RunSAFER: A Novel Runtime Fault Detection Approach for Systolic Array Accelerators

Eleonora Vacca, Giorgio Ajmone, Luca Sterpone
Politecnico di Torino, Dipartimento di Automatica e Informatica
Turin, Italy
{eleonora.vacca, giorgio.ajmone, luca.sterpone}@polito.it

***Abstract*- In this study, we introduce a new runtime fault detection technique for systolic array accelerators oriented to neural network applications. The method exploits the functional path of the systolic array to compute and process checksum values during the execution of the current application instructions flow and integrates self-testing capabilities within the systolic array Instruction Set Architecture. The proposed technique does not require additional hardware self-testing modules, and the test pattern penalty is limited to 3 clock cycles independent of the size of the systolic array. Experimental analysis performed with fault injection campaigns demonstrates full fault detection capabilities of stuck-at faults with an average computing overhead 4 times lower than state-of-the-art solutions. Additionally, our approach exhibits diminished hardware overhead in contrast to conventional techniques.**

Keywords—Systolic Arrays, Fault Detection, TPU, ABFT, DFT

I. INTRODUCTION

The adoption of deep neural networks (DNNs) in safety-critical applications such as automotive and avionics has sparked scientific interest in solutions that are both high-performing, to achieve real-time response requirements, and reliable, to guarantee the correct functionalities even in harsh environments. Among the different computing devices, tensor processing units (TPUs) have raised interest due to their systolic array-based architecture suitable for accelerating the intensive matrix calculation operations of DNNs [1]. These accelerators can address the demand for DNN's high-performance and low-power consumption thanks to their highly parallel multiply and accumulate (MAC)-based datapath equipped with a large amount of on-chip memory, which enables fast and efficient DNN inference. However, the MAC grid is used to process all the neural network (NN) layers in a computing stream. Hence, if a fault occurs in the grid's processing element (PE), it may result in a cascading erroneous effect across all the layers, compromising the DNN model accuracy.

Permanent or transient faults may happen within the TPU hardware architecture due to environmental conditions (e.g., radiation effects or temperature variations), aging, or manufacturing defects.

Several works developed approaches to assess the robustness and reliability of TPU accelerators when affected by faults, mostly based on fault injection rather than detection [2]-[5]. When addressing fault detection in systolic array-based architectures, two branches of methods are commonly considered. On one side, there are algorithm-based fault tolerance (ABFT) approaches, which employ algorithmic techniques to enhance matrix multiplication by introducing data redundancy and performing additional computations, often involving checksum calculations. Even though the ABFT approach enables runtime fault detection on systolic array resources during application execution, it implies both area and performance overhead while not testing other crucial components of TPU architectures, such as accumulators [6]-[8]. These components become mandatory when operation slicing is implemented to accommodate the core size or when executing complex layers such as convolution on 3D tensors.

On the other side, commonly used design for testability (DFT) techniques include test scan chains. In particular, when applied to systolic arrays, the scan chains are implemented by exploiting the functional paths among the PEs already available in the datapath, enabling efficient testing and fault diagnosis. However, scan methodologies cannot be executed in real-time during the inference process. This is because they require the application of specific test patterns, which necessitates interrupting the ongoing application. As a result, scan-based testing techniques are typically employed during the design phase or during dedicated testing periods rather than during regular runtime execution [9][10].

In this work, we propose a new methodology that combines the properties of ABFT techniques and the test pattern propagation in the TPUs datapath. The proposed solution detects permanent and transient faults during the application execution. The main contribution of this work consists of developing a new methodology for runtime self-testing on the main resources of TPU architectures. In particular, we minimized the hardware overhead compared to previous approaches by efficiently exploiting the available resources of the datapath while introducing negligible performance degradation. We integrated this methodology as a custom instruction within the TinyTPU [11] open-core instruction set architecture (ISA).

We experimentally evaluated the effectiveness of the developed methodology, implementing the TPU accelerator enhanced with the proposed methodology on the system-on-a-programmable-chip (SoPC) Xilinx Zynq 7020. The accelerator was mapped to the field-programmable gate array (FPGA) as a co-processor of an ARM core. The ARM core controls the TPU accelerator and manages the information about the fault conditions. We evaluated the robustness and detection capabilities by conducting two fault injection campaigns, one focused on stuck-at faults and the other on the wider FPGA faults scenario. As a running application, we adopted a complete set of NN classification tasks. Experimental results demonstrated the effectiveness of the proposed approach, achieving 100% error rate detection at runtime for stuck-at faults

when utilizing an ASIC model.

