Efficient Error Detection for Matrix Multiplication with Systolic Arrays on FPGAs

Fabiano Libano, Paolo Rech, *Senior Member, IEEE,* and John Brunhaver, *Member, IEEE*

Abstract—Matrix multiplication has always been a cornerstone in computer science. In fact, linear algebra tools permeate a wide variety of applications: from weather forecasting, to financial market prediction, radio signal processing, computer vision, and more. Since many of the aforementioned applications typically impose strict performance and/or fault tolerance constraints, the demand for fast and reliable matrix multiplication (MxM) is at an all-time high. Typically, increased reliability is achieved through redundancy. However, coarse-grain duplication incurs an often prohibitive overhead, higher than 100%. Thanks to the peculiar characteristics of the MxM algorithm, more efficient algorithm-based hardening solutions have been designed to detect (and even correct) some types of errors with lower overhead. We show that, despite being more efficient, current solutions are still sub-optimal in certain scenarios, particularly when considering persistent faults in Field-Programmable Gate-Arrays (FPGAs). Based on a thorough analysis of the fault model, we propose an error detection technique for MxM that decreases both algorithmic and architectural costs by over a polynomial degree, when compared to existing algorithm-based strategies. Furthermore, we report arithmetic overheads at the application level to be under 1% for three state-of-the-art Convolutional Neural Networks (CNNs).

Index Terms—Error Detection, Matrix Multiplication, Systolic Array, FPGA.

1 Introduction

From high-performance computing, to low-latency radio, and image processing tasks, matrix operations are commonly at the core of a variety of compute-intensive algorithms. Scientific applications, such as climate and particle physics simulations [1], compute their solutions by solving large systems of equations, which frequently reduce to matrix multiplication. Matrix operations are so representative and relevant in computer science, that the list of the top 500 supercomputers in the world is determined by execution time figures of benchmarks mostly composed of linear algebra calculations [2]. Additionally, useful computer vision tasks, like edge-detection and image segmentation can be accomplished using the convolution operation, which by its turn can be expressed as an equivalent MxM [3].

Recent advances in the Machine Learning (ML) field have led to significant improvements in the accuracy of Convolutional Neural Networks (CNNs), making them attractive alternatives to more traditional image processing algorithms [4]. Such improved capability for object-detection and classification tasks sparked the interest of the trillion dollar automotive industry to deploy CNNs in autonomous cars [5]. Likewise, military organizations have increased fleets of Unmanned Aerial Vehicles (UAVs) [6], while the space exploration sector is investing in autonomous rovers and helicopters for missions on Mars [7]. As the aforementioned applications fall into the *real-time safety-critical* category, where failures can lead to human casualties and/or substantial losses of capital, the need for *fast and reliable* computer vision (matrix multiplication) becomes fundamental [8] [9].

In order to comply with the performance constraints imposed by real-time applications, compute intensive workloads based on matrix multiplication require some sort of hardware acceleration. Graphics Processing Units (GPUs) offer a highly parallel general purpose architecture, and design flexibility through optimized software libraries. In fact GPUs are the most commom device for matrix multiplication, mainly due to the General Matrix Multiplication (GEMM) algorithm, which is tightly coupled with the device's memory hierarchy [10]. However, GPUs are still general purpose, which means that some performance is sacrificed for genericism. Furthermore, GPUs are also known to be power-hungry, which prohibits their deployment in power-constrained scenarios, such as space missions [11]. Alternatively, Application-Specific Integrated-Circuits (ASICs) can be designed to deliver top-tier performance and energy efficiency. However, once fabricated, there is little to no design flexibility to be explored, which can be a problem for long-life consumer products (such as self-driving cars, for example, with system improvements and bug fixes being considerably more challenging). On the other hand, Field-Programmable Gate Arrays (FPGAs) deliver high performance along with full hardware reprogrammability and low power consumption [12]. Such characteristics make FPGAs very attractive candidates for accelerating real-time applications and, thus, are the focus of our study.

Unfortunately, computing devices are prone to experiencing radiation-induced faults, potentially manifesting as errors and failures at the application and system level, respectively. Specifically in the case of SRAM-based FPGAs, the main concerns are upsets in their configuration mem-

- *F. Libano and J. Brunhaver are with Arizona State University (ASU), Tempe, AZ, 85287.*
  E-mail: {flibano, jbrunhaver}@asu.edu
- *P. Rech is with Federal University of Rio Grande do Sul (UFRGS), Porto Alegre, RS, Brazil, and with Polytechnic of Torino (PoliTO), Turin, TO, Italy.*
  E-mail: prech@{inf.ufrgs.br, polito.it}

Manuscript received November 1, 2021; revised November 1, 2021.

ories [13], which can disrupt the functioning of the logic circuit implemented on the fabric, ultimately leading to untrustworthy computation. As a practical example, a faulty DSP element within a systolic array architecture can generate errors in a matrix multiplication, which can then lead to misdetections in a CNN, potentially causing a vehicle to crash. Another very important peculiarity regarding the fault model of FPGAs is that the aforementioned configuration faults are persistent until actively corrected (through the loading of a new bitstream or scrubbing [14]). To put it another way, after a fault has been installed, every subsequent computation running on top of such faulty circuit is very likely to also present output errors. Therefore, it becomes of paramount importance to quickly and cheaply detect erroneous behaviors, to then trigger repairing mechanisms on the FPGA's configuration memory, ultimately restoring functional correctness in the system.

In order to achieve mandatory levels of reliability for safety-critical applications, computing systems must employ appropriate hardening techniques. Most commonly, modular redundancy can be considered with different granularities (i.e. multiple devices, circuit replication [15]). The number of redundant modules also determines whether the system is able to *detect* or *correct* errors: Duplication With Comparison (DWC) allows for single error detection [16], while Triple Modular Redundancy (TMR) allows for single error correction [17]. In fact, the gold-standard hardening strategy for FPGAs is a combination of circuit-level TMR along with active scrubbing of configuration bits [18]. However, it is not always possible to triplicate a circuit that utilizes a high number of resources as is. Furthermore, the adoption of modular redundancy is also known to hinder performance, since routing tasks become more difficult as fabric congestion increases [19], and, in the case of DWC, the added redundancy is also prone to producing *false detections* (i.e. when the output is correct, but a detection is wrongly signaled) [20] [21]. As an alternative, Algorithm-Based Fault Tolerance (ABFT) can be used to increase the reliability of specific applications with a lower overhead than traditional modular redundancy, but, as we will discuss in later sections, existing ABFT methods [22] [23] are still too expensive for FPGAs. We show that, in order to maximize hardening efficiency (i.e. deliver maximum error detection rates with minimum added costs), one must first obtain an accurate depiction of the fault model.

In pursuance of addressing the aforementioned issues, and advancing the state-of-the-art towards reliable matrix-multiplication-based computation, the main contributions in our paper are:

- A multi-level fault propagation model, that provides an overview of how faults transition across abstraction layers, considering the following case-study stack: FPGA - Systolic Array - Matrix Multiplication - CNN.
- A novel ABFT technique for low overhead error detection in matrix multiplication operations, tailored for systolic arrays on SRAM-based FPGAs.
- A state-of-the-art, open-source, systolic array implementation for Xilinx FPGAs, along with an RTL code generator that allows for seamless deployment of hardened matrix multiplication cores.

The remainder of the paper is organized as follows. Section 2 provides a background on radiation effects in FPGAs and the systolic computation paradigm, while hinting why this superposition significantly impacts the fidelity of existing fault models. Section 3 presents architectural details of our case-study, state-of-the-art, systolic array implementation. Section 4 explains our fault-injection experimental methodology, while Section 5 provides an in-depth analysis of how faults propagate across levels of abstraction, highlighting the importance of appropriate hardening mechanisms along the way. Section 6 then presents our novel error detection strategy for matrix multiplication operations, while explaining the reasons why it is the most sensible choice for FPGA-accelerated systems. Section 7 contains an experimental validation of our idea, which confirms our formal demonstration. Section 8 concludes the paper and briefly mentions future work opportunities.

2 Background And Related Work

2.1 Radiation Effects on Computing Devices

Radiation is a naturally occurring phenomenon that can simply be viewed as transmission of energy. Due to 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 away, but sufficiently energetic cosmic rays collide with nuclei in our atmosphere, producing a variety of secondary particles, including alpha, protons, gamma, and, mainly, neutrons [13]. For example, the neutron flux at sea level has been measured to be of about $13n/(cm^2 \cdot h)$ [24].

Since computing devices are made out of silicon, radiation is a threat. Ionizing particles generate electron-hole pairs within the transistor's oxide, eventually releasing enough charge to force state changes from *ON* to *OFF* (or the other way around) [25]. Non-ionizing radiation (neutrons) on the other hand, does not deposit any charge, but instead leaves a trail of ionization on the silicon as it passes through the device, creating charged particles that then lead to state changes [26]. Furthermore, the advances in fabrication processes and overall scaling of technology have allowed for reduced transistor sizes, increased transistor density, and reduced operating voltages, which in turn led to an even greater radiation sensitivity [27].

Ultimately, each type of particle will interact with each type of computing device in a distinct manner. In particular, as our work focuses on SRAM-based FPGAs, the vast majority of Single-Event Effects (SEEs) are actually Single-Event Upsets (SEUs) on the FPGA's configuration memory [13]. SEUs are *transient* faults, in the sense that the device is not permanently damaged, but the information stored in memory or the output of operations is wrong. Such corruptions, when affecting the configuration memory of an FPGA, can potentially modify the content and settings of logic resources (LUTs, DSPs, BRAMs, FFs), as well as affect routing connections. Furthermore, such SEUs, will exhibit a *persistent effect* until a new (corrected) bitstream is loaded into the FPGA's configuration memory. This specific characteristic of FPGAs fundamentally changes the fault model of hypothetical hardware accelerators which only consider non-reprogrammable devices, in which SEUs only affect the current operation or data. In other words, while on a CPU, GPU, or ASIC the SEU corrupts only one operation, in FPGAs most of SEUs will likely impact all operations that follow its occurrence, up until the point where corrective action is taken, through partial or full reloading of the configuration memory. If left undetected/unmitigated, such hardware faults can propagate to the application level and eventually become Silent Data Corruptions (SDCs).

