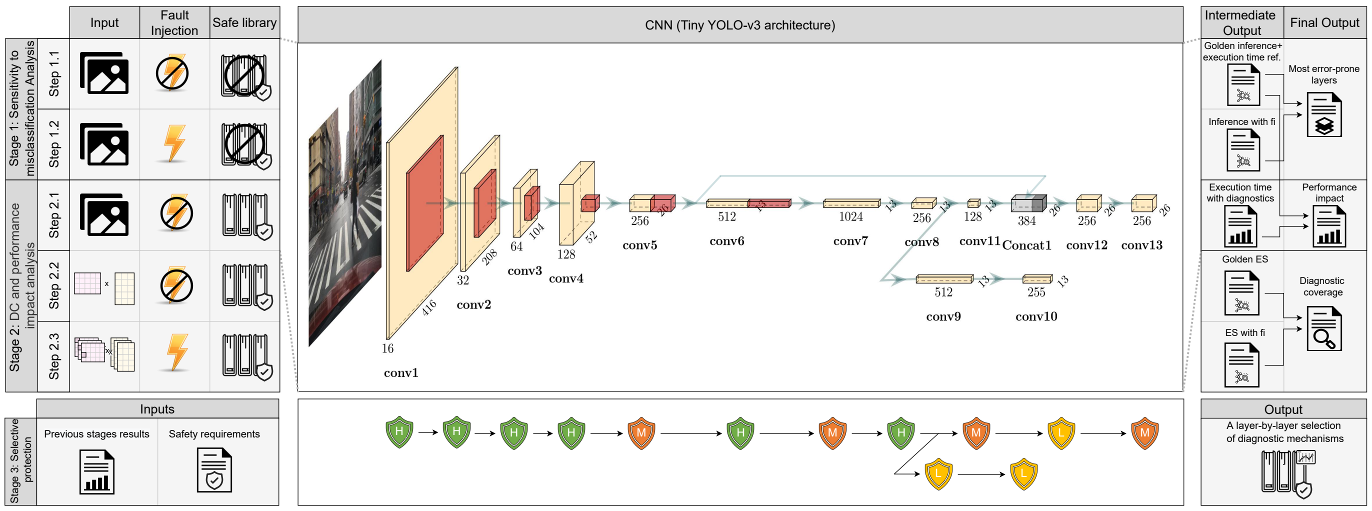


Fig. 2: Selective CNN layer protection methodology

In Fig.2, we depict the input and intermediate and final outputs from each stage. Finally, we propose a strategy to obtain the DC of large matrices in an efficient way, significantly reducing the fault[1] injection campaign effort.

A. First Stage: CNN's Sensitivity to Misclassification Analysis.

This section evaluates the behavior of CNN layers against single-bit errors affecting weights in classification tasks. We focus on identifying the most misclassification-prone layers by performing a fault injection campaign in weights. We define the first two steps as follows (see Fig.2):

Step 1.1 performs the classification with a set of predefined images to obtain a reference inference value (golden reference) for each image and measures the execution time required by each layer.

Step 1.2 executes a fault injection campaign with the same set of images. Exhaustive fault injection at bit level is hardly manageable due to the required testing time, even in small CNNs. For example, Tiny Yolo-v3 weight configuration file has 9.06e9 bits [8] that would require the same amount of single-bit error injection executions. According to our GPU-based implementation, the execution time of a single image classification is about 45,82 ms. Therefore, performing an exhaustive single-bit fault injection campaign with a single Xavier Nx GPU would take approximately 13,163 years.

CNN-based applications such as Tiny YOLO-v3 commonly rely on floating-point data types. In these data types, the impact in the classification varies depending on the bit position affected by the error [14]–[16]. Errors affecting the weights in the exponent bits have the greatest potential impact on generating incorrect classifications in multiplication operations (with respect to mantissa and sign bits) [14]–[16]. So, we propose to concentrate the fault injection campaign on the exponent bits of the weights. After defining classification features such as the confidence level, error margin, and the total number of possible errors in the weights, we compute a statistically representative random sampling size as stated in [17].

Then, we compare the inference results (with fault injections) against the golden reference (Step 1.1) using a misclassification criterion, obtaining the most misclassification-prone layers (final output of this first stage). We propose a semantic comparison of the detected classes against the golden reference according to the specific CNN application. E.g., in object detection applications, such as YOLO, this criterion would be based on defining acceptance ranges for features such as the accuracy of the detected classes, box size (height and width), or location of the box center for each of the detected classes.

B. Second Stage: Layer-by-layer Performance Impact and DC

After the identification of the most misclassification-prone layers, this stage evaluates the performance impact incurred by the inclusion of diagnostics in each layer and the maximum achievable DC. To this end, we rely on the safe MMM library that provides a diagnostic techniques catalog [3]. This stage follows the next three steps depicted in Fig. 2:

Step 2.1 evaluates the execution time penalty incurred by each diagnostic technique included in the safe catalog. To this end, we apply the diagnostics in all CNN layers and measure the execution time of each one. This process is repeated for the different types of protection techniques provided in the diagnostics catalog. The performance penalty is then determined as a ratio between the mean execution time of each CNN layer with the diagnostic techniques divided by the mean execution time to compute the same layer without diagnostics (previously measured in Step 1.1).

Step 2.2 computes an array of golden ESs by including the safe library of diagnostic techniques in the MMM execution of each layer (without fault injections). This value is then taken as a golden reference in next steps to determine whether injected errors are detected by diagnostics (bit identical) or not.

Step 2.3 performs a fault injection campaign and obtains the DC associated with each layer. However, an exhaustive fault injection campaign may be unaffordable for large matrices due

[1]Since the injection consists of flipping bits, they can be considered faults or errors. Hence, we could use both terms interchangeably in our discussion. However, we generally refer to them as errors except when talking about injection since *fault injection* is the most common term in the domain.

to the required number of iterations to cover all input combinations. That is why, in Subsection III-D we propose a strategy to compute the DC of large matrices implemented on GPUs, relying on the CUDA programming model characteristics.