Furthermore, the employed methodology resulted in a runtime detection rate exceeding 94% when evaluating hardware fault models in static random access memory (SRAM)-FPGA implementations. In both cases, the detection penalty is of 3 clock cycles, which does not depend on the systolic array size and is four times faster than existing state-of-the-art scan solutions.

The paper is organized as follows. Section II presents the TPU architecture characteristics, while Section III deeply overviews state-of-the-art approaches. Section IV outlines the systolic array fault models and Section V details the proposed methodology. Section VI proposes the experimental results. Finally, Section VII presents the concluding remarks.

II. Background on Tensor Processing Unit

The TPU is an application-specific architecture designed to support the computational workload of NN inference which consists of the computation of several wide matrix multiplications. Equipped with its own ISA, the TPU is a (complex instruction set computer) CISC-type architectural model, characterized by a large amount of on-chip memory, for storing NN model weights and activations. The main computation core is the matrix multiply unit (MMU), which consists of a two-dimensional systolic array of MAC units, interconnected in a mesh-like structure. Each MAC multiplies an activation input $x_{i,j}$ by a weight $w_{i,j}$ and accumulates the result in a single clock cycle. The peculiar interconnection layout among MACs enables efficient data propagation between processing elements and performs on-site accumulation of partial products, avoiding memory accesses to store and load intermediate results. Alongside the systolic array, external accumulators store and process intermediate results during the computation. These accumulators are crucial when employing operation tiling to accommodate the core size when executing large layers.

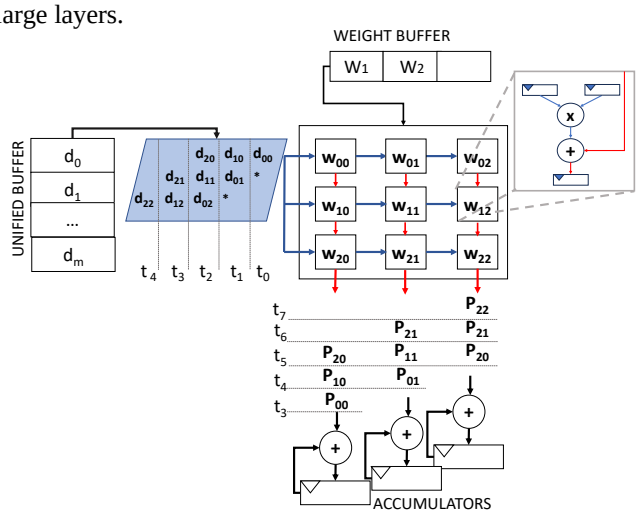

Fig. 1. The systolic-array accelerator architecture with weight stationary data mapping.

Since NN's weight matrices are typically larger than the input tensor, the weight stationary (WS) data mapping policy, shown in Fig.1, minimizes data transfer and memory accesses. Specifically, before the input data starts flowing row-wise, all the weights are loaded in the MAC grid. Each weight is loaded in the register of a specific MAC unit for the entire computation process. After the weights loading, at each clock cycle, a new input vector is fetched, with the leftmost column receiving the new input while the previous vector is processed by the subsequent column. At the same time, each MAC produces one output element for the MAC on its bottom, contributing to the per-column output accumulation through the dedicated interconnection path. Hence, when the WS policy is adopted in systolic arrays, it involves a dual data flow within the core. Firstly, there is a left-to-right propagation of the activation input between consecutive MAC units. Secondly, there is a top-to-bottom propagation of the partial sum with accumulation.

III. Related Works

The introduction of Google's TPUs has reignited interest in systolic arrays, prompting analysis of their performance and reliability [2][4] while fault detection and correction techniques in systolic arrays have been explored since the 90s [12]-[14]. The growing use of systolic array-based accelerators in critical applications like autonomous driving and avionics has emphasized the need for effective fault diagnosis and fault-tolerant design with marginal power, performance, and area overheads. Indeed, the high density of MAC units in systolic arrays and routing congestion, combined with technology scaling, make them more sensitive to permanent and transient faults [3][5]. As the systolic array workload moved towards NN inference, new fault-tolerant architecture proposals arose based on weight pruning, model compression, and bypass of faulty PEs [15]-[17]. However, these approaches assume prior knowledge of the fault locations, not accounting for fault detection and diagnosis. In contrast, when facing real-time applications requiring high reliability, ABFT can be an effective solution since it aims at both error detection and correction on the fly. The main idea is to enhance the data encoding process through checksum, which enables detecting computational errors, while correction can be done in limited cases provided that hardware support has been included in the design [18]-[21]. However, ABFTs require additional hardware to compute and process the checksum values, impacting the chip area, power, and performance. As a result, scientific research has made headway in exploring lighter alternatives.

