UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Departament d'Arquitectura de Computadors

Universitat Politècnica de Catalunya
Computer Architecture Department

Dissertation

Software Diagnostics for Autonomous Safety-Critical Control-Systems Based On Artificial Intelligence

Javier Fernández Muñoz

Advisor Jaume Abella Ferrer
Barcelona Supercomputing Center (BSC)
Computer Architecture and Operating Systems Group (CAOS)

Co-advisor Irune Agirre Troncoso
Ikerlan Technology Research Centre
Dependable Embedded Systems Department

Tutor Miquel Moretó Planas
Universitat Politècnica de Catalunya (UPC)
Computer Architecture and Operating Systems Group (CAOS)

May, 2023

Javier Fernández Muñoz
Software Diagnostics for Autonomous Safety-Critical Control-Systems Based On Artificial Intelligence
Dissertation, May, 2023
Advisors: Jaume Abella Ferrer and Irune Agirre Troncoso
Tutor: Miquel Moretó Planas

Universitat Politècnica de Catalunya
Computer Architecture Department
Carrer de Jordi Girona, 1, 3
08034 and Barcelona

Abstract

Machine Learning (ML) systems allow the efficient implementation of functionalities that can be hard to program by traditional software due to the high spectrum of inputs that hinder the definition of a specific procedural rule set. This characteristic of ML systems has encouraged their adoption in applications such as object detection or image classification in several safety-related domains, which are subject to safety certification. This certification is usually achieved by adhering to traditional functional safety standards such as IEC 61508 [1] or ISO 26262 [2]. However, these standards were not devised to accommodate technologies such as ML in safety-related systems due to their development process, which is based on probabilistic models generated from training data, as opposed to traditional software components coded from specifications. Additionally, new challenges arise due to the fact that these ML algorithms need to process large volumes of data, and this requires High-Performance Embedded Computing (HPEC) platforms with computing capabilities far superior to traditional safety systems, such as multicore devices and GPU accelerators. Current functional safety standards do not provide explicit guidance for the use of HPEC platforms in safety-relevant systems, and the inherent complexity of those highly parallel architectures challenges certifications. With this Thesis, we attempt to address these challenges and give a step forward towards the functional safety certification of safety control systems integrating ML components in HPEC platforms.

Acknowledgement

I am deeply grateful to the many people who have contributed to this Thesis's success. My advisors, Dr. Jaume Abella and Dr. Irune Agirre, and my tutor, Dr. Miquel Moretó, have my utmost appreciation. Their knowledge, guidance, and encouragement were invaluable throughout the entire process. This Thesis has become a reality thanks to their continued support and contributions.

I also extend my gratitude to Mr. Peio Onaindia and Dr. Jon Perez for their supervision and willingness to offer advice and assistance when needed. Their encouragement kept me motivated during difficult times. I am thankful to Ikerlan for allowing me to join their team and for funding this Thesis. This opportunity is truly a lifetime achievement. I would also like to thank the Dependable Embedded Systems area, particularly my colleagues from the Real-Time Systems team. Working alongside such a reliable and supportive team has made daily challenges more enjoyable, and I am constantly learning from them. I would especially like to thank all the people who have supported me in the roller coaster that has been the Thesis. In particular to Alex (the good, the bad, and the maje), Aritz, Segura, Agirre, Yarza, Angel, Juan, and many people that I will not mention so as not to make the list too long but with whom I appreciate the time shared.

I am grateful to the CAOS group from the BSC for their warm welcome during my time working with them in Barcelona. It was a valuable learning experience to collaborate with such qualified professionals. My wonderful family and friends have been by my side throughout this process, and I am forever grateful for their unwavering support. I extend special thanks to my mother (Rosita), father (Manolo), and brothers (Angelito y Manuel Jesús), who have always believed in me. Their contribution to this achievement cannot be overstated. I also want to express my appreciation to my wonderful friends: my "Kuadrila" from Bilbao (Rafita, Moni, Cris, Ritxi...), my "Pandilla" from my local village (Tuki, Josan, Adri...), friends from the university that stay today supporting me (Almu, "Guitarra"...), "los de siempre" (Mochi, Luichi, "Jamon"...) and my amazing flatmate and friend Pep, whose support has been a blessing. Of course, I am deeply grateful to Belén, who has been there suffering and enjoying the Thesis even more than me, thanks to her endless calls and unforgettable trips.

Contents

List of Figures

List of Tables

List of Acronyms

AAM	Advanced Air Mobility
ADAS	Advanced Driver-Assistance System
AI	Artificial Intelligence
AV	Autonomous Vehicles
ABFT	Algorithm-Based Fault Tolerance
ABED	Algorithm-Based Error Detection
APU	Application Processor Unit
ASIL	Automotive Safety Integrity Level
ANSI	American National Standards Institute
AVX	Advanced Vector Extensions
BC	Block Column
BR	Block Row
CCF	Common Cause Failure
CNN	Convolutional Neural Network
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
cuBLAS	CUDA Basic Linear Algebra Subroutine
CUTLASS	CUDA Templates for Linear Algebra Subroutines
DC	Diagnostic Coverage
DMR	Dual Modular Redundancy
DNN	Deep Neural Network
DTI	Diagnostic Test Interval
ES	Execution Signature
EUC	Equipment Under Control
FC	Fully-Connected
FCS	Frame Check Sequence
FN	False Negative
FP	False Positive
FPGA	Field-Programmable Gate Array
GEMM	General Matrix Multiplication
GPS	Global Positioning System
GPU	Graphics Processing Unit
HARA	Hazard Analysis and Risk Assessment
HFT	Hardware Fault Tolerance
HPEC	High-Performance Embedded Computing
HR	Highly-Recommend

IFM	Input Feature Map
IMU	Inertial Measurement Unit
ISA	Instruction Set Architecture
LSB	Least Significant Bit
MISRA	Motor Industry Software Reliability Association
ML	Machine Learning
MMA	Matrix Multiply-Accumulate
MMM	Matrix-Matrix Multiplication
MPSoC	Multi-Processor System-on-Chip
MSB	Most Significant Bit
NN	Neural Network
NR	Non-Recommended
PAS	Publicly Available Specification
PFD	Probability of Failure on Demand
PFH	Probability of Failure per Hour
PL	Programmable Logic
PS	Processing System
PST	Process Safety Time
PTX	Parallel Thread Execution
QM	Quality Management
R	Recommended
RL	Research Line
RPU	Real-time Processing Unit
SBST	Software-Based Self Test
SIL	Safety Integrity Level
SIMD	Single Instruction Multiple Data
SIMT	Single Instruction Multiple Thread
SRAM	Static Random-Access Memory
SSE	Streaming SIMD Extensions
SOTIF	Safety Of The Intended Functionality
SPI	Safety Performance Indicators
TMR	Triple Modular Redundancy
OFM	Output Feature Map
UAS	Unmanned Aircraft Systems
UCI	Uncertainty Confidence Indicator
UL	Underwriters Laboratories
XOR	Exclusive OR
YOLO	You Only Look Once

1 Introduction

In recent years, the presence of autonomous systems has become pervasive, even in everyday activities from vacuum cleaners to self-driving cars. ISO/IEC 22989:2022 defines *autonomous systems* as those "*capable of modifying their operating domain or goal without external intervention, control, or oversight*" [3]. The autonomy of the systems or capacity for self-governance is a characteristic that has fostered their application in several domains over the past few years. Among them are Autonomous Vehicles (AV) [4], Advanced Air Mobility (AAM) [5], [6], Unmanned Aircraft Systems (UAS) [7], [8], and a wide variety of implementations in the field of robotics, such as surgery robots [9], [10].

Some standards classify autonomous systems across different levels of automation by taking different reference criteria. For example, the SAE J3016 standard defines six levels of automation in the autonomous driving domain ranging from no driving automation (level 0) to full driving automation (level 5) [11]. ISO/IEC 22989:2022 shares the same classification as SAE J3016, but denotes them as heteronomous systems and does not define an application domain. In fact, ISO/IEC 22989:2022 differentiates between heteronomous systems, which operate under the constraint of external intervention, control, or oversight, and autonomous systems, which possess the ability to modify their operating domain or goals without the need for external intervention, control, or oversight.

The advent of AI technology and its ability to perform complex tasks with competitive performance supports the potential for achieving the highest levels of heteronomy and, finally, autonomy. This feature has attracted the attention of autonomous systems designers, triggering the widespread adoption of AI-based solutions in general and Machine Learning (ML)-based solutions in particular in multiple domains. The ability of Deep Neural Networks (DNNs), a subfield of ML, and more particularly Convolutional Neural Networks (CNNs), a subfield of DNN, to efficiently and accurately perform complex functionalities such as those related to object detection, location and classification tasks, is crucial in the development of computing vision, one of the most critical functionalities in autonomous driving [12].

However, failures in autonomous systems performing safety-related tasks could have catastrophic consequences (e.g., failures in the control of the braking systems can lead to accidents causing fatalities). Therefore, these safety-related autonomous

systems are subject to certification, usually achieved by providing evidence of adherence to functional safety standards. On the one side, it is necessary to guarantee that the AI model is safe for its intended purpose and that uncertainty-related risks are sufficiently mitigated. On the other side, risks associated with deploying such AI models in embedded platforms shall also be controlled and mitigated. Common solutions rely on High-Performance Embedded Computing (HPEC) platforms that meet the performance demands and AI software frameworks that facilitate the integration of such models into hardware platforms.

1.1 Safety Implications of Artificial Intelligence

Safety-related systems are those whose failure could result in casualties, significant property damage, or environmental, equipment or machinery damage [1]. For that reason, those systems are usually subject to certification requiring a robust system design in the development process that shall guarantee that design faults (i.e., systematic faults) are mitigated in the design process, and that unpredictable faults (i.e., random faults) are controlled at run-time in such a way that the residual risk of failure of the system is acceptably low. Traditionally, this has been achieved by proving adherence to applicable functional safety standards such as IEC 61508 [1] or ISO 26262 [2]. These standards define the procedures, requirements, techniques, and safety measures to reduce such risk to acceptable levels. However, they were not originally designed to accommodate the use of ML in the deployment of safety functions; therefore, the certification of these systems is currently an open research challenge.

ML algorithms are based primarily on statistical learning. A task is performed based on a probabilistic model generated from training data instead of from its specifications. This technique enables the implementation of functionalities that are harder to program by traditional software programming paradigms because manually formulating rules for the large spectrum of possible inputs is a tedious task that may be impracticable due to the large and multidimensional input space. Nevertheless, the traditional functional safety standards were designed for the latest, in which the designer programs according to a set of requirements in a deductive way. The inductive way ML operates (probabilistic models built from training data) supposes a shift in the programming paradigm that these standards did not contemplate when they were designed. E.g., notions such as the specifications that conventionally apply to the code itself may now encompass the learning process, and the standards do not consider this process [13].

ML algorithms require handling massive volumes of data with the consequent demand of high-performance computational capabilities for their execution, both at the software and hardware levels.

Regarding software, ML usually relies on widely used frameworks and tools built upon highly efficient low-level libraries that ease software development and increase hardware utilization [14]. These libraries include the required low-level operations and usually target to achieve the highest performance on the underlying implementation platform. In CNNs, these compute-intensive operations are linear algebra operations such as Matrix-Matrix Multiplications (MMMs). There are several platform-dependant implementations focusing on these algebraic operations— for instance, i) ATLAS [15] and OpenBLAS [16] libraries for CPU-based implementations and ii) CUDA Basic Linear Algebra Subroutine (cuBLAS) [17] and CUDA Templates for Linear Algebra Subroutines (CUTLASS) [18] libraries for Graphics Processing Units (GPUs) [14].

Nevertheless, using these highly-optimized libraries involves new certification challenges from a safety point of view. These libraries usually share a closed-source nature because of competition concerns and are not developed according to safety standards. Their use in safety domains implies their adaptation to safety procedures and requirements by library owners or the application of black box testing by the end users, which may limit their applicability [19]. As an alternative to closed-source libraries, another option lies in employing an existing open-source library offering competitive performance as a baseline and modifying it to accomplish functional safety standards.

Regarding hardware, the required performance for ML algorithms is higher than those provided by traditional dependable embedded hardware platforms that perform more conventional safety tasks. This performance is, instead, usually achieved by HPEC platforms. In the following section, we explain the safety challenges posed by their use.

1.2 Safety Implications of High-Performance Embedded Computing Platforms

The efficient deployment of ML can only be realized by HPEC platforms that are commonly comprised of a multicore Central Processing Unit (CPU) and accelerators such as GPUs. GPUs and Field-Programmable Gate Arrays (FPGAs) have become

broadly used devices for accelerating DNN applications [20], [21], boosted by the computational power and the re-programmability associated to these platforms [22]. However, the inherent complexity of these platforms makes the certification of safety-related systems involving ML on HPEC platforms an open research challenge.

Common Cause Failures (CCFs) at the hardware level do not differ between conventional and HPEC platforms' hardware [23]. However, the high volume of data movements and the compute-intense arithmetic operations involved in the execution of ML algorithms can produce an increment in the Probability of Failure on Demand (PFD) and Probability of Failure per Hour (PFH) of HPEC platforms in contrast with conventional hardware [23]. Then, they have to be minimized or reduced to failure rates that guarantee the absence of unacceptable risks according to the Safety Integrity Level (SIL) of the target application.

Among those errors that can jeopardize the reliability of HPEC platforms (i.e., environmental perturbations, and voltage or temperature fluctuations [24]), soft errors require special attention in platforms integrating GPUs and FPGAs in particular [25]–[28]. These radiation-induced errors produce single event effects caused by high-energy particles striking the electronic device. As a consequence, particle strikes can result in a single event upset, changing the current state of a transistor (bit-flip in case of memory), or they can induce a voltage and current spike (single event transient).

These errors become a concern since HPEC platforms are scaling down, reducing the dimension with a corresponding increment in density. Hence, the soft-error rate drastically increases in these silicon devices, becoming a challenge to be overcome in safety-critical systems [29]. Especially in embedded components such as GPUs, whose memory hierarchy and high levels of parallelism can quickly spread a soft error affecting a single memory bit to multiple locations and lead to catastrophic consequences [30]. Moreover, parallel architectures entail higher risks of timing errors (E.g., race conditions or deadlocks) than conventional single-core implementations traditionally employed to implement safety tasks [23]. Hence, increasing the chances of experiencing an error due to a particle strike.

Besides, the safe deployment and execution of safety-related systems on HPEC devices entails guaranteing the safe behavior of multiple components (e.g., core, private cache, interconnect, shared cache, shared memory, memory management unit) and built-in mechanisms (e.g., cache coherency) [31]. However, traditional solutions such as Software-Based Self Tests (SBSTs) [32] and implementation-specific tests [33] require fine-grained knowledge of the components design and implementation, which are often barely documented by the platform manufacturer.

Nevertheless, diagnosis of all components involved in executing safety-related tasks remains a must, even if the functional safety standards do not establish common practice solutions.

1.3 Objectives

As introduced in the previous section, the topic of autonomous systems safety is extensive, with many open challenges. This Thesis focuses on the challenges of safe AI system execution on HPEC platforms. This has motivated the goal of this Thesis: *'the safe deployment of safety-related systems involving the use of AI on HPEC platforms'*. More specifically, we focus on adopting software diagnostics in ML-based implementations over HPEC platforms for accomplishing the requirements imposed by functional safety standards such as IEC 61508 or ISO 26262. For this purpose, this Thesis formulates the following objectives:

O1 *Adopt functional safety practices in ML code subsets*. This objective intends to guarantee the absence of unacceptable risks caused by the malfunctioning of safety-related systems involving ML, initially focusing on code subsets or low-level libraries employed by those systems. We aim to mitigate and avoid systematic faults at design and control random faults at runtime.

O2 *Foster performance improvement of ML solutions while preserving safety.* This Thesis seeks to achieve a compromise between safety assurance and performance in the deployment of ML. Parallelization can jeopardize the functional safety compliance of safety-related systems. We intend to analyze the achievable performance degree without compromising the safety assurance.

O3 *Implement a 'safe ML' solution prototype in a HPEC platform.* We attempt to verify that the methodology followed in the Thesis paves the way towards safely deploying ML applications. We will adapt a representative ML application in a safety-related domain integrating the solutions carried out during this Thesis.

1.4 Contributions

This section summarizes the contributions of this Thesis. For that, we decompose the contributions into three subsections that collect our three main research outcomes. For an easier understanding, we depict in Fig. 1.1 the main contributions presented in each chapter of this document, as well as the objectives addressed by each chapter.

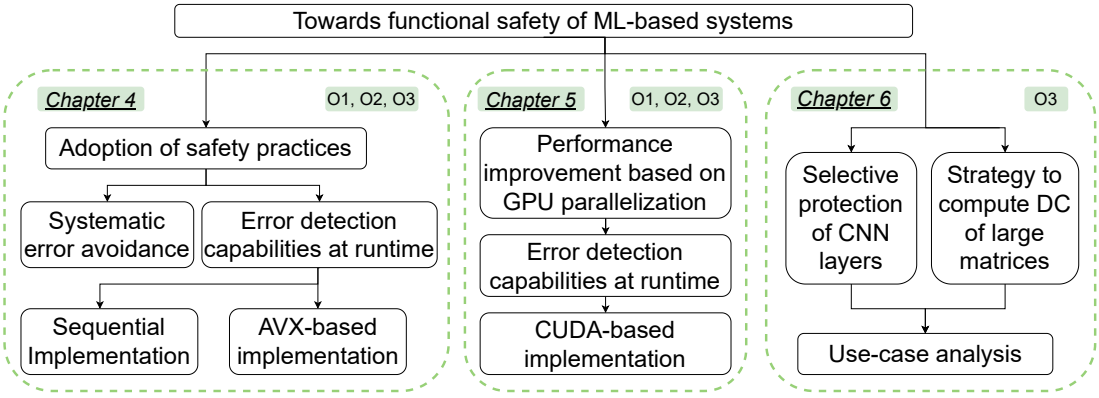


Fig. 1.1: Logical organization of Thesis's contributions

1.4.1 Adaptation of Sequential and Vectorization Based ML Libraries to Accomplishing Functional Safety Standards

The first contribution of this Thesis focuses on adopting safety practices recommended by traditional functional safety standards (objective O1), such as IEC 61508[1] [1], tackling smaller pieces of an entire ML library. As the challenges to be met in order to overcome the requirements imposed by these standards can be currently unfeasible, we limit the scope of this contribution towards accomplishing the software requirements imposed by the previously cited standard (IEC 61508-3). In that way, we base on the core component of the CNN, the MMM.

Complex embedded platforms involve additional challenges, as explained in Section 1.2. That has motivated us to focus on single-core implementations based on sequential code in our first approximation, to achieve the highest adherence to safety standards. However, these implementations can not always provide the performance required by ML algorithms. To overcome this, as a second step, we rely on vectorization to improve performance without leaving safety aside (objective O2). For the sequential code, we adopt safety practices to avoid systematic errors at design time. Additionally, we provide error detection capabilities through combining safety architectural patterns and the use of diagnostic techniques[2] to generate an Execution Signature (ES). These ESs allow checking both the correct order of execution and provide an indirect diagnostics of the components involved in the MMM. As a result, we obtain a catalog of diagnostics with varying levels of DC and performance impact applicable across different architectural patterns. Additionally, we include these 'safe MMM' libraries in YOLO-v3 and Tiny YOLO-v3, measuring the performance

[1]We mainly reference this standard since it is the reference in many domain-specific functional safety standards, such as ISO 26262.

[2]During this Thesis we use indistinctly the terms checksums, diagnostics or diagnostic techniques since we employ the checksums as diagnostic techniques

impact incurred by the adoption of our catalog of diagnostic techniques. This work is presented in Chapter 4.

1.4.2 Fostering Performance Improvement While Preserving Safety: GPU-based Implementations

For the sake of performance and seeking the completeness of objective O2, we focus on HPEC platforms integrating embedded GPUs. These highly parallel platforms provide the required performance to handle the huge amount of data involved in the deployment of CNN applications. However, their use implies additional safety challenges. Among them, we can list that they usually rely on high-performance libraries that are commonly closed-source, the difficulty of guaranteeing determinism in their execution (the internal behavior of these platforms is highly complex and poorly documented), or the intrinsic level of parallelism that these platforms exploit. In the second contribution of this Thesis, we adapt the previous catalog of diagnostic techniques to GPU platforms and include them on a widely used high-performance MMM library, CUTLASS, while continuing to apply the safe architectural patterns proposed in the previous contribution (objectives O1 and O2). We address part of the design and implementation of a safe ML solution on HPEC platforms (objective O3) by including our GPU-based 'safe ML' library in YOLO-v3 and Tiny YOLO-v3 and evaluate the performance impact associated with the inclusion of our GPU-specific diagnostics catalog. This contribution is described in Chapter 5.

1.4.3 Metholodology to Selectively Protect CNNs

Safety-related systems involving CNN commonly operate in real-time environments, which impose stringent timing constraints. In those cases, the performance impact associated with protecting all layers through adopting the previously defined diagnostic techniques in the CUTLASS library may not be affordable. We propose a three-stage methodology to determine which CNN's layers to protect selectively according to parameters such as performance impact and achievable DC. For its evaluation, we apply the presented methodology in a safety-related object detection task based on a widely used object detector (Tiny YOLO-v3) in the automotive domain as a representative use case application. This contribution is directly related to fulfilling objective O3. For the completeness of this contribution, we carried out two complementary tasks: i) characterizing the most error-prone layer of Tiny YOLO-v3 through a fault injection campaign and ii) designing and implementing a strategy to compute the achievable DC. We detail this contribution in Chapter 6.

1.5 Thesis Organization

After introducing the main challenges and defining the objectives of this Thesis, we structure the rest of this Thesis as follows:

- Chapter 2 presents the background on functional safety, CNNs, and HPEC platforms. Additionally, this Chapter details the related work devoted to ML certification according to functional safety standards.
- Chapter 3 describes the methodology followed throughout this Thesis and explains the experimental set-up used to quantify and evaluate the proposals of this Thesis.
- Chapter 4 tackles the safety software deployment of CNNs in sequential and vectorized implementations, focusing on protecting the MMM, CNN's most timing demanding component.
- In chapter 5, we adapt the sequential and vectorized solutions proposed in Chapter 4 to be implemented on massively parallel GPU-based platforms and evaluate them after their adaptation.
- Chapter 6 proposes a three-stage methodology to protect CNN layers according to the target DC and the performance penalty incurred by our protection techniques. We propose a strategy to efficiently determine the achievable DC of large matrices implemented on GPUs and apply the proposed methodology and strategy in an object detection use case.
- In Chapter 7, we finally draw the most relevant conclusions of this Thesis, reviewing the achievement of the initially defined objectives. In addition, we present potential future research lines.

1.6 Publications

In this section, we enumerate the publications carried out in this Thesis:

- **Towards safety compliance of matrix-matrix multiplication for machine learning-based autonomous systems**
  J. Fernández, J. Perez, I. Agirre, I. Allende, F. J. Cazorla, and J. Abella
  International Conference on Reliable Software Technologies (AEiC), 2021
  Extension to Journal of Systems Architecture (JSA), 2021
  DOI: `https://doi.org/10.1016/j.sysarc.2021.102298`

- **On the Safe Deployment of Matrix Multiplication in Massively Parallel Safety-Related Systems**
  J. Fernández, J. Perez-Cerrolaza, I. Agirre, A. J. Calderon, F. J. Cazorla, and J. Abella
  Special Issue: "Computing and Artificial Intelligence" in (Switzerland) Applied Sciences, 2022
  DOI: `https://doi.org/10.3390/app12083779`

- **A Methodology for Selective Protection of Matrix Multiplications: a Diagnostic Coverage and Performance Trade-off for CNNs Executed on GPUs**
  J. Fernández, I. Agirre, J. Perez-Cerrolaza, F. J. Cazorla, and J. Abella
  International Conference on System Reliability and Safety (ICSRS), 2022
  DOI: `https://doi.org/10.1109/ICSRS56243.2022.10067299`

2 Background

This chapter outlines the background concepts of this Thesis and summarizes the research outcomes devoted to the safety certification of systems involving the use of ML components deployed on HPEC platforms.

2.1 Basic concepts

This section introduces the basic concepts that will be used throughout this Thesis. These can be decomposed into three main research fields: i) AI technology, ii) safety-related systems, and iii) widely used diagnostic mechanisms to guarantee data integrity.

2.1.1 Artificial Intelligence

This subsection explains concepts related to AI, deepening from the most general term, AI itself, to more specific ones, such as the CNNs. To facilitate understanding of this section, Fig. 2.1 represents a hierarchy of terms.

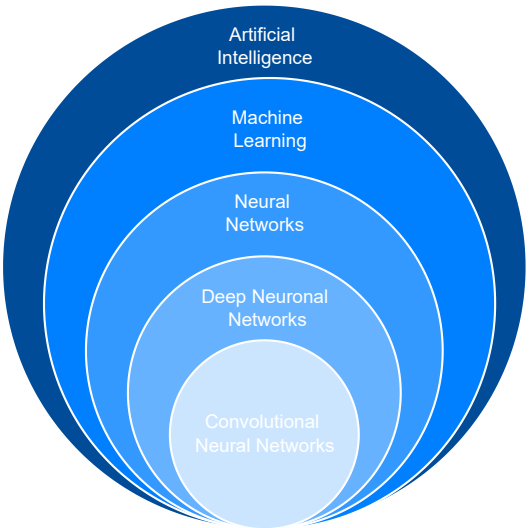


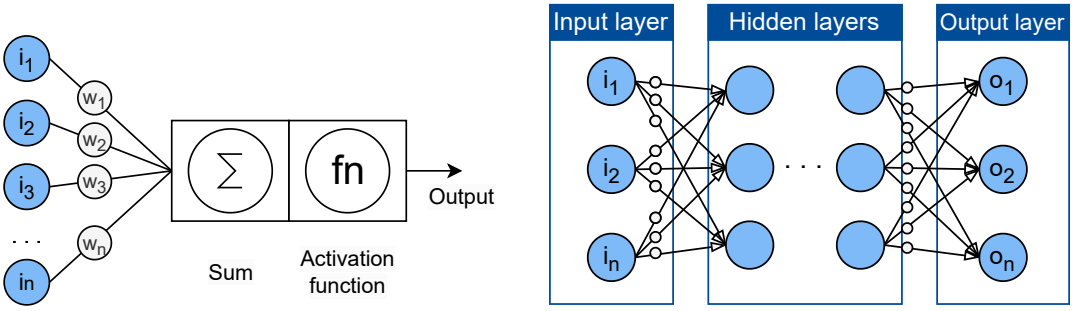
Fig. 2.1: AI terminology hierarchy

AI is defined as *'a system's ability to interpret external data correctly, to learn from such data, and to use that knowledge to achieve specific goals and tasks through flexible adaptation'* [34]. It can be decomposed into different fields such as Planning, ML or Natural Language Processing [35]. Among them, this Thesis focuses on ML, which can be defined as the process by which a machine/computer/system learns things that it can use to perform better in the future [36]. These algorithms can be classified into three groups:

- Supervised learning. These algorithms generate a model based on learning the relationships and dependencies extracted from a set of input features and the target prediction outputs (provided at design), so that the model can extrapolate the appropriate output from new input data. These algorithms can be further divided into two groups regarding whether the output inference values take discrete or continuous values:
  - Classification. The model predicts from input variables discrete output values in the form of a class label or category. I.e., filtering emails as 'spam' or 'not spam'.
  - Regression. The model predicts a continuous quantity from observing the variability of a dependent variable regarding one or a series of other changing variables. I.e., determining the probability of rain.
- Unsupervised learning. The main difference with the above lies in the absence of an expected output to compare with the achieved results. The system learns the structure of the data without employing explicit labels by analyzing the available data to detect correlations and determines relationships. Its goal is to detect patterns in the data that may not be obvious.
- Reinforcement learning. This algorithm, known as the agent, interacts with its environment performing a task and learning with positive and negative rewards feedback from the actions taken to maximize the reward. It exploits the entire set of possible states iteratively by learning from its previous decisions.

This Thesis focuses on supervised learning performing regressions. Specifically, it centers on one of the most common branches, the Neural Networks (NNs). This ML subclass consists of interconnected layers composed of neurons. Neurons or nodes are mathematical functions based on models of biological neurons that constitute the elementary units of the NNs. Fig. 2.2a depicts the architecture of a neuron. As it can be seen, each neuron has n inputs (i_n in the figure) with their associated weights (w_n) that are multiplied and the results are added and passed into the activation function (i.e., a step function). These activation functions produce an output that

can serve as input for the next layer of neurons or as a final output. As illustrated in Fig. 2.2b, the layers can be denoted into three categories regarding the position of the layer in the neural network (a.k.a. network): i) an input layer, ii) a hidden layer (weight layer), and iii) an output layer (decision layer). When the complexity of the NN increases and includes multiple hidden layers, it is denoted as DNN. A DNN is essentially a NN with three or more layers of interconnected nodes that simulate the behavior of the human brain.



(a) Model of an artificial neuron

(b) Types of layers in a Neural Network

Fig. 2.2: Neural Networks

According to the specific problem to be addressed by DNNs, they can be newly subcategorized. In this Thesis, we specifically focus on CNNs, which present outstanding accuracy in performing tasks such as classification, segmentation, and object detection. The CNN computation proceeds across each layer of its particular network configuration. These layered structures are composed of several kinds of layers. The main CNN computation comes from the convolutional ones that extract or detect features from an input (i.e., an image in object detection applications). For that extraction, these layers apply filters (also denoted as weights or kernels) of multiple dimensions on the input data (denoted as Input Feature Map (IFM)[1]) by performing convolution operations. Considering the IFM and filters as matrices, the convolutional layers can be formulated as MMMs. The filter slides over the IFM, multiplying and accumulating products to generate the Output Feature Map (OFM) as Fig. 2.3 depicts, where the OFM of a layer becomes the IFM of the following one. Notice that in Fig. 2.3, the input feature map has been extended by adding zero values (grey boxes) in the borders of the original one (light cyan boxes). This process is known as padding and often increases the accuracy of the predictions.

The pooling and batch normalization layers usually follow the convolutional layers. These layers decrease the computation with different techniques. Pooling layers compute the maximum value of groups of feature maps with specific strides discard-

[1]In the first layer, the IFM can be images, text or sound depending on the specific CNN application.

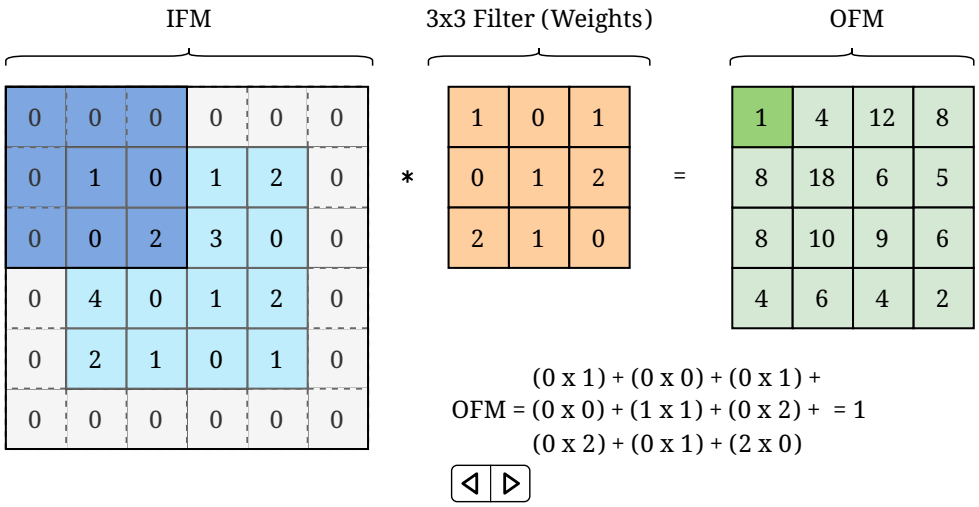


Fig. 2.3: Example of a convolutional operation

ing the rest. Batch normalization layers focus on standardizing the feature maps to avoid overfitting. This standardization is performed by transforming their values to achieve zero mean and unit variance. After these layers, activation ones (i.e., ReLU) are usually applied, which decide whether a neuron (or a specific weight) is activated or not. The primary purpose of these layers is to introduce non-linearity into the output of a neuron. Finally, Fully-Connected (FC) layers classify the inputs.

The implementation of CNNs have two stages. The first is the training phase, in which the CNN iterates with a specific dataset until it reaches the requirements established in the design phase, e.g., a given accuracy. This phase is usually very time consuming since it entails processing vast amounts of data. The second phase is the deployment of the trained CNN into the platform that performs the inference. In this Thesis, we focus on the latter.

2.1.2 Dependability, Safety and Functional Safety

Dependability is, by definition, *'the ability to deliver service that can justifiably be trusted'* [37]. Traditionally, dependability is divided into five attributes: availability, reliability, integrity, maintainability, and *safety*. This Thesis centers on the *safety* attribute or *'absence of catastrophic consequences on the user(s) and the environment'* [37]. Particularly, we focus on functional safety, which is defined as *'part of the overall safety relating to the Equipment Under Control (EUC) and the EUC control system which depends on the correct functioning of the Electrical/Electronic/Programmable Electronic (E/E/PE) safety-related systems, other technology safety-related systems and external risk reduction facilities'*, according to the IEC 61508 [1].

Functional Safety Certification and Standards

IEC 61508 [1] is an international functional safety standard that guides the development process of safety-related systems composed of E/E/PE elements across different industry sectors. IEC 61508 is the reference standard for the definition of many domain-specific standards, such as ISO 26262 [2] for road-vehicles or EN 5012X [38] for the railway, which by inheritance share many common requirements, techniques, and measures for certification. Figure 2.4 provides an overview of the relationships among some of the current safety certification standards.