2.2 Matrix Multiplication on Systolic Arrays

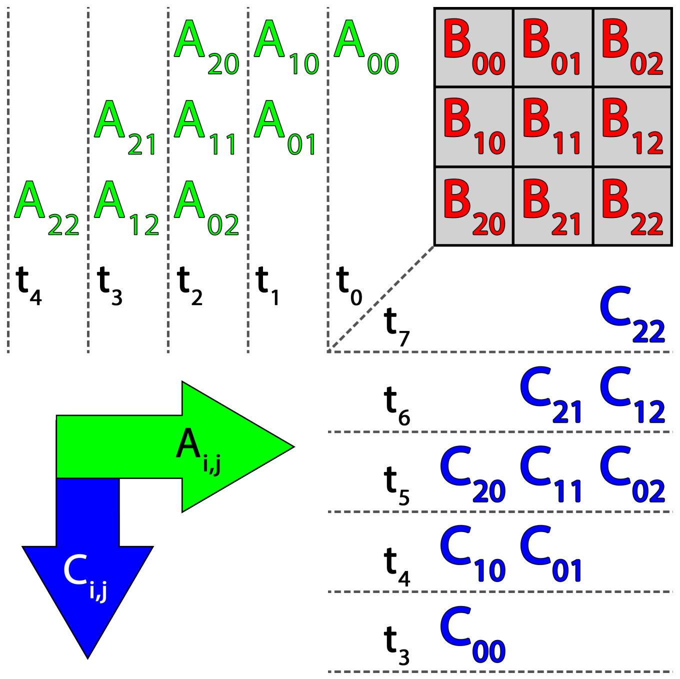
As previously stated, matrix multiplication is a staple in computer science. However, the MxM algorithm is inherently expensive: Assuming square matrices of size M, we need to perform a total of M^3 multiply-accumulate (MAC) operations. Luckily, as there are no data dependencies between output elements, MxM is also extremelly parallelizable. In GPUs, the aforementioned GEMM algorithm cleverly positions data in cache and distributes the workload across available cores. However, input elements must still be read from memory multiple times. Systolic arrays on the other hand establish specific interconnection and data movement patterns between computing units to reduce memory accesses, ultimately having an edge over other architectures in MxM computation.

A systolic array is simply a network of processing elements (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 decades ago by [28], 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 like LU-decomposition and Fourier transform.

Much more recently, there has been an increased interest in systolic computation due to the rise of neural network accelerators, such as Google's Tensor Processing Unit (TPU) [29] and Xilinx's Deep Processing Unit (DPU) [30]. Since the workload of a modern CNN is dominated by convolution (which can be translated as an equivalent MxM [3]) and inner product operations, weight-stationary systolic arrays became the perfect fit for these workloads. Fig. 1 illustrates how MxM can be performed on such an architecture. At each clock cycle, inputs are pumped in. Once the pipeline fills up, outputs start flowing out.

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 simple observation, combined with the aforementioned predominance of persistent faults in FPGAs, massively impacts the fault model (as we will discuss in Section 5), and consequently the relevance/effectiveness of hardening strategies (as we will discuss in Section 6).

Morever, large MxM operations often need to be broken down into smaller *blocks*, ahead of their execution. In other words, large input matrices are partitioned, and execution produces pieces of the output in a step-by-step fashion. This characteristic becomes particularly useful in GPUs when it comes to fitting just the right amount of data into each cache level, as a way of minimizing latency in memory operations. Likewise, in the context of systolic accelerators for FPGAs/ASICs, the size of the array is set at implementation time. Effectively this means that, in order to execute sizeable matrix operations, multiple sub-matrices must pass through the array. This, of course, also impacts the fault model and the inherent overhead of hardening solutions.



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

Figure 1. The functioning of a generic NxN weight-stationary systolic array for matrix multiplication. The calculation in this example is *AxB=C*. The values of *B* are pre-loaded 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.

2.3 Related Work

Traditional Hardening for FPGAs: Strategies for increasing the reliability of SRAM-based FPGA systems have existed for long time, as FPGAs were within the first high-performance devices to be deployed in space [13]. Generally, increased hardness is achieved through modular redundancy, such as TMR (circuit triplication followed by majority voting) [17], and DMR/DWC (circuit duplication followed by comparison) [16]. In fact, the number of redundant modules determines the number of simultaneous faults that can be tolerated: An N-Modular-Redundant (NMR) system is able to detect *N-1* and correct *N-2* faults. Moreover, as the upsets in the FPGA's configuration memory tend to have a persistent effect, modular redundancy must be paired with active scrubbing [14] [18], in order to prevent the accumulation of faults over time. However, these circuit replication strategies are not always feasible (depending on original circuit complexity and target device capabilities), as they inherently incur significant overheads in resource utilization (100N+% for NMR). Furthermore, modular redundancy is also known to negatively affect performance, as the increased fabric congestion makes routing efforts more difficult [19]. Finally, we must highlight that, although DWC theoretically delivers an error detection rate of 100%, by doubling the circuit's area it also introduces a high probability of false detections [20] [21]. Moreover, if the checker module itself is not hardened (i.e. with a dual rail scheme [16]), it can become the root cause for undetected errors in the system.

Algorithm-Based Fault Tolerance: ABFT techniques go beyond naive redundancy as they tend to provide similar error detection and correction capabilities, but with considerably lower overheads. This is because good ABFT strategies are typically based on a deep knowledge of the algorithm's characteristics and fault model. In fact, the first ever ABFT scheme was proposed specifically for error detection and correction on matrix operations, back in the 80s [22]. We will revisit the authors contributions in the following sections, while highlighting two of its shortcomings (lack of consideration for persistent faults and underestimation of algorithmic/architectural overheads). More recently, an extended version of the MxM ABFT [23] addressed the original's limited correction capability (single errors only). By adding a number of extra steps (and computation) to the data reconstruction process, authors have reported correction of a wider variety of error patterns. We, however, go in the opposite direction of the spectrum, eliminating the overhead necessary for error correction, and solely focusing on extremely low-cost detection. This is mainly because, as the majority of faults in FPGAs have a persistent effect, without (partially) reconfiguring the device all subsequent computations would very likely trigger the algorithmic correction procedures. Moreover, for not-so-harsh deployment environments with a low expected error rate (i.e. terrestrial), the cost of recovery (recomputation) is diluted across all the error-free executions. In simpler words, it is preferable to pay a slightly higher price every once in a while, instead of paying for insurance every time.

Application-Specific Fault Tolerance: Although we have previously mentioned a diverse number of MxM-based applications, the current prominence of CNNs, and the desire to use them in real-time safety-critical applications, has significantly skewed research interest towards the proposal of CNN-specific hardening strategies. Most notably, authors have very recently presented a low-cost, checksum-based, error detection technique for convolution operations, and integrated their solution within standard ML flows to facilitate the deployment of hardened CNNs in state-of-the-art GPUs [31]. The authors reported arithmetic overheads between 1% and 7%, while runtime overheads were between 6% and 23% in their case-study neural networks. An experimental demonstration of the technique's ability to detect all errors in convolutional layers was also provided. Such work builds on top of the algebraic observations made by [32], regarding convolution. The authors of [32] were particularly interested in mitigating faults derived from overclocking, as opposed to radiation-induced. Furthermore, the authors of [33] have implemented the aforementioned MxM ABFT in convolutional and inner product layers of CNNs, while also proposing a novel ABFT technique for pooling operations, ultimately achieving over 90% correction on critical SDCs when executing state-of-the-art neural network topologies in GPUs. Finally, from a more computer-vision perspective, authors have exploited spatio-temporal correlations in CNNs' input image frames, achieving over 80% error detection while adding less than 5% of runtime overhead in a CPU execution [34].

Fault Modeling: Also worth noting are some works predominantly concerned with fault modeling in a variety of scenarios. For instance, the authors of [35] have investigated the error patterns that emerged in matrix multiplication as a result of spatially and/or temporally separated faults in arithmetic and/or memory components, within the context of GPU-based clusters for High-Performance Computing (HPC). Some of the error patterns discussed by the authors are also observed in our experiments, despite occurring for different reasons, to be explained in Section 5. From another standpoint, the authors of [36] have analyzed the potential impact of permanent fabrication defects in a TPU-like systolic array, by injecting faults directly at the gate-level netlist of a synthesized design. The authors reported that, after introducing only 0.005% of *faultyness* to the array, the test accuracy on their case-study neural network dropped from 74% to 39%. Moreover, the authors of [37] have conducted software-based fault injection experiments corrupting weight values of floating-point and fixed-point variants of case-study CNNs. In this case, authors have reported weight corruptions in convolutional layers to be significantly more likely to generate classification errors than weight corruptions in inner product layers, due to the inherent weight reutilization that exists in convolution. Our fault model analysis is dissimilar to these works mainly because the persistent faults in FPGAs tend to alter the *functioning* of the circuit, as opposed to simply disturbing the inputs/outputs. Moreover, we further emphasize the impact of fault propagation across the computing stack, up to the application level.

3 A State-Of-The-Art Systolic Array Implementation on FPGAs

In this section we present details of our high-performance systolic array implementation on FPGAs. Even though this is not our main contribution, it is a necessary component of our work, as we later use it as a Design Under Test (DUT) on two separate occasions: First, to construct the fault model that is presented in Section 5. Second, to experimentally validate our novel error detection strategy in Section 7. Nonetheless, we have decided to make our implementation publicly available, along with a parametric RTL code generator (https://github.com/lllibano/LABFT) that allows anyone to deploy custom-sized arrays in a matter of minutes. Fig. 2 comprehensively illustrates our discussion.