Considering the A x B operation, traditional ABFT requires the computation of column checksums for matrix A and row checksums for matrix B, followed by augmenting the original matrices with these checksums before performing matrix multiplication. In [7], authors propose a streamlined version of ABFT to address voltage scaling-induced transient errors, which solely compute the row checksum for matrix B. To achieve this at runtime, they allocated N additional PEs into a systolic array of size N x N, along with an additional column of digital integrators and comparators for error detection. Moreover, this approach introduces a penalty of 2 clock cycles on the matrix multiplication process, while missing exhaustive fault coverage. Similarly, authors in [8] propose the *Light ABFT* technique for systolic arrays. They identify the presence of faults by comparing the sum of all the elements of the outputs matrix C = A x B with the product (L) of the column checksum of matrix A and the row checksum of matrix B.

In the proposed method [8], checksums and L values are computed in parallel with matrix multiplication execution, requiring an overall architectural cost of 1 multiplier, 2N+2

adders, and N comparators, if considering an N x N systolic array. However, both techniques [7][8] do not perform any fault diagnosis.

Another branch of approaches involves the DFT solutions, which include using scan chains and built-in self-test (BIST) methods. Traditional scan chains are commonly employed in digital circuits for the purpose of enhancing testability, allowing a more efficient application of test patterns and collection of responses. In the context of systolic arrays, the concept of scan chains has been adapted to exploit the existing functional paths and registers within the array structure. Recalling the dataflow explained in Section II, where activation inputs flow row-wise from left-to-right of the array, and the partial sum column-wise from top to bottom, both authors in [9] and [10] exploit these interconnections to build scan chains. Likewise, the weight registers within each column are transformed into a scan chain using the existing functional paths.

Specifically, [9] proposes a methodology that utilizes inter-MAC communication to propagate test patterns achieving 100% stuck-at fault coverage with zero hardware overhead on the systolic array. However, to ensure accurate testing when utilizing the partial sum functional path, a specific condition must be met. The weight data of each MAC unit needs to be set to zero to prevent any computational interference caused by the simultaneous application of test patterns to both the activation inputs and partial sum. By setting the weights to zero, it is ensured that the output produced by each MAC unit is solely influenced by the test pattern introduced through the partial sum scan chain, nullifying the product of weight per activation contribution. In their method, the authors assume the correct behavior of the external accumulators and use their dedicated register banks to store the partial sum propagated output, which is further compared using an MBIST methodology but not discussed in the paper. This technique has a severe disadvantage related to the testing routine timing penalties, which are proportional to 3 times the number of the array size. Besides, although this solution does not introduce hardware overhead on the systolic array, fault detection cannot be performed in parallel with the main application workload since it requires setting specific patterns in all PEs. Moreover, even if not explicitly mentioned, handling test pattern applications and monitoring requires BIST logic which has not been addressed.

Finally, in [10], the authors propose STRAIT which is equivalent to [9] with only a difference in the propagation of partial sum values. They introduce a mux for each MAC unit, to avoid forcing the weights registers to zero. In this way, they build an additional functional path, where partial sum registers of each $MAC_{i,j}$ unit can be either fed by the outcome of the multiply and accumulate operation (i.e., canonical) or directly from the partial sum register of $MAC_{i-1,j}$, hence building proper scan chain of partial sum registers per each column in the array. After constructing the scan chains, they introduce the concept of Hybrid BIST, which combines Logic and Memory, since external accumulators are also utilized in this case for result comparison and storage.

While the number of test patterns to be applied is limited, typically around twelve for both [9] and [10], and the associated hardware overhead is also minimal, these approaches are not conducive to concurrent inference execution. This limitation arises due to the necessity of activating dedicated paths, performing data shifting, and substituting weights, which are incompatible with the execution of NN models. Compared to previous approaches, our solution harnesses the significant advantages offered by ABFT and scan chains. Similarly to [9][10] we employ test input patterns encompassing both activations and partial sums. By effectively utilizing the interconnection layout and data flow characteristics of the WS policy these test patterns are designed to generate and propagate complementary values among the resources in consecutive clock cycles, allowing for stuck-at-fault detection. Furthermore, the test patterns generate column checksum values specific to the currently processed weight matrix as outputs of the systolic array. Similarly to the approach presented in [7][8], these checksums are compared against expected values that in our proposed solution are computed in parallel using the external accumulators. Subsequently, following the canonical data flow, the accumulators are used to compare the generated checksums. The resulting comparisons enable an error unit to perform fault diagnosis, identifying fault location and distinguishing whether it resides in MACs or in the accumulators, if it concerns weights, partial sum propagation, or activations.