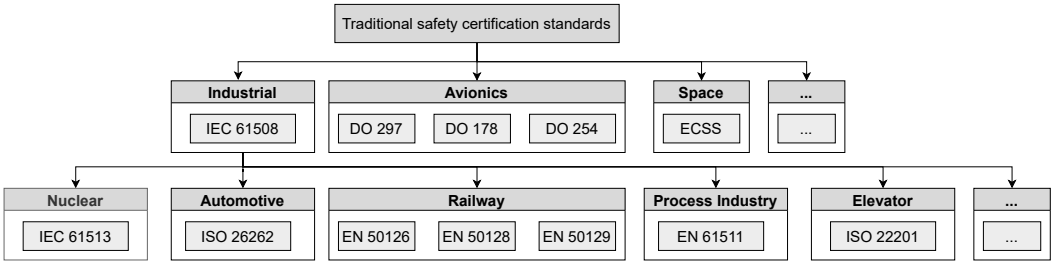


Fig. 2.4: Relation among safety certification standards [39]

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 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) and incorporates a fifth non-hazardous level denoted as Quality Management (QM) level. 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).

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 define 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 their occurrence is sporadic [2].

Regardless of whether faults are systematic or random, they can be classified as *Common Cause Failures (CCFs)* when they meet the following definition: *'the failure is the result of one or more events, causing concurrent failures of two or more separate channels in a multiple channel system, leading to system failure'*. Hence, safety standards recommend the deployment of safety measures to detect faults or errors and control those errors in such a way that they do not lead to failure. Among these measures, diagnostic mechanisms are used to detect errors within the Diagnostic Test Interval (DTI). In addition, this safety standard requires specific Hardware Fault Tolerance (HFT), or the ability of a functional function to perform as required in the presence of faults or errors, according to the SIL of the function. The achievable SIL on a system is partially determined by the adopted fault detection and tolerance.

The assessment of the effectiveness of diagnostic mechanisms is generally evaluated in the form of DC. *"Diagnostic Coverage (DC) denotes the effectiveness of diagnosis techniques to detect dangerous errors, expressed in coverage percentage with respect to all possible dangerous errors"* [31]. DC is classified as low ($60\,\% < DC < 90\,\%$), medium ($90\,\% \leq DC < 99\,\%$) and high ($99\,\% \leq DC$) [1]. As stated in [31], the implementation of software-based DC techniques becomes relevant 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, 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 fault 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 [1]. As an example, a *1oo2* architecture consists of two channels connected in parallel ($M = 2$), and either channel can process the safety function on 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 (HFT=1).

2.1.3 Emerging Standards and Initiatives for AI

A variety of emerging standards relate to AD safety, such as ISO 21448 [40], American National Standards Institute (ANSI)/Underwriters Laboratories (UL) 4600 [41], VDE-AR-E 2148-61 [42], ISO/IEC JTC 1/SC 42 [43], ISO/IEC 5469 [44] and ISO/PAS[2] 8800 [45]. They are shown in Fig. 2.5 specifying the application sector and including some under development. These standards define high-level goals or objectives rather than explicitly prescriptive requirements.

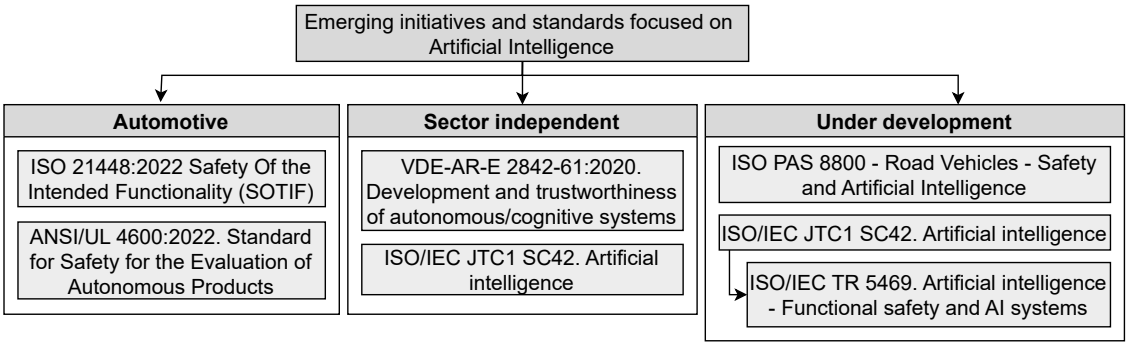


Fig. 2.5: Emerging standards and initiatives to certify ML in safety-related systems

ISO 21448 - Safety Of The Intended Functionality (SOTIF)

SOTIF is an ISO document focusing on avoiding system safety violations in the absence of hardware and software faults for achieving the intended functionality. It is complementary to current functional safety standards, such as IEC 61508 and ISO 26262, which seek to avoid such hardware and software failures. Specifically, it covers critical safety systems operating in open, situationally-aware environments and the limitations caused by the use of complex sensors, probabilistic algorithms, and foreseeable misuse. For example, an unknown scenario in an Advanced Driver-Assistance System (ADAS) in which the algorithm works as trained but not as intended. It may be because designing these systems to address the large spectrum of possible inputs can be tedious and highly problematic.

The proposed lifecycle starts with a Hazard Analysis and Risk Assessment (HARA) followed by defining safety requirements for their mitigation (a.k.a, safety goals). It is crucial to identify the scenarios in which the systems operate and verify the correct operation for each of them. The main idea is to identify and mitigate those events that can jeopardize the intended functionality and, in an iterative way, monitor those situations and generate an ever-expanding catalog of operating scenarios.

[2] A PAS is an intermediate document expected to become a standard within six years. Otherwise, the document is withdrawn.

ANSI/UL 4600 - the Standard for Safety for the Evaluation of Autonomous Products

ANSI/UL 4600, the Standard for Safety for the Evaluation of Autonomous Products, is a safety standard approved by ANSI. It addresses the safety involved in fully autonomous systems to perform as intended according to the system's current state and perception of the operating environment. Additional aspects, such as the reliability of hardware and software necessary for ML technologies, are also handled.

ANSI/UL 4600 defines safety principles and processes based on a claim-based approach. It covers issues such as safety arguments in safety case construction, validation, life cycle concerns, metrics, or conformance assessment, among others. Additionally, it addresses the security attribute without defining details on its achievement and it does not cover any ethical aspect of the product's behavior or decision.

ANSI/UL 4600 is technology neutral. It does not require specific technology to create autonomous systems or a specific design approach, which involves design process flexibility. Instead, it relies on verification and validation via methodology and metrics. For that purpose, ANSI/UL 4600 defines Safety Performance Indicators (SPI) as operational metrics and defends the importance of continuous safety compliance instead of the punctual approval of adherence to functional safety standards. Further, it postulates that it is particularly relevant to learning systems trusting in AI.

VDE-AR-E 2842-61

The German application rule VDE-AR-E 2842-61, 'Development and trustworthiness of autonomous/cognitive systems', defines a general framework for developing trustworthy solutions and autonomous/cognitive systems. This trustworthiness is considered a generic concept that must guarantee functional safety, security, privacy, usability, reliability, and intended functionality (among others). Analogous to current functional safety standards and taking the safety life cycle of ISO 26262 as a reference, it presents a reference life cycle with the logical flow to the involved activities. However, it is not domain specific.

This application rule is under development at the moment of writing this document. It tries to cope with the 'uncertainty' related to AI. It consists of six parts where three of them have already been published. Part one states that in the future, "it might be possible to calculate a fault rate for AI elements, consequently called λ_{AI}". However, for the moment, it defines the Uncertainty Confidence Indicator (UCI) for dealing with these uncertainty-related failures and different UCI levels according to the safety required by the application. This application rule proposes demonstrating the achievement of the specific UCI-level requirements in an assurance

case. Nevertheless, VDE-AR-E 2842-61 does not specify how to deal with these uncertainty-related failures or how to define this assurance case.

ISO/IEC JTC 1/SC 42

ISO/IEC JTC 1/SC 42 - Artificial Intelligence is a joint committee between ISO and IEC international standards with a scope in the area of AI. SC 42's program work has, as its focal point, the standardization of AI. This subcommittee includes foundational AI standards, data standards related to AI, Big Data and Analytics, AI trustworthiness, use cases and applications, AI governance implications, computational AI approaches, and ethical and societal concerns. To the date of writing this document, ISO/IEC JTC 1/SC 42 lists 16 published standards (including two updates) and 25 under development. Some published standards are listed below:

- ISO/IEC TR 24028:2020 Information technology — Artificial Intelligence — Overview of trustworthiness in artificial intelligence. This document surveys issues related to trustworthiness in AI systems.
- ISO/IEC TR 24029-1:2021 Artificial Intelligence — Assessment of the robustness of neural networks — Part 1: Overview, which collects existing methods to evaluate the robustness of neural networks.
- ISO/IEC TR 24030:2021 Information technology — Artificial Intelligence — Use cases, which collects use cases of AI applications in several domains.

Although they offer insight into possible ways to overcome current problems, none of them provides specific solutions for using ML systems. Improvements are expected with ISO/IEC 5469 Artificial intelligence — Functional safety and AI systems [44], under development.

ISO/IEC TR 5469 - Artificial intelligence – Functional safety and AI systems

ISO/IEC TR 5469 [44] standard aims to cover the application of AI-based solutions on safety-critical systems by identifying properties, risk factors of safety, available methods, and potential constraints towards the appropriate adoption of AI approaches in safety functions. The standard is not associated to any application domain. At the time of writing, this standard is still at a development phase and the information on this section is based on early drafts.

This standard is of particular interest for AI-based systems development, as it covers different aspects of AI safety functions. For instance, it defines a high-level lifecycle that combines the V-model and ML lifecycle activities, identifies the properties to be considered, and evaluates the potential compliance of AI-based solutions with existing functional safety standards.

On the platform side, the standard identifies the technological elements required for ML model creation and execution and differentiates those that traditional functional safety techniques can cover from those that require further considerations. It also mentions that GPU-based systems may have special failure modes to be addressed and some architectural considerations are proposed (like the use of supervisors, redundancy and diversity and detection mechanisms).

ISO/PAS 8800 - Road Vehicles - Safety and Artificial Intelligence

This document sets the definition of safety-related properties and risk factors that impact the insufficient performance and malfunctioning behavior of AI for road vehicles. It sets a framework that addresses all development phases and the life cycle of IA components. This framework takes into consideration the derivation of suitable safety requirements on the function and factors related to data quality and completeness. It provides architectural measures for the control and mitigation of failures and defines tools used to support AI as well as verification and validation techniques. Additionally, the evidence required to support an assurance argument for the overall safety of the system is described. Their objectives are the following:

- Define suitable safety principles, methods and evidence satisfying objectives with ISO 26262 and ISO 21448.
- Harmonize concepts described in ISO/TR 4804 [15] and ISO 21448 Annexes's.
- Rely on generic guidance from ISO/IEC TR 5469.

At the time of writing, this standard is still at a development phase and information is based on early drafts.

2.1.4 Checksum Algorithms

Since CNNs involve managing large amounts of data, it is essential to ensure their integrity. This section summarizes some widely used diagnostic mechanisms for detecting errors in data transmission. We focus on those based on arithmetic operations to generate a Frame Check Sequence (FCS) to ensure network data integrity. It is worth mentioning that besides those based on sums of data chunks are known as checksums, from now on, we will generalize under the term checksums those based on other arithmetic operations, such as divisions.

Exclusive OR

Exclusive OR (XOR) or exclusive disjunction checksum is based on the logical XOR operation, denoted with '$\oplus$' symbol. The truth table for the XOR operation is depicted in Fig. 2.6. It sets out the functional output values of the XOR operation ($X_0 \oplus X_1$) from the possible combinations of two binary inputs (X_0 and X_1). The logic is as follows: if both input values are equal (1,1 or 0,0), the output is false or zero; otherwise, if only one input value is one (1,0 or 0,1), then the output is one or true.

X_1 \ X_0	0	1
0	0	1
1	1	0

(a) XOR's Karnaug table

X_0	X_1	$X_0 \oplus X_0$
0	0	0
0	1	1
1	0	1
1	1	0

(b) XOR's truth table

Fig. 2.6: Exclusive OR (XOR)

Although the data to be protected may be higher than two data words, for a better understanding, we depict in Fig. 2.7 the protection of two ($Data_A$ and $Data_B$). Data blocks are protected with XOR checksum by applying the logic XOR operation in parallel across each bit position of those data blocks. That is, by XORing those blocks of data together ($A \oplus B$). Notice that each of the checksum bits values denotes the parity computation of all blocks at the corresponding bit position. So, for example, the bit position 30 of the checksum represents the parity of bit 30 of all data blocks protected.

Data A	$\oplus$	1111	1111	0000	0000	0011	1010	1100	0000
Data B		1010	1011	0001	0101	1111	1111	0111	0000
A $\oplus$ B		0101	0100	0001	0101	1100	0101	1011	0000

Fig. 2.7: Example of performing a XOR checksum

The order in which these blocks are XORed does not influence either the final result or the error detection. In other words, the effectiveness of the XOR checksum is data-independent and, additionally, order-independent. This checksum detects all single-bit errors in data blocks, failing to detect errors aligning in the same bit position of an even number of data blocks. In addition, this checksum detects all odd errors and any combination of bit errors that result in an odd number of errors in at least one-bit position.

Two's complement checksum

The two's complement checksum consists of adding the data blocks to be protected and applying the two's complement of the resulting binary number, that is, inverting the sum and adding one to the Least Significant Bit (LSB) of the given result. Carry bit from the sum of the Most Significant Bits (MSBs) is discarded. Fig. 2.8 depicts an example of applying the two's complement checksum to protect two 32-bit data blocks (Data_A and Data_B). Note that inversion is represented by the symbol '~'.

Data A		1111	1111	0000	0000	0011	1010	1100	0000
Data B	+	1010	1011	0001	0101	1111	1111	0111	0000
① A + B		~~1~~ 1010	1010	0001	0110	0011	1010	0011	0000
② ~(A + B)		0101	0101	1110	1001	1100	0101	1100	1111
③ To add 1	+								1
		0101	0101	1110	1001	1100	0101	1101	0000

Fig. 2.8: Example of performing two's complement checksum

The final two's complement checksum value is not affected by the order in which the data blocks are processed. Therefore, it is a data-dependent checksum but not order-dependent. Although it detects all one-bit errors in the data blocks, there are significant sources of even numbers of bit errors that remain undetected:

S1 An even number of errors affects several blocks at the same bit positions without producing an additional carry-generating bit. The overall number of inversions from zeros to ones has to be the same as from ones to zeros. Otherwise, the carry bit would modify the final checksum value, and the error would be detected.

S2 An even number of errors impacts the MSBs positions of any two data blocks, independently of their value. Two's complement checksum disregards the MSB's carry-out. Therefore, this type of error is undetected.

S3 A third source of undetected errors is a non-carry generating bit being inverted in the data word and the bit in the corresponding bit position in the checksum also being inverted [46]. We have not contemplated the simultaneous occurrence of these two errors in the experiments conducted during this Thesis.

The effectiveness of detecting two errors of two's complement checksum is higher than the XOR checksum. This improvement is because two's checksum primary source of undetected errors are those in which two bits with different values (0 and 1 or 1 and 0) of two data blocks are inverted. In the case of XOR checksum, it neither detects those errors in which the values of the bits do not differ.

One's complement checksum

One's complement checksum consists of summing data blocks and applying the binary one's complement to the resulting sum, which is obtained by inverting all the bits from 0s to 1s and, vice-versa, 1s to 0s. One's complement checksum does not disregard the carry bit of the MSB of the sum, which entails re-summing the carry bit back into the LSB. Fig. 2.9 depicts an example of performing one's complement checksum to two 32-bit data blocks (Data_A and Data_B).

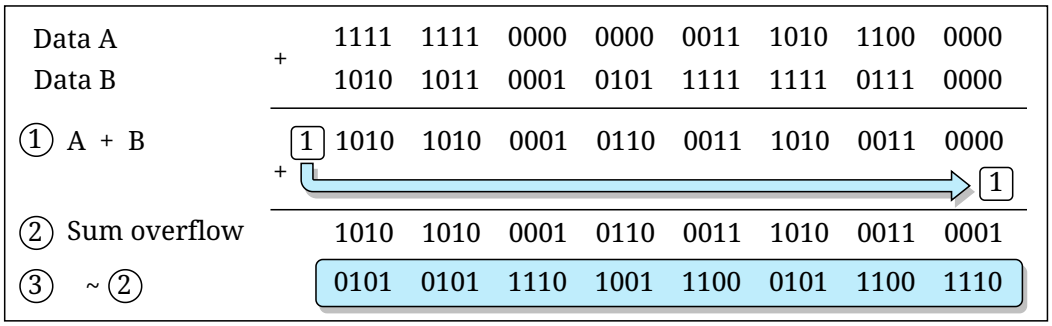


Fig. 2.9: Example of how to perform one's complement checksum

The effectiveness of one's complement checksum is not affected by the order in which the data blocks are processed. This checksum shares the S1 error source of the two's complement checksum. However, preserving the carry bit of the data block's MSBs and adding it back to the resulting sum overcomes the S2 source of error and, therefore, one's complement has enhanced effectiveness.

Cyclic Redundant Code

Cyclic Redundancy Check (CRC) is an error detection code based on a division instead of sums of values. However, it aims to ensure data integrity independently of the mathematical operations involved. The data to be protected is mathematically considered as a bit string whose bits are the binary coefficients of a polynomial, the most significant being the coefficients of a higher degree. This code disregards the quotient of the division and treats the remainder of the operation as the FCS.

CRC calculations perform binary arithmetic operations with no carries for additions and no borrows for subtractions. That way, CRC arithmetic division is based on polynomial division on base-2, primarily about performing XOR operations between a chunk of the data block to be protected and the polynomial generator and shifting one bit rightwards after performing each XOR operation. Before performing this division, it is necessary to add as many zeros to the data block to be protected as

the highest degree of the polynomial generator. It is necessary to explain that in division modulo 2, a polynomial is divisible by another if both have the same degree. Suppose the dividend is not divisible by the polynomial. In that case, single bits shift rightwards until it is divisible or until all the bits of the data block are traversed (including the previously appended zeros). Fig. 2.10 represents a CRC computation indicating with a red box in the most significant bit those cases in which the dividend is not divisible by the generator polynomial and, in green, those in which it is.

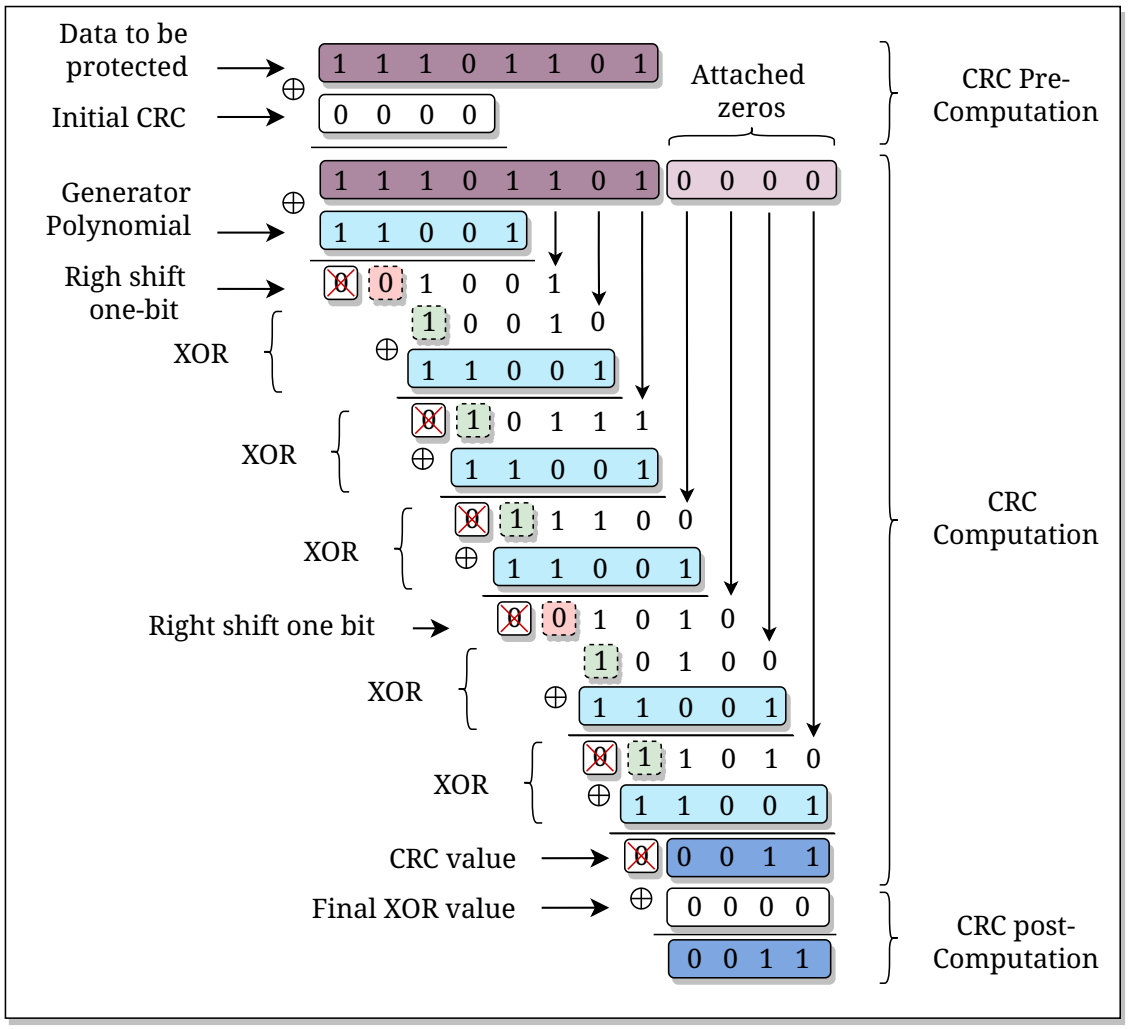


Fig. 2.10: Example of how to perform CRC checksum

Additionally, Fig. 2.10 includes a pre-computation related to the initial value of the CRC register (usually 0x0000 or 0xFFFF in 32-bit CRCs in hexadecimal) and a post-computation consisting in XORing the final CRC value. In this example, we consider that the CRC register has been initialized to zero values, and the final CRC value is XORed with zero values. Therefore, these values do not affect the CRC

computation. Another concept to be aware of, in addition to the initial CRC value and the value to be XORed with the final CRC value, is the concept of reversed Polynomials. However, we consider the election of the two initial values as well the concept of reversed Polynomials out of this Thesis's scope and refer the reader to [47] for further discussion of these concepts.

Fletcher

The Fletcher checksum processes the data blocks to be protected into chunks half the size of the checksum. For example, a 32-bit fletcher checksum iteratively computes 16-bit block sizes until all data chunks are processed. For the calculation, Fletcher decomposes into two data chunks (Sum_A and Sum_B) with half the checksum size, concatenating them once the calculation is complete (being concatenation represented by the symbol '$\|$'). Furthermore, the data is divided into i groups computed from D_0 to D_i. The operations involved in the calculation are as follows (Note that 'MOD' is a value dependent on the Fletcher checksum size):

$$
\begin{aligned}
&\text{Initial Values:} && \text{Sum}_A = \text{Sum}_B = 0 \\
&\text{For increasing } i\text{:} && \begin{cases} \text{Sum}_A = \text{Sum}_A + \text{D}_i \\ \text{Sum}_B = \text{Sum}_B + \text{Sum}_A \\ \text{Sum}_A = \text{Sum}_A \ \% \ \text{MOD} \\ \text{Sum}_B = \text{Sum}_B \ \% \ \text{MOD} \end{cases} \\
&\text{Fletcher checksum :} && \text{Sum}_B \ \| \ \text{Sum}_A
\end{aligned}
$$

Fig. 2.11 depicts an example of a 32-bit data size Fletcher checksum of a 32-bit data word which decomposes into two data chunks, denoting the MSBs D_1 and the LSBs D_0. The Fletcher checksum is also divided into two 16-bit data blocks initialized to zero (the MSBs of Data_B are referred to as Sum_B, and the LSBs as Sum_A). To sum the values across the Fletcher calculation, one can use a standard binary addition, a two's complement addition, or a one's complement addition. For simplicity, Fig. 2.11 represents a standard binary addition.

The accumulation of Sum_B turns the checksum sensitive to the order in which Fletcher processes the data blocks and increases its error detection effectiveness. In fact, the Fletcher checksum exhibits higher effectiveness for detecting errors than the previously defined checksums but incurring in an increment in the computational cost [46]. However, its effectiveness is lower than that of the CRC, which we present next, but it has a lower performance cost.

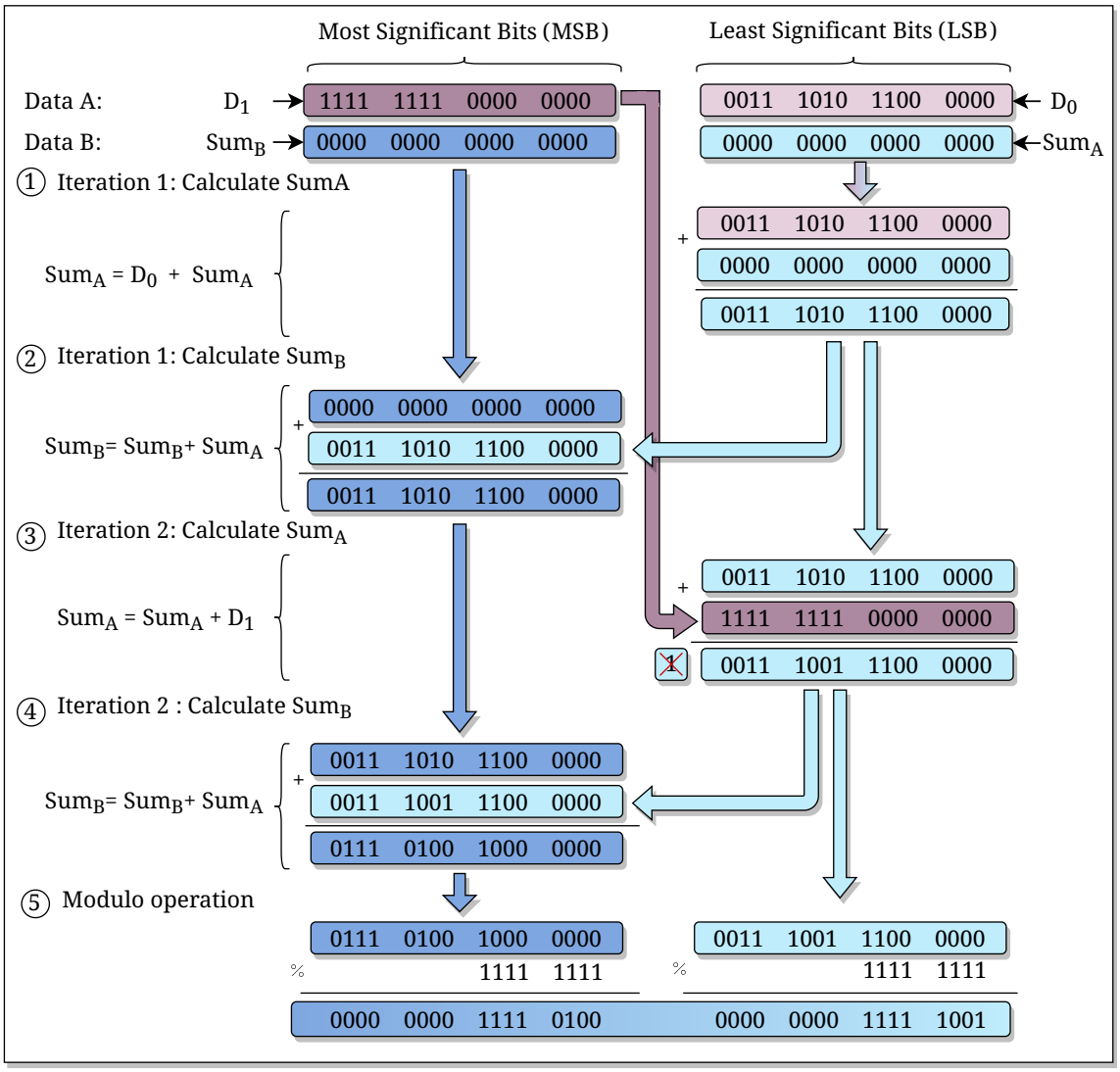


Fig. 2.11: Example of how to perform fletcher checksum

2.2 Related Work

This section compiles the works related to the safety certification of systems employing AI components over HPEC platforms to perform safety functions. First, we collect the research effort devoted to the adhesion to current functional safety standards for systems involving AI components, along with those adopting emerging initiatives designed to accommodate AI. Furthermore, we summarize the main proposals for the detection and mitigation of errors in CNN, in general, and MMM, in particular, implemented on top of HPEC platforms.

2.2.1 Emergent Initiatives to Address ML Certification and Traditional Safety Standards Adaptations

Initial research on the alignment with emerging standards, including ANSI/UL 4600, SOTIF, and VDE-AR-E 2842-61, has already been conducted. According to ANSI/UL 4600, the authors in [48] evaluate their adoption's viability and explore their application in an auditable safety case in the aerospace domain. They conclude that, with some modifications, ANSI/UL 4600 may be employed for safety assurance evidence in this domain. They also postulate that it allows overcoming challenges in autonomous and ML component approval. In a study on SOTIF, the authors in [49] argue that this standard may result in a mandatory requirement for DNNs oversight through supervisors, being the first paper to frame two supervisors within the SOTIF process. Complementarily, the authors in [50] focus on the alignment of high-level automation driving functionalities with SOTIF, employing a safety argument structure to ensure safety. Regarding VDE-AR-E 2842-61, the lifecycle of this application rule became the foundation of [51]. The authors identify the necessity of a standardized process model for AI components and provide an AI-blueprint for DNN. Similarly, several aspects of VDE-AR-E 2842-61 have influenced the design of a data-driven engineering process for applying ML in the industry [52] (e.g., the hierarchical approach or the management of data sets and their quality).

However, these standards are still under development (IEC/ISO 5469 or VDE-AR-E 2842-61) or in their incipient or early stage, as is the case of ANSI/UL 4600, which currently defines UCI to deal with uncertainty-related failures without defining how to deal with them [53].

In addition to the emerging standards, there are plenty of works devoted to the safety certifiability of ML-based solutions in the current literature [14], [19], [54]–[57]. There is a research line focused on identifying and analyzing the main gaps for the adoption of ML components in safety-related system development processes, according to the requirements of functional safety standards such as ISO 26262 or IEC 61508 [19], [55], [56]. In this research line, Falcini and Lami [54] give an initial insight into the applicability of current automotive standards to artificial intelligence systems focusing on software development. They postulate that, according to the standards, learning algorithm development is a partially addressable issue, although data-driven development remains the main challenge. The work in [56], which is an extended work of [55], identifies five problems to adhere to the ISO 26262 lifecycle with ML approaches and proposes five recommendations to address them. In the same manner, Hamid et al. [19] 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 [14], 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 motivated our work, where we have defined diagnostic mechanisms to detect errors in the computation of MMMs, the backbone of CNNs.

The research community's enthusiasm for the safety assurance of AI systems has triggered several systematic literature reviews [13], [58] and surveys [59], [60] that address related issues. After identifying the state of the art, [13] considers six main pillars covering the certification: "Robustness, Uncertainty, Explainability, Verification, Safe Reinforcement Learning, and Direct Certification". In [58], the authors identify five main approaches: "performing black-box testing, using safety envelopes, designing fail-safe AI, combining white-box analyses with explainable AI, and establishing a safety assurance process throughout systems' lifecycles". In addition, authors in [61] explore the main challenges related to the inclusion of ML in safety-related systems, providing an analysis of the safety hazards through all the phases of the ML safety lifecycle. Similarly, authors in [60] assess the safety implications of ML use, with particular regard to robustness and explainability.

As mentioned previously, the computational requirements of ML systems require them to be implemented in HPEC platforms that hinder their safety certification. That reason has motivated the research community to focus on the mitigation of faults of ML systems implemented on HPEC platforms, intending to reduce the risk as much as possible down to tolerable safety levels.

2.2.2 Safe ML Deployment on HPEC Platforms

The research community has made significant endeavors in mitigating hardware errors in HPEC platforms. Traditionally, the adopted methods have focused on the replication of hardware components, i.e., Dual Modular Redundancy (DMR) or Triple Modular Redundancy (TMR) [62]. Some processors provide such redundancy, along with diversity, by hardware means with lockstep architectures, such as the Infineon AURIX processor family [63], the ST Microelectronics SPC56XL70 [64], and some Arm-based designs [65]. Other works provide redundancy with lighter-weight approaches based on single-core thread redundancy [66], [67] or multi-core thread redundancy [68]–[70]. Some works even consider only partial redundancy [71], [72]. Software-only mechanisms for execution redundancy have also been widely

studied in the context of CPUs [73]–[78], including mechanisms such as a monitor process to detect errors or leveraging compilers to inject redundancy. This type of solution has also been explored for accelerators such as the Kalray MPPA family and GPUs, either at hardware level [79]–[83] or at software-only level [79], [84]–[86]. Additionally, we have identified a hardware-based approach based on hardening memory cells on FPGA-based implementations [29], [87]. In the same research line, Li et al. [28] propose partial redundancy with a previous 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 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.

Several fault-injection experiments have analyzed the reliability of GPU-based (e.g., work in [88], [89]) and FPGA-based CNN software implementations (e.g., work in [90], [91] analyzes Static Random-Access Memory (SRAM)-based FPGA against soft-errors errors striking their configuration memory and after binary quantization, respectively). These analyses are fundamental to understanding the importance of random-hardware-errors management and understanding the nature of errors and their propagation. Some of them focus on identifying DNN reliability challenges and summarize analytic and mitigation techniques [92], [93]. Other research is based on studying the CNN reliability [25], [26], [89], [92], [94]–[98], yet without providing systematic solutions.

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 them. Some works assess software-based defect detection by means of mutation testing and fault injection in the context of autonomous driving systems [99]. Other works focus on how to test the system with different sets of inputs [100]–[102]. In a more classical strand, model-based software design and testing can be used for verification and validation activities [103].

