Towards Functional Safety Compliance of Matrix-Matrix Multiplication for Machine Learning-based Autonomous Systems

Javier Fernandez[a,b], Jon Perez[a], Irune Agirre[a], Imanol Allende[a], Jaume Abella[c], Francisco J. Cazorla[c]

[a] *Ikerlan Technological Research Center, Basque Research and Technology Alliance (BRTA), Mondragon, Spain*
[b] *Universitat Politècnica de Catalunya (UPC), Barcelona, Spain*
[c] *Barcelona Supercomputing Center (BSC), Barcelona, Spain*

Abstract

Autonomous systems execute complex tasks to perceive the environment and take self-aware decisions with limited human interaction. This autonomy is commonly achieved with the support of machine learning algorithms. The nature of these algorithms, that need to process large data volumes, poses high-performance demands on the underlying hardware. As a result, the embedded critical real-time domain is adopting increasingly powerful processors that combine multi-core processors with accelerators such as GPUs. The resulting hardware and software complexity makes it difficult to demonstrate that the system will run safely and reliably. This is the main objective of functional safety standards, such as IEC 61508 or ISO 26262, that deal with the avoidance, detection and control of hardware or software errors. In this paper, we adopt those measures for the safe inference of machine learning libraries on multi-core devices, two topics that are not explicitly covered in the current version of standards. To this end, we adapt the matrix-matrix multiplication function, a central element of existing machine learning libraries, according to the recommendations of functional safety standards. The paper makes the following contributions: i) adoption of recommended programming practices for the avoidance of programming errors in the matrix-matrix multiplication, ii) inclusion of diagnostic mechanisms based on widely used checksums to control runtime errors, and iii) evaluation of the impact of previous measures in terms of performance and a quantification of the achieved

November 12, 2021

diagnostic coverage. For this purpose, we implement the diagnostic mechanisms on one of the ARM R5 cores of a Zynq UltraScale+ multi-processor system-on-chip and we then adapt them to an Intel i7 processor with native code employing vectorization for the sake of performance.

Key words: Machine learning, functional safety, error detection

1. Introduction

The use of machine learning (ML) is increasing in many safety critical domains for the deployment of complex autonomous functionalities, such as, advanced visual perception. For example, deep neural networks relying on convolutional neural networks (CNNs) have shown to provide high accuracy and adaptability in camera-based object detection [1]. As a result, there is an emerging number of software libraries deployed for object detection [2], such as, you only look once (YOLO) [3].

These ML algorithms need to process large volumes of data in real-time (e.g., images), and this requires high-performance embedded computing (HPEC) platforms with computing capabilities far beyond those of traditional safety systems, such as multi-core devices and GPU accelerators.

ML solutions are increasingly being adopted in safety-related tasks that are subject to safety certification. This certification is achieved by proving adherence to applicable functional safety standards with respect to the application domain, such as, IEC 61508 [4] or ISO 26262 [5]. Nevertheless, neither emerging ML algorithms nor the HPEC platforms are commonly captured in current versions of standards and their certification is an open research challenge. The lack of determinism –a crucial property for safety-critical systems– is a clear example of an open issue, caused by the inherent statistical nature of ML and the high degree of parallelism of their inference platforms.

Only the recent ISO/PAS 21448 automotive standard [6] (referred as safety of the intended functionality (SOTIF)) explicitly addresses some of the implications of ML algorithms, focusing mainly on the offline training method. SOTIF sets its focus on system safety, in this case at a vehicle level. It deals with potential hazards caused by limitations of the system to operate autonomously in an open environment where situational awareness is required (e.g., system malfunction due to limitations in the ML training process in the absence of hardware and software errors). This is complemen-

tary to functional safety standards, such as IEC 61508 and ISO 26262, that deal with hardware and software errors in the safety-critical control units of the system. The scope of this paper relates to the latter, by evaluating and proposing solutions for the mitigation of systematic and random errors in ML algorithms-based software libraries executed on HPEC platforms.

More specifically, this paper builds on the open source YOLO object detector algorithm. We have selected it among the huge variety of current object detection algorithms [2] not only for its high accuracy on perceptual tasks [7], but also because its reliability has been widely studied [7, 8, 9, 10]. Our contribution lies in the definition of a "safe matrix-matrix multiplication (MMM)" software module . This arithmetic operation is the backbone of many ML libraries [11], as it takes 67% of the CNN execution time according to the results obtained from YOLO v3 implemented on NVIDIA GPUs [8]. An error in the execution of the MMM could lead to miss-classification in the object detection tasks. As an illustrative example, the authors in [12, 13] perform a fault-injection campaign to demonstrate how a single-bit error could result in a wrong classification of a horse as a sheep, or a truck as a bird respectively. Their results reveal that 25% of the faults forced in a specific case of study employing the YOLO object detector lead to an unsafe situation [12].

YOLO, as other object detectors such as Lenet [14], is based on the Darknet CNN [15]. This open source CNN offers the user a selection of MMM algorithms based on implementation needs: i) general matrix multiplication (GEMM) function for the sequential implementation on CPUs, ii) CUBLAS for the implementation in GPUs with CUDA and iii) other processor-specific variants such as the use of AVX with specific instructions for Intel processors. This paper employs as baseline the sequential GEMM implementation and the AVX-based MMM implementation. The former is interesting from a safety perspective, as it is the closest option to current safety practices. However, at the same time, the sequential implementation provides the poorest performance, which is also an important property for ML inference. Trying to find a balance between safety and performance, in the first place we advocate for the AVX-based MMM instead of the CUDA-based implementation. The main reason for this is that the closed-source nature of CUBLAS library, none or limited support for the development of safety-critical software by means of available GPU programming languages [16] and GPU's features such as dynamic memory allocation, imply bigger challenges to comply with functional safety standards [17]. The paper con-

tributes with the following modifications to these two YOLO MMM baselines, referenced as sequential MMM and AVX-based MMM:

1. We define and implement the required modifications for the avoidance of systematic errors in the implementation of the MMM function with the help of Polyspace [18], a static analysis tool. Likewise, we evaluate the complete CNN used by YOLO (i.e., Darknet CNN) to demonstrate that complying with the specifications demanded by coding guidelines such as motor industry software reliability association (MISRA) C is feasible with limited effort.
2. We identify and integrate a variety of suitable diagnostic mechanisms to attain different levels of diagnostic coverage (DC) against random errors in HPEC platforms and residual systematic errors in the design. For that, we integrate existing checksums in the ARM R5 architecture and we adapt them to the Intel AVX instructions. In addition, we define two common safety architectural patterns where the previously mentioned diagnostics could be used to implement error detection.
3. We assess the DC and the performance penalty incurred by previous measures for a given set of representative matrix dimensions. For that, we perform exhaustive single-bit error injection and, as a result, we provide a catalogue of mechanisms with varying levels of DC and performance impact, allowing the final user to choose a solution based on system needs.

The rest of this paper is organized as follows: Section 2 outlines the basic concepts. Sections 3 and 4 define the proposed adaptations to obtain a "safe MMM" software module with configurable DC levels. Specifically, the focal point of Section 3 is the avoidance of systematic errors, while Section 4 targets the detection of errors. Section 5 presents the results of the evaluation of the proposed modifications and Section 6 analyses the related work. Finally, Section 7 draws the conclusions and and outlines future work.

2. Background

This section provides the necessary background in order to ease the understanding of basic concepts throughout this paper.

2.1. Functional Safety Standards

IEC 61508 [4] is an international functional safety standard that guides the development process of safety-related systems composed of electrical, electronic, or programmable electronic elements across different industry sectors. IEC 61508 is the reference standard for the definition of many domain specific standards, such as ISO 26262 [5] for road-vehicles or EN 5012X [19] for railway. These standards define the necessary requirements, techniques and measures to guarantee the absence of unacceptable risks caused by the malfunction of the system. To this end, the IEC 61508 standard defines the safety integrity level (SIL) metric for each safety function according to its criticality from SIL 1 (minimun) to SIL 4 (maximun). Similarly, ISO 26262 uses the term automotive safety integrity level (ASIL), which ranges from ASIL A (least stringent) to ASIL D (most stringent). According to these integrity levels, functional safety standards require the adoption of different safety measures and mechanisms in the development-cycle and in the design. In the case of IEC 61508, the importance of applying a specific technique or measure for each integrity level is signified by the following notation: i) mandatory (M), ii) highly-recommend (HR), iii) recommended (R) and iv) non-recommended (NR).

2.2. Types of faults and diagnostic coverage

Faults are classified into two major categories in the aforementioned safety standards: *systematic* and *random* faults. Systematic faults are associated with the development process and method, and may relate to hardware and/or software. Safety standards impose a development process intended to make the residual risk of systematic faults negligible. Instead, random faults relate to hardware faults caused by electromagnetic interference, voltage drops, component wear-out and the like. Additionally, random faults can be classified according to the frequency into *permanent*, if they persist indefinitely, and *transient*, if the occurrence is sporadic [5]. Hence, safety standards recommend the deployment of safety measures to detect and control those faults in such a way that they do not lead to failure.

The assessment of the effectiveness of diagnostic mechanisms is generally evaluated in the form of DC. As defined in [20], *"Diagnostic Coverage (DC) denotes the effectiveness of diagnosis techniques to detect dangerous errors, expressed in coverage percentage with respect to all possible dangerous errors"*. DC is classified as low ($60\% < DC < 90\%$), medium ($90\% \leq DC < 99\%$)

and high ($99\% \leq DC$) [4]. As stated in [20], the implementation of software-based DC techniques becomes relevant in order to periodically diagnose the correct operation of the hardware components or the safe operation of the device with respect to possible faults not covered by hardware diagnosis or to complement them (usually classified as low or medium DC).

To achieve appropriate error detection and tolerance levels, safety measures are often deployed following specific *architectural patterns*. There is a large variety of patterns, and most of them are safety measure dependent, but some of the most common ones build upon the use of *redundancy* (e.g. full or partial time or space replication) and *diversity* (e.g., making redundancy non-identical so that a single error does not lead to the same erroneous output in all redundant instances).