The proposed method enables the evaluation of faults without compromising the integrity of the original NN weight data, ensuring the undisturbed progress of the main computation and facilitating robust inference. Compared to [7][8] we have no hardware overhead related to checksums, and we perform diagnosis by identifying the fault locations. Compared to [9][10] fault detection occurs with less granularity, but it is a good tradeoff with the ability to run the testing mode in runtime during inference.

Overall, the proposed approach has demonstrated seamless integration within an actual system executing an NN model, with minimal overhead on the hardware and performance.

IV. The Systolic Array Fault Model

In TPU architectures, various types of faults can arise, potentially causing disruptions in the architecture's operation. Stuck-at faults can impact individual processing elements like MACs, accumulators, registers, and interconnections. These faults result in signals permanently stuck at a high or low logic level. Additionally, bitflips can occur in internal registers used by MAC units to store weights and input tensors. These bitflips can lead to erroneous computations and affect the accuracy of the inference. Shorts and opens in the interconnection layout can also introduce faults that hinder proper data flow within the architecture.

By examining the faulty behavior at the output of the systolic array and considering the operations performed by each PE along with the data flow, we can effectively determine the fault locations. This allows us to investigate a self-test methodology capable of covering various scenarios with negligible overhead. When considering the WS policy, which involves partial product propagation per column and activation propagation per row from left to right, we can deduce the source of fault by certain error pattern manifestations. Consider each $MAC_{i,j}$ produces one item per clock cycle as follows,

$$p_{o\,i,j} = p_{i_{i,j}} + (d_i * w_{i,j}) = p_{o_{i-1,j}} + (d_i * w_{i,j}) \quad (1)$$

where p_o is the output partial product generated, p_i is the input partial product, coming from $MAC_{i-1,j}$, d_i is the activation, and $w_{i,j}$ is the weight data.

If a MAC unit is faulty, due to the column-wise accumulation, it will produce a faulty p_o that will propagate as a faulty p_i contribution for all the MACs in the same column, reaching the output. Similarly, faults in weights or partial sums will result in a column producing output values different from the expected ones.

In contrast, when a fault affects an activation input, due to the left-to-right flow of activation inputs explained in Section II, where at each clock cycle each $MAC_{i,j}$ receives d_i input from the $MAC_{i,j-1}$, we observe the propagation of compromised data along the row. Specifically, the corrupted data spreads from the faulty PE toward the right side of the array. However, the fault does not remain confined to the row alone. Due to the WS policy, every MAC unit receiving the corrupted d_i term will contribute to generating a corrupted output p_o, which, again, propagates throughout the entire column. Consequently, a fault in the activation results in multiple adjacent corrupted values at the output of the systolic array. Note that the index of the first corrupted output value aligns with the column's index affected by the activation input fault.

Therefore, the propagation mechanism can be effectively exploited for implementing self-testing methodologies without incurring additional hardware costs and with minimal computational delay. When employing the WS policy, combining the propagation map with a column-wise detection mechanism allows for identifying stuck-at and bitflips that affect any resource of the array. When identifying a computational error in a single column, the issue probably pertains to a weight or partial product associated with that specific column. Conversely, when multiple faults are identified in adjacent columns, it is more probable that the fault lies within the inputs of the initial column within the block of faulty columns. Hence, if two or more non-adjacent columns generate inaccurate outputs, multiple faults are raised within the system, as shown in Fig. 2.

When considering ASIC and FPGA implementations, the cause of faults may vary. However, the manifestation of these errors and their propagation mechanisms are equivalent, as they primarily depend on the circuit topology rather than the specific implementation technology. Therefore, the classifications of errors remain applicable to systolic array-based accelerator implementations in both ASIC and FPGA technologies.

V. The Proposed Approach

The proposed approach detects any fault-induced anomaly in the functionality of the systolic array computational units, allowing coarse-grained fault localization. To achieve this goal, we used the available resources and the unique data flow of the accelerator datapath to develop a self-test methodology that integrates checksum computation and the scan chains methodology. Unlike previous works [7][8] that rely on dedicated external modules for checksum computation, our approach exploits the resources of the systolic array itself to generate the checksums. Additionally, AI accelerators based on systolic arrays often employ external accumulators to support tiling operations on matrices larger than the available resources. In our method, these accumulators are utilized to generate and process comparative checksums, in parallel, on the same data processed by the systolic array. Therefore, by comparing the two units' results, we simultaneously detect faults both in the systolic array core and in the accumulators, with the ability to distinguish which of these units is affected by a fault and where the fault is located (i.e., which accumulator and which column of PEs). Furthermore, the checksums' generation process involves stimulating the datapath with test patterns that shift both in MAC's activations and partial sums inputs bitwise complemented data between consecutive clock cycles. This allows the stuck-at fault detection without area overhead since exploiting the embedded inter-MAC connections as scan chains as demonstrated in [9][10]. By integrating both methodologies, we can execute a self-test routine while the main application is running, achieving runtime error detection while ensuring minimal impact on performance and area.