C. Third Stage: Selective DC-level Protection

The performance impact and DC analysis of all CNN layers, Step 2.1 and Step 2.3, are followed by analyzing the most appropriate diagnostics layer-by-layer. The SIL of the safety function and the HFT of the system determine the minimum DC range to achieve. Thus, the diagnostics with the lowest performance impact that achieves at least the imposed DC is selected in a layer-by-layer process.

However, many safety-related applications, especially those in real-time systems, have stringent timing requirements. In those cases, the selective DC-level protection of the CNN is the most reasonable option instead of complete protection of all CNN layers. The results obtained in the analysis of the CNN's sensitivity to missclassification determine the layers less likely to cause misclassification. We propose to selectively protect each layer based on these results and taking into account the percentage of the execution time of each layer according to the complete CNN. The selective protection depends on CNN's propensity to misclassifying, and it is always subject to a final fault injection campaign to simulate the achieved DC according to the selection. As a representative example, we have depicted in Fig. 2 a particular diagnostics selection in the form of shields including the chosen DC range: i) low (L), ii) medium (M) and iii) high (H).

Note that a valid solution is always reached since at least one of the diagnostics provides high DC, so such diagnostic in all layers would be a valid solution from the DC standpoint. Hence, the goal is preserving a sufficiently high DC while minimizing the performance impact due to diagnostics according to the maximum affordable by the specific application.

D. DC Analysis in Large Matrices

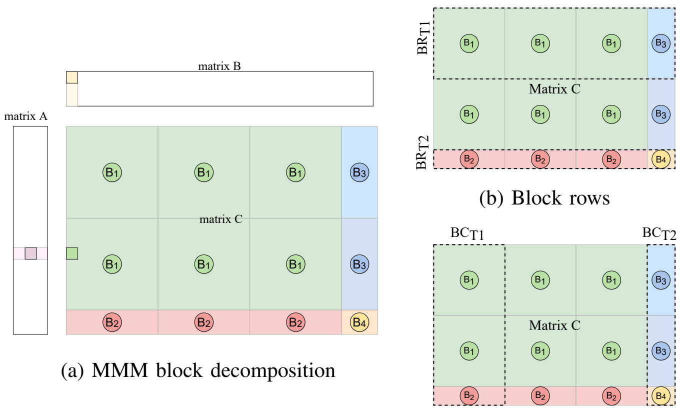
This subsection decomposes into two parts. First, we explain the base of our strategy to evaluate the DC of large matrices and the main factors that affect the effectiveness of the diagnostics. Then, we describe how to compute the final DC according to the source of the error in the MMM implementation.

1) Block Decomposition: Before launching a CUDA kernel, it is necessary to define built-in variables to decompose the function to parallelize into a grid of blocks of threads (see Fig.1 in background). These blocks execute according to the single instruction multiple thread (SIMT) model. That is, all active threads process the data in the same way. Since those threads access shared memory (block dependant memory), we can consider that all blocks with the same number of active threads handle the same amount of data independently. This independence among blocks is the cornerstone of our strategy to evaluate the achievable DC of large MMMs since we can compute it from the DC evaluation of smaller blocks.

The following factors shall be considered in this strategy:

- $F1$ *Block parity*: techniques such as XOR, one's, and two's complement checksums do not detect errors affecting the same bit position of an even number of data words.
- $F2$ *Block dimensions*: determines the amount of data computed by each block. A higher amount of data to be protected may lead to lower diagnostics effectiveness [12].
- $F3$ *Error source*: errors affecting global memory spread into several ones instead of those appearing in a register, which can affect a single arithmetic operation [2].
- $F4$ *Fault type*: the effectiveness of the diagnostics varies according to the type of errors (e.g., single bit, burst, random errors). This paper focuses on single-bit errors.
- $F5$ *MMM implementation*: the specific software implementation and the location of diagnostics instrumentation in end user's code are crucial in the effectiveness of the diagnostics [3], [18].

Fig. 3a depicts a MMM with a representative matrix dimensions (e.g., output matrix $C = 133 \times 200$ with a grid of threads blocks $<64, 64, 8>$) to explain the DC computation and the influence of previously enumerated factors. As explained in the background, the CUTLASS library decomposes the matrix into blocks to run the MMM in the GPU. We denote as B_1 those blocks whose dimensions match the size of the blocks launched to the GPU, B_2 as those with an equal number of columns but different rows, B_3 if the number of rows matches but columns differ, and finally, B_4 if both the number of rows and columns differ.



(a) MMM block decomposition

(b) Block rows

(c) Block columns

Fig. 3: MMM decomposition into blocks

2) DC Computation per Error Source: Depending on the error source ($F3$), the computation of the DC differs. We propose two scenarios according to this source:

a) Faults injected at the arithmetic operation level or at register level: In this scenario, the final DC can be obtained after independently computing the DC associated with previously defined blocks. As Eq. 1 summarizes, this value is computed as the ratio between the sum of errors detected (N_{det}) by each block divided by the total number of fault injections (N_{fi})[2], being i the type of block previously defined (from $B1$-$B4$):

[2]In an exhaustive fault injection campaign at bit-level, this value is N times the number of arithmetic operations. Being N the size of the data type in which the matrices are stored.

$$DC = \frac{\sum_{i=1}^{4} N\_blocks_{Bi} \times N_{det\_Bi}}{N_{fi}} \tag{1}$$

In this scenario, the effectiveness of XOR, one's, and two's complement checksums is not affected by the block parity factor ($F1$) since the error is not spread across multiple blocks, and is not affected by the amount of data to be protected ($F2$).

b) Fault injected at the global memory level: This kind of errors entails changes in the DC computation since the error count diverges from the previous scenario. In this case, the errors injected in the input matrices A and B affect several blocks. Therefore, a proper DC computation requires verifying if previous blocks have already counted the detected errors. To do this, we propose distinguishing between errors detected from the fault injection in A (Det_A) and B (Det_B) matrices. Faults injected in A affect block rows (BRs) and those injected in B matrix affect block columns (BCs), as shown in Fig. 3b and Fig. 3c, respectively. In both cases, we define two types of blocks: i) type 1 ($T1$) as those block rows/columns (BR$_{T1}$/ BC$_{T1}$) in which at least one block is $B1$, and ii) type 2 ($T2$) as those block rows/columns (BR$_{T2}$/ BC$_{T2}$) in which none is $B1$. The number of detected errors can be computed according to Eq. 2 and Eq. 3:

$$Det_A = (B1^{\dagger}_{det_A} + B3^{\dagger}_{det_A}) \times N\_BR_{T1} + B2^{*}_{det_A} + B4^{*}_{det_A} \tag{2}$$

$$Det_B = (B1^{\otimes}_{det_B} + B2^{\otimes}_{det_B}) \times N\_BC_{T1} + B3^{\circledast}_{det_A} + B4^{\circledast}_{det_A} \tag{3}$$

