Artificial Neural Networks for Space and Safety-Critical Applications: Reliability Issues and Potential Solutions

Paolo Rech, *Senior Member, IEEE*

***Abstract*— Machine learning is among the greatest advancements in computer science and engineering and is today used to classify or detect objects, a key feature in autonomous vehicles. Since neural networks are heavily used in safety-critical applications, such as automotive and aerospace, their reliability must be paramount. However, the reliability evaluation of neural network systems is extremely challenging due to the complexity of the software, which is composed of hundreds of layers, and the underlying hardware, typically a parallel device or an embedded accelerator. This article reviews fundamental concepts of artificial intelligence, deep neural networks, and parallel computing device reliability. Then, the reliability studies that consider the radiation effects in the hardware, their propagation through the computing architecture, and their final impact on the software output are summarized. A detailed survey of the available strategies to measure the sensitivity of neural network frameworks and observe fault propagation is given, together with a summary of the data obtained so far. Finally, a discussion on how to use the experimental evaluation to design effective and efficient hardening solutions for artificial neural networks is provided. The available hardening solutions are critically reviewed, highlighting their benefits and drawbacks.**

***Index Terms*— Artificial intelligence (AI), convolutional neural network (CNN), deep learning (DL), deep neural network (DNN), detected unrecoverable error (DUE), EdgeAI, failures in time (FITs), fault injection, field-programmable gate array (FPGA), graphics processing unit (GPU), hardening, machine learning (ML), particle beams, silent data corruption (SDC), single-event effects, tensor processing unit (TPU).**

I. Introduction

ARTIFICIAL intelligence (AI) has changed the programming philosophy and the modern computing paradigm. Machine learning (ML) has enabled the algorithm to learn how to adapt to solve a problem [1]. An artificial neural network (ANN), depending on its complexity, is capable of solving problems that could be impossible with traditional imperative programming languages. The most interesting type of ML is deep learning (DL), in which several processing layers are used to extract progressively higher level features from the input data. The resulting deep neural networks (DNNs) allow us to find patterns in the input data, such as identifying objects in a frame [2].

DL is more and more pervasive in our daily lives, with the number of AI-based applications sharply increasing and the deployment of intelligent systems becoming ubiquitous. We count a number of novel technologies that are enabled by ML, ranging from diagnosis of malignancies to automatic predictive maintenance of industrial machines and to fully autonomous vehicles [3]. This latter technology is particularly interesting for both automotive (self-driven cars) and aerospace applications (deep space exploration). While the advantages of this trend are tautological, the potential harm due to the adoption of this technology should not be underestimated. Since AI-based applications are used to control safety-critical applications, it is fundamental to investigate their reliability and understand how to prevent failures from occurring.

The high number of operations required to execute DNNs (hundreds of matrix multiplications per layer) forces reliance on complex and high-performance parallel devices. The graphics processing unit (GPU) is one of the most widely adopted devices for accelerating the execution of DNNs. GPUs can execute several threads in parallel, highly reducing the time required for the training and inference (i.e., execution) of DNNs [4]. Lately, some dedicated accelerators for DNNs have been developed using field-programmable gate arrays (FPGAs) or specific hardware, such as low-power EdgeAI devices and tensor processing units (TPUs) [5]. These devices, similar to GPUs, have a parallel structure. However, DNN-dedicated accelerators are efficient only when executing specific operations such as convolutions and filters. Exotic solutions to overcome the inefficiency of devices built with Von Neumann architecture include neuromorphic chips that propagate neural network signals mimicking human brain synapses [6]. The common characteristic of all the hardware devices employed for the execution of neural networks is parallelism. The capability of executing several operations in parallel is essential for DNNs and not-naive networks; otherwise, the neural network execution would take an excessive amount of time. Despite the computing benefit of executing a high number of operations in parallel, when it comes to reliability parallelism has some drawbacks, in fact, a fault in particularly critical units (such as the scheduler or the control units) or shared resources (such as the caches) can impact the correctness of multiple values, leading to malfunctions [7], [8].

Recent findings indicate that transient hardware faults, such as those generated by radiation, may corrupt the ML

Manuscript received 22 October 2023; revised 14 November 2023 and 2 December 2023; accepted 2 January 2024. Date of publication 4 January 2024; date of current version 18 April 2024. This work was supported in part by the Project PERIOD of the European Union's Horizon 2020 Research and Innovation Programme under the Marie Sklodowska-Curie under Agreement 88620 and in part by the Italian Ministry for University and Research (MUR) through the "Departments of Excellence 2023-27" Program (L.232/2016) awarded to the Department of Industrial Engineering of the University of Trento.

The author is with the Department of Industrial Engineering, Università di Trento, 38123 Trento, Italy, and also with the Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre 91509-900, Brazil (e-mail: paolo.rech@unitn.it).

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TNS.2024.3349956.

Digital Object Identifier 10.1109/TNS.2024.3349956

model prediction dramatically [9], [10]. Unfortunately, some experimental data show that the radiation-induced misprediction probability can be so high as to impede a safe deployment of DNN models at scale, urging the need for efficient and effective hardening solutions. Several beam experiments have been documented, characterizing the radiation response to high-energy and thermal neutrons, heavy ions, and protons of AI accelerators [11], [12], [13], [14]. What is clear is that due to the large area and the high amount of (critical) resources available, the radiation-induced error rate of the available commercial off-the-shelf (COTS) products for AI is far from being negligible [15], [16]. Chip vendors have been improving the reliability of their products [17], eventually making them compliant with strict automotive reliability standards such as ISO9696 [18]. However, rad-hard components sufficiently powerful to execute DNNs are not available, yet. In fact, the design and/or operation overhead necessary to make the chip rad-hard is too costly for parallel accelerators. In this scenario, it is then essential to understand at which level the available COTS devices are sufficiently reliable for being adopted as part of safety-critical applications and design effective and efficient hardening solutions for DNNs. Some excellent and complete surveys of the available reliability studies have already been published [9], [10], [19]. The goal of this work is to focus on the experimental procedure and results and to provide an overview of the challenges and opportunities of testing AI accelerators.

The accurate experimental characterization of the complex hardware required to execute DNNs is challenging. The amount of resources to characterize an AI accelerator is huge, and designing dedicated benchmarks to target a specific unit is not always possible. Moreover, as mentioned, there are some hardware units that can be potentially more critical than others. A corruption in the scheduler, for instance, can impact multiple parallel processes. In addition, faults in the computing units have a not trivial effect on the operations output correctness. While a fault in memory changes one (or multiple) bits, a fault during computation modifies the output in an unpredictable way, which depends on the operation type and its input.

To make the reliability evaluation even more challenging, the software executed on the hardware to test is highly complex as well. Since DNNs are probabilistic, it is hard to predict the effect of a hardware fault on the software's correctness. In addition, the data propagation highly depends on the DNN training and the input frame. The choice of both training and input frames, then, will bias the experimental results. A naive frame or a poorly trained DNN might significantly mislead the radiation test data.

One possible way to have an accurate reliability evaluation of AI accelerators is to combine beam experiments, fault injection at different levels of abstractions, and application analysis. Beam experiments provide the realistic fault probability and fault model (i.e., how the hardware fault manifests in the software's visible state). Fault injection helps track fault propagation in the architecture or the software. Application analysis is necessary to understand the impact of the error on the system's correctness. As we will discuss in this article, the combination of these studies can help in better understanding the impact of radiation-induced faults in AI accelerators.

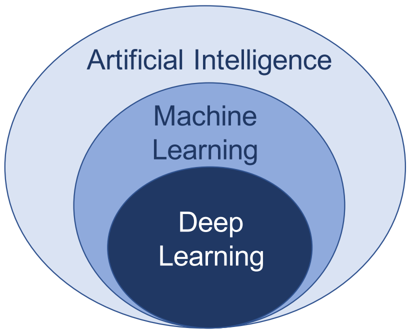

Fig. 1. Classification of AI, ML, and DL.

The understanding of fault generation, propagation, and impact on output correctness is essential to design dedicated, effective, and efficient, hardening solutions. Traditional hardening solutions, based on replication, might not be efficient for DNNs. In fact, as we will show, not all faults are critical for DNNs. A fault modifying the color of a pixel is not as critical as a fault that causes a misdetection. Replication would mask both kinds of faults, introducing possible unnecessary overhead. We will list and discuss the available dedicated hardening solutions for DNNs, from selective replication to check-sums, and fault-aware training. The available data attest that exploiting DNN's potential provides much more efficient hardening solutions than adapting to existing DNN mitigation strategies.

To provide an overview of the available hardware and software AI frameworks and to propose guidelines to perform radiation experiments and design efficient hardening solutions, this article is structured as follows. Section II reviews the basic concepts of AI and ML and details the architecture of the available AI accelerators. Section III describes possible reliability evaluation methodologies, highlighting the criticality of testing parallel hardware. Section IV presents the radiation experimental setup, while Section V describes some of the available fault injection frameworks. Section VI presents the available hardening solutions. Section VII discusses possible implications and future projections, and draws conclusions.

II. Background

This section provides the necessary background information to understand the reliability evaluation and improvement of neural networks and complex hardware systems. The information included in this section is not meant to be exhaustive but rather to provide the description of concepts and definitions used in the rest of this article. The provided citations are references useful for further investigating the discussed topics.

A. ANN Essentials

AI, as depicted in Fig. 1, is the broad definition of a program that can sense, reason, act, and adapt. Any technique that enables computers to mimic human intelligence, using logic, if-then rules, or decision trees is considered AI. ML is a subset

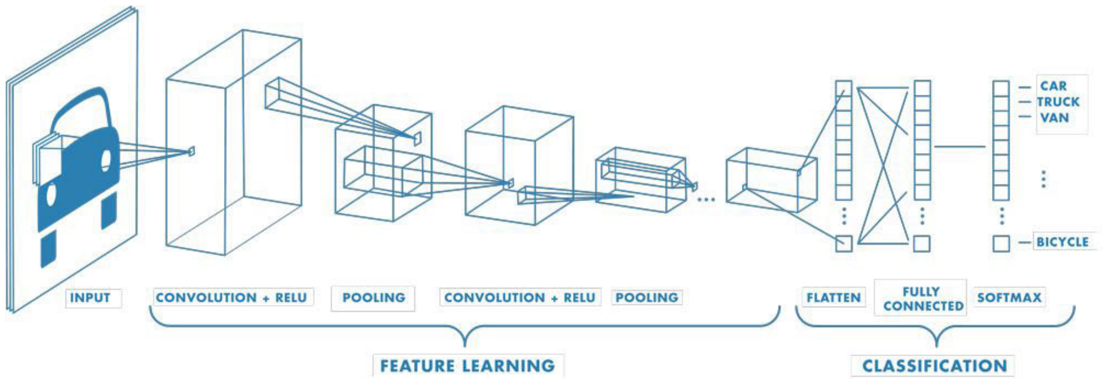

Fig. 2. Visualization of the structure of a CNN. The convolution layers extract features from the input, ReLU operations ensure nonlinear input–output correlation, pooling layers reduce the amount of data propagated, and the fully connected layers use the output of previous layers to detect and classify objects. Taken from [20].

of AI consisting of algorithms whose performance improves as they are exposed to more data over time. In other words, using statistical techniques, the algorithm *learns* how to improve the output accuracy as it processes additional data. The inner part of Fig. 1 is DL, which is that subset of AI algorithms composed of multilayer neural networks that learn from a vast amount of data.

ANNs are universal function approximators that, thanks to Backpropagation (short for backward propagation of errors) training [20], [21], [22] and sufficient network complexity, enable the solution of a variety of tasks, e.g., classification, detection, and regression. Today, most ANNs used in autonomous vehicles or, more in general, pattern recognition are composed of various or several layers and are, thus, named DNNs. A specific class of DNNs, particularly efficient in image processing (and, thus, object detection), is *convolutional* neural networks (CNNs) [20]. In a CNN, as shown in Fig. 2, most layers perform convolution, i.e., apply a filter to the image to extract features (feature maps) that are then used to detect and classify objects. As shown in Fig. 3, a filter is convolved with a window of the input image. Computationally, a filter is coded as a *kernel*, which is a matrix of values that, once convolved over the input image, extracts the information needed to perform classification or detection. The size of the window is a design choice and is typically 3 × 3 elements (normally floating point numbers). The values of the kernel decide the kind of information that can be extracted in that layer and, as discussed later, are decided in the training phase. The filters adopted to extract the most important information from the input are not intuitive and are decided in the training process of the neural networks. The choice of the convolution kernels is then made by the network itself and is not directly decided by the programmer. As shown in Fig. 4, the processed image after convolution is not easily identified by humans, but the combination of several feature maps leads the neural network to correctly classify the object.

The amount of data produced after convolution can be reduced since there is no need to maintain high resolution given that we are extracting abstract information from the images. Thus, to avoid unnecessary computation, the convolutional layers are interleaved with **pooling** layers. The pooling layers filter the data to propagate to downstream

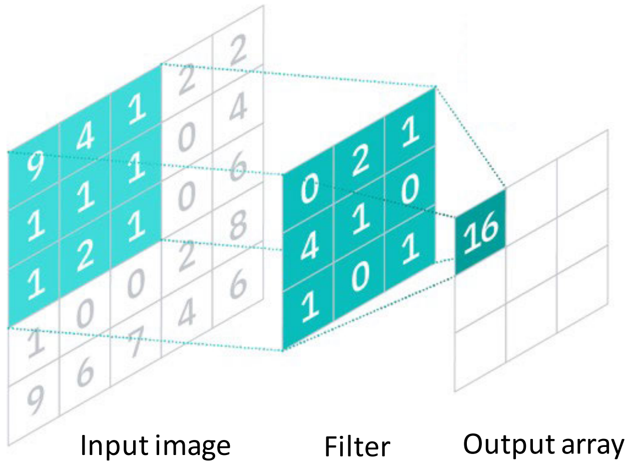

Fig. 3. Example of the convolution operation. The convolution needs to be performed in every window of the input feature map and, thus, is highly computationally demanding. Adapted from [20].

layers. The pooling can be implemented in various ways, the most common ones being max pooling and average pooling. As shown in Fig. 5, max pool propagates only the element with the highest value, while average pool propagates the average value of the elements in the window. We anticipate that, as detailed in Section VI, pooling can be significantly beneficial to filter radiation-induced errors [11].

The **design** process of DNNs consists of the identification of the number and typology of layers that, once properly interconnected, can be adapted to the specific task. The adaptation is performed through a **training** phase. The network training can be seen as the programming of the network weights. It is a computationally very expensive process, during which the network is forced to process a list of thousands of labeled images (dataset). Based on the dataset, the network will learn to identify specific classes of objects. In the training phase, the parameters of the network are modified in such a way that, for each training input–output pair, the network response is as close as possible to the ground truth. The distance measured between true and predicted values is called the loss function.

The process of tuning the network parameters to fit the dataset is called training and is usually performed via

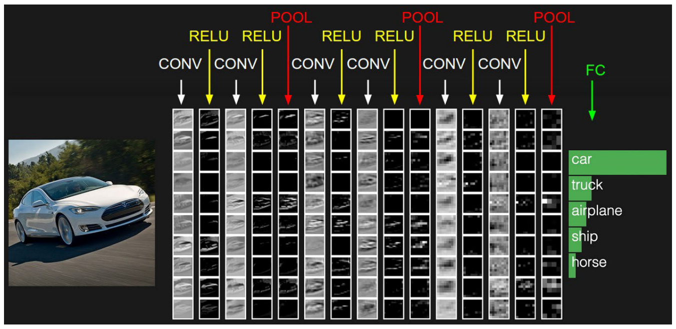

Fig. 4. Example of features extracted by the convolution layers from the input image and the effect of ReLU and pooling. Taken from [24].

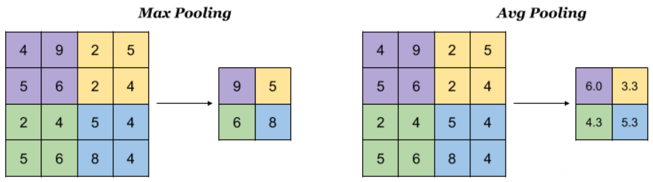

Fig. 5. Example of max pooling and average pooling. Max pooling propagates only the element with the highest value, while average pooling propagates the average value of the elements in the window.

Backpropagation of the output error from the last layer all the way back to the input one. First, a batch of data is forwarded through the network, and the output is compared with the ground-truth labels by means of a loss function, e.g., cross-entropy or L2. Since neural networks implement differentiable operations only, it is possible to compute the gradients of the loss with respect to the network weights. Given the gradients, an optimizer, e.g., stochastic gradient descent [23], updates the weights in order to minimize the loss function. These forward and backward steps are repeated for each batch of training data for a certain number of epochs (i.e., a complete pass over the whole training set). It is of the utmost importance to use as a training set a highly heterogeneous set of data because this enhances the generalization capabilities of the model. Indeed, DNNs generally suffer a significant performance drop when deployed to scenarios that they were not trained for. For this reason, it is standard practice to use heavy data augmentation strategies to obtain a more varied training set that can contain useful information not present in the original data, e.g., change light conditions if the original training set presents day scenes only. As discussed in Section VI-C, a similar approach can be efficiently applied to mitigate the effect of transient faults.

Interestingly, even if the training phase is very long and computationally demanding, the radiation impact during the training of large DNNs is unlikely to impact the final model accuracy. In fact, even if a transient fault occurs during training, its impact on the DNN parameters would be smoothed with the other (thousands) frames. Nonetheless, faults during training can impact the time required to reach convergence, and in small models, the radiation-induced fault may impact accuracy. In the following, we will focus on *inference* (i.e., the execution of a trained ANN) only. However, the design choices, together with the methods that are used to train the network, strongly impact the overall performance of the model in solving the desired task.

The network design is responsible for the expressivity and the trainability of the architecture, i.e., its capability to encode the knowledge required by the task. The training oversees an effective tune of all the network parameters. Only a judicious combination of proper techniques can result in a neural architecture capable of solving the task with good performance. In addition, as we show in this article, only a proper design/training can make the DNN intrinsically more reliable to transient faults.

Each network design has a specific set and organization of layers. **Convolutional layers** have different hyperparameters, specifically kernel size (the number of rows/columns in the convolution kernel), stride (amount of movement over the image), and padding (how many zeros to add to the input image borders to perform convolution in the edges). Each kernel is independent and produces a different feature map, with as many output feature maps as the number of filters [24].

Besides convolutions (that are layers that are most computationally demanding and, thus, vulnerable to radiation), **activation functions** are used for ensuring a nonlinear input–output relationship in DNNs and are very often implemented through rectified linear units (ReLUs) [25] $\mathrm{ReLU}(x) = \max(0, x)$, where x stands for the input tensor. This definition for activation layers enables an easy gradient flow, which is fundamental for the training [26]. Building upon this function, several other ReLU-like activations have been developed, e.g., the scaled exponential linear unit (SELU) [27], the Gaussian error linear unit (GELU) [28], and ReLU6 [29]. **Normalization layers** play a role in the stabilization of neural architecture training by smoothing the optimization landscape [30] and preventing the weight and gradient explosion. The most common normalization layer is BatchNorm that learns, at training time, an approximation of the first and second statistical moments of each feature map to normalize the input tensors. After the normalization, it is standard practice to apply an *affine transform*, namely, an additive bias β and a scale parameter γ. The normalization operation can then be defined as

$$\mathrm{BatchNorm}(x) = \frac{x - \mathbb{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta \tag{1}$$

where $\mathbb{E}[x]$ is the expected value of the input tensor x, $\mathrm{Var}[x]$ is its variance, and ϵ is a correction to improve stability [26].

The output of a DNN is a vector of *tensors* containing the probability of eligible objects. Objects identified with a sufficiently high probability are classified. The DNN, then, detects several objects with different probabilities. A filter needs to be applied to select the objects that have a sufficiently high probability of being actually detected. The probability threshold selection is a critical choice that should be carefully engineered. A threshold that is too high can lead the system to miss an existing object, exposing the vehicle to the risk of hitting it. A threshold that is too low is likely to induce the system to provide a high number of false positives (nonexisting objects are detected), leading the vehicle to unnecessary sudden stops. In object detection DNNs, a tensor also contains the coordinates of a bounding box (BB, i.e., potential object), which is then used to describe the detected object.

Given their high computational cost, DNNs and CNNs, in particular, efficiently map in parallel processors and, therefore, benefit from the heavy usage of GPUs or dedicated accelerator computation for both the training and inference processes. An overview of these computing architectures will be presented in Section II-B.

B. Accelerating Hardware for ANNs

The inference (i.e., execution) of an ANN and a DNN, in particular, is highly computationally demanding. The filters defined in the training process need to be convolved with each window of each convolutional layer to extract features. In most applications, such as autonomous vehicles, object detection needs to be performed in real time (i.e., at least 40 frames/s). This requires the use of parallel and complex hardware devices for DNN execution. In this section, we will describe the main characteristics of the devices used for DNN inference, highlighting the characteristics that can increase their vulnerability to radiation.