There are two main particularities about our implementation (both of which are closely tied to state-of-the-art CNN accelerators such as Google's TPU [29] and Xilinx's DPU [30]. First, we should point out that our inputs are *8-bit integers*. This is because industry-standard ML frameworks, offer lossless quantization tools for reducing the precision of data representation in trained models. In other words, after the training process is complete (using 32-bit floating point), the model goes through an additional step, in which it is converted to an 8-bit integer equivalent, while maintaining the original accuracy level. Both TPU and DPU also use 8-bit integers as inputs. Second, we shall mention that our MxM array is *weight-stationary*. This simply means that each element of the second input matrix is loaded into its respective $PE_{i,j}$, before the first input matrix starts flowing into the array (instead of both moving together). Having one stationary matrix mimics the behavior of convolutional filters in a CNN, and allows for optimal DSP utilization.

For matrix multiplication, the PEs in the systolic array must be MAC units. On modern FPGAs, DSP slices are the fastest arithmetic elements. As such, DSPs are the center-point of our PEs. We simultaneously employ two techniques that allow us to extract top-tier performance out of each DSP. Such techniques are also known to be used in Xilinx's DPU [30]. First, we take advantage of the adder that precedes the multiplier (conveniently named pre-adder) to execute $(A + B) \cdot E$. Given that the inputs are 8 bits wide, and that the pre-adder has a bitwidth of 25 and 27 (for DSP48E1 and DSP48E2, respectively [38]), it is possible to pack two 8-bit integer multiplications (AE, BE) into a single DSP, merely through clever shifting and slicing of bits. This DSP implementation is pretty much the same as [39]. Second, we adopt a standard time-multiplexing scheme for the DSPs, as suggested by Xilinx [38]. With proper muxing and instantiation of registers, we are able to run the DSPs at twice the clock frequency used by the rest of the circuit. The combined effect of these two techniques is a multiplier with a *parallelism factor of 4* (i.e. it *simultaneously* calculates AE, BE, CE, DE). To match such throughput, each PE also has four adders that are implemented with abundant CLB elements (LUTs and CARRYs). At the top level, our implementation simply accommodates the degree of parallelism established by the PEs. In other words, there are four simultaneous matrix calculations being executed by the systolic array, at all times (*W=AxE*, *X=BxE*, *Y=CxE*, *Z=DxE*).

In terms of resource utilization, we should highlight that on top of being the fastest, DSP slices are also the most scarce type of resource in Xilinx FPGAS. Therefore an efficient FPGA accelerator must make good use of them. Looking at Fig. 2, we can immediately see that our systolic arrays are *N*-squared, and that each processing element uses a DSP. Therefore, DSP utilization is predictable and parametric: an *N*-sized array will use N^2 *DSP48E*{*1,2*}.

In terms of performance, each of the N^2 PEs executes 4 multiplications and 4 additions at each clock cycle, totaling $8N^2$ *Operations Per Cycle (OPC)*. Then, if we multiply OPC by frequency, we get the Giga Operations Per Second (GOPS) metric. Since the maximum operating frequency depends on target device, GOPS will vary. However, OPC exclusively depends on array size. In order to prove that our implementation is representative of the state-of-the-art, we will compare two of Xilinx's DPU configurations (one *B1152* core, and three *B4096* cores) [30] to two of our design combinations (one *14x14* array, and two *32x32* arrays). These two DPU configurations (*B1152*x1 and *B4096*x3) were chosen because they respectively max out the resources on a low-end (XC7Z020-3) and on a high-end (XCZU9EG-3) Xilinx FPGA device [30]. In order to conduct a fair comparison, our design combinations were constrained to the DSP budgets of those same devices. Although not explicitly declared by Xilinx, the DPU product guide suggests array sizes of *12x12* and *16x16* for *B1152* and *B4096*, respectively [30]. All numbers are shown in Table 1, and are also publicly available in our repository.

Although our peak theoretical performance is higher than that reported by Xilinx (36% and 19% for the 7 Series

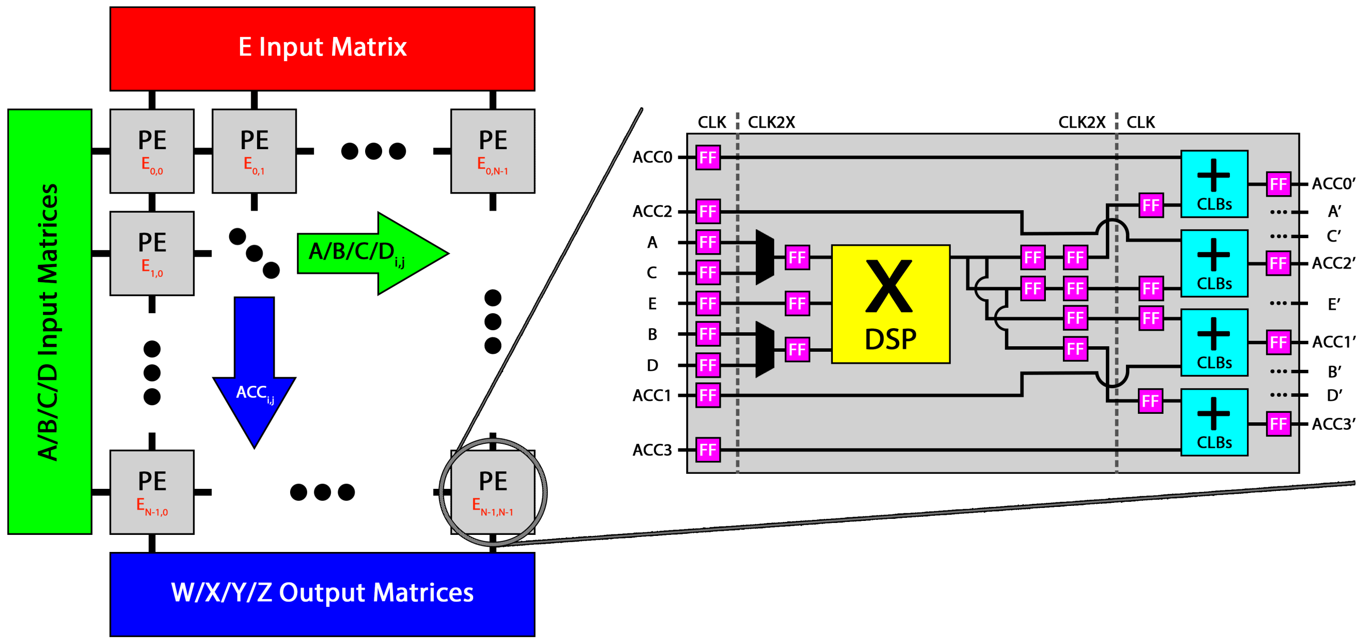


Figure 2. Our systolic array implementation, viewed from different granularity standpoints. At the top level, input and output matrices get in and out of the unit through AXI-Stream interfaces. Inside the array, the values of *E* are pre-loaded into their respective PEs. Then, the values of *A/B/C/D* flow from left to right, while the accumulation results are propagated from top to bottom. Inside each PE, we combine clever shifting and slicing of bits, with a standard time-multiplexing scheme, to achieve an effective throughput of 4 multiplications per cycle, using a single DSP. In order to match such degree of parallelism, we use CLBs to implement 4 adders.

and the UltraScale, respectively), so is our DSP utilization (34% and 32%). Therefore, the numbers in Table 1 indicate that our MxM systolic array architecture is very close to (if not past) the state-of-the-art for 8-bit integer computation on modern Xilinx FPGAs.

Table 1. Resource utilization and peak theoretical performance comparison between DPU configurations [30], and our design combinations, in a 7 Series (XC7Z020-3) FPGA, and in an UltraScale (XCZU9EG-3) FPGA.

Device	Design Combo	# DSPs	Frequency	GOPS
XC7Z020-3	*B1152* x 1 [30]	146	200MHz	230.0
	14x14 x 1	196	200MHz	313.6
XCZU9EG-3	*B4096* x 3 [30]	1542	333MHz	4100.0
	32x32 x 2	2048	300MHz	4915.2

4 Fault Injection Methodology

In this section we provide details pertaining to our fault injection setups and methodologies. First, we run injection campaigns on the FPGA, in order to gain insights regarding the fault model of general MxM systolic arrays. Second, we run application-level campaigns, where we inject the earlier observed MxM error patterns, and quantify the criticality of persistent fault propagation in the context of a CNN execution. Both of these steps are included in our multi-level fault propagation model (Section 5). Finally, we use our FPGA fault injection setup an additional time, to experimentally validate the effectiveness of our novel error detection strategy against persistent configuration upsets (Section 7).

4.1 FPGA Fault Injection Setup

In order to emulate SEUs in the FPGA's configuration memory, as well-established and well-validated in prior work, we take advantage of the Internal Configuration Access Port (ICAP) [40] that is present in modern Xilinx FPGAs. More specifically, since we have chosen the XC7Z020 device from Xilinx's Zynq-7000 family, we are able to use its heterogeneity to facilitate our fault injection campaigns. The XC7Z020 is a System-On-Chip (SOC), composed of a Programmable Logic (PL) and pair of ARM A9 processors, which Xilinx calls Processing System (PS). In practice, this means that system designers can write C code to run on the PS and easily control units instantiated in the PL. In our case, the PS is used to send fault injection commands to the ICAP-controlling unit, and to send/receive inputs/outputs to/from the Design Under Test (DUT). For each iteration, we also use the PS to perform a comparison of the outputs against pre-computed golden values, as a way of evaluating the effect of every injected fault. Finally, a monitor PC logs all the relevant data for a later, more comprehensive analysis. Fig. 3 shows a summarized flowchart of our fault injection campaigns.