Device and software implementation-specific techniques (e.g., references [28], [104]) can potentially provide high error-detection rate with a low-performance impact (e.g., detecting 84.5% of errors that lead to misclassification with 0.3% performance impact [104]). However, the analysis, selection, evaluation (e.g., DC estimation), and implementation of techniques become software implementation-

and device-specific (reduced portability). For instance, SBST [32] for specific components generally builds on low-level knowledge of the device under testing (e.g., GPU gate-level implementation) to devise software-only solutions with high coverage, and hence, SBST is neither portable across designs nor usable in COTS GPUs, whose low-level (circuit) design is often unknown (no public information is available).

The mitigation of soft-errors and permanent faults in safety-critical systems implementing ML is of special interest in MMM [27]. MMMs are at the heart of DNN and CNN software solutions; for example, over 90% of a CNN execution time is due to MMM-based convolutions [105] and MMM accounts for 67% of YOLO's execution time [14]. The performance of the MMM in diverse platforms such as GPUs and multi-core processors has also been widely evaluated due to its critical importance [106]–[108].

Concerning MMM implementations, different technical approaches have been proposed for random-hardware-errors management (detection, correction, and mitigation), ranging from device and MMM implementation-specific techniques to generalizable techniques. Generalizable techniques for MMM algorithms, such as Algorithm-Based Error Detection (ABED) and Algorithm-Based Fault Tolerance (ABFT) [105], are algorithm-specific but less dependent on specific GPU architectures. Several research works propose the adoption of these solutions to enhance the reliability both on FPGAs [27], [109] and on GPUs [26], [110]–[112]. 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 behavior of all platform components. ABFT leverages MMM algorithmic knowledge to provide fault-tolerance with the detection and correction of random hardware errors with low-performance impact, as demonstrated by the authors in [111] with an overhead from $4\,\%$ to $8\,\%$, performance impacts of 20% for square matrices or higher than 50% for non-square matrices [105], $13.8\,\%$ when employed with small matrices dimensions [110] and $3\,\%$ with 500×500 square matrices [27]. Moreover, ABED-based techniques can potentially lead to high error-detection rate claims with low-performance impact (e.g., 100% hardware errors with 6–23% performance impact) [105]. However, these techniques need to consider software implementation restrictions, such as considered data type (e.g., fixed-point integers or rounding errors with floating-point numbers [105]) and detect errors (only) in the generated output values.

In all cases, however, those solutions are orthogonal to those studied in this Thesis and need to be carefully applied only whenever needed due to their costs and the specific needs to meet some safety requirements in ISO 26262 (e.g., such as diverse redundancy for the highest integrity levels).

3 Methodology and Experimental Set-up

In this chapter, we describe the methodology followed in this Thesis and the experimental set-up employed in developing our research work.

3.1 Methodology

The fundamental structure followed for the work of this Thesis is depicted in Fig. 3.1. We split the methodology into three stages or phases: i) exploratory, ii) development and iii) analysis phase.

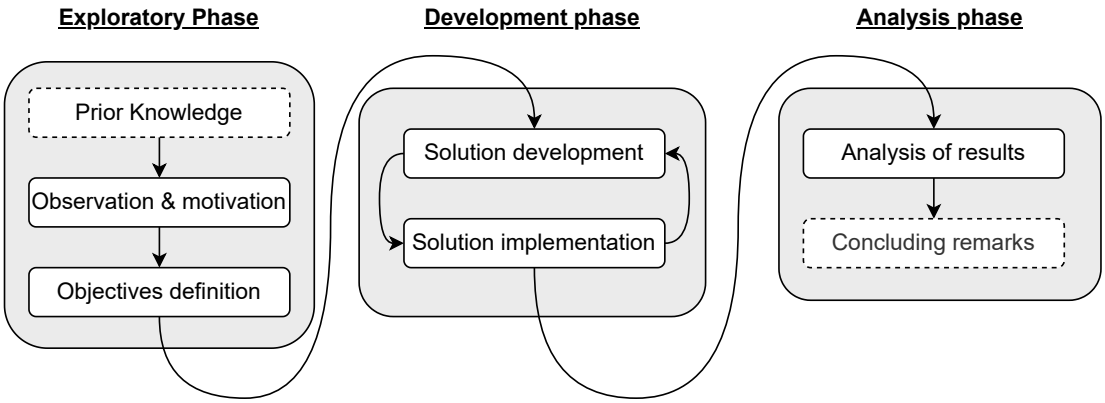


Fig. 3.1: Thesis methodology

In the exploratory phase, we analyze the available literature related to concepts such as AI, functional safety certification, and safe deployment and execution on HPEC platforms. From it, we define the goal of this Thesis: *"the safe deployment of safety-related systems involving the use of AI on HPEC platforms"* by including software diagnostics. Having defined and formulated the goal in the observation and motivation step, we describe and define complementary objectives to achieve the goal of this Thesis.

In the development phase, we deal with these objectives in an iterative development phase in which we initially define a theoretical solution and a development framework. Then, we carry out the implementation in the embedded platforms in the solution implementation step and come back to the solution development to address new challenges arising during the implementation.

Finally, the analysis phase evaluates the resulting solutions concluding with disseminating the research work. These works have been partially validated by the research community and published in journals and conferences [113]–[115].

3.2 Experimental Set-up

This section describes the elements composing the evaluation framework of this Thesis. For an easier understanding, we depict in Fig. 3.2 the elements involved in the development of each chapter: 1) coding guidelines (MISRA C) and static code analysis tools (Polyspace®), 2) automotive datasets (Berkeley DeepDrive), 3) embedded platforms considered, 4) object detectors based on ML (YOLO-v3 and Tiny YOLO-v3) and 5) high-performance MMM (CUTLASS). We follow an incremental strategy from lower to higher performance, motivated by the fact that the safety implications increase when increasing the complexity of the platforms where these systems are deployed.

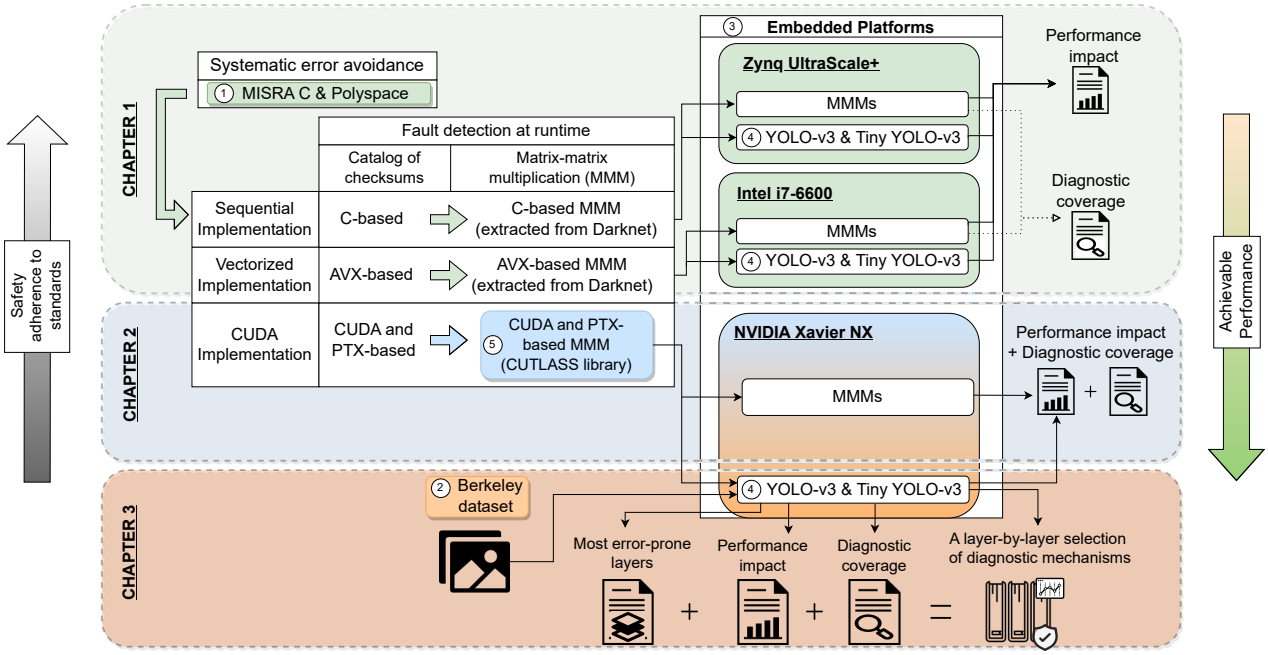


Fig. 3.2: Thesis contributions and elements involved in their development

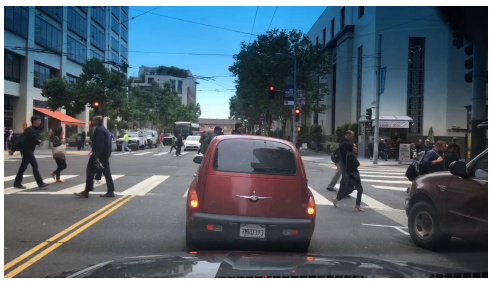
3.2.1 MISRA C and Polyspace®

MISRA C and MISRA C++ are a set of C and C++ coding guidelines focusing on the safe, secure and reliable development of embedded software [116]. The Motor Industry Software Reliability Association (MISRA) developed these guidelines targeting the automotive domain during their design phase. However, their use has spread across multiple domains, such as medical or automation, and functional safety standards, such as ISO 26262 and IEC 61508, enforce their use. Regarding whether the information provided by these guidelines can be unambiguously checked or are open to interpretation, they split into a set of rules and directives, respectively, and label them according to the compulsory compliance as advisory, required, or mandatory.

Decidable rules are those rules that can be statically checked and conclusively verified, e.g., using static commercial tools. Polyspace® [117] is one of them, offering a compliance analysis of C, C++, and Ada code according to MISRA guidelines. It supports the adherence to coding rules in several editions of these standards MISRA-C:2004, MISRA-C:2012, MISRA AC AGC, and MISRA-C++:2008, AUTOSAR C++14, allowing the generation of independent confirmation of compliance with standards.

3.2.2 Berkeley DeepDrive dataset

Berkeley DeepDrive is a public benchmark providing a large-scale driving video dataset for heterogeneous multitask learning based on image recognition [118]. This dataset collects a set of videos from tens of thousands of drivers in a crowdsourced manner, providing 100k forty seconds of driving videos recorded at different times of the day under several weather conditions with a focus on diversity. For this purpose, the videos include diverse scenarios such as residential areas, city streets, and highways recorded with a 30 fps frame rate, 720p resolution, and Global Positioning System (GPS)/Inertial Measurement Unit (IMU). All are publicly available in `https://www.bdd100k.com/`, where videos are decomposed into three categories: i) training, ii) validation, and iii) testing. In Fig. 3.3, we depict five images extracted from the validation category and label them according to their names in this dataset.



(a) 4b2662a8-00000000

(b) 82b5764d-00000000

(c) 96009a72-7eb1ba47

(d) 9e75b2a9-98437b5b

(e) b499ff48-6a0b212e

Fig. 3.3: Set of images extracted from Berkeley DeepDrive dataset

3.2.3 Embedded platforms

In this subsection, we briefly describe the characteristics of the embedded platforms employed during the development of this Thesis. We have followed an incremental strategy from lower to higher platform complexity, focusing initially on single-core implementations towards higher levels of parallelism (e.g., GPUs).

Zynq UltraScale+ MPSoC

In this Thesis we use the Xilinx® UltraScale+ Multi-Processor System-on-Chip (MPSoC) [119], more specifically the EG family, for single-core experiments. The architecture of this platform comprises a high-performance FPGA, MPSoC and a dedicated Arm® Mali™-400 MP GPU. In Fig. 3.4 we depict UltraScale+ architecture.

Zynq UltraScale+ MPSoC provides a Processing System (PS) including a quad-core Arm v8-based Cortex®-A53 (64-bit) Application Processor Unit (APU) combined with a dual-core Arm Cortex®-R5F Real-time Processing Unit (RPU), being a representative safety device that is certified up to SIL 3 according to IEC 61508 and up to ASIL C regarding to ISO 26262 [31]. Additionally, the PS includes a large number of peripherals and dedicated functions to support the functionalities of the processors. The APU includes L1 and L2 cache hierarchy memories of 32KB I/D per core and 1MB, respectively. The RPU also includes a L1 cache memory of 32KB I/D per core and includes, in addition, a Tightly Coupled memory subsystem. Both RPU and APU have access to a 256KB on-chip memory.

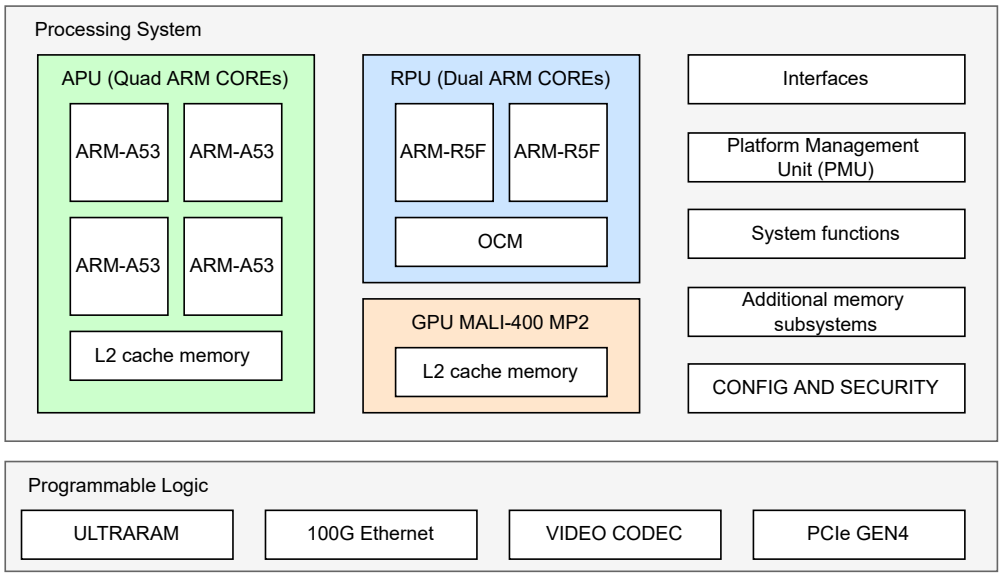


Fig. 3.4: Zynq UltraScale+ Architecture (EG device family)

Intel® core™ i7-6600U Processor

Intel® core™ i7-6600U [120] is a processor from the 6^{th} generation of Intel launched in 2015. Its specifications are summarized in Table 3.1. This processor includes vector capabilities through integrating Single Instruction Multiple Data (SIMD) instructions and vector registers, accelerating the most computing-intensive workloads when performing identical operations on multiple data elements. In particular, this processor incorporates Intel Streaming SIMD Extensions (SSE)4.1, Intel SSE4.2 and Intel Advanced Vector Extensions (AVX)2 Instruction Set Architecture (ISA). Intel offers the possibility of programming these extensions with assembly code and using C intrinsic functions that implement a sequence of compiler instructions directly [121].

Tab. 3.1: Intel core™ i7-6600U specifications [120]

CPU Specifications	
Total cores	2
Total Threads	4
Max Turbo Frequency	3.40 GHz
Intel Turbo Boost Technology 2.0 Frequency	3.40 GHz
Processor Base Frequency	2.60 GHz
Cache	4 MB

NVIDIA® Jetson Xavier Nx

NVIDIA® Jetson Xavier NX [122] is a high-performance platform offering a compact implementation. This platform integrates an embedded NVIDIA® Volta GPU architecture (compute capability version 7.2) with a six-core NVIDIA® Carmel Arm v8.2 (64-bit) heterogeneous multiprocessing CPU architecture. The Carmel cores have 64 KB L1 data cache and 128 KB instruction cache per core and share 6 MB and 4 MB L2 and L3 memory cache, respectively. The Volta GPU offers six Streaming Multiprocessors (SMs) of 64 CUDA cores and 8 Tensor cores each SM, allowing a simultaneous operation of a total of 384 CUDA cores and 48 Tensor cores. In Fig. 3.5, we depict the Nvidia® Jetson Xavier NX architecture.

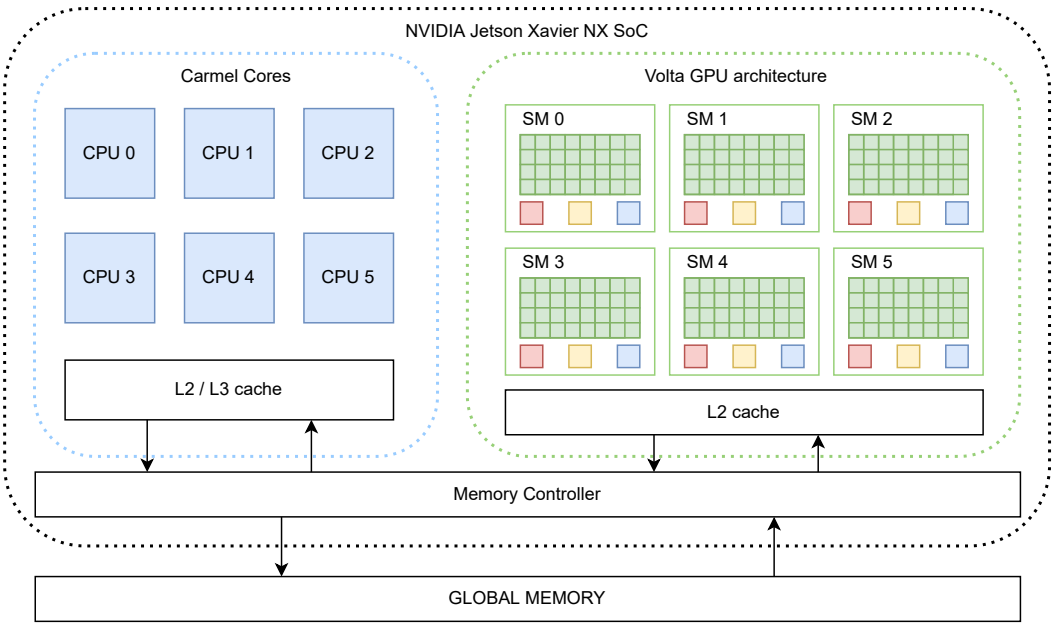


Fig. 3.5: NVIDIA® Jetson Xavier NX architecture

3.2.4 YOLO-v3 and Tiny YOLO-v3

You Only Look Once (YOLO) is a one-stage object detector whose backbone network is Darknet [123], a CNN coded in C and CUDA. We have selected it among the huge variety of current object detection algorithms [124] not only for its high accuracy on perceptual tasks [94], but also because its reliability has been widely studied [19], [26], [94], [95]. There are several versions of this object detector [125]–[127], and all of them 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. This Thesis focuses on Tiny YOLO-v3 [125] and YOLO-v3 [126].

YOLO is a multi-scale object detector that resizes the input image or frame to predefined values and splits it into grids of $S \times S$ cells. These S values vary according to the YOLO configuration and version. The higher the S value, the better the detection of larger objects and vice-versa. YOLO employs multi-scale predictions; that is, the prediction relies on multiple-scale feature maps associated with the above cell. In particular, YOLO-v3 uses the feature extractor Darknet-53, a residual network with 53 convolutional layers that produce feature maps on three scales (13×13, 26×26, and 52×52). In contrast, Tiny YOLO-v3 is the reduced version of YOLO-v3 with a smaller feature extractor that splits into a lower number of convolutional layers (13 nested layers), computing the feature maps on two scales (13×13 and 26×26). As a representative example, we depict its architecture in Fig. 3.6.

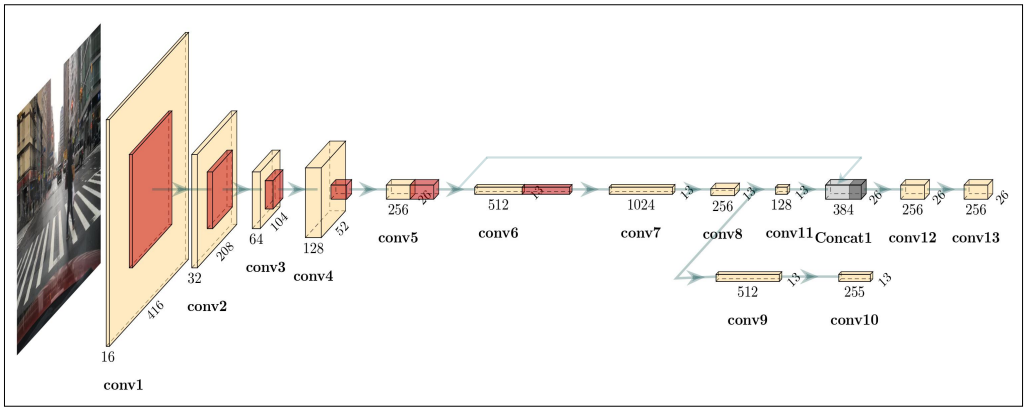


Fig. 3.6: Tiny YOLO-v3

These feature maps contain a list of bounding boxes with the classes recognized at each scale. These bounding boxes extend in four features: confidence score, the probability that the box contains a specific class, center coordinates, and box dimensions. According to Fig. 3.7, the coordinates of the box center are b^x and b^y, and b^h and b^w correspond to the box height and weight dimensions. After post-processing

of the feature maps from the different scales, YOLO obtains the final bounding boxes predictions by applying concepts such as Non-Max Suppression or the intersection over union (IoU) (although this stage is out of the scope of this Thesis).

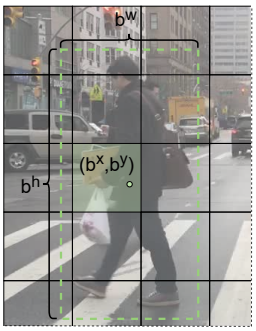


Fig. 3.7: Bounding box and nomenclature employed by YOLO object detector

Darknet offers the user a selection of MMM based on implementation needs [123]: i) General Matrix Multiplication (GEMM) function for the sequential implementation on CPUs, ii) CUBLAS for the GPU implementations and iii) other processor-specific variants such as the use of AVX for Intel processors. Additionally, the software developer can include its own MMM implementation or any other (i.e., CUTLASS).

3.2.5 CUTLASS

CUTLASS is a collection of templates coded in CUDA and C++ that abstract the high-performance matrix-multiplication implementation. This open-source and low-level library decomposes this algebraic operation into software modules using C++ template classes. These modules divide matrix multiplication into thread, warp, block and device levels, as can be seen in Fig. 3.8. This figure highlights the memory transfer in each iteration of the most external loop of the Matrix Multiply-Accumulate (MMA) ($C \mathrel{+}= A \times B$). Additionally, CUTLASS allows tuning through custom data types, tiling sizes and other algorithmic policies.

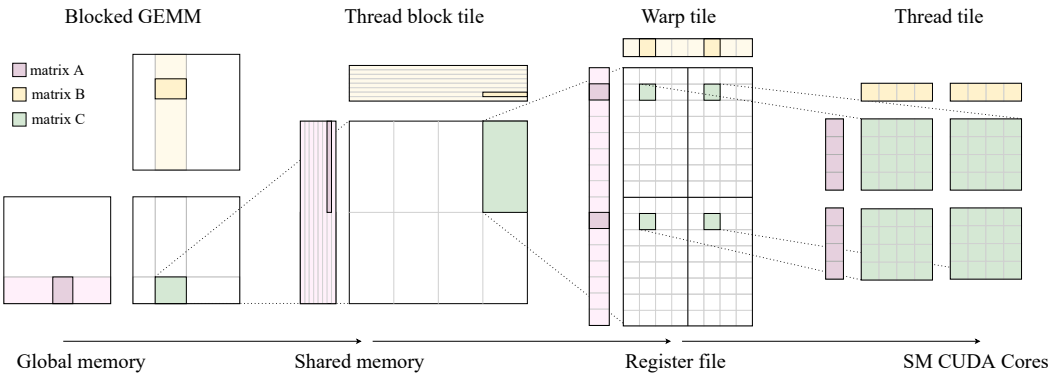


Fig. 3.8: Cutlass GEMM hierarchy [18]

4 Safe Deployment of MMM in Sequential Implementations

The scope of this chapter relates to solutions for mitigating systematic and random errors in MMM software libraries executed on HPEC platforms. Our contribution lies in defining a 'safe MMM' software module. This arithmetic operation is the central computing element of many ML libraries [14], as it takes $67\,\%$ of the CNN execution time according to the results obtained from YOLO-v3 implemented on NVIDIA GPUs [26] and $98.5\,\%$ and $87\,\%$ in single Arm and Intel cores, respectively, according to our experiments. An error in the MMM execution could lead to miss-classification in the object detection tasks. As an illustrative example, the authors in [28], [95] perform a fault[1] 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 study employing the YOLO object detector lead to an unsafe situation [95]. The next chapter depicts the miss-classification in a GPU-based object detection task when a single bit-flip occurs in the CNN weights.

Therefore, in this chapter, we provide solutions to mitigate systematic errors and to detect and control random errors in the MMM execution, using as baseline the non-vectorized GEMM implementation (scalar MMM) and the vectorized MMM implementation (AVX-based MMM) extracted from the Darknet CNN [123]. The former is interesting from a safety perspective, as it is the closest option to current safety practice. However, at the same time, the scalar implementation provides the poorest performance, which is also an important property for ML inference. Trying to find a balance between safety and performance, we first advocate for the AVX-based MMM instead of the CUDA-based MMM. The main reason for this is the safety implications involved with CUDA-based implementations: the closed-source nature of the high-performance MMM libraries, the lack of or limited support for developing safety-critical software through available GPU programming languages [128], and GPU's features, such as dynamic memory allocation [19], among others. We will partially cope with these challenges in the following chapter. This one contributes with the following modifications to the scalar and AVX-based MMMs:

[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.

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 [117]. Likewise, we evaluate Darknet CNN to demonstrate that complying with the specifications demanded by coding guidelines such as 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 one of the Arm R5 cores of a Zynq UltraScale+ MPSoC, and we then adapt them to an Intel i7 processor with native code employing vectorization for the sake of performance (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. As a result, we provide a catalog of mechanisms with varying levels of DC and performance impact, allowing the final user to choose a solution based on system needs.

4. Additionally, we compute the overall performance impact incurred by individually including each catalog diagnostics mechanism in YOLO-v3 and Tiny YOLO-v3 with both implementations.

The rest of this chapter is organized as follows: Sections 4.1 and 4.2 define the proposed adaptations to obtain a 'safe MMM' with configurable DC levels. Specifically, the focal point of Section 4.1 is the avoidance of systematic errors, while Section 4.2 targets the detection of errors. Section 4.3 presents the evaluation results of the proposed modifications. Finally, Section 4.4 draws a summary.

4.1 Systematic error avoidance in the MMM

In this section, we focus on the avoidance of systematic software errors. To that end, we have verified the source code of the scalar 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 [116]. 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 [2] and for SIL 3 and higher according to IEC 61508-3 Table A.3 [1].

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 check that pointers to matrices do not have "NULL" values.

After using the Polyspace tool to check the adherence to MISRA C, we identify 33 violations in the scalar MMM as shown in Fig. 4.1. That is not the case with vectorized code, for which the MISRA C specifies that all the used compiler extensions must be analyzed from a safety point of view and approved or rejected on a case-by-case basis. We observe 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, we explicitly define the required types.

- D4.14: Six violations were caused by not checking the correctness of input parameters. As explained before, we implement 'defensive programming' as a corrective action.

- 12.1: Three violations were due to not explicitly defining the desired precedence of operators within expressions. In this case, we explicitly define the operator precedence.

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

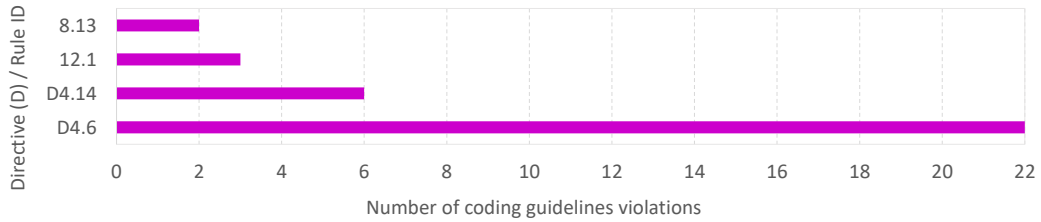


Fig. 4.1: Scalar MMM software MISRA-C:2012 compliance analysis: rules and directives (D) violated according to a Polyspace analysis.

It is worth mentioning that we achieve full MISRA C compliance with the Polyspace tool once we apply the above recommendations.

Additionally, we analyze the complete Darknet CNN, identifying 2,332 violations. In the same manner, these violations do not require a significant effort to accomplish MISRA C. However, the implementation of the respective modifications is beyond the scope of this chapter, as we follow an incremental strategy in which we initially focus on small subsets of code in this chapter (the MMM), rather than the entire CNN. For illustrative purposes, in Fig. 4.2 we depict the five most repeated violations, which account for $64.5\,\%$ of the total.

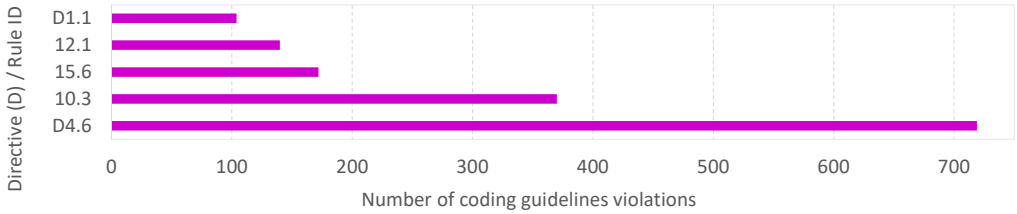


Fig. 4.2: Darknet CNN MISRA-C:2012 compliance analysis: top 5 of rules and directives violated in the Darknet CNN code according to a Polyspace analysis.

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 behavior that affects the outputs, such as casting from integers to floating-points. As suggested by MISRA C, the corrective action is to elaborate a conformance matrix with the procedures to be followed by the developer and to document it to ensure that the code complies with all MISRA C standards.
- 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 in 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.

4.2 Error Detection in the MMM

Providing a safe inference execution environment for the YOLO MMM software module requires the deployment of diagnostics for runtime error detection [129]. Diagnostics techniques implementation could be considered outside the scope of YOLO MMM, exporting such requirements to the system architect and integrator.

However, the implementation of simple generic diagnostics 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 [31], [130]. 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 [31] and also from a system integration perspective [31], [130].

In this section, we propose the use of an ES as a means of diagnosis, which can provide a scalable strategy with varying levels of achievable DC and the associated computational cost. The ES, described in Section 4.2.1, can summarize in a reduced number of bits (e.g., 32 bits) the computed data and program sequence for later comparison in one of the architectural patterns described in Section 4.2.2.

4.2.1 Execution Signatures

Among the state of the art, we have identified XOR, two's and one's complement addition, Fletcher and CRC as potential diagnostic techniques to compute the ESs [46], [131]. These error detection techniques are widely used to assure network message data integrity at different levels of error detection effectiveness [46]. For details about the effectiveness of each of these techniques, we refer the reader to [46] and [131], 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 [132]). The novelty of this chapter 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 scalar and AVX-based implementations) and the diagnosis of all internal platform components that take part in the computations.

The MMM is an algebraic operation usually coded and implemented through nested loops as shown in Algorithm 1. The ES can be calculated by 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 contribution can be extrapolated to MMM involving a different number of loops. In our case, both the scalar 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 the higher is the amount of data and computation summarized in the ES, at the cost of increasing the required computational penalty required 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. In this chapter, instead of defining a specific solution, we explore all these different alternatives, providing a catalog of solutions that can be tailored to the concrete needs of the applications in terms of performance and DC.

The implemented checksums compute the ES of A, B and C matrices (where the duty of the MMM is to compute ($C = A \times B$)) and store these ESs values in three independent variables (one for each matrix). 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 catalog with individual and combined checksums to provide the user with a wide variety of DC and performance penalty alternatives. The catalog can be divided into two groups according to the checksum involved in the MMM:

- Individuals: employ 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 (I) and higher performance impact, and higher DC, in the intermediate loop (M) (e.g., XOR_Fletcher means that a XOR is computed in the internal loop and a Fletcher in the intermediate loop).

We decompose this subsection into two additional ones to better explain how the MMM coding and other aspects such as the compiler instructions influence the implementation of the diagnostics with respect to being the scalar and AVX-based implementation.

Scalar Implementation

We refer the reader to appendix 8.1, where we detail the code employed in scalar implementation (Algorithms 6 to 11). However, in favor of the understandability of the checksum combinations, Algorithm 2 shows an example of the code employed to compute the ES with a XOR_Fletcher checksum combination. 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.