The complexity and performance requirements of the hardware to execute DNNs are so high as to make the design of dedicated, rad-hard, devices extremely challenging. The only available devices that can execute DNNs at speed are commercial chips, not specifically designed for safety-critical or space applications. It is then of utmost importance to measure their radiation response before adopting them in a product or a mission. As we will discuss in Section III, the difficulty of the reliability qualification of modern COTS devices for DNNs is exacerbated by the limited information available about the hardware. Usually, the information about the technology, the implementation, and the architecture is sparse, which requires reverse engineering the COTS product to investigate its reliability.

The market offers various available hardware devices that are efficient in executing DNNs. GPUs are the most common and adopted, mostly for their flexibility and the software framework availability that eases the training, design, and inference of DNNs. FPGAs or application-specific integrated circuits (ASICs) can also be adopted to implement a specific DNN, reducing the inefficiency that comes from a general-purpose device, such as the GPU. In addition, low-power and low-cost EdgeAI accelerators have been designed to improve the efficiency of DNN executions. EdgeAI devices are normally dedicated to process specific operations (convolutions, mainly) and need to be coupled with a host device that manages the DNN execution. Moreover, analog devices that resemble the human brain, as known as neuromorphic chips, have lately been tested with exciting results in terms of performance and efficiency [6]. While this novel technology is not detailed in this article, interesting reliability data [single event upset (SEU) and total ionizing dose (TID)] on neuromorphic devices are reported in [31], [32], [33], and [34].

1) Graphics Processing Units: GPUs were designed specifically to accelerate the image rastering that was incredibly inefficient using sequential CPUs. Then, GPUs have evolved from being devices dedicated to gaming, graphics, and video rendering to flexible accelerators for a variety of high-performance computing (HPC) and safety-critical applications, such as autonomous vehicles. The introduction of general-purpose programming languages for GPU [OpenCL and Compute Unified Device Architecture (CUDA)] enables programmers to exploit GPU parallelism for computation. In particular, GPUs are the reference architecture for the training and inference of DNNs, which are required to detect and classify objects in a scene. The main reasons for the success of GPUs in DNN acceleration are the high efficiency in executing matrix operations (convolution can be translated into a matrix multiplication [35], [36]) and the availability of easy-to-use frameworks to map the DNN training and inference in GPUs. This market shift led to a burst in the GPU's computing capabilities and efficiency, significant improvements in the programming frameworks and performance evaluation tools, and a sudden concern about their hardware reliability.

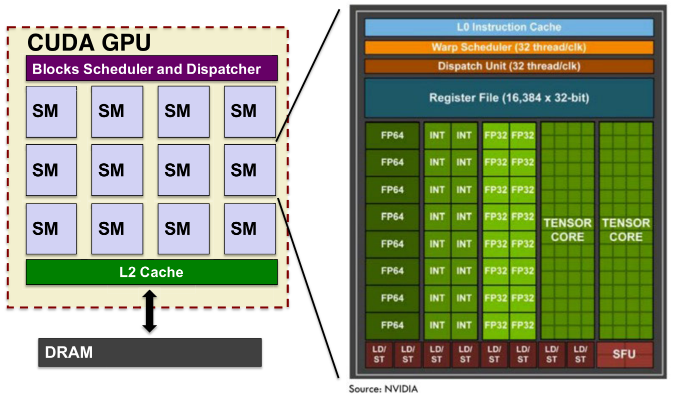
CUDA GPU
Blocks Scheduler and Dispatcher
SM
L2 Cache
DRAM
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
FP64
INT
FP32
TENSOR CORE
LD/ ST
SFU
Source: NVIDIA

Fig. 6. Simplified view of the architecture of modern GPUs that are composed of an array of SMs that share L2 cache. Each SM has several computing cores with different precisions. The view is based on NVIDIA CUDA devices but can be applied to any GPU architecture.

Modern GPUs are divided into various computing units, named streaming multiprocessors (SMs) in CUDA architecture, each of which has the ability to execute several threads in parallel [see Fig. 6 (left)]. Each basic computing unit (named CUDA core in NVIDIA devices) in the SM executes one thread with dedicated registers, avoiding complex resource sharing or the need for long pipelines [37] [see Fig. 6 (right)]. Each of the thousands of CUDA cores in a GPU disposes of hundreds of functional units of different precision (64-bit floating point, 32-bit floating point, and integer) and tensor cores, which are used to speed up convolutions. The instruction and data caches are shared among all the active parallel processes in the SM.

It is the programmer's task to divide the instantiated parallel threads into a grid of blocks when designing a kernel. It is easy to modify the thread distribution, as the block size and the grid size are both parameters that have to be specified when launching a CUDA kernel to be executed on a GPU. The number of blocks assigned to an SM in the GPU will depend on the number of registers, the amount of shared memory available in the SM, and the resources required by each block to be executed. The number of blocks assigned to an SM varies based on the architecture (hundreds of blocks can be scheduled in modern GPUs). Some blocks will be queued for later computation if the grid size exceeds the number of blocks that can be dispatched among the SMs available in the GPU. Before dispatching a queued block to the first SM that becomes available, the GPU's block scheduler needs to check if some SM completed the current block execution and, if so, it transfers the results to the onboard DDR memories. The queued block is then assigned to the SM, the input data are eventually read from the DDR, and, finally, the queued block execution is triggered and synchronized [38].

The GPU allows each SM to execute *warps* (groups) of hundreds of parallel threads in a single computing cycle. If the block size exceeds the number of available CUDA cores, the execution of some threads will be delayed until the computation of the preceding warps of the block has been completed. It is worth noting that the next block to be treated will be assigned to the SM only when all threads in the current block have been processed. Therefore, if the number of threads in a block is not a multiple of the warp size, in the last cycle, the SM will execute fewer threads, wasting parallel capabilities. The trend followed by NVIDIA is to increase the parallel capabilities of the SM more than increasing the number of SMs available in the GPU.

Each SM disposes of hardware schedulers that manage the parallelism (see Fig. 6). At every instruction issue time, the first scheduler issues one instruction for some warp with an odd ID, and the second scheduler issues one instruction for those with an even ID. When double-precision floating-point instructions have to be executed, the second scheduler cannot issue any instruction.

A parallel code to be executed on a GPU is typically composed of several independent threads, all executing the same set of instructions on a dedicated memory location. Increasing the number of threads brings then higher throughput to the application. To do so, the programmer can choose either to increase the block size, which will require more computational effort in each SM and delay the assignment of the next blocks, or to increase the grid size, thus having more blocks to be dispatched. The GPU parallel management is strictly related to the chosen thread distribution. The scheduling and computational load required for block and warp assignment, as well as resource distribution, are strictly related to the chosen grid and block sizes, which is then likely to influence also the GPU radiation response.

From a radiation test point of view, the computing units are isolated such that a single radiation-induced event in one computing unit will only corrupt the thread assigned to it. Threads that follow the corrupted one or threads assigned to computing units near the struck one will not be affected. Nonetheless, the corruption of shared resources (like the caches) or critical resources (like the schedulers) can impact the execution of several processes. In addition, the corruption of functional units can have nontrivial outcomes in the code execution. In most HPC and high-end GPUs, a single error correction double error detection (SEDEC) error correcting code (ECC) protects the main memories. As experimentally shown in Section IV, the ECC can reduce by one order of magnitude the error rate of GPUs but is not very effective in reducing the number of radiation-induced misclassifications in CNNs.

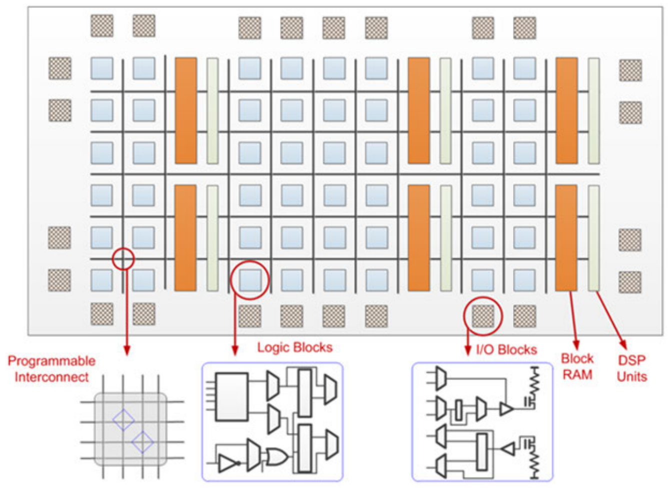

Fig. 7. Simplified view of the architecture of FPGAs, showing the programmable elements used to define a circuit to be executed [39].

2) FPGAs: FPGAs are flexible devices that allow the user to define a circuit to be executed [39]. Logic blocs, memory, and interconnections can be programmed by the user, taking advantage of synthesis tools, as shown in Fig. 7. The high number of available logic blocks allows the user to create large parallel circuits.

Due to their intrinsically high level of parallelism and number of connections, ANNs map efficiently also on FPGAs [40], [41], [42]. In addition, ANNs are also very modular, meaning that the description of a given network, using a hardware description language (HDL), becomes fairly straightforward once the neuron component has been developed. Nonetheless, while GPUs and CPUs have efficient functional units that can execute high-precision complex operations, the FPGAs are more efficient when simple operations are implemented. Thus, to map an ANN in an FPGA, some simplifications are normally adopted. For instance, the neuron function is normally described with a sigmoid, which is mathematically expressed with an exponential function as

$$\text{sigmoid}(x) = \frac{1}{1+e^{-x}}. \tag{2}$$

The sigmoid output is comprised between 0 and 1. The implementation of the exponential, for fixed point data, in HDL is not trivial and leads to inefficiencies in the FPGA. A few solutions have been proposed to efficiently implement the exponential function in hardware [43], [44]. The most adopted solution is to discretize the sigmoid, as in [45] and [46]. The higher the number of discrete steps, the higher the complexity of neurons, and the higher the precision of the ANN output. However, higher complexity is likely to increase the sensitivity of the ANN to radiation since a larger area is needed for the implementation.

An alternative solution to reduce the amount of FPGA resources to implement an ANN is to reduce the precision of data and operations. In fact, as mentioned, while CNNs can be very effective, they also require extremely high computational power. In order to achieve lower execution times and, thus, higher throughput, a number of techniques have been developed, such as weight trimming [47] and weight quantization [48]. When it comes to the latter, the main idea is to reduce the precision in which we choose to represent the trained weights (which are originally in 32-bit floating point format, as a standard for training frameworks). Such reduction can be completely arbitrary, going as far as utilizing a single bit to represent the weights in a model. These are called binary neural networks (BNNs) [49], where both the filters in the convolutional layers and the neurons in the fully connected layers use weights constrained to $\{-1, 1\}$. The adoption of BNNs instead of CNNs essentially eliminates all multiplications for a hardware implementation of a given neural network, which decreases resource utilization, but also brings down the model's accuracy.

Independently of the chosen precision and activation function, the implementation of the full ANN requires a great number of connections and processing elements (PEs). Thus, researchers have been working to reuse the resources of the FPGAs to execute various layers. This is the case of the systolic array implementation of ANNs. In the specific case of CNNs, most operations are matrix multiplication related. Matrix multiplication algorithms are inherently expensive, mostly for FPGAs. Assuming squared matrices of size N, we need to perform a total of N^3 multiply-and-accumulate (MAC) operations. Since there are no data dependencies between output elements, matrix multiplication is also extremely parallelizable, but input elements must still be read from memory multiple times, reducing efficiency, particularly on FPGAs. Systolic arrays have then been introduced to establish specific interconnection and data movement patterns between computing units to reduce memory accesses, ultimately having an edge over any other architecture in matrix multiplication computation.

A systolic array is simply a network of PEs that work together to accomplish some higher level computation. The term *systolic* is a reference to the functioning of a biological heart since the computation is performed in a rhythmic fashion, with input data being pumped in and output data being pumped out, at every clock cycle. These ideas were first introduced in [50], as the authors showed how systolic systems could be viable as application-specific hardware. In fact, depending on geometry and interconnect, systolic arrays can also be used to solve problems such as linear unit (LU) decomposition and Fourier transform.

Recently, there has been an increased interest in systolic computation due to the rise of neural networks. Since the workload of a modern CNN is dominated by convolution (which can be translated as an equivalent matrix multiplication [35], [36]) and inner product operations, weight-stationary systolic arrays became the perfect fit for these workloads. Fig. 8 is a generic illustration of a systolic array structure and gives an idea of how a matrix multiplication can be performed on it.

Note that the interconnected nature of the architecture, and the systolic pattern of dataflow, makes it so that multiple PEs are used to compute each output element. At the same time, each PE contributes to the calculation of multiple output elements. This characteristic makes systolic array implementations particularly efficient in FPGAs.

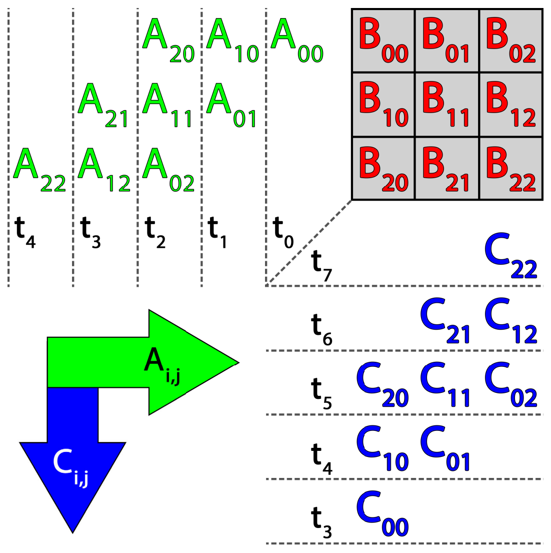
A_{20} A_{10} A_{00}
A_{21} A_{11} A_{01}
A_{22} A_{12} A_{02}
t_4 t_3 t_2 t_1 t_0
B_{00} B_{01} B_{02}
B_{10} B_{11} B_{12}
B_{20} B_{21} B_{22}
t_7 C_{22}
t_6 C_{21} C_{12}
t_5 C_{20} C_{11} C_{02}
t_4 C_{10} C_{01}
t_3 C_{00}
$A_{i,j}$
$C_{i,j}$

Fig. 8. Functioning of a generic $N \times N$ weight-stationary systolic array for matrix multiplication. The calculation in this example is $A \times B = C$. The values of B are preloaded into the array. Then, the values of A flow from left to right, while accumulations are propagated from top to bottom. The timing for inputs and outputs is specified as t_x.

The process of tailoring the ANN to an FPGA can be particularly complex and discouraging for the user who prefers easier frameworks that map on GPUs. Thus, the FPGA vendors have been working to ease the translation of the ANN, even of DNNs of great complexity, to the FPGA fabric. Modern high-level synthesis (HLS) tools, in fact, are compliant with ML developing frameworks. The user can design, train, and tune the DNN and then use HLS to translate the resulting network in the FPGA, without caring about the specific implementation. While this solution is definitely easier, it does not allow any control over the final circuit implementation and can potentially increase the challenge of evaluating its reliability.

3) EdgeAI: Lately, vendors have developed low-cost accelerators for CNN execution, named *EdgeAI* devices, such as NeuroShield or Google Coral TPUs. These EdgeAI devices are only able to execute elementary operations (i.e., convolutions and some other matrix operations) in low precision (16-bit floating point or even 8-bit integer). Coupled with a good software framework (e.g., Tensor Flow) that runs on a host device, EdgeAI devices significantly reduce the time and power consumption of the convolution, which is the most computationally demanding operation of CNNs.

Fig. 9 shows the high-level architecture of the Coral TPU, which is mainly composed of a systolic array fed by a large set of input buffers, not necessarily protected by error correction code (ECC). The array outputs the product of the model weights and each layer's input into the activation unit, where the partial sums are accumulated and the activation function is applied. Therefore, this device can perform a set of operations, mainly convolutions, which are a fundamental block for ML applications, in an extremely power- and performance-efficient

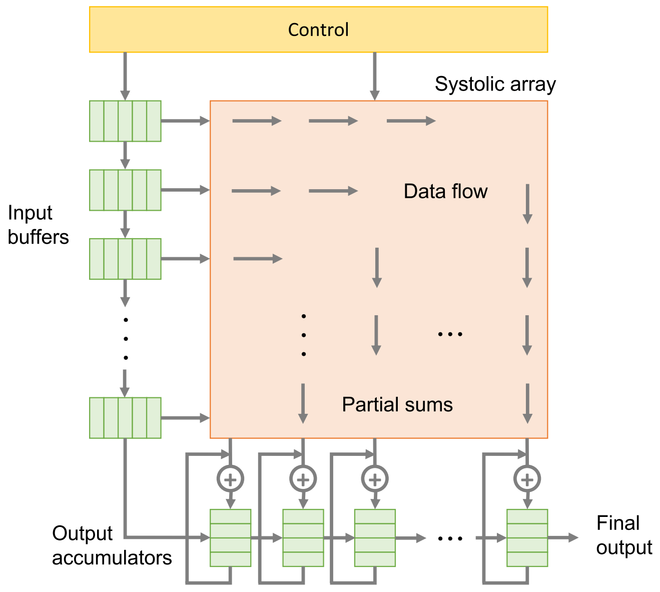

Fig. 9. High-level schematic of the Coral Edge TPU architecture that is basically composed of a hardwired systolic array. Adapted from [5].

manner, i.e., the Google Coral TPU delivers 2 tera operations per second (TOPS) per watt.

To minimize data transfers and storage and to speed up calculations, all data that are computed and stored within the TPU are represented as 8-bit unsigned integers (UINT8). The device is capable of performing the quantization and dequantization steps for interfacing with the host floating-point representations.

Since the Coral TPU is simply an accelerator, it must be connected to a host device. Google provides two versions of the accelerator: one that interfaces with the host via PCIe and the other uses USB 3.0. The software layer of the Coral TPU is based on TensorFlow Lite, which is a light version, optimized for embedded devices, of the TensorFlow framework developed by Google for ML. Most of the development effort is very similar as if the ML model would run on a normal central processing unit (CPU); however, there is an EdgeTPU compiler that is responsible for deploying the TensorFlow Lite model targeting the Coral Edge TPU architecture.

C. Radiation Effects in Computing Devices for DNNs

Radiation is a naturally occurring phenomenon. Due to the radioactive emissions of stars and major celestial events, charged ions are constantly released and gain energy as they wander around in the universe. Luckily, Earth's magnetic field acts as a shield, deviating the majority of particles (including most of the solar wind) away, but sufficiently energetic cosmic rays collide with nuclei in our atmosphere, producing a variety of secondary particles, including alphas, protons, gammas, and, mainly, neutrons. A flux of about 13 neutrons/((cm^2) $\times$ h) can reach ground. The flux exponentially increases with altitude [51].

The energy, mass, and kind of radiation (and, thus, of radiation effect) to which the device will be exposed are strictly related to the environment in which the device will operate. In this article, we will just briefly introduce the possible radiation effects based on the radiation source. For more detail about specific radiation effects, we suggest referring to the vast literature in this field [52], [53]. In particular, while cumulative, permanent, or destructive effects, such as TID or single-event latchup (SEL), are of extreme importance for the deployment of a device in a space mission, they are not considered in detail in this article. This choice is made since, while these effects are fundamental to qualify a device's reliability (mostly for space applications), the radiation response is not directly related to DNNs or to the specific architecture, but rather to the technology implementation. In other words, a device can be or not be compliant with space reliability standards for TID and SEL independently of the executed code.

Radiation is a threat to computing devices for the very basic reason that they are made out of silicon. Ionizing particles generate electron–hole pairs within the transistor's oxide, eventually releasing and depositing charge [53]. A charge can also be generated in the semiconductor material, which influences border traps [54]. The charge can accumulate, modifying the electrical characteristics of the transistor possibly reducing the operation frequency of the device or even causing a permanent failure. This is the basic issue caused by TID and is particularly critical for space applications (neutrons, which are the main radiation source at sea level, deposit negligible charge) [55].

If the particle hit deposits enough charge, it can force a transistor state to change from ON to OFF (or the other way around) [56]. Nonionizing radiation (neutrons) does not deposit any charge but, instead, hits the silicon lattice, generating secondary particles on the silicon as it passes through the device, creating charged particles (e.g., alphas) that then lead to state changes [57].

A particle strike that perturbs a transistor's state can generate bit flips in memory, activating the inverter loop, or current spikes in logic circuits that, if latched, lead to an error [58], [59]. A radiation-induced transient error in a computing device executing a code leads to: 1) no effect on the program output (i.e., the fault is masked or the corrupted data are not used); 2) a silent data corruption (SDC), i.e., an incorrect program output; or 3) a detected unrecoverable error (DUE), i.e., a program crash or device reboot.