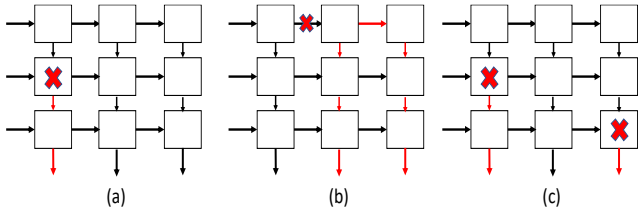

Fig. 2. Fault propagation model when WS data mapping is adopted. (a) single fault affecting either weight or per column output propagation (b) single fault affecting input (c) concurrent multiple faults.

A. The Fault Detection Methodology

The proposed approach consists of three phases:

1. Compute two checksums on the weight matrix loaded to the systolic array by the NN application using the resources of the systolic array.
2. Compare the checksums with golden values, computed at runtime by dedicated resources available in the Datapath, using the accumulators present in the Datapath.
3. Evaluate checksums and comparison results to detect and locate any potential fault affecting both the systolic array and the golden value generation unit.

The generation of the golden checksums depends on the specific datapath. When accumulators are available, the golden checksums on weight data can be computed by exploiting them with a few architectural modifications. This will be fully discussed in the following section concerning the hardware implementation of the proposed method. Otherwise, additional dedicated units should be inserted to fulfill the purpose. Please consider that comparing the checksum generated by the systolic array with those generated by specialized modules, like the accumulators, helps identify which part of the Datapath is malfunctioning.

Starting with the first phase, the systolic array's activation register chain is sequentially fed by two 1D test patterns. These test patterns are designed to serve two purposes. Firstly, to generate column checksums for the current weight matrix. Secondly, to assess the inter-MACs propagation of (i) the activation data, from the left to the right of the systolic array, (ii) partial sums, from top to the bottom of the systolic array, to address the row fault model and the column fault model

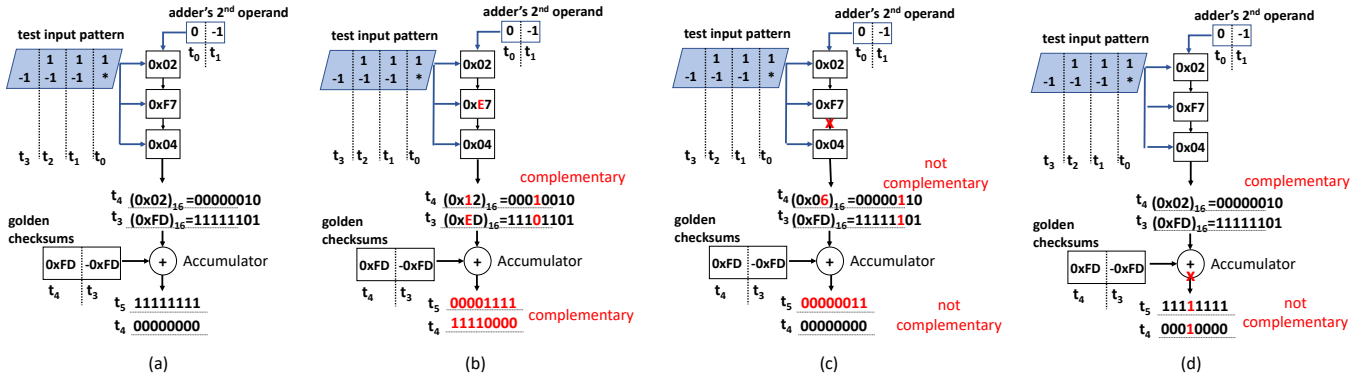

Fig. 3. Data flow example of the proposed algorithm solution on a single column of MAC resources and relative Accumulator. (a) fault-free scenario (b) fault in a MAC weight (c) stuck-at in the output propagation path (d) fault in the Accumulator. By comparing results coming from Accumulators and those produced in the systolic array, the fault location and the variation with respect to correct output are identified.

discussed in Section IV and shown in Fig. 2a and Fig. 2b respectively.

The proper functioning of the inter-MACs data transfer in both directions is achieved by enforcing the propagation of bitwise complemented values in two consecutive clock cycles, enabling stuck-at fault detection.