IEC 61508 defines the abbreviation Noo$M$ (N out of M) to describe the architecture of the system: M is the total number of channels in the architecture (where channel refers to the group of elements that implement a safety function) and, N is the minimum number of channels that are required to complete the safety function [4]. As an example, a *1oo2* architecture consists of two channels connected in parallel (M= 2) and either channel can process the safety function by its own (N= 1). Therefore, in case of a dangerous failure in one of the channels, the second one can still safely perform the safety function.

2.3. ML and YOLO

ML algorithms are primarily based on statistical learning. A task is performed based on a probabilistic model generated from training data instead of derived from its specifications. This technique enables the implementation of functionalities that are harder to be programmed by traditional software due to the impossibility of manually formulating rules for generating output from the large spectrum of possible inputs.

The selection of the most appropriate ML algorithm depends on the application domain. In domains such as camera-based object detection or image classification, CNNs are widely used. For instance, YOLO is a multi-scale object detector traditionally based on Darknet, a CNN coded in C and CUDA. There are several versions of this object detector [3, 21, 22] whose operation is mainly based on three stages: i) the convolutional layers extract the features from the input image, ii) max-pooling layers reduce the feature map and iii) fully connected layers classify the input.

The development of convolutional operations entails the execution of MMMs. As mentioned in the introduction, the MMM accounts for most of the execution time of YOLO in particular, and prediction and perception processes in general [11]. Darknet provides several options for the implementation of this algebraic operation depending on the target platform where it is to be deployed [15]. In the following two sections, we present the proposed method and strategy to adapt the sequential and AVX-based MMM implementations to "safe MMMs".

3. Systematic error avoidance in the MMM

In this section, we focus on the avoidance of software systematic errors. To that end, we have verified the source code of the sequential MMM, and implemented the resulting verification comments, according to the following recommendations:

- Usage of a safe subset of the "C language" according to the MISRA C coding guideline [23]. According to ISO 26262-6 Table 1 the use of a language subset (where unsafe language features are excluded) is a *HR* technique for any ASIL [5] and for SIL 3 or higher according to IEC 61508-3 Table A.3 [4]. More explicitly, IEC 61508 states in 7.4.4.12 that *"programming languages for the development of all safety-related software shall be used according to a suitable programming language coding standard"*.

- Use of defensive programming where input parameters are checked with respect to coherence and correctness. This is also a *HR* technique in same tables of both ISO 26262 and IEC 61508 for ASIL D / SIL 3 or higher. Concretely, the standards recommend their use to check data or control anomalies at runtime. In particular, we have checked that pointers to matrices do not have "NULL" values.

The adherence to MISRA C can be checked manually by the developer or with the help of static tools such as Polyspace [18]. In our case, we have used this tool for identifying 33 violations in the sequential MMM as shown in Fig. 1. We have seen that the corrections required by the MISRA C directives (identified with the letter "D") and rules need limited engineering effort as described below:

- D4.6: Twenty two violations were due to not explicitly defining types with the size and signedness for basic numerical types. In this case, required types were explicitly defined.

- D4.14: Six violations were caused by not checking the correctness of input parameters. As a corrective action, "defensive programming" has been implemented, as explained before.

- 12.1: Three violations were due to not explicitly defining the desired precedence of operators within expressions. In this case, adherence has been achieved with the explicit definition of precedence.

- 8.13: Two violations were caused by not explicitly defining input parameter pointers as const-qualified type. As a corrective action, all input parameter pointers to data content that are not internally modified have been explicitly qualified as constant.

Figure 1: Sequential MMM software MISRA-C compliance analysis result: number of rules and directives (D) violated in the sequential MMM code according to an analysis performed by the Polyspace analysis tool.

Additionally, we have analyzed the complete Darknet CNN with the Polyspace tool identifying 2,332 violations. In the same manner, these violations do not require a significant effort to accomplish MISRA C. However, the application of the respective modifications is out of the scope of this paper. For illustrative purposes, in Fig. 2 we depict the 5 most repeated violations, which suppose the 64,5 % of the total. The adherence to directive D4.6 and rule 12.1 can be achieved with previous corrective actions and the additional ones described below:

- D1.1: These violations are related to explicitly defining the implementation-defined behaviour that affects the outputs, such as casting from integer

to floating-point numbers. As MISRA C suggests, a compliance matrix with the procedures to follow by the developer should be produced to ensure that the code complies with all MISRA C violations. This action requires a manual review.

- 15.6: These violations concern the use of a compound statement to enclose the body of an iteration or selection-statement. The corrective action lies on the inclusion of this compound statement to clarify which statements form the body.

- 10.3: The assignment of a value from an expression to an object with a narrower essential type shall not be made. The easier corrective action is to cast to the same essential type.

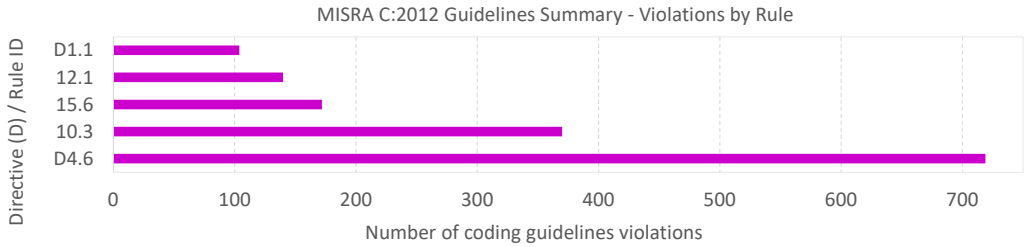

Figure 2: Darknet CNN MISRA-C compliance analysis result: top 5 of rules and directives (D) violated in the Darknet CNN code according to an analysis performed by Polyspace.

4. Error detection in the MMM

Providing a safe inference execution environment for the YOLO MMM software module requires the deployment of diagnostic mechanisms for run-time error detection [24]. Moreover, the execution on a HPEC multi-core device depends on the safe behaviour of multiple components (e.g., core, private cache, interconnect, shared cache, shared memory, memory management unit) and built-in mechanisms (e.g., cache coherency) required for the execution of the given software [20]. Additionally, embedded platforms are scaling down reducing their dimension resulting in an increase in the component density. Hence, the soft error rate drastically increases in these silicon devices, becoming a challenge with a dire need to overcome in safety-critical systems [20, 25].

The implementation of DC techniques could be considered outside the scope of YOLO MMM, exporting such requirements to the system architect and integrator. However, the implementation of simple generic DC techniques built-in in the MMM software module can provide an efficient solution to achieve the required DC with a reduced effort for the system integrator [20, 26]. For example, detecting errors due to cache coherency, cache errors, interconnect errors and core execution that could lead to incorrect computation errors can be a challenge in HPEC multi-core devices as explained in [20] and also from a system integration perspective [20, 26].

In this section, we propose the use of an execution signature (ES) computed by diagnostic techniques, which can provide a scalable strategy with a trade-off between the desired potential DC level and the associated computational cost. The ES, described in Section 4.1, can summarize in a reduced number of bits (e.g., 32 bits), the computed data and program sequence for later comparison in the architectural patterns described in Section 4.2.

4.1. Execution signature

Among the state of the art, we have identified XOR, 2's and 1's complement addition, Fletcher and cyclic redundancy check (CRC) as potential diagnostic mechanisms to compute the ESs. These error detection mechanisms are widely used to assure the network message data integrity with different error detection effectiveness [27]. For details about each of these techniques, we refer the reader to [27] and [28], which make an evaluation of the error detection properties of each of the checksum algorithms, proposing methods for the selection of the most appropriate one (based on parameters such as the length of the code, the kind of errors or the selection of a polynomial generator used in the CRC algorithm [29]). The novelty of this paper rests in the integration of these existing diagnostics into the MMM and the evaluation of both, the performance impact and achievable DC based on the data to be protected and the checksum or combination of checksums employed. This approach allows verifying not only the bit-wise correctness of the data involved in the MMM execution, but also a proper program sequence with a reduced number of bits (32 in our case).

The algebraic MMM operation is usually coded and implemented through nested loops as shown in Algorithm 1. The ES can be calculated integrating the checksum algorithm(s) in any of the loops, or a combination of loops and checksum algorithms. The methodology that we employ in this paper can be extrapolated to MMM involving a different number of loops. In our case,

both the sequential MMM and AVX-based MMM implementations use three loops for computing the MMM, denoted as inner (I), intermediate (M) and external (E) loops as shown in Algorithm 1.

Algorithm 1 MMM loops

```
1: for each column of the first matrix do
2:     External loop statements
3:     for each row of the first matrix do
4:         Intermediate loop statements (Store the value of the first matrix in a register)
5:         for Each column of the second matrix do
6:             Internal loop statements (Compute the multiplication)
7:             [Checksum (I)]
8:         end for
9:         [Checksum (M)]
10:    end for
11:    [Checksum (E)]
12: end for
```

The deeper the loop where the checksum is implemented, the higher is the potential achievable DC because higher is the amount of data and computation summarized in the ES, at the cost of increasing the required computational cost to generate the ES. Therefore, the potentially achievable DC level depends on the length of the ES and the data protected, the selected checksum algorithm and the loop level where it is implemented. These statements are independent of the number of loops employed for the implementation of the MMM.

The implemented checksums compute the ES of A, B and C matrices (where the duty of the MMM is to compute $C = A \cdot B$) and store these ESs values in independent variables. Once the multiplication is complete, the selected checksum is again employed to combine all signatures into a single one. However, the inclusion of certain checksums, such as Fletcher and CRC in the internal loop, can be an unaffordable solution in terms of performance. Fortunately, it is expected that combining these algorithms in the outermost loops with checksums with a lower performance penalty in the inner loop will provide a reduction in overhead and an increase in DC over individual implementations. For that reason, we have designed a catalogue with a combination of checksums to provide the user with a wide variety of DC and performance penalty alternatives. The catalogue can be divided into two groups according to the checksum involved in the multiplication:

- Individual: use a single checksum algorithm in one of the three loops (I, M, E) of Algorithm 1.
- Combinations: use different checksum algorithms with lower performance impact in the internal loop and higher performance impact, and higher DC, in the intermediate loop (e.g., XOR_Fletcher means that a XOR is computed in the internal loop and a Fletcher in the intermediate loop).