In the above equations, we include the same superscripts for two blocks to indicate that those errors detected by one do not have to be accounted for by the other. E.g., $B1^{\dagger}_{det_A}$ and $B3^{\dagger}_{det_A}$ denote the number of detected errors in $B1$ and $B3$ respectively when performing a fault injection campaign in matrix A. Since they belong to the same BR, we indicate by means of the same superscript (†) that they are complementary and do not have to account the same errors injected in A. For that, we store the index positions of the errors detected by each block in an array that the complementary block will check. Eq. 4 presents the final step to compute the DC. As we are performing a bit-level fault injection campaign, the number of injected errors depends on the dimension of the input matrices and the data size of the data type employed ($data\_size$).

$$DC = \frac{Det_A + Det_B}{N_{fi}} = \frac{Det_A + Det_B}{(M+N) \times K \times data\_size} \tag{4}$$

In those errors appearing in the global memory for the previously defined $F1$ and $F5$ factors, the other (smaller) block matrices might detect additional errors with respect to those detected by $B1$. As $B1$ dimensions are multiple of 32 (number of threads per warp) for performance reasons by design, the amount of data protected according to our MMM implementation ($F5$) is multiple of an even number. Therefore, the effectiveness of some algorithms can be affected by $F1$ since any number multiplied by an even number leads to an even result. Those other blocks protecting an odd number of data can additionally detect those errors not detected by $B1$. That occurs if the following conditions are met:

1) Diagnostics in the internal loop (I): the number of iterations computed by each thread is odd. That is, M_t, N_t, and K_t take odd values. Note that, since $B3$ shares the $B1$ row dimensions, and $B2$ the column dimensions, these blocks do not detect additional errors since $B1$ dimensions are multiple of 32.
2) Diagnostics in the intermediate loop (M): The number of protected data types is odd only if N_t and K_t are odd.
3) Diagnostics in the external loop (E): K_t is odd, independently of M_t and N_t.

IV. Evaluation

In this section, we evaluate our methodology and present the results for each of the stages. We initially evaluate the performance impact incurred by the adoption of each of the diagnostic techniques from the safe MMM library in Tiny YOLO-v3 layers, as well as their achievable DC. Finally, according to these results, we discuss the most appropriate diagnostics to perform selective protection of Tiny YOLO-v3 CNN based on the target SIL.

A. Experimental Set-up

We collect the performance impact and the DC analysis in an NVIDIA Xavier NX embedded GPU. We replicate the experimental set-up presented in [3] employing the clang compiler with CUDA (both version 10) over the Ubuntu operating system running a PREEMPT-RT patch to minimize system interference. With this intention, we also configure one of the NVIDIA Carmel ARM cores to execute the program with the highest priority and we program the system to operate at maximum frequency. Finally, we launch a single MMM stream to avoid interference and uncertainty in the execution order (which can affect the proper execution of the ESs array) [19].

We denote the layers from Tiny YOLO-v3 as $L1$-$L13$, where the number refers to the order position in the CNN. Table I gathers the CNN's configuration through the parameters M, N and K, being A=$M \times K$, B=$K \times N$ and C=$M \times N$.

B. Stage 1: CNN's Sensitivity to Misclassification Analysis

We perform a statistical fault injection campaign on the exponent bits of the weights to analyze the sensitivity of the CNN's layers to misclassification when they are affected by single errors. First, we compute a statistically representative random sampling size ($Injections$) with a 95% confidence level and a 1% error margin, taking as reference the number of potential faults targeting each layer ($Faults\ target$). As we focus on the exponent bits (8 bits) of the weights, the number of faults targets of each layer is $faults = N \times K \times 8$. Table I summarizes the above mentioned features.

Then, we perform the fault injection campaign for five images extracted from Berkeley dataset [20]. As a classification criterion, we consider that the objects are detected if, by comparing against the golden result: 1) accuracy differs less than 15%, 2) width and height of the boxes vary less than 25 pixels, and 3) the central point of the box is less than 50 pixels away. Note that this criterion depends on the specific application (e.g., resolution of the input image). Applying this criterion and performing the average across the five images,

TABLE I. LAYER-BY-LAYER SIZE, TOTAL ERRORS AND STATISTICALLY REPRESENTATIVE FAULT INJECTIONS PER LAYER

Features	**Tiny Yolo-v3**											
	L1	**L2**	**L3**	**L4**	**L5**	**L6 & L9**	**L7**	**L8**	**L10**	**L11**	**L12**	**L13**
M	173056	43264	10816	2704	676	169	169	169	169	169	676	676
N	16	32	64	128	256	512	1024	256	255	128	256	255
K	27	144	288	576	1152	2304	4608	1024	512	256	3456	256
Faults target	3456	36864	147456	589824	2359296	9437184	37748736	2097152	1044480	262144	7077888	522240
Injections	413	3114	6314	8497	9301	9526	9584	9265	8946	7427	9501	8372
Timing (%)	20,30	16,75	8,60	4,71	3,93	6,21	18,40	1,75	1,07	0,71	10,10	1,28

we obtain the results collected in Table II. In this table, we depict the undetected objects or false negatives (FNs) and the average of new objects that appear or false positives (FPs) in percentage terms.