It is worth recalling the nomenclature to avoid confusion during the discussion. In this work, we will use the taxonomy defined by Avizienis et al. [60]. When the particle deposits sufficient charge to modify the state of a transistor, a *fault* is generated. The fault can be masked or propagated to a visible state, becoming an *error* (see Fig. 10). A visible state can be a register or a flip-flop. If the corrupted visible state is used for computation and the error further propagates till the software output, it becomes a *failure*. SDCs or DUEs are, then, failures.

The advances in fabrication processes and overall scaling of technology have allowed for reduced transistor sizes, increased transistor density, and reduced operating voltages. Given the shrinking dimensions of CMOS transistors, the pursuit of lower power consumption, and the integration of several resources in a single chip, the probability of neutron-induced

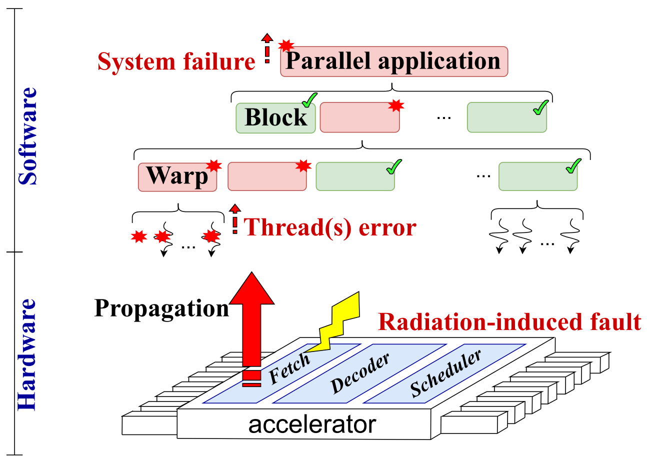

Fig. 10. Possible effect of a single-event effect corrupting critical or shared resources of a parallel device. As a result of the impact, multiple parallel processes can produce a wrong output.

faults, in both memory and logic resources, has increased significantly [53].

Since the hardware accelerators required to execute at speed DNNs have a large area with high availability of computing resources, they are particularly likely to experience radiation-induced faults. In addition, the hardware parallelism management and control units of parallel computing systems are particularly critical since their corruption affects multiple parallel processes [7]. As shown in Fig. 10, when a single-event effect occurs in the fetch, decode, scheduler, or shared resources, it is possible that multiple software threads (managed by the struck resource or using the shared resource) produce wrong output. What makes the reliability evaluation of complex hardware particularly challenging is the fact that, while the fault occurs in the hardware, it can only be observed once it reaches the software output. As discussed in Section III, the combination of dedicated benchmarks and fault injection is necessary to understand the sources of the observed errors. When performing a beam experiment, it is important to log all the necessary information from the output to reconstruct the error model. A simple correct/wrong flag might not be sufficient.

Single versus multiple corruptions are particularly critical in DNNs and image processing algorithms in general. Intuitively, a single thread corruption can be seen as a single wrong pixel, while multiple thread corruptions can lead to the wrong computation of several pixels. In the former case, a misdetection is unlikely, while, in the latter, the probability of missing an object or having a false positive increases. The peculiar parallelism of hardware accelerators, which provides unquestionable benefit in terms of performance, is, then, one of the most vulnerable parts of the device in terms of reliability.

In the specific case of DNNs, the fault propagation from the corrupted transistor to the detection/classification output is extremely hard to track. The DNN output is, by itself, probabilistic (details in Section II-A) and not deterministic as in most computing. A fault can modify the low probabilities that are, in any case, not selected in the final detection/classification or can completely change the output vector. Despite the applied filters (pooling; see Section II-A) and the redundancy

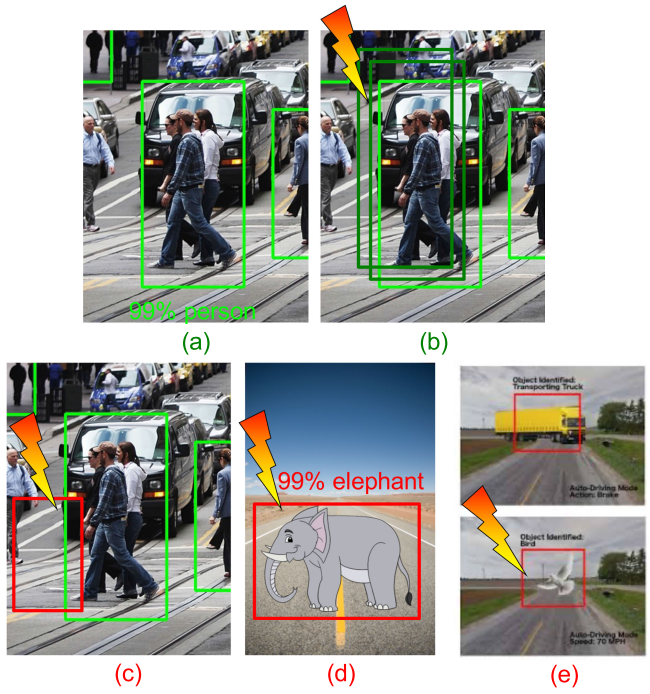
99% person
(a)
(b)
(c)
99% elephant
(d)
Object Identified: Transporting Truck
Auto-Driving Mode Action: Brake
Object Identified: Bird
Auto-Driving Mode Speed: 70 MPH
(e)

Fig. 11. Example of (a) expected output of a classification DNN, (b) tolerable errors (more than 50% of the original object is still detected in the BB, (c) misdetection (the existing object is not detected, thus reducing recall), (d) false positive (a nonexisting object is identified, thus reducing precision), and (e) classification error (misclassification).

in computation, it is unquestionable that transient faults can propagate through DNNs. Luckily, not all the SDCs are critical in object detection/classification frameworks. If the SDC (then the radiation-induced fault reaches the DNN output) does not impact detection and classification, the SDC could be considered tolerable. The question is how to distinguish between tolerable and critical error.

As shown graphically in Fig. 11, SDCs that modify the object probability (the tensor output; see Section II-A) with respect to the expected output [see Fig. 11(a)] such that they do not impact an object's rank or, for a detection framework, change the coordinates of a low-probability (BB, i.e., the object coordinates in the frame), are not considered critical. Errors that only slightly modify the coordinates of an object, still allowing a sufficiently good detection for not modifying the vehicle behavior, are also to be considered tolerable [see Fig. 11(b)]. On the contrary, SDCs that induce a misdetection [see Fig. 11(c)], a false positive [see Fig. 11(d)], or a misclassification [see Fig. 11(e)] are to be considered critical.

One way to distinguish between critical and tolerable errors is to measure the precision and recall of the corrupted output. Recall is the fraction of existing objects that were detected (or classified), even in the event of a radiation-induced error (Recall < 1 means that some objects were not detected). Precision measures the fraction of the detections produced by the classifier that actually relate to an existing object (Precision < 1 means that some not existing objects were detected, i.e., a false positive occurred). To distinguish between critical and tolerable SDCs for classification, we should consider just the probability vector. When an SDC modifies an object's rank in a frame, it is necessary to count the number

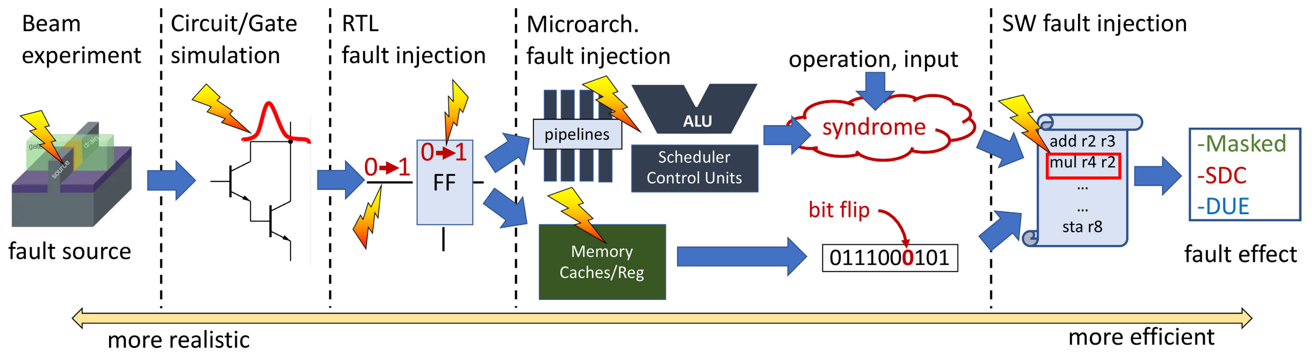

Fig. 12. Fault propagation and reliability evaluation methodologies in complex computing devices. The fault originates in the physical transistor and then propagates to the circuit, the microarchitecture, and eventually to the software output. Beam experiments evaluate the probability for a fault in the physical layer to be generated and propagated till the software output. Fault injection at different levels of abstraction can be used to better understand fault propagation. Evaluations closer to the physical layer are more realistic, while evaluations closer to the software layer are more efficient.

of objects in the fault-free execution, which are not correctly classified (i.e., false negatives, reducing recall) and how many objects that should not be classified appear in the corrupted eligible objects list (i.e., false positives, reducing precision).

A correct classification is not sufficient to guarantee correct detection; it is also necessary to consider the position of the object. To evaluate error criticality for detection, it is necessary to consider the BB's area, adopting the methodology used in the image processing community [61]. We can consider an object i in a radiation-corrupted output as correctly detected if, for any object j of the radiation-free output, the following condition on the Jaccard distance (a measure of how dissimilar two sets are) can be verified:

$$\text{Jaccard}(i, j) > T_J \tag{3}$$

where T_J is the acceptance threshold. Otherwise, we can consider i as a false positive [reducing precision; see Fig. 11(d)]. If, for a given object j of the radiation-free output, there is no BB i in the corrupted output, which satisfies this condition, a false negative is detected [reducing recall; see Fig. 11(d)]. Some studies also show that transient faults can result in a correct detection (the BB area and position are correct) but in a wrong classification [62], as depicted in Fig. 11(e).

In the image processing community, T_J is an arbitrary threshold, with $0 \leq T_J \leq 1$. Values of T_J close to 1 force the classifier to be very precise. In a reliability context, $T_J = 1$ imposes any radiation-induced modification to BB coordinates to be marked as critical. We adopt $T_J = 0.5$ to compare a CNN's corrupted output with the radiation-free output, following the T_J tradeoff discussion presented by Fawcett [61], which is valid, independent of the source of detection imprecision (intrinsic algorithm detection imprecision or radiation-induced corruption).

In Section III, we will review the possible methodologies adopted to investigate the reliability of DNN hardware and software.

III. Reliability Evaluation Methodologies

As discussed in Section II-C, when evaluating the reliability of complex computing devices executing DNNs, we need to consider that the radiation-induced fault occurs in the physical transistor and then, possibly, propagates through the (parallel) architecture, reaching the software and eventually modifying the output. As shown in Fig. 12 and listed in Table I, there are various reliability evaluation methodologies available to understand fault propagation in computing devices, from the gate level to the architectural level and the system level. It is worth noting that memory errors are the easiest to model at the software layer. Since the fault simply flips a (or multiple) bits in the memory word(s), to model the fault, it is sufficient to modify the value in software variables. With a simple static memory beam test, it is possible to understand the number of bits to be corrupted. However, when it comes to faults in computing resources (such as the pipelines, the control units, functional units, or the scheduler), the impact on the software is not trivial. The operation output is corrupted with a syndrome that depends on the operation executed and its input. If the fault impacts shared or critical resources, multiple operations can be corrupted. A software fault-injection, then, needs to be carefully engineered so as not to have unrealistic results.

Each evaluation methodology has some benefits and limitations, which will be detailed next. We also discuss why the complexity of hardware accelerators exacerbates the limitations associated with the available methodologies. In general, methodologies that act closer to the fault's physical source (i.e., the silicon implementation) are more realistic (and costly in terms of processing time), while methodologies closer to the output manifestation of the fault are more efficient (but less realistic in terms of the fault effect in real applications). In the following, we give an overview of all the available reliability evaluation methodologies for focusing on the two most widely adopted ones (beam experiments and fault injection). Further details about the testability and dependability of AI hardware can be found in [9] and [10].

Field test studies expose the computing device to the natural flux of particles, counting the number of observed errors. While field tests are probably the most accurate and realistic way to measure the sensitivity of a device, they have to be based on statistically significant amounts of data and, thus, require a huge number of devices and, obviously, are very time-consuming (because the natural error rate is very low) [63], [64], [65].

TABLE I
RELIABILITY EVALUATION METHODOLOGY CHARACTERISTICS

Evaluation Method	Time Needed	Cost	Accessible Resources	Fault Source	Availability	Observability
Field, Lifetime data [63]–[65]	months/years	very high	all	natural	final product	limited
Beam testing [52], [53], [66], [67]–[80]	hours	high	all	natural	final product	limited
RTL fault injection [82]–[86]	years	low	all	synthetic	late	very high
Microarch. fault inj. [82], [83], [87]–[91]	days/weeks	low	most	synthetic	early	very high
Arch. fault injection [92], [94]	days	low	limited	synthetic	early	medium
Software fault injection [70], [94]–[100]	hours	low	limited	synthetic	early/final product	medium

Beam experiments induce faults directly in the transistors by the interaction of accelerated particles with the silicon lattice, providing highly realistic error rates [53]. Accelerated particle beams reduce the cost and time of field tests taking advantage of a high particle flux intensity [53], [66], [67]. Since errors are observed only when they appear at the output, generally, beam experiments do not allow tracking fault propagation. This prevents one from associating observed behaviors with the fault source and, thus, identifying the most vulnerable device resources. In addition, results are valid only for the specific codes and configurations that have been tested. On parallel devices, the number of configurations that should be tested increases significantly. For instance, even a slight change in the code's degree of parallelism can impact the code error rate [7].

Software fault injection is performed at the highest level of abstraction, and it was proven efficient in identifying those code portions that, once corrupted, are more likely to affect computation [68], [69], [70], [71], [72], [73]. However, the analysis is limited as faults can be injected only on that subset of resources, which is visible to the programmer. Unfortunately, critical resources for highly parallel devices (i.e., hardware scheduler, threads control units, and so on) are not accessible to the programmer and, thus, cannot be characterized via high-level fault injection. In addition, the adopted fault model (typically single-/double-bit flip) might be accurate for the main memory structures (register files, caches) but risks being unrealistic when considering faults in the computing cores or control logic, as also shown in [74]. In fact, as shown in Fig. 12, while a fault in the memory array directly translates into a corrupted value, the single transient fault in a resource used for the execution of an operation (pipelines, arithmetic logic unit (ALU), scheduler, and so on) can have not-obvious effects on the operation output. We call this not-obvious effect a *syndrome*. The syndrome induced in the instruction output by faults in the computing core depends on the operation, its input, and the corrupted resource. The only possible way to find this syndrome is to perform lower level fault injection or dedicated beam experiments targeting a single operation or functional unit.

Microarchitecture fault injection provides a higher fault coverage than software fault injection as faults can, in principle, be injected in most modules. A preliminary work, based on Multi2Sim, presented microarchitectural fault injection data on GPUs, but the analysis is limited to just memories [75]. One of the issues of microarchitectural fault injection in AI accelerators is that the description of some modules (including the scheduler and pipelines) is behavioral and their implementation is not necessarily similar to the realistic one. Recent work has demonstrated that microarchitectural fault injection provides a sufficiently accurate reliability evaluation on ARM-embedded CPUs [76]. On AI parallel accelerators, such a demonstration is still missing and is likely to be more challenging due to the complexity of the hardware underneath the microarchitecture.

Register-transfer level (RTL) fault injection accesses all resources (flip-flops and signals) and provides a more realistic fault model, given the proximity of the RTL description with the actual implementation of the final hardware [74], [77], [78]. However, the time required to inject a statistically significant number of faults makes RTL injections impractical. The huge amount of modules and units in a complex parallel accelerator and the complexity of modern HPC and safety-critical applications exacerbate the time needed to have an exhaustive RTL fault injection (hundreds of hours for small codes), making it unfeasible. Previous work that evaluates GPU reliability through RTL fault injection is limited to naive benchmarks [78].

Circuit- or gate-level simulations induce analog current spikes or digital faults in the lowest abstraction level that still allows tracking fault propagation (not available with beam tests). There are two main issues with the level of detail required to perform this analysis on AI accelerators: 1) a circuit- or gate-level description of the device is not publicly available and 2) even if it was, the time required to evaluate the whole circuit would definitely be excessive (the characterization of a small circuit takes weeks [79]).

Hybrid or combined fault injections at different levels of abstraction have been adopted to increase the reliability evaluation efficiency without jeopardizing its accuracy. Some works have proposed to use a detailed RTL fault injection in specific portions of the circuit and a fast fault simulation in others [77], [101]. Recent works combined an extremely detailed gate-level fault injection in tandem with a faster (but still impracticable for complex devices) RTL evaluation [79], [102]. Cho et al. [103] used high-level simulation (not using real hardware) triggering an RTL model when the fault needs to be injected. Subasi et al. [74] focus on RTL injection to provide a more detailed fault model but are limited to embedded processors ALU. Recent studies have also proposed to combine software fault injection with beam experiments [104], [105] or RTL and software fault injection [8] showing that accurate details about fault propagation in the system can be achieved.

A. Radiation Experiments

Accelerated beam experiments are the primary way to accurately measure the radiation sensitivity of a device. This also applies to AI accelerators, with a possible more complex setup and a higher risk of designing a less-than-optimal setup, wasting beam time. The goal of this subsection is to provide some suggestions and advise, based on the author's experience, on how to prepare a good radiation experiment setup.

The first choice to make regards the facility to select for the experiments. This choice is clearly dictated by the radiation you want to simulate and the characteristics of the facility and the available beam.

For TID studies, following the European Space Agency test standard [106], the device should be exposed to Cobalt-60 sources or, eventually, X-rays [107]. In the latter case, it is essential to ensure that the energy deposition of the X-rays actually occurs in the active region of the device [108]. As mentioned earlier, in this article, we will mainly focus on single-event effects since the TID response does not depend on the executed code or the device architecture but mainly on the device technology [55].

Single-event effects evaluation can be done for space or terrestrial applications. Given the importance of AI reliability for safety-critical terrestrial applications (e.g., self-driving cars), it is fundamental to measure the error rate of AI accelerators and also for the Earth's radioactive environment. Space reliability studies require the test of the device with heavy ions and/or protons, while terrestrial reliability studies require the test of the device with neutrons.

There are various facilities available for the test of electronic components with heavy ions. Each facility provides a cocktail of ions with different energies. The first choice to make regards the sample preparation [109], [110], [111]. COTS devices have plastic or ceramic packages that eventually should be removed to allow the interaction between the ions and the silicon active area. Moreover, most of the available AI accelerators are flip-chip, which means that the silicon substrate possibly needs to be thinned. This choice normally needs to be made based on a device's physical study since the information about the package and the silicon thickness of COTS AI accelerators is sparse. To measure the thickness of the device, it is then necessary to cut the device and look at its cross section. Such a procedure can be particularly challenging for high-end GPUs that are installed in complex (and thick) printed circuit boards (PCBs).

Another important issue to consider when testing power-hungry devices with heavy ions is the cooling. Removing the package (and, consequently, the built-in fan and thermal heat sink) and thinning the substrate significantly reduce the chip's ability to dissipate heat. This issue is exacerbated in the facilities that require operating the device under test in a vacuum. Edward J. Wyrwas, NASA Goddard, designed and presented an interesting cooling system, based on thermoelectric cooling plates, which is effective in dissipating heat in modern GPU during a heavy ion test [112].

Most of the neutron/heavy ion/proton beams are pulsed. This means that particles are not delivered continuously, but packets of particles are produced with a given frequency. When testing codes with a critical timing issue, such as ANNs, it is fundamental to ensure that the particle production frequency is much higher than the operation execution frequency or that the execution is not synchronized with the beam. Otherwise, there is a risk not to have all the operations executed, while the device is hit by particles or to have the particle hitting the device always on a specific portion of the execution. State-of-the-art ANNs normally process 40 frames/s, which can be comparable with the frequency of particle production in some facilities (10–100 Hz at ISIS). Nonetheless, the time required to start the inference depends on the frame since the time to load the frame depends on its size and on the state of memory (caches are free or not). Thus, it is highly unlikely to have the execution and particle production synchronized, intrinsically reducing the timing issue.

It is worth noting that most of the AI accelerators require a host device to be controlled. This host device can be a motherboard (for stand-alone GPU), a Raspberry Pi, or other low-power systems (for EdgeAI). The radiation experiment setup needs to consider the interface between the host and the device under test. The former should be placed in a position sufficiently far from the beam to consider negligible the probability of having a radiation-induced corruption on the controlling hardware. If the device under test needs to operate in a vacuum, the interface to control it could be complex to be designed. Modern stand-alone GPUs are controlled from a host device through a high-speed PCI Express interface, EdgeAI devices are controlled via USB 3.0 or higher, and the Systems on Chip (SoCs) that embed an accelerator can be controlled via Ethernet. All these communication interfaces need specific cables and plugs in the vacuum chamber, which are not necessarily available.