Algorithm 2 Scalar 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
```

Additionally, we show the code of the Fletcher checksum in Algorithm 3. As it can be seen, the Fletcher function receives the union datatype *ui32_to_ui16_t* (Algorithm 4) with the current Fletcher ES and an *uint32_t* datatype with the data to be protected.

Algorithm 3 Scalar Fletcher

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

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 motivated by the nature of the Fletcher checksum, which involves decomposition into two smaller blocks to carry out the ES computation.

Algorithm 4 Union definition

```
1: typedef union ui32_to_ui16 {
2:         uint32_t ui32;
3:         uint16_t  ui16[2u];
4: } ui32_to_ui16_t;
```

Regarding the implementation of the CRC algorithm, we resort to lookup tables for its execution. This method accelerates the protection of MMA operations as follows: 1) we precompute the ES applying the CRC algorithm to all chunk's possible values of a prefixed number of bits (n). These values are stored in a 2^n lookup table, and 2) we access these CRCs values at runtime.

AVX-based Implementation

The protection of the MMM in the AVX-based implementation has certain peculiarities. AVX execution lies on the SIMD paradigm. The maximum data length computed with each instruction depends on the ISA, the size of the registers, and the instruction itself. This length determines the data management performed in the MMM computation and it affects its protection. In this way, we perform the MMM

with AVX instructions going through the rows of A matrix and the columns of B matrix multiplying data chunks, which match the length of the data vector managed by the processor until reaching the last data chunk. For the last data chunk, we check if the length is lower than that managed by the processor and, if it is, we perform the MMM of this chunk of data and its protection sequentially to avoid incorrect accesses to undefined memory locations. In Alg. 5, we depict our AVX-based MMM implementation for an easier understanding. In our case, the ISA of the employed intel core i7-6600 supports AVX2 and carries 256-bit numeric processing capabilities, allowing it to handle operations with eight floating point values of 32 bits. In Alg. 5, the reader can observe that we employ the variable $prev\_end$ to check whether the number of rows of A and columns of B matrices (referred to as $ui32\_n$ variable) matches the data size of the data managed by the compiler and, if not, to compute the MMM of that data chunk sequentially.

Algorithm 5 MMM implementation based on AVX instructions