In favour of understandability of the checksum combinations, Algorithm 2 shows the code employed to compute the ES with a XOR_Fletcher checksum combination in the MMM. We can see how in lines 3-5 we have applied defensive programming to check that the pointers to the arrays do not have "NULL" values. In lines 14, 15 and 17 we code the XOR checksum to compute the ES of the matrices"B", "C" and "A" respectively. All these values are summarized into a single ES in line 18 with the XOR checksum. Finally, the Fletcher checksum is coded in line 19 to perform a new ES from the ES computed by the XOR checksum.

Additionally, in Algorithm 3 we show the code of the Fletcher checksum. As it can be seen, the Fletcher function receives the union datatype *ui32_to_ui16_t* with the current Fletcher ES and an *uint32_t* datatype with the data to be protected. The union has been employed to access the same memory position with two datatypes: i) *uint32_t* and ii) an array of two *uint16_t* values. The use of the union has been fostered by the nature of the Fletcher checksum, which involves decomposition into two smaller blocks to carry out the ES computation.

Algorithm 3 Fletcher Checksum

```
 1: function FLETCHER32C_UI32(ui32_to_ui16_t Fletcher, uint32_t data)
 2:     ui32_to_ui16_t v;
 3:     v.ui32 = ui32_data;
 4:     Fletcher.ui16[0u] += v.ui16[0u];
 5:     Fletcher.ui16[1u] += Fletcher.ui16[0u];
 6:     Fletcher.ui16[0u] += v.ui16[1u];
 7:     Fletcher.ui16[1u] += Fletcher.ui16[0u];
 8:     Fletcher.ui16[0u] %= 255u;
 9:     Fletcher.ui16[1u] %= 255u;
10:     return Fletcher.ui32;
11: end function
```

Algorithm 2 Sequential MMM with XOR_Fletcher checksums implemented

```
 1: function SMM_XOR_INTERMEDIATE(uint32_t ui32_m, uint32_t ui32_n,
          uint32_t ui32_k, float32_t f32_alpha, const float32_t* const paf32_ma,
          const float32_t* const paf32_mb, const float32_t* const paf32_mc)
 2:     //Definition of local variables
 3:     assert(paf32_ma != NULL);                          ▷ Defensive programming
 4:     assert(paf32_mb != NULL);
 5:     assert(paf32_mc != NULL);
 6:     for (ui32_idx_i = 0u; ui32_idx_i < ui32_m; ui32_idx_i++) do
 7:         ui32_idx_b_ref = 0u;
 8:         for (ui32_idx_k = 0u; ui32_idx_k < ui32_k; ui32_idx_k++,
             ui32_idx_a++)  do
 9:             f32_a_part = f32_alpha * paf32_ma[ui32_idx_a];
10:             for (ui32_idx_j = 0u, ui32_idx_b = ui32_idx_b_ref,
            ui32_idx_c = ui32_idx_c_ref; ui32_idx_j < ui32_n; ui32_idx_j++,
            ui32_idx_b++, ui32_idx_c++) do
11:                 f32_b = paf32_mb[ui32_idx_b];                  ▷ Multiplication
12:                 paf32_mc[ui32_idx_c] += f32_a_part * f32_b;
13:                 f32_c = paf32_mc[ui32_idx_c];
14:                 ui32_xor_b  ⊕= (uint32_t) * ((uint32_t *) &f32_b);        ▷ XOR ES
15:                 ui32_xor_c  ⊕= (uint32_t) * ((uint32_t *) &f32_c);
16:             end for
17:             ui32_xor_a ⊕= (uint32_t) * ((uint32_t *) &f32_a_part);
18:             ui32_xor = (ui32_xor_a ⊕ui32_xor_b) ⊕ ui32_xor_c;
19:             Fletcher.ui32 = Fletcher32c_ui32(Fletcher, ui32_xor);      ▷ Fletcher ES
20:             ui32_idx_b_ref += ui32_n;
21:         end for
22:         ui32_idx_c_ref += ui32_n;
23:     end for
24:     return Fletcher.ui32;
25: end function
```

However, the computation of the ES by itself is not sufficient for detecting errors at runtime. The use of safety architectural patterns, explained in next subsecion, becomes an inherent part of their implementation as diagnostic mechanisms.

4.2. Architectural Patterns

Taking into consideration the architectural patterns and DC techniques for HPEC multi-core devices described in [20], and the safety measures proposed by the safety standards (IEC 61508, ISO 26262) considered in this work, this section defines two basic and common architectural patterns that support safe detection of faults based on previous diagnostic techniques.

Periodic diagnosis with design time fixed data pattern(s). In this pattern, in addition to the standard MMM periodic execution (τ), the "safe MMM" software variant is executed with a period L times lower (τ/L). This period is associated to the process safety time (PST) (IEC 61508) or diagnostic time interval (DTI) (ISO 26262) of the system. During this "safe MMM" execution, design time fixed data inputs are used, for which a design time ES is known. This pattern enables the periodic diagnosis of device components and built-in mechanisms. If the obtained ES does not match the expected design time ES value, a random error may have occurred and the repetition of this error can determine whether it is transient or permanent.

Redundancy (with or without diversity). The "safe MMM" software, or a complete safe YOLO library that integrates the "safe MMM", is executed with redundancy by M replicas (e.g., 1oo2, 2oo3) and for each redundant execution, both an output and ES values are generated. After that, the voting mechanism compares ES values (and optionally the output) and the replica(s) with discrepancies are discarded. The comparison of just the output values would not be generally sufficient to detect latent errors (e.g., faults in matrix B –activation weights matrix– cannot be detected in the output matrix C for a given set of A matrices if those matrices (A) take zero values in the positions computed with the faulty position of matrix B). This can provide a "correct" output while it masks a latent error that would be detected on the ES. This pattern implies a higher computational cost than previous as the "safe MMM" is executed in each execution period (τ) by the M replicas. However, the correctness of components and of the output is diagnosed in every execution period and it can support fault-tolerance as later explained in Section 5.5 (e.g., a 2oo3 architecture could tolerate one discrepancy). In order to further improve diagnostic coverage by the detection of common cause failures (CCFs), the redundant pattern can be complemented with different types of *diversity* [20], such as component diversity (e.g., implementing the MMM in different types of cores of a multi-core platform).

5. Evaluation

In this section, we evaluate the execution time penalty caused by the inclusion of the different ESs from Section 4 in the original MMM software module, as well as the maximum achievable DC of each ES for the detection

of single-bit errors. For that, we consider the sequential MMM and AVX-based MMM extracted from YOLO. In addition, we provide a discussion of compliance of the contributions of this paper to functional safety standards.

5.1. Experimental set-up

We have implemented the sequential MMM function on one of the ARM R5 lock-step cores of a Zynq UltraScale+ multi-processor system-on-chip device as a representative safety device that is certified up to SIL 3 according to IEC 61508 and up to ASIL C regarding to ISO 26262 [20]. While there are more effective approaches for ML inference in terms of performance, like the use of accelerators such as GPUs, they pose additional challenges for safety certification [20]. Therefore, it is necessary to find a balance between performance and safety. For this reason, we have first focused on a more conservative multi-core solution based on R5 cores designed for functional safety. However, the overall approach is platform independent and can be adapted to diverse platforms. For instance, with the aim of improving overall performance, we then implement and evaluate an AVX-based solution, which employs vectorization on an Intel i7 processor. In the future, the same approach could be used for other implementations.

The sequential MMM has been compiled employing the ARM v8 gcc compiler while AVX-based MMM has been compiled with MSVC2018, both without optimization to obtain worst-case performance impact results regardless of the employed compiler. Regarding the time measurements, we have employed the following libraries: i) "time.h" C library in AVX-based MMM and ii) "xtime_l.h", a specific Xilinx C library, in the sequential MMM function.

For performance experiments, matrix sizes have been selected aiming to assess the performance sensitivity on matrix sizes (square matrices) and the representativeness for performance evaluation (unbalanced matrices). We have assessed the performance impact of both the sequential and AVX-based MMM with the following matrix dimensions:

- Square matrices: we have performed a first set of experiments with square matrices (A, B and C) of dimensions NxN. We have run the experiments with matrix dimensions of $80x80$, $160x160$ and $320x320$.

- Unbalanced matrices: we have also evaluated the performance impact of unbalanced matrices with dimensions extracted from one of the most

repeated layers of our Darknet configuration ($L91$, where 91 refers to the position of the extracted layer in the CNN). In this case, the dimensions for matrices are MxK for A, KxN for B, and MxN for C. The dimensions of the matrices we have employed are: M=18, N=230400, K=64.

The assessment of the DC requires executing the MMM as many times as the number of bit positions in the matrices A and B (single-bit error). The computational cost required to perform this exhaustive fault-injection campaign at bit level has lead us to the choice of smaller matrix dimensions than those for the performance experiments:

- Square matrices: The dimensions of the matrices we have employed for the performance assessment of DC are $20x20$, $40x40$ and $80x80$.
- Unbalanced matrices: we have evaluated smaller matrices keeping the relationship between rows and columns proportional to some of the sequential MMM implementations of Darknet. The matrices are $L1$ (M=32, N=29, K=144), $L2$ (M=8, N=900, K=8) and $L3$ (M=15, N=225, K=48). These matrices have been chosen as an illustrative example of the variability in error detection with respect to the dimensions of the matrices A or B and the loop where the checksum is implemented. Up to here all the experiments have been analysed in both, sequential and AVX-based MMM. For completeness, we have also evaluated the DC of unbalanced matrices extracted from Darknet employing the sequential MMM. In this case,In this case, we have chosen the $L59$ layer due to the reduced size of its matrices, when compared to other Darknet layers. Its dimensions are as follows: M=18, N=900, K=1024.