4.2 Application-Level Fault Injection

To better understand fault propagation at the application level, besides flipping configuration bits in the FPGA, we also conduct a separate fault injection experiment. Instead of corrupting a systolic array's architecture and observing

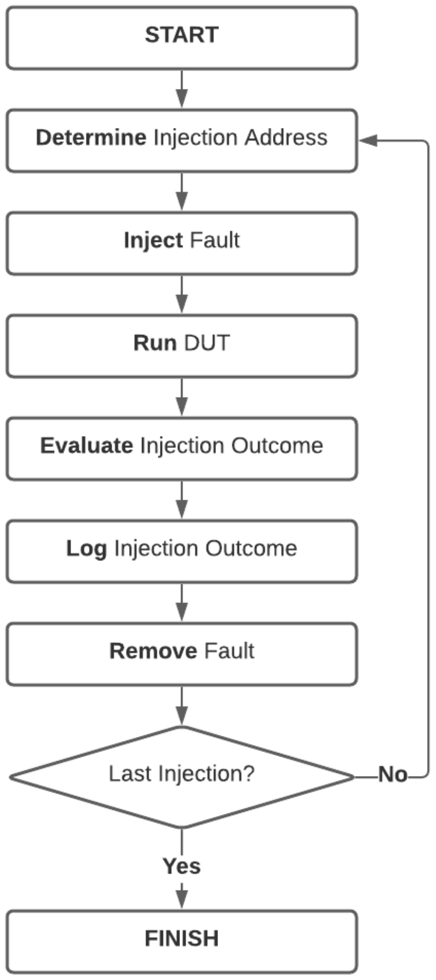


Figure 3. The steps involved in our FPGA fault injection campaigns.

matrix multiplication errors, during this second phase we corrupt data on MxM operations and observe CNN errors (application-level). In order to evaluate how corrupted matrix calculations affect overall CNN computation, we have developed a software-based fault injection methodology. Given case-study CNNs and their topologies, we select (1) target layer and (2) corruption severity. We stop computation halfway through (at target layer), and inject a *persistent* matrix corruption pattern (with the given severity). We then continue the CNN's execution until the end and compare the output against golden values. Our software-based injection emulates the behavior of a systolic-based execution in all MxM operations (breaking them down in appropriately-sized blocks), as we are able to specify array size N as a parameter. Moreover, our injection patterns are *persistent* in the sense that we corrupt all subsequent MxM operations (from target layer onwards), mimicking the behavior of a persistent fault in a systolic architecture. Performing this set of experiments in software (as opposed to hardware) allows for (1) increased injection detail/control, and (2) speed of execution. We will discuss which matrix corruption patterns were injected in our case-study CNNs when we present our multi-level fault propagation model in Section 5.

5 Multi-Level Fault Propagation Model

As previously stated, we are not the first to study the fault model of matrix multiplication [23] [35]. However, prior work mostly describes faulty behavior as a result of transient events in GPUs and CPUs. As our focus is on FPGAs, our model revolves around the incidence of *persistent* faults. Furthermore, we aim to provide a broader context, by discussing the bottom-up propagation of faults/errors/failures across abstraction levels. We analyze a *device, architecture, algorithm, and application* case-study stack, respectively composed by: FPGA, Systolic Array, MxM, and CNN.

From the device (FPGA) perspective, permanent upsets in the configuration memory are the main concern. Whenever configuration bits are affected, the circuit implemented on the fabric can start malfunctioning and providing erroneous outputs. Once the programmable logic's behavior is altered, faults are able to propagate to the architectural level. We should also mention that, technically, logic resources on the FPGA are prone to experiencing transient faults. However, as FPGAs do not run at frequencies nearly as fast as a modern ASICs/CPUs/GPUs, the SET cross section pales in comparison to the SEU cross section of the configuration memory (the former typically being orders of magnitude higher) [41]. Specifically, the SET cross section (i.e. likelihood of capturing transient events) depends on switching frequency because transient spikes at the circuit level need to be latched by edge-triggered register-like elements in order to manifest and propagate from the logic level upwards.

As, in this case, architecture (Systolic Array) and algorithm (MxM) are tightly coupled, we are going to discuss them together. Previous works [23] [35] have already outlined the following MxM error categories from both analytical and experimental standpoints:

- **Single:** only one element is corrupted.
- **Partial Line:** only one row/column is partially corrupted.
- **Full Line:** only one row/column is fully corrupted.
- **Full Matrix:** the entire matrix is corrupted.
- **Random:** none of the above.

Fig. 4 showcases some pattern examples of each of these error categories.

Using the FPGA-based setup described in Section 4.1, we have injected over 618,000 faults in a small 4x4 version of our systolic array. For each injected fault, we have ran ten different matrix multiplication operations, meaning that, effectively, we have performed more than 6 million tests. Approximately 300 thousand of such tests ended with corrupted outputs. Our expected statistical error is ±0.025% (95% confidence). Table 2 shows the distribution of errors within the aforementioned categories. Very clearly, we can see that *Partial Line* and *Full Line* errors account for the vast majority of cases (69.96% to be exact). As we previously discussed, this is due to the pattern of interconnection and data movement in a systolic array, where each processing element contributes to the calculation of multiple output elements. Therefore, whenever an upset alters the functioning of a PE, it has a high probability of affecting multiple outputs that share the same row/column. Namely, after the fault has been installed, every subsequent multiplication/addition in that given PE is likely to produce errors (except for possible input-dependent masking effects). In fact, for over 90% of injections that result in matrix errors, all ten output matrices present corruptions of the same error category. Since the patterns persist across the following operations in the pipeline, in an application context such as a CNN (executing MxM over and over again), stuck-at faults in a systolic array will exhibit a dangerous compounding effect. If left unmitigated, catastrophic failures may occur.

Table 2. The distribution of MxM error categories, as a result of architectural faults injected in a systolic array.

Single	Partial Line	Full Line	Full Matrix	Random
27.07%	43.28%	26.68%	0.25%	2.72%

Prior work has also observed a predominance of line errors (49%) in a GPU context [23]. The authors argued that upsets in the memory hierarchy are the most probable reason why. Likewise, the authors of [35] analytically noted that a single corruption in an input element will cause an entire line (row or column) corruption in the output matrix. As we will later explain in Section 6, the prevalence of line errors is one of the key motivations behind our novel hardening strategy. Finally, we shall state that, although our experiments were performed with a simple 4x4 array, the prevalence of line errors holds for any size *N*, since it is inherently linked to the superposition of persistent faults in PEs and systolic computation itself. In fact, we might intuitively argue that the frequency of line errors shall actually increase with *N*, as it only takes a single faulty PE within the array to induce them.

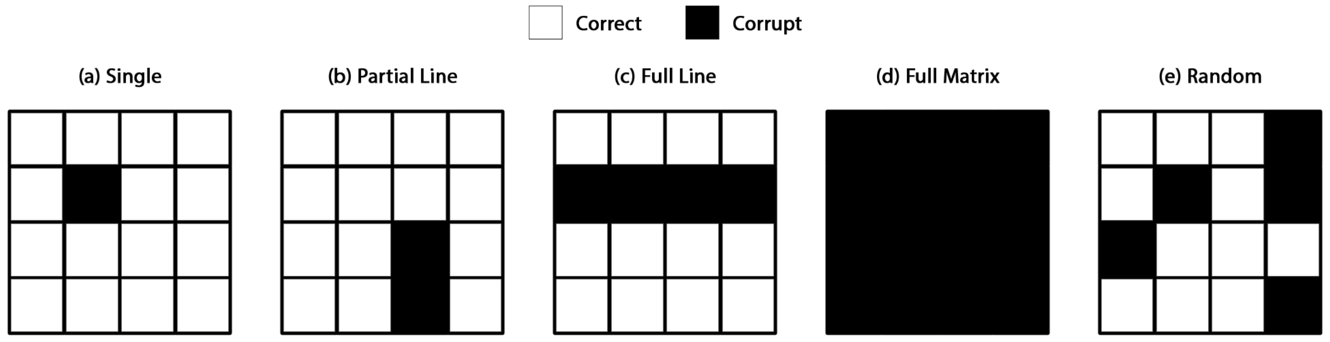


Figure 4. Pattern examples for each of the output error categories in matrix multiplication.

In order to contextualize the severity of propagating matrix multiplication errors in CNNs, we perform application-level fault injection campaigns as described in Section 4.2. As a foundational first case-study, we have chosen the simple and well known LeNet CNN for handwritten digit classification [42].

After observing five well-defined matrix error categories in our FPGA-based injections (Fig. 4 and Table 2), we purposely corrupt matrices in our CNN execution to generate random patterns within those five categories. These patterns are *random* in the sense that, for example, there are 32 different ways of creating a *Full Line* error in a 16x16 matrix. In fact, for generating patterns within *Partial Line* and *Full Line* categories, we first randomly choose between row or column, then randomly choose the line index. Specifically for *Partial Line* patterns, the number of corrupted elements is also randomly chosen (between 2 and the line's width). For injecting *Single* corruptions, we randomly draw row and column indexes. Once the corruption pattern is defined, we apply it to every subsequent MxM operation from the CNN's target layer onwards (thus emulating a persistent fault). It is worth restating (as mentioned in Section 4.2) that all matrix operations are properly broken down into blocks, given the size of the underlying systolic array being emulated in our software-based injection campaign.

We present the results in Fig. 5 measured as Program Vulnerability Factor (PVF) [43] (i.e. the likelihood of an injected fault generating an observable error), subdivided in *Tolerable Errors* and *Critical Errors*, according to whether or not the final output corruption was significant enough to compromise the image classification task. For each bar plotted in Fig. 5, 1000 faults were injected, which establishes a statistical error of around ±6% in our PVF estimates.

Very clearly, PVF is lower and less critical in larger arrays. This is because, given a large MxM operation, the number of sub-blocks required to complete the execution is inversely proportional to the available systolic array size. For instance, if we were to execute a 256x256 operation in a 4x4 array, then we would have to compute 2^{18} 4x4 blocks. Given that a persistent fault exists in the array, then most of the block computations are very likely to exhibit the same error pattern. In other words, a persistent fault in a *KxK* array will cause many more corruptions in a particular matrix multiplication operation than the same persistent fault in a *LxL* array, given *K<L*. Moreover, we see that the PVF increases with the criticality of MxM errors, regardless of array size, which is rather intuitive. For example, a persistent fault generating *Partial Line* errors in a 64x64 array has a 75.6% probability of generating output errors, compared to only 13.7% with a stuck-at *Single* error. This, of course, is also true in terms of criticality: a persistent fault generating *Full Line* errors in a 64x64 array has a 97% probability of generating critical errors in the CNN, compared to only 30.6% with *Partial Line* errors. Finally, we see in Fig. 5 that the earlier the persistent fault is injected, the higher the probability of compromising the CNN's correctness. If we inject a persistent *Full Line* error at the beginning of LeNet's computation (layer 1) using a 256x256 array, the likelihood of observing critical errors is 43.5%, whereas if we only start having faulty MxM operations from layer 2 onwards, that probability diminishes to 10.8%.