```
   Auxiliar variables:
   uint32_t prev_end = (ui32_n % 8);
 1: for each column of A matrix do
 2:     External loop statements
 3:     for each column from B matrix do
 4:         Intermediate loop statements
 5:         for group of eight rows from A matrix [a] do          } AVX-based implementation
 6:             Internal loop statements (compute the multiplication)
 7:             [Checksum (I)]
 8:         end for
 9:         if 0 ≠ prev_end then
10:             for each value of the last group do               } Scalar implementation
11:                 Internal loop statements (compute the multiplication)
12:                 [Checksum (I)]
13:             end for
14:         end if
15:         [Checksum (M)]
16:     end for
17:     [Checksum (E)]
18: end for
```

[a] Excepting last group if it is < 8, in that case the MMM and its protection is performed sequentially

We refer the reader to appendix 8.2, where we detail the code employed in AVX-based implementation (Algorithm 12 to 16).

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

4.2.2 Architectural Patterns

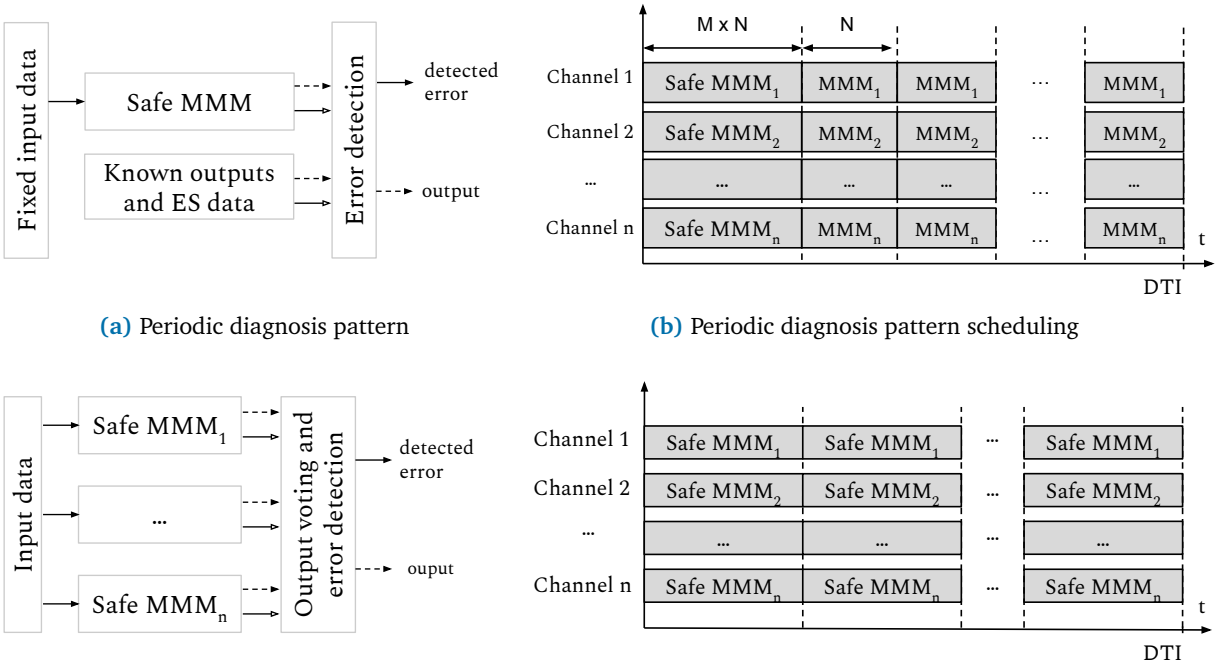
Taking into consideration the architectural patterns and DC techniques for HPEC multi-core devices described in [31], and the safety measures proposed by the safety standards (IEC 61508, ISO 26262) considered in this work, this section defines two basic and common architectural patterns that support safe detection of faults based on previous diagnostic techniques (see Fig. 4.3) with different HFT levels:

Periodic diagnosis with design time fixed data pattern(s)[2]. Applying this pattern the 'safe MMM' executes at least once every DTI (Fig. 4.3b) or Process Safety Time (PST) according to ISO 26262 or IEC 61508, respectively. In the rest of executions the MMM operates without diagnosis, as the reader can observe in Fig. 4.3b. The 'safe MMM' takes predefined reference input data vectors at design time, being correct if it leads to known reference outputs and known ES (see Fig. 4.3a). 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. This pattern enables the periodic diagnosis of device components and built-in techniques, carrying a performance impact against the normal matrix multiplication operation (M). Additionally, it can be used in a single channel architecture (HFT=0) or in redundant architectures (e.g., triplicated architecture with HFT=2). Error detection can be used to detect the erroneous channel prior to the voting process, and application-specific measures can be implemented (e.g., restart erroneous channel, activate safe state...).

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 n 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) (Fig. 4.3c), discarding the replica(s) with discrepancies for each computation cycle (Fig. 4.3d) and/or implementing application-specific measures (e.g., restart erroneous channel). The comparison of just the output values would not generally be sufficient to detect latent errors (e.g., faults in matrix B –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 the previous as the 'safe MMM' is executed in each execution period (τ) by the n replicas. However, the correctness of components and of the output is

[2]It should be noted that MMM in normal operation executes sequentially without diagnostics (implying a run time per run denoted as N in Fig. 4.3b).

diagnosed in every execution period, and it can support fault-tolerance as later explained in Section 4.3.5 (e.g., a 2oo3 architecture could tolerate one discrepancy). In order to further improve DC by the detection of CCFs, the redundant pattern can be complemented with different types of *diversity* [31], such as component diversity (e.g., implementing the MMM in different types of cores of a multi-core platform).



(a) Periodic diagnosis pattern

(b) Periodic diagnosis pattern scheduling

(c) Redundancy pattern

(d) Redundancy pattern scheduling

Fig. 4.3: Safety architectural patterns

4.3 Evaluation

In this section, we evaluate the execution time penalty caused by the inclusion of the different diagnostics from Section 4.2 in the scalar MMM and AVX-based MMM extracted from YOLO, as well as the maximum achievable DC of each ES for the detection of single-bit errors. In addition, we provide a discussion of compliance of our diagnostics catalog to DC ranges established by functional safety standards.

4.3.1 Experimental Set-up

In this chapter, we have implemented the scalar MMM function on one of the Arm R5 lock-step cores of a Zynq UltraScale+ MPSoC device and an AVX-based solution employing vectorization on the Intel core i7-6600U processor. However, the overall approach is platform-independent and can be adapted to diverse platforms.

In the upcoming chapters, we will apply the same approach to platforms that include GPU accelerators. Nevertheless, this methodology could be used for other implementations, such as those based on FPGAs.

The scalar MMM has been compiled employing the Arm v8 gcc 7.3.1 compiler, while AVX-based MMM has been compiled with Microsoft Visual C++ (MSVC) compiler version 14.16, both without optimizations. Despite the fact that compiler optimizations are desirable from a performance point of view, they may bring challenges for safety-related systems. For instance, altering the control flow or the organization of multi-condition branches. In this chapter, we have opted for disabling compiler optimizations in both implementations to avoid side effects. Notwithstanding, we have performed a subset of experiments employing high-level compiler optimizations to explore its influence on the performance impact incurred by the adoption of checksums. We have employed the highest optimization level provided by gcc ($O3$) and MSVC ($O2$) compilers to evaluate the highest possible impact with respect to the unoptimized experiments.

We have decomposed both the DC and performance impact experiments into two groups based on the MMMs dimensions: i) square matrices seeking to evaluate the influence of matrices sizes when we preserve the same relationship between the number of rows and columns and ii) unbalanced matrices focusing on assessing the representativeness of matrices dimensions in performance impact experiments and the variability of the DC when the relationship between rows and columns changes. It should be pointed out that, as we explain above, the chosen matrices dimensions are smaller than those involved in the CNNs executions. The reasons vary according to the experiments. On the one hand, performance impact experiments aim to identify variability in performance impact by modifying the size of the matrices, which do not require large matrices to extract a common tendency. On the other hand, the evaluation of the DC requires executing the MMM with a single fault injection (bit flip) as many times as the number of bit positions in the matrices A and B, which can involve an unaffordable computational cost if matrices are too large.

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 scalar MMM function. The procedure followed in the performance experiments consists of a loop of ten thousand iterations for each MMM of a subsequently defined set of matrices dimensions to obtain a mean time value. Additionally, we disregarded the first ten time measurements to avoid cold-start problems from the caches. We have assessed the performance impact with the matrices dimensions depicted in Table 4.1, dividing them into two groups:

- Square matrices: we have performed the first set of experiments with square matrices (A, B, and C) of dimensions $N \times N$. These experiments have also been performed with high levels of compiler optimization.
- 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 $M \times K$ for A, $K \times N$ for B, and $M \times N$ for C.

Tab. 4.1: Matrices dimensions employed in performance impact experiments.

Square matrices		
Name	**N**	**N**
80 × 80	80	80
160×160	160	160
320×320	320	320

Unbalanced matrices			
Name	**M**	**N**	**K**
L91	18	230400	64

To conclude these experiments, we have evaluated the performance impact incurred by the adoption of our catalog of diagnostics in the MMM employed in YOLO-v3 and its Tiny version, employing both the scalar and AVX-based implementations. For this purpose, we have employed ten thousand images extracted from the Berkeley DeepDrive dataset as input to obtain the mean time value (disregarding the initial ten timing measurements).

Regarding DC experiments, the computational cost required to perform this exhaustive fault-injection campaign at bit level has lead us to the choice of smaller matrices dimensions than those for the performance experiments as shown in Table 4.2:

- Square matrices: The dimensions of the matrices we have employed for the performance assessment of DC are depicted in Table 4.2.
- Unbalanced matrices: we have evaluated smaller matrices keeping the relationship between rows and columns proportional to some of the scalar MMM implementations of Darknet. We denote these matrices $L1$, $L2$ and $L3$, which have been chosen as a representative example for the evaluation 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. All these experiments have been analysed in both, scalar and AVX-based MMM. For completeness, we have also evaluated the DC of matrices extracted from Darknet employing the scalar MMM. In Section 4.3.1 we have chosen the $L59$ layer due to the reduced size of its matrices, when compared to other Darknet layers.

Tab. 4.2: Matrices dimensions employed in DC experiments.

Square matrices		
Name	**N**	**N**
20×20	20	20
40×40	40	40
80×80	80	80

Unbalanced matrices			
Name	**M**	**N**	**K**
L1	32	29	144
L2	8	900	8
L3	15	225	48
L59	18	900	1024

4.3.2 Performance Impact

First of all, we define the performance impact as a ratio (n) in terms of execution time as shown in Eq. (4.1) (where X and Y vary in function of the experiments):

$$n = \frac{Execution\ time_X}{Execution\ time_Y} \quad \text{[133] (4.1)}$$

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 scalar MMM (Y). We have observed that these adaptations of the original MMM do not cause a relevant overhead in the execution time (below $1\,\%$, which may be influenced by the standard deviation of the experiments themselves). 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 of 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. (4.1), in this experiment, X refers to the scalar MMM accomplishing MISRA C after the optimizations, and Y refers to the original scalar MMM.

For evaluating the performance slowdown incurred by the inclusion of the checksums, we first obtain the baselines (Y) for the optimized scalar and AVX-based MMM. To this end, we have measured the execution time incurred by each of the aforementioned matrices dimensions defined in Section 4.3.1 in both their scalar (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 times faster than the scalar MMM for the different matrices 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. (4.1), X relates to the MMM after the implementation of the diagnostics catalog. The results are depicted in Fig. 4.4 and 4.5 respectively (the complete catalog can be identified in the y-axis of both figures).

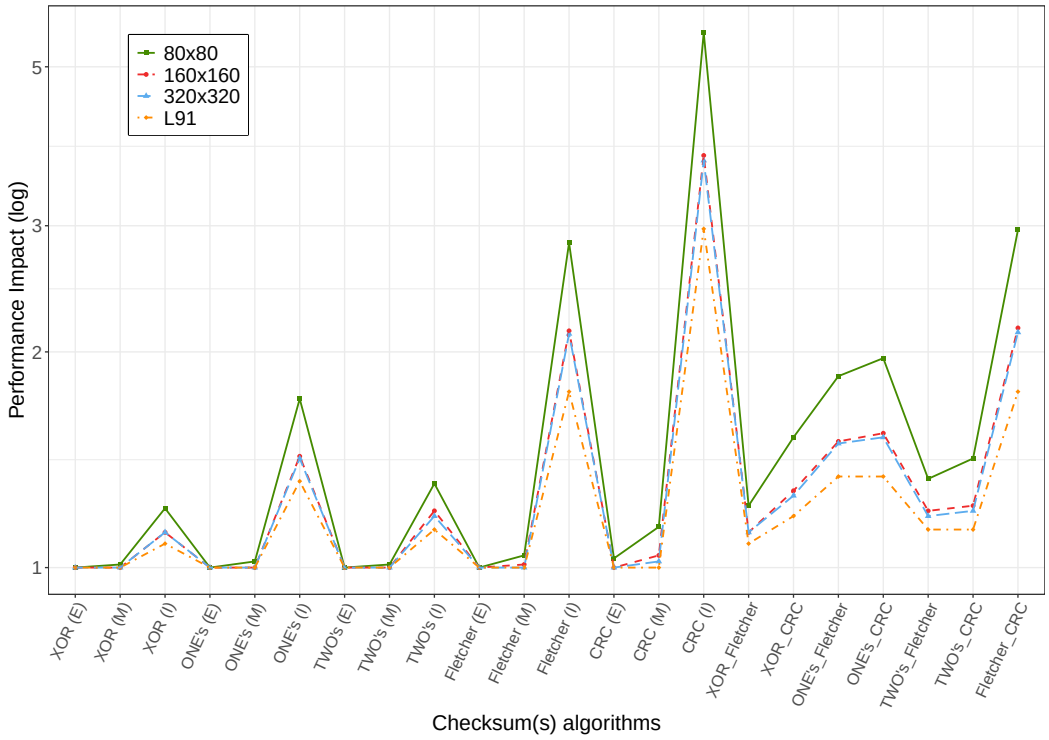


Fig. 4.4: Scalar MMM: performance impact caused by the inclusion of a catalog of checksum algorithms disabling compiler optimizations.

The results of Fig. 4.4 are represented as follows: i) 80×80 is depicted with a green solid line and a square marker, ii) 160×160 with a red dashed line and round marker, iii) 320×320 with a blue dashed line and a triangular marker and iv) $L91$ with a yellow dot-and-dash line and a rhomboid marker. As shown in Fig. 4.4, the performance impact decreases with increasing matrices size, approaching asymptotically specific values for each ES. This reduction is expected since larger matrices require an increasing number of memory accesses, which decreases the performance impact incurred by the ES computation in relative terms. Furthermore, in this scalar implementation, the individual experiments confirm the increase in performance impact from the straightforward (XOR, two's and one's complement) to the more intricate algorithms (Fletcher and CRC) as well as a smaller 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 matrices sizes, the impact of the intermediate checksum (M) is comparatively lower than the impact of the internal checksum (I) in relative terms.

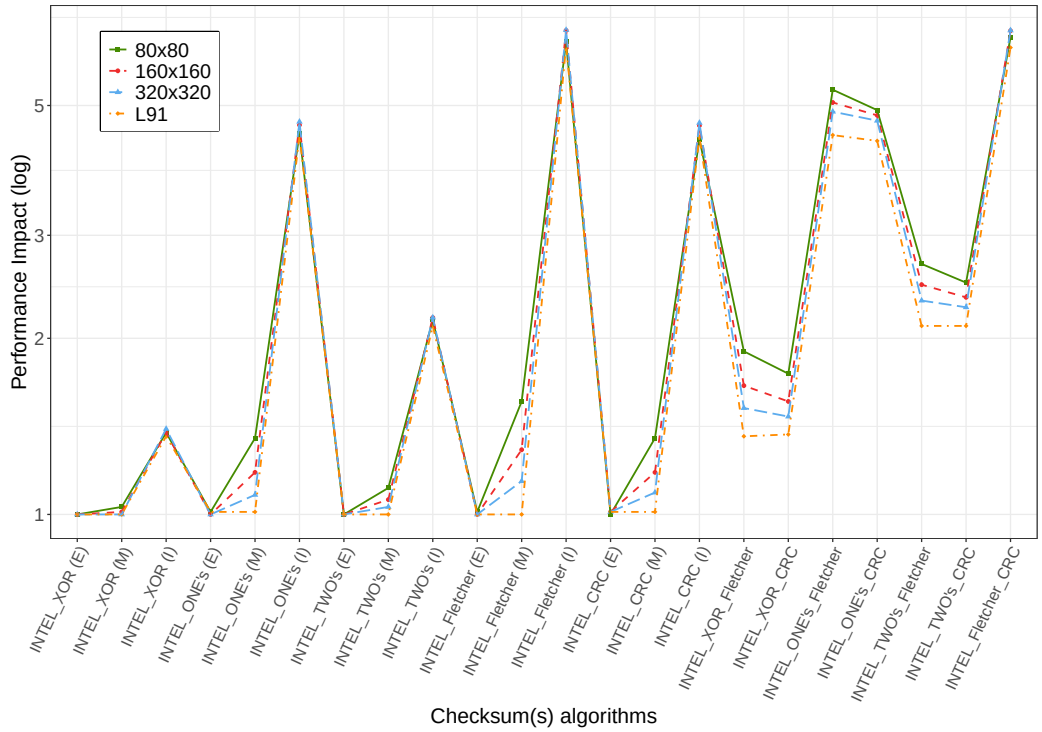


Fig. 4.5: AVX MMM: performance impact incurred by the adoption of the catalog of checksums in the MMM disabling compiler optimizations.

Fig. 4.5 keeps the same color and type of lines used in Fig. 4.4. In Fig. 4.5, we can see an increase in the performance impact of the one's complement checksum with respect to Fig. 4.4. This increase occurs due to the fact that one's complement checksum requires the addition of all values to be checked and subsequently adding the carry bit back into the result before a final inversion [46]. 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 one'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 scalar code that increases the performance impact of that diagnostic mechanism in the AVX-based MMM.

As we mentioned in Section 4.3.1, we have executed a set of experiments with square matrices employing high-level compiler optimizations. The obtained performance impact ratio is shown in Table 4.3 together with the non-optimized results for comparison purposes. It should be noted that in order to have a fair comparison, the baseline for computing the ratio of the optimized results is the execution time of an optimized MMM with the same optimization level.

Tab. 4.3: Performance impact ratio in square matrices with varying compiler optimization

Checksum implemented	GCC Compiler						MSVC Compiler					
	Arm (-O3)			Arm (-O0)			AVX (-O2)			AVX (-O0)		
	80	160	320	80	160	320	80	160	320	80	160	320
XOR (E)	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
One's (E)	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.01	1.00	1.00
Twos's (E)	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Fletcher (E)	1.00	1.00	1.00	1.00	1.00	1.00	1.02	1.00	1.00	1.01	1.00	1.00
CRC (E)	1.00	1.00	1.00	1.03	1.00	1.00	1.01	1.00	1.00	1.00	1.00	1.00
XOR (M)	1.01	1.00	1.00	1.01	1.00	1.00	1.03	1.00	1.00	1.03	1.01	1.00
One's (M)	1.01	1.01	1.00	1.02	1.01	1.00	1.20	1.07	1.02	1.35	1.18	1.08
Two's (M)	1.00	1.00	1.00	1.01	1.00	1.00	1.06	1.01	1.00	1.11	1.06	1.03
Fletcher (M)	1.06	1.03	1.01	1.04	1.01	1.00	2.54	1.75	1.26	1.56	1.29	1.14
CRC (M)	1.05	1.02	1.01	1.14	1.04	1.02	1.24	1.10	1.00	1.35	1.18	1.09
XOR (I)	1.09	1.09	1.08	1.21	1.12	1.12	1.11	1.08	1.07	1.38	1.38	1.38
One's (I)	1.33	1.33	1.33	1.72	1.43	1.42	3.06	3.03	2.88	4.69	4.63	4.47
Two's (I)	1.17	1.17	1.17	1.31	1.20	1.18	1.40	1.36	1.34	2.17	2.17	2.17
Fletcher (I)	3.65	3.64	3.64	2.84	2.14	2.12	15.16	15.07	13.97	6.73	6.72	6.42
CRC (I)	6.04	6.03	6.03	5.58	3.76	3.70	4.65	4.55	4.25	4.67	4.62	4.38
XOR_Fletcher	1.27	1.26	1.26	1.22	1.12	1.12	2.63	1.84	1.40	1.90	1.66	1.52
XOR_CRC	1.12	1.10	1.09	1.52	1.28	1.26	1.39	1.22	1.08	1.74	1.56	1.47
One's_Fletcher	1.62	1.60	1.59	1.85	1.50	1.49	4.73	3.87	3.24	5.32	5.06	4.88
One's_CRC	1.63	1.61	1.59	1.96	1.54	1.52	3.61	3.39	2.95	4.91	4.81	4.71
Two's_Fletcher	1.35	1.34	1.34	1.33	1.20	1.18	2.91	2.12	1.65	2.68	2.47	2.32
Two's_CRC	1.21	1.19	1.18	1.42	1.22	1.20	1.73	1.53	1.34	2.49	2.35	2.26
Fletcher_CRC	3.67	3.66	3.66	2.96	2.16	2.13	15.23	15.08	13.97	6.54	6.71	6.54

We can observe that the performance impact in the external loops implemented with high-level optimizations remains unaltered or with low execution time slowdown. This is aligned with the non-optimized results where we have seen that the execution time required to compute the checksums in this loop is almost negligible in contrast with MMM computation time. Suppose we focus on the intermediate and inner loops. In that case, a decrease in performance impact is appreciated after the application of compiler optimizations in most of the checksums, except for the Fletcher and CRC implementation, whose performance impact is generally higher, especially for Fletcher. This could be explained by the fact that the instructions employed in the Fletcher checksum do not equally benefit from compiler optimization, especially in the AVX implementation, where the employed modulo operation is not natively supported. Therefore, as the baseline MMM execution time used for comparison in these experiments is improved with optimization, the resulting impact is bigger than in the non-optimized solution that already has a higher baseline execution time. Additionally, we observe that the high-level compiler optimization in Arm (gcc compiler) involves a lower performance impact increment than that observed in AVX (MSVC compiler). Optimizations improve the overall result for most techniques regarding checksum combinations, except for some exceptions on those that include the Fletcher checksum as one of the combined techniques.

To conclude the performance experiments, we have analyzed the impact of the 'safe MMM' implemented in the Darknet CNN extracted from YOLO-v3 and in the Tiny YOLO-v3 CNN [125]. We have depicted the results in Fig. 4.6. To offer an insight

into the execution time required to process one image in the selected architectures, we provide the baselines for our experiments, which are obtained with the optimized MISRA C compliant scalar MMM and the AVX-based MMM, both without diagnostic mechanisms. The obtained values are 639.8 and 235.5 seconds for the scalar (Arm cores) and AVX-based MMM (Intel cores) with YOLO-v3 implementation, which can be reduced down to 8.5 and 0.226 seconds in Tiny YOLO-v3. In fact, there is high variability on the inference time of CNNs such as YOLO, as it depends on the input resolution of the network, YOLO version, compiler optimization, number of layers of the network architecture, image resolution or inference platform, among others. As the experiments of this section aim to focus on the performance penalty incurred by the checksums, such inference time analysis is considered out of the scope of this chapter.

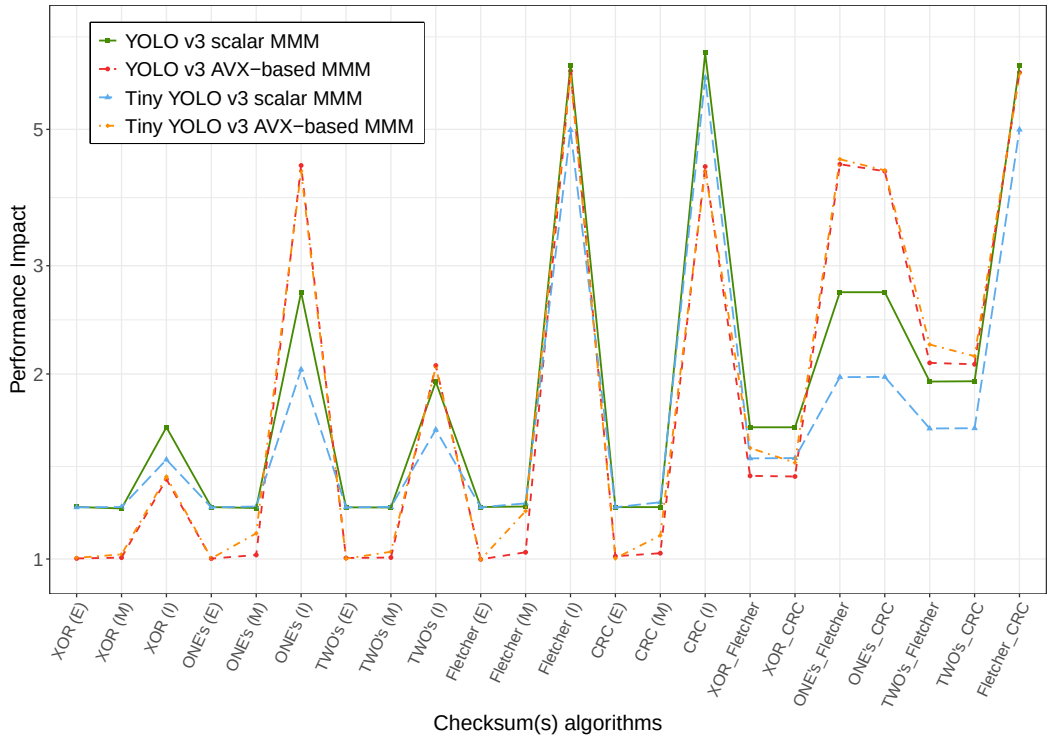


Fig. 4.6: Darknet CNN: performance impact caused by the inclusion of a catalog of checksum algorithms evaluated in YOLO-v3 and Tiny YOLO-v3 CNN.

Regarding the impact of the different checksums depicted in Fig. 4.6, we can remark that both Tiny and YOLO-v3 share the same tendency. In general terms, we can observe that the slowdown caused by the individual experiments in the external and intermediate loops is very close. Consequently, it is preferable to apply the checksum in the intermediate loop rather than in the external loop, since it allows achieving a higher DC, as shown in next section. 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 one'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-v3 algorithm, the performance impact of YOLO-v3 is expected to be lower than that for Darknet in relative terms.

4.3.3 Diagnostic Coverage

In order to quantify the potential achievable DC of diagnostics techniques, common approaches are to make analytical calculations, validation through experimental measurements with fault-injection campaigns, or a combination of both [46].

In this section, we evaluate the DC of the individual ESs, their combinations, and their implementation in the different loops (I, M, E) of the MMM based on fault injection campaigns. We perform 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 4.2.2 (*periodic diagnosis with design time fixed data pattern*). First, we obtain the ES with 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 4.4 gathers the results of the scalar and AVX-based MMM implementations for individual ES and their combinations in the previously mentioned matrices sizes.

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 4.4 show that the XOR, one's complement, and two's complement checksums in the internal loop do not reach a $100\,\%$ DC neither in scalar 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 4.2.1, 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

Tab. 4.4: DC of the scalar and AVX-based MMM

Checksum implemented	Scalar MMM						AVX-based MMM					
	Square			Unbalanced			Square			Unbalanced		
	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_Flet	99.8	99.8	99.8	99.8	99.8	99.8	100.0	100.0	100.0	100.0	100.0	100.0
XOR_CRC	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
One's_Flet	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
One's_CRC	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
Two's_Flet	97.9	97.8	97.7	99.6	99.8	99.6	100.0	100.0	100.0	99.9	99.9	99.9
Two's_CRC	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0

intermediate loops, the DC is highly dependent on the dimension of the matrices involved in the MMM. For instance, implementing Fletcher in the intermediate loop makes bit-error detection vary from more than $50\,\%$ in the square matrices to a $1\,\%$ in a given set of unbalanced matrices ($L2$), according to the results extracted from the scalar 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 matrices 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.

Accordingly, the internal loop solutions seem to be the most suitable ones from a DC perspective, but as we have seen in a 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 4.4, it is possible to see the difference between scalar and AVX-based checksum implementations. According to the XOR checksum, we can

observe that for square matrices, AVX-based and scalar 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, which 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 scalar instructions. However, in experiments with unbalanced matrices implementing the Fletcher checksum in the internal loop, we can appreciate that with matrices dimensions $L1$ and $L2$ there is a decrease in the achieved DC, which remarks that this diagnostic mechanism is less appropriate in AVX-based implementations. Additionally, the experiments with unbalanced matrices highlight the impact of the matrices dimensions in the achievable DC, where for instance, the DC of XOR (I) varies from $0.9\,\%$ to $100\,\%$.

To conclude the DC experiments in scalar MMM, we have computed the DC of a layer extracted from YOLO ($L59$) with one's and two'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 4.4. In the same manner, we have not evaluated the achievable DC of the XOR checksum implemented in the internal loop since from Table 4.4, 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 one's complement with CRC and the combination two'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 DC in the intermediate one. The results confirm our hypothesis, producing an increase from $98.5\,\%$ in one's complement checksum and $96.9\,\%$ in two's complement checksum, up to $100\,\%$ DC when they are implemented in combination with the CRC.

4.3.4 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 80×80 with the scalar MMM and with AVX-based implementation (see Fig. 4.7).

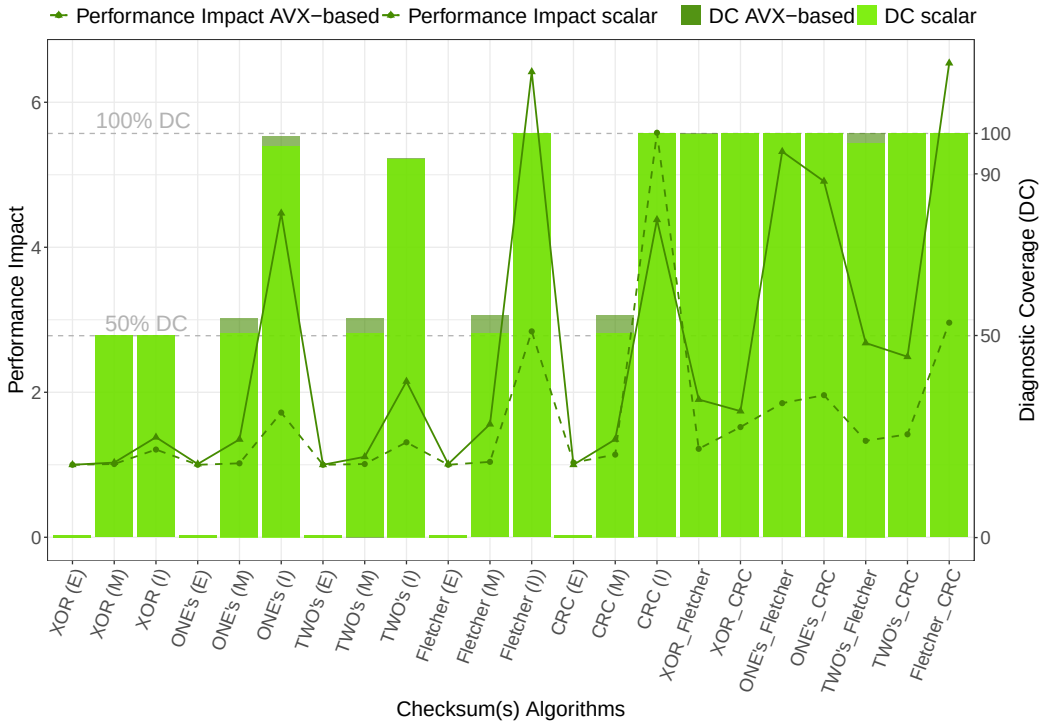


Fig. 4.7: Trade-off between performance impact vs. DC for square matrices of dimension 80×80.

DC is represented in the right-hand y-axis and depicted with two green bar diagrams (the lighter one for the scalar implementation and 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 scalar 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. 4.7 depicts how the two's complement and one's complement checksums implemented in the internal loop and the combinations XOR_Fletcher and Twos_Fletcher do not reach $100\,\%$ DC with the scalar implementation. However, the AVX implementation allows to reach $100\,\%$ DC with all checksum combinations and almost $100\,\%$ with two's complement implemented in the most internal loop. Although it might not be evident in Fig. 4.7, XOR_Fletcher reaches $99.8\,\%$ error detection instead of $100\,\%$ in the scalar MMM (as described in Table 4.4). The reason for 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 to 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, two's_CRC, one's_CRC and one's_Fletcher checksums in scalar MMM

and two's_Fletcher, two's_CRC, XOR_Fletcher and XOR_CRC in the AVX-based MMM. In particular, XOR_CRC offers the lowest performance impact for our particular scalar evaluation framework (Arm core of the Zynq UltraScale+ platform) and AVX-based evaluation framework (Intel core i7-6600U core).

4.3.5 IEC 61508 compliance

The evaluation shown in previous subsections results in a catalog 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 HFT of the safety-related system (depicted in this document in Table 4.5). This allows the safety designer to select the most suitable checksum for each of the safety architectural patterns described in Subsection 4.2.2.

Tab. 4.5: Maximum allowable SIL according to the HFT (Table 3 of IEC 61508-2)

Safe Failure Fraction of an element	Hardware Fault Tolerance (HFT)		
	0	1	2
<60 %	Not allowed	SIL 1	SIL 2
60 % - <90 %	SIL 1	SIL 2	SIL 3
90 % - <99 %	SIL 2	SIL 3	SIL 4
≥ 99 %	SIL 3	SIL 4	SIL 4

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 Table 4.4, most of the individual checksums applied in the external or intermediate loops are not suitable by their own as a diagnostic mechanism for 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 case, 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 matrices 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 4.6, we provide the selected checksums for each SIL and HFT in square matrices of 80×80 dimensions. The grayscale of its cells refers to the range of DC, where the darker the gray, the higher the DC required.

Tab. 4.6: Selected checksums for 80×80 matrices dimension according to SIL and HFT.

			SIL			
			4	3	2	1
HFT	0	Scalar	Non achievable	XOR_Flet (1.22) $^{(iv)}$	XOR_Flet (1.22)$^{(iii)}$	XOR_Flet (1.22)$^{(ii)}$
		AVX		XOR_CRC (1.52)$^{(iv)}$	XOR_CRC (1.52)$^{(iii)}$	XOR_CRC (1.52)$^{(ii)}$
	1	Scalar	XOR_Flet (1.22)$^{(iv)}$	XOR_Flet (1.22)$^{(iii)}$	XOR_Flet (1.22)$^{(ii)}$	CRC (M) (1.14)$^{(i)}$
		AVX	XOR_CRC (1.52)$^{(iv)}$	XOR_CRC (1.52)$^{(iii)}$	XOR_CRC (1.52)$^{(ii)}$	XOR_CRC (1.52)$^{(i)}$
	2	Scalar	XOR_Flet (1.22)$^{(iii)}$	XOR_Flet (1.22)$^{(ii)}$	CRC (M) (1.14)$^{(i)}$	Non Specified
		AVX	XOR_CRC (1.52)$^{(iii)}$	XOR_CRC (1.52)$^{(ii)}$	CRC (M) (1.14) $^{(i)}$	

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

Table 4.6 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 $50.6\,\%$ (scalar) and $55.0\,\%$ (AVX) DC for the selected matrices dimensions. For high DC, the XOR_Flet (scalar) 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.

4.4 Summary

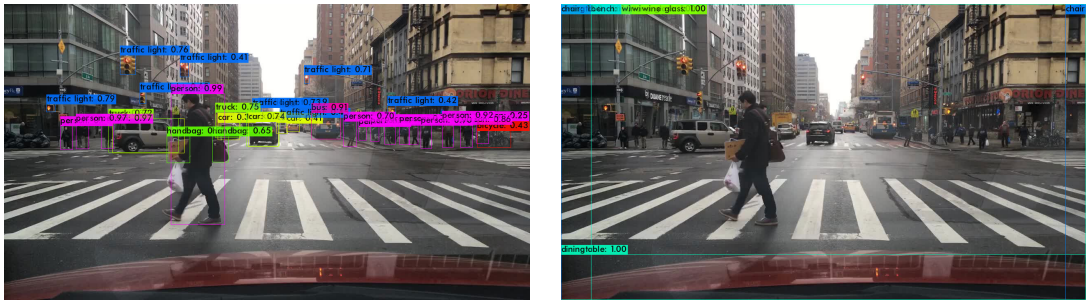
In this chapter, we present an approach to adapt the MMM functions present in ML software libraries in order to avoid and control systematic and random errors according to the considered functional safety standards. On the one hand, for systematic error avoidance, we have seen that the effort to adapt the original MMM code to MISRA C coding guidelines was relatively low and with a negligible

performance impact. This same conclusion can also be extrapolated to the complete Darknet CNN, where we propose solutions for the most frequent violations. On the other hand, this chapter presents an approach for runtime fault detection based on a combination of ESs and safe architectural patterns. We evaluate 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 matrices sizes computed with the scalar 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 matrices 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 provide a pre-selection of the most suitable checksums for the CNN layer that corresponds to the 80×80 matrices dimensions 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 observe 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 scalar MMM, improving execution time by a factor between 3.97 and 6.57 for different matrices sizes.

In summary, the contribution of this chapter defines, implements, and evaluates a catalog of diagnostic techniques by proposing methods for the system safety designers to select the most suitable option based on the affordable performance impact and the target DC of their specific safety application.

5 Exploiting Safe Parallelization on GPUs

As we explained in Chapter 1, the high level of computing parallelism of GPUs, together with their complex memory hierarchy, may spread a single random hardware error to multiple errors [134], jeopardizing the computation correctness. As a GPU-based implementation example in the automotive domain, Fig. 5.1 illustrates the possible consequences of a single error on a CNN-based object detector. We classify an image[1] from Berkeley DeepDrive dataset [118] employing YOLO-v3 [126] based on a pre-trained model with the COCO dataset[2] [135]. In the absence of errors, the CNN correctly detects 35 classes, such as people or traffic lights, among others (Fig. 5.1a). However, a single-bit error injection in a single weight of the first layer of the CNN leads to detecting 11 classes not present in the image, such as three wine glasses or four dining tables (Fig. 5.1b), but no people or traffic lights. In contrast, if the error is injected at bit 705, it does not impact the CNN inference, becoming a latent error if further diagnostics are not in place. These errors are of particular concern in safety-related systems, where misclassifications can lead to catastrophic consequences.



(a) Error-free or single-bit error (bit 705) inference

(b) Single-bit error (bit 734) inference

Fig. 5.1: Example of a single-bit error impact in an CNNs-based object detection application

Following the preceding chapter's approach, we continue focusing on the most computationally expensive operation of the CNNs, the MMM. This chapter paves the

[1] "9e75b2a9-98437b5b.jpg" from "10K Image" package

[2] "yolov3.cfg" and "coco.data" extracted from the configuration folder (`https://github.com/AlexeyAB/darknet/tree/darknet_yolo_v3`). Weights from `https://pjreddie.com/media/files/yolov3.weights`

way towards the safe implementation of CNN-based safety solutions on massively parallel GPU-based platforms through the adaptation of the diagnostic catalog presented in Chapter 4. Additionally, this solution supports an indirect functional diagnosis of the used GPU components (without needing component design and implementation details knowledge) and is complementary and can be combined with other techniques, such as Algorithm-Based Fault Tolerance (ABFT) or Algorithm-Based Error Detection (ABED). In this chapter we have taken CUTLASS library [18] as baseline and applied to it the following modifications:

1. We adapt the catalog of diagnostic mechanisms defined for scalar and vectorized implementations (Chapter 4) for being implemented in CUDA cores with Single Instruction Multiple Threads (SIMTs) math instructions. We include this catalog in CUTLASS library to compute an array of ESs jointly implemented with safety architectural patterns, such as those defined in Chapter 4, to reach various levels of DC against random errors in HPEC platforms integrating GPUs.
2. We analyze the performance impact and the DC of this catalog of mechanisms for multiple matrices dimensions in an NVIDIA Jetson Xavier NX GPU and compare them against scalar and AVX-based implementations from Chapter 4.
3. We perform an analysis of the trade-off between DC and performance for a specific matrices dimensions evaluating the required achievable DC by the IEC 61508 standard for different SILs and architectural patterns.
4. In addition, we calculate the overall performance impact incurred by individually including each GPU-based catalog diagnostic mechanism in YOLO-v3 and Tiny YOLO-v3 for two GPU memory architectures.

The rest of this chapter is structured as follows: Section 5.1 describes the error-detection adaptations for the MMMs implementations based on GPUs. Section 5.2 describes and discusses the results providing a comprehensive analysis of them. Finally, Section 5.3 summarizes this chapter.

5.1 Enhancing MMM Safety

In this section, we describe the adaptations performed to the catalog of diagnostic techniques to be implemented in the high-performance MMM CUTLASS library. As in the previous chapter, those diagnostics are implemented in several loops of the MMM.

In this case, they compute an array of ESs instead of a single one to detect MMM execution errors. We also study their reproducibility and memory management to minimize the performance impact on HPEC platforms that include GPUs.

5.1.1 Diagnostic Techniques

The main idea is to compute an array of ESs at runtime to provide the MMM execution with error detection capabilities. For this purpose, we propose to protect the data employed at the arithmetic operation level by including a GPU-based adaptation of the catalog of diagnostic mechanisms described in the previous chapter. We focus on thread-level GEMM, the lowest loop nests of CUTLASS [18]. At this level, each thread is responsible for processing a certain number of Matrix Multiply-Accumulate (MMA) operations in a triple nested loop denoted as inner (I), intermediate (M), and external (E) in the same way as in Algorithm 1.

Regarding the GPU-based adaptations applied to the catalog of diagnostics, XOR, one's, and two's complement checksums employ a low-level Parallel Thread Execution (PTX) [136]. We code them with an instruction set provided by this intermediate language, aiming to be portable across multiple GPU architectures. Among the benefits, the extended-precision integer arithmetic instructions allow holding carry-in and carry-out with a carry-bit flag in an integer addition operation. This feature is highly desirable in one's complement checksum as it reduces the performance impact incurred by its inclusion. For implementation details, we refer the readers to Section 8.3 where we expound the two's and one's complement checksum implementation based on PTX instructions (Algorithms 17 and 18, respectively). For the Fletcher and CRC, we employ the same implementations in CUDA than in scalar; we refer the reader to Algorithm 3 and 11, respectively.

5.1.2 Reproducibility

Another crucial factor for the safe deployment of CNN algorithms in HPEC platforms exploiting parallelism is the execution order. The MMM involves using floating point data types that do not satisfy the associative property in addition and multiplication operations. Therefore, a different execution order may lead to different results. Hence, the use of floating points is considered a source of numerical reproducibility errors [137], [138]. In [139], NVIDIA highlights that despite all individual operations accomplish with IEEE 754[3] standard [140], the result may

[3]IEEE 754 is the IEEE standard for floating-point arithmetic, specifying formats and methods for binary and decimal floating-point arithmetic in computer programming. This standard seeks to address problems related to floating-point implementations that compromise their portability and make them difficult to use reliably.

not be bit identical. For instance, the operation $(1 + 2^{100}) - 2^{100} \neq 1 + (2^{100} - 2^{100})$ since $(1 + 2^{100}) - 2^{100} = 0$ and $1 + (2^{100} - 2^{100}) = 1$. Consequently, neither the order-independent checksums based on sums operations (XOR, one's and two's complement) nor the order-dependant ones (Fletcher and CRC) can be implemented directly without assuring a deterministic execution order, since both types of algorithms are data dependant.

For guarantying that the ESs are computed in the same order and ensure determinism, we propose employing as many ESs as threads are involved in the MMM. Each thread executes a tile of the MMM in scalar order in a CUDA core. We use the global identifier of each thread to store the final ES computed by each thread at its relative address in an array of ESs. Note that, instead, combining ESs from different threads into a single ES would challenge reproducibility if we cannot enforce a specific computation order, which is not trivial since the internal scheduler of the GPUs can be considered as a black box.

5.1.3 Memory Hierarchy

Applying previous diagnostics techniques to the MMM involves as many accesses to memory as variables we intend to protect. As some implementations protect all variables involved in the MMM (A, B, and intermediate values of C matrices), the MMM computation entails protecting three times the number of performed MMA operations. In GPU-based implementations, memory management usually becomes the main bottleneck in these highly-parallelized platforms [141], and the memory access speed is essential for achieving good performance, mainly for real-time applications that are subject to strict execution time requirements.

We depict the employed memory hierarchy in Fig. 5.2. Initially, we allocate the ESs arrays into the global memory device. At this point, we compare two approaches to address the memory transfer of the ESs arrays (as can be seen in Fig. 5.2):

A) Direct transfer to registers (Glob2Reg): during the MMM execution, these ESs are transferred to registers (the memory with the highest speed) and returned to global memory when the computation finalizes.

B) Intermediate transfer to shared memory (Glob2Shared2Reg): We allocate these ESs in shared memory before transferring them to registers. Iteratively ESs are stored in registers and eventually are transferred from registers to shared memory. Once the computation is complete, the final results are transferred from shared to global memory.

The aim is to compare the two memory hierarchies based on a selection of matrix dimensions. The greyscale used in the GPU memories (Fig. 5.2) denotes the memory access time, with the darkest grey requiring a higher latency.

A particular concern arises in the CRC implementation, which requires further accesses to the lookup table on which this implementation relies. As all threads share these values, we have chosen shared memory instead of global memory to reduce memory access time. However, the shared memory has a reduced size that limits the CRC lookup table dimension. The safety designer has to consider such a limitation at the design phase according to the target GPU's memory. In this chapter, we perform CRC execution byte-by-byte, which requires 2^8 shared memory addresses and, hence, four shared memory accesses per protected 32-bit data word.

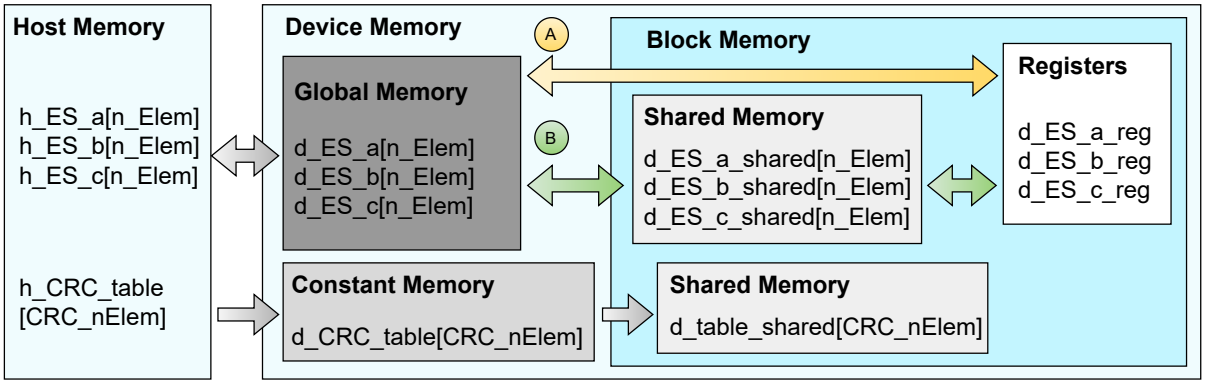


Fig. 5.2: ES transference among GPU memory hierarchies (CRC_nElem: number of CRC lookup table memory addresses. n_Elem: number of ES memory addresses).

In addition to the ESs, we schematize the transfer of the CRC lookup table across the memory hierarchy of the selected platform in Fig. 5.2. Both accesses have in common that the transfer starts in the host memory, where it is initialized, and reaches the shared memory, where each thread finally accesses it. However, the CRC lookup table has static values and is not required to be writeable from the device. Then, we employ the constant memory instead of the global memory. This memory presents lower latency than global memory, especially in accesses where several threads consecutively access the same addresses. We can exploit this CRC lookup table transfer since the MMM executes as many blocks as k tiles it is decomposed in, and all these blocks store the CRC table in their own shared memory. Although the first block accesses occur serialized, the remaining blocks access the constant cache memory (which has the required 1 KB to store the entire table) that is faster than the global memory. We initially store the CRC lookup table in constant memory from host memory before it is transferred to the shared memory at runtime since GPU shared

memory can not be statically initialized. This memory can not be employed for the ESs array since it is only writeable from the host. Then, the device can not store the final ESs array computation, which must be accessible from the host for comparison. This has led to using global memory instead of constant memory for storing the ESs arrays.

5.2 Evaluation

Next, we evaluate the DC and performance impact incurred by the different diagnostic techniques in the GPU-based MMM.

5.2.1 Experimental Set-Up

We use an NVIDIA Jetson Xavier NX platform employing the *clang* compiler with CUDA (both version 10) in an Ubuntu system. In order to minimize system interference, we employ the PREEMPT-RT patch and isolate the NVIDIA Carmel Arm core that executes the program with the highest real-time priority and configure it to run at the maximum frequency. We launch a single MMM stream to the GPU to avoid the uncertainty in the order of execution of the streams associated with several applications running simultaneously [142]. We employ the same matrices dimensions and nomenclature as in Chapter 4.3.1, seeking a fair comparison with the scalar and AVX-based implementations. These values are collected in Table 4.1 and 4.2 for performance impact and DC experiments, respectively.

Regarding performance impact, we run the MMM function ten thousand times and measure the execution time with "xtime_l.h" library to reach a nanoseconds resolution and calculate the mean value. We disregard the initial hundred measurements to avoid the cold-start problems associated with caches and the delays associated with the initial kernel launches [143]. To conclude these experiments, we analyze the performance impact incurred in the execution of YOLO-v3 and Tiny YOLO-v3 when employing the CUTLASS library with the catalog of diagnostics in the two memory hierarchies proposed in the previous section.