5.2. Performance Impact

First of all, we define the performance impact as a ratio (n) in terms of execution time as shown in eq. (1) (where X and Y vary in function of the experiments):

$$n = \frac{Execution\ time_X}{Execution\ time_Y}\ [30] \tag{1}$$

As a first step, we have measured the performance impact incurred by the adoption of MISRA C coding guidelines and defensive programming. Here,

the performance impact is represented by the execution time of performing the MMM after adhering MISRA C (X) divided by the execution time of the original sequential MMM (Y). We have observed that these adaptations of the original MMM do not cause a relevant overhead in the execution time (below 1%). Additionally, we have slightly adapted the original code to optimize its performance in a 5% while still complying with MISRA C guidelines. This modification consists in the avoidance of unnecessary re-computations in the internal and intermediate loops by the insertion of auxiliary variables in the intermediate and external loops. According to eq. (1), in this experiment X refers to the sequential MMM accomplishing MISRA C after the optimizations and Y refers to the original sequential MMM.

For evaluating the performance slowdown incurred by the inclusion of the checksums, we first obtain the baselines (Y) for the sequential MMM and AVX-based MMM. To this end, we have measured the execution time incurred by each of the aforementioned matrix dimensions defined in the section 5.1 in both their sequential (MISRA C compliant MMM) and AVX-based versions with no integrated diagnostic mechanisms. These baselines reveal the execution time improvement achieved with the AVX-based implementation, which ranges between 3.97 and 6.57 faster that the sequential MMM for the different matrix sizes. Based on these values, we then obtain the performance impact incurred by the adoption of the ES on both implementations, where according to eq. (1), X relates to the MMM after the implementation of the catalogue of checksums. The results are depicted in Fig. 3 and Fig. 4 respectively.

The results of Fig. 3 are represented as follows: i) 80x80 is depicted with a green solid line and a square marker, ii) 160x160 with a red dashed line and round marker, iii) 320x320 with a blue dashed line and a triangular marker and iv) L91 with a yellow dot-and-dash line and a rhomboid marker. As previously mentioned, the performance impact of all these results are with respect to the execution time of the sequential MMM optimized without diagnostic mechanisms with the same matrix dimensions. As shown in Fig. 3, the performance impact decreases with increasing matrix size, approaching asymptotically specific values for each ES. This reduction is expected since larger matrices require an increasing number of memory accesses, which make decrease the performance impact of ES in relative terms. Furthermore, in this sequential implementation, the individual experiments confirm the increase in performance impact from the straightforward (XOR, 2's and 1's complement) to the more intricate algorithms (Fletcher and CRC) as well as a smaller

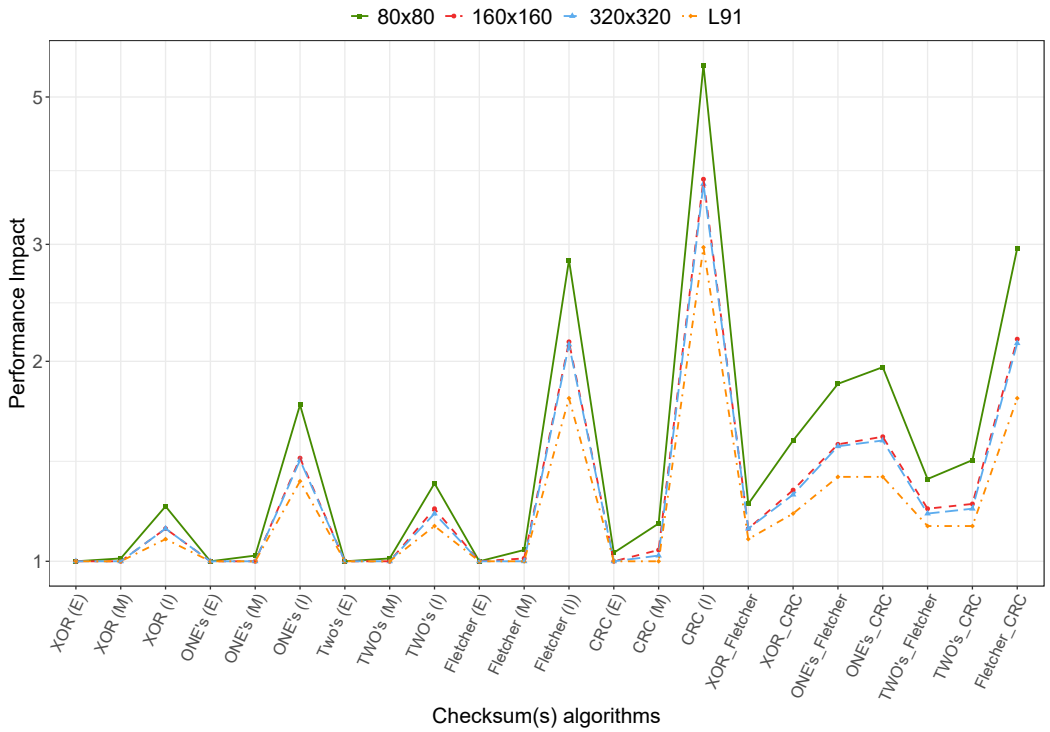

Figure 3: Sequential MMM: performance impact caused by the inclusion of a catalogue of checksum algorithms evaluated in square matrices of dimensions 80x80, 160x160 and 320x320 and in unbalanced matrices of dimensions M=18, N=230400, K=64 ($L91$).

impact on the most external loops (M, E) with respect to the internal loop (I). Additionally, we observe that, when the size of the matrices increases, the performance impact of the checksum combinations tends to approximate to the performance impact incurred by the individual checksum implemented in the internal loop. This means that for increasing matrix sizes, the impact of the intermediate checksum (M) is comparatively lower than the impact of the internal checksum (I) in relative terms.

Fig. 4 keeps the same colour and type of lines used in Fig. 3. In this case, the performance impact of those experiments are evaluated with respect to the execution time of the AVX-based MMM without diagnostic mechanisms with the same matrix dimensions. In Fig. 4, we can see an increase in the performance impact of the 1's complement checksum with respect to Fig. 3. This increase occurs due to the fact that 1's complement checksum requires the addition of all values to be checked and subsequently adding the carry

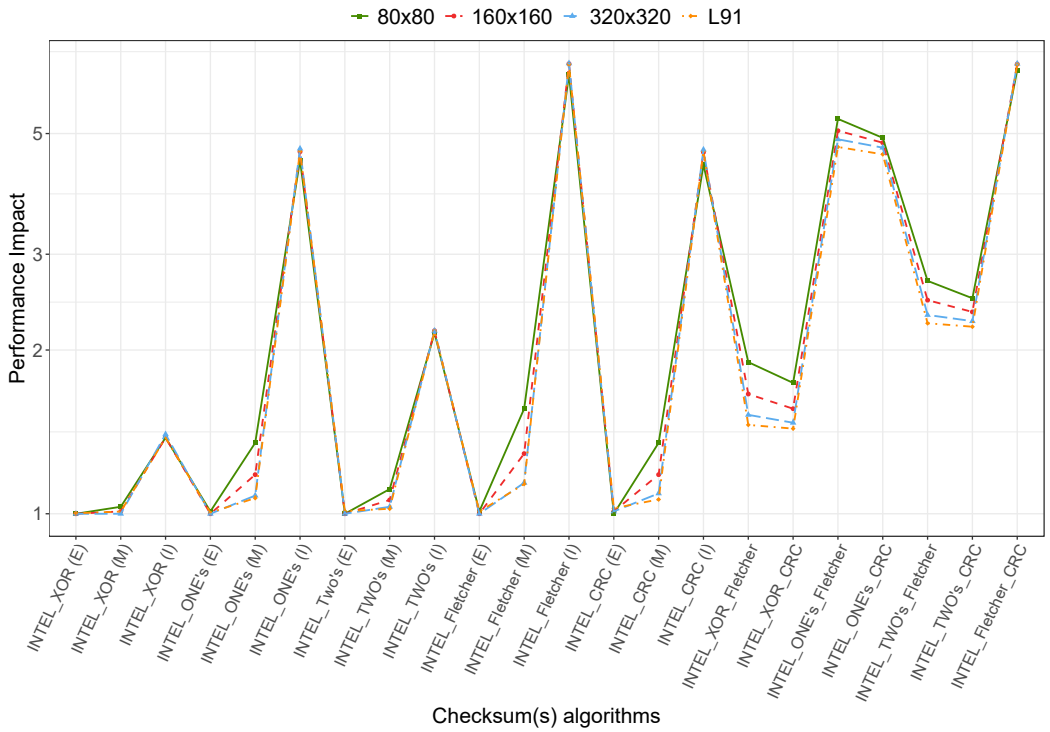

Figure 4: AVX MMM: performance impact incurred by the adoption of the catalogue of checksums in the MMM evaluated in square matrices of dimensions 80x80, 160x160 and 320x320 and in unbalanced matrices of dimensions M=18, N=230400, K=64 ($L91$).

bit back into the result before a final inversion [27]. With AVX instructions, the arithmetic operations lack of a carry bit and hence, this checksum is not suitable for AVX instruction-based implementations in terms of performance. Our solution to overcome this limitation rests in using larger data-types to compute 1's complement checksum (where we add the carry bit), causing higher overhead. In a similar way, the Fletcher's checksum implies a modulo operation for the extraction of the remainder from a specific division that is not considered by AVX instructions. Such arithmetic operation has to be implemented with sequential code that increases the performance impact of that diagnostic mechanism in the AVX-based MMM.