TABLE II. LAYER-BY-LAYER ANALYSIS OF ITS SENSITIVITY TO MISCLASSIFICATION

Layer position	FNs (%)	FPs (%)
L1	96,39	0,16
L2	90,64	1,64
L3	93,74	1,19
L4	92,29	0,85
L5	83,37	2,70
L6	96,79	1,45
L7	87,06	1,16
L8	99,59	0,04
L9	59,75	6,50
L10	55,16	6,50
L11	74,34	0,07
L12	55,70	27,63
L13	66,80	27,29

In FNs column, we observe that modifying a single bit in the exponent of a weight leads to a failure to detect most of the objects regardless of the layer where the weight is used. Moreover, we observe that the impact of the initial layers (from $L1$-$L8$) on the classification is higher in contrast with the final layers ($L9$-$L13$). As mentioned in the background, YOLO is a multi-scale object detector that performs multi-layer feature extraction. Tiny YOLO-v3 performs this feature extraction from $L13$ and $L10$ layers (see Fig 2). These layers belong to different branches whose origin is the output from $L8$. Errors in the initial layers propagate virtually to all outputs causing catastrophic errors in the form of very high FNs. Instead, errors in the final layers have a lower impact due to the lower propagation of errors, and is more frequent the case where only the bounding box or object location is affected, which translates into an FP if impact is large enough, or into no semantic error if impact is small. Note that $L11$ errors do not produce as many FNs and FPs as the rest of the final layers since the concatenation with $L5$ and the absence of errors on the other branch ($L9$ and $L10$) mitigate their appearance. In addition, it should be noted that errors in $L10$ and $L11$ produce a greater number of FPs than those in $L9$ and $L10$ because the scale of the former is larger than that of the latter, detecting smaller objects with smaller scales.

C. Stage 2: Layer-by-layer Performance and DC Analysis

In this subsection, we evaluate the achievable DC and performance impact associated with the inclusion of the diagnostics catalog in each of the Tiny YOLO-v3 layers.

1) Step 2.1: Performance Impact: The work in [3] emphasizes the importance of memory management in GPU platforms, which usually becomes the main bottleneck in those highly-parallelized platforms [2]. Hence, the following memory usage optimization in the implementation of the diagnostics catalogue is proposed and evaluated by launching a set of experiments that replicate those carried out in paper [3]. In contrast with them, we include an intermediate allocation in the shared memory of the ESs between the global memory and the register for the sake of performance, as seen in Fig. 4.

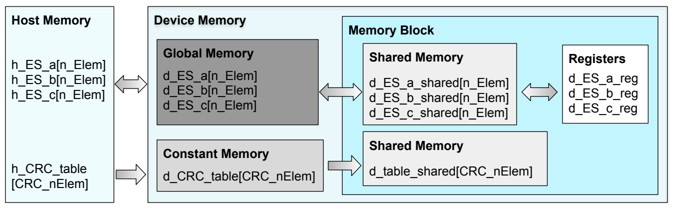


Fig. 4: Memory hierarchy

This enhancement is measured as a ratio between the execution time presented in [3] and those obtained with the new memory hierarchy. We obtain an enhancement ranging between 15% and 65% execution time reduction using the maximum compiler optimization.

We include in Table I in percentual terms the execution time breakdown across layers for the complete CNN performed without diagnostics and with maximum compiler optimizations. These values will be decisive in Stage 3. For instance, $L1$ is the most time consuming layer accounting for 20.3% of the overall execution time of the CNN.

Then, we measure the performance impact incurred by adopting the diagnostics catalog following Step 2.1 of the methodology. In this experiment, we measure the layer-by-layer execution time required to predict single images with a set of a thousand images extracted from the Berkeley DeepDrive dataset [20]. Additionally, we dispensed with the initial hundred timing measurements to avoid the impact of problems related to cache cold-starting, and the delays involved with the initial kernel launches [21].

We perform the first set of experiments disabling the compiler optimizations (compiler option -O0) to reduce the safety challenges that optimizations may entail (e.g., altering the control flow). We present the performance impact layer-by-layer of Tiny YOLO-v3 CNN in Fig. 5 (results normalized with respect to the layer execution without diagnostics). It should be noted that $L6$ and $L9$ are depicted in the same figure since they have identical dimensions, and hence, identical execution times.

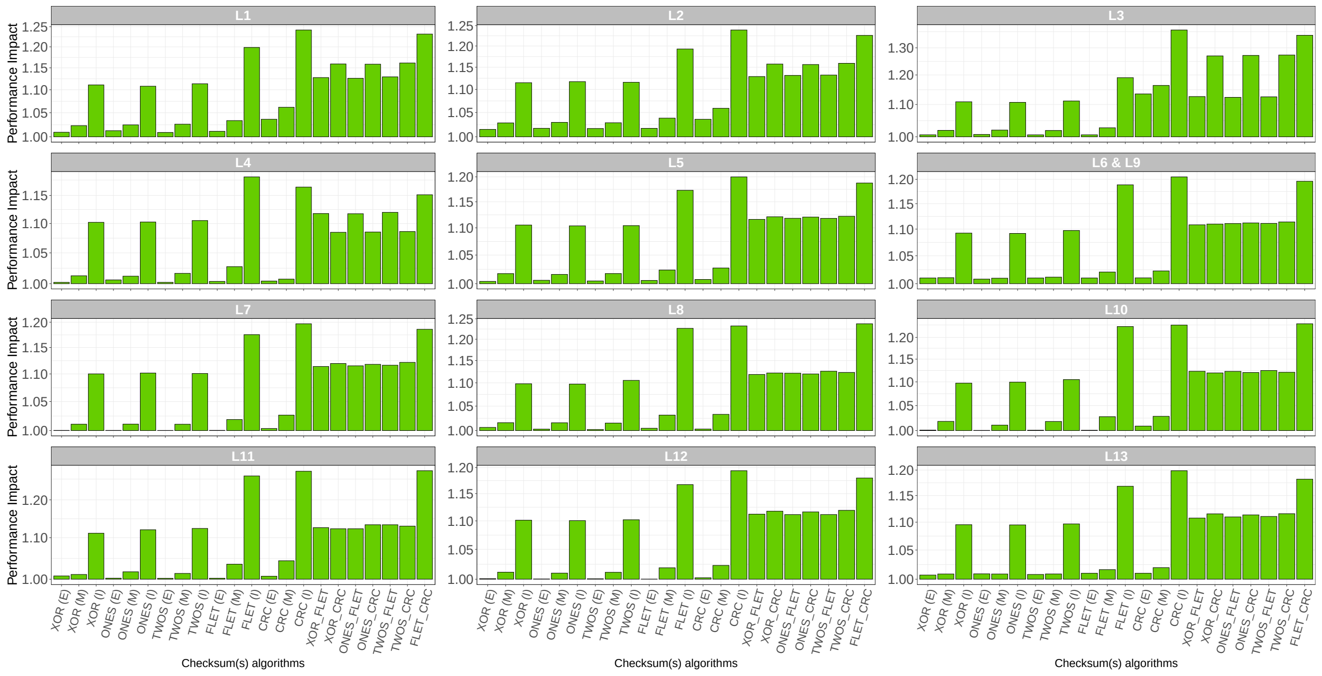


Fig. 5: Layer-by-layer performance impact without compiler optimizations (-O0)