Regarding DC experiments, we perform a bit-exhaustive fault-injection campaign in both A and B matrices (as in the previous chapter). It is worth mentioning that we perform these bit-flips in the host before launching the MMM kernel to the GPU.

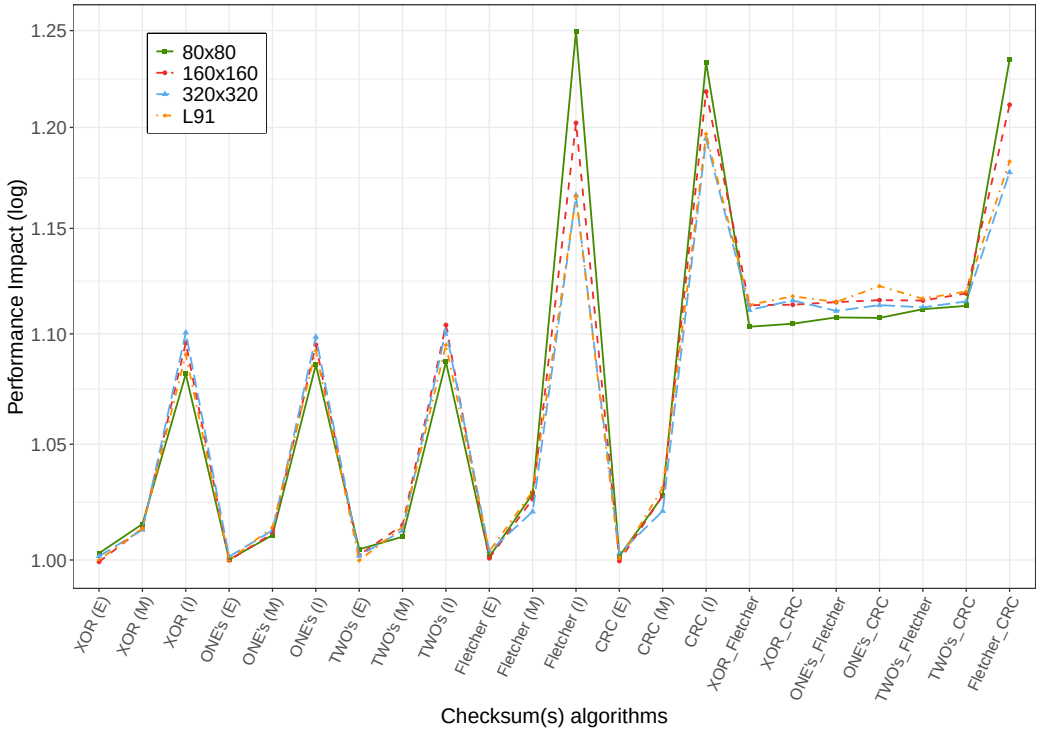
5.2.2 Performance Impact

In order to derive the relative performance impact (n according to Eq. 4.1), we divide the execution time of the MMM including the diagnostic techniques (X), by the execution time of the original MMM (Y), with identical matrices dimensions. This performance impact is relative to single-matrices execution. However, performance impact varies at the system level depending on the safety architectural pattern in which the safety designer includes the diagnostic mechanism(s). Thus, the impact on the periodic diagnostic pattern described in Section 4.2.2 is calculated by dividing the single 'safe MMM' execution by the Diagnostic Test Interval (DTI), not depending only on the performance impact of a single 'safe MMM' execution but also on the number of iterations of the MMM in the DTI. Depending on the implementation of the redundant patterns, the performance impact can either: i) be multiplied by a factor of two when the MMM is re-executed on the same hardware or ii) maintain the same value as that observed for single MMMs, which implies an additional execution platform (double hardware cost). This subsection provides the performance impact relative to single-matrices execution, which can be computed from the timing measurements and according to the implemented safety architectural pattern.

Initially, we perform timing experiments by disabling compiler optimizations to avoid additional safety challenges brought by optimizations (compiler optimization option -O0). We use compiler optimizations throughout this chapter and in the next one to refer to both host and device compilers. In this way, we perform the performance impact experiments with the same optimizations. Fig. 5.3 shows that the performance impact caused by the inclusion of the diagnostic(s) is relatively small in the two memory approaches (between $1\times$ and $1.25\times$ for all matrices dimensions in approach A (Fig. 5.3a) and between $1\times$ and $1.23\times$ in the approach B (Fig. 5.3b)).

In the previous chapter, we show that increasing matrix dimension sizes lead to decreasing performance impacts for scalar and AVX-based implementations. Furthermore, we also show significant differences across different checksums. Moreover, in Fig. 5.3a (memory approach A), we observe a very similar impact for different matrices dimensions in our GPU-based implementation, as well as across different checksums. This behavior relates to the fact that memory accesses dominate execution time. Hence, due to the different checksums, arithmetic operations have a minimal impact that barely changes when varying the matrices dimensions or the specific checksum used. The reader can observe the same behavior in Fig. 5.3b (memory approach B), but this one increases the performance impact variability among dimensions in contrast with approach A. For example, the combination of

(a) Approach A: Direct transfer to registers



(b) Approach B: Intermediate transfer to shared memory

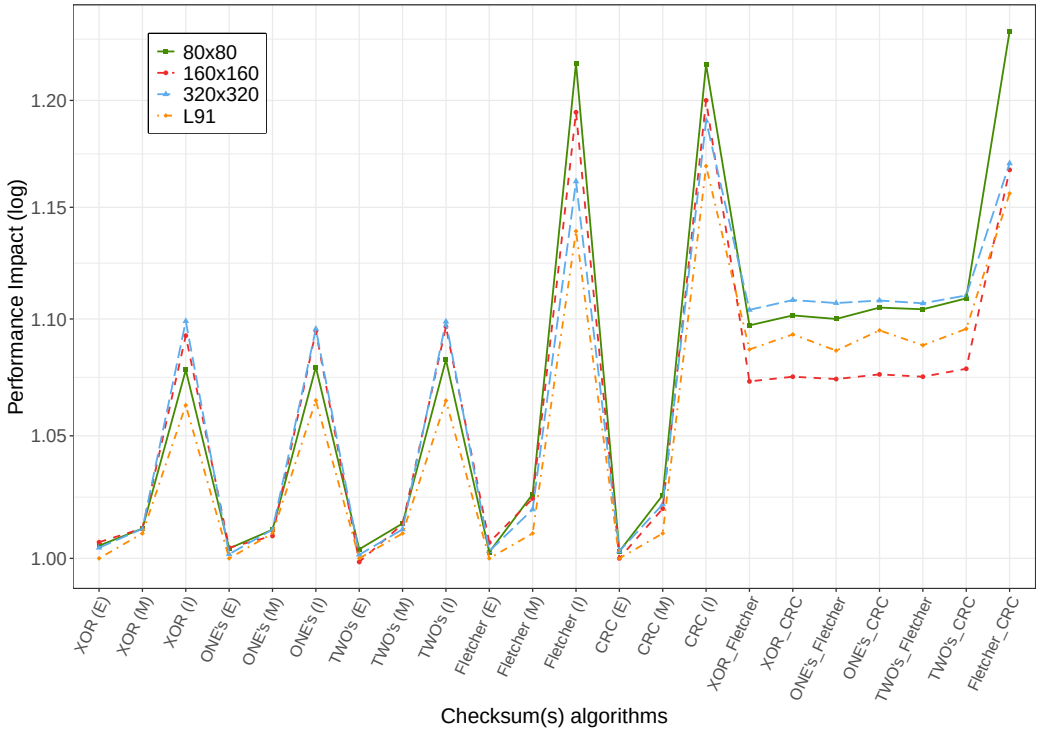


Fig. 5.3: Performance impact with -O0 compiler optimization in GPU-based implementations

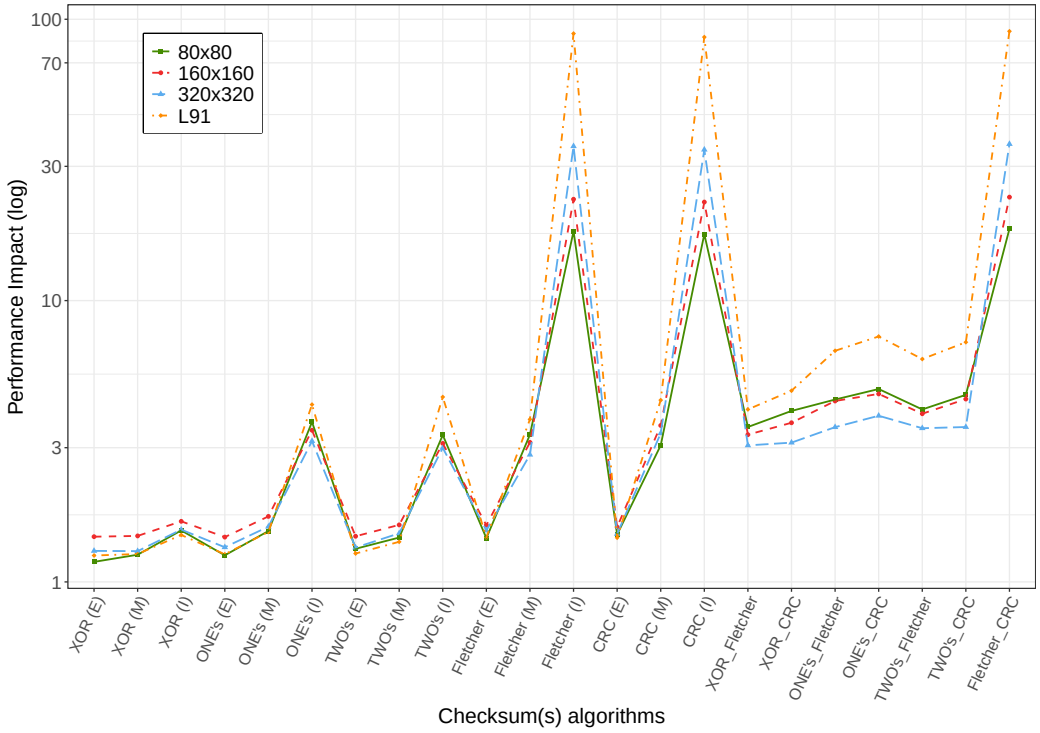
diagnostics varies less than 5% for the set of matrices dimensions evaluated for approach A, increasing up to 7,5% in approach B. Despite this, the variability is moderate, contrasting with the one observed in Section 4.3.2 for scalar and AVX-based implementations. Besides, comparing these two memory approaches, we observe that the intermediate transfer of the ESs arrays to the shared memory produces a performance impact reduction in contrast with the direct transfer to register when at least one diagnostic is implemented in the internal loop. This tendency is more pronounced with the higher matrix dimension $L91$ and remains similar in the square matrix dimensions.

We further evaluate the performance impact with a higher compiler optimization (compiler optimization option -O3). We depict the performance impact results obtained for the two proposed memory approaches in Fig. 5.4. In Fig. 5.4a and Fig. 5.4b, we can observe that in the two approaches, the performance impact increases across all checksums, but particularly for the Fletcher and CRC if implemented in the internal loop. Such impact further exacerbates when increasing the matrices dimensions, as opposed to the case of scalar and AVX-based implementations. CRC implementation involves four accesses to the shared memory to protect each word in the loop. In GPUs, memory latency is crucial from a performance point of view. That explains the performance impact produced by the CRC implementation, reaching an impact of up to $\approx 100\times$ the original implementation. Concerning the Fletcher implementation, the performance impact relates to the modulo operation, used twice in each protected word, whose implementation is not efficient in NVIDIA GPUs compared to the scalar implementation evaluated in Chapter 4.

These results show that with compiler optimizations, not all checksum combinations may be affordable in terms of computing performance impact in contrast with the -O0 compiler optimization option. It should be noted that with the maximum compiler optimization option (-O3), the MMM execution time is three orders of magnitude smaller than disabling compiler optimizations (-O0). This decrease explains the higher relative impact of -O3 compiler optimization experiments. Additionally, comparing the two memory approaches from Fig. 5.4a and Fig. 5.4b, we observe that the results are almost the same, which suggests that the compiler has optimized the code for shared memory buffer transfer in both cases.

Finally, we analyze YOLO-v3 and Tiny YOLO-v3 for concluding performance impact experiments setting up the -O3 compiler optimization. Fig. 5.5 depicts the performance impact incurred by the inclusion of our catalog of diagnostics employing the two memory architectures described in Section 5.1.3. The average execution time obtained as a baseline to process an image in YOLO-v3 and Tiny YOLO-v3 with -O3 compiler optimization is 176.5 and 45.8 ms, respectively.

(a) Approach A: Direct transfer to registers



(b) Approach B: Intermediate transfer to shared memory approach

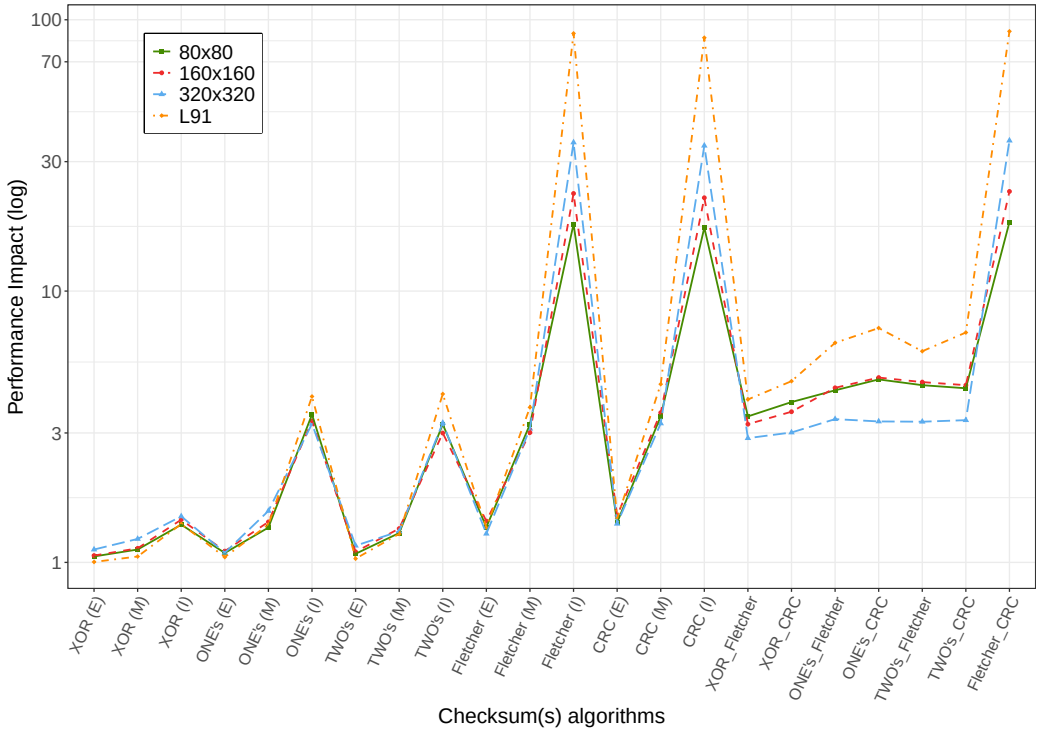


Fig. 5.4: Performance impact with -O3 compiler optimization in GPU-based implementations

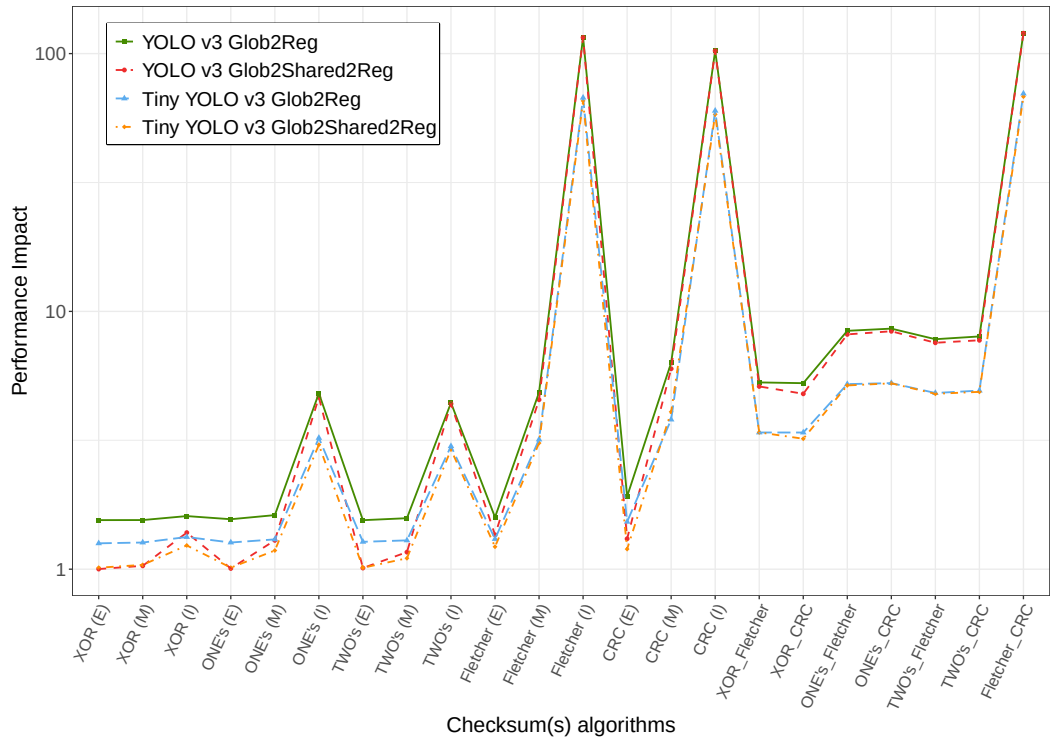


Fig. 5.5: Performance impact caused by the inclusion of our checksums catalog evaluated in YOLO-v3 and Tiny YOLO-v3 for the two proposed memory hierarchies with -O3 compiler optimization

In Fig. 5.5, we observe that intermediate memory allocation of the ESs arrays in shared memory reduces the performance impact in contrast with the direct transfer from global to shared memory when the diagnostics are individually implemented on the outermost loops (intermediate and external loops). However, in the experiment implementing the diagnostics in the internal loops and the diagnostics combinations, we do not appreciate a significative improvement, which can be motivated by the fact that the compiler performs this intermediate allocation as an optimization independently of the fact that we have not explicitly implemented it. In this case, the standard deviation involved in these experiments may be the source of these performance impact differences between the two memory architecture implementations.

5.2.3 Diagnostic Coverage

For the DC experiments, we have followed the same procedure as in the previous chapter performing a fault-injection campaign at the bit-level in all bit positions.

Table 5.1 gathers the achievable DC obtained from including our GPU-based diagnostics catalog in CUTLASS MMM together with the achievable DC in scalar and

AVX-based implementations (Table 4.4 from the previous chapter). This table shows that GPU-based implementations achieve a higher DC than the less parallelized implementations for external (E) implementations. This increment occurs due to the specific implementation details of the GPU-based MMMs, where the entire matrix is decomposed into block tiles that independently compute partial MMMs. In this implementation, the number of values protected in the most external loop increases with a consequent increment in the achievable DC.

Tab. 5.1: DC in scalar (Sca), AVX-based, and GPU-based implementations.

Checksum Implemented	Square									Unbalanced								
	20			40			80			L1			L2			L3		
	Sca	AVX	GPU	Sca	AVX	GPU	Sca	AVX	GPU	Sca	AVX	GPU	Sca	AVX	GPU	Sca	AVX	GPU
XOR (E)	2.5	2.5	10.0	1.3	1.3	10.0	0.6	0.6	12.5	0.4	0.4	6.6	0.1	0.1	12.4	0.1	0.1	12.1
XOR (M)	50.0	50.0	50.0	50.0	50.0	50.0	50.0	50.0	50.0	52.5	52.5	52.5	0.9	0.9	0.9	6.6	10.0	6.3
XOR (I)	50.0	50.0	50.0	50.0	50.0	50.0	50.0	50.0	50.0	52.5	52.5	52.5	0.9	0.9	0.9	100.0	100.0	6.3
One's (E)	2.5	2.5	10.0	1.3	1.3	10.0	0.6	0.6	12.5	0.4	0.4	6.6	0.1	0.1	12.4	0.1	0.2	12.1
One's (M)	52.5	79.2	62.5	51.2	59.2	62.5	50.6	54.4	62.5	54.1	72.9	63.9	1.0	2.2	25.7	7.1	9.9	30.0
One's (I)	98.5	99.2	100.0	97.7	99.2	100.0	96.9	93.8	100.0	98.4	98.9	100.0	97.7	99.2	100.0	96.9	99.9	100.0
Two's (E)	2.5	2.5	10.0	1.3	1.3	10.0	0.6	0.6	12.5	0.4	0.4	6.6	0.1	0.1	12.4	1.0	0.2	12.1
Two's (M)	52.3	68.8	61.7	51.1	59.1	61.7	50.6	54.2	61.7	54.1	63.5	63.2	1.0	1.7	24.1	7.1	9.6	28.5
Two's (I)	96.9	96.9	95.7	95.3	95.3	95.7	93.8	99.2	95.7	98.4	92.6	95.9	90.7	90.7	91.5	96.9	100.0	92.0
Fletcher (E)	2.6	3.5	10.0	1.3	1.5	10.0	0.6	0.7	12.5	0.4	0.5	6.6	0.1	0.2	12.4	0.1	0.2	12.1
Fletcher (M)	52.2	68.8	62.5	51.1	60.0	62.5	50.6	55.0	62.5	54.1	73.8	63.9	1.0	2.2	25.7	7.1	10.0	30.0
Fletcher (I)	100.0	96.9	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	99.9	100.0	100.0	99.9	100.0	100.0
CRC (E)	2.6	3.5	10.0	1.3	1.5	10.0	0.6	0.7	12.5	0.4	0.5	6.6	0.1	0.2	12.4	0.1	0.2	12.1
CRC (M)	52.5	80.0	62.5	51.3	60.0	62.5	50.6	55.0	62.5	54.1	73.8	63.9	1.0	2.2	25.7	7.1	10.0	30.0
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	100.0	100.0	100.0	100.0	100.0	100.0
XOR_Flet	99.8	100.0	90.6	99.8	100.0	90.6	99.8	100.0	90.6	99.8	100.0	91.0	99.8	100.0	81.4	99.8	100.0	82.5
XOR_CRC	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
One's_Flet	100.0	100.0	96.3	100.0	100.0	95.6	100.0	100.0	95.6	100.0	100.0	95.5	100.0	100.0	90.7	100.0	100.0	91.3
One's_CRC	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
Two's_Flet	97.9	100.0	96.3	97.8	100.0	95.6	97.7	100.0	95.6	99.6	100.0	95.5	99.8	99.9	90.7	99.6	100.0	91.3
Two's_CRC	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
Fletcher_CRC	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0

We observe the same trend in the intermediate implementations in square matrices with dimensions greater than 20 and unbalanced matrices L2 and L3, with an exception in the XOR implementation in L3, which was motivated by the nature of XOR which does not detect even single-bit errors. In the rest of the matrices, the DC of AVX is superior to that achieved in GPUs. The reason lies in the AVX-based implementation that protects eight values in the intermediate loop and, in small matrices, the number of values protected is larger than those for the GPU.

Internal loop implementations reach $100\,\%$ DC in all matrices dimensions with one's complement, Fletcher, and CRC. XOR implementation has identical DC to scalar and AVX in all dimensions except in the L3 matrix. DC drops because our GPU-based implementation for these matrices sizes divides the MMM into an even number of computations performed by each thread, and this checksum fails to detect even failures. Two's complement provides similar results to the other implementations.

Finally, all combinations of checksums reach the maximum DC, excluding the ones presented in Table 5.1 that still provide a high DC. In the GPU-based implementation, the combinations, including Fletcher in the intermediate loop, do not provide as much DC as the other implementations. This reduction relates to the location of the Fletcher checksum computation in the code, which could not be kept identical to the other implementations in the GPU-based implementation.

5.2.4 Trade-off Between Performance Impact and DC

In this subsection, we analyze the suitability of our catalog of diagnostics for an 80×80 square MMM, searching for a proper trade-off between DC and performance impact. As explained in a previous subsection, the performance impact varies substantially from the non-optimized compilation to the highest optimization, which has led to assessing both implementations, as it can be seen in Fig. 5.6.

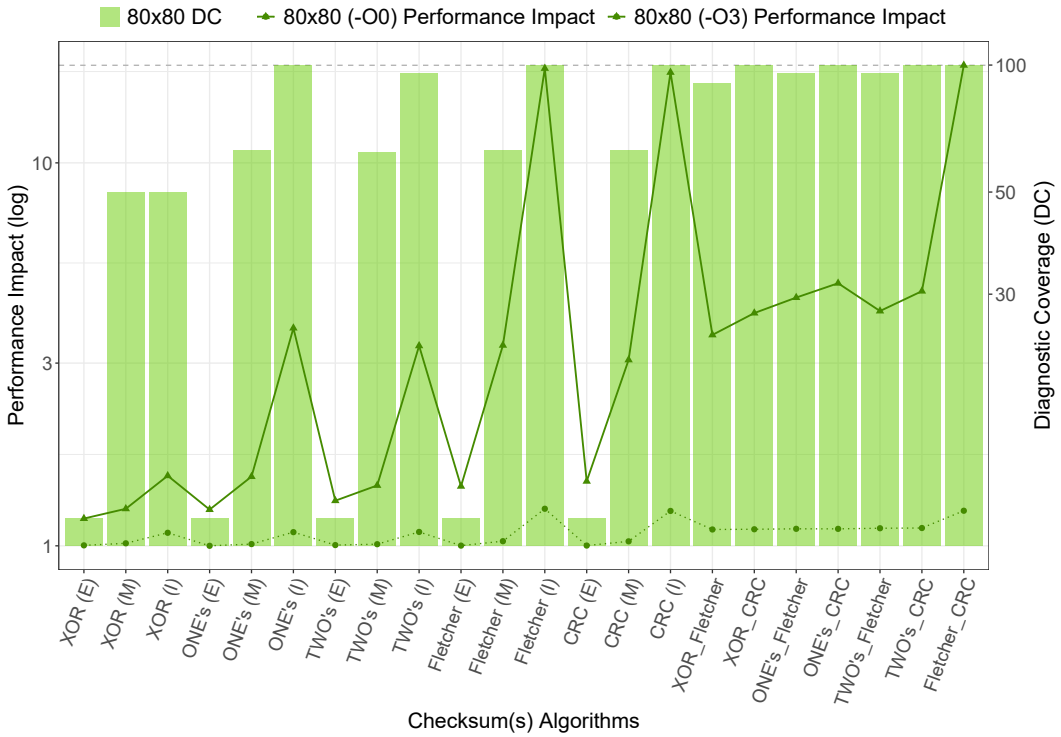


Fig. 5.6: Trade-off between performance impact vs. DC: GPU-based MMM implementations for square matrices of dimension 80×80.

Regardless of the compiler optimizations, both reach the same DC. However, in the experiments with -O0 compiler optimization, we obtain performance impacts between 1.01-1.23 in contrast with -O3, which increases this range to 1.18-17.98.

We can realize from Fig. 5.6 that all combinations, including CRC in the intermediate loop as well as the individual implementations of Fletcher, CRC, and one's complement in the most internal loops allow us to reach $100\,\%$ DC. Among all of them, the one's complement incurs the smallest performance impact, and it is, therefore, the most suitable one for detecting single-bit errors for this MMM dimension (1.09 in non-optimization compilation and 3.70 the highest compiler optimization). For medium DC, the most suitable option for -O0 compilation remains the same, but with -O3 optimization, the best trade-off can be reached with two's complement (I) (3.33 performance impact). Finally, the best performance for low DC is achieved in both optimizations with two's complement (M) (1.01 and 1.44, respectively).

5.2.5 IEC 61508 compliance

After analyzing the most suitable checksums according to the DC range, performance impact, and compiler optimization for 80×80 square matrices dimensions, we examine the DC required to achieve the SIL and HFT established in the functional safety standards. For that, we recur to IEC 61508-2 Table 2 (depicted in Fig. 4.5, where those relationships are defined. As DC and performance impact are MMM dimension-dependent, safety designs must analyze both values based on their particular CNN dimensions. As a representative example, Table 5.2 collects the most suitable checksums and the incurred performance impact to reach the required DC for each HFT and SIL with square matrices of dimension 80×80. We employ a greyscale to denote the DC ranges defined in IEC 61508, with higher DCs being darker gray cells.

Tab. 5.2: Selected checksums for 80×80 matrices dimensions according to SIL and HFT.

			SIL			
			4	3	2	1
HFT	0	-O0	Non achievable	One's (I) (1.09) $^{(iv)}$	One's (I) (1.09)$^{(iii)}$	One's (M) (1.01)$^{(ii)}$
		-O3		One's (I) (3.7)$^{(iv)}$	Two's (I) (3.33)$^{(iii)}$	Two's (M) (1.44)$^{(ii)}$
	1	-O0	One's (I) (1.09)$^{(iv)}$	One's (I) (1.09)$^{(iii)}$	One's (M) (1.01)$^{(ii)}$	XOR (M) (1.01)$^{(i)}$
		-O3	One's (I) (3.7)$^{(iv)}$	Two's (I) (3.33)$^{(iii)}$	Two's (M) (1.44)$^{(ii)}$	XOR(M) (1.25)$^{(i)}$
	2	-O0	One's (I) (1.09)$^{(iii)}$	One's (M) (1.01)$^{(ii)}$	XOR (M) (1.01)$^{(i)}$	Non Specified
		-O3	Two's (I) (3.33)$^{(iii)}$	Two's (M) (1.44)$^{(ii)}$	XOR (M) (1.25) $^{(i)}$	

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

In Table 5.2, it is possible to observe the compiler optimization influence on the choice of the most appropriate diagnostic mechanism. For instance, for a SIL=2 with a single-channel implementation (HFT=0), the chosen checksum changes from one's (M) to two's (M) according to the compilation (-O0 and -O3, respectively).

5.3 Summary

This chapter describes, implements, and evaluates the DC and performance impact of our catalog of diagnostic techniques integrated into the high-performance MMM CUTLASS library ('safe GPU-based MMM') to detect GPU transient and permanent errors. This 'safe GPU-based MMM' software module can be used to detect random hardware errors in the MMM software execution itself and perform an indirect diagnosis of the GPU components that compute the MMM and reduce the probability of undetected latent errors. This chapter paves the way towards the safe implementation of CNN-based safety solutions implemented in GPU-based platforms by providing the MMM with error detection capabilities, and represents a step forward in adherence to the current functional safety standards of safety systems involving ML components implemented in HPEC platforms. With the selected GPU, the developed 'safe GPU-based MMM' software implementation and 80×80 square matrices dimensions, low, medium, and high DCs are achieved with a minimum performance impact of 1.01, 1.09, and 1.09 for the -O0 compiler optimization option and 1.44, 3.33, and 3.7 for the -O3 option.

Moreover, the 'safe GPU-based MMM' can be integrated in different safety architectural patterns with different diagnostic approaches (e.g., design-time fixed pattern, real-time input data) and different fault-tolerance levels based on redundant architectures. Furthermore, as explained in the introduction of this chapter, this approach should also be considered potentially complementary with respect to other existing techniques, such as ABFT and ABED.

The 'safe GPU-based MMM' technique is generalizable to different GPU devices and matrices dimensions. However, achievable DC and the associated performance impact varies with GPU devices/architectures, software libraries, matrices dimensions, and compiler optimizations. Thus, whenever this technique is instantiated, experiments should be re-executed for the given GPU device/architecture, compiler, software library, and application-specific matrices dimensions to select the most suitable candidate from the catalog of integrated diagnostic techniques (e.g., DC vs. performance impact).

6 Methodology to Selectively Protect CNNs: Use Case Application Analysis

This chapter proposes a step-by-step methodology for the selective protection of MMMs to achieve a performance-efficient DC of CNNs deployed on GPUs. In Chapter 5, we propose protecting the MMM by using our catalog of diagnostic techniques implemented at the arithmetic operation level. Since our catalog provides different degrees of DC incurring an execution time penalty according to the implementation and the dimensions of the MMM, a trade-off between performance and DC for each CNN layer shall be analyzed before its application. However, the safety-related systems where our solution could be deployable 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 whole CNN. In those cases, we propose to reduce the performance impact by adopting our step-by-step methodology. We enumerate the main contributions of this chapter as follows:

1. We define a step-by-step methodology to selectively protect CNNs deployed on GPUs by implementing diagnostics in the MMMs. 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.
2. We present a strategy to efficiently determine the achievable DC of big dimension matrices implemented on GPUs. This strategy is based on the DC analysis of the grid of thread blocks in which they are launched on the GPU.
3. Finally, we apply the methodology in Tiny YOLO-v3 employing the previously mentioned strategy for the DC analysis.

The rest of this chapter is structured as follows: Section 6.1 describes our methodology for protecting CNNs and proposes a strategy to evaluate the achievable DC of matrices of big dimension. Then, we implement and evaluate our methodology in Tiny YOLO-v3 in Section 6.2. Finally, we provide a summary of this chapter in Section 6.3.

6.1 Methodology to Selectively Protect CNNs

This section presents a methodology for the selective protection of MMM-based CNNs applied in safety-related applications. This methodology seeks to find a trade-off between DC and the performance impact incurred by including diagnostic techniques. The layered architecture of CNNs is the main foundation of the methodology (see Fig. 6.1). In this layered CNN architecture, an error's impact is highly dependent 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 the three stages depicted from top to bottom in Fig. 6.1. This figure depicts each stage's input (left side of the figure) and intermediate and final outputs (right side). The upper middle part includes an example of MMM-based CNN, particularly Tiny YOLO-v3. The bottom center figure depicts a specific proposal to protect CNN in the form of colored shields.

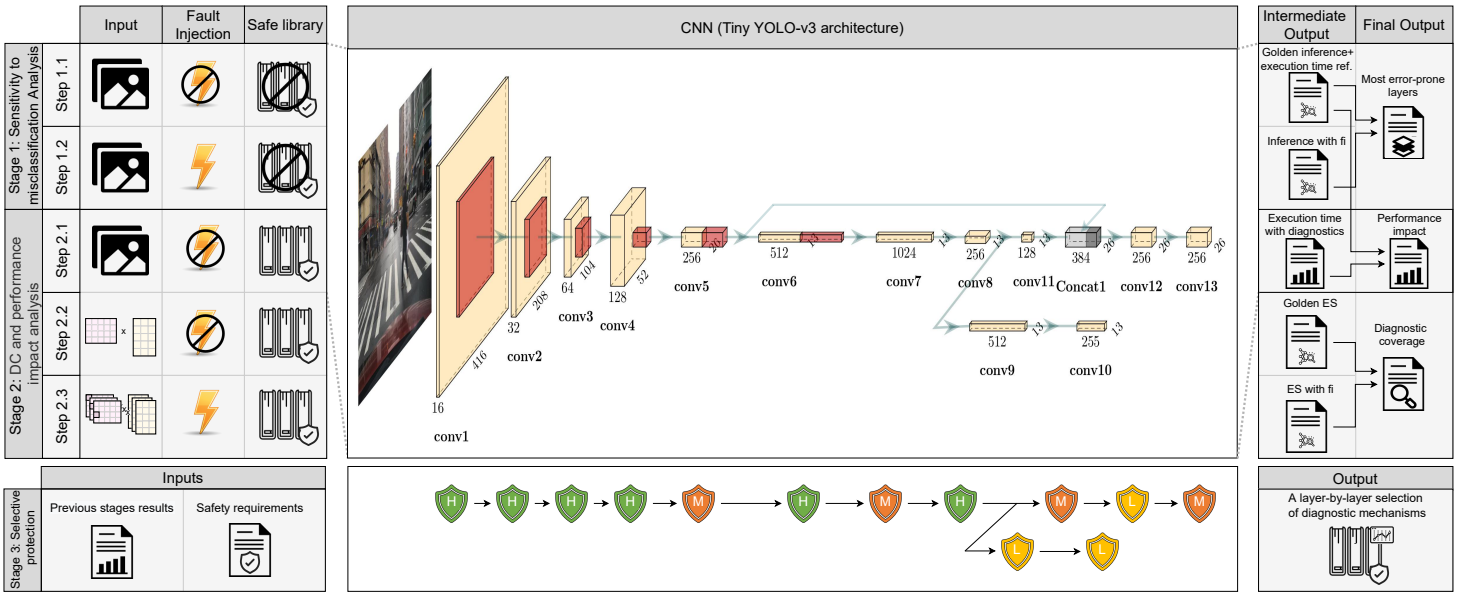


Fig. 6.1: Selective CNN layer protection methodology

Finally, we propose a strategy to efficiently obtain the DC of big dimension matrices, significantly reducing the fault injection campaign effort.

6.1.1 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 identify the most misclassification-prone layers by performing a fault injection campaign in weights, which are values obtained in the training phase of the CNNs, corresponding to B values in the MMM ($C = A \times B$). We define the first two steps as follows (see Fig. 6.1):

Step 1.1 performs the classification with a set of predefined images to obtain a golden reference inference value of 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, the Tiny Yolo-v3 weight configuration file has $9.06e9$ bits [125] that would require the same amount of single-bit error injection executions. According to our GPU-based implementation, the execution time of 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. Instead, we propose injecting faults in the weights' most prone-error bits.

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 [89], [94], [95]. In fact, the errors in the sign and mantissa bits do not significantly impact the prediction, in contrast with those in exponent bits, which cause critical misclassifications [89], [94], [95]. We propose performing a statistical 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 [144].

The resulting inference is compared against the golden inference from Step 1.1, and based on a misclassification criterion, we obtain the most misclassification-prone layers as the final output of this first stage. We propose a semantic comparison of the detected classes against the golden output 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.

In Fig. 6.2, we define the flow of our classification criteria employed to evaluate the sensitivity of each layer to misclassification. We apply this criterion for every classification result from the fault injection campaign and obtain the average value of all the classified images in all the bits belonging to the same layer. We consider that the objects are detected if, by comparing against the golden result: 1) the central point of the box is less than 50 pixels away, 2) the width and height of the boxes vary less than 25 pixels, and 3) the accuracy differs less than 15%. Note that this criterion depends on the specific application (e.g., the input image resolution).

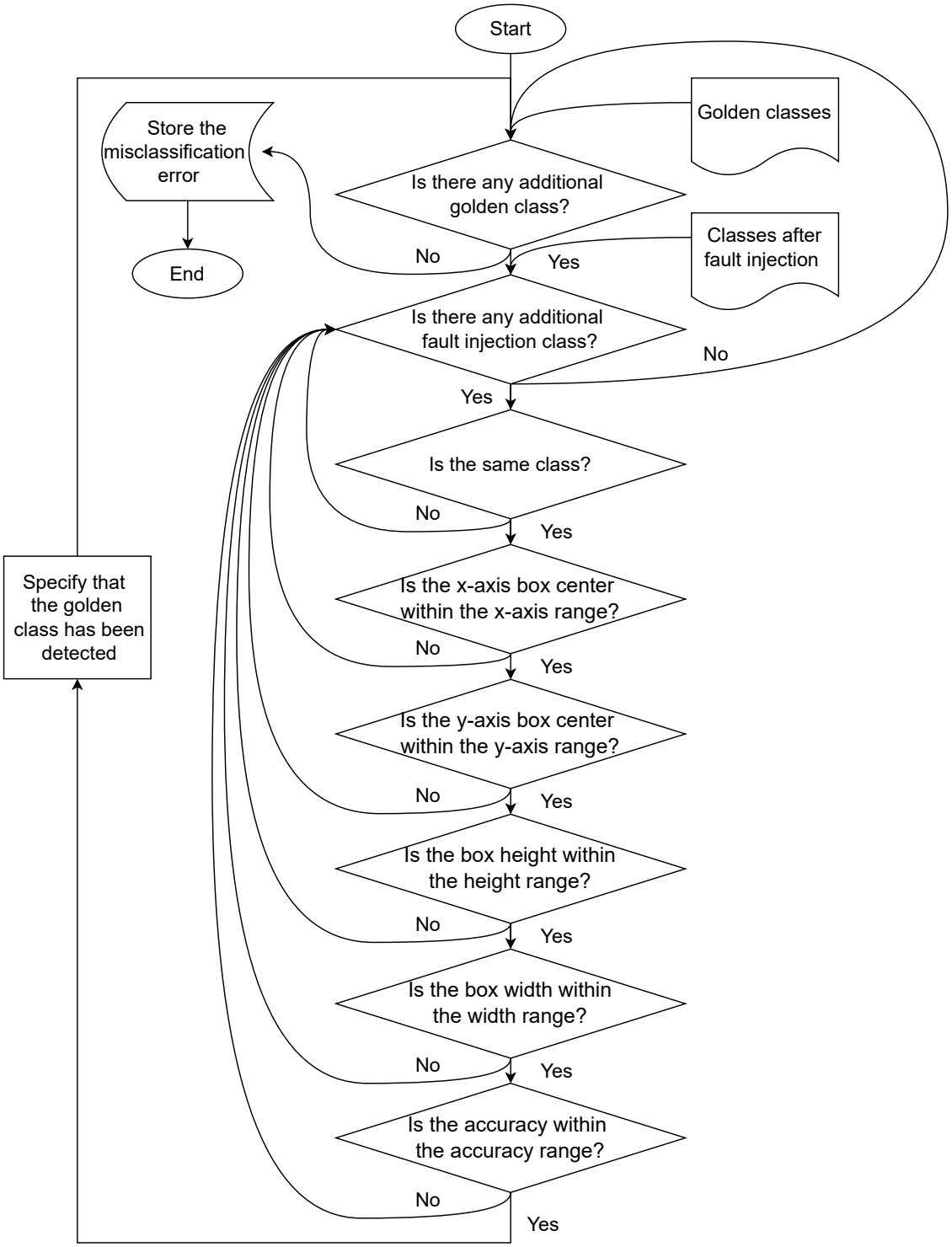


Fig. 6.2: Sensitivity criterion flow to assess the layer's sensitivity to misclassification

From the results obtained after applying the misclassification criterion, we can obtain the undetected objects or False Negatives (FNs) and the average of new objects that appear or False Positives (FPs).

6.1.2 Second Stage: Layer-by-layer Performance Impact and DC

After identifying the most misclassification-prone layers, this stage evaluates the performance impact incurred by including diagnostics in each layer and the maximum achievable DC. To this end, we rely on the 'safe MMM' library proposed in Chapter 5. This stage follows the next three steps depicted in Fig. 6.1:

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 catalog of diagnostic techniques in the MMM execution of each layer without fault injections. This value is then taken as a reference in the next steps to determine whether injected errors are detected by diagnostics (identical ESs at bit-level) 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 matrices with big dimension due to the required number of iterations to cover all input combinations. That is why, in Subsection 6.1.4 we propose a strategy to compute the DC of big dimension matrices implemented on GPUs, relying on the CUDA programming model characteristics.

6.1.3 Third Stage: Selective Protection

The performance impact and DC analysis of all CNN layers, Step 2.1 through Step 2.3, are followed by analyzing the most appropriate diagnostics layer-by-layer. The SIL of the application and the HFT of the system determine the DC range to achieve. Thus, the diagnostic with the lowest performance impact that achieves at least the imposed DC is selected in a layer-by-layer process. However, the selective protection of the CNN's layers is the most reasonable option instead of the complete one, especially in systems under tight timing requirements. The results obtained in the analysis of the CNN's sensitivity to misclassification 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 entire CNN. This 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. 6.1 a particular diagnostics election 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 100% DC in each layer so that such specific diagnostics would be a proper 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.

6.1.4 DC Analysis in Big Dimension Matrices

This subsection decomposes into two parts. First, we explain the base of our strategy to evaluate the DC of big 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.

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. 3.8 in the background). These blocks execute according to the 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 big dimension 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 do not detect errors affecting the same bit position of an even number of data words.

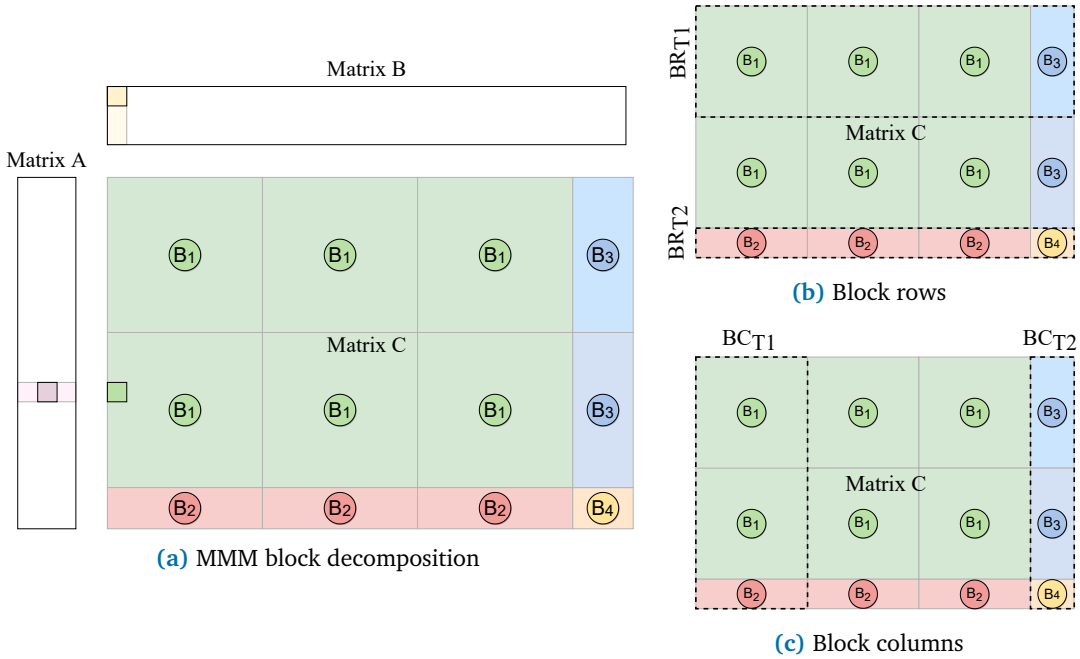
(F2) Block dimensions: determines the amount of data computed by each block. A higher amount to be protected may lead to lower diagnostics effectiveness [46].

(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 [141].

(F4) Fault type: the diagnostics effectiveness varies according to the type of errors (e.g., single-bit, burst, random errors...). This chapter focuses on single-bit errors.

(F5) MMM implementation: the specific software implementation and the location of diagnostics techniques in the end user's code are crucial in the effectiveness of the diagnostics (as we postulate in the previous chapters).

Fig. 6.3a depicts a MMM with representative matrices 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 Section 3.2.5, 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. 6.3: MMM decomposition into blocks

DC computation per error source

Depending on the error source (*F3*), the computation of the DC differs. We propose two scenarios according to this error source:

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. 6.1 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})[1], being i the block type previously defined (from $B1$-$B4$):

$$DC = \frac{\sum_{i=1}^{4} N\_blocks_{Bi} \times N_{det\_Bi}}{N_{fi}} \tag{6.1}$$

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*).

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. 6.3b and Fig. 6.3c, 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. 6.2 and Eq. 6.3:

$$Det_A = (B1^{\dagger}_{det_A} + B3^{\dagger}_{det_A}) \times N\_BR_{T1} + B2^{*}_{det_A} + B4^{*}_{det_A} \tag{6.2}$$

$$Det_B = (B1^{\otimes}_{det_B} + B2^{\otimes}_{det_B}) \times N\_BC_{T1} + B3^{\circledast}_{det_A} + B4^{\circledast}_{det_A} \tag{6.3}$$

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 for 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. 6.4 presents the final step to compute

[1] 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.

the DC. As we perform a fault injection campaign at the bit-level, the number of injected errors depends on the dimension of the input matrices and the data size of the data type employed ($data\_size$).

$$DC = \frac{Det_A + Det_B}{N_{fi}} = \frac{Det_A + Det_B}{(M+N) \times K \times data\_size} \tag{6.4}$$

In those errors that appear in the global memory for the previously defined *F1* and *F5* factors, the blocks $B2$, $B3$, and $B4$ (the smallest) could detect other errors as opposed to those detected by $B1$ (the biggest). As $B1$ dimensions are multiples 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.

6.2 Evaluation

In this section, we evaluate our methodology and present the results for each of its 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.

6.2.1 Experimental Set-up

For the set-up of the Nvidia Xavier Nx GPU over which we perform the experiments, we continue with the one proposed in Chapter 5 and then refer the reader to

Subsection 5.2.1. Clarify that we denote the layers from Tiny YOLO-v3 as $L1$-$L13$, where the number refers to the order position in the CNN. Table 6.1 gathers the CNN's configuration through the parameters M, N and K. As it was explained in previous chapters: $A{=}M \times K$, $B{=}K \times N$ and $C{=}M \times N$.

Tab. 6.1: Layer-by-layer size, total errors and statistically representative fault injections per layer

Tiny Yolo-v3	Features					
	M	N	K	Target faults	Injections	Timing (%)
L1	173056	16	27	3456	413	20,30
L2	43264	32	144	36864	3114	16,75
L3	10816	64	288	147456	6314	8,60
L4	2704	128	576	589824	8497	4,71
L5	676	256	1152	2359296	9301	3,93
L6 & L9	169	512	2304	9437184	9526	6,21
L7	169	1024	4608	37748736	9574	18,40
L8	169	256	1024	2097152	9265	1,75
L10	169	255	512	1044480	89446	1,07
L11	169	128	256	262144	7427	0,71
L12	676	256	3456	7077888	9501	10,10
L13	676	255	256	522240	8372	1,28

6.2.2 Stage 1: CNN's Sensitivity to Misclassification Analysis

We employ a statistical fault injection campaign on the exponent bits of the weights to analyze the CNN's layers sensitivity to misclassification when they are affected by single-bit 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 ($Target faults$). As we focus on the exponent bits (8 bits) of the weights, the number of target faults of each layer is $target faults = N \times K \times 8$. Table 6.1 summarizes the above mentioned features.

Then, we perform the fault injection campaign for five images extracted from Berkeley DeepDrive dataset [118] (these images are depicted in Subsection 3.3). Applying our previously defined criterion and performing the average across the five images, we obtain the results collected in Table 6.2. This table depicts the average of FNs and FPs in percentage terms.

Tab. 6.2: Layer-by-layer analysis of its sensitivity to misclassification

Sensitivity	Layer Position												
	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	L11	L12	L13
FNs (%)	96,4	90,6	93,7	92,3	83,4	96,8	87,1	99,6	59,8	55,2	74,3	55,7	66,8
FPs (%)	0,2	1,6	1,2	0,9	2,7	1,5	1,2	0,1	6,5	6,5	0,1	27,6	27,3

In FNs column, we observe that modifying a single-bit in the exponent of a unique 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 than 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 $L10$ and $L13$ layers (see Fig 6.1). These layers belong to different branches whose origin is the output from $L8$. Errors in the initial layers propagate virtually to all outputs resulting in very high FNs. Instead, errors in the final layers have a lower impact due to their propagation is lower, and it is more frequent in the case where only the bounding box or object location is affected, which translates into a FP if the 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.

6.2.3 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.

Step 2.1: Performance impact

Table 6.1 depicts the percentage execution time breakdown across layers for the entire 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 [118]. Additionally, we dispensed with the initial hundred timing measurements to avoid problems related to cache cold-starting and the delays involved with the initial kernel launches [143].

As in the previous chapters, we perform the first set of experiments disabling the compiler optimizations (-O0). We present the layer-by-layer performance impact of Tiny YOLO-v3 CNN in Fig. 6.4 (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. 6.4: 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, the relative performance impact is quite insensitive to layer dimensions. The suitability of the diagnostic catalog in terms of performance impact needs to be analyzed according to the specific application's safety function(s) timing requirements.

However, for those applications with stringent timing constraints, the no-compiler optimizations may not be an option. Thus, this has led us to perform a second set of experiments configuring the maximum compiler optimization (compiler option -O3). Fig. 6.5 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 the graphs depicted in Fig. 6.5.

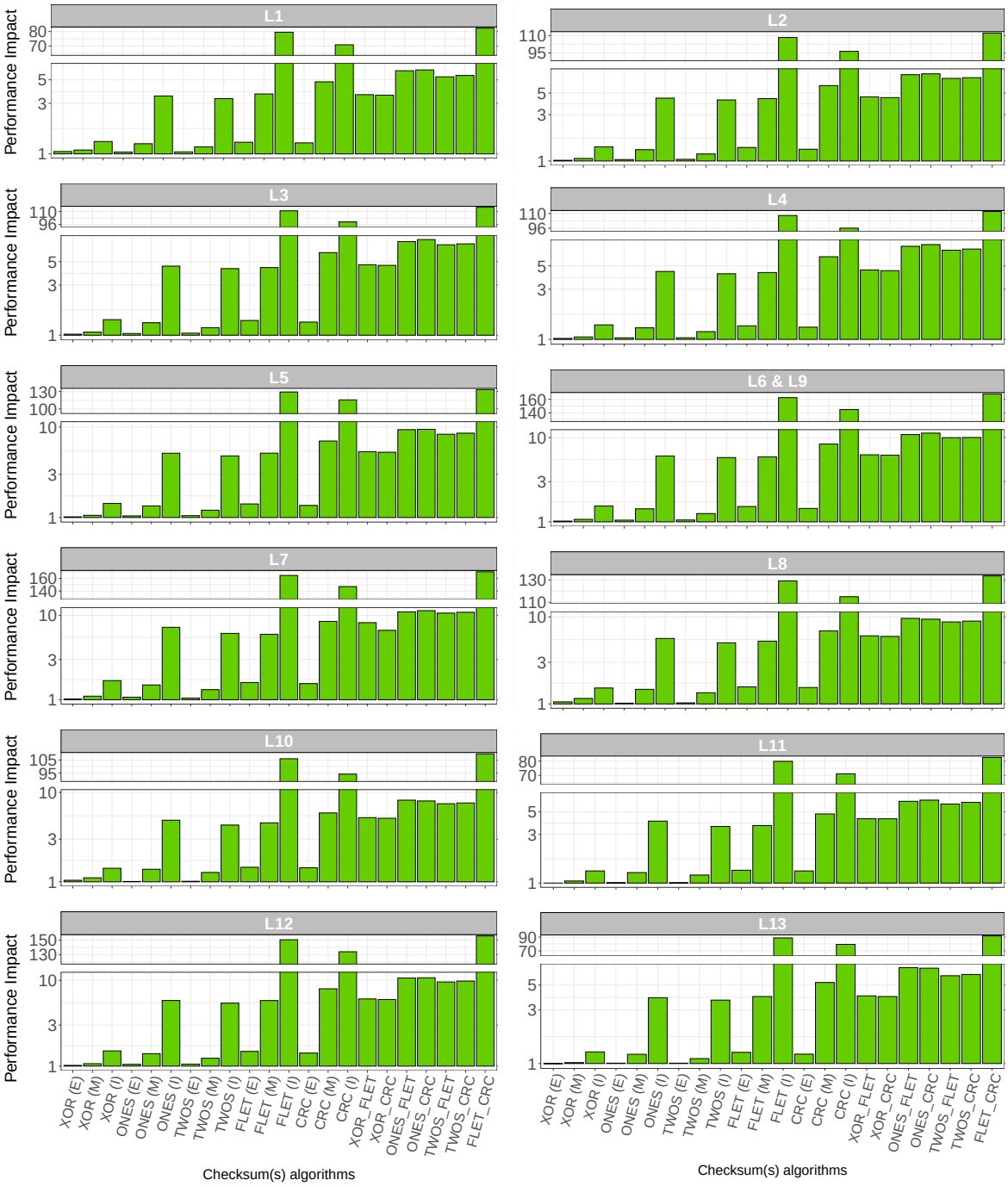


Fig. 6.5: Layer-by-layer performance impact with compiler optimization -O3

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 MMMs on GPUs. Including 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.

Step 2.2- Step 2.3: Diagnostic coverage

As explained in Subsection 6.1.4, 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 Fig. 6.3a), 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 6.1, we present in Table 6.3 the individual grid of thread blocks dimensions into which layers are decomposed. Additionally, we include the number of $T1$ BRs/BCs according to each layer and the selected grid of threads blocks employed. Note that the default block size, $<64, 64, 8>$, has been selected 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. We summarize in Table 6.4 the resulting unique blocks, assign to each one an identifier (Block$_{Id}$) and specify the number of injected errors in input matrices A (Injections$_A$) and B (Injections$_B$) and in which layers they are used. For some blocks, errors detected in $B2$ and/or $B3$ may have already been 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.

Tab. 6.3: 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		

Tab. 6.4: Single block dimensions employed in the DC computation of each layer

Layers	Block	M	N	K	Block$_{Id}$	Injected$_A$	Injected$_B$
All except L1-2	B1	64	64	8	1	16384	16384
L1	B1	64	16	27	2	55296	13824
L2	B1	64	32	8	3	16384	8192
L10	B2*	64	63	8	4	16384	16128
L13	B2*	64	63	8	5	16384	16128
L6-L9 & L11	B3*	41	64	8	6	10496	16384
L10	B3*	41	64	8	7	10496	16384
L5 & L12	B3*	36	64	8	8	9216	16384
L13	B3*	36	64	8	9	9216	16384
L4	B3	16	64	8	10	4096	16384
L10	B4	41	63	8	11	10496	16128
L13	B4	36	63	8	12	9216	16128

We then evaluate the errors detected in the blocks with the dimensions defined in Table 6.4 according to the diagnostic techniques of our catalog and collect them in Table 6.5. We decompose the detected errors according to the input matrix where the fault injection campaign injects the error (Det_A and Det_B). Those blocks that do not appear in the table do not detect any other error (i.e., there are $B2$ and $B3$ blocks that detect errors that $B1$ and $B4$ blocks have already detected).

Tab. 6.5: Faults detected in the single blocks defined in Table 6.4

Diagnostics	Faults Detected for each Block$_{Id}$												
	1		2		3		6	8	10	11		12	
	Det$_A$	**Det$_B$**	**Det$_A$**	**Det$_B$**	**Det$_A$**	**Det$_B$**	**Det$_A$**	**Det$_A$**	**Det$_A$**	**Det$_A$**	**Det$_B$**	**Det$_A$**	**Det$_B$**
XOR (E)	2048	2048	6912	3456	2048	1024	512	256	0	512	2048	256	2048
XOR (M)	16384	0	55296	0	16384	0	10496	9216	4096	10496	0	9216	0
XOR (I)	16384	0	55296	0	16384	0	10496	9216	4096	10496	0	9216	0
One's (E)	2048	2048	6912	3456	2048	1024	512	256	0	512	2048	256	2048
One's (M)	16384	4096	55296	3456	16384	2048	10496	9216	4096	10496	3840	9216	3840
One's (I)	16384	16384	55296	13824	16384	8192	10496	9216	4096	10496	16128	9216	16128
Two's (E)	2048	2048	6912	3456	2048	1024	512	256	0	512	2048	256	2048
Two's (M)	16384	3840	55296	3240	16384	1920	10496	9216	4096	10496	3600	9216	3600
Two's (I)	16384	14976	55296	12636	16384	7488	10496	9216	4096	10496	14736	9216	14736
Fletcher (E)	2048	2048	6912	3456	2048	1024	512	256	0	512	2048	256	2048
Fletcher (M)	16384	4096	55296	3456	16384	2048	10496	9216	4096	10496	3840	9216	3840
Fletcher (I)	16384	16384	55296	13824	16384	8192	10496	9216	4096	10496	16128	9216	16128
CRC (E)	2048	2048	6912	3456	2048	1024	512	256	0	512	2048	256	2048
CRC (M)	16384	4096	55296	3456	16384	2048	10496	9216	4096	10496	3840	9216	3840
CRC (I)	16384	16384	55296	13824	16384	8192	10496	9216	4096	10496	16128	9216	16128
XOR_Fletc	16384	13312	55296	11232	16384	6656	10496	9216	4096	10496	13056	9216	13056
XOR_CRC	16384	16384	55296	13824	16384	8192	10496	9216	4096	10496	16128	9216	16128
One's_Fletc	16384	14848	55296	12960	16384	7424	10496	9216	4096	10496	14592	9216	14592
One's_CRC	16384	16384	55296	13824	16384	8192	10496	9216	4096	10496	16128	9216	16128
Two's_Fletc	16384	14848	55296	12960	16384	7424	10496	9216	4096	10496	14592	9216	14592
Two's_CRC	16384	16384	55296	13824	16384	8192	10496	9216	4096	10496	16128	9216	16128
Fletc_CRC	16384	16384	55296	13824	16384	8192	10496	9216	4096	10496	16128	9216	16128
Injected Faults	16384	16384	55296	13824	16384	8192	10496	9216	4096	10496	16128	9216	16128

The evaluation of the errors detected in the single blocks (see Table 6.5) continues by applying the strategy described in Subsection 6.1.4 to compute the achievable DC of each CNN layer according to the diagnostic techniques. We present in Table 6.6 the achievable DC of each layer. As a representative example, we evaluate the DC of $L10$, which includes 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. 6.2 and Eq. 6.3. In this particular layer, the complementary blocks do not detect additional errors. These are the expected results since the conditions stated in Subsection 6.1.4 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:

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

The complete set of results is shown in Table 6.6. From them, 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 $\frac{64}{16} = 4$ $(B1)$ for $L1$, and $\frac{64}{64} = 1$ $(B1)$ and $\frac{41}{64} = 0,64$ $(B3)$ for $L7$, and whose DC in XOR (M) decreases from 99,99% to 14,17%, respectively.

Tab. 6.6: 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_Fletc	99,99	99,99	99,89	99,15	94,85	85,90	83,91	88,71	88,68	91,92	94,85	94,84
XOR_CRC	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00
One's_Fletc	99,99	99,99	99,94	99,58	97,42	92,95	91,95	94,35	94,34	95,96	97,42	97,42
One's_CRC	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_Fletc	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_CRC	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00
Flet_CRC	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00

6.2.4 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 6.7. Additionally, we include the lowest performance impact ratio (PI) incurred in each range to protect with the combination of these diagnostics.

Tab. 6.7: Trade-off of performance impact vs DC (HFT=0)

Layers	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

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 used periodically as described in Subsection 4.2.2. 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 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 a 5%. In other words, we could execute the diagnosis once every 76 CNN executions.

In this chapter, we stick to a simple approach based on selecting for each layer the diagnostic with the 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 high enough. Then, the safety designer has to selectively protect each layer based on its propensity to misclassify and the percentage of the execution time of each one according to the entire CNN. Highlight that this approach is subject to iteratively performing fault injection campaigns to verify if the achievable DC with the proposed selection of diagnostics is in the application's target DC range.

6.3 Summary

This chapter provides a methodology decomposed into three stages to selectively protect CNNs implemented on GPUs, focusing on controlling errors at runtime in the MMM through the inclusion of our catalog of diagnostics. 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 and measure the relative execution time of each CNN's layer against the entire CNN to determine the most timing consuming ones. 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 incurred performance impact and the DC layer-by-layer for our catalog of diagnostics explained in Chapter 5. 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. 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.

7 Conclusions and Future Work

Nowadays, there is a clear trend towards systems with a higher degree of autonomy in industry and daily life activities. The AI capacity to carry out complex tasks has become particularly interesting to autonomous systems. This AI feature allows for high levels of precision, surpassing human accuracy in some application areas. However, since some systems are safety-related, they are subject to certification. They must ensure that the AI models are safe for their intended purpose and sufficiently mitigate uncertainty-related risks.

Additionally, these safety-related AI-based systems rely on HPEC platforms to meet their performance demands. Risks associated with deploying safety-related systems on top of these platforms shall be controlled and mitigated. However, these platforms pose an additional challenge for safety certification since traditional functional safety standards do not contemplate their intrinsic hardware complexity.