Considering a systolic array of size M rows and N columns, a weight matrix W of size M x N, and a unit row vector U of size 1 x M, with each element initialized to 1. Due to WS policy, each $w_{i,j}$ element in W is always mapped to the $MAC_{i,j}$ in the systolic array. By applying as the first test pattern the vector U, we stimulate the systolic array to compute the matrix multiplication U x W, resulting in a row vector R of size 1 x N. The value of each element in R corresponds to the sum of the weights in the corresponding column of matrix W. Mathematically, this can be represented as:

$$R = U \times W \quad \text{with} \quad R[j] = \sum_{i=1}^{M} w_{i,j} \quad \text{with } j \in [0, N[\tag{2}$$

Hence, R is the column checksum of matrix W. Then, to evaluate the inter-MACs communication against stuck-at, we need to invert each register's bit and driving signal in the architecture, which means computing and propagating the bitwise complement of the previous computation, i.e., $\overline{R}$. Considering the 2s'complement of number x, shown in (3), to compute its complemented value we need to invert the sign and subtract 1.

$$\overline{x}_{2'\text{complement}} = -x - 1 \tag{3}$$

Therefore, if we apply as an activation test pattern a vector $U^{\dagger}$ of size 1 x M, with each element initialized to -1, the systolic array is stimulated to compute $U^{\dagger}$ x W, resulting in $-R$. If, at the same time we shift -1 in the partial sum chain of each column, then according to (4) we obtain $\overline{R}$.

$$R^{\dagger} = U^{\dagger} x\, W - 1 = -R - 1 = \overline{R} \tag{4}$$

Hence, by applying two test patterns $U^{\dagger}$ and U to the activation chains and concurrently two test patterns (0 and -1) to the partial sum propagation chains, just letting the systolic array simply process the test patterns as normal input vectors, we are able to evaluate the inter-MACs communication through the propagation of bitwise complemented data.

In the following, R_{SA} and $\overline{R_{SA}}$ indicate the checksums produced exploiting the systolic array.

After computing the two complemented checksums, the second phase commences, during which the accumulators are introduced into the self-test routine. Typically, the number of accumulators is N, which matches the size of the first dimension of the systolic array. Accumulators are employed within the test routine in a manner consistent with their usage in the normal execution routine. They are intended to gather and accumulate the outcomes of consecutive computations, mirroring their typical behavior. What differs is the meaning of the operands, which are the checksums derived from the systolic array and the golden checksums. To be more precise, once checksum vector R_{SA} has been produced, each accumulator A_j produces one output value a_j as follows:

$$a_j = R_{SA}[j] - R_{gold}[j] \;\; with\; j \in [0, N[\tag{5}$$

Similarly, after the production of $\overline{R_{SA}}$, which is equal to -R_{gold} -1 according to (4) in a fault-free system, each accumulator A_j produces one output value a_j^* as follows:

$$a_j^* = \overline{R}_{SA}[j] + R_{gold}[j] \quad with\; j \in [0, N[\tag{6}$$

In the absence of faults, regardless of the weights' values, each a_j will assume value 0 and each a_j^* -1.

Once obtained the complementary checksums and the pair (a_j, a_j^*), where j corresponds to both the systolic array's column index and the accumulator index, we can evaluate and classify faults, being the last step of the methodology. Specifically, when $a_j \neq 0$ and $a_j^* \neq -1$.

- if $a_j = not(a_j^*)$

  The fault concerns a weight resource in column j. If the weight $w_{i,j}$ is affected by an upset, multiplying it by 1 and then by -1 will result in an erroneous yet complementary value being added to the column sum. Therefore, when the accumulators perform the sum with the golden values, the resulting value will differ from 0 and -1, but still complementary. This discrepancy indicates that there are no stuck-at faults in the resources, but rather a bit-flip in the weight. A numerical example is shown in Fig. 3b.

- if $a_j \neq not(a_j^*)\; and\; not(R_{SA}[j]) \neq \overline{R_{SA}}[j]$

  The fault concerns stuck-at in the systolic array. Indeed, if the column sums produced by the systolic array in the two

iterations are not complementary, it signifies the presence of a stuck-at fault. This fault hinders the ability to flip a value, resulting in the lack of complementarity between the sums. A numerical example is shown in Fig. 3c.

- if $a_j \neq not(a_j^*)$ and $not(R_{SA}[j]) = \overline{R_{SA}}[j]$

  The fault is located in the accumulator A_j since the column of MACs produced complementary sums, but the accumulator results are different from the expected (i.e., all 0s and all 1s). A numerical example is shown in Fig. 3d.