What is challenging in the radiation experiment of complex devices executing ML applications is to decide what to log and to automatize the detection of crashes or system hangs. The details of a possible software and hardware setup, adopted in various radiation test campaigns, are presented in Section IV.

B. Fault Injection

One of the limitations of beam experiments is that the radiation-induced fault is observed only when it reaches the software output. This limitation is actually not an issue if the test targets memory. In fact, as depicted in Fig. 10, a fault in the hardware that implements a memory bit simply manifests as a bit flip. Thus, with a static (or dynamic) memory test, we can measure the probability of faults and the fault model (how many bits are corrupted). This does not apply to logic and computing resources. A fault in an adder, for instance, can have a syndrome that depends on the stuck register or on the inputs. If the add operation is used as part of a complex algorithm (such as DNNs), this error can propagate (or not) till the output and can (or not) modify the object detection.

To better understand how a fault propagates in the computing device, it is necessary to rely on fault injection that, as shown in Fig. 12 and listed in Table I, can be performed at different levels of abstraction (circuit, RTL, microarchitecture, and software). The lower the abstraction layer (i.e., the fault is injected closer to the physical implementation), the higher the accuracy of the results, but the higher the computation time required to perform the evaluation.

RTL fault injection requires the gate-level description of the computing device. This is, unfortunately, a sheer illusion for

TABLE II
RELIABILITY EVALUATION METRICS DEFINITIONS

Metric	Unit	Definition	Calculation
Cross Section	cm^2	circuit area that, if hit by a particle, generate an error	errors/fluence
FIT	$errors/10^9h$	expected errors in 10^9h of operation	cross section × natural flux
AVF	%	percentage of hardware faults that propagate to the software output	errors/injected faults
PVF	%	percentage of software errors that propagate to the code output	errors/injected faults
MTBF	h	length of the time window of correct operation	1/error rate
MFBF	$particles/cm^2$	fluence to have a failure	average fluence
MIBF	number of inst.	number of correct instructions executed	MTBF/number of instructions
MEBF	number of exec.	number of correct executions completed	MTBF/exec. time
MWBF	data	amount of data correctly produced	MEBF × workload

COTS AI accelerators. To protect business-critical information, in fact, the vendor does not release the internal circuit description, thus preventing the RTL fault injection. In fact, an RTL description of a GPU core exists [113] but is based on an old NVIDIA architecture (G80). Nonetheless, as discussed in Section V, this RTL description was fruitfully used to understand the effects of faults in peculiar GPU resources, such as the scheduler and control units.

Microarchitectural fault injection modifies the values of signals in the description of the computing core. This level of abstraction does not include the implementation details of the RTL circuit, and often, it only disposes of behavioral descriptions. A fault injection in the microarchitecture, then, allows studying how a fault in a computing resource (such as the pipelines, the scheduler, and the memories) impacts the software execution. It is clear that, without the implementation details of the computing resources, it is impossible to predict the error rate of the device. Nonetheless, some preliminary studies showed that the fault injection at the microarchitecture level of ARM devices can predict the radiation-induced error rate of a code, given some information about the technology sensitivity [114]. As said for RTL fault injection, also, the microarchitectural fault injection requires the availability of the microarchitecture details of the computing device, which is, once again, not easy to get.

Software fault injection is the easiest possible fault injection. The experiments are performed on live hardware and rely on dedicated procedures to modify the values of variables or instruction output. Since the experiments are performed on the COTS hardware, there is no need to have details about the device. However, software fault injection risks being unrealistic if not properly designed. Software fault injection is normally used to understand critical software operations or algorithm procedures more than to estimate the error rate of the device. When performing a software fault injection, as detailed in Section V, it is important to profile the code executed. This step allows counting the number and type of instructions and the input used by the code. A statistical fault injection should target the various instructions and the final result normalized with the probability for an instruction to be actually executed [105].

C. Evaluation Metrics

There are various metrics to measure and describe the reliability of a computing system running DNNs. Table II lists the most common metrics, their measurement unit, definition, and an indication of how they are calculated. Other metrics, derived from the automotive and safety-critical domains and their evaluation, are discussed in [115].

The primary metric to measure the radiation sensitivity of a device is the **cross section**. The cross section measures the probability for a particle to generate a *fault* in the device. In the case of computing devices, the fault is observed when it becomes an *error*, which can be a DUE or an SDC, i.e., the fault must be generated in the hardware layer and propagate until being observed. Thus, the faults that are masked are not counted. The higher the complexity of the hardware and software, the harder to understand how many faults have not been counted. When it comes to DNNs, the SDC can be critical or tolerable (as discussed in Section II-A). It is then a good practice to distinguish the critical and tolerable SDC cross section. Please note that, given the dependence of the fault propagation through the abstraction stack of the device and the software, the cross section can vary significantly depending on the executed DNN and even on the processed frame (an empty frame can be less susceptible than a busy frame).

In the case of automotive, avionics, or industrial applications, the metric to evaluate the reliability of a device (running an application) is the **failure in time** (FIT), i.e., failures in 10^9 h of operation. Depending on the criticality of the action performed by the device, there are different maximum FITs allowed. In the case of automotive applications, according to ISO2626-2, devices used in the Automotive Safety Integrity Level (ASIL) D level (the most strict level) must have an error rate lower than 10 FITs [18]. If the (neutron) cross section has been measured by an experimental beam with a spectrum of energies that resemble the natural one, the FIT rate can be measured by multiplying the cross section by the natural flux (13 n/cm^2/h at the sea level [51]) and by 10^9. The **mean time between failure** (MTBF) indicates the average time during which the device is correctly operating. By definition, the MTBF is simply the inverse of the error rate (i.e., of the FIT). Alternatively, one can use the **mean fluence between failure** (MFBF) that indicates the average fluence necessary for the device to experience an error [116].

The cross section of a device executing a code depends on the number of resources used for computation, i.e., the sensitive area or *cross section* [53], and their corruption probability of affecting the output. The probability for a fault to propagate from the transistor to the software visible state (see Fig. 12) is called **architectural vulnerability factor** (AVF) [117]. The AVF assumes that a fault has occurred and measures its probability of reaching a register, a memory, and an operation output. When a code is running, the probability for an error to affect the output is the **program vulnerability factor** (PVF) [118]. The PVF assumes that the fault has reached the software's visible state (thus becoming an error) and measures its probability to modify the software execution (thus becoming a failure, such as an SDC or DUE). The AVF and PVF are measured with fault injection, which tracks fault propagation to the output. Fault injection, then, assumes that a fault occurred and identifies if the fault affects the output or not.

The cross section does not depend on the execution time. The cross section, in fact, is measured, with accelerated beam experiments, dividing the number of observed application output errors by the particle *fluence* (n/cm^2). The fluence is given per cm^2 to ease the calculation, as measuring the exact area of the transistors/device would be unfeasible. This implies that, when testing a code, the execution time and the performance are not considered. The impact on some applications is that a very slow code that uses only few resources for a long time could be evaluated more reliably than a highly efficient code that uses a lot of resources to speed up the execution. In other words, what influences the cross section is the amount of resources used for computation and not the execution time. In fact, if the same amount of memory is exposed for a time interval t or $2t$, its cross section will not change since, under a constant flux (neutrons/cm^2/sec), in $2t$, we expect twice the errors and a double fluence (n/cm^2) to hit the device, resulting in the same error rate. This is under the assumption that the flux of particles is constant and sufficiently low not to have two corruptions from two different particles in $2t$. On the contrary, if we double the amount of memory exposed for the same time interval t, we expect twice the number of errors but the same neutron fluence (n/cm^2) to hit the device: we are doubling the error rate. Similarly, executing x or $2x$ *sequential* (independent, for simplicity) instructions does not change the code cross section. On the contrary, if the additional x instructions are executed in *parallel* with the original sequence, the error rate is expected to double (same execution time and same fluence but doubled error rate).

The interesting aspect, only apparently playing against parallel devices, is that a slower execution *does not* increase the FIT rate, while using more parallel resources or bigger hardware cores, with a potential benefit on performance, increases the FIT rate. To combine error rate and performance, the mean instructions, executions, or work between failure metrics (MIBF, MEBF, and MWBF) were introduced [7], [119], [120]. The idea is to consider how many instructions, executions, or workloads can be correctly completed before the output error occurrence. Considering a constant error rate, then the faster configuration will have a higher MIBF, MEBF, or MWBF. It is clear that these metrics are useful only if what is important for the mission is the amount of data correctly produced. If what matters is simply the error rate, then the cross section is the main metric to use.

IV. Radiation Experiments

In this section, we will review the requirements of a radiation test setup for AI accelerators. We will consider

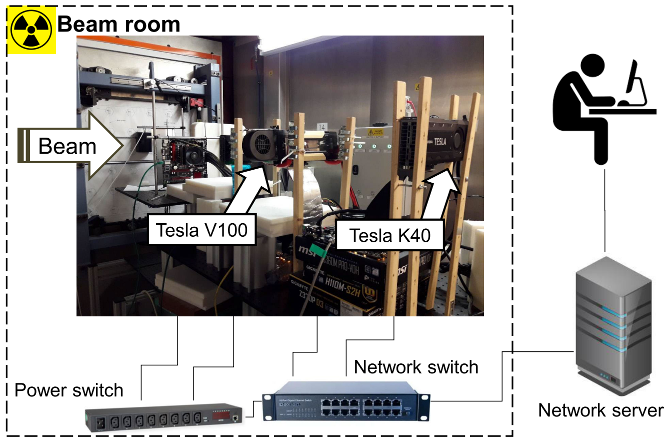

Fig. 13. Scheme of a beam experiment setup mounted at the ChipIR facility at Rutherford Appleton Laboratory in the U.K. The devices under test are placed in the irradiation room and aligned with the neutron beam of $\approx 3.5 \times 10^6$ n/(cm^2/s). The DUTs (GPUs) are controlled by the host motherboard, which is connected to a private network. The power given to the motherboards is controlled with an Ethernet-controlled power switch. The network server is composed of a set of Python scripts, which controls the devices through the Ethernet and performs power cycles through a power switch.

both the hardware and software setups, including the choice of the algorithm, the code, the input, and the output to check and log. All the presented discussion is the result of several years of beam experiments on AI accelerators, and thus, it is not meant to be theoretical but practical. The test setup that we present in this article, including the scripts to generate the logs, several benchmarks, the watchdog, and some easy examples, is publicly available in the UFRGS CAROL radiation benchmarks repository [121].

A. Experimental Setup

Most of the available AI accelerators require a host to operate. A host can be a motherboard, for GPUs, a PC or Raspberry Pi, for EdgeAI devices, or an embedded CPU, for SoCs. The host has the role of stimulating and controlling the AI accelerator but also of checking the output correctness. The host itself should then be controlled by a server that controls the entire experiment, tracks the test being performed, maintains a time stamp, counts errors, and determines, using watchdogs, if the host or the device under test needs to be rebooted or power-cycled. Thus, an experimental setup for AI accelerators, as shown in Fig. 13, is composed of: 1) the device under test; 2) the host; and 3) the server.

Normally, while the DUT should be aligned with the particle beam, the host and the server should be placed as far as possible from the beam spot. As shown in Fig. 13, in the case of GPUs that need to be connected to a motherboard using the PCIe interface, we use PCIe bus extensions to increase the distance between the DUT (thus the beam spot) and the host device. We also include boron plastic bricks to protect the host devices and the power sources from scattered neutrons. We suggest using high-end devices for the host device (server motherboards, DDR memories with ECC, gold power sources, and so on) since the stress imposed by a beam experiment, together with the effect of neutrons, can jeopardize the functionality of the controlling hardware. According to our experience, the power source is particularly critical when testing power-hungry GPUs. Moreover, the DDR memories are highly susceptible to scattering neutrons. Unfortunately, after hours of testing, it is not unlikely to experience permanent or intermittent errors in the DDR. These errors corrupt the input/output vectors in the host, which are used for the experiment, making the test output completely useless. When an error occurs in the host DDR, we expect most (if not all) of the executions to generate an error. These errors should obviously not be considered when calculating the DUT cross section. The same applies to the host hard drive. It is fundamental to protect it from scattered neutrons and avoid excessive writing operations. An error in the host DDR, for instance, can have the side effect of inducing several subsequent log (write) operations in the hard drive, either saturating the available space or inducing a disk failure.

The setup should include both software and hardware **watchdogs** to monitor the experiments (see Fig. 13). These watchdogs are used to monitor the experiment and automatically detect when the application or the DUT stops responding. In these events, the watchdog should be able to relaunch the application, reboot the host, or power cycle the system. The watchdogs are particularly useful for overnight shifts of experiments when multiple boards are tested in parallel so as to quickly detect malfunctioning, thus improving the accuracy of the measurement and reducing the waste of beam time.

The *software* watchdog is executed either on the host (the motherboard) or the server (external control system). The software watchdog is a script that checks if the application under test is running, and if it stops responding in a predefined time interval, the kernel is killed and relaunched. This watchdog detects kernel crashes or software hangs, i.e., application crashes or control flow errors that prevent the DUT from completing assigned tasks (e.g., an infinite loop). The best way to implement the software watchdog is to let the host and the server communicate (with signals). It is a good practice to avoid infinite loops or to continuously ping the host since this would unnecessarily stress the setup.

The *hardware* watchdog is an Ethernet-controlled switch that performs the host computer's power cycle if the host computer itself does not acknowledge any ping requests in a predefined time interval. The hardware watchdog is necessary to detect operating system hangs. To implement the hardware watchdog, it is necessary to include in the setup a power switch that can be controlled with a script via Ethernet. The hardware watchdog is particularly critical and should be carefully designed. Some events can make the watchdog continuously perform host power cycles, risking to damage (or destroy) the power unit, the host itself, or the power switch. Damage to the power switch can actually destroy all the hosts and DUTs connected to it. This has actually happened during one of our beam experiments. We suggest to place also the power switch far from the beam and protect it from scattered neutrons.

A critical decision when performing a radiation experiment of AI accelerators regards the data to log. The obvious answer would be to log everything and eventually parse data after the experiment. Unfortunately, this choice is impractical when DNNs or portions of a DNN (a layer) are tested since the output is vast and might contain a lot of unnecessary information. Even when testing a matrix multiplication or a convolution, the data to log should be carefully engineered. If a single element of the output matrix is corrupted, it is not necessary to log the whole output matrix. The need to reduce the data to log is not only necessary to avoid saturating the hard drive but also to reduce the time required to write the log.

An effective strategy to reduce the log size is to log only the differences between the expected output and the corrupted output, ensuring that the position of the corrupted element is logged, too. This solution can help to reconstruct the corrupted vector without storing it all. In the case of DNNs, not all the outputs matter. As discussed in Section II-A, the tensor vector at the output of a DNN contains the information about all the objects detected or classified, regardless of their probability. The output is then filtered, and only objects detected with a sufficiently high confidence are considered. It can be sufficient to log only the faults that modify the outputs that have a probability higher than the confidence threshold. Considering all the objects can actually lead to false positives since the very low-confidence object characteristics can change even without radiation due to the intrinsic probabilistic nature of DNNs.

An essential element of information that must be maintained when logging data during beam experiments is the time stamp. It is fundamental to correlate the observed event with the beam status or flux intensity. To do so, for every observed error, we need to track also the time stamp, ensuring that the server clock is synchronized with the facility counter clock. Losing the synchronization between the data acquisition and the facility clock can result in useless data since, while we have information about the number of events, we cannot calculate the cross section as the particle count is out of sync.

To measure the cross section, as detailed in Table II, we need to divide the number of observed events by the fluence, i.e., the number of particles per cm^2 that hits the DUT *while computing the code*. Thus, we need to discard, from the fluence count, all that time during which the DUT is irradiated, but it is not performing the calculation. Otherwise, we risk (significantly) underestimating the DUT cross section (we consider a longer exposure time, thus a higher fluence, reducing the cross section). Since the AI accelerator interacts with the host (not irradiated), it is not always easy to have a fine-grain estimation of the effective time the DUT spends in computing.

What we suggest is to log, per each DNN (or code) execution, the actual execution time in the DUT. Thus, besides logging the observed events, we are logging also, for each execution (correct or not), the execution time *in the DUT*. After the experiment, we can calculate the accumulated execution time, summing the execution times of all the executions in the DUT. Then, the cross section is calculated by dividing the error rate (observed events/accumulated execution time) by the average flux (particles/cm^2/s) during the experiment. The length of the experiment should be sufficient to reduce the experimental error that comes from particle count uncertainly and flux fluctuations. In most neutron facilities, experiments that last for 1 or more hours are suggested.

A very critical aspect of the setup for the AI accelerator radiation test is the execution versus setup time. To test a DNN, it is necessary to load the input (that can be a big image or even a video) to the host or DUT, execute the DNN, and then check for errors. The useful time for the experiment is only the DNN execution in the DUT. Thus, the load/download of data and error checking should be much faster than the code execution time in the DUT. Otherwise, most of the particles hitting the DUT would be wasted. As a general rule, the time required to setup the execution and check errors should be less than 10% of the code execution time in the DUT. This is not always easy to achieve, mostly when testing parallel devices. For example, testing matrix multiplication (or a convolution) on a GPU can be challenging. The execution of matrix multiplication is $O(N^3)$, where N is the size of the (square) matrix, since, for each of the $N \times N$ elements, we need to perform $2N$ operations (sums and multiplications). However, if the parallel DUT has sufficient parallel resources (one functional unit per element, which is normally the case), the computing complexity becomes $O(N)$ since each of the elements is computed in parallel with the others. If, as normally is, the host is a CPU, the time required to check the output correctness is $O(N^2)$ since each output element needs to be compared with the golden copy. As a result, the time spent in checking for correctness risks to be asymptotically longer than the actual test time. To solve this issue, it is important to engineer solutions to speed the correctness check, eventually using memory comparisons and checksum, or even to let the DUT itself perform the check (ensuring this lasts a negligible fraction of the actual code execution time).

The postprocessing of experimental data is fundamental to understand the reliability of DNNs. Besides the cross section calculation, in fact, it is necessary to identify the effect of the faults in the output, eventually distinguishing between critical and tolerable errors (see Section II-C).

B. Facilities and Setup Criticality

There are several facilities that can be used to test AI accelerators; the choice depends on the evaluation of interest. One fundamental aspect of AI accelerators is that most of them are available only on big boards (GPUs) or as embedded in SoCs. In both cases, the beam is likely to irradiate both the DUT and the supporting hardware. For the specific case of stand-alone GPUs, for instance, a focused beam (up to 3×3 cm) is to be preferred to avoid the irradiation of the onboard power control circuitry.

One of the critical challenges that we have addressed in the radiation experiments of GPUs is the errors in the onboard DDR. The GPU board has several gigabytes of DDR installed close (or even over) the GPU chip. This device memory is fundamental to avoid costly host-device memory transfer. When it comes to radiation experiments, DDR corruption needs to be avoided or reduced as much as possible. An issue that we have experienced with modern devices is that the onboard DDR experiences also permanent and intermittent faults, risking bias in the measured cross section. Unfortunately, since the DDR is close to the GPU chip, it is difficult not to have it irradiated, and since the GPU chip needs to dissipate a lot of heat, the DDR temperature risks are very high (thus increasing the aging and permanent faults occurrence). The suggestion, when testing modern stand-alone GPUs, is to select devices that include ECC on the DDR; otherwise, the data risk being useless and the device unusable after a few hours of experiment.

The permanent errors' effect is a severe issue also for some SoCs, in particular, those with the system boot-loader stored in flash memory. The radiation-induced corruption of this memory risks preventing the system to boot. Some SoCs that we have tested experienced these failures after a few hours of neutron irradiation.

There might be other critical aspects for the experiment, specific to the device under test. The fact that we are testing commercial devices normally used for AI model prototyping intrinsically requires attention and experience.

In Section IV-C, we will list the most interesting experimental results obtained by irradiating AI accelerators. There are various facilities used for these experiments. The facilities that have been most frequently used to measure the error rate of AI accelerators are the Los Alamos Neutron Science Center (LANSCE) in the Los Alamos National Laboratory (LANL), USA, and ChipIR at the ISIS neutrons and muons source at the Rutherford Appleton Laboratory, U.K., for neutrons. Brookhaven's Booster accelerator at the NASA Space Radiation Laboratory (NSRL), USA, the Lawrence Berkeley National Laboratory (Berkeley Lab), USA, and the Particle Therapy Research Center (PARTREC), The Netherlands, has been used for heavy ions and the Massachusetts General Hospital (MGH) for protons. All these facilities have been shown to have beam characteristics that suit the radiation experiment of AI accelerators. It is definitely of extreme importance to discuss your setup with the instrument scientists to check possible criticalities. Discussing with users who have already performed experiments on similar devices is also extremely helpful.