To conclude the performance experiments, we have analyzed the impact of the "Safe MMM" in the full Darknet CNN, obtaining the results depicted in Fig. 5. To offer an insight into the execution time required to process one image with YOLO in the selected architectures, we provide the baselines for

our experiments, which are obtained with the optimized MISRA C compliant sequential MMM and the AVX-based MMM, both without diagnostic mechanisms. The obtained values are 639.8 ms and 235.5 ms for the sequential (R5 cores) and AVX-based MMM (Intel i7) respectively. Regarding the impact of the different checksums, in Fig. 5, we can observe that the slowdown caused by the individual experiments in the external and intermediate loops is very close. As a consequence, it is preferable to apply the checksum in the intermediate loop rather than in the external loop, since it allows achieving a higher DC. In the combination experiments, the results reveal that the checksums added in the intermediate loop do not produce a significant slowdown with respect to that caused by the checksums applied in the internal loop. This is so except for the 1's_Fletcher combination in the AVX-based MMM, which is expected based on the results we have previously seen for the MMM, where both one's complement and fletcher are the two least suitable checksums for AVX-based instructions. Finally, it should be noted that since the CNN is a portion of the full YOLO algorithm, the performance impact of YOLO is expected to be lower that for Darknet in relative terms.

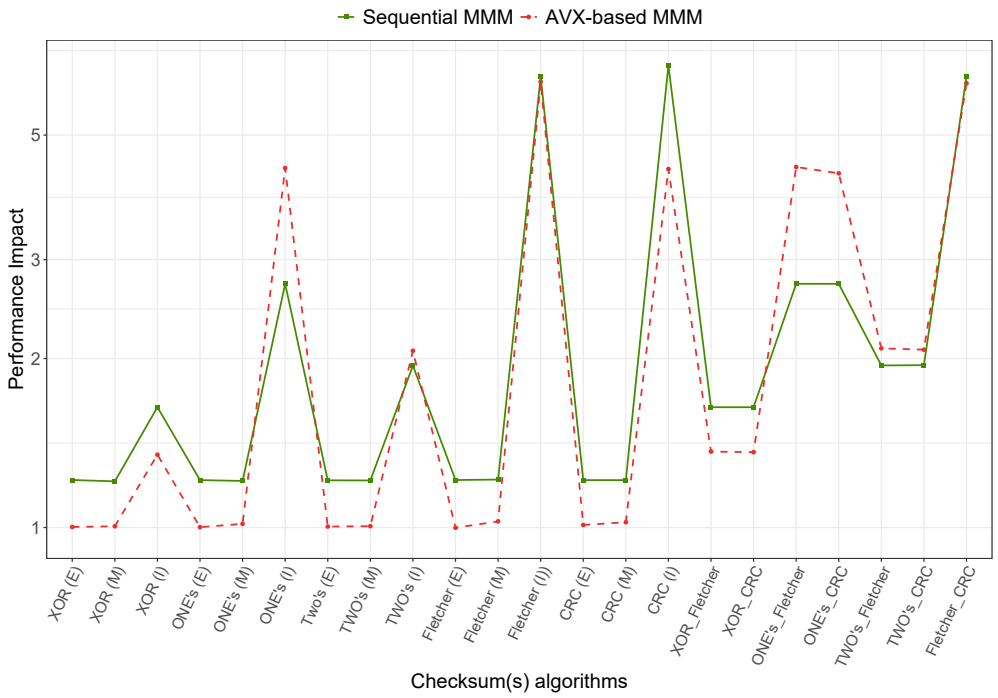

Figure 5: Darknet CNN: performance impact caused by the inclusion of a catalogue of checksum algorithms evaluated in Darknet CNN.

5.3. Diagnostic coverage

In order to quantify the potential DC of the individual ESs, their combinations and their implementation in the different loops (I, M, E) of the MMM, common approaches are to make analytical calculations, validation through experimental measurements with fault-injection campaigns, or a combination of both [27].

In this paper, we have performed an exhaustive fault-injection campaign in all bit positions of the values of matrices A and B for the evaluation of the DC. To this end, we base on the architectural pattern a) presented in Section 4.2 (periodic diagnosis with design time fixed data pattern). First, we obtain the ES with a fixed data, which is later used as reference ES. Then, we induce exhaustive single-bit fault-injections in matrices A or B and the resulting ES is compared with respect to the reference ES. In this way we are able to evaluate whether the checksums detect injected single-bit errors and we can compute a DC percentage.

We have evaluated individual ES, as well as some particularly relevant combinations. Table 1 gathers the results of the sequential and AVX-based MMM implementations for individual ES and their combinations in the previously mentioned matrix sizes. For simplicity, the combined techniques that reach a 100% DC for all matrix dimensions are not represented in Table 1. As a general rule, the DC is higher when the checksums are applied in the more internal loops, as the granularity of the diagnostics increases (i.e., in the internal loop all values of A, B and C matrices are contemplated in the checksum). However, the results in Table 1 show that the XOR, 1's complement and 2's complement checksums in the internal loop do not reach a 100% DC neither in sequential nor in AVX-based MMM. The reason is that, although the values of A, B and C are summarized with independent checksums, the final ES is obtained by combining these three variables and, therefore, some bit-errors can be masked. As explained in Section 4.1, this can be solved by combining these individual checksums with a Fletcher or CRC in the intermediate loop or by obtaining three different signatures (one for each matrix) instead of a combined one. Regarding the external and intermediate loops, the DC is highly dependent on the dimension of the matrix involved in the MMM. For instance, implementing Fletcher in the intermediate loop makes bit-error detection vary from more than 50% in the square matrix to a 1% in a given set of unbalanced matrices ($L2$), according to the results extracted from the sequential MMM and from 80% to 2.2% in the AVX-based MMM. The reason rests in the row/column proportion of the

input matrices. In the intermediate loop, the ES evaluates each of the values of the overall matrix A and only the last column of the matrix B and C. In unbalanced matrices, such as, $L2$ and $L3$ where the dimension of A and the number of columns of B is proportionally lower than the dimension of B, a considerable decrease of the achievable DC can be observed. Finally, the checksums in the external loop provide a weak DC as expected, since the ES only contemplates M (number of rows of matrix A) of the total possible combinations.

Table 1: Diagnostic coverage of the sequential MMM

	Sequential MMM						AVX-based MMM					
Checksum	Square			Unbalanced			Square			Unbalanced		
implemented	20	40	80	L1	L2	L3	20	40	80	L1	L2	L3
XOR (E)	2.5	1.3	0.6	0.4	0.1	0.1	2.5	1.3	0.6	0.4	0.1	0.1
XOR (M)	50.0	50.0	50.0	52.5	0.9	6.6	50.0	50.0	50.0	52.5	0.9	10.0
XOR (I)	50.0	50.0	50.0	52.5	0.9	100.0	50.0	50.0	50.0	52.5	0.9	100.0
One's (E)	2.5	1.3	0.6	0.4	0.1	0.1	2.5	1.3	0.6	0.4	0.1	0.2
One's (M)	52.5	51.2	50.6	54.1	1.0	7.1	79.2	59.2	54.2	72.9	2.2	9.9
One's (I)	98.5	97.7	96.9	98.4	97.7	96.9	99.2	99.2	99.2	98.9	99.2	99.9
Two's (E)	2.5	1.3	0.6	0.4	0.1	0.1	2.5	1.3	0.6	0.4	0.1	0.2
Two's (M)	52.3	51.1	50.6	54.1	1.0	7.1	68.8	59.1	54.4	63.5	1.7	9.6
Two's (I)	96.9	95.3	93.8	98.4	90.7	96.9	96.9	95.3	93.8	92.6	90.7	100.0
Fletcher(E)	2.6	1.3	0.6	0.4	0.1	0.1	3.5	1.5	0.7	0.5	0.2	0.2
Fletcher(M)	52.2	51.1	50.6	54.1	1.0	7.1	80.0	60.0	55.0	73.8	2.2	10.0
Fletcher(I)	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	99.9	99.9	100.0
CRC(E)	2.6	1.3	0.6	0.4	0.1	0.1	3.5	1.5	0.7	0.5	0.2	0.2
CRC(M)	52.5	51.3	50.6	54.1	1.0	7.1	80.0	60.0	55.0	73.77	2.2	10.0
CRC(I)	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
XOR_Fletcher	99.8	99.8	99.8	99.8	99.8	99.8	100.0	100.0	100.0	100.0	100.0	100.0
Two's_Fletcher	97.9	97.8	97.7	99.6	99.8	99.6	100.0	100.0	100.0	99.9	99.9	99.9

NOTE: The combinations not shown in the table reach a 100% DC

Accordingly, the internal loop solutions seem to be the most suitable ones from a DC perspective, but as we have seen in previous section, the execution time penalty they involve is considerably higher (e.g., CRC (I)). For this reason, we have defined the combined solutions that provide different checksum techniques in different loops, achieving a 100% DC with less performance impact (e.g., ONE'S_CRC).

Additionally, in Table 1 it is possible to see the difference between sequential and AVX-based checksum implementations. According to the XOR checksum, we can observe that for square matrices AVX-based and sequential MMM reach a very similar DC in the individual experiments, regardless of the loop where it is implemented. The greatest difference comes from the one's complement checksum, that reaches slightly higher DC in the AVX-based

implementation due to the required adaptation to overcome the absence of the carry bit. For the remaining individual experiments, we can appreciate a DC increase when AVX-based MMM is employed, which is explained by the single instructions with multiple data used by AVX. This causes the protected data in each checksum calculation with AVX to be higher than with sequential instructions. However, in experiments with unbalanced matrices implementing the Fletcher checksum in the internal loop, we can appreciate that with matrix dimensions $L1$ and $L2$ there is a decrease in the achieved DC, which remarks that this diagnostic mechanism is less appropriated in AVX-based implementations. Additionally, the experiments with unbalanced matrices highlight the impact of the matrix dimension in the achievable diagnostic coverage, where for instance, the DC of XOR (I) varies from 0.9% to 100%.

To conclude the DC experiments in sequential MMM, we have computed the DC of a layer extracted from YOLO ($L59$). We have computed the DC of the 1's and 2's complement checksums in the internal loop with respect to individual experiments. The individual CRC and Fletcher experiments in the internal loop have not been carried out since the achievable DC is 100% regardless of the dimension and the type of matrices involved in the MMM, as shown in Table 1. In the same manner, we have not evaluated the achievable DC of the XOR checksum implemented in the internal loop since from Table 1 we consider that its combination with another checksum, such as CRC or Fletcher, can be more interesting in terms of DC and performance impact. In contrast, we have evaluated the combination of 1's complement with CRC and the combination 2's complement with CRC to verify that the DC increases when we combine checksums with lower DC detection in the internal loop with checksum with higher a DC in the intermediate one. The results confirm our hypothesis, producing an increase from 98.5 % in 1's complement checksum and 96.9 % in 2's complement checksum, up to 100% DC when they are implemented in combination with the CRC checksum.