From these results, we observe that the layer incurring the highest performance impact is $L3$ (varying from 1.01 to 1.37), with the lowest performance impact values in $L4$ (from 1.002 to 1.18). Hence, relative performance impact is quite insensitive to layer dimensions. Overall, the performance impact incurred by the DC catalog is affordable without compiler optimization.

However, safety-related applications in real-time systems usually have stringent timing constraints to meet. For those applications, the no compiler optimizations may not be an option. Thus, this has lead us to perform a second set of experiments configuring the maximum compiler optimization (compiler option -O3). Fig. 6 presents the obtained results layer-by-layer. These results evidence that diagnostics based on Fletcher and CRC in the most internal loop have significantly higher performance impact than the rest. In fact, this penalty difference has motivated us to break the performance impact axis (y-axis) in every graph depicted in Fig. 6. Comparing against performance impact experiments without optimization, we notice a significant impact increase. We observe the minimum performance impact in layer $L1$ (ranging from 1,02 to 82,5, hence increasing execution time by more than 82x in the worst case) and the maximum in $L7$ (from 1,04 to 171,5). This increase is associated with the high optimization of the MMM on GPUs. Including a new data (array of ESs) in the computation exacerbates one of the main problems associated with GPU platforms, the bottleneck created for data access. This bottleneck is the main reason for the high-performance impact of the CRC implementation since this diagnostic is based on memory access. Moreover, Fletcher diagnostic has a similar performance to CRC. However, a key reason for this timing penalty lies in using the modulo operator, which is highly inefficient in GPU implementations.

2) Step 2.2- Step 2.3: Diagnostic Coverage: As explained in subsection III-D, we build on the DC evaluation of the single blocks to calculate the global DC. All the experiments are configured with a grid of blocks of $<64, 64, 8>$ for $B1$ blocks (see Figure 3a), except those related to $L1$ and $L2$ that employ $<64, 16, 8>$ and $<64, 32, 8>$ respectively. We use a specific grid for these matrices for performance reasons since, with the initial grid configuration, the execution would use non-active threads. As explained before, $B2$, $B3$ and $B4$ blocks have fewer rows, columns or both since they are at the boundaries of the kernel. Therefore, according to the layer dimensions summarized in Table I, we present in Table III the individual grid of thread blocks dimensions into which layers are decomposed. Additionally, we include the number of $T1$ block columns/rows according to each layer and the selected grid of threads blocks employed. Note that the default block size, $<64, 64, 8>$, has been chosen as it is among those chosen by CUDA to maximize performance in NVIDIA GPUs, and it is small enough to allow decomposing most layers' computations into blocks of this size.

By dividing grids of thread blocks into individual blocks, we obtain that block dimensions repeat many times inside a given layer and across layers. Hence, we only need to perform fault injection once per unique block, and results are reused for all instances of any given block across layers. In Table IV, we summarize the resulting unique blocks, the number of injected errors in input matrices A (Injections$_A$) and B (Injections$_B$) and the layers in which they are used. For some blocks, errors detected in $B2$ and/or $B3$ may have been already detected in $B1$ or $B4$. In those cases, we carefully avoid counting error detections twice. The specific blocks where this effect can happen are marked with an asterisk in the "*Block*" column.

The evaluation of the errors detected in the single blocks (see Table III) continues by applying the strategy described in Subsection III-D to compute the achievable DC of each CNN layer according to the diagnostic techniques (the complete catalog can be identified in the y-axis of Fig. 5 and Fig. 6).