As a way of investigating how the criticality of persistent faults would change when increasing the complexity of the CNN, we have conducted additional fault injection experiments with the state-of-the-art 25-layer VGG16 topology [44]. Out of the 25 layers, 16 are MxM-based (13 of which are convolutional and 3 of which are inner product). Once again, we have separately injected random persistent patterns of *Single, Partial Line, Full Line, and Full Matrix* errors in matrix multiplication operations from each of the target layers onwards. Furthermore, each operation is broken down into 256x256 blocks. We present the results in Fig. 6 measured as PVF, subdivided in *Tolerable Errors* and *Critical Errors*. For each bar plotted in Fig. 6, 1000 faults were injected, establishing a statistical error of around ±6%.

We can clearly perceive that, again, the criticality of CNN errors increases with the severity of persistent MxM error categories. In fact, for both *Full Line* and *Full Matrix* patterns, VGG16 always experienced misclassifications, regardless of which first target layer was chosen. In the case of *Single* and *Partial Line* patterns, it is clear that the earlier the fault is installed, the higher the probability of it manifesting as a

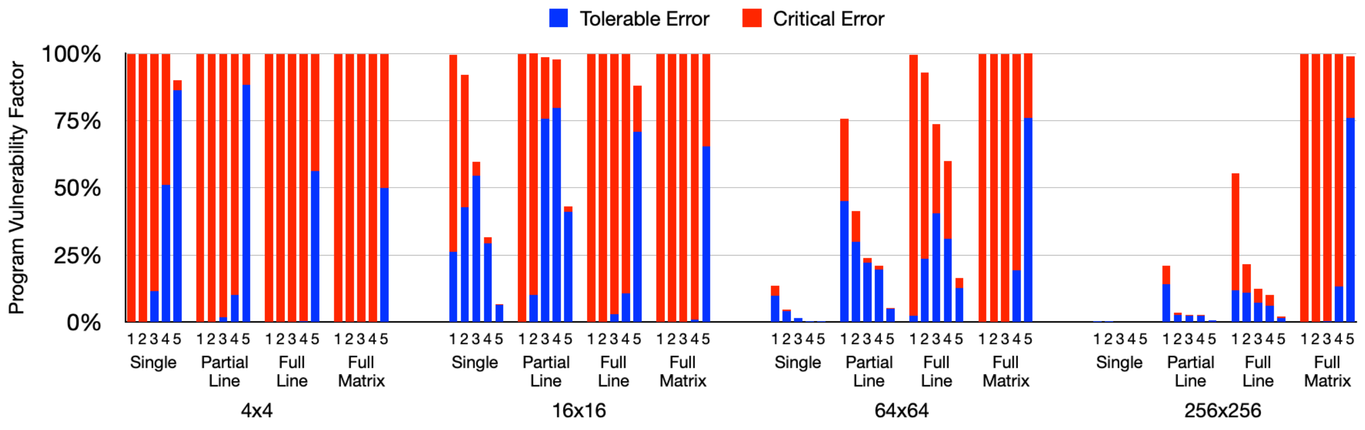


Figure 5. Program Vulnerability Factor (PVF) of the case-study LeNet CNN [42]. Persistent matrix multiplication error patterns were injected at convolutional (1,2,3) and inner product (4,5) layers. Separate campaigns were performed for different systolic array sizes *N={4,16,64,256}*. Application errors are considered tolerable or critical according to whether or not the final output corruption was significant enough to compromise the image classification task.

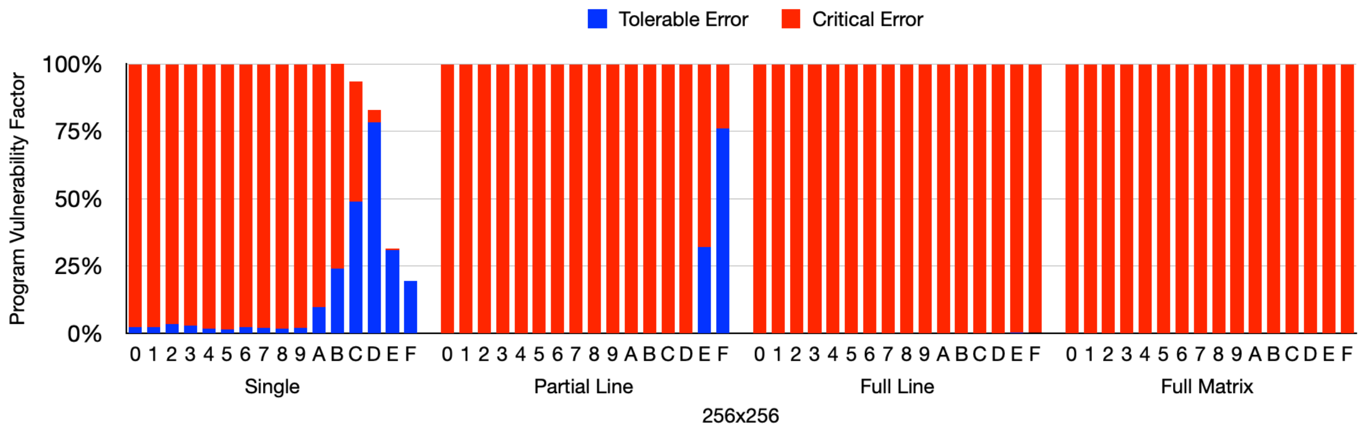


Figure 6. Program Vulnerability Factor (PVF) of the case-study VGG16 CNN [44]. Persistent matrix multiplication error patterns were injected at convolutional (0-C) and inner product (D,E,F) layers. The campaign was performed considering a systolic array size *N=256*. Application errors are considered tolerable or critical according to whether or not the final output corruption was significant enough to compromise the image classification task.

CNN output error (and of such error being considered critical), which was expected, and is now properly quantified. Moreover, we also tested VGG16 considering an underlying 32x32 array. However, with the exception of *Single* errors injected on the MxM computation of the last layer of the network, all test cases led to 99+% probabilities of critical errors. As this graph would not be very insightful, we only report the 256x256 case. Conveniently, as 256 is the dimension of Google's TPU design [29], we believe it to be the single most important scenario to evaluate.

Finally, we must highlight that, as real-time CNNs work at high frame rates, even if a persistent fault is first installed at the last layer of frame f_i, it will very likely affect the computation of the first layer of frame f_{i+1}, and all the following frames. This scenario then further motivates the design of a fast error detection and reconfiguration scheme.

6 Light ABFT

Motivated by the need for low-cost hardening strategies, and the aforementioned predominance of line errors, we present ***Light ABFT***: a lightweight error detection technique for matrix multiplication, specially tailored for high-performance systolic arrays on FPGAs. Next, we shall discuss the idea itself, and its costs of implementation, from algorithmic (arithmetic operations), and architectural (arithmetic units) perspectives. We will also compare our costs to the original and most traditional ABFT [22], and to the most recent work on convolution-specific error detection [31].

6.1 Overview & Formal Demonstration

At the core, the original ABFT algorithm [22] works because of redundancy of information. The calculation and attachment of summation vectors to the input matrices, prior to the execution of matrix multiplication, makes it possible to detect, and potentially correct, arithmetic errors at the end of computation. As it can be seen in Fig. 7, input matrices A and B grow by one row/column respectively.

Given an input matrix A with i rows and j columns, then the j elements in A's summation vector (row i+1) are:

$$SV_{A_z} := A_{i+1,z} = \sum_{x=1}^{i} A_{x,z} \qquad \text{for } 1 \leq z \leq j \tag{1}$$

Given an input matrix B with j rows and k columns, then the j elements in B's summation vector (column k+1) are:

$$SV_{B_z} := B_{z,k+1} = \sum_{y=1}^{k} B_{z,y} \qquad \text{for } 1 \leq z \leq j \tag{2}$$

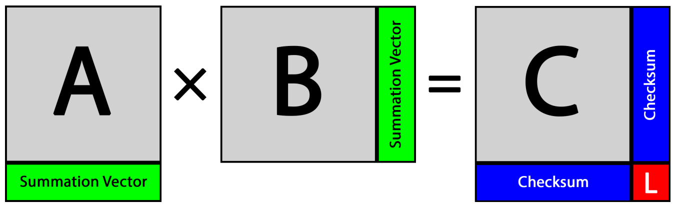


Figure 7. Visual comparison between [22] and Light ABFT in terms of information redundancy and cost. Light ABFT only calculates L, instead of the entire checksum row/column.

Carrying out the calculation with the original ABFT algorithm would then result in an output matrix C with i+1 rows an and k+1 columns. Effectively, this means that an extra i+k+1 elements would have to be computed, on top of the useful workload. As proved by [22], the checksum information can be used to detect errors in C, and, in the case of single errors, to reconstruct corrupted data. However, as per Section 5, when considering systolic arrays in FPGAs, line errors are much more frequent, meaning that the benefit of having error correction capabilities would vanish, most of the time.

Therefore, the fundamental insight behind *Light ABFT* is that, by focusing solely on error detection, it is possible to eliminate a significant amount of unnecessary computation. More specifically, instead of calculating i+k+1 extra output elements, we only calculate one, which is showcased in red with the letter L in Fig. 7. Directly from Fig. 7, we can see that ***L* is the dot product of summation vectors A and B**. At the same time, it is easy to prove that ***L* also equals to the sum of all elements in *C***.

Theorem 6.1.

$$L = SV_A \cdot SV_B = \sum_{x=1}^{i} \sum_{y=1}^{k} C_{x,y} \tag{3}$$

Proof.