C. Experimental Results

There are several documented datasets about AI accelerator reliability available in the literature (listed in Table III); here, we just provide an overview of the most interesting results. The scope of this data survey is to help the reader in finding the data of interest and understand the data collection and parsing. Most of the available data refer to neutron experiments, while some preliminary results with protons and heavy ions are being published. We will first consider GPUs that have been extensively studied. Then, we will also list the available data on FPGAs and EdgeAI accelerators for AI. It is fundamental to recall that when testing an ML application, the input frame (how many objects of how many classes are present in the picture), the dataset (what is the network doing), the training (how has the network been trained), and the accuracy (how well the network behaves without faults) of the model are all factors that can significantly bias the error rate.

1) Neutrons: **GPUs** have been tested extensively since 2012 when the first Radiation Effect Data Workshop paper appeared [122]. Most of the first papers regarding GPUs

TABLE III
AVAILABLE EXPERIMENTAL DATA OVERVIEW

Paper	Device(s)	Code(s)	Radiation	Main Considerations
[122]	GPU	MxM	neutrons	first public data on GPUs reliability
[7], [123]	GPU	various	neutrons	scheduler and parallelism management is vulnerable and critical
[124], [125]	GPU	MxM, FFT	neutrons	multiple output elements can be corrupted by a single particle
[126]	GPU, Xeon Phi	various	neutrons	the parallel architecture influences the code sensitivity and error criticality
[127]	GPU, ARM, FPGA	various	neutrons	strong dependence between computing architecture and code sensitivity
[11]	GPU	MxM, CNNs	neutrons	multiple corruptions cause misclassification on CNNs
[128]	tensor cores	MxM	neutrons	tensor cores have higher error rate and different fault model
[129]	GPU, Xeon Phi, FPGA	various	neutrons	low precision reduces the error rate but has a higher impact on the output
[130]	GPU	MxM, Yolov3	neutrons	most DUEs are generated in hidden hardware resources
[131]	GPU DDR	various	neutrons	on-board DDR are prone to experience permanent faults
[132], [133]	FPGA	MNIST	neutrons	high error rate, reduced with lower precision implementation
[134]	NeuroShield	CNNs	neutrons	robust setup and simple fault model
[13]	Google TPU	conv., CNNs	neutrons	characterization of atomic operations and CNN fault model
[135]	Versal SoC	various	neutrons	neutrons and protons data, no permanent effect
[136]	Flashed-based FPGA	LeNet	neutrons	low precision increase fault criticality
[137], [138]	GPU SoC	MxM, LuD	protons	software implementation and parallelism impact the GPU error rate
[139]	AMD GPU	various	protons	FIT rate and behavior under protons
[140]	Versal ACAP	various	protons	neutrons and 64MeV protons SEL and SEU data on Programable Logic
[141]	Versal SoC	various	ions	comparison of protons and ios, no SEL
[142]	GPU	various	ions	overview of heavy ion test setup and data
[143]	AI accelerators	various	ions	extensive comparison of the reliability of AI accelerators for in space
[144], [145]	Myriad VPU	various	ions	no latchup, low error rate in DDR, potentially good for space mission
[146], [147]	Snapdragon SoCs	memory	ions	SEL, crash, and stuck bit data reported
[148]	various	various	ions	overview of reliability, performance, power of COTS AI accelerators

consider neutron radiation, given the importance of GPUs for terrestrial applications (mainly autonomous vehicles).

A key criticality highlighted by the reliability evaluation of GPUs is that, besides the high error rate caused by the large device area, there are some additional vulnerabilities that come from the device parallelism management. Since GPUs have hardware schedulers (in CPUs, the scheduling is performed, in software, by the operating system) that need to orchestrate the execution of thousands of threads, they can be corrupted by radiation and their corruption can have catastrophic outcomes. In [7] and [123], neutron beam experiments demonstrate that increasing the number of parallel processes imposes a higher scheduler strain that increases the GPU error rate.

It has been shown that the corruption of shared resources (like caches) or the scheduler affects the execution of multiple parallel processes [124], [125]. Interestingly, the number of parallel processes that can be corrupted and the impact of the corruption (how different the corrupted value is from the expected one) depends both on the parallel architecture and the executed algorithm [67], [126], [127]. The corruption of multiple parallel processes is particularly critical for CNNs, as shown in [11]. Multiple parallel process corruption, in fact, modifies a portion of the feature map (output of a convolution layer), which can induce the network to misclassifications. Faults during convolution in GPUs manifest at the output as corrupted row(s)/column(s) or huge corrupted block(s).

Novel architectural solutions to improve the efficiency of convolutions, such as tensor cores (hardware that performs up to 16×16 matrix multiplication in one clock cycle) and mixed-precision cores (dedicated functional units for low precision data and operations), have also been tested with neutrons [128], [129]. Tensor cores, having a large area, have a higher error rate compared to the software execution of the matrix multiplication, while mixed-precision hardware has a smaller error rate. In contrast, a fault in a low-precision operation has a higher impact on the output correctness since a corruption in a 16-bit value is likely to have a higher impact than a corruption in a 64-bit value. This trend has been confirmed also on dedicated accelerators for multi-bit-width CNNs implemented on a flash-based FPGA [136].

Some effort has also been carried out to identify the causes for DUEs, i.e., application crashes or device hangs. The comparison of beam data in GPUs, FPGAs, and CPUs executing different algorithms highlights that DUEs have a strong component that is independent of the executed code and is, then, related just to the hardware [127]. A more detailed analysis, on GPUs, presented in [130], categorizes the DUE by analyzing the Syslog that records GPU errors combining beam experiments and fault injection experiments that disturb the control flow. Graphics engine exception, GPU memory page fault, GPU processing stop, and internal microcontroller halt are observed (the latter only during beam experiments). This result suggests the hardware that is not disclosed to the users contributes to DUE errors substantially. An independent study, purely based on beam experiments, confirmed most of these findings [149].

During radiation experiments of modern computing devices, it is of paramount importance to pay particular attention to the onboard DDR. Unfortunately, shrinking DDR technology is prone to experience permanent or intermittent failures [131]. This risk is exacerbated with temperature, and the DDR in most modern devices is placed in close proximity to the computing core (that can reach high temperatures). Boards with ECC in the DDR are to be preferred when deciding which device to test.

Various AI accelerators have lately been tested, both implemented on FPGAs or EdgeAI chips. On an **FPGA**, as mentioned in Section II-B, the neural network can be implemented as a whole pipeline or using systolic arrays. The former implementations have been shown to be particularly susceptible to neutrons. In addition, reducing data precision and simplifying the sigmoid function (2) reduce the neural network error rate [132], [133]. Systolic array implementation allows to map larger models in the FPGA since the same hardware is used to process various convolutions [150]. The drawback of systolic array implementation is that, once the circuit is corrupted, all convolutions mapped on it will produce a wrong output, reducing the reliability of the network.

EdgeAI accelerators, such as the NeuroShield or Google's TPU, have been tested [13], [134]. According to the reported experience, the setup for EdgeAI accelerator testing is easy since the device is connected via USB to a host CPU (to be placed out of the beam). Reported data highlight that the setup hardly fails (while GPU setup fails quite often), and once mounted, it does not require to replace any failing parts. The test on the NeuroShield and the TPU shows a lower error rate compared to GPUs and a simpler error model since few output elements are corrupted and the corrupted value is similar to the expected one. As a result, also, the misclassification rate of neural networks on EdgeAI devices is lower than on other devices.

2) Protons: **GPUs** have also been tested with protons, showing similar trends and behavior as neutron tests [137], [138], [139]. In particular, three different strategies to implement matrix multiplication have been tested. Results highlight that the slower memory-bound algorithm is more error-prone, while the most efficient algorithm has a smaller cross section [137]. Parallel strategies reliability has been tested on lower–upper decomposition and a comparison of a memory bound, and a compute-bound implementation of the decomposition has been proposed. Results show that more intensive use of the resources of the GPU increases the cross section [138].

3) Heavy Ions: Heavy ion experiments have been performed on **AI accelerators** to evaluate at which level they can be used as part of a space mission [142], [143]. Particular attention is given to the device preparation and the setup to be sure that the experiment can be carried out. AI accelerators are normally powerful devices that need to dissipate heat efficiently. If the chip needs to be delidded for allowing ions to penetrate or if the test must be performed in vacuum, this might not be possible, requiring the use of cold fingers or other thermal solutions [142].

The **vision processing unit** (VPU) embedded in the Myriad SoC has also been exposed to heavy ions [144], [145]. The promising news is that no Latchup was observed in any of the tested chips up to an effective LET of 110 $\text{MeV} \times \text{cm}^2/\text{mg}$. The measured single-event functional interrupt cross section of the SoC is about 10^{-4} $\text{cm}^2/\text{device}$. In addition, a side experiment on the onboard DDR3 reveals the possible presence of an error correction scheme in the memory.

An overview of the reliability and power/performance of various commercial devices for space application attests sufficiently positive results, paving a path toward low Earth orbit trials and the complete life cycle for space-based AI classifiers on orbital platforms and spacecraft [148].

V. Fault Injection and Fault Propagation

Fault injection is a fundamental step in the understanding of AI framework reliability. By injecting faults at different levels of abstraction, it is possible to track how faults propagate, identifying the most critical procedures or code portions. Fault injection is, indeed, a complementary approach to beam experiments that can be particularly useful when dealing with complex hardware. In fact, if the fault propagation path is short or naive (as in the case of memories), beam experiments can be used to correlate the observed fault (wrong data) with the fault source (memory cell corrupted). On the contrary, if there is a significant pipeline of stages and abstraction levels from the fault origin and the observable points (the network output), it is impossible, with beam experiments alone, to have full visibility of the fault propagation limiting the understanding of the critical portions of the hardware and software. A neural network is normally described in python, which is a very high-level programming model. This description is then implemented in C, VHDL, or CUDA for then being compiled (or synthesized). Each executed machine instruction involves the use of complex hardware resources (multiplier, tensor core, and systolic array) that can be corrupted. The way the fault in a gate of such complex hardware affects the whole abstraction stack is very hard to reverse engineer observing just the final output (see Fig. 12).

Instead of exposing the manufactured chip to radiation, fault injection is based on models of the system and artificially injects faults through simulation at different levels of abstraction: from RTL to architecture, microarchitecture, and software. The main difference between fault injection and beam experiments is that, in the latter, faults are generated by the particles and then propagate, while, in the former, the fault is assumed to have modified the resource (gate, memory, code, and operation—depending on the chosen abstraction level). In other words, fault injection does not include the information regarding the probability for the fault to be generated but just measures the probability for the fault to propagate. This probability for a fault to propagate to the output of an application is measured by injecting faults in the accessible resources of each level's model (gates, registers, hardware arrays, variables, instructions, and so on).

Fault injection is interesting since it provides complete observability of the abstraction layer details but has two main limitations: 1) the fault model and fault injection probabilities are synthetic (i.e., defined/modeled by the user and/or the simulator), and thus, the obtained results may not correspond to the physical phenomena and 2) faults can be injected only in that subset of available resources that are accessible at each abstraction layer, and thus, the evaluation may not be complete nor exhaustive. Depending on the abstraction level at which the fault injection is performed, the limitations can be more or less critical [151].

When SRAM-based FPGA is considered, fault injection becomes a primary evaluation tool. In fact, since the configuration memory is accessible and represents the most critical resources of the device, in FPGAs, fault injection can draw accurate evaluations [152], [153] and is highly suggested for ML applications.

To inject an error at the software level (source code) executed on a chip, it is necessary to have characterized the fault model, i.e., how the hardware corruption modifies the software being executed. Injecting a single bit flip in a variable is simply wrong [151] since corruption in a complex AI accelerator can hardly be translated in such a naive fault model. Injection of a single transient fault in a gate at RTL might be accurate, though, since the propagation is performed in the downstream simulation. It is obvious that the more accurate the injection, the longer the process time. Just to put this time in perspective, an accurate RTL fault injection of a simple CNN as LeNet can take hundreds of *years* [154].

In addition, it is fundamental to perform a sufficiently high number of injections to guarantee good accuracy [71], [155]. Otherwise, the results risk being misleading, given the high amount of operations executed and data processed.

In the following, we will list some of the available frameworks for fault injection at different levels of abstraction, from RTL (see Section V-A) to microarchitectural (see Section V-B), software (see Section V-C), and hybrid approaches (see Section V-D). The goal of this list is to highlight the benefits and problems associated with each level of abstraction and to summarize the main observed results.

A. RTL Fault Injection

RTL fault injection, by simulating the circuit of the device under test, potentially allows observing the propagation of the fault from the gate to the software output. Nonetheless, injecting faults at this low level of abstraction comes with two main limitations. The first one is the high simulation time. A *single* fault injection can take *hours*, depending on the complexity of the hardware and the executed application [8], [78]. The second limitation is the availability of detailed descriptions of COTS circuits. Most available RTL descriptions are either obsolete (based on G80 NVIDIA chip) or lack significant details [156].

Some interesting fault injection campaigns on GPUs' RTL description highlighted that the criticality of functional unit corruption depends on the executed instructions, and a single transient fault in the GPU hardware can hardly manifest as a single output corruption at the neural network output. On the contrary, when convolution is executed on a GPU, the single fault is likely to spread affecting various or several elements of the feature map (convolution output) [154]. With the insights derived from the RTL fault injection, it is, indeed, possible to identify the most vulnerable and critical resources of GPUs. However, given the complexity of the computing architecture, the challenge of implementing effective selective hardening solutions for those critical resources is still unsolved.

RTL descriptions are available also for systolic-array-like accelerators, such as the NVIDIA DL Accelerator (NVDLA) [157]. Again, the availability of the low-level description of the circuit allows a fine-grain reliability evaluation. A fault-injection framework dedicated to ML models executed on the NVDLA accelerator is publicly available [158].

Despite the potential high accuracy, RTL injection should be carefully engineered since the injection time can easily become prohibitive. An exhaustive campaign that considers all injection sites and possible executed operations and inputs is simply unfeasible. Some strategies to reduce the characterization time have been proposed [154], [159], dividing the layers of the network and composing the propagation in a later stage, for instance. However, it is necessary to evaluate at which level the introduced simplifications reduce the fault injection accuracy.

B. Microarchitectural Fault Injection

Microarchitectural fault injection acts at a higher level of abstraction compared to RTL. The device is described using performance models, significantly simplifying the circuit to simulate. This simplification, on the one hand, speeds up the fault injection (by orders of magnitude) but, on the other hand, can potentially reduce the accuracy of fault injection. This latter aspect is particularly critical when only behavioral models are available, as in the case of GPUs [68]. As a result, the microarchitectural fault injection can potentially quickly indicate the high-level module that is more critical for the neural network execution but can hardly be accurate in predicting the error rate or defining the error model (i.e., how the hardware corruption modifies the operation output).

Lately, an effort has been carried out to evaluate how close the error rate prediction based on microarchitectural fault injection is to that measured with beam experiments [76], [114]. On ARM devices, the correlation is proven to be very strong, with the fault-injection SDC prediction being inside one order of magnitude of difference from the beam experiment result, even when the CPU is embedded in a SoC or an operating system is employed. Unfortunately, the functional interrupt cannot be predicted with just microarchitectural fault injection [76], [114]. In addition, a comparison between microarchitectural fault injection prediction and beam experiment is not yet available for more complex devices.

C. Software Fault Injection

The fastest way to simulate fault propagation is by injecting errors directly into the source code. With an interruption-based approach, it is possible to freeze the code execution, modify a memory location, and then restore the execution, checking the effect of the corruption at the output. Since the injection is purely software, the execution can be performed on the live hardware, significantly reducing the injection overhead (one fault injection takes as much time as a normal execution) and the fault injection design (no need to simulate and control the circuit). The ease of use and efficiency of software fault injection makes it popular among research groups evaluating the reliability of neural networks [62], [69], [70], [100], [160], [161], [162]. However, the very high level of abstraction at which software fault injection is performed imposes a need to carefully select the error model. Injecting a single bit flip in a random variable at a random time risks being unrealistic. In other words, injecting a single-bit flip in software assumes that the propagation of the particle interaction from the transistor all the way through the circuit, architecture, and operation is manifestly a single corrupted bit, which is clearly very unlikely. In addition, very high-level frameworks designed in Python are now available to ease the development and execution of AI models. The model is then translated into machine code with complex compile stages. Injecting a single-bit flip at the Python level worsens the accuracy loss. As discussed in Section V-D, one way to combine the efficiency of software fault injection with the accuracy of lower level evaluation is to perform hybrid fault injection, characterizing the fault model to inject.

Despite the intrinsic limitation of software fault injection, it can be a useful tool to have a first understanding of how the model behaves when corrupted. A detailed analysis of the distribution of errors in matrix multiplication (essential operation in DNNs) has been performed in [160]. The low cost of software fault injection allows authors to evaluate the effect of modifying the size of the matrix and the thread-block size, which would have been highly costly to be performed with beam experiments or lower level fault injection. The flexibility of software fault injection also allows us to understand the effect of data precision on the reliability of neural networks [136], [161]. These results confirmed data obtained previously with beam experiments [11], [128], [129], [133], adding more details and configurations.

A tentative comparison between software fault injection and beam experiments has been proposed for ARM processors [105]. Nonetheless, unlike microarchitectural fault injection (limited to simple ARM architectures), a good and accurate way to predict the realistic error rate of a device from the results of a software fault injection has not been found, yet. This tool, then, should be used carefully.

D. Hybrid Approaches

Several attempts have been proposed to combine the flexibility and efficiency of high-level fault injection with the accuracy of low-level evaluations. The common idea of these approaches is to characterize the effect of low-level faults in the higher abstraction layers. In other words, the fault propagation is split in two. The low-level injection is used to build a database of possible fault manifestations, typically considering a limited set of instructions and inputs. Then, the injection at a higher level is performed taking the error to inject from this database. It is clear that the challenge is to build a representative database.

There are two main hybrid approaches that have been applied to AI accelerators.

1) *Beam and Software Fault Injection [104]:* Beam experiments are used to measure the fault probability of basic GPU instructions. This has been done by executing microbenchmarks composed of a list of machine instructions in the GPU. By also checking the output correctness, it is also possible to understand what is the impact of the hardware fault in the instruction output. As shown, a radiation-induced fault in computing resources hardly produces single-/double-bit flips [104]. Then, a fault model has been injected in software, picking from errors observed with the beam, the one that is most suitable for the instruction to be corrupted (based on the opcode and input values). Interestingly, a beam experiment validation proved that this methodology is quite accurate in predicting the error rate of applications executed on GPUs [104]. Thus, it might be sufficient to characterize the radiation effects on basic and common instructions for then predicting any code error rate.
2) *Low Level and Software Fault Injection [154], [163], [164]:* Similar to the previous hybrid approach, the RTL fault injection or low-level fault simulation is used only to characterize the fault effect at the output of basic machine instructions. These effects are then propagated in software. Interestingly, GPUs have particularly critical resources, such as the scheduler, whose corruption can modify the output of various threads. These multiple corruptions are then the ones more likely to induce a misclassification in CNNs [154]. This has also been validated for permanent faults caused by aging of the GPU control and parallelism management units [164].

VI. Available Hardening Solutions

Improving the reliability of complex AI accelerators is a challenging task. Building a fault-tolerant AI or HPC system is possible [165] but requires a great effort. Modifying the fabrication technology or the device layout would be way too costly, and even if it was feasible from a budget point of view, it would risk jeopardizing the device's performance and efficiency. Adapting software or circuit traditional error mitigation approaches, such as replication, might not be the best option, either. In fact, the overhead of duplicating or triplicating the DNN operations can be very high. In addition, since not all the errors are critical for the functionality of the network, full replication imposes unnecessary overhead (we are duplicating even operations whose corruption can have a minor impact on the classification output).

Interestingly, embedded solutions such as SECDED ECC have been shown to be less effective for neural networks compared to traditional HPC algorithms [11], [16], [166]. In particular, enabling ECC only slightly reduces the number of misdetections, suggesting that most faults causing critical errors do not originate in memory.