It is worth noticing that, as illustrated in (5) and (6), the comparison between the checksums involves addition and subtraction operations with R_{gold}. By leveraging the properties of two's complement, the accumulators perform the subtraction as an addition on complemented data and with the CarryIn set to 1. Therefore, following the same principle adopted for the systolic array, by executing computations on complemented data on all its input (*operandA*, *operandB*, *CarryIn*) in two consecutive clock cycles, we are able to detect potential stuck-at faults even in the accumulators' resources. The accumulator data flow is illustrated in Fig. 4.

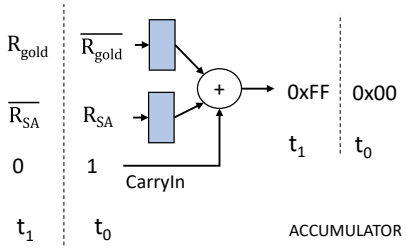

Fig.4 Data flow on the Accumulators when the self-test routine is running. In two consecutive clock cycles the accumulators receive and produce complemented data which enables stuck-at detection.

Moreover, since both vectors U and U† have the LSB value of the first item set to 1, to prevent any stuck-at-1 issues in the resources associated with the activation input tensor, exhaustive fault detection methodology requires an additional test pattern to be propagating along the systolic arrays.

The proposed method generates checksums and compares them with the systolic array's column-level granularity. Consequently, unlike approaches solely based on scan chains, we are unable to identify the specific corrupted MAC unit, but rather the affected column. On the other hand, by processing the evaluation results, we can determine whether the error is a row fault or a column fault, as explained in Section IV, based on the distribution of faulty columns. Moreover, although lacking in granularity, our method stimulates and evaluates two computational units of the datapath simultaneously, which is typically deferred or performed at different times, thereby reducing system availability in previous approaches.

Furthermore, as discussed in the following section, this method can be integrated into the runtime execution of an application with a reduced penalty. It is also important to consider that when exploiting a fully pipelined datapath as illustrated in the following section, that processes one vector per clock cycle, the overall clock penalty introduced by the detection system is stable and independent of the size of the systolic array, being 3 clock cycles (one for each test pattern) despite the number of MACs unit. Therefore, the lower granularity represents a tradeoff with the scalability and intrusiveness of the proposed approach, which, as shown in the experimental results, enables the detection of faults during inference execution. Consequently, in an ideal system, this method can be employed as a preliminary runtime detection of higher-level anomalies, subsequently triggering more comprehensive and fine-grained detection methods.

B. The Implementation

To validate the proposed approach, we implemented the methodology as an extended version of a matrix multiplication instruction in the ISA of an open-source TPU architecture [11]. Hence, the architecture now supports the execution of plain matrix multiplication and testing-mode matrix multiplication.

In TPU architecture, the matrix multiplication (*matmul*) operation typically follows a pipeline flow to perform the computation efficiently. Considering a systolic array core of size N x N, the execution starts with a *load weight* instruction instructing the core to fetch an N x N weight matrix from memory and load it into the systolic grid. Since the pipelined structure, while the weight matrix is being positioned, the *matmul* instruction starts in parallel after 1 clock cycle (CC). This operation includes the fetching of N activation vectors from memory. This timing reflects the need for the first column of MACs to be filled with weights before receiving the inputs, as required by the weight stationary policy. Considering that the loading of weights takes N CC, the output vectors, one per CC, start after N+4 CC, while the overall duration of the *matmul* operation is 2N-1 CC. As soon as result vectors are produced, they are transferred to the accumulators, which either perform accumulation with the data stored in the destination register or simply overwrite it. Just after the production of the first vector result, the second load weights instruction can begin filling the systolic array with new data.

To support the self-test proposed methodology we modified the data flow inside the pipeline as follows. When the testing mode is active, and the *load weight* is issued, the weights flow both in the systolic arrays and in the accumulators to compute the golden checksum R_{gold} discussed in the previous section. The accumulators may be busy accumulating results from the last executed matrix multiplication if required by the program code. Since efficient NNs rely on data quantization and models are trained on 8 or even fewer bits, we assumed that the parallelism of the accumulators could be such to enable the development of an asymmetric SIMD (Single Instruction, Multiple Data) on weights checksum and output product accumulation.

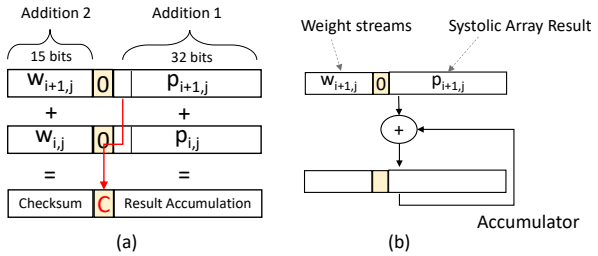

Fig. 5. Asymmetric SIMD on Accumulators to compute the checksums while processing application workload results. (a) detailed overview of data handling (b) schematic data path.