5.4. Trade-off between DC and performance impact

Based on previous results, in this section we evaluate the relationship between DC and performance impact for all considered checksum algorithms and their combinations for square matrices of dimension $80x80$ with the sequential MMM and with AVX-based implementation (see Fig. 6):

DC is represented in the right-hand y-axis and it is depicted with two green bar diagrams (the lighter one for the sequential implementation and

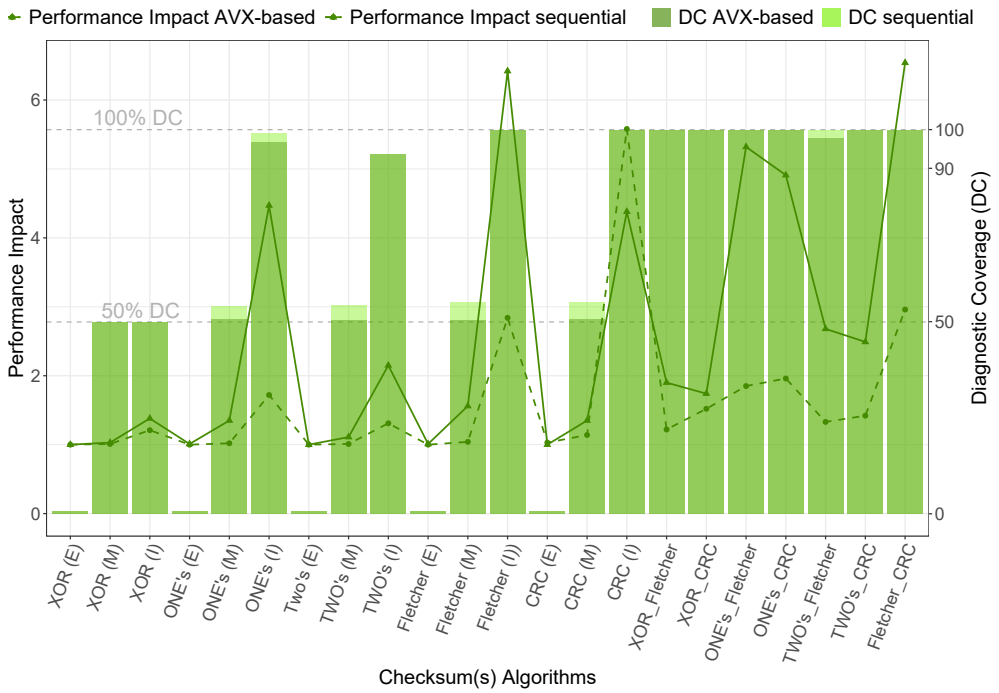

Figure 6: Trade-off between performance impact vs. DC: performance impact and DC values obtained after adopting the checksums catalogue with Sequential and AVX-based MMM implementation for square matrices of dimension $80x80$.

the darker one for AVX-based implementation). On the other side, the performance impact is represented in the left-hand y-axis, with the AVX-based implementation illustrated with a green solid line with triangular markers and the sequential implementation depicted with a green dashed line and round markers. As previously stated, not all selected combined approaches reach 100% DC and their execution time impact considerably varies from one checksum to another. Among the options with the highest DC, Fig. 6 depicts how the 2's complement and 1's complement checksums implemented in the internal loop and the combinations XOR_Fletcher and Twos_Fletcher do not reach 100% DC with the sequential implementation. However, AVX implementation allows to reach 100% DC with all checksum combinations and almost 100% with 2's complement implemented in the most internal loop. Although might not be evident in Fig. 6, XOR_Fletcher reaches 99,8% error detection instead of 100% in the sequential MMM (as described in Table 1). The reason of not reaching the maximum DC in comparison with the

checksum combination XOR_CRC is that the DC reached by the Fletcher checksum in the intermediate loop is slightly inferior than that reached by the CRC in the same loop and therefore some single-bit errors remain undetectable. Hence, the most promising performance results are provided by XOR_CRC, 2's_CRC, 1's_CRC and 1's_Fletcher checksums in sequential MMM and 2's_Fletcher, 2's_CRC, XOR_Fletcher and XOR_CRC in the AVX-based MMM. In particular, XOR_CRC offers the lowest performance impact for our particular sequential evaluation framework (R5 core of the Zynq UltraScale+ platform) and AVX-based evaluation framework (Intel i7 Cores).

5.5. Discussion on compliance

The evaluation shown in previous subsections results in a catalogue of checksums with varying degrees of performance impact and DC against single-bit errors. While the required performance is application dependant, for the DC IEC 61508-2 Table 3 determines the required coverage based on the SIL and the hardware fault tolerance (HFT) of the safety-related system. This allows the safety designer to select the most suitable checksum for each of the safety architectural patterns described in Section 4.2.

For the *periodic diagnosis with design time fixed data pattern*, which is based on a single channel architecture with diagnostics (HFT = 0), the standard requires a DC of at least 60% for complex elements whose failure modes cannot be easily determined. Therefore, as shown in previous Table 1, most of the individual checksums applied in the external or intermediate loops are not suitable by their own as a diagnostic mechanism for a safety-critical systems without redundancy. The DC is improved when the individual checksums are applied in the internal loop, but we have already seen that the slowdown caused in this case is considerably bigger, which could not be affordable from a real-time perspective. For this reason, the best options for this architectural pattern are within the combined checksums. In contrast, the architectural pattern based on *redundancy* can reach a HFT > 0. In this cases the standard permits reaching the same SIL as before with lower DC. For instance, for a HFT = 1 a DC below 60% is acceptable for up to SIL 1, and up to SIL 2 if the HFT is at least 2. Even in these cases, although the standard does not specify it, in practice a diligent safety system design requires a DC closer to 60% than to 0%.

Following these requirements from IEC 61508, we select the solution that provides best DC and performance impact ratio for Darknet CNN. However,

as we have already seen in previous subsections, the DC varies depending on the considered matrix dimensions and, for that reason, the adequacy of the checksum or combination of checksums should be evaluated for each of the layers of the CNN (which in Darknet are known at design time). As an example, in Table 2 we provide the selected checksums for each SIL and HFT in square matrices of $80x80$ dimensions. The grayscale of its cells refers to the range of diagnostic coverage, where the darker the gray the higher the diagnostic coverage required.

Table 2: Selected checksums for $80x80$ matrix dimension according to SIL and HFT.

	HFT					
	0		1		2	
SIL	Sequential	AVX	Sequential	AVX	Sequential	AVX
4	Non achievable		XOR_Fletcher$^{(iv)}$	XOR_CRC$^{(iv)}$	XOR_Fletcher$^{(iii)}$	XOR_CRC$^{(iii)}$
3	XOR_Fletcher$^{(iv)}$	XOR_CRC$^{(iv)}$	XOR_Fletcher$^{(iii)}$	XOR_CRC$^{(iii)}$	XOR_Fletcher$^{(ii)}$	XOR_CRC$^{(ii)}$
2	XOR_Fletcher$^{(iii)}$	XOR_CRC$^{(iii)}$	XOR_Fletcher$^{(ii)}$	XOR_CRC$^{(ii)}$	CRC (M)$^{(i)}$	CRC (M)$^{(i)}$
1	XOR_Fletcher$^{(ii)}$	XOR_CRC$^{(ii)}$	CRC (M)$^{(i)}$	XOR_CRC$^{(i)}$	Non specified	

NOTE: i) $DC < 60\%$ ii) $60\% < DC < 90\%$ iii) $90\% \leq DC < 99\%$ iv) $99\% \leq DC$

Table 2 shows how for the same SIL, a checksum with lower DC can be selected if it is implemented in a redundant architecture. For instance, for SIL 2, the single-channel pattern (HFT = 0) involves a high DC ($90\% \leq DC < 99\%$ (iii)) for which checksum combinations are the preferred option. For the redundant pattern instead, with HFT = 2 a low DC ($< 60\%$) is sufficient, and the CRC (M) individual checksum provides best performance vs required DC ratio, reaching a 50.6% (sequential) and 55.0% (AVX) DC for the selected matrix dimension. For high DC the XOR_Fletcher (sequential) and XOR_CRC (AVX) combinations allow achieving the required DC with a lower performance impact. The same solution turns out to be the most suitable for a medium DC ($60\% < DC < 90\%$ (ii)) too, since in our results there is not any checksum within this specific DC range.

6. Related Work

In the current literature, there are plenty of works devoted to the safety certifiability of ML-based solutions [9, 11, 31, 32, 33, 34]. There is a research line focused on identifying and analysing the main gaps for the adoption of ML components in safety-related system development processes, according to the requirements imposed by functional safety standards such as ISO 26262 or IEC 61508 [17, 32, 33]. In this research line, the paper [33], which is an extended work of [32], identifies five problems to adhere to ISO 26262

lifecycle with ML approaches and proposes five recommendations to address them. In the same manner, Hamid et al. [17] identify the main challenges of ML to adhere to the requirements for software development described in part 6 of the ISO 26262 standard and they state the necessity of a safe ML library. Besides, after an analysis of the deep learning framework in [11], the authors assert the direct impact of low-level libraries, mostly based on matrix operations, on these frameworks. They postulate that the optimization or development of new low-level libraries would be beneficial to address issues such as fault tolerance and promote reusability. This has served as motivation of our work where we have defined diagnostic mechanisms to detect errors in the computation of MMMs, the backbone of CNNs.