These challenges have motivated the contributions of this Thesis to the state-of-the-art of safety-related systems integrating AI-based components for performing safety-related functions. In this chapter, we summarize the main findings of the research and its impact and present open areas for future research.

7.1 Summary of Contributions

This Thesis contributes to the state-of-the-art safety-related systems using AI by including a safe catalog of diagnostics techniques into one of the main operations of these algorithms, the MMM. In particular, this Thesis's contributions and which of its objectives it addresses (see Section 1.3) can be decomposed into the following:

- The first contribution focuses on adapting the scalar implementation of the MMM to avoid and control systematic and random errors according to the recommendations of current functional safety standards, such as IEC 61508. On the one hand, for the avoidance of systematic errors, we adapt the MMM according to MISRA C coding guidelines and include defensive programming, providing a MMM implementation amenable to software development practices of safety standards. This is a step forward in AI safety certification, as

the low-level libraries of most widely used AI solutions were not originally conceived for functional safety. On the other hand, we design, implement and deploy a safe catalog of widely used diagnostic techniques implemented in scalar code. Combined with a safe architectural pattern, this catalog provides fault detection capabilities to the MMM at runtime. This contribution directly addresses O1, *'Adopt functional safety practices in ML code subsets'*. In conclusion, our results show how high levels of protection are subject to an impact on code performance, requiring a trade-off between both features, which has led us to the next contribution.

- As a second contribution, we follow an incremental strategy to achieve higher performance than the scalar implementation while trying to find a balance with safety. For this reason, we first adapt the diagnostic catalog to be implemented with vectorization based on AVX instructions and then CUDA code employing PTX instructions when possible. This contribution is a step towards achieving the O2, *'Foster performance improvement of ML solutions while preserving safety', and O1*.As a result, we notice that the suitability of the diagnostics differs when contrasting the three implementations. Therefore, the performance impact and the achievable DC vary according to the optimization of the instructions involved in the computation and the MMM implementation.

- We have assessed all the previously mentioned implementations of the diagnostic catalog regarding the achievable DC and the incurred performance impact when adopted in MMMs. The experiments have been performed using the same set of square and unbalanced matrix dimensions for comparison purposes among the implementations. The MMMs for the scalar and AVX-based implementations have been extracted from YOLO v3 object detector. Instead, we employed the CUTLASS library for evaluating the GPU-based implementation. These experiments address objectives O1 and O2 and partially address O3, *'Implement a 'safe ML' solution prototype in a HPEC platform'*, as a set of matrix dimensions has been extracted from the Darknet CNN. Our findings indicate that, as the matrix dimensions increase, the relative performance impact decreases in both the scalar and AVX-based implementations. However, this is not observed in GPU-based implementations, which we relate to the fact that the chosen MMM dimensions do not fully utilize the GPU resources. Regarding DC, we observe that the effectiveness of the diagnostics significantly varies depending on the relationship between rows and columns of the matrices involved in the MMM. Therefore, it is essential to perform the effectiveness assessment of the diagnostics according to the MMM dimensions.

- The fourth contribution of this Thesis focuses on the implementation of the catalog in an object detector application based on CNNs and deployed over HPEC platforms. For that purpose, we include the catalog of diagnostics and measure the performance impact incurred by their adoption in the MMM employed by YOLO v3 and its tiny version, Tiny YOLO v3. In the case of the GPU-based implementation, we modified YOLO to employ an open-source MMM library, CUTLASS. These experiments evidence that the higher the performance capabilities of the platform over which we deploy the experiments, the higher the performance impact associated with the diagnostics because modifications of the MMMs involve a decrease of its high degree of optimization. Additionally, we observe that the performance impact incurred in the most external loop against the intermediate is very close. Therefore, applying the diagnostic in the intermediate is preferable since the DC is generally higher than in the most external loop. This contribution is directly related to O3 and also deals with O2.

- Finally, the last contribution of this Thesis focuses on those systems with stringent timing constraints in which the performance impact can be critical when protecting the entire CNN. For those cases, we design, implement and deploy a methodology to selectively protect layers of the CNN. It is decomposed into three steps: i) CNN's sensitivity to misclassification analysis, ii) performance impact and DC analysis of each layer, and iii) protection of the layers. Since the time required to analyze the DC of our catalog of diagnostics in big matrices dimensions can be prohibitive in terms of execution time, we provide a strategy that focuses on the DC analysis of the grid of thread blocks in which these matrices are launched on the GPU. We demonstrate the viability of applying our strategy requiring a handleable effort in terms of execution time. Also, we apply the proposed methodology in a Tiny CNN, obtaining performance impacts ranging from 3.8x, 3.3x and 2.6x for achieving the high, medium, and low DC ranges. We demonstrate with a practical example how the maximum performance impact (3.8x) can get down to 0.05% by applying the diagnosis periodically (at least once in each PST period). This contribution is directly related to O3 and also deals with O2.

7.2 Impact

AI is becoming increasingly used in many sectors for deploying complex functionalities since AI solves problems where traditional rule-based algorithms are challenging or even impractical to construct. AI is becoming a source of competitive advantage,

and there is a tendency towards its adoption. This is evidenced by the expected size and growth of the global AI market, which was valued at nearly 136 billion in 2022, with a revenue forecast of 1,800 billion by 2030, representing a compound annual growth rate of 37.3% [145]. Overall, it is clear that there is a growing interest and investment in AI technology.

These ML algorithms require handling massive volumes of data that demand higher computational capabilities than traditional dependable hardware. However, when these systems involving the use of AI are safety-related, they can be subject to certification imposed by the legislation or by the customer who will use these systems. The impact of this Thesis lies in these systems and in those involving the use of AI that have to achieve certain reliability levels.

To the best of our knowledge, at the moment of this Thesis's writing, it was not any initiative devoted to developing a 'safe ML' library at the software level. We have designed a catalog of diagnostic techniques to provide the MMM with detection capabilities that safety designers can quickly adapt in their implementations. Additionally, we adopted this catalog in a high-performance MMM, CUTLASS, going a step towards a 'safe CUTLASS' library. In this way, the development of a 'safe ML' library aims to reduce the design costs associated with ensuring the reliability of ML systems and, therefore, increase the competitiveness of those products entailing the deployment of AI components.

By providing suitable diagnostics solutions for MMM, the heart of ML software implementations, we pave the road towards the adoption of ML solutions in safety-critical systems. Accordingly, with the development of our safe catalog, we attempt to anticipate the software requirements that are expected to be imposed by emerging standards, such as ISO/IEC 5469, to achieve safety certification of systems involving the use of AI components.

The impact of this Thesis is not limited to academic contributions, as it has the potential to make a significant impact on the industry as well. The use of ML algorithms has consolidated as the most effective solution to deploy complex functionalities like perception, opening up new opportunities for innovation in industries such as automotive or railway. The outcomes of this Thesis have a direct impact in these domains since the implementation of the 'safe GPU-based MMM' in CNNs based on MMMs fosters the safe execution of perception tasks, such as object detection. This Thesis has implemented and evaluated the performance impact and achievable DC associated with the protection of object detection tasks through the inclusion of our 'safe GPU-based MMM'. This can pave the way towards the safety software certification of object detector applications based on CNNs.

7.3 Future Work

The results of this Thesis open the door to new research opportunities and directions to extend this Thesis. We identify the following future Research Lines (RLs):

RL1 This Thesis focuses on implementing diagnostics in MMMs deployed over platforms integrating single-cores (based on scalar and vectorized code, C and AVX languages, respectively) and those integrating embedded GPUs (parallelizing CUDA-cores implemented in CUDA language). The first direction in which the contributions of this Thesis can be extended is by analyzing the viability and challenges associated with implementing our catalog diagnostics in further accelerators, such as the Programmable Logic (PL) of FPGAs and the Tensor Cores of NVIDIA GPUs. To better understand the suitability of our catalog of diagnostics, it would be interesting to conduct a comparative analysis of a wider variety of platforms on which our catalog is deployed. This analysis would help to determine the suitability of our diagnostics in different environments and provide insights into the best-performing diagnostic according to the platforms, and it would help to identify any limitations or advantages associated with each platform.

RL2 A further RL consists of defining a catalog of synchronization mechanisms and analyzing computational load balancing in high-performance computing operations and their adequacy to fulfill the requirements imposed by functional safety standards. It may be of particular interest for those systems implemented on platforms with a large number of resources capable of running simultaneously, such as GPUs combining Tensor cores with CUDA cores or the combination of the PL and the Processing System (PS) in the FPGAs.

RL3 We have protected one of the main components of the CNNs by generating an ES or arrays of ESs of the process. However, the rest of the CNN is still subject to errors. We propose to analyze the feasibility of protecting the rest of CNN's elements by generating ESs (i.e., protecting batch and pooling layers would require hardly any modifications) and devise appropriate solutions for the remaining parts (e.g., the rest of layers, image pre-processing...) where protection is convenient.

RL4 In this Thesis, we have briefly analyzed the sensitivity to misclassification of a CNN when single-bit errors impact the MSBs of the activation weights by performing a statistically representative number of fault injections. The main goals focused on verifying the most error-prone layers to center the CNN

protection on them. At this point, we identify two interesting RLs. On the one hand, we propose to evaluate a deeper CNN (higher number of and types of layers) instead of a tiny version of the CNN for the completeness of the analysis. On the other hand, to study the suitability of AI-based techniques for fault injection to identify those inputs or configuration parameters that maximize the impact of errors on the unit under test. The European project Horizon Europe METASAT is partially addressing this RL with a focus on systems in the space domain.

RL5 During this Thesis, we have identified two safe architectural patterns that, combined with the safe catalog of diagnostics, provide the MMM with error detection capabilities. Hence, identifying safe architectural patterns applicable to applications involving AI (i.e., safe envelopes, diversity...) can be an engaging RL. The European project Horizon Europe SAFEXPLAIN will at least partially address this RL. In fact, one of its tasks is devoted to the definition of safety architectural design patterns, such as diagnostic measures or redundancy. Our solution directly fits into the realization of this RL.

RL6 Additionally, explainability techniques are acquiring a crucial role in assuring functional safety in systems involving the use of AI. In this Thesis, we develop a strategy to selectively protect CNNs according to criteria such as performance and sensitivity to misclassification. An engaging RL may be to consider the output of explainability techniques in the protection of CNNs.

RL7 Finally, we aim to transfer this Thesis's contributions to an industrial domain. As a first step, we expect to accommodate them in a case study in the railway domain under the European project Horizon Europe SAFEXPLAIN, which is currently under execution. In addition, we expect to leverage the broad network of industrial collaborators from the institutions involved in developing this Thesis for implementation in industrial domains, as well as collaborate with industrial users focused on the safe deployment of safety-related system software and its functional safety certification.

Bibliography

[1] *IEC 61508(-1/7): Functional safety of electrical / electronic / programmable electronic safety-related systems*, Legal Rule or Regulation, 2010 (cit. on pp. iii, 2, 6, 14–16, 40).

[2] *ISO 26262(-1/11) road vehicles – functional safety*, Legal Rule or Regulation, ISO, 2018 (cit. on pp. iii, 2, 15, 40).

[3] *ISO/IEC 22989: Information technology — Artificial intelligence — Artificial intelligence concepts and terminology*, Legal Rule or Regulation, 2022 (cit. on p. 1).

[4] S. Mozaffari, O. Y. Al-Jarrah, M. Dianati, *et al.*, "Deep learning-based vehicle behavior prediction for autonomous driving applications: A review," *IEEE Transactions on Intelligent Transportation Systems (T-ITS)*, vol. 23, no. 1, pp. 33–47, 2020 (cit. on p. 1).

[5] A. P. Cohen, S. A. Shaheen, and E. M. Farrar, "Urban Air Mobility: History, Ecosystem, Market Potential, and Challenges," *IEEE Transactions on Intelligent Transportation Systems (T-ITS)*, vol. 22, no. 9, pp. 6074–6087, 2021 (cit. on p. 1).

[6] K. Jang, Y. V. Pant, A. Rodionova, *et al.*, "Learning-to-Fly RL: Reinforcement Learning-based Collision Avoidance for Scalable Urban Air Mobility," in *AIAA/IEEE 39th Digital Avionics Systems Conference (DASC)*, 2020, pp. 1–10 (cit. on p. 1).

[7] X. Chen, Z. Li, Y. Yang, *et al.*, "High-resolution vehicle trajectory extraction and denoising from aerial videos," *IEEE Transactions on Intelligent Transportation Systems (T-ITS)*, vol. 22, no. 5, pp. 3190–3202, 2021 (cit. on p. 1).

[8] R. Azoulay, Y. Haddad, and S. Reches, "Machine Learning Methods for UAV Flocks Management-A Survey," *IEEE Access*, vol. 9, pp. 139 146–139 175, 2021 (cit. on p. 1).

[9] A. A. Gumbs, I. Frigerio, G. Spolverato, *et al.*, "Artificial Intelligence Surgery: How Do We Get to Autonomous Actions in Surgery?" *Sensors*, vol. 21, no. 16, p. 5526, 2021 (cit. on p. 1).

[10] A. E. Abdelaal, J. Liu, N. Hong, *et al.*, "Parallelism in Autonomous Robotic Surgery," *IEEE Robotics and Automation Letters (RA-L)*, vol. 6, no. 2, pp. 1824–1831, 2021 (cit. on p. 1).

[11] *SAE J3016: Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles*, Legal Rule or Regulation, 2021 (cit. on p. 1).

[12] M. Cerino, "Review of Fault Mitigation Approaches for Deep Neural Networks for Computer Vision in Autonomous Driving," Interesting a image that shown that 1's to 0's bit transition produce a lower misclassification than 0's to 1's, Thesis, 2020 (cit. on p. 1).

[13] F. Tambon, G. Laberge, L. An, *et al.*, "How to Certify Machine Learning Based Safety-critical Systems? A Systematic Literature Review," *ArXiv*, vol. abs/2107.12045, 2021 (cit. on pp. 2, 28).

[14] H. Tabani, R. Pujol, J. Abella, *et al.*, "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 (cit. on pp. 3, 27, 28, 30, 39).

[15] R. C. Whaley and A. Petitet, "Minimizing development and maintenance costs in supporting persistently optimized BLAS," *Software: Practice and Experience*, vol. 35, no. 2, pp. 101–121, 2005 (cit. on p. 3).

[16] OpenBLAS, *OpenBLAS: An optimized BLAS library*, `https://www.openblas.net/`, Web Page, 2011 (cit. on p. 3).

[17] NVIDIA, *NVIDIA cuBLAS*, `https://developer.nvidia.com/cublas`, Web Page, 2022 (cit. on p. 3).

[18] NVIDIA, *CUTLASS: CUDA Templates for Linear Algebra Subroutines*, `https://github.com/NVIDIA/cutlass`, available online: Dec-2021, 2020 (cit. on pp. 3, 38, 66, 67).

[19] H. Tabani, L. Kosmidis, J. Abella, *et al.*, "Assessing the Adherence of an Industrial Autonomous Driving Framework to ISO 26262 Software Guidelines," in *56th ACM/IEEE Design Automation Conference (DAC)*, 2019, pp. 1–6 (cit. on pp. 3, 27, 37, 39).

[20] Z. You, S. Wei, H. Wu, *et al.*, "White Paper on AI Chip Technologies (2018)," Report, 2018 (cit. on p. 4).

[21] E. Nurvitadhi, S. Subhaschandra, G. Boudoukh, *et al.*, "Can FPGAs Beat GPUs in Accelerating Next-Generation Deep Neural Networks?," Association for Computing Machinery, 2017, pp. 5–14 (cit. on p. 4).

[22] J. E. R. Condia, P. Rech, F. F. d. Santos, *et al.*, "An Effective Method to Identify Microarchitectural Vulnerabilities in GPUs," *IEEE Transactions on Device and Materials Reliability (T-DMR)*, vol. 22, no. 2, pp. 129–141, 2022 (cit. on p. 4).

[23] A. Steimers and M. Schneider, "Sources of Risk of AI Systems," *International Journal of Environmental Research and Public Health*, vol. 19, no. 6, p. 3641, 2022 (cit. on p. 4).

[24] M. Nicolaidis, "Time redundancy based soft-error tolerance to rescue nanometer technologies," in *Proceedings 17th IEEE VLSI Test Symposium (Cat. No.PR00146)*, 1999, pp. 86–94 (cit. on p. 4).

[25] Y. Ibrahim, H. Wang, M. Bai, *et al.*, "Soft Error Resilience of Deep Residual Networks for Object Recognition," *IEEE Access*, vol. 8, pp. 19 490–19 503, 2020 (cit. on pp. 4, 29).

[26] F. F. d. Santos, L. Draghetti, L. Weigel, *et al.*, "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, pp. 169–176 (cit. on pp. 4, 29, 30, 37, 39).

[27] 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 (AERO)*, 2020, pp. 1–9 (cit. on pp. 4, 30).

[28] G. Li, S. K. S. Hari, M. Sullivan, *et al.*, "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*, 2017, p. 8 (cit. on pp. 4, 29, 39).

[29] A. Azizimazreah, Y. Gu, X. Gu, *et al.*, "Tolerating Soft Errors in Deep Learning Accelerators with Reliable On-Chip Memory Designs," in *IEEE International Conference on Networking, Architecture and Storage (NAS)*, 2018, pp. 1–10 (cit. on pp. 4, 29).

[30] Y. Ibrahim, H. Wang, J. Liu, *et al.*, "Soft errors in DNN accelerators: A comprehensive review," *Microelectronics Reliability*, vol. 115, p. 113 969, 2020 (cit. on p. 4).

[31] J. Perez Cerrolaza, R. Obermaisser, J. Abella, *et al.*, "Multi-core devices for safety-critical systems: A survey," *ACM Comput. Surv.*, vol. 53, no. 4, 2020 (cit. on pp. 4, 16, 35, 43, 48, 49).

[32] N. Kranitis, A. Paschalis, D. Gizopoulos, *et al.*, "Software-based self-testing of embedded processors," *IEEE Transactions on Computers*, vol. 54, no. 4, pp. 461–475, 2005 (cit. on pp. 4, 30).

[33] D. Siewiorek and L. K.-W. Lai, "Testing of digital systems," *Proceedings of the IEEE*, vol. 69, no. 10, pp. 1321–1333, 1981 (cit. on p. 4).

[34] A. Kaplan and M. Haenlein, "Siri, Siri, in my hand: Who's the fairest in the land? On the interpretations, illustrations, and implications of artificial intelligence," *Business Horizons*, vol. 62, no. 1, pp. 15–25, 2019 (cit. on p. 12).

[35] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015 (cit. on p. 12).

[36] I. H. Witten, E. Frank, and M. A. Hall, *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann Publishers Inc., 2011 (cit. on p. 12).

[37] A. Avizienis, J. C. Laprie, B. Randell, *et al.*, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing (TDSC)*, vol. 1, no. 1, pp. 11–33, 2004 (cit. on p. 14).

[38] *EN50128 - railway applications: Communication, signalling and processing systems - software for railway control and protection systems*, Legal Rule or Regulation, 2011 (cit. on p. 15).

[39] I. Agirre Troncoso, "Development and certification of mixed-criticality embedded systems based on probabilistic timing analysis," Thesis, 2018 (cit. on p. 15).

[40] *ISO 21448 road vehicles – safety of the intended functionality*, Legal Rule or Regulation, ISO, 2022 (cit. on p. 17).

[41] *ANSI/UL 4600 Standard for Safety for the Evaluation of Autonomous Products*, Legal Rule or Regulation, 2020 (cit. on p. 17).

[42] *VDE-AR-E 2842-61 - Design and Trustworthiness of autonomous/cognitive systems*, Legal Rule or Regulation, VDE Std., 2020 (cit. on p. 17).

[43] *ISO/IEC JTC 1/SC 42 - Artificial Intelligence*, Legal Rule or Regulation, ISO/IEC (cit. on p. 17).

[44] ISO/IEC, "ISO/IEC CD TR 5469 Artificial intelligence — Functional safety and AI systems," (cit. on pp. 17, 19).

[45] *ISO/AWI PAS 8800 Road Vehicles — Safety and artificial intelligence*, Legal Rule or Regulation, ISO, 2023 (cit. on p. 17).

[46] T. C. Maxino and P. J. Koopman, "The effectiveness of checksums for embedded control networks," *IEEE Transactions on Dependable and Secure Computing (TDSC)*, vol. 6, no. 1, pp. 59–72, 2009 (cit. on pp. 22, 25, 43, 54, 57, 86).

[47] R. N. Williams, "A painless Guide to CRC Error Detection Algorithms," 1993 (cit. on p. 25).

[48] U. D. Ferrell and A. H. A. Anderegg, "Applicability of UL 4600 to Unmanned Aircraft Systems (UAS) and Urban Air Mobility (UAM)," in *AIAA/IEEE 39th Digital Avionics Systems Conference (DASC)*, 2020, pp. 1–7 (cit. on p. 27).

[49] J. Henriksson, C. Berger, M. Borg, *et al.*, "Performance Analysis of Out-of-Distribution Detection on Various Trained Neural Networks," in *45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2019, pp. 113–120 (cit. on p. 27).

[50] J. Birch, D. Blackburn, J. Botham, *et al.*, "A Structured Argument for Assuring Safety of the Intended Functionality (SOTIF)," in 2020, pp. 408–414 (cit. on p. 27).

[51] E. Wozniak, H. J. Putzer, and C. Cârlan, "AI-Blueprint for Deep Neural Networks," in *Proceedings of the Workshop on Artificial Intelligence Safety (SafeAI)*, H. Espinoza, J. McDermid, X. Huang, *et al.*, Eds., ser. CEUR Workshop Proceedings, vol. 2808, CEUR-WS.org, 2021 (cit. on p. 27).

[52] M. Kläes, R. Adler, I. Sorokos, *et al.*, "Handling Uncertainties of Data-Driven Models in Compliance with Safety Constraints for Autonomous Behaviour," in *17th European Dependable Computing Conference (EDCC)*, VDE-AR-E, 2021, pp. 95–102 (cit. on p. 27).

[53] R. Zhang, A. Albrecht, J. Kausch, *et al.*, "DDE process: A requirements engineering approach for machine learning in automated driving," in *IEEE 29th International Requirements Engineering Conference (RE)*, 2021, pp. 269–279 (cit. on p. 27).

[54] F. Falcini and G. Lami, "Challenges in Certification of Autonomous Driving Systems," in *IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2017, pp. 286–293 (cit. on p. 27).

[55] 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 (cit. on p. 27).

[56] 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 (cit. on p. 27).

[57] A. Biondi, F. Nesti, G. Cicero, *et al.*, "A safe, secure, and predictable software architecture for deep learning in safety-critical systems," *IEEE Embedded Systems Letters (ESL)*, pp. 1–1, 2019 (cit. on p. 27).

[58] A. V. S. Neto, J. B. Camargo, J. R. Almeida, *et al.*, "Safety Assurance of Artificial Intelligence-Based Systems: A Systematic Literature Review on the State of the Art and Guidelines for Future Work," *IEEE Access*, vol. 10, pp. 130 733–130 770, 2022 (cit. on p. 28).

[59] S. Dey and S.-W. Lee, "Multilayered review of safety approaches for machine learning-based systems in the days of AI," *Journal of Systems and Software*, vol. 176, p. 110 941, 2021 (cit. on p. 28).

[60] G. Vidot, C. Gabreau, I. Ober, *et al.*, "Certification of embedded systems based on Machine Learning: A survey," *ArXiv*, vol. abs/2106.07221, 2021 (cit. on p. 28).

[61] A. Pereira and C. Thomas, "Challenges of Machine Learning Applied to Safety-Critical Cyber-Physical Systems," *Machine Learning and Knowledge Extraction*, vol. 2, no. 4, pp. 579–602, 2020 (cit. on p. 28).

[62] R. E. Lyons and W. Vanderkulk, "The Use of Triple-Modular Redundancy to Improve Computer Reliability," *IBM Journal of Research and Development*, vol. 6, no. 2, pp. 200–209, 1962 (cit. on p. 28).

[63] Infineon, *AURIX Multicore 32-bit Microcontroller Family to Meet Safety and Powertrain Requirements of Upcoming Vehicle Generations*, 2012 (cit. on p. 28).

[64] STMicroelectronics, *32-bit Power Architecture microcontroller for automotive SIL3/ASILD chassis and safety applications*, 2014 (cit. on p. 28).

[65] X. Iturbe, B. Venu, E. Ozer, *et al.*, "The Arm triple core lock-step (TCLS) processor," *ACM Transactions on Computer Systems (TOCS)*, vol. 36, no. 3, 2019 (cit. on p. 28).

[66] S. K. Reinhardt and S. S. Mukherjee, "Transient fault detection via simultaneous multithreading," in *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA)*, vol. 28, Association for Computing Machinery, 2000 (cit. on p. 28).

[67] E. Rotenberg, "AR-SMT: A microarchitectural approach to fault tolerance in microprocessors," *Digest of Papers. Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing (FTC)*, pp. 84–91, 1999 (cit. on p. 28).

[68] S. Mukherjee, M. Kontz, and S. Reinhardt, "Detailed design and evaluation of redundant multi-threading alternatives," in *Proceedings 29th Annual International Symposium on Computer Architecture (ISCA)*, 2002, pp. 99–110 (cit. on p. 28).

[69] M. Gomaa, C. Scarbrough, T. Vijaykumar, *et al.*, "Transient-fault recovery for chip multiprocessors," in *30th Annual International Symposium on Computer Architecture (ISCA)*, 2003, pp. 98–109 (cit. on p. 28).

[70] C. LaFrieda, E. Ipek, J. F. Martinez, *et al.*, "Utilizing Dynamically Coupled Cores to Form a Resilient Chip Multiprocessor," in *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2007, pp. 317–326 (cit. on p. 28).

[71] B. H. Meyer, B. H. Calhoun, J. Lach, *et al.*, "Cost-effective safety and fault localization using distributed temporal redundancy," in *Proceedings of the 14th International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, 2011, pp. 125–134 (cit. on p. 28).

[72] J. Fu, Q. Yang, R. Poss, *et al.*, "On-demand thread-level fault detection in a concurrent programming environment," in *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, 2013, pp. 255–262 (cit. on p. 28).

[73] G. Reis, J. Chang, N. Vachharajani, *et al.*, "SWIFT: Software implemented fault tolerance," in *International Symposium on Code Generation and Optimization (SGO)*, 2005, pp. 243–254 (cit. on p. 29).

[74] H. So, M. Didehban, Y. Ko, *et al.*, "EXPERT: Effective and flexible error protection by redundant multithreading," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2018, pp. 533–538 (cit. on p. 29).

[75] F. Haas, S. Weis, T. Ungerer, *et al.*, "Fault-Tolerant Execution on COTS Multicore Processors with Hardware Transactional Memory Support," in *Architecture of Computing Systems (ARCS)*, J. Knoop, W. Karl, M. Schulz, *et al.*, Eds., Springer International Publishing, 2017, pp. 16–30 (cit. on p. 29).

[76] H. Mushtaq, Z. Al-Ars, and K. Bertels, "Efficient software-based fault tolerance approach on multicore platforms," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2013, pp. 921–926 (cit. on p. 29).

[77] A. Shye, T. Moseley, V. J. Reddi, *et al.*, "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 (cit. on p. 29).

[78] A. Shye, J. Blomstedt, T. Moseley, *et al.*, "PLR: A Software Approach to Transient Fault Tolerance for Multicore Architectures," *IEEE Transactions on Dependable and Secure Computing (TDSC)*, vol. 6, no. 2, pp. 135–148, 2009 (cit. on p. 29).

[79] J. Wadden, A. Lyashevsky, S. Gurumurthi, *et al.*, "Real-world design and evaluation of compiler-managed GPU redundant multithreading," in *ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, 2014, pp. 73–84 (cit. on p. 29).

[80] H. Jeon and M. Annavaram, "Warped-DMR: Light-weight error detection for GPGPU," in *45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2012, pp. 37–47 (cit. on p. 29).

[81] M. B. Sullivan, S. K. S. Hari, B. Zimmer, *et al.*, "SwapCodes: Error Codes for Hardware-Software Cooperative GPU Pipeline Error Detection," in *51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 762–774 (cit. on p. 29).

[82] R. Nathan and D. J. Sorin, "Argus-G: Comprehensive, low-cost error detection for GPGPU cores," *IEEE Computer Architecture Letters (CAL)*, vol. 14, no. 1, pp. 13–16, 2015 (cit. on p. 29).

[83] S. Alcaide, L. Kosmidis, C. Hernandez, *et al.*, "Software-only diverse redundancy on GPUs for autonomous driving platforms," in *IEEE 25th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, 2019, pp. 90–96 (cit. on p. 29).

[84] 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, Washington, D.C., USA: Association for Computing Machinery, 2009, pp. 94–104 (cit. on p. 29).

[85] S. Jain, I. Baek, S. Wang, *et al.*, "Fractional GPUs: Software-Based Compute and Memory Bandwidth Reservation for GPUs," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2019, pp. 29–41 (cit. on p. 29).

[86] S. Alcaide, L. Kosmidis, C. Hernandez, *et al.*, "High-integrity GPU designs for critical real-time automotive systems," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2019, pp. 824–829 (cit. on p. 29).

[87] F. Libano, B. Wilson, J. Anderson, *et al.*, "Selective Hardening for Neural Networks in FPGAs," *IEEE Transactions on Nuclear Science*, vol. 66, no. 1, pp. 216–222, 2019 (cit. on p. 29).

[88] J. E. R. Condia, F. F. d. Santos, M. S. Reorda, *et al.*, "Combining Architectural Simulation and Software Fault Injection for a Fast and Accurate CNNs Reliability Evaluation on GPUs," in *IEEE 39th VLSI Test Symposium (VTS)*, 2021, pp. 1–7 (cit. on p. 29).

[89] A. Ruospo, A. Bosio, A. Ianne, *et al.*, "Evaluating Convolutional Neural Networks Reliability depending on their Data Representation," in *23rd Euromicro Conference on Digital System Design (DSD)*, 2020, pp. 672–679 (cit. on pp. 29, 83).

[90] B. Du, S. Azimi, C. d. Sio, *et al.*, "On the Reliability of Convolutional Neural Network Implementation on SRAM-based FPGA," in *2019 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, 2019, pp. 1–6 (cit. on p. 29).

[91] F. Libano, B. Wilson, M. Wirthlin, *et al.*, "Understanding the Impact of Quantization, Accuracy, and Radiation on the Reliability of Convolutional Neural Networks on FPGAs," *IEEE Transactions on Nuclear Science*, vol. 67, no. 7, pp. 1478–1484, 2020 (cit. on p. 29).

[92] M. A. Hanif, F. Khalid, R. V. W. Putra, *et al.*, "Robust Machine Learning Systems: Reliability and Security for Deep Neural Networks," in *IEEE 24th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, 2018, pp. 257–260 (cit. on p. 29).

[93] 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 26th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, 2020, pp. 1–4 (cit. on p. 29).

[94] 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 (cit. on pp. 29, 37, 83).

[95] A. Bosio, P. Bernardi, A. Ruospo, *et al.*, "A Reliability Analysis of a Deep Neural Network," in *IEEE Latin American Test Symposium (LATS)*, 2019, pp. 1–6 (cit. on pp. 29, 37, 39, 83).

[96] A. Mahmoud, S. K. S. Hari, C. W. Fletcher, *et al.*, "Optimizing Selective Protection for CNN Resilience," in *IEEE 32nd ISSRE*, IEEE Computer Society, 2021, pp. 127–138 (cit. on p. 29).

[97] M. A. Neggaz, I. Alouani, S. Niar, *et al.*, "Are CNNs Reliable Enough for Critical Applications? An Exploratory Study," *IEEE Design & Test*, vol. 37, no. 2, pp. 76–83, 2020 (cit. on p. 29).

[98] L. H. Hoang, M. A. Hanif, and M. Shafique, "FT-ClipAct: Resilience Analysis of Deep Neural Networks and Improving their Fault Tolerance using Clipped Activation," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2020, pp. 1241–1246 (cit. on p. 29).

[99] R. Rana, M. Staron, C. Berger, *et al.*, "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 (cit. on p. 29).

[100] K. Pei, Y. Cao, J. Yang, *et al.*, "DeepXplore: Automated Whitebox Testing of Deep Learning Systems," *Commun. ACM*, vol. 62, no. 11, pp. 137–145, Oct. 2019 (cit. on p. 29).

[101] R. B. Abdessalem, A. Panichella, S. Nejati, *et al.*, "Testing autonomous cars for feature interaction failures using many-objective search," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*, ser. ASE, Montpellier, France: Association for Computing Machinery, 2018, pp. 143–154 (cit. on p. 29).

[102] 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, Florence, Italy: IEEE Press, 2015, pp. 2–7 (cit. on p. 29).

[103] M. Broy, S. Kirstan, H. Krcmar, *et al.*, *What is the Benefit of a Model-Based Design of Embedded System in the Car Industry?* IGI global, 2012 (cit. on p. 29).

[104] K. Adam, I. I. Mohamed, and Y. Ibrahim, "A Selective Mitigation Technique of Soft Errors for DNN Models Used in Healthcare Applications: DenseNet201 Case Study," *IEEE Access*, vol. 9, pp. 65 803–65 823, 2021 (cit. on p. 29).

[105] S. K. S. Hari, M. Sullivan, T. Tsai, *et al.*, "Making Convolutions Resilient via Algorithm-Based Error Detection Techniques," *IEEE Transactions on Dependable and Secure Computing (TDSC)*, pp. 1–1, 2021 (cit. on p. 30).

[106] M. Salim, A. O. Akkirman, M. Hidayetoglu, *et al.*, "Comparative benchmarking: Matrix multiplication on a multicore coprocessor and a GPU," in *Computational Electromagnetics International Workshop (CEM)*, 2015, pp. 1–2 (cit. on p. 30).

[107] Z. Huang, N. Ma, S. Wang, *et al.*, "GPU computing performance analysis on matrix multiplication," *The Journal of Engineering*, vol. 2019, no. 23, pp. 9043–9048, 2019 (cit. on p. 30).

[108] V. Kelefouras, A. Kritikakou, I. Mporas, *et al.*, "A high performance Matrix-Matrix Multiplication Methodology for CPU and GPU architectures," *The Journal of Supercomputing*, vol. 72, 2016 (cit. on p. 30).

[109] I. C. Lopes, F. Benevenuti, F. L. Kastensmidt, *et al.*, "Reliability analysis on case-study traffic sign convolutional neural network on APSoC," in *IEEE 19th Latin-American Test Symposium (LATS)*, 2018, pp. 1–6 (cit. on p. 30).

[110] 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 (cit. on p. 30).

[111] K. Zhao, S. Di, S. Li, *et al.*, "FT-CNN: Algorithm-Based Fault Tolerance for Convolutional Neural Networks," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 32, pp. 1677–1689, 2020 (cit. on p. 30).

[112] J. Kosaian and K. V. Rashmi, *Arithmetic-intensity-guided fault tolerance for neural network inference on GPUs*, Conference Paper, 2021 (cit. on p. 30).

[113] J. Fernández, J. Perez-Cerrolaza, I. Agirre, *et al.*, "Towards Safety Compliance of Matrix-Matrix Multiplication for Machine Learning-based Autonomous Systems," *Journal of Systems Architecture*, 2021 (cit. on p. 32).

[114] J. Fernández, J. Perez-Cerrolaza, I. Agirre, *et al.*, "On the Safe Deployment of Matrix Multiplication in Massively Parallel Safety-Related Systems," *Applied Sciences*, vol. 12, no. 8, 2022 (cit. on p. 32).

[115] J. Fernández, I. Agirre, J. Perez-Cerrolaza, *et al.*, "A Methodology for Selective Protection of Matrix Multiplications: a Diagnostic Coverage and Performance Trade-off for CNNs Executed on GPUs," in *7th International Conference on System Reliability and Safety (ICSRS)*, IEEE, 2023 (cit. on p. 32).

[116] *MISRA C:2012 - guidelines for the use of the C language in critical systems*, Legal Rule or Regulation, MISRA, 2012 (cit. on pp. 33, 40).

[117] J.-L. Boulanger, "Polyspace," in *Static Analysis of Software: The abstract Interpretation*. John Wiley & Sons, Inc, 2013, ch. 3, pp. 113–142 (cit. on pp. 33, 40).

[118] F. Yu, H. Chen, X. Wang, *et al.*, "BDD100K: A Diverse Driving Dataset for Heterogeneous Multitask Learning," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2020 (cit. on pp. 33, 65, 90, 91).

[119] Xilinx, *Zynq UltraScale+ MPSoC*, xilinx.com/products/silicon-devices/soc.html, Web Page, available online: Dec-2022 (cit. on p. 34).

[120] I. Corporation©, *Intel® Core™ i7-6600U Processor*, https://www.intel.co.uk/content/www/uk/en/products/sku/88192/intel-core-i76600u-processor-4m-cache-up-to-3-40-ghz/specifications.html, Web Page, 2015 (cit. on pp. 35, 36).

[121] I. Corporation©, *Intel® Intrinsics Guide*, https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html, Web Page, 2022 (cit. on p. 35).

[122] NVIDIA Corporation & affiliates, *NVIDIA® Jetson Xavier™ NX*, https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-xavier-nx/, Web Page, available online: Dec-2022 (cit. on p. 36).

[123] J. Redmon, *Darknet: Open source neural networks in C*, http://pjreddie.com/darknet/, Web Page, 2016 (cit. on pp. 37–39).

[124] L. Jiao, F. Zhang, F. Liu, *et al.*, "A Survey of Deep Learning-Based Object Detection," *IEEE Access*, vol. 7, pp. 128 837–128 868, 2019 (cit. on p. 37).

[125] P. Adarsh, P. Rathi, and M. Kumar, "YOLO v3-Tiny: Object detection and recognition using one stage improved model," in *International Conference on Advanced Computing and Communication Systems (ICACCS)*, 2020, pp. 687–694 (cit. on pp. 37, 55, 83).

[126] J. Redmon and A. Farhadi, *YOLOv3: An incremental improvement*, 2018. arXiv: 1804.02767 [cs.CV] (cit. on pp. 37, 65).

[127] A. Bochkovskiy, C.-Y. Wang, and H.-Y. M. Liao, "YOLO v4: Optimal speed and accuracy of object detection," *arXiv preprint arXiv:2004.10934*, 2020 (cit. on p. 37).

[128] 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 (TPDS)*, vol. 23, no. 8, pp. 1369–1386, 2012 (cit. on p. 39).

[129] J. Athavale, A. Baldovin, R. Graefe, *et al.*, "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, pp. 74–77 (cit. on p. 42).

[130] J. Perez, D. Gonzalez, C. F. Nicolas, *et al.*, "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 (cit. on p. 43).

[131] 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 (cit. on p. 43).

[132] J. Ray and P. Koopman, "Efficient high Hamming distance CRCs for embedded networks," in *International Conference on Dependable Systems and Networks (DSN)*, 2006, pp. 3–12 (cit. on p. 43).

[133] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. 5th ed. Amsterdam: Morgan Kaufmann, 2011 (cit. on p. 52).

[134] Y. Ibrahim, H. Wang, and K. Adam, "Analyzing the Reliability of Convolutional Neural Networks on GPUs: GoogLeNet as a Case Study," in *International Conference on Computing and Information Technology (ICCIT-1441)*, 2020, pp. 1–6 (cit. on p. 65).

[135] T.-Y. Lin, M. Maire, S. Belongie, *et al.*, "Microsoft COCO: Common Objects in Context," in *European conference on computer vision*, Springer, 2014, pp. 740–755 (cit. on p. 65).

[136] NVIDIA Corporation & affiliates, *Parallel Thread Execution ISA*, `https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#integer-arithmetic-instructions`, Web Page, [available online: Dec-2021], 2021 (cit. on p. 67).

[137] F. Suenobu, M. Ito, and F. Kubo, "Algebras over floating point numbers," *JP Journal of Algebra, Number Theory and Applications*, vol. 31, 2013 (cit. on p. 67).

[138] S. F. J. Apostal, D. Apostal, and R. Marsh, "Improving Numerical Reproducibility of Scientific Software in Parallel Systems," in *IEEE International Conference on Electro Information Technology (EIT)*, 2020 (cit. on p. 67).

[139] N. Whitehead and A. Fit-Florea, *Floating Point and IEEE 754 Compliance for NVIDIA GPUs*, `https://docs.nvidia.com/cuda/floating-point/index.html#floating-point`, Web Page, 2021 (cit. on p. 67).

[140] "ISO/IEC/IEEE International Standard - Floating-point arithmetic," *ISO/IEC 60559:2020(E) IEEE Std 754-2019*, pp. 1–86, 2020 (cit. on p. 67).

[141] J. Perez-Cerrolaza, J. Abella, L. Kosmidis, *et al.*, "GPU Devices for Safety-critical Systems: A Survey," *ACM Comput. Surv.*, 2022 (cit. on pp. 68, 86).

[142] I. S. Olmedo, N. Capodieci, J. L. Martinez, *et al.*, "Dissecting the CUDA scheduling hierarchy: a Performance and Predictability Perspective," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2020 (cit. on p. 70).

[143] A. J. Calderón, L. Kosmidis, C. F. Nicolás, *et al.*, "GMAI: Understanding and Exploiting the Internals of GPU Resource Allocation in Critical Systems," *ACM Trans. Embed. Comput. Syst.*, vol. 19, no. 5, Article 34, 2020 (cit. on pp. 70, 91).

[144] R. Leveugle, A. Calvez, P. Maistri, *et al.*, "Statistical fault injection: Quantified error and confidence," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2009, pp. 502–506 (cit. on p. 83).

[145] Grand View Research, *Artificial Intelligence Market Size, Share & Trends Analysis Report By Solution, By Technology (Deep Learning, Machine Learning), By End-use, By Region, And Segment Forecasts*, `https://www.grandviewresearch.com/industry-analysis/artificial-intelligence-ai-market`, Web Page, available online: May-2023 (cit. on p. 104).

8 Code appendix

8.1 Sequential code

Algorithm 6 Sequential XOR