Luckily, the probabilistic nature of neural networks and their intrinsic redundancy and computational flexibility (the same accuracy can be achieved in various ways) allow the designing of innovative dedicated mitigation solutions. Lately, various strategies have been proposed to improve the reliability of neural networks executed on accelerators. However, as discussed in Sections IV and V, a hardware fault can corrupt various bits of various elements in the computation of convolutions and AI-related operations. Thus, when designing or validating a hardening solution, special attention must be given to the adopted fault model. A very efficient and effective solution to protect the neural network from single-bit flips might be ineffective when adopted in the field. As a general rule, only hardening solutions that have been designed with

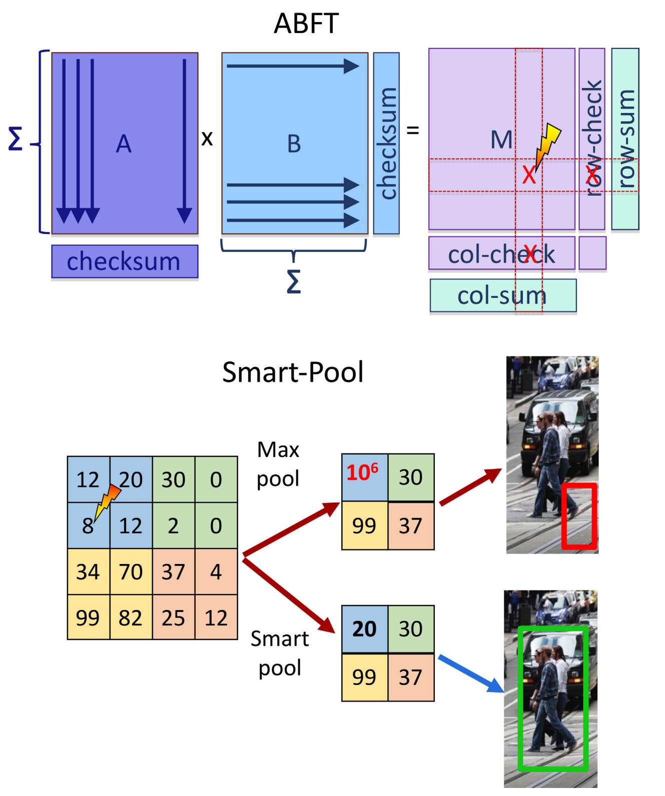

Fig. 14. Description of the basic concept of ABFT applied to matrix multiplication (top) and smart pooling (bottom). These hardening solutions are highly effective in reducing the impact of radiation in CNNs' execution [11].

an experimentally obtained error model or have been validated with experiments should be adopted in the field.

Hardening can be applied at various levels. In this work, we will focus on software and architecture hardening. The former is highly flexible and, if properly designed, can be very effective in reducing the failures in neural networks. It is worth noting that the cost–benefit of software hardening can be much higher for neural networks than for traditional codes since neural networks have intrinsic redundancies that could be exploited. Architectural hardening can be also efficient, mostly in modern accelerators that include several heterogeneous resources. By using the idle hardware concurrently with the neural network execution, it is possible to implement cheap replication. As a final note, we will also discuss how it is possible to exploit ML potential to train the neural network to deal with transient faults. This latter solution is extremely powerful and promising.

A. Software

Modifying the software, adding replication or checksum, is one of the easiest ways to increase the reliability of a code. This also applies to neural networks. Full duplication has been shown to be effective but highly costly [168], [169]. Some works have proposed *selective* replication in order to save overhead by protecting only the most critical layers or portion of the neural network [12], [170], [171], [172]. Interestingly, the overhead of duplication can be lowered by up to 50%, yet maintaining comparable error detection capabilities.

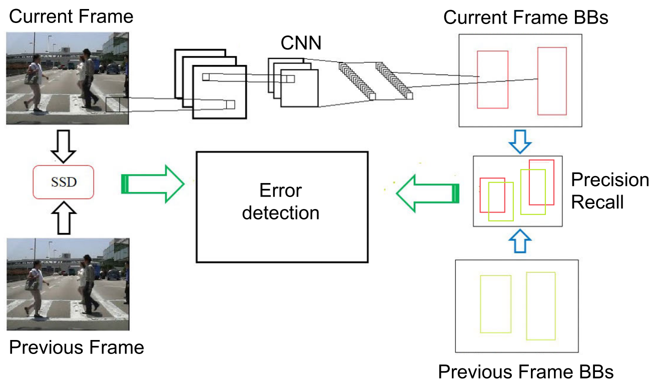

Fig. 15. Structure of the error detection framework implemented comparing subsequent input images and the correspondent detection. Similar input frames should provide similar detection [167].

Since most operations in current neural networks are matrix-multiply related, some interesting works have proposed to adapt existing algorithm-based fault-tolerant (ABFT) solutions to CNNs, as shown in Fig. 14. ABFT adds invariants to the code and takes advantage of those for quick error detection or correction. For matrix multiplication in GPUs and parallel accelerators, in general, ABFT is particularly effective, efficient, and able to detect and correct more than 80% of errors in linear time [124]. When adapted to CNNs, ABFT has been shown to outperform ECC or even duplication [11]. Lately, smart light-ABFT solutions have been introduced to further reduce the overhead of the mitigation solutions for GPUs [173] and FPGAs [150]. Other light-weighted solutions have also been proposed for generic image processing applications [174].

A smart solution to filter errors propagating in CNNs is to check whether the propagated values during MaxPooling layers are *reasonable*. Max pooling, as depicted in Fig. 14, selects only the element with the highest value to be propagated in downstream layers. Rather than simply propagating the element, it is possible to check if the value of the element is inside a range of possible values. This strategy is promising since most of the values propagated in neural networks are restricted to a narrow range, while the effect of radiation corruption can be wide. This solution can detect up to 85% of critical errors in CNNs [11].

Moreover, the time correlation between input frames and output detection can be exploited to identify errors. A CNN processes each frame independently of the previous ones. However, when looking at a scene, we know that there is a strong correlation between subsequent frames. The same correlation should hold for the detection, as shown in Fig. 15. Thus, one way to detect errors is to compare the subsequent input frames and the corresponding detection output. If the two frames are very similar, then the resulting detection should also be similar. Otherwise, an error flag can be triggered. This correlation has been shown to detect about 70% of critical errors while incurring some false positives [167].

B. Architecture

The computing architecture of AI accelerators, in particular, GPUs, has unused resources that can be exploited to implement duplication. Novel GPUs, as shown in Fig. 16, include dedicated functional units to execute operations in 64, 32, or 16 bits and specific units for the execution of matrix multiplication (tensor cores). When a functional unit of a given precision is used, the others are idle, opening the possibility to implement architectural duplication. The fact that the redundant copy needs to operate in a different precision forces the comparison not to be bit-by-bit (the two copies are naturally different) but rather to use a threshold of acceptable difference. Reduced precision duplication with comparison (RP-DWC) has been implemented and experimentally validated in GPUs [175], showing promising error detection capabilities (75%, on average) and exiting overhead (less than 20%). It is worth noting that the undetected faults fall in the intrinsic difference between the two copies and, thus, are likely to be precision errors with little impact on neural network detection. A similar approach has been later applied to image processing pipelines [176], [177] with similar performances.

The other option to improve the accelerator architecture reliability is to include specific fault detection mechanisms. For instance, a built-in self-test (BIST) that allows monitoring the performance of the inference engine hardware has been embedded on the Myriad VPU [145]. The in-flight diagnostics tests for the VPU inference engine indicate that the device performed as expected, without experiencing any functional upsets, or any functional degradation effects due to radiation.

C. Exploiting AI Potential for Error Mitigation

All the hardening solutions presented so far adapt to ML application mitigation strategies derived from traditional computing. Still, AI has an intrinsic potential that can be exploited for error mitigation. A preliminary study focused on reducing the number of object classes to be classified by the model [179]. The intuition is that the higher the number of classes, the smaller the Hamming distance between two different classes. Thus, with a high number of classes, a smaller variation in the output tensor can induce a wrong classification. Normally, models are trained on standard datasets that include hundreds or thousands of object classes from different domains. By tuning the number of classes to the specific application need, we can spread the object probabilities halving the change of misclassifications [179]. This solution can be particularly effective in space applications, for which only few objects are of interest.

Neural networks are highly adaptable to different scenarios, as long as sufficient structured information is provided at training time. In other words, DNNs are powerful tools to perform the tasks that they have been trained for, but DNNs usually perform poorly when deployed in scenarios not seen during the training phase. For instance, a model trained exclusively with day scenes will experience a significant drop in performance when tested on night scenes with poor illumination. A solution to this problem is the use of *data augmentation* to allow the network to also experience situations not present in the original training data. As depicted in Fig. 17, we can impose on the DNN to still provide correct output even if we inject faults

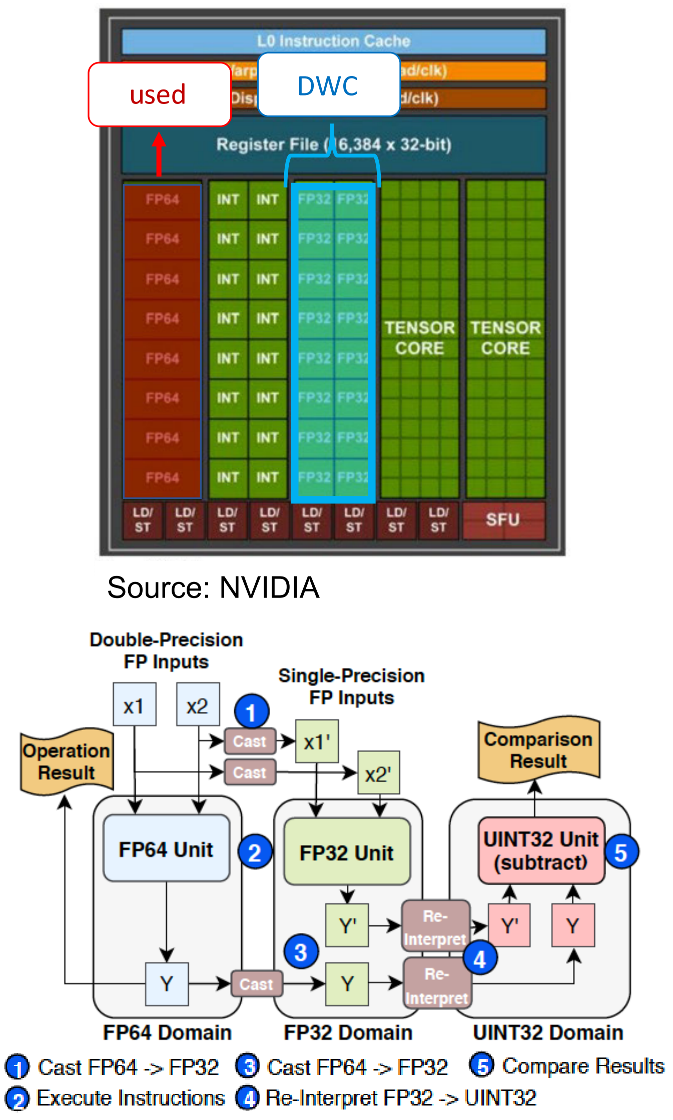

Fig. 16. Redundant low-precision functional units can be used to implement efficient DWC strategies. On GPUs, when a code is executed in a given precision, the functional units of different precision are idle and can be leveraged for duplication [175].

at training time. If the DNN is trained to classify objects correctly, even with some selected transient faults, it is possible to produce a more reliable model while maintaining the original accuracy. The idea is to allow the network to familiarize itself with the occurrence of neutron-induced errors by *injecting noise (transient faults) during training*. In particular, while performing the forward pass of the DNN training, a random corruption is imposed on the feature maps (convolution output) in a given layer of the network, injecting an experimentally observed fault model. As a result, the model autonomously learns how to properly deal with these kinds of faults by adjusting the learned weights to reduce the likelihood of a misprediction. This solution has been introduced for quantized DNNs [180] and extended to complex CNNs executed on GPUs [178].

The challenge that needs to be addressed is the selection of the faults to be injected into the network during the training

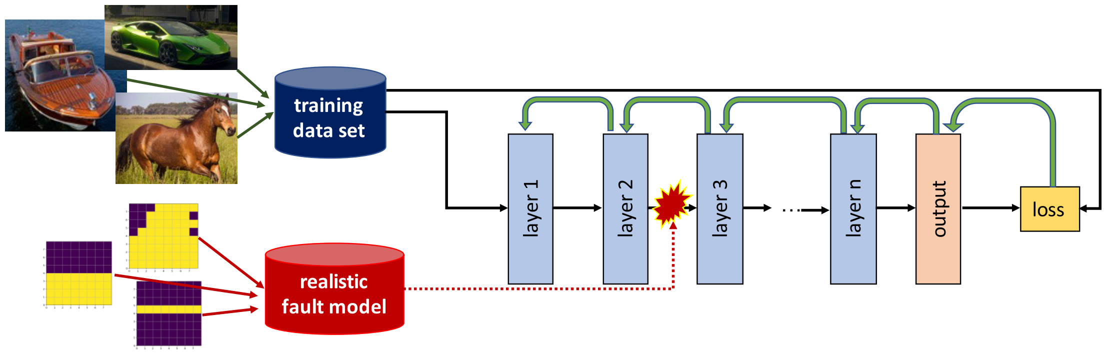

Fig. 17. Fault-aware training is a promising solution to make the DNN more reliable [178].

phase. Considering a high number of faults could increase the network experience in dealing with faults but risks to prevent the training convergence. A low number of faults will result in quick training but might be ineffective. Moreover, it is not possible to ensure that a particular random fault in a given layer will lead to an error (thus, the model will learn how to deal with it). Injecting single-bit flips, for instance, is transparent to the network training. The best approach is to identify the fault models that are more likely to induce mispredictions and inject these faults in the majority of the training samples.

As a very promising result, fault-aware training has been shown to reduce by up to one order of magnitude the number of critical faults in DNNs without reducing performance (nearly zero overhead).

VII. Conclusion

ML is becoming the new computing paradigm. The potentiality and performance of the available models allow incredible flexibility. With neural networks, it is possible to quickly identify patterns or objects, train the system to make a decision, and extract hidden information from data. This potentiality is extremely useful in a variety of applications for which reliability is paramount, from self-driving vehicles to space exploration. Nonetheless, there are various critical aspects that must be considered before integrating ML in a safety-critical application.

First, the model is intrinsically probabilistic. There is no deterministic output, but the decision needs to be made based on probability. The accuracy of such a probability is a function of the training and the conditions in which the code is executed. It is very hard (if not impossible) to estimate the accuracy of an ML model when employed in the field.

Second, the hardware necessary to execute the ML model is highly complex. A single fault can have unpredictable effects on the software execution. Simply considering memory errors or single-bit flip is too naive and is likely to largely underestimate the impact of faults in the field. The complexity of the hardware is such that requiring a full understanding of radiation effects from the transistor to the tensor output could be impractical (if not impossible). Novel strategies to bind the error rate and the fault effects are then required.

The hardening solution designed for ML accelerators cannot be adapted from classical computing. In fact, since not all the error matters in a neural network and since the computing power required to perform inference is extremely high, protecting everything will introduce unnecessary (high) overhead. To implement an efficient and effective hardening solution, it is necessary to study the model implementation and the hardware response to radiation. By identifying the faults that are more likely to cause misdetections or misclassifications, it is possible to design specific mitigation solutions reducing the overhead. In addition, exploiting ML potential is likely to significantly reduce the impact of faults in the network prediction, maintaining performance unaltered.

There are numerous unexplored aspects of ML reliability, from both the software and hardware sides. Various communities are considering neural network reliability from different perspectives. Nonetheless, it is the belief of the author that the radiation reliability community needs to take the lead in the quest for efficient ML reliability. In fact, without an experimentally validated fault model, it is impossible to understand the problem that we need to address, and without an experimental validation of the hardening solution, it is impossible to ensure its effectiveness. It is clear that the challenge is not trivial, but it is about time to build a new generation of reliable computing, going beyond the traditional devices that we are used to dealing with.

Acknowledgment

This work would not have been possible without the support and help of many people who deserve to be acknowledged. Luigi Carro had the pioneering idea of testing the reliability of the graphics processing unit (GPU) and convinced the author to follow him on this path. The author's students at the Universidade Federal do Rio Grande do Sul (UFRGS), in particular Dr. Fernando Fernandes dos Santos, Pablo Bodmann, Caio Lunardi, Dr. Fabiano Libano, and Lucas Klein, did most of the work here presented. The beam experiments were supported by the ISIS muons and neutrons source at the Rutherford Appleton Laboratory, thanks to Carlo Cazzaniga and Maria Kastriotou, and by Department of Energy (DOE) Los Alamos Neutron Science Center (LANSCE) at the Los Alamos National Laboratory (LANL), thanks to Steve Wender. The author particularly thanks Chris Frost (ChipIR) and Heather Quinn (LANL) who have been fundamental in improving the quality and motivating this—and future—research. The Department of Industrial Engineering (DII) of the University of Trento and Prof. Dario Petri, in particular, are thanked for their support and the opportunity to continue this research. Finally, the author would like to acknowledge the precious help of the anonymous reviewers and editors in improving the quality of this paper.

References

[1] E. Alpaydin, *Introduction to Machine Learning* (Adaptive Computation and Machine Learning), 3rd ed. Cambridge, MA, USA: MIT Press, 2014.

[2] C. M. Bishop, *Pattern Recognition and Machine Learning* (Information Science and Statistics). Berlin, Germany: Springer-Verlag, 2006.

[3] M. Wani, M. Kantardzic, and M. Sayed-Mouchaweh, *Deep Learning Applications* (Advances in Intelligent Systems and Computing). Singapore: Springer, 2020.

[4] NVIDIA. (2023). *NVIDIA Ampere GPU Whitepaper*. Accessed: Oct. 21, 2023. [Online]. Available: https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf

[5] Q-Engineering. *Google Coral Edge TPU Explained in Depth*. Accessed: Aug. 27, 2023. [Online]. Available: https://qengineering.eu/google-corals-tpu-explained.html

[6] C. D. Schuman, S. R. Kulkarni, M. Parsa, J. P. Mitchell, P. Date, and B. Kay, "Opportunities for neuromorphic computing algorithms and applications," *Nature Comput. Sci.*, vol. 2, no. 1, pp. 10–19, Jan. 2022.

[7] P. Rech, L. L. Pilla, P. O. A. Navaux, and L. Carro, "Impact of GPUs parallelism management on safety-critical and HPC applications reliability," in *Proc. 44th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, Atlanta, GA, USA, Jun. 2014, pp. 455–466.

[8] J. E. R. Condia, J.-D. Guerrero-Balaguera, F. F. Dos Santos, M. S. Reorda, and P. Rech, "A multi-level approach to evaluate the impact of GPU permanent faults on CNN's reliability," in *Proc. IEEE Int. Test Conf. (ITC)*, Anaheim, CA, USA, Sep. 2022, pp. 278–287.

[9] F. Su, C. Liu, and H.-G. Stratigopoulos, "Testability and dependability of AI hardware: Survey, trends, challenges, and perspectives," *IEEE Des. Test. Comput.*, vol. 40, no. 2, pp. 8–58, Apr. 2023.

[10] Y. Ibrahim et al., "Soft errors in DNN accelerators: A comprehensive review," *Microelectron. Rel.*, vol. 115, Dec. 2020, Art. no. 113969.

[11] F. F. D. Santos et al., "Analyzing and increasing the reliability of convolutional neural networks on GPUs," *IEEE Trans. Rel.*, vol. 68, no. 2, pp. 663–677, Jun. 2019.

[12] F. Libano et al., "Selective hardening for neural networks in FPGAs," *IEEE Trans. Nucl. Sci.*, vol. 66, no. 1, pp. 216–222, Jan. 2019.

[13] R. L. Rech et al., "High energy and thermal neutron sensitivity of Google tensor processing units," *IEEE Trans. Nucl. Sci.*, vol. 69, no. 3, pp. 567–575, Mar. 2022.

[14] M. B. Sullivan et al., "Characterizing and mitigating soft errors in GPU DRAM," in *Proc. 54th Annu. IEEE/ACM Int. Symp. Microarchitecture*. New York, NY, USA: Association for Computing Machinery, Oct. 2021, pp. 641–653.

[15] Y. Ibrahim et al., "Soft error resilience of deep residual networks for object recognition," *IEEE Access*, vol. 8, pp. 19490–19503, 2020.

[16] D. A. G. G. de Oliveira, L. L. Pilla, T. Santini, and P. Rech, "Evaluation and mitigation of radiation-induced soft errors in graphics processing units," *IEEE Trans. Comput.*, vol. 65, no. 3, pp. 791–804, Mar. 2016.

[17] NVIDIA. (2018). *NVIDIA Jetson AGX Xavier Developer Kit | NVIDIA Developer*. Accessed: Oct. 21, 2023. [Online]. Available: https://developer.nvidia.com/embedded/buy/jetson-xavier-devkit

[18] *Preview Road Vehicles Functional Safety*, Standard ISO 26262-9:2011, ISO, Geneva, Switzerland, 2011. Accessed: Oct. 19, 2023. [Online]. Available: https://www.iso.org/standard/51365.html

[19] J. Perez-Cerrolaza, J. Abella, L. Kosmidis, A. J. Calderon, F. Cazorla, and J. L. Flores, "GPU devices for safety-critical systems: A survey," *ACM Comput. Surv.*, vol. 55, no. 7, pp. 1–37, Dec. 2022.