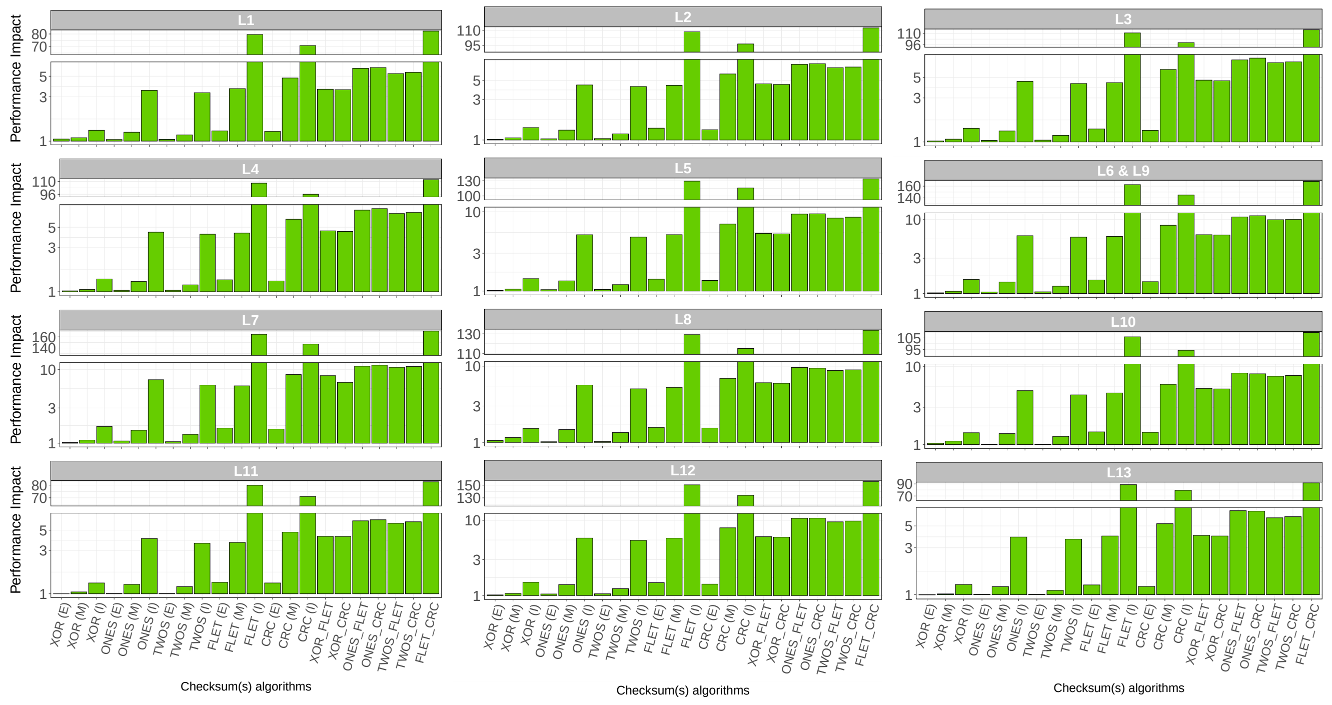


Fig. 6: Layer-by-layer performance impact with compiler optimization -O3

TABLE III. SINGLE GRID OF THREAD BLOCKS DIMENSIONS INVOLVED IN THE DC COMPUTATION OF EACH LAYER

Layer	Block	M	N	K	N_BR$_{T1}$	N_BC$_{T1}$
L1	B1	64	16	27	2704	1
L2	B1	64	32	144	676	1
L3	B1	64	64	288	169	1
L4	B1	64	64	576	42	1
	B3	16	64	576		
L5	B1	64	64	1152	10	4
	B3	36	64	1152		
L6 & L9	B1	64	64	2304	2	8
	B3	41	64	2304		
L7	B1	64	64	4068	2	16
	B3	41	64	4068		
L8	B1	64	64	1024	2	4
	B3	41	64	1024		
L10	B1	64	64	512	2	3
	B2	64	63	512		
	B3	41	64	512		
	B4	41	63	512		
L11	B1	64	64	512	2	2
	B3	41	64	256		
L12	B1	64	64	3456	10	4
	B3	36	64	3456		
L13	B1	64	64	256	10	3
	B2	64	63	256		
	B3	36	64	256		
	B4	36	63	256		

TABLE IV. SINGLE BLOCK DIMENSIONS EMPLOYED IN THE DC COMPUTATION OF EACH LAYER

Layers	Block	M	N	K	Injections$_A$	Injections$_B$
All except L1-2	B1	64	64	8	16384	16384
L1	B1	64	16	27	55296	13824
L2	B1	64	32	8	16384	8192
L10	B2*	64	63	8	16384	16128
L13	B2*	64	63	8	16384	16128
L6-L9 & L11	B3*	41	64	8	10496	16384
L10	B3*	41	64	8	10496	16384
L5 & L12	B3*	36	64	8	9216	16384
L13	B3*	36	64	8	9216	16384
L4	B3	16	64	8	4096	16384
L10	B4	41	63	8	10496	16128
L13	B4	36	63	8	9216	16128

We present in Table V the achievable DC of each layer. As a representative example, we evaluate the DC of $L10$, which includes by all types of blocks ($B1$, $B2$, $B3$ and $B4$), using XOR (E) diagnostic. First, we evaluate the errors detected according to the source of the error (matrix A or matrix B) as mentioned in Eq. 2 and Eq. 3. In this particular layer the complementary blocks do not detected additional errors. These are the expected results since the conditions stated in Subsection III-D2 are not satisfied (K_t is even in the external loops of the complementary blocks).

$$Det_A = (2048 + 0) \times 2 + 0 + 512 = 4608$$

$$Det_B = (2048 + 0) \times 3 + 0 + 2048 = 8192$$

We have reduced 64 times the K dimension in $L10$ matrix, i.e. $K = 512$ for all $L10$ grids of thread blocks and we used blocks with $K = 8$. Therefore, the errors detected by A and B must be multiplied by this number to obtain the final DC:

$$DC = \frac{(4608 + 8192) \times 64}{(169 + 255) \times 512} = 11.79$$

TABLE V. ACHIEVABLE DC LAYER-BY-LAYER ACCORDING TO THE DIAGNOSTIC TECHNIQUES CATALOG

Diagnostics	**Tiny Yolo-v3 layers**											
	L1	**L2**	**L3**	**L4**	**L5**	**L6&9**	**L7**	**L8**	**L10**	**L11**	**L12**	**L13**
XOR (E)	12,50	12,50	12,50	12,43	12,12	12,04	12,24	11,76	11,79	11,45	12,12	12,14
XOR (M)	99,99	99,93	99,41	95,48	72,53	24,82	14,17	39,76	39,86	56,90	72,53	72,61
XOR (I)	99,99	99,93	99,41	95,48	72,53	24,82	14,17	39,76	39,86	56,90	72,53	72,61
One's (E)	12,50	12,50	12,50	12,43	12,12	12,04	12,24	11,76	11,79	11,45	12,12	12,14
One's (M)	99,99	99,94	99,56	96,61	79,40	43,61	35,62	54,82	54,72	67,68	79,40	79,38
One's (I)	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00
Two's (E)	12,50	12,50	12,50	12,43	12,12	12,04	12,24	11,76	11,79	11,45	12,12	12,14
Two's (M)	99,99	99,94	99,55	96,54	78,97	42,44	34,28	53,88	53,79	67,00	78,97	78,95
Two's (I)	100,00	99,99	99,95	99,61	97,64	93,54	92,62	94,82	94,83	96,30	97,64	97,64
Fletcher (E)	12,50	12,50	12,50	12,43	12,12	12,04	12,24	11,76	11,79	11,45	12,12	12,14
Fletcher (M)	99,99	99,94	99,56	96,61	79,40	43,61	35,62	54,82	54,72	67,68	79,40	79,38
Fletcher (I)	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00
CRC (E)	12,50	12,50	12,50	12,43	12,12	12,04	12,24	11,76	11,79	11,45	12,12	12,14
CRC (M)	99,99	99,94	99,56	96,61	79,40	43,61	35,62	54,82	54,72	67,68	79,40	79,38
CRC (I)	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00
XOR_Fletcher	99,99	99,99	99,89	99,15	94,85	85,90	83,91	88,71	88,68	91,92	94,85	94,84
One's_Fletcher	99,99	99,99	99,94	99,58	97,42	92,95	91,95	94,35	94,34	95,96	97,42	97,42
Two's_Fletcher	99,99	99,99	99,94	99,58	97,42	92,95	91,95	94,35	94,34	95,96	97,42	97,42