This assumption is not an uncommon scenario. For instance, state-of-the-art implementations of systolic array-based accelerators in FPGA map the MAC units to digital signal

processor (DSP) units, which can support operands on more than 48 bits (Xilinx 7 Series 48 bits, Xilinx Versal 58bits, Intel Agilex 54 bits). Hence, drawing on the method proposed in [22] we can apply asymmetric SIMD by concatenating weights data and *matmul* results when feeding accumulators operands. To account for the carry condition, we insert one zero as a guard bit between the two data being processed as shown in Fig. 5a. This allows us to perform canonical accumulation while producing the golden checksum. In the scenario where asymmetric SIMD is not feasible due to the maximum value of the column weight sum exceeding the representable value given the available bits, one option could be to limit the testing mode to matrices in the pipeline that are temporally separated from matrices involving accumulation, preserving the whole bits either to results or to checksums computation. Ultimately, incorporate dedicated computing units to support checksum calculation.

Once the accumulators have finished the golden checksum computation, values are stored in preserved registers R0, R1 within the accumulators' registers bank. The control unit then waits for the *matmul* instruction to complete its workload processing, but before freeing the unit, applies to the current weight matrix the test patterns discussed in Section IV-A to generate systolic array's checksums. As soon as they're produced, they are passed to the accumulators as regular results from the *matmul* operation. The accumulation process for the checksums differs slightly from the other output vectors. For the rest of the output vectors concerning that *matmul* instruction (operation 2 in Fig. 6b), the accumulation occurs with target registers whose address is determined by a specific field in the instruction format. Instead, for the last received results related to the checksums, the accumulation always takes place using the values stored in the R0 and R1 registers, performing phase 2 discussed in the previous section. The result of the accumulation between the accumulator checksums and the systolic array checksums is passed to a comparison unit based on XNOR and OR logic, which checks the result conditions mentioned in Section IV-A. This unit returns an N-bit error array status, with each bit associated with a MAC column of the systolic array. In the case of a fault, the corresponding bit is set to 1.

In our proposed system design, where the TPU operates as a coprocessor, we have established a mapping between the error array status and the interrupt port of the main processor to make the runtime detection more effective. Since we targeted FPGA design, this information may be used to trigger the autonomous FPGA reconfiguration with golden bitstream. In fact, any detected error in this scenario is caused by an alteration of the content of the device CRAM. So, to correct and solve the fault-induced problem, CRAM reconfiguration is sufficient. Concerning ASIC implementation, once a stuck-at is detected, and faulty resources individuated, methods such as model compression [16] and fault-aware pruning [15] may be adopted to bypass the faulty resource.

In Fig. 6b the modified pipeline is reported. We added a testing mode for the *matmul* instruction, labeled as *tmatmul*, in order to provide the ability to control when to perform calculations in testing mode. This instruction introduces an additional latency of 3 CC compared to the plain *matmul* operation as it processes the additional test patterns.

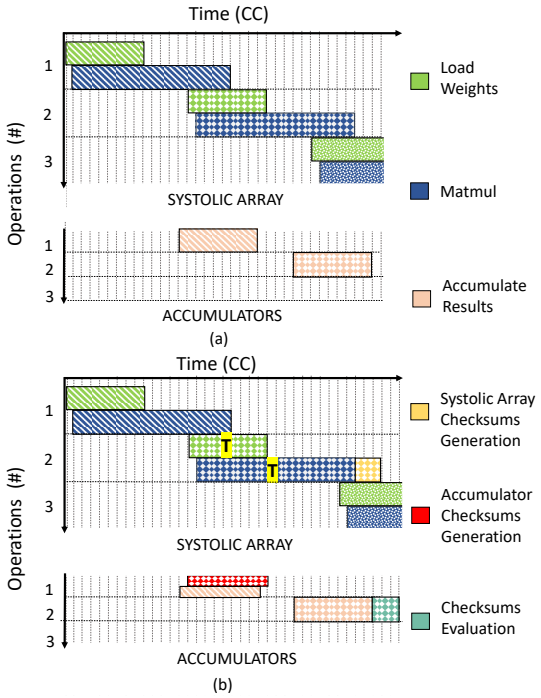

Fig. 6. The original pipeline in (a) and the pipeline when supporting self-test instruction (b). Different colors but the same pattern relates operations required for single matrix multiplication with accumulation. Execution of operations in testing mode (T) introduces a latency of 3 CC while enabling runtime fault detection.

When considering complex NN, their execution on systolic arrays necessitates the transformation of layers into general matrix multiplication operations [23]. Typically, layer size does not match