```
1: function SEQ_XOR(uint32_t ui32_prev_xor, float32_t f32_data)
2:     return ui32_prev_xor ∧= (uint32_t) *((uint32_t *) &f32_data);
3: end function
```

Algorithm 7 Sequential two's complement checksum

```
1: function SEQ_2C(uint32_t ui32_prev_twos, float32_t f32_data)
2:     ui32_prev_twos += (uint32_t) * (uint32_t *) &f32_data;
3:     return ((∼ui32_prev_twos) + 1u);
4: end function
```

Algorithm 8 Union definition

```
1: typedef union ui64_to_ui32 {
2:         uint64_t ui64;
3:         uint32_t  ui32[2u];
4: } ui64_to_ui32_t;
```

Algorithm 9 Sequential one's complement checksum

```
1: function SEQ_1C(ui64_to_ui32_t un_prev_ones, float32_t f32_data)
2:     un_prev_ones.ui64 += (uint64_t) * ((uint32_t *) &f32_data);
3:     un_prev_ones.ui32[0] += un_prev_ones.ui32[1];
4:     un_prev_ones.ui32[0] = ∼(un_prev_ones.ui32[0]);
5:     return un_prev_ones.ui64;
6: end function
```

Algorithm 10 Union definition

```
1: typedef union ui32_to_ui8 {
2:         uint32_t ui32;
3:         uint8_t  ui8[4u];
4: } ui32_to_ui8_t;
```

In Table 8.1, we gather the parameters employed in the experiments carried out during this Thesis. This particular CRC configuration is referred to as CRC32-C:

Tab. 8.1: CRC-32C (Castagnoli)

CRC look up table configuration	
Polynomial Generator	0x1EDC6F41
Initial Value	0xFFFFFFFF
Final Value	0xFFFFFFFF

We have computed the following look-up table employing the above parameters:

Tab. 8.2: CRC-32C (Castagnoli) Lookup Table

CRC Look Up Table Values							
0x00000000L	0xF26B8303L	0xE13B70F7L	0x1350F3F4L	0xC79A971FL	0x35F1141CL	0x26A1E7E8L	0xD4CA64EBL
0x8AD958CFL	0x78B2DBCCL	0x6BE22838L	0x9989AB3BL	0x4D43CFD0L	0xBF284CD3L	0xAC78BF27L	0x5E133C24L
0x105EC76FL	0xE235446CL	0xF165B798L	0x030E349BL	0xD7C45070L	0x25AFD373L	0x36FF2087L	0xC494A384L
0x9A879FA0L	0x68EC1CA3L	0x7BBCEF57L	0x89D76C54L	0x5D1D08BFL	0xAF768BBCL	0xBC267848L	0x4E4DFB4BL
0x20BD8EDEL	0xD2D60DDDL	0xC186FE29L	0x33ED7D2AL	0xE72719C1L	0x154C9AC2L	0x061C6936L	0xF477EA35L
0xAA64D611L	0x580F5512L	0x4B5FA6E6L	0xB93425E5L	0x6DFE410EL	0x9F95C20DL	0x8CC531F9L	0x7EAEB2FAL
0x30E349B1L	0xC288CAB2L	0xD1D83946L	0x23B3BA45L	0xF779DEAEL	0x05125DADL	0x1642AE59L	0xE4292D5AL
0xBA3A117EL	0x4851927DL	0x5B016189L	0xA96AE28AL	0x7DA08661L	0x8FCB0562L	0x9C9BF696L	0x6EF07595L
0x417B1DBCL	0xB3109EBFL	0xA0406D4BL	0x522BEE48L	0x86E18AA3L	0x748A09A0L	0x67DAFA54L	0x95B17957L
0xCBA24573L	0x39C9C670L	0x2A993584L	0xD8F2B687L	0x0C38D26CL	0xFE53516FL	0xED03A29BL	0x1F682198L
0x5125DAD3L	0xA34E59D0L	0xB01EAA24L	0x42752927L	0x96BF4DCCL	0x64D4CECFL	0x77843D3BL	0x85EFBE38L
0xDBFC821CL	0x2997011FL	0x3AC7F2EBL	0xC8AC71E8L	0x1C661503L	0xEE0D9600L	0xFD5D65F4L	0x0F36E6F7L
0x61C69362L	0x93AD1061L	0x80FDE395L	0x72966096L	0xA65C047DL	0x5437877EL	0x4767748AL	0xB50CF789L
0xEB1FCBADL	0x197448AEL	0x0A24BB5AL	0xF84F3859L	0x2C855CB2L	0xDEEEDFB1L	0xCDBE2C45L	0x3FD5AF46L
0x7198540DL	0x83F3D70EL	0x90A324FAL	0x62C8A7F9L	0xB602C312L	0x44694011L	0x5739B3E5L	0xA55230E6L
0xFB410CC2L	0x092A8FC1L	0x1A7A7C35L	0xE811FF36L	0x3CDB9BDDL	0xCEB018DEL	0xDDE0EB2AL	0x2F8B6829L
0x82F63B78L	0x709DB87BL	0x63CD4B8FL	0x91A6C88CL	0x456CAC67L	0xB7072F64L	0xA457DC90L	0x563C5F93L
0x082F63B7L	0xFA44E0B4L	0xE9141340L	0x1B7F9043L	0xCFB5F4A8L	0x3DDE77ABL	0x2E8E845FL	0xDCE5075CL
0x92A8FC17L	0x60C37F14L	0x73938CE0L	0x81F80FE3L	0x55326B08L	0xA759E80BL	0xB4091BFFL	0x466298FCL
0x1871A4D8L	0xEA1A27DBL	0xF94AD42FL	0x0B21572CL	0xDFEB33C7L	0x2D80B0C4L	0x3ED04330L	0xCCBBC033L
0xA24BB5A6L	0x502036A5L	0x4370C551L	0xB11B4652L	0x65D122B9L	0x97BAA1BAL	0x84EA524EL	0x7681D14DL
0x2892ED69L	0xDAF96E6AL	0xC9A99D9EL	0x3BC21E9DL	0xEF087A76L	0x1D63F975L	0x0E330A81L	0xFC588982L
0xB21572C9L	0x407EF1CAL	0x532E023EL	0xA145813DL	0x758FE5D6L	0x87E466D5L	0x94B49521L	0x66DF1622L
0x38CC2A06L	0xCAA7A905L	0xD9F75AF1L	0x2B9CD9F2L	0xFF56BD19L	0x0D3D3E1AL	0x1E6DCDEEL	0xEC064EEDL
0xC38D26C4L	0x31E6A5C7L	0x22B65633L	0xD0DDD530L	0x0417B1DBL	0xF67C32D8L	0xE52CC12CL	0x1747422FL
0x49547E0BL	0xBB3FFD08L	0xA86F0EFCL	0x5A048DFFL	0x8ECEE914L	0x7CA56A17L	0x6FF599E3L	0x9D9E1AE0L
0xD3D3E1ABL	0x21B862A8L	0x32E8915CL	0xC083125FL	0x144976B4L	0xE622F5B7L	0xF5720643L	0x07198540L
0x590AB964L	0xAB613A67L	0xB831C993L	0x4A5A4A90L	0x9E902E7BL	0x6CFBAD78L	0x7FAB5E8CL	0x8DC0DD8FL
0xE330A81AL	0x115B2B19L	0x020BD8EDL	0xF0605BEEL	0x24AA3F05L	0xD6C1BC06L	0xC5914FF2L	0x37FACCF1L
0x69E9F0D5L	0x9B8273D6L	0x88D28022L	0x7AB90321L	0xAE7367CAL	0x5C18E4C9L	0x4F48173DL	0xBD23943EL
0xF36E6F75L	0x0105EC76L	0x12551F82L	0xE03E9C81L	0x34F4F86AL	0xC69F7B69L	0xD5CF889DL	0x27A40B9EL
0x79B737BAL	0x8BDCB4B9L	0x988C474DL	0x6AE7C44EL	0xBE2DA0A5L	0x4C4623A6L	0x5F16D052L	0xAD7D5351L

Algorithm 11 Sequential Cyclic Redundant Code

```
1: function CRC32_UI32(uint32_t ui32_crc, uint32_t ui32_data)
2:     uint32_to_ui8_t u;
3:     u.ui32 ui32_data;
4:     ui32_crc = kaui32_crc_table[(ui32_crc ∧ u.ui8[0u]) & 0x00ffu] ∧ (ui32_crc >> 8u);
5:     ui32_crc = kaui32_crc_table[(ui32_crc ∧ u.ui8[1u]) & 0x00ffu] ∧ (ui32_crc >> 8u);
6:     ui32_crc = kaui32_crc_table[(ui32_crc ∧ u.ui8[2u]) & 0x00ffu] ∧ (ui32_crc >> 8u);
7:     ui32_crc = kaui32_crc_table[(ui32_crc ∧ u.ui8[3u]) & 0x00ffu] ∧ (ui32_crc >> 8u);
8:     return ui32_crc;
9: end function
```

8.2 AVX code

Algorithm 12 AVX XOR

```
1: function __AVX_XOR(__m256i m256i_prev_xor, __m256i m256i_data)
2:     return _mm256_xor_si256(m256i_prev_xor, m256_i_data);
3: end function
```

Algorithm 13 AVX two's complement checksum

```
    Auxiliar variables:
    __m256i m256i_ones = _mm256_set1_epi32(-1)
    __m256i m256i_singleOne = _mm256_set1_epi32(1)
1: function __AVX_2C(__m256i m256i_prev_twos, __m256i m256i_data)
2:     __m256i m256i_ES_twos;
3:     m256i_ES_twos = _mm256_add_epi32(m256i_prev_twos, m256i_data);
4:     m256i_ES_twos = _mm256_xor_si256(m256i_ES_twos, m256i_ones);
5:     m256i_ES_twos = _mm256_add_epi32(m256i_ES_twos, m256i_singleOne);
6:     return m256i_ES_twos;
7: end function
```

Algorithm 14 AVX one's complement checksum

```
    Auxiliar variables:
    __m256i m256i_ones = _mm256_set1_epi32(-1);
    __m256i m256i_zeros = _mm256_setzero_si256();
1: function __AVX_1C(__m256i m256i_prev_ones, __m256i m256i_data)
2:     m256i_ES_ones = _mm256_add_epi64(m256i_prev_ones, m256i_data);
3:     m256i_ES_ones = _mm256_hadd_epi32(m256i_ES_ones, m256i_ES_ones);
4:     m256i_ES_ones = _mm256_xor_si256(m256i_ES_ones, m256i_ones);
5:     m256i_ES_ones = _mm256_unpackhi_epi32(m256i_ES_ones, m256i_zeros);
6:     return m256i_ES_ones;
7: end function
```

Algorithm 15 AVX Fletcher

```
    Auxiliar variables:
    uint32_t aui32_ES_hi[4] = { 0u };
    uint32_t aui32_ES_lo[4] = { 0u };
 1: function __AVX_1C(__m128i *m128i_Flet_lo, __m128i *m128i_Flet_hi, __m256i m256i_data)
 2:     aux_128i_Flet_lo = _mm256_extractf128_si256(m256i_data, 0);
 3:     aux_128i_Flet_hi = _mm256_extractf128_si256(m256i_data, 1);
 4:     m128i_Flet_lo = _mm_add_epi32(m128i_Flet_lo, aux_128i_Flet_lo);
 5:     m128i_Flet_hi = _mm_add_epi32(m128i_Flet_hi, m128i_Flet_lo);
 6:     m128i_Flet_lo = _mm_add_epi32(m128i_Flet_lo, aux_128i_hi);
 7:     m128i_Flet_hi = _mm_add_epi32(m128i_Flet_hi, m128i_Flet_lo);
 8:     memcpy(aui32_ES_lo, &m128i_Flet_lo, sizeof(aui32_ES_lo));
 9:     memcpy(aui32_ES_hi, &m128i_Flet_hi, sizeof(aui32_ES_hi));
10:     for each 32 bit positions in a m128i data type do
11:         aui32_ES_lo[0, ···, 3] %= 65535;
12:         aui32_ES_hi[0, ···, 3] %= 65535;
13:     end for
14:     memcpy(&m128i_Flet_b_lo, aui32_ES_lo, sizeof(aui32_ES_lo));
15:     memcpy(&m128i_Flet_b_hi, aui32_ES_hi, sizeof(aui32_ES_hi));
16:     return ;
17: end function
```

Modulo operation (not implemented with AVX instructions)

Algorithm 16 AVX Cyclic Redundant Code

```
 1: function AVX_CRC32_UI32(uint32_t ui32_prev_crc, uint32_t ui32_data)
 2:     uint32_t aui32_val[8];
 3:     uint32_t ui32_crc;
 4:     memcpy(aui32_val, &b256, sizeof(aui32_val));
 5:     for each element of the array aui32_val do
 6:         ui32_crc = _mm_crc32_u32(ui32_prev_crc, (uint32_t) *(uint32_t *) &aui32_val[0, ···, 7]);
 7:     end for
 8:     return ui32_crc;
 9: end function
```

8.3 CUDA code

Algorithm 17 Two's complement checksum

```
 1: function __A2C(uint32_t ui32_a, uint32_t ui32_b)
 2:     uint32_t acc;
 3:     asm ("add.u32   %0, %1, %2; "
 4:          "not.b32   %0, %0; "
 5:          "add.u32   %0, %0, 1; "
 6:          "=r" (acc)
 7:          "=r" (ui32_a), "r"(ui32_b));
 8:     return acc;
 9: end function
```

Algorithm 18 One's complement checksum

```
1: function __A1C(uint32_t ui32_a, uint32_t ui32_b)
2:     uint32_t acc;
3:     asm ("add.cc.u32   %0, %1, %2; "
4:          "addc.u32     %0, %0, 0; "
5:          "not.b32      %0, %0; "
6:          "=r" (acc)
7:          "=r" (ui32_a), "r"(ui32_b));
8:     return acc;
9: end function
```