The dot product between the summation vectors A and B can be rewritten as:

$$SV_A \cdot SV_B = \sum_{z=1}^{j} SV_{A_z} SV_{B_z} \tag{4}$$

Substituting Eq. 1 and Eq. 2 in Eq. 4 we get:

$$SV_A \cdot SV_B = \sum_{z=1}^{j} (\sum_{x=1}^{i} A_{x,z} \sum_{y=1}^{k} B_{z,y}) \tag{5}$$

Using distributive properties, we can rewrite Eq. 5 as:

$$SV_A \cdot SV_B = \sum_{x=1}^{i} \sum_{y=1}^{k} \sum_{z=1}^{j} A_{x,z} B_{z,y} \tag{6}$$

By definition, each element $C_{x,y}$ of output matrix C is calculated as the dot product between the *x*th row of input matrix A (A_x) and the *y*th column of input matrix B (B_y):

$$C_{x,y} = A_x \cdot B_y = \sum_{z=1}^{j} A_{x,z} B_{z,y} \tag{7}$$

Therefore, it is also true that:

$$\sum_{x=1}^{i} \sum_{y=1}^{k} C_{x,y} = \sum_{x=1}^{i} \sum_{y=1}^{k} \sum_{z=1}^{j} A_{x,z} B_{z,y} \tag{8}$$

Since the right-hand side of Eq. 6 is equal to the right-hand side of Eq. 8, by transitivity we have proved that Eq. 3 is true, as we intended to do.

□

As a consequence of having two distinct ways for calculating *L* (via the inputs, and via the outputs), a simple comparison between the two is sufficient to verify the correctness of the matrix multiplication.

6.2 Cost Analysis

To facilitate the discussion about costs, we shall first enumerate the steps involved in the traditional ABFT [22]:

1) Calculation of summation vectors
2) Matrix multiplication (useful work)
3) Calculation of checksums
4) Error detection
5) Error correction

For each step (with the exception of error correction), we will express cost in algorithmic (raw workload) and architectural (systolic array) terms. Once again, we will consider that input matrix A has *i* rows and *j* columns, and that input matrix B has *j* rows and *k* columns. As a direct consequence of the matrix multiplication algorithm itself, output matrix C has *i* rows and *k* columns.

In order to calculate the summation vectors, we must perform *j* sums of *i* elements for matrix A and *j* sums of *k* elements for matrix B, totaling a cost of *j(i+k)* sums. Architecturally, we need *2j* extra adders to accumulate the summation vectors as data enters the systolic array, while maintaining a fully-streaming behavior.

Given the aforementioned matrix sizes, the intrinsic cost of the matrix multiplication is to perform *j* MACs for each of the *ik* output elements, totaling *j(ik)* MACs. Since the original ABFT grows matrix C to *i+1* rows and *k+1* columns, the cost becomes the calculation of *j* MACs for *(i+1)(k+1)* output elements, totaling *j(ik+i+k+1)*. Therefore, ABFT's added cost is *j(i+k+1)*. Architecturally, the authors of [22] suggest growing the systolic array accordingly, adding a total of *i+k+1* extra PEs. In the case of *Light ABFT*, the sizes of the matrices remain unaltered, and we only calculate *L* as the dot product between the summation vectors, incurring a cost of *j* MAC operations, which can be accommodated with a single extra PE. Interestingly, as [31] pointed out, the increase in size for the input matrices is actually one of the main reasons why Traditional ABFT incurs such a high runtime overhead in GPUs: As a larger GEMM must be executed, inefficiencies in cache arise due to additional online data management procedures, directly translating to increased execution times.

In the error detection phase, the original ABFT algorithm performs *i* accumulations of *k* elements horizontally (row-wise), and *k* accumulations of *i* elements vertically (column-wise), along with *i+k* comparisons against checksum values, totaling *2ik* additions and *i+k* comparisons. The authors do not explicitly suggest an architectural implementation cost, but *i+k* adders and *2* comparators would be needed, at a minimum, to maintain a fully utilized systolic pipeline. In the case of *Light ABFT*, it is necessary to add all the *ik* output elements of *C*, as they come out of the array, and then compare it to *L*. This can be architecturally accomplished with *k+1* adders and a single comparator. It must be mentioned, however, that the original ABFT is able to pin-point the *x* and *y* indexes of the error (which is needed for the following error correction step), while *Light ABFT* only signals that something went wrong (which can then trigger a system-level action, such as moving to a fail-safe state until corrective measures take place).

We summarize the algorithmic and architectural costs of *Light ABFT* in Table 3, while comparing it to [22]. As a way to simplify the data, we assume that a comparison/comparator costs the same as an addition/adder. Moreover, if we assume square matrices of size *M* (i.e. *i=j=k=M*), or square arrays of size *N* (i.e. *i=j=k=N*), we are able to reduce our cost expressions to single variables, making it is easier to perceive the savings of *Light ABFT*. In particular, the architectural cost of *Light ABFT* is made evident in our suggested architectural implementation (Fig. 8).

Previously, in Section 2, we have mentioned that large matrix operations are often completed in steps. Since the dimensions of the systolic array are fixed at implementation time, chunks of input data must pass through the accelerator multiple times to finish computation. Precisely, if we intend to run an operation with square matrices of size *M*, and we choose to break such operation into square blocks of size *B*, with *M*≥*B*, and both *M* and *B* being positive powers of 2, then the number of *B*-sized matrix multiplications to be executed is $(M/B)^3$. This is somewhat intuitive if we recall

Table 3. Costs of [22] compared to Light ABFT, in algorithmic (raw workload) and architectural (systolic array) terms.

Step	Cost	Type	[22]	Light ABFT
1)	Algorithmic	ADD	*j(i+k)*	*j(i+k)*
	Architectural	ADD	*2j*	*2j*
3)	Algorithmic	MUL	*j(i+k+1)*	*j*
		ADD	*j(i+k+1)*	*j*
	Architectural	MUL	*i+k+1*	*1*
		ADD	*i+k+1*	*1*
4)	Algorithmic	ADD	*2ik+i+k*	*ik+1*
	Architectural	ADD	*i+k+2*	*k+2*
Total	Algorithmic	ADD	*2(ik+ji+jk)+i+j+k*	*ik+ji+jk+j+1*
		MUL	*ji+jk+j*	*j*
	Architectural	ADD	*2(i+j+k)+3*	*2j+k+3*
		MUL	*i+k+1*	*1*
i=j=k	Algorithmic	ADD	$6M^2+3M$	$3M^2+M+1$
		MUL	$2M^2+M$	M
	Architectural	ADD	$6N+3$	$3N+3$
		MUL	$2N+1$	1

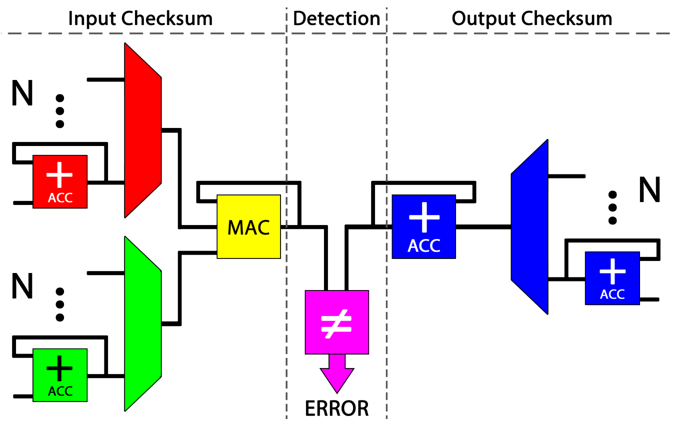


Figure 8. Our suggested Light ABFT architecture for a systolic array context. The array is assumed to be square and of size N. Input matrices enter the unit from the left, while the output matrix enters the unit from the right. The unit then outputs a signal indicating if errors are present in the MxM computation.

that the total number of MAC operations involved in matrix multiplication must be maintained regardless of blocking:

$$M^3 = \left(\frac{M}{B}\right)^3 \cdot B^3 \tag{9}$$

However, when applying error detection techniques to blocks as opposed to the entire matrix, the algorithmic cost expressions shown in Table 3 no longer tell the whole story. In general, for both [22] and *Light ABFT*, we have that the following relationship holds for the hardening cost H:

$$H(M) \leq \left(\frac{M}{B}\right)^3 \cdot H(B) \tag{10}$$

Eq. 10 effectively states that hardening an MxM operation is cheaper than hardening all blocks of that same operation (regardless of block size, since $M/B \leq 1$). As an example, for M=B=4 (i.e. no blocking), the algorithmic cost *H(4)* of *Light ABFT* would be 53 ADDs and 4 MULs. However, for M=4 and B=2, 8 blocks, with hardening cost *H(2)* of 15 ADDs and 2 MULs each, would total 120 ADDs and 16 MULs.

Simultaneously though, block utilization increases the *granularity of checking*, meaning that (1) errors can be detected in the middle of computation, as opposed to just at the end, and (2) the cost of recomputation diminishes, as we do not need to re-run the entire operation after an error detection.

To get a sense of precisely when the benefits of blocking outweigh its added costs, we came up with yet another cost expression F. In addition to matrix size (M), and block size (B), our function F also depends on the expected number of executions between errors (R). The idea is that, if we expect to have, on average, one error after every R executions, then the cost of block recomputation is diluted throughout those R executions. Hence, F actually represents the *failure-free* execution cost, and can be broken in two components. The F_1 component is the cost of R executions:

$$F_1(M,B,R) = R \cdot \left(\left(\frac{M}{B}\right)^3 \cdot (B^3 + H(B))\right) \tag{11}$$

The inner-most parenthesis in Eq. 11 $(B^3 + H(B))$ represents the cost of computing plus hardening a block. This cost is multiplied by the number of blocks $(M/B)^3$, and then by our newly introduced average number of executions between errors R. Therefore, F_1 represents how much math will be done, on average, in between errors.