TABLE VI. TRADE-OFF OF PERFORMANCE IMPACT VS DC

Layer	**DC ranges (%)**		
	$99 \leq$ DC	**$90 \leq$ DC < 99**	**$60 <$ DC < 90**
L1	XOR (M)	XOR (M)	XOR (M)
L2	XOR (M)	XOR (M)	XOR (M)
L3	XOR (M)	XOR (M)	XOR (M)
L4	One's (I)	XOR (M)	XOR (M)
L5	One's (I)	Two's (I)	XOR (M)
L6	One's (I)	Two's (I)	Two's (I)
L7	One's (I)	Two's (I)	Two's (I)
L8	One's (I)	Two's (I)	Two's (M)
L9	One's (I)	Two's (I)	Two's (I)
L10	One's (I)	Two's (I)	Two's (I)
L11	One's (I)	Two's (I)	Two's (M)
L12	One's (I)	Two's (I)	XOR (M)
L13	One's (I)	Two's (I)	XOR (M)
PI	3,80	3,33	2,61

The complete set of results is shown in Table V. From these results, we observe the significant influence of the matrices dimensions in the achievable DC. The higher the ratio $\frac{M}{N}$, the higher the achievable DC is. This can be appreciated by comparing $L1$ and $L7$, whose respective ratios are $\frac{64}{16} = 4$ for $L1$, and $\frac{64}{64} = 1$ and $\frac{41}{64} = 0,64$ for $L7$, and whose DC in XOR (M) decreases from 99,99% to 14,17%, respectively.

D. Stage 3: Selective protection

In this section, we perform a selective layer-by-layer protection of Tiny YOLO-v3. Instead of selecting the same diagnostic for all layers, we select the diagnostic with the lowest performance impact in each layer for each of the DC ranges established by IEC 61508. Those diagnostics are shown in Table VI. Additionally, we include the lowest performance impact ratio (PI) incurred in each range to protect with the combination of these diagnostics.

We observe that the lowest performance impact on achieving high, medium, and low DC ranges is 3,8, 3,33, and 2,61, respectively. Note that, while such performance impact is high, it could be reduced if diagnostics are just executed once periodically as described in [3]. That work shows that the safety architectural pattern where hardware is diagnosed periodically with predefined data so that output is known can be tuned as needed to trade off between performance impact and diagnostics frequency. For instance, if the process safety time (PST) is one hundred times the execution of a single classification task and the individual performance impact for the high DC range is 3,8, as shown above, the periodic diagnosis can be executed once in each PST period incurring in a performance impact of just 5%. In other words, we could execute the diagnosis once every 76 CNN executions.

In this paper, we stick to a simple approach based on selecting for each layer the diagnostic with lowest performance cost that achieves the target DC individually for that layer. However, this approach may not be an option in performance terms even for those systems based on the periodic diagnostic pattern if the PST is not big enough. Then, the safety designer has to selectively protect each layer based on its propensity to misclassifying and the percentage of the execution time of each one according to the complete CNN. Highlight that this approach is subject to performing fault injection campaigns to verify whether the proposed selection of diagnostics achieves the application's target DC range.

V. Related Work

To the best of our knowledge, this is the first work performing an exhaustive layer-by-layer analysis of the achievable DC with a variety of diagnostics and presenting a strategy to effectively compute the achievable DC of large matrices. Several works focus on analyzing the DNN and CNN reliability. Some of them focus on the identification of DNN reliability challenges, and summarize analytic and mitigation techniques [22], [23]. In our case, we additionally apply our method in a particular CNN application based on object detection. Other research is based on studying the CNN reliability [14]–[16], [22], [24]–[26], yet without providing systematic solutions such as the one in our work, where we provide a way to select per-layer diagnostics.

Other papers focus on protecting the MMM, aiming at a safe CNN implementation. We identify three technical approaches: those based on full or partial redundancy at the software or hardware level of the CNN [16], [24], those based on adopting algorithm-based fault detection or fault tolerance [27]–[29], and the recent approach lying on the adoption of a catalog of widely used diagnostics techniques to compute an array of ES [3]. We build on the latter, which is complementary to the others and provides the system with fault detection capabilities to detect random errors (permanent and transient). In our contribution, we perform a complete performance impact and achievable DC analysis, building on an efficient block decomposition for fault injection, of a CNN based on MMM, layer by layer, complementing [3].

VI. Conclusions and Future Work

This paper provides a three-stage methodology to selectively protect with the required DC CNNs implemented on GPUs focusing on its most expensive computation part, the MMM. We apply this methodology to the Tiny YOLO-v3 object detector as an application example. The first stage consists of the sensitivity analysis of each CNN's layer to identify the most misclassification-prone layers. For this CNN, we observe a higher tendency to misclassify (from 83,4 to 99,6%) in the initial layers ($L1$-$L8$). However, the final layers also present lower but still high misclassification rates (from 55,2 to 74,34%). In the second stage, we analyze the achievable DC and the incurred performance impact per layer for the catalog of diagnostics provided in [3]. For the DC analysis, we offer a strategy that computes the entire MMM DC based on analyzing the blocks in which MMM is decomposed before launching it to the GPU with an exhaustive fault injection campaign at the bit-level of these smaller blocks. Finally, we selectively protect each layer according to the three DC ranges providing the most appropriate diagnostics that achieve the minimum required DC in each range on an NVIDIA Xavier Nx GPU. For the given example, we observe that the lowest performance impact to achieve high, medium, and low DC ranges is 3,8, 3,33, and 2,61, respectively. As explained, this impact might be affordable in the context of the safety architectural pattern where diagnostics are performed periodically, in accordance with the timing constraint imposed by the PST, by trading-off between diagnostics frequency and performance impact.