The evaluation of the performance of the MMM in diverse platforms such as GPUs and multi-core processors has also been carried out in terms of execution time [35, 36, 37], among others. According to the reliability enhancement, there are both hardware and software solutions. For hardware, Li et al. [13] propose the characterization of the CNN error propagation to select the most suitable latches to be hardened. However, these partial-hardware redundancies depend on the CNN and require specific hardware modifications that entail a great effort in view of the large variety of CNN models. According to their results, this latch hardening incurs in an area overhead between 20% and 25%. As the suitability of these hardware solutions is evaluated based on the required embedded area rather than by the execution time, it is not possible to provide a performance comparison with our solution. For software-level solutions, several research works propose the introduction of algorithm-based fault tolerance (ABFT) to enhance the reliability both on FPGA [38, 39] and on GPU [8, 40, 41]. These papers focus on the avoidance of soft-errors at runtime but they do not consider the errors caused by built-in mechanisms like cache coherency and the safe behaviour of all platform components. ABFT solutions offer low performance impact, as demonstrated by the authors in [40] with an overhead from 4% to 8%, up to 13.8% when employed with small matrix dimensions [41] and 3% with square matrices of dimensions $500x500$ [39]. In contrast to these works, the slowdown caused by the checksum catalogue of this paper has a considerably higher variability, ranging from 0.1% to 197% (in the sequential implementation) and up to 533% (in the AVX-based implementation). While the worst overhead is significantly higher with our approach, with the combined solutions we reached a reasonable overhead reaching a 100% diagnostic coverage for single-bit and program sequence errors at bit level, fact that cannot be guaranteed at the

same granularity with the ABFT approach. In addition, the wide variety of defined combinations allows the safety designer the selection of the most adequate solution according to the required SIL and architectural pattern.

With a broader scope, multiple testing and fault injection approaches exist for the verification and validation of automotive systems. While some of them are not explicit for ML-based systems, they can also be applied to those. Some works assess software-based defect detection by means of mutation testing and fault injection in the context of autonomous driving systems [42]. Other works focus on how to test the system with different sets of inputs [43, 44, 45]. In a more classical strand, model-based software design and testing can be used for verification and validation activities [46].

While the target of this work is diagnostics based on checksums, fault detection can also be performed with execution redundancy. Some processors provide such redundancy, along with diversity, by hardware means with lockstep architectures, such as the Infineon AURIX processor family [47], the ST Microelectronics SPC56XL70 [48] some Arm-based designs [49]. Other works provide redundancy with lighter-weight approaches based on single-core thread redundancy [50, 51] or multi-core thread redundancy [52, 53, 54]. Some works even consider only partial redundancy [55, 56]. Software-only mechanisms for execution redundancy have also been widely studied in the context of CPUs [57, 58, 59, 60, 61, 62], including mechanisms such as a monitor process to detect errors, or leveraging compilers to inject redundancy. This type of solutions have also been explored for accelerators such as the Kalray MPPA family and GPUs, either at hardware level [63, 64, 65, 66, 67] or at software-only level [68, 63, 69, 70]. In all cases, however, those solutions are orthogonal to those studied in this paper and need to be carefully applied only whenever needed due to their costs, and meeting safety requirements in ISO 26262 (e.g. such as diverse redundancy for the highest integrity levels).

Overall, while checksums themselves are not new, their applicability in ML-related software for automotive systems and the different performance and coverage tradeoffs, which are the main contributions of this work, have not been considered in existing work yet. Existing works, instead, focus on the correctness of ML at functional level, on the performance of MMM for high-performance systems (without safety or diagnostics considerations), on fault detection during system development (not online diagnostics), and on fault detection through redundancy (with or without diversity). The latter is normally used to reach ASIL C or D building on ASIL A and B components. However, checksums allow reaching ASIL A and B, and hence, both types of

mechanisms are complementary.

Regarding MISRA-C compliance, while the work in [11] does some initial steps towards generally assessing the compliance of a full autonomous driving framework. Differently, our work performs a detailed analysis of a specific – and central – software component in those frameworks, and adapts it to be compliant.

7. Conclusions and future work

In this paper we have presented an approach to adapt the MMM functions present in ML software libraries in order to avoid and control systematic and random errors according to the considered functional safety standards. On the one hand, for systematic error avoidance, we have seen that the effort to adapt the original MMM code to MISRA C coding guidelines was relatively low and with negligible performance impact. This same conclusion can also be extrapolated to the complete Darknet CNN, where we have proposed solutions for the most frequent violations. On the other hand, the paper has presented an approach for runtime fault detection based on a combination of ESs and safe architectural patterns. We have evaluated the trade-off between performance penalty ratio and DC, achieving up to 100% DC for single-bit inversions with a performance impact from *1.001* to *2.97* for the analyzed largest matrix sizes computed with the sequential MMM and from *1.001* to *6.33* with the AVX-based MMM. Additionally, the experimental results confirm that for a given DC and performance penalty target, the selection of the appropriate combination of checksums depends on the considered matrix dimensions (which for YOLO are known at design time) and this allows the adoption of the most appropriate ones according to the specific application. In particular, we have provided a pre-selection of most suitable checksums for the CNN layer that corresponds to the $80x80$ matrix dimension for two different safety architectural patterns, one based on periodic diagnostics and the second for redundant architectures with varying degrees of fault tolerance. Additionally, we confirm that our catalogue can be implemented employing AVX-based vectorization for the sake of performance. The experiments demonstrate that the vectorization of certain checksums, such as 1's complement and Fletcher, implies a performance penalty that makes them less appropriate in these implementations in terms of performance. Besides that, the AVX-based MMM allows reaching higher performance than that provided

by the sequential MMM, improving execution time by a factor between 3.97 and 6.57 for different matrix sizes.

Overall, this paper is an initial step to pave the way towards the development of a 'safe ML library' adhering to functional safety standards. This first development provides a platform independent safe inference execution environment for the MMM. As part of our future work, following the incremental strategy exposed in this paper, the authors plan to adapt the proposed techniques to other possible inference alternatives that exploit parallelization [34], such as GPUs and FPGAs. In addition, with the aim of reducing the slowdown of the proposed techniques when applied to a specific CNN, we propose the application of the checksums only on the most error prone layers and most sensitive bits, preceded by a detailed study of the error propagation. In addition, the most suitable checksum or checksum combinations for each layer shall be selected, based on the matrix dimension, the target SIL and the architectural pattern, extending the analysis done for $80x80$ matrices in Section 5.5 of this paper. Based on the selected options, an additional future research topic could extend the performance impact and DC evaluation of the complete CNN considering also multiple-bit errors.

References

[1] D. Casini, A. Biondi, and G. Buttazzo, "Deep neural networks for safety-critical applications : Vision and open problems," in *Proc. of the 9th Int. Real-Time Scheduling Open Problems Seminar (RTSOPS)*, 2018, Conference Proceedings.

[2] L. Jiao, F. Zhang, F. Liu, S. Yang, L. Li, Z. Feng, and R. Qu, "A survey of deep learning-based object detection," *IEEE Access*, vol. 7, pp. 128 837–128 868, 2019.

[3] J. Redmon and A. Farhadi, "YOLOv3: An incremental improvement," 2018.

[4] "IEC 61508(-1/7): Functional safety of electrical / electronic / programmable electronic safety-related systems," 2010.

[5] "ISO 26262(-1/11) road vehicles – functional safety," ISO, 2018.

[6] "ISO/PAS 21448 road vehicles – safety of the intended functionality," ISO, 2019.

[7] F. dos Santos, L. Carro, and P. Rech, "Kernel and layer vulnerability factor to evaluate object detection reliability in GPUs," *IET Computers & Digital Techniques*, vol. 13, 2018.

[8] F. F. d. Santos, L. Draghetti, L. Weigel, L. Carro, P. Navaux, and P. Rech, "Evaluation and mitigation of soft-errors in neural network-based object detection in three GPU architectures," in *47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, 2017, Conference Proceedings, pp. 169–176.

[9] H. Tabani, L. Kosmidis, J. Abella, F. J. Cazorla, and G. Bernat, "Assessing the adherence of an industrial autonomous driving framework to ISO 26262 software guidelines," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*, 2019, Conference Proceedings, pp. 1–6.

[10] A. Bosio, P. Bernardi, A. Ruospo, and E. Sanchez, "A Reliability Analysis of a Deep Neural Network," in *2019 IEEE Latin American Test Symposium (LATS)*, 2019, Conference Proceedings, pp. 1–6.

[11] H. Tabani, R. Pujol, J. Abella, and F. J. Cazorla, "A cross-layer review of deep learning frameworks to ease their optimization and reuse," in *IEEE 23rd International Symposium on Real-Time Distributed Computing (ISORC)*, 2020, pp. 144–145.

[12] A. Bosio, P. Bernardi, A. Ruospo, and E. Sanchez, "A reliability analysis of a deep neural network," in *2019 IEEE Latin American Test Symposium (LATS)*, 2019, Conference Proceedings, pp. 1–6.

[13] G. Li, S. K. S. Hari, M. Sullivan, T. Tsai, K. Pattabiraman, J. Emer, and S. W. Keckler, "Understanding error propagation in deep learning neural network (DNN) accelerators and applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. Association for Computing Machinery, 2017, Conference Proceedings, p. 8, doi: 10.1145/3126908.3126964.

[14] A. Ruospo, A. Bosio, A. Ianne, and E. Sanchez, "Evaluating convolutional neural networks reliability depending on their data representation," in *2020 23rd Euromicro Conference on Digital System Design (DSD)*, 2020, pp. 672–679.

[15] J. Redmon, "Darknet: Open source neural networks in C," 2013–2016. [Online]. Available: http://pjreddie.com/darknet/

[16] J. Diaz, C. Muñoz-Caro, and A. Niño, "A survey of parallel programming models and tools in the multi and many-core era," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 8, pp. 1369–1386, 2012.

[17] H. Tabani, L. Kosmidis, J. Abella, F. J. Cazorla, and G. Bernat, "Assessing the adherence of an industrial autonomous driving framework to ISO 26262 software guidelines," in *Proceedings of the 56th Annual Design Automation Conference*, New York, NY, USA, 2019.

[18] J.-L. Boulanger, *Polyspace*. John Wiley & Sons, Inc, 2013, book section 3, p. 113–142, doi: 10.1002/9781118602867.