The F_2 component is simply the cost of recomputing a block:

$$F_2(M,B,R) = 1 \cdot (B^3 + H(B)) \tag{12}$$

Hence, the actual *failure-free* execution cost F is the sum of all arithmetic operations performed in between errors (F_1) with the cost of recomputing a block after a detection (F_2), times the expected rate of errors ($1/R$):

$$F(M,B,R) = \frac{F_1 + F_2}{R} \tag{13}$$

Expanding the numerator in Eq. 13 gives us a final expression in terms of M, B and R:

$$F(M,B,R) = \left(\left(\frac{M}{B}\right)^3 + \frac{1}{R}\right) \cdot (B^3 + H(B)) \tag{14}$$

Notice that, if R tends to infinity (i.e. no errors), the term *1/R* vanishes, symbolizing a lack of recomputation payments.

To ease our discussion, in Fig. 9 we plot the cost F divided by the inherent cost of matrix multiplication (M^3), minus 100%, with *M=256* and increasing values of R. Then in Fig. 10 we plot the same curves, but for a very large matrix size (*M=4096*). As we can see, for each R curve there is a different choice of B that delivers the minimum arithmetic overhead. Moreover, the higher the expected error rate, the sooner that inflection point appears. Given specific M=M_0 and R=R_0, the optimal B is found by solving:

$$\frac{\partial F(M_0,B,R_0)}{\partial B} = 0 \tag{15}$$

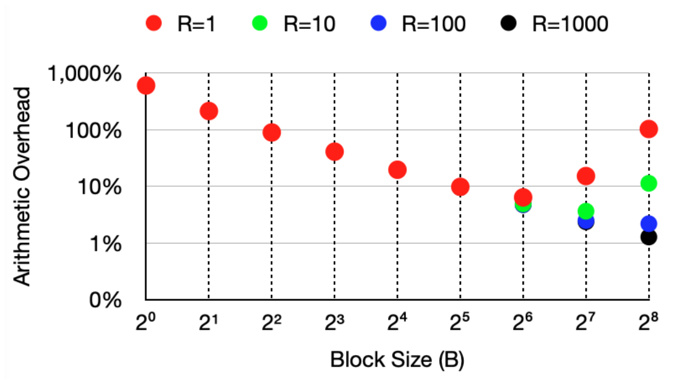


Figure 9. Arithmetic overhead of Light ABFT, for matrix size *M=256*, growing block size (*B*) and expected executions between errors (*R*).

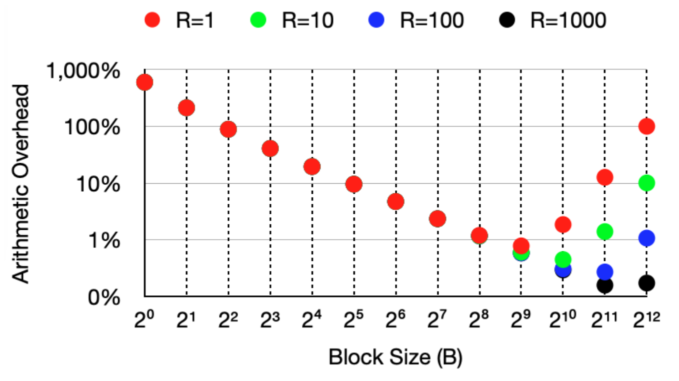


Figure 10. Arithmetic overhead of Light ABFT, for matrix size *M=4096*, growing block size (*B*) and expected executions between errors (*R*).

Finally, we thought that it would be helpful to also include a cost evaluation in a real-world application scenario. Since we have previously mentioned CNNs as being one of the most prominent MxM-based applications, and [31] being the most recent and efficient piece of related work for CNN-specific ABFT-based error detection, an in-context comparison becomes very relevant. More specifically, the authors of [31] have chosen three well-known CNNs (VGG16, ResNet18 and ResNet50), and estimated the overheads incurred by two of their proposed convolution error detection mechanisms. They have included numbers for *average increase in number of operations*, as well as *runtime overhead in a GPU execution*. As the latter is not directly comparable to our work, we will stick to the former. Authors have reported their average increase of arithmetic operations to be <7% and <1% for their *FC* and *FIC* techniques, respectively.

In order to employ *Light ABFT* in a convolution operation, it suffices to translate it to an equivalent matrix multiplication, using the well-known *im2col* method [3]. Then we must calculate the cost of adding *Light ABFT* to each of such MxM operations (using the expressions in Table 3), and the cost of the (unhardened) MxM operation itself. Finally, the ratio between the two aforementioned costs gives us the overhead of having *Light ABFT* as a percentage of useful workload. The calculated overheads were 0.35%, 0.81%, and 0.71% for VGG16, ResNet18 and ResNet50, respectively. Therefore, the average overhead of adding *Light ABFT* to the convolutional layers of the case-study CNNs comes out to 0.63%, which is, at least, as good as the state-of-the-art convolution-specific technique presented by [31]. Moreover, *Light ABFT* can also be directly employed in the inner product (i.e. fully connected) layers of neural networks, as these are, by definition, matrix multiplication operations between inputs and weights.

7 Experimental Validation

Since we have already formally demonstrated that *Light ABFT* works for arbitrarily-sized matrix multiplication operations in Section 6, the goal of this section is simply to experimentally evaluate *Light ABFT* in a concrete FPGA scenario. To be specific, we integrate the *Light ABFT* module (proposed in Section 6) into the hierarchy of our systolic array implementation (detailed in Section 3). As the unhardened DUT was already tested in Section 5, we simply repeat the experiments (using the FPGA setup described in Section 4) to measure the newly added error detection capabilities.

After injecting over 623,000 faults in our design, we have measured a detection rate of 97.4%, with a statistical error of ±0.15% (95% confidence). We believe that the remaining 2.6% of undetected faults are due to errors (1) in the interface of the module, (2) in control signals throughout the unit, or (3) in the *Light ABFT* hardware itself. Prior studies have investigated hardening alternatives for (1) and (2) [17] [45], but they are out of scope for this work. Nonetheless, as the intention of *Light ABFT* is to protect MxM computation, our experimental results are very much indicative of success. Moreover, prior work has also reported experimental FPGA fault injection numbers that do not quite reach the theoretical 100% detection rate, even for coarse grain DMR implementations. For instance, [20] achieves a maximum of 96.1% detection when duplicating an FIR filter design. Similarly, [21] proposes and experimentally evaluates a handful of ABFT-based error detection techniques for Fast Fourier Transform (FFT), reporting detection rates between 94.12% and 99.77%. This means that, in practice, there is an asymptotic limit for achievable error detection rates with redundant hardware.

Furthermore, the authors of [21] also report the occurrence of another known event in redundant circuits: *false detections*. As we mentioned in Sections 1 and 2, the added redundancy is also prone to experiencing upsets, in which case untrue detections may arise. In our experiments with a 4x4 array, we have measured a 2.39% probability for such events. For comparison purposes, [21] reports false detection rates as high as 45%. Luckily, in our case, since the percentage area overhead of *Light ABFT* diminishes with increasing array sizes (as per Section 6), so does the ratio between true and false detections in real radiation-rich environments. Thus, for sufficiently large arrays, the rate of false detections would pale in comparison to true detections. Nevertheless, for extremely strict scenarios, it would suffice to instantiate a *DMR Light ABFT* module, as a way of eliminating false detections, while still paying much less than with full DMR.

8 Conclusion

We discussed the importance of having fast and reliable matrix multiplication, and key steps to achieve it, such as selecting inherently parallel devices, and adopting proper error detection techniques at different abstraction levels. We pointed out that the combination of stuck-at faults with the data movement pattern in systolic arrays makes current fault models incomplete. To address this, we showcased an extended fault propagation model, particularly accounting for permanent upsets in SRAM-based FPGAs. After acknowledging that most matrix errors tend to affect lines instead of single elements (case in which traditional ABFT strategies fail to correct [22], or become too expensive [23]), we argued that it is preferable to opt for error detection mechanisms that incur the lowest possible cost. Then, we proposed, formally proved, and experimentally validated *Light ABFT*: a lightweight error detection technique for matrix multiplication, tailored for systolic arrays on FPGAs. We also discussed optimal design decisions for minimizing the arithmetic overhead of *Light ABFT*, given expected error rates, and the involvement of arbitrarily large matrices.

Acknowledgments

This material is based on research sponsored by Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) under agreement number FA8650-18-2-7860. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. This research has been supported by the European Union's Horizon 2020 research and innovation programme under the Marie Sklodowska-Curie grant agreement No 886202.

References

[1] J. Faeldon, K. Espana, and D. J. Sabido, "Data-centric HPC for Numerical Weather Forecasting," in *2014 43rd International Conference on Parallel Processing Workshops*. New York, NY, USA: IEEE, 2014, pp. 79–84.

[2] Top500.org, "The Linpack Benchmark," 2020. [Online]. Available: https://www.top500.org/project/linpack/

[3] K. Chellapilla, S. Puri, and P. Simard, "High Performance Convolutional Neural Networks for Document Processing," in *10th International Workshop on Frontiers in Handwriting Recognition*, Université de Rennes 1. La Baule (France): Suvisoft, Oct. 2006. [Online]. Available: https://hal.inria.fr/inria-00112631

[4] H. Shin, H. R. Roth, M. Gao, L. Lu, Z. Xu, I. Nogues, J. Yao, D. Mollura, and R. M. Summers, "Deep Convolutional Neural Networks for Computer-Aided Detection: CNN Architectures, Dataset Characteristics and Transfer Learning," *IEEE Transactions on Medical Imaging*, vol. 35, no. 5, pp. 1285–1298, May 2016.

[5] Tesla. Autopilot. [Online]. Available: https://www.tesla.com/autopilot

[6] T. Samad, J. S. Bay, and D. Godbole, "Network-Centric Systems for Military Operations in Urban Terrain: The Role of UAVs," *Proceedings of the IEEE*, vol. 95, no. 1, pp. 92–107, 2007.

[7] NASA. Mars Helicopter. [Online]. Available: https://mars.nasa.gov/technology/helicopter/

[8] V. Neagoe, A. Ciotec, and A. Bărar, "A Concurrent Neural Network Approach to Pedestrian Detection in Thermal Imagery," in *2012 9th International Conference on Communications*, June 2012, pp. 133–136.