In future work, we propose to analyze the behavior of greater CNNs, such as YOLO-v3, to identify whether larger CNNs reduce the propensity to misclassification in some layers. Additionally, for those applications demanding a higher performance impact, we propose the implementation of the MMM and the diagnostics techniques catalog into the tensor cores included in GPUs, which accelerate MMMs.

Acknowledgement

Ikerlan authors have received funding from Elkartek grant project KK-2021/00123 of the Basque government. BSC authors have been partially supported by the Spanish Ministry of Science and Innovation under grant PID2019-107255GB-C21/AEI/10.13039/501100011033

References

[1] L. Jiao *et al.*, "A Survey of Deep Learning-Based Object Detection," *IEEE Access*, vol. 7, pp. 128 837–128 868, 2019.
[2] J. Perez-Cerrolaza *et al.*, "GPU devices for safety-critical systems: A survey," *ACM Comput. Surv.*, 2022.
[3] J. Fernández *et al.*, "On the safe deployment of matrix multiplication in massively parallel safety-related systems," *Applied Sciences*, vol. 12, no. 8, 2022.
[4] "IEC 61508(-1/7): Functional safety of electrical / electronic / programmable electronic safety-related systems," 2010.
[5] H. Tabani *et al.*, "A Cross-Layer Review of Deep Learning Frameworks to Ease Their Optimization and Reuse," in *ISORC*, 2020, pp. 144–145.
[6] J. Perez Cerrolaza *et al.*, "Multi-core Devices for Safety-critical Systems: A Survey," *ACM Comput. Surv.*, vol. 53, no. 4, 2020.
[7] J. Redmon, "Darknet: Open source neural networks in C," 2013–2016. [Online]. Available: http://pjreddie.com/darknet/
[8] P. Adarsh *et al.*, "YOLO v3-Tiny: Object detection and recognition using one stage improved model," in *ICACCS*, 2020, pp. 687–694.
[9] J. Redmon and A. Farhadi, "YOLOv3: An incremental improvement," 2018.
[10] A. Bochkovskiy *et al.*, "YOLO v4: Optimal speed and accuracy of object detection," *arXiv preprint arXiv:2004.10934*, 2020.
[11] NVIDIA, "CUTLASS: CUDA Templates for Linear Algebra Subroutines," https://github.com/NVIDIA/cutlass, 2020, [Online; Dec-2021].
[12] T. C. Maxino and P. J. Koopman, "The Effectiveness of Checksums for Embedded Control Networks," *IEEE TDSC*, vol. 6, pp. 59–72, 2009.
[13] P. Koopman *et al.*, "Selection of cyclic redundancy code and checksum algorithms to ensure critical data integrity," Carnegie Mellon University, Report, 2015.
[14] A. Bosio *et al.*, "A Reliability Analysis of a Deep Neural Network," in *IEEE LLATS*, 2019, Conference Proceedings, pp. 1–6.
[15] A. Ruospo *et al.*, "Evaluating Convolutional Neural Networks Reliability depending on their Data Representation," in *DSD*, 2020, pp. 672–679.
[16] F. dos Santos *et al.*, "Kernel and Layer Vulnerability Factor to Evaluate Object Detection Reliability in GPUs," *IET Computers & Digital Techniques*, vol. 13, 2018.
[17] R. Leveugle *et al.*, "Statistical fault injection: Quantified error and confidence," in *DATE*, 2009, Conference Proceedings, pp. 502–506.
[18] J. Fernández *et al.*, "Towards Safety Compliance of Matrix-Matrix Multiplication for Machine Learning-based Autonomous Systems," *JSA*, 2021.
[19] I. S. Olmedo *et al.*, "Dissecting the CUDA scheduling hierarchy: a Performance and Predictability Perspective," in *RTAS*, 2020.
[20] F. Yu *et al.*, "BDD100K: A Diverse Driving Dataset for Heterogeneous Multitask Learning," in *The IEEE CVPR*, June 2020.
[21] A. J. Calderón *et al.*, "GMAI: Understanding and Exploiting the Internals of GPU Resource Allocation in Critical Systems," *ACM Trans. Embed. Comput. Syst.*, vol. 19, no. 5, p. Article 34, 2020.
[22] M. A. Hanif *et al.*, "Robust Machine Learning Systems: Reliability and Security for Deep Neural Networks," in *IEEE IOLTS*, 2018, Conference Proceedings, pp. 257–260.
[23] M. Hanif and M. Shafique, "Dependable Deep Learning: Towards Cost-Efficient Resilience of Deep Neural Network Accelerators against Soft Errors and Permanent Faults," in *IEEE IOLTS*, 2020, Book, pp. 1–4.
[24] Y. Ibrahim *et al.*, "Soft error resilience of deep residual networks for object recognition," *IEEE Access*, vol. 8, pp. 19 490–19 503, 2020.
[25] A. Mahmoud *et al.*, "Optimizing Selective Protection for CNN Resilience," in *IEEE 32nd ISSRE*. IEEE Computer Society, 2021, Conference Proceedings, pp. 127–138.
[26] M. A. Neggaz *et al.*, "Are CNNs Reliable Enough for Critical Applications? An Exploratory Study," *IEEE Design & Test*, vol. 37, no. 2, pp. 76–83, 2020.
[27] K. Zhao *et al.*, "FT-CNN: Algorithm-Based Fault Tolerance for Convolutional Neural Networks," *IEEE TPDS*, vol. 32, pp. 1677–1689, 2020.
[28] S. K. S. Hari *et al.*, "Making Convolutions Resilient via Algorithm-Based Error Detection Techniques," *IEEE TDSC*, pp. 1–1, 2021.
[29] J. Kosaian and K. V. Rashmi, "Arithmetic-intensity-guided fault tolerance for neural network inference on GPUs," p. Article 79, 2021.