[20] S. Sumit. (2018). *A Comprehensive Guide to Convolutional Neural Networks*. Accessed: Oct. 21, 2023. [Online]. Available: https://saturncloud.io/blog/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way/

[21] F. Scarselli and A. C. Tsoi, "Universal approximation using feedforward neural networks: A survey of some existing methods, and some new results," *Neural Netw.*, vol. 11, no. 1, pp. 15–37, Jan. 1998.

[22] P. J. Werbos, "Backpropagation through time: What it does and how to do it," *Proc. IEEE*, vol. 78, no. 10, pp. 1550–1560, Oct. 1990.

[23] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *Proc. 19th Int. Conf. Comput. Statist.* Paris, France. Cham, Switzerland: Springer, 2010, pp. 177–186.

[24] S.University. (2023). *CS231n Convolutional Neural Networks for Visual Recognition*. Accessed: Oct. 21, 2023. [Online]. Available: https://cs231n.github.io/convolutional-networks/

[25] A. F. Agarap, "Deep learning using rectified linear units (ReLU)," 2018, *arXiv:1803.08375*.

[26] Y. Bengio, I. Goodfellow, and A. Courville, *Deep Learning*, vol. 1. Cambridge, MA, USA: MIT Press, 2017.

[27] G. Klambauer, T. Unterthiner, A. Mayr, and S. Hochreiter, "Self-normalizing neural networks," 2017, *arXiv:1706.02515*.

[28] D. Hendrycks and K. Gimpel, "Gaussian error linear units (GELUs)," 2016, *arXiv:1606.08415*.

[29] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "MobileNetV2: Inverted residuals and linear bottlenecks," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, Salt Lake City, UT, USA, Jun. 2018, pp. 4510–4520.

[30] S. Santurkar, D. Tsipras, A. Ilyas, and A. Mądry, "How does batch normalization help optimization?" in *Proc. 32nd Int. Conf. Neural Inf. Process. Syst.* Red Hook, NY, USA: Curran Associates, 2018, pp. 2488–2498.

[31] T. Spyrou and H.-G. Stratigopoulos, "On-line testing of neuromorphic hardware," in *Proc. IEEE Eur. Test Symp. (ETS)*, Venice, Italy, May 2023, pp. 99–105.

[32] T. Spyrou, S. A. El-Sayed, E. Afacan, L. A. Camuñas-Mesa, B. Linares-Barranco, and H.-G. Stratigopoulos, "Reliability analysis of a spiking neural network hardware accelerator," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Antwerp, Belgium, Mar. 2022, pp. 370–375.

[33] H.-G. Stratigopoulos, T. Spyrou, and S. Raptis, "Testing and reliability of spiking neural networks: A review of the state-of-the-art," in *Proc. IEEE Int. Symp. Defect Fault Tolerance VLSI Nanotechnol. Syst. (DFT)*, Juan-les-Pins, France, Oct. 2023, pp. 1–6.

[34] T. P. Xiao et al., "Ionizing radiation effects in SONOS-based neuromorphic inference accelerators," *IEEE Trans. Nucl. Sci.*, vol. 68, no. 5, pp. 762–769, May 2021.

[35] K. Chellapilla, S. Puri, and P. Simard, "High performance convolutional neural networks for document processing," in *Proc. 10th Int. Workshop Frontiers Handwriting Recognit.* La Baule, France: Université de Rennes, Oct. 2006, pp. 1–7.

[36] A. Vasudevan, A. Anderson, and D. Gregg, "Parallel multi channel convolution using general matrix multiplication," in *Proc. IEEE 28th Int. Conf. Appl.-Specific Syst., Archit. Processors (ASAP)*, New York, NY, USA, Jul. 2017, pp. 19–24.

[37] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, Mar. 2008.

[38] W. mei W. Hwu, D. B. Kirk, and I. El Hajj, *Programming Massively Parallel Processors*, 4th ed. Burlington, MA, USA: Kaufmann, 2023.

[39] Xilinx. *Zynq-7000 All Programmable SoC*. Accessed: Sep. 11, 2022. [Online]. Available: https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html

[40] C. He, A. Papakonstantinou, and D. Chen, "A novel SoC architecture on FPGA for ultra fast face detection," in *Proc. IEEE Int. Conf. Comput. Design*, Lake Tahoe, CA, USA, Oct. 2009, pp. 412–418.

[41] P. Q. Dzung, L. D. Khoa, H. H. Lee, L. M. Phuong, and N. T. D. Vu, "The new MPPT algorithm using ANN-based PV," in *Proc. Int. Forum Strategic Technol.*, Ulsan, South Korea, Oct. 2010, pp. 402–407.

[42] A. Rahnamaei, N. Pariz, and A. Akbarimajd, "FPGA implementation of an ANN for detection of anthelmintics resistant nematodes in sheep flocks," in *Proc. 4th IEEE Conf. Ind. Electron. Appl.*, Xi'an, China, May 2009, pp. 1899–1904.

[43] H. Bui and S. Tahar, "Design and synthesis of an IEEE-754 exponential function," in *Proc. Eng. Solutions Next Millennium. IEEE Can. Conf. Electr. Comput. Eng.*, vol. 1, Edmonton, AB, Canada, May 1999, pp. 450–455.

[44] C. C. Doss and R. L. Riley, "FPGA-based implementation of a robust IEEE-754 exponential unit," in *Proc. 12th Annu. IEEE Symp. Field-Program. Custom Comput. Mach.*, Napa, CA, USA, Apr. 2004, pp. 229–238.

[45] H. Amin, K. M. Curtis, and B. R. Hayes-Gill, "Piecewise linear approximation applied to nonlinear function of a neural network," *IEE Proc. Circuits, Devices Syst.*, vol. 144, no. 6, pp. 313–317, Dec. 1997.

[46] P. Leekul, P. Soontornwong, and S. Chivapreecha, "Low complexity artificial neural network unit for sugar content detection in microwave sensor system," in *Proc. Signal Inf. Process. Assoc. Annu. Summit Conf. (APSIPA), Asia–Pacific*, Krong Siem Reap, Cambodia, Dec. 2014, pp. 1–4.

[47] J.-K. Kim, M.-Y. Lee, J.-Y. Kim, B.-J. Kim, and J.-H. Lee, "An efficient pruning and weight sharing method for neural network," in *Proc. IEEE Int. Conf. Consum. Electron.-Asia (ICCE-Asia)*, Seoul, South Korea, Oct. 2016, pp. 1–2.

[48] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized neural networks: Training neural networks with low precision weights and activations," *J. Mach. Learn. Res.*, vol. 18, no. 1, pp. 6869–6898, Jan. 2017.

[49] C. Yuan and S. S. Agaian, "A comprehensive review of binary neural network," *Artif. Intell. Rev.*, vol. 56, no. 11, pp. 12949–13013, Nov. 2023.

[50] H. T. Kung and E. L. Charles, "Systolic arrays (for VLSI)," in *Sparse Matrix Proceedings, 1978*, vol. 1. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1978, pp. 256–285.

[51] *Measurement and Reporting of Alpha Particle and Terrestrial Cosmic Ray-Induced Soft Errors in Semiconductor Devices*, Standard JESD89A, JEDEC, Arlington County, VA, USA, 2006.

[52] J. F. Ziegler and H. Puchner, *SER—History, Trends and Challenges: A Guide for Designing With Memory ICs*. San Jose, CA, USA: Cypress, 2004.

[53] R. Baumann, "Soft errors in advanced computer systems," *IEEE Design Test Comput.*, vol. 22, no. 3, pp. 258–266, May 2005.

[54] D. M. Fleetwood, "Total-ionizing-dose effects, border traps, and 1/f noise in emerging MOS technologies," *IEEE Trans. Nucl. Sci.*, vol. 67, no. 7, pp. 1216–1240, Jul. 2020.

[55] D. M. Fleetwood, "Evolution of total ionizing dose effects in MOS devices with Moore's law scaling," *IEEE Trans. Nucl. Sci.*, vol. 65, no. 8, pp. 1465–1481, Aug. 2018.

[56] J. R. Srour and J. M. McGarrity, "Radiation effects on microelectronics in space," *Proc. IEEE*, vol. 76, no. 11, pp. 1443–1469, Nov. 1988.

[57] R. C. Baumann, "Soft errors in advanced semiconductor devices—Part I: The three radiation sources," *IEEE Trans. Device Mater. Rel.*, vol. 1, no. 1, pp. 17–22, Mar. 2001.

[58] S. Buchner, M. Baze, D. Brown, D. Mcmorrow, and J. Melinger, "Comparison of error rates in combinational and sequential logic," *IEEE Trans. Nucl. Sci.*, vol. 44, no. 6, pp. 2209–2216, Dec. 1997.

[59] N. N. Mahatme et al., "Comparison of combinational and sequential error rates for a deep submicron process," *IEEE Trans. Nucl. Sci.*, vol. 58, no. 6, pp. 2719–2725, Dec. 2011.

[60] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Trans. Depend. Secure Comput.*, vol. 1, no. 1, pp. 11–33, Jan. 2004.

[61] T. Fawcett, "An introduction to ROC analysis," *Pattern Recognit. Lett.*, vol. 27, no. 8, pp. 861–874, Jun. 2006.

[62] G. Li et al., "Understanding error propagation in deep learning neural network (DNN) accelerators and applications," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.* New York, NY, USA: Association for Computing Machinery, Nov. 2017, pp. 84–96.

[63] A. Lesea, S. Drimer, J. J. Fabula, C. Carmichael, and P. Alfke, "The Rosetta experiment: Atmospheric soft error rate testing in differing technology FPGAs," *IEEE Trans. Device Mater. Rel.*, vol. 5, no. 3, pp. 317–328, Sep. 2005.

[64] D. Tiwari et al., "Understanding GPU errors on large-scale HPC systems and the implications for system design and operation," in *Proc. IEEE 21st Int. Symp. High Perform. Comput. Archit. (HPCA)*, San Francisco, CA, USA, Feb. 2015, pp. 331–342.

[65] V. Sridharan et al., "Memory errors in modern systems: The good, the bad, and the ugly," in *Proc. 20th Int. Conf. Architectural Support Program. Lang. Operating Syst.* New York, NY, USA: Association for Computing Machinery, Mar. 2015, pp. 297–310.

[66] N. Seifert, X. Zhu, and L. W. Massengill, "Impact of scaling on soft-error rates in commercial microprocessors," *IEEE Trans. Nucl. Sci.*, vol. 49, no. 6, pp. 3100–3106, Dec. 2002.

[67] D. Oliveira et al., "Experimental and analytical study of Xeon Phi reliability," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.* New York, NY, USA: ACM, Nov. 2017, pp. 1–12.

[68] S. Tselonis and D. Gizopoulos, "GUFI: A framework for GPUs reliability assessment," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, Uppsala, Sweden, Apr. 2016, pp. 90–100.

[69] B. Fang, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi, "GPU-Qin: A methodology for evaluating the error resilience of GPGPU applications," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, Monterey, CA, USA, Mar. 2014, pp. 221–230.

[70] J. Wei, A. Thomas, G. Li, and K. Pattabiraman, "Quantifying the accuracy of high-level fault injection techniques for hardware faults," in *Proc. 44th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.* Atlanta, GA, USA: IEEE Computer Society, Jun. 2014, pp. 375–382.

[71] NVLABS. (2020). *NVBitFI: An Architecture-level Fault Injection Tool for GPU Application Resilience Evaluations*. Accessed: Sep. 11, 2022. [Online]. Available: https://github.com/NVlabs/nvbitfi

[72] S. K. S. Hari, T. Tsai, M. Stephenson, S. W. Keckler, and J. Emer, "SASSIFI: An architecture-level fault injection tool for GPU application resilience evaluation," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, Santa Rosa, CA, USA, Apr. 2017, pp. 249–258.

[73] B. Nie, L. Yang, A. Jog, and E. Smirni, "Fault site pruning for practical reliability analysis of GPGPU applications," in *Proc. 51st Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Fukuoka, Japan, Oct. 2018, pp. 749–761.

[74] O. Subasi, C.-K. Chang, M. Erez, and S. Krishnamoorthy, "Characterizing the impact of soft errors affecting floating-point ALUs using RTL-level fault injection," in *Proc. 47th Int. Conf. Parallel Process.* New York, NY, USA: Association for Computing Machinery, Aug. 2018, p. 59.

[75] A. Vallero, D. Gizopoulos, and S. Di Carlo, "SIFI: AMD southern islands GPU microarchitectural level fault injector," in *Proc. IEEE 23rd Int. Symp. On-Line Test. Robust Syst. Design (IOLTS)*, Thessaloniki, Greece, Jul. 2017, pp. 138–144.

[76] A. Chatzidimitriou, P. Bodmann, G. Papadimitriou, D. Gizopoulos, and P. Rech, "Demystifying soft error assessment strategies on ARM CPUs: Microarchitectural fault injection vs. neutron beam experiments," in *Proc. 49th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Portland, OR, USA, Jun. 2019, pp. 26–38.

[77] A. Ejlali, S. G. Miremadi, H. Zarandi, G. Asadi, and S. B. Sarmadi, "A hybrid fault injection approach based on simulation and emulation co-operation," in *Proc. Int. Conf. Dependable Syst. Netw.*, San Francisco, CA, USA, Jun. 2003, pp. 479–488.

[78] J. E. R. Condia, B. Du, M. S. Reorda, and L. Sterpone, "FlexGripPlus: An improved GPGPU model to support reliability analysis," *Microelectron. Rel.*, vol. 109, Jun. 2020, Art. no. 113660.

[79] M. A. Kochte et al., "Efficient simulation of structural faults for the reliability evaluation at system-level," in *Proc. 19th IEEE Asian Test Symp.*, Denver, CO, USA, Dec. 2010, pp. 3–8.

[80] H. T. Nguyen, Y. Yagil, N. Seifert, and M. Reitsma, "Chip-level soft error estimation method," *IEEE Trans. Device Mater. Rel.*, vol. 5, no. 3, pp. 365–381, Sep. 2005.

[81] C. Constantinescu, "Impact of deep submicron technology on dependability of VLSI circuits," in *Proc. Int. Conf. Dependable Syst. Netw.*, Washington, DC, USA, Jun. 2002, pp. 205–209.

[82] X. Iturbe, B. Venu, and E. Ozer, "Soft error vulnerability assessment of the real-time safety-related ARM cortex-R5 CPU," in *Proc. IEEE Int. Symp. Defect Fault Tolerance VLSI Nanotechnol. Syst. (DFT)*, Storrs, CT, USA, Sep. 2016, pp. 91–96.

[83] J. Blome, S. Mahlke, D. Bradley, and K. Flautner, "A microarchitectural analysis of soft error propagation in a production-level embedded microprocessor," in *Proc. 1st Workshop Archit. Rel.*, 2005, pp. 1–8.

[84] H. Cho, S. Mirkhani, C.-Y. Cher, J. A. Abraham, and S. Mitra, "Quantitative evaluation of soft error injection techniques for robust system design," in *Proc. 50th ACM/EDAC/IEEE Design Autom. Conf. (DAC)*, Austin, TX, USA, May 2013, pp. 1–10.

[85] M. Maniatakos, N. Karimi, C. Tirumurti, A. Jas, and Y. Makris, "Instruction-level impact analysis of low-level faults in a modern microprocessor controller," *IEEE Trans. Comput.*, vol. 60, no. 9, pp. 1260–1273, Sep. 2011.

[86] N. J. Wang, A. Mahesri, and S. J. Patel, "Examining ACE analysis reliability estimates using fault-injection," *ACM SIGARCH Comput. Archit. News*, vol. 35, no. 2, pp. 460–469, Jun. 2007.

[87] A. Chatzidimitriou et al., "RT level vs. microarchitecture-level reliability assessment: Case study on ARM(R) Cortex(R)-A9 CPU," in *Proc. 47th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. Workshops (DSN-W)*, Denver, CO, USA, Jun. 2017, pp. 117–120.

[88] C. Constantinescu, M. Butler, and C. Weller, "Error injection-based study of soft error propagation in AMD bulldozer microprocessor module," in *Proc. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Boston, MA, USA, Jun. 2012, pp. 1–6.

[89] G. Yalcin, O. S. Unsal, A. Cristal, and M. Valero, "FIMSIM: A fault injection infrastructure for microarchitectural simulators," in *Proc. IEEE 29th Int. Conf. Comput. Design (ICCD)*, Amherst, MA, USA, Oct. 2011, pp. 431–432.

[90] A. Chatzidimitriou and D. Gizopoulos, "Anatomy of microarchitecture-level reliability assessment: Throughput and accuracy," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, Uppsala, Sweden, Apr. 2016, pp. 69–78.

[91] M. Kaliorakis, S. Tselonis, A. Chatzidimitriou, N. Foutris, and D. Gizopoulos, "Differential fault injection on microarchitectural simulators," in *Proc. IEEE Int. Symp. Workload Characterization*, Oct. 2015, pp. 172–182.

[92] G. S. Rodrigues and F. L. Kastensmidt, "Soft error analysis at sequential and parallel applications in ARM cortex-A9 dual-core," in *Proc. 17th Latin-Amer. Test Symp. (LATS)*, Foz do Iguaçu, Brazil, Apr. 2016, pp. 147–152.

[93] F. Rosa, F. Kastensmidt, R. Reis, and L. Ost, "A fast and scalable fault injection framework to evaluate multi/many-core soft error reliability," in *Proc. IEEE Int. Symp. Defect Fault Tolerance VLSI Nanotechnol. Syst. (DFTS)*, Amherst, MA, USA, Oct. 2015, pp. 211–214.

[94] D. Ferraretto and G. Pravadelli, "Simulation-based fault injection with QEMU for speeding-up dependability analysis of embedded software," *J. Electron. Test.*, vol. 32, no. 1, pp. 43–57, Jan. 2016.

[95] F. de Aguiar Geissler, F. L. Kastensmidt, and J. E. P. Souza, "Soft error injection methodology based on QEMU software platform," in *Proc. 15th Latin Amer. Test Workshop (LATW)*, Foz do Iguaçu, Brazil, Mar. 2014, p. 179.

[96] A. Höller, G. Macher, T. Rauter, J. Iber, and C. Kreiner, "A virtual fault injection framework for reliability-aware software development," in *Proc. IEEE Int. Conf. Dependable Syst. Netw. Workshops*, Rio de Janeiro, Brazil, Jun. 2015, pp. 69–74.

[97] L. Wanner, S. Elmalaki, L. Lai, P. Gupta, and M. Srivastava, "VarEMU: An emulation testbed for variability-aware software," in *Proc. Int. Conf. Hardw./Softw. Codesign Syst. Synth. (CODES+ISSS)*, Montreal, QC, Canada, Sep. 2013, pp. 1–10.

[98] M. Becker, D. Baldin, C. Kuznik, M. M. Joy, T. Xie, and W. Mueller, "XEMU: An efficient QEMU based binary mutation testing framework for embedded software," in *Proc. 10th ACM Int. Conf. Embedded Softw.*, New York, NY, USA, Oct. 2012, pp. 33–42.

[99] R. Amarnath, S. N. Bhat, P. Munk, and E. Thaden, "A fault injection approach to evaluate soft-error dependability of system calls," in *Proc. IEEE Int. Symp. Softw. Rel. Eng. Workshops (ISSREW)*, Memphis, TN, USA, Oct. 2018, pp. 71–76.

[100] Z. Chen, G. Li, K. Pattabiraman, and N. DeBardeleben, "BinFI: An efficient fault injector for safety-critical machine learning systems," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.* New York, NY, USA: Association for Computing Machinery, Nov. 2019, p. 69.

[101] A. L. Sartor, P. H. E. Becker, and A. C. S. Beck, "A fast and accurate hybrid fault injection platform for transient and permanent faults," *Des. Autom. Embedded Syst.*, vol. 23, nos. 1–2, pp. 3–19, Jun. 2019.

[102] S. Nimara, A. Amaricai, O. Boncalo, and M. Popa, "Multi-level simulated fault injection for data dependent reliability analysis of RTL circuit descriptions," *Adv. Electr. Comput. Eng.*, vol. 16, no. 1, pp. 93–98, 2016.

[103] H. Cho, C.-Y. Cher, T. Shepherd, and S. Mitra, "Understanding soft errors in uncore components," in *Proc. 52nd ACM/EDAC/IEEE Design Autom. Conf. (DAC)*, San Francisco, CA, USA, Jun. 2015, pp. 1–6.

[104] F. F. D. Santos, S. K. S. Hari, P. M. Basso, L. Carro, and P. Rech, "Demystifying GPU reliability: Comparing and combining beam experiments, fault simulation, and profiling," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, Portland, OR, USA, May 2021, pp. 289–298.

[105] P. R. Bodmann, D. Oliveira, and P. Rech, "Accurate FIT rate estimation through high-level software fault injection," *IEEE Trans. Nucl. Sci.*, vol. 69, no. 9, pp. 2018–2026, Sep. 2022.

[106] ESA. (2016). *ESCC Basic Specification No. 22900*. Accessed: Sep. 11, 2022. [Online]. Available: http://escies.org/escc-specs/published/22900.pdf

[107] V. Girones et al., "The use of high-energy X-ray generators for TID testing of electronic devices," *IEEE Trans. Nucl. Sci.*, vol. 70, no. 8, pp. 1982–1989, Aug. 2023.

[108] N. Rezzak, J.-J. Wang, V. Nguyen, and D. Dsilva, "Combined X-ray and gamma ray testing to investigate the TID tolerance of flip-chip FPGAs," in *Proc. 17th Eur. Conf. Radiat. Effects Compon. Syst. (RADECS)*, Geneva, Switzerland, Oct. 2017, pp. 1–7.

[109] A. C. Daniel and G. R. Allen, "Heavy-ion test results of several commercial components for use in a class D interplanetary mission payload," in *Proc. IEEE Radiat. Effects Data Workshop (REDW)*, Waikoloa, HI, USA, Jul. 2018, pp. 1–5.

[110] A. D. Topper et al., "NASA Goddard space flight center's compendium of total ionizing dose, displacement damage dose, and single-event effects test results," in *Proc. IEEE Radiat. Effects Data Workshop*, San Antonio, TX, USA, Jul. 2019, pp. 1–10.

[111] K. A. Label. *Are Current See Test Procedures Adequate for Modern Devices and Electronics Technologies*. Accessed: Sep. 11, 2022. [Online]. Available: https://radhome.gsfc.nasa.gov/radhome/papers/HEART08_LaBel_pres.pdf

[112] E. J. Wyrwas. (2019). *Standardizing Microprocessor and GPU Radiation Test Approaches*. Accessed: Sep. 11, 2022. [Online]. Available: https://ntrs.nasa.gov/api/citations/20190028690/downloads/20190028690.pdf

[113] B. Du, J. E. R. Condia, M. S. Reorda, and L. Sterpone, "On the evaluation of SEU effects in GPGPUs," in *Proc. IEEE Latin Amer. Test Symp. (LATS)*, Santiago, Chile, Mar. 2019, pp. 128–134.

[114] P. R. Bodmann, G. Papadimitriou, R. L. R. Junior, D. Gizopoulos, and P. Rech, "Soft error effects on arm microprocessors: Early estimations versus chip measurements," *IEEE Trans. Comput.*, vol. 71, no. 10, pp. 2358–2369, Oct. 2022.

[115] O. Ballan et al., "Evaluation of ISO 26262 and IEC 61508 metrics for transient faults of a multi-processor system-on-chip through radiation testing," *Microelectron. Rel.*, vol. 107, Apr. 2020, Art. no. 113601.

[116] M. D. Berg. (2018). *A New Approach to System-Level Single Event Survivability Prediction*. Accessed: Sep. 11, 2022. [Online]. Available: https://ntrs.nasa.gov/api/citations/20190002701/downloads/20190002701.pdf

[117] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor," in *Proc. 22nd Digit. Avionics Syst. Conf.* Washington, DC, USA: IEEE Computer Society, Dec. 2003, pp. 29–36.

[118] V. Sridharan and D. R. Kaeli, "Eliminating microarchitectural dependency from architectural vulnerability," in *Proc. IEEE 15th Int. Symp. High Perform. Comput. Archit.*, Raleigh, NC, USA, Feb. 2009, pp. 117–128.

[119] C. Weaver, J. Emer, S. Mukherjee, and S. Reinhardt, "Techniques to reduce the soft error rate of a high-performance microprocessor," in *Proc. 31st Annu. Int. Symp. Comput. Archit.*, Munich, Germany, Jun. 2004, pp. 264–275.

[120] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, D. I. August, and S. S. Mukherjee, "Design and evaluation of hybrid fault-detection systems," in *Proc. 32nd Int. Symp. Comput. Archit. (ISCA)*, Madison, WI, USA, Jun. 2005, pp. 148–159.

[121] UC Group. (Jun. 2016). *Carol Radiation Setup*. Accessed: Sep. 11, 2022. [Online]. Available: https://github.com/radhelper

[122] P. Rech et al., "Neutron-induced soft errors in graphic processing units," in *Proc. IEEE Radiat. Effects Data Workshop*, Miami, FL, USA, Jul. 2012, pp. 136–142.

[123] P. Rech, T. D. Fairbanks, H. M. Quinn, and L. Carro, "Threads distribution effects on graphics processing units neutron sensitivity," *IEEE Trans. Nucl. Sci.*, vol. 60, no. 6, pp. 4220–4225, Dec. 2013.

[124] P. Rech, C. Aguiar, C. Frost, and L. Carro, "An efficient and experimentally tuned software-based hardening strategy for matrix multiplication on GPUs," *IEEE Trans. Nucl. Sci.*, vol. 60, no. 4, pp. 2797–2804, Aug. 2013.

[125] L. L. Pilla et al., "Software-based hardening strategies for neutron sensitive FFT algorithms on GPUs," *IEEE Trans. Nucl. Sci.*, vol. 61, no. 4, pp. 1874–1880, Aug. 2014.

[126] D. A. G. D. Oliveira et al., "Radiation-induced error criticality in modern HPC parallel accelerators," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Austin, TX, USA, Feb. 2017, pp. 577–588.

[127] V. Fratin, D. Oliveira, C. Lunardi, F. Santos, G. Rodrigues, and P. Rech, "Code-dependent and architecture-dependent reliability behaviors," in *Proc. 48th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Luxembourg, Europe, Jun. 2018, pp. 13–26.

[128] P. M. Basso, F. F. D. Santos, and P. Rech, “Impact of tensor cores and mixed precision on the reliability of matrix multiplication in GPUs,” *IEEE Trans. Nucl. Sci.*, vol. 67, no. 7, pp. 1560–1565, Jul. 2020.

[129] F. F. dos Santos, C. Lunardi, D. Oliveira, F. Libano, and P. Rech, “Reliability evaluation of mixed-precision architectures,” in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Washington, DC, USA, Feb. 2019, pp. 238–249.

[130] K. Ito, Y. Zhang, H. Itsuji, T. Uezono, T. Toba, and M. Hashimoto, “Analyzing DUE errors on GPUs with neutron irradiation test and fault injection to control flow,” *IEEE Trans. Nucl. Sci.*, vol. 68, no. 8, pp. 1668–1674, Aug. 2021.

[131] M. Hashimoto, Y. Zhang, and K. Ito, “Neutron-induced stuck error bits and their recovery in DRAMs on GPU cards,” in *Proc. Int. Conf. Solid State Devices Mater. (SSDM)*, Chiba, Japan, 2022, pp. 24–36.

[132] F. Libano, P. Rech, L. Tambara, J. Tonfat, and F. Kastensmidt, “On the reliability of linear regression and pattern recognition feedforward artificial neural networks in FPGAs,” *IEEE Trans. Nucl. Sci.*, vol. 65, no. 1, pp. 288–295, Jan. 2018.

[133] F. Libano, B. Wilson, M. Wirthlin, P. Rech, and J. Brunhaver, “Understanding the impact of quantization, accuracy, and radiation on the reliability of convolutional neural networks on FPGAs,” *IEEE Trans. Nucl. Sci.*, vol. 67, no. 7, pp. 1478–1484, Jul. 2020.

[134] S. Blower, P. Rech, C. Cazzaniga, M. Kastriotou, and C. D. Frost, “Evaluating and mitigating neutrons effects on COTS EdgeAI accelerators,” *IEEE Trans. Nucl. Sci.*, vol. 68, no. 8, pp. 1719–1726, Aug. 2021.

[135] P. Maillard, Y. P. Chen, J. Arver, V. Merugu, A. Shui, and M. Voogel, “Neutron and 64MeV proton characterization of Xilinx 7nm VersalTM multicore scalar processing system (PS),” in *Proc. IEEE Radiat. Effects Data Workshop (REDW) (Conjunct NSREC)*, Kansas City, MO, USA, Jul. 2022, pp. 18–23.

[136] Q. Cheng et al., “Reliability exploration of system-on-chip with multi-bit-width accelerator for multi-precision deep neural networks,” *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 70, no. 10, pp. 3978–3991, Oct. 2023.

[137] J. M. Badia, G. Leon, J. A. Belloch, M. Garcia-Valderas, A. Lindoso, and L. Entrena, “Comparison of parallel implementation strategies in GPU-accelerated system-on-chip under proton irradiation,” *IEEE Trans. Nucl. Sci.*, vol. 69, no. 3, pp. 444–452, Mar. 2022.

[138] J. M. Badia et al., “Reliability evaluation of LU decomposition on GPU-accelerated system-on-chip under proton irradiation,” *IEEE Trans. Nucl. Sci.*, vol. 69, no. 7, pp. 1467–1474, Jul. 2022.

[139] E. J. Wyrwas and C. Szabo. *Proton Testing of AMD v1202b System on Chip*. Accessed: Sep. 11, 2022. [Online]. Available: https://nepp.nasa.gov/files/30378/NEPP-TR-2019-Wyrwas-NEPPweb-TR-19-024-AMD-2200G-Microprocessor-2019June02-TN72756.pdf

[140] P. Maillard, Y. P. Chen, J. Barton, and M. L. Voogel, “Single event latchup (SEL) and single event upset (SEU) evaluation of Xilinx 7 nm Versal^{TM} ACAP programmable logic (PL),” in *Proc. IEEE Nucl. Space Radiat. Effects Conf. (NSREC)*, Vienna, Austria, Jul. 2021, pp. 41–46.

[141] P. Maillard et al., “Heavy-ion and proton evaluation of AMD 7nm Versal^{TM} multicore scalar processing system (PS),” in *Proc. IEEE Radiat. Effects Data Workshop (REDW) (Conjunct NSREC)*, Kansas City, MO, USA, Jul. 2023, pp. 56–62.

[142] E. J. Wyrwas and T. Wilcox. *Radiation Effects and Analysis Lessons: A Scientist’s Field Instruction to Explain Radiation Testing*. Accessed: Sep. 11, 2022. [Online]. Available: https://nepp.nasa.gov/workshops/etw2022/talks/16-JUN-THU/1600-Wyrwas-20220002512.pdf

[143] R. K. Barry et al., “CLEoPATRA: Contemporaneous lensing parallax and autonomous transient assay,” *Proc. SPIE*, vol. 12180, Aug. 2022, Art. no. 121800D.

[144] Á. B. de Oliveira, O. Lexell, and F. Sturesson, “Multi chips heavy ions SEE testing of the COTS Myriad-2 vision processing unit,” in *Proc. 21st Eur. Conf. Radiat. Effects Compon. Syst. (RADECS)*, Vienna, Austria, Sep. 2021, pp. 301–305.

[145] L. Buckley, A. Dunne, G. Furano, and M. Tali, “Radiation test and in orbit performance of MpSoC AI accelerator,” in *Proc. IEEE Aerosp. Conf. (AERO)*, Big Sky, MT, USA, Mar. 2022, pp. 199–218.

[146] S. M. Guertin, W. P. Parker, A. C. Daniel, and P. Adell, “Recent SEE results for snapdragon processors,” in *Proc. IEEE Radiat. Effects Data Workshop*, San Antonio, TX, USA, Jul. 2019, pp. 62–157.

[147] S. M. Guertin and M. Cui, “SEE test results for the snapdragon 820,” in *Proc. IEEE Radiat. Effects Data Workshop (REDW)*, New Orleans, LA, USA, Jul. 2017, pp. 155–161.

[148] G. Furano et al., “Towards the use of artificial intelligence on the edge in space systems: Challenges and opportunities,” *IEEE Aerosp. Electron. Syst. Mag.*, vol. 35, no. 12, pp. 44–56, Dec. 2020.

[149] F. F. dos Santos, S. Malde, C. Cazzaniga, C. Frost, L. Carro, and P. Rech, “Experimental findings on the sources of detected unrecoverable errors in GPUs,” *IEEE Trans. Nucl. Sci.*, vol. 69, no. 3, pp. 436–443, Mar. 2022.

[150] F. Libano, P. Rech, and J. Brunhaver, “Efficient error detection for matrix multiplication with systolic arrays on FPGAs,” *IEEE Trans. Comput.*, vol. 72, pp. 2390–2403, 2023.

[151] G. Papadimitriou and D. Gizopoulos, “Demystifying the system vulnerability stack: Transient fault effects across the layers,” in *Proc. ACM/IEEE 48th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2021, pp. 902–915.

[152] Z. Gao et al., “Systematic reliability evaluation of FPGA implemented CNN accelerators,” *IEEE Trans. Device Mater. Rel.*, vol. 23, no. 1, pp. 116–126, Mar. 2023.

[153] B. Du, S. Azimi, C. de Sio, L. Bozzoli, and L. Sterpone, “On the reliability of convolutional neural network implementation on SRAM-based FPGA,” in *Proc. IEEE Int. Symp. Defect Fault Tolerance VLSI Nanotechnol. Syst. (DFT)*, Noordwijk, The, Netherlands, Oct. 2019, pp. 143–149.

[154] F. F. D. Santos, J. E. R. Condia, L. Carro, M. S. Reorda, and P. Rech, “Revealing GPUs vulnerabilities by combining register-transfer and software-level fault injection,” in *Proc. 51st Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Jun. 2021, pp. 292–304.

[155] A. Ruospo et al., “Assessing convolutional neural networks reliability through statistical fault injections,” in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Antwerp, Belgium, Apr. 2023, pp. 354–359.

[156] J. E. R. Condia and M. S. Reorda, “Testing permanent faults in pipeline registers of GPGPUs: A multi-kernel approach,” in *Proc. IEEE 25th Int. Symp. On-Line Test. Robust Syst. Design (IOLTS)*, Rhodes, Greece, Jul. 2019, pp. 97–102.

[157] M. T. Sanic, C. Guo, J. Leng, M. Guo, and W. Ma, “Towards reliable AI applications via algorithm-based fault tolerance on NVDLA,” in *Proc. 18th Int. Conf. Mobility, Sens. Netw. (MSN)*, Guangzhou, China, Dec. 2022, pp. 736–743.

[158] Y. He, P. Balaprakash, and Y. Li, “FIdelity: Efficient resilience analysis framework for deep learning accelerators,” in *Proc. 53rd Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2020, pp. 270–281.

[159] A. Ruospo, A. Balaara, A. Bosio, and E. Sanchez, “A pipelined multi-level fault injector for deep neural networks,” in *Proc. IEEE Int. Symp. Defect Fault Tolerance VLSI Nanotechnol. Syst. (DFT)*, Frascati, Italy, Oct. 2020, pp. 1–6.

[160] G. León, J. M. Badía, J. A. Belloch, A. Lindoso, and L. Entrena, “Evaluating the soft error sensitivity of a GPU-based SoC for matrix multiplication,” *Microelectron. Rel.*, vol. 114, Nov. 2020, Art. no. 113856.

[161] A. Ruospo, E. Sanchez, M. Traiola, I. O’Connor, and A. Bosio, “Investigating data representation for efficient and reliable convolutional neural networks,” *Microprocessors Microsyst.*, vol. 86, Oct. 2021, Art. no. 104318.

[162] Y. Zhang, H. Itsuji, T. Uezono, T. Toba, and M. Hashimoto, “Estimating vulnerability of all model parameters in DNN with a small number of fault injections,” in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*. Leuven, Belgium: European Design and Automation Association, Mar. 2022, pp. 60–63.

[163] C. Bolchini, L. Cassano, A. Miele, and A. Toschi, “Fast and accurate error simulation for CNNs against soft errors,” *IEEE Trans. Comput.*, vol. 72, no. 4, pp. 984–997, Apr. 2023.

[164] J. D. G. Balaguera, J. E. R. Condia, F. F. Dos Santos, M. S. Reorda, and P. Rech, “Understanding the effects of permanent faults in GPU’s parallelism management and control units,” in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.* Denver, CO, USA: Association for Computing Machinery, Nov. 2023, p. 46.

[165] J. E. Navarro. (2021). *High-Performance Compute Board—A Fault-Tolerant Module for On-Boards Vision Processing*. Accessed: Nov. 6, 2023. [Online]. Available: https://zenodo.org/record/5521624/files/10.05_OBDP2021_Espana_Navarro.pdf?download=1

[166] D. A. G. Oliveira, P. Rech, L. L. Pilla, P. O. A. Navaux, and L. Carro, “GPGPUs ECC efficiency and efficacy,” in *Proc. IEEE Int. Symp. Defect Fault Tolerance VLSI Nanotechnol. Syst. (DFT)*, Oct. 2014, pp. 209–215.

[167] L. K. Draghetti, F. F. D. Santos, L. Carro, and P. Rech, "Detecting errors in convolutional neural networks using inter frame spatio-temporal correlation," in *Proc. IEEE 25th Int. Symp. On-Line Test. Robust Syst. Design (IOLTS)*, Jul. 2019, pp. 310–315.

[168] D. Sabena, M. S. Reorda, L. Sterpone, P. Rech, and L. Carro, "On the evaluation of soft-errors detection techniques for GPGPUs," in *Proc. 8th IEEE Design Test Symp.*, Morocco. Marrakesh, Dec. 2013, pp. 97–103.

[169] D. A. G. Oliveira et al., "Modern GPUs radiation sensitivity evaluation and mitigation through duplication with comparison," *IEEE Trans. Nucl. Sci.*, vol. 61, no. 6, pp. 3115–3122, Dec. 2014.

[170] L. Weigel, F. Fernandes, P. Navaux, and P. Rech, "Kernel vulnerability factor and efficient hardening for histogram of oriented gradients," in *Proc. IEEE Int. Symp. Defect Fault Tolerance VLSI Nanotechnol. Syst. (DFT)*, Cambridge, U.K., Oct. 2017, pp. 1–6.

[171] A. Ruospo, G. Gavarini, I. Bragaglia, M. Traiola, A. Bosio, and E. Sanchez, "Selective hardening of critical neurons in deep neural networks," in *Proc. 25th Int. Symp. Design Diag. Electron. Circuits Syst. (DDECS)*, Prague, Czech Republic, Apr. 2022, pp. 136–141.

[172] C. Bolchini, L. Cassano, A. Miele, and A. Nazzari, "Selective hardening of CNNs based on layer vulnerability estimation," in *Proc. IEEE Int. Symp. Defect Fault Tolerance VLSI Nanotechnol. Syst. (DFT)*, Austin, TX, USA, Oct. 2022, pp. 142–148.

[173] S. K. S. Hari, M. B. Sullivan, T. Tsai, and S. W. Keckler, "Making convolutions resilient via algorithm-based error detection techniques," *IEEE Trans. Dependable Secure Comput.*, vol. 19, no. 4, pp. 2546–2558, Jul. 2022.

[174] C. Bolchini, L. Cassano, A. Miele, and M. Biasielli, "Lightweight fault detection and management for image restoration," in *Proc. IEEE Int. Symp. Defect Fault Tolerance VLSI Nanotechnol. Syst. (DFT)*, Frascati, Italy, Oct. 2020, pp. 55–61.

[175] F. F. dos Santos et al., "Reduced precision DWC: An efficient hardening strategy for mixed-precision architectures," *IEEE Trans. Comput.*, vol. 71, no. 3, pp. 573–586, Mar. 2022.

[176] M. Biasielli, C. Bolchini, L. Cassano, E. Koyuncu, and A. Miele, "A neural network based fault management scheme for reliable image processing," *IEEE Trans. Comput.*, vol. 69, no. 5, pp. 764–776, May 2020.

[177] M. Biasielli, C. Bolchini, L. Cassano, A. Mazzeo, and A. Miele, "Approximation-based fault tolerance in image processing applications," *IEEE Trans. Emerg. Topics Comput.*, vol. 10, no. 2, pp. 648–661, Apr. 2022.

[178] N. Cavagnero, F. D. Santos, M. Ciccone, G. Averta, T. Tommasi, and P. Rech, "Transient-fault-aware design and training to enhance DNNs reliability with zero-overhead," in *Proc. IEEE 28th Int. Symp. On-Line Test. Robust Syst. Design (IOLTS)*, Italy, Sep. 2022, pp. 1–7.

[179] R. L. Rech and P. Rech, "Reliability of Google's tensor processing units for embedded applications," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*. Leuven, Belgium: European Design and Automation Association, Mar. 2022, pp. 376–381.

[180] G. Gambardella, N. J. Fraser, U. Zahid, G. Furano, and M. Blott, "Accelerated radiation test on quantized neural networks trained with fault aware training," in *Proc. IEEE Aerosp. Conf. (AERO)*, Big Sky, MT, USA, Mar. 2022, pp. 1966–1973.

Open Access funding provided by 'Università degli Studi di Trento' within the CRUI CARE Agreement