[9] H. R. Kerner, K. L. Wagstaff, B. D. Bue, P. C. Gray, J. F. Bell, and H. Ben Amor, "Toward Generalized Change Detection on Planetary Surfaces With Convolutional Autoencoders and Transfer Learning," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 12, no. 10, pp. 3900–3918, Oct 2019.

[10] V. Volkov and J. W. Demmel, "Benchmarking GPUs to Tune Dense Linear Algebra," in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, ser. SC '08. IEEE Press, 2008, paper 31, pp. 1–11.

[11] T. Piovesan, H. C. Sartori, J. E. Baggio, and J. R. Pinheiro, "CubeSat Electrical Power Supplies Optimization — Comparison Between Conventional and Optimal Design Methodology," in *2016 12th IEEE International Conference on Industry Applications (INDUSCON)*, 2016, pp. 72–78.

[12] M. Qasaimeh, J. Zambreno, P. H. Jones, K. Denolf, J. Lo, and K. Vissers, "Analyzing the Energy-Efficiency of Vision Kernels on Embedded CPU, GPU and FPGA Platforms," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2019, pp. 336–336.

[13] M. Wirthlin, "High-Reliability FPGA-Based Systems: Space, High-Energy Physics, and Beyond," *Proceedings of the IEEE*, vol. 103, no. 3, pp. 379–389, March 2015.

[14] J. Heiner, B. Sellers, M. Wirthlin, and J. Kalb, "FPGA Partial Reconfiguration via Configuration Scrubbing," in *2009 International Conference on Field Programmable Logic and Applications*, 2009, pp. 99–104.

[15] H. Quinn, K. Morgan, P. Graham, J. Krone, M. Caffrey, and K. Lundgreen, "Domain Crossing Errors: Limitations on Single Device Triple-Modular Redundancy Circuits in Xilinx FPGAs," *IEEE Transactions on Nuclear Science*, vol. 54, no. 6, pp. 2037–2043, 2007.

[16] J. Johnson, W. Howes, M. Wirthlin, D. L. McMurtrey, M. Caffrey, P. Graham, and K. Morgan, "Using Duplication with Compare for On-line Error Detection in FPGA-based Designs," in *2008 IEEE Aerospace Conference*, 2008, pp. 2322–2333.

[17] F. L. Kastensmidt, L. Sterpone, L. Carro, and M. S. Reorda, "On the Optimal Design of Triple Modular Redundancy Logic for SRAM-based FPGAs," in *Design, Automation and Test in Europe*, 2005, pp. 1290–1295 Vol. 2.

[18] A. M. Keller and M. J. Wirthlin, "Benefits of Complementary SEU Mitigation for the LEON3 Soft Processor on SRAM-Based FPGAs," *IEEE Transactions on Nuclear Science*, vol. 64, no. 1, pp. 519–528, 2017.

[19] L. Sterpone and N. Battezzati, "A Novel Design Flow for the Performance Optimization of Fault Tolerant Circuits on SRAM-based FPGA's," in *2008 NASA/ESA Conference on Adaptive Hardware and Systems*, 2008, pp. 157–163.

[20] L. A. Aranda, P. Reviriego, and J. A. Maestro, "A Comparison of Dual Modular Redundancy and Concurrent Error Detection in Finite Impulse Response Filters Implemented in SRAM-Based FPGAs Through Fault Injection," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 65, no. 3, pp. 376–380, 2018.

[21] R. González-Toral, P. Reviriego, J. A. Maestro, and Z. Gao, "A Scheme to Design Concurrent Error Detection Techniques for the Fast Fourier Transform Implemented in SRAM-Based FPGAs," *IEEE Transactions on Computers*, vol. 67, no. 7, pp. 1039–1045, 2018.

[22] Kuang-Hua Huang and J. A. Abraham, "Algorithm-Based Fault Tolerance for Matrix Operations," *IEEE Transactions on Computers*, vol. C-33, no. 6, pp. 518–528, 1984.

[23] P. Rech, C. Aguiar, C. Frost, and L. Carro, "An Efficient and Experimentally Tuned Software-Based Hardening Strategy for Matrix Multiplication on GPUs," *IEEE Transactions on Nuclear Science*, vol. 60, no. 4, pp. 2797–2804, 2013.

[24] JEDEC. Tech (2006). rep. jesd89a, jedec standard. [Online]. Available: https://www.jedec.org/sites/default/files/docs/jesd89a.pdf

[25] J. R. Srour and J. M. McGarrity, "Radiation Effects on Microelectronics in Space," *Proceedings of the IEEE*, vol. 76, no. 11, pp. 1443–1469, 1988.

[26] R. C. Baumann, "Soft Errors in Advanced Semiconductor Devices-Part I: The Three Radiation Sources," *IEEE Transactions on Device and Materials Reliability*, vol. 1, no. 1, pp. 17–22, 2001.

[27] R. C. Baumann., "Radiation-induced Soft Errors in Advanced Semiconductor Technologies," *IEEE Transactions on Device and Materials Reliability*, vol. 5, no. 3, pp. 305–316, 2005.

[28] H. T. Kung and C. E. Leiserson, "Systolic Arrays (for VLSI)," in *Sparse Matrix Proceedings 1978*, vol. 1. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1978, pp. 1–29.

[29] N. P. Jouppi *et al.*, "In-Datacenter Performance Analysis of a Tensor Processing Unit," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. New York, NY, USA: IEEE, 2017, pp. 1–12.

[30] Xilinx. (2019) Zynq DPU. Xilinx. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/dpu/v3_1/pg338-dpu.pdf
[31] S. K. S. Hari, M. Sullivan, T. Tsai, and S. W. Keckler, "Making Convolutions Resilient via Algorithm-Based Error Detection Techniques," *IEEE Transactions on Dependable and Secure Computing*, pp. 1–1, 2021.
[32] T. Marty, T. Yuki, and S. Derrien, "Enabling Overclocking Through Algorithm-Level Error Detection," in *2018 International Conference on Field-Programmable Technology (FPT)*, 2018, pp. 174–181.
[33] F. F. d. Santos, P. F. Pimenta, C. Lunardi, L. Draghetti, L. Carro, D. Kaeli, and P. Rech, "Analyzing and Increasing the Reliability of Convolutional Neural Networks on GPUs," *IEEE Transactions on Reliability*, vol. 68, no. 2, pp. 663–677, 2019.
[34] 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 *2019 IEEE 25th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, 2019, pp. 310–315.
[35] P. Wu, Q. Guan, N. DeBardeleben, S. Blanchard, D. Tao, X. Liang, J. Chen, and Z. Chen, "Towards Practical Algorithm Based Fault Tolerance in Dense Linear Algebra," in *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 31–42. [Online]. Available: https://doi.org/10.1145/2907294.2907315
[36] J. J. Zhang, T. Gu, K. Basu, and S. Garg, "Analyzing and mitigating the impact of permanent faults on a systolic array based neural network accelerator," in *2018 IEEE 36th VLSI Test Symposium (VTS)*, 2018, pp. 1–6.
[37] A. Ruospo, A. Bosio, A. Ianne, and E. Sanchez, "Evaluating Convolutional Neural Networks Reliability depending on their Data Representation," in *2020 23rd Euromicro Conference on Digital System Design (DSD)*, 2020, pp. 672–679.
[38] Xilinx. (2019) UltraScale Architecture DSP Slice User Guide. Xilinx. [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug579-ultrascale-dsp.pdf
[39] Xilinx. (2017) Deep Learning with INT8 Optimization on Xilinx Devices. Xilinx. [Online]. Available: https://www.xilinx.com/support/documentation/white_papers/wp486-deep-learning-int8.pdf
[40] S. Di Carlo, P. Prinetto, D. Rolfo, and P. Trotta, "A Fault Injection Methodology and Infrastructure for Fast Single Event Upsets Emulation on Xilinx SRAM-based FPGAs," in *2014 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, 2014, pp. 159–164.
[41] E. Keren, S. Greenberg, N. M. Yitzhak, D. David, N. Refaeli, and A. Haran, "Characterization and Mitigation of Single-Event Transients in Xilinx 45-nm SRAM-Based FPGA," *IEEE Transactions on Nuclear Science*, vol. 66, no. 6, pp. 946–954, 2019.
[42] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-Based Learning Applied to Document Recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov 1998.
[43] V. Sridharan and D. R. Kaeli, "Eliminating Microarchitectural Dependency from Architectural Vulnerability," in *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, 2009, pp. 117–128.
[44] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2015. [Online]. Available: http://arxiv.org/abs/1409.1556
[45] S. Niranjan and J. Frenzel, "A Comparison of Fault-Tolerant State Machine Architectures for Space-Borne Electronics," *IEEE Transactions on Reliability*, vol. 45, no. 1, pp. 109–113, 1996.

Fabiano Libano received his B.S and M.S degrees from Federal University of Rio Grande do Sul, Porto Alegre, Brazil. He is currently a PhD candidate at Arizona State University. He is mainly interested in the reliability of hardware accelerators and high-performance architectures for real-time safety critical applications. His most recent work has focused on efficient radiation hardening strategies for matrix-multiplication-based computation, such as High-Performance Computing and Neural Networks.

Paolo Rech received his M.S. and Ph.D. degrees from Padova University, Padova, Italy. Since 2012 Paolo is an associate professor at UFRGS in Brazil. He is the 2019 Rosen Scholar Fellow at the Los Alamos National Laboratory. He received the 2020 impact in society award from the Rutherford Appleton Laboratory, UK. Since 2020 Paolo is a Marie Curie Fellow at Politecnico di Torino. His research interests include the evaluation and mitigation of radiation effects in HPC and autonomous vehicles.

John Brunhaver is an Assistant Professor at Arizona State University in the School of Electrical Computer Energy Engineering as of 2015. His research focuses on developing a machine understanding of computation, VLSI-design productivity, and radiation-hardened circuits and architectures. His Stanford University Ph.D. thesis, The Design and Optimization of A Stencil Engine, examines the virtual machine model for an image processing and image understanding domain-specific processor.