[19] "EN50128 - railway applications: Communication, signalling and processing systems - software for railway control and protection systems," 2011.

[20] J. Perez Cerrolaza, R. Obermaisser, J. Abella, F. J. Cazorla, K. Grüttner, I. Agirre, H. Ahmadian, and I. Allende, "Multi-core devices for safety-critical systems: A survey," *ACM Comput. Surv.*, vol. 53, no. 4, 2020.

[21] J. Redmon and A. Farhadi, "Yolo9000: Better, faster, stronger," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 6517–6525.

[22] A. Bochkovskiy, C.-Y. Wang, and H.-Y. M. Liao, "Yolov4: Optimal speed and accuracy of object detection," *arXiv preprint arXiv:2004.10934*, 2020.

[23] "MISRA C:2012 - guidelines for the use of the C language in critical systems," MISRA, 2012.

[24] J. Athavale, A. Baldovin, R. Graefe, M. Paulitsch, and R. Rosales, "AI and reliability trends in safety-critical autonomous systems on ground and air," in *50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, 2020, Conference Proceedings, pp. 74–77.

[25] A. Azizimazreah, Y. Gu, X. Gu, and L. Chen, "Tolerating Soft Errors in Deep Learning Accelerators with Reliable On-Chip Memory Designs," in *IEEE International Conference on Networking, Architecture and Storage (NAS)*, 2018, Conference Proceedings, pp. 1–10.

[26] J. Perez, D. Gonzalez, C. F. Nicolas, T. Trapman, and J. M. Garate, "A safety certification strategy for IEC-61508 compliant industrial mixed-criticality systems based on multicore partitioning," in *Euromicro conference on Digital System Design (DSD)*, 2014, Conference Proceedings.

[27] T. C. Maxino and P. J. Koopman, "The effectiveness of checksums for embedded control networks," *IEEE Transactions on Dependable and Secure Computing*, vol. 6, no. 1, pp. 59–72, 2009.

[28] P. Koopman, K. Driscoll, and B. Hall, "Selection of cyclic redundancy code and checksum algorithms to ensure critical data integrity," Carnegie Mellon University, Report, 2015.

[29] J. Ray and P. Koopman, "Efficient high hamming distance CRCs for embedded networks," in *International Conference on Dependable Systems and Networks (DSN)*, 2006, Conference Proceedings, pp. 3–12.

[30] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.

[31] F. Falcini and G. Lami, "Challenges in certification of autonomous driving systems," in *2017 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2017, Conference Proceedings, pp. 286–293.

[32] R. Salay, R. Queiroz, and K. Czarnecki, "An Analysis of ISO 26262: Using Machine Learning Safely in Automotive Software," *ArXiv*, vol. abs/1709.02435, 2018.

[33] R. Salay and K. Czarnecki, "Using Machine Learning Safely in Automotive Software: An Assessment and Adaption of Software Process Requirements in ISO 26262," *arXiv preprint arXiv:1808.01614*, 2018.

[34] A. Biondi, F. Nesti, G. Cicero, D. Casini, and G. Buttazzo, "A safe, secure, and predictable software architecture for deep learning in safety-critical systems," *IEEE Embedded Systems Letters*, pp. 1–1, 2019.

[35] M. Salim, A. O. Akkirman, M. Hidayetoglu, and L. Gurel, "Comparative benchmarking: matrix multiplication on a multicore coprocessor and a GPU," in *Computational Electromagnetics International Workshop (CEM)*, 2015, Conference Proceedings, pp. 1–2.

[36] Z. Huang, N. Ma, S. Wang, and Y. Peng, "GPU computing performance analysis on matrix multiplication," *The Journal of Engineering*, vol. 2019, 2019.

[37] V. Kelefouras, A. Kritikakou, I. Mporas, and V. Kolonias, "A high performance matrix-matrix multiplication methodology for CPU and GPU architectures," *The Journal of Supercomputing*, vol. 72, 2016.

[38] I. C. Lopes, F. Benevenuti, F. L. Kastensmidt, A. A. Susin, and P. Rech, "Reliability analysis on case-study traffic sign convolutional neural network on apsoc," in *2018 IEEE 19th Latin-American Test Symposium (LATS)*, Conference Proceedings, pp. 1–6.

[39] S. Roffe and A. D. George, "Evaluation of algorithm-based fault tolerance for machine learning and computer vision under neutron radiation," in *IEEE Aerospace Conference*, 2020, Conference Proceedings, pp. 1–9.

[40] K. Zhao, S. Di, S. Li, X. Liang, Y. Zhai, J. Chen, K. Ouyang, F. Cappello, and Z. Chen, "FT-CNN: Algorithm-Based Fault Tolerance for Convolutional Neural Networks," *IEEE Transactions on Parallel and Distributed Systems*, p. arXiv:2003.12203, 2020. [Online]. Available: https://ui.adsabs.harvard.edu/abs/2020arXiv200312203Z

[41] C. Braun, S. Halder, and H. J. Wunderlich, "A-ABFT: Autonomous Algorithm-Based Fault Tolerance for Matrix Multiplications on Graphics Processing Units," in *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2014, pp. 443–454.

[42] R. Rana, M. Staron, C. Berger, J. Hansson, M. Nilsson, and F. Törner, "Early Verification and Validation According to ISO 26262 by Combining Fault Injection and Mutation Testing," in *Software Technologies*, J. Cordeiro and M. van Sinderen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 164–179.

[43] K. Pei, Y. Cao, J. Yang, and S. Jana, "DeepXplore: Automated Whitebox Testing of Deep Learning Systems," *Commun. ACM*,

vol. 62, no. 11, p. 137–145, Oct. 2019. [Online]. Available: https://doi.org/10.1145/3361566

[44] R. B. Abdessalem, A. Panichella, S. Nejati, L. C. Briand, and T. Stifter, "Testing autonomous cars for feature interaction failures using many-objective search," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 143–154. [Online]. Available: https://doi.org/10.1145/3238147.3238192

[45] C. Berger, "Accelerating regression testing for scaled self-driving cars with lightweight virtualization: A case study," in *Proceedings of the First International Workshop on Software Engineering for Smart Cyber-Physical Systems*, ser. SEsCPS '15. IEEE Press, 2015, p. 2–7.

[46] M. Broy, S. Kirstan, H. Krcmar, and B. Schätz, *What is the Benefit of a Model-Based Design of Embedded System in the Car Industry?* IGI global, 2012.

[47] Infineon, "AURIX Multicore 32-bit Microcontroller Family to Meet Safety and Powertrain Requirements of Upcoming Vehicle Generations," 2012.

[48] STMicroelectronics, "32-bit Power Architecture microcontroller for automotive SIL3/ASILD chassis and safety applications," 2014.

[49] X. Iturbe et al., "The Arm triple core lock-step (TCLS) processor," *ACM Transactions on Computer Systems*, 2019.

[50] S. K. Reinhardt et al., "Transient fault detection via simultaneous multithreading," in *ISCA*, 2000.

[51] E. Rotenberg, "AR-SMT: a microarchitectural approach to fault tolerance in microprocessors," *FTC*, 1999.

[52] S. S. Mukherjee et al., "Detailed design and evaluation of redundant multithreading alternatives," in *ISCA*, 2002.

[53] M. Gomaa et al., "Transient-fault recovery for chip multiprocessors," in *ISCA*, 2003.

[54] C. LaFrieda et al., "Utilizing dynamically coupled cores to form a resilient chip multiprocessor," in *DSN*, 2007.

[55] B. H. Meyer et al., "Cost-effective safety and fault localization using distributed temporal redundancy," in *CASES*, 2011.

[56] J. Fu et al., "On-demand thread-level fault detection in a concurrent programming environment," in *SAMOS*, 2013.

[57] G. A. Reis et al., "SWIFT: Software implemented fault tolerance," in *CGO*, 2005.

[58] H. So et al., "Expert: Effective and flexible error protection by redundant multithreading," *DATE*, 2018.

[59] F. Haas et al., "Fault-tolerant execution on cots multi-core processors with hardware transactional memory support," in *ARCS*, 2017.

[60] H. Mushtaq et al., "Efficient software-based fault tolerance approach on multicore platforms," in *DATE*, 2013.

[61] A. Shye, T. Moseley, V. J. Reddi, J. Blomstedt, and D. A. Connors, "Using process-level redundancy to exploit multiple cores for transient fault tolerance," in *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2007, pp. 297–306.

[62] A. Shye, J. Blomstedt, T. Moseley, V. J. Reddi, and D. A. Connors, "Plr: A software approach to transient fault tolerance for multicore architectures," *IEEE Transactions on Dependable and Secure Computing*, vol. 6, no. 2, pp. 135–148, 2009.

[63] J. Wadden, A. Lyashevsky, S. Gurumurthi, V. Sridharan, and K. Skadron, "Real-world design and evaluation of compiler-managed gpu redundant multithreading," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, 2014, pp. 73–84.

[64] H. Jeon et al., "Warped-DMR: Light-weight error detection for GPGPU," in *MICRO*, 2012.

[65] M. B. Sullivan et al, "SwapCodes: Error Codes for Hardware-Software Cooperative GPU Pipeline Error Detection," in *MICRO*, 2018.

[66] R. Nathan and D. J. Sorin, "Argus-G: Comprehensive, low-cost error detection for GPGPU cores," *IEEE Computer Architecture Letters*, 2015.

[67] S. Alcaide et al., "Software-only Diverse Redundancy on GPUs for Autonomous Driving Platforms," in *IOLTS*, 2019.

[68] M. Dimitrov, M. Mantor, and H. Zhou, "Understanding software approaches for gpgpu reliability," in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, ser. GPGPU-2. New York, NY, USA: Association for Computing Machinery, 2009, p. 94–104. [Online]. Available: https://doi.org/10.1145/1513895.1513907

[69] S. Jain et al., "Fractional GPUs: Software-based compute and memory bandwidth reservation for GPUs," in *RTAS*, 2019.

[70] S. Alcaide et al., "High-Integrity GPU Designs for Critical Real-Time Automotive Systems," in *DATE*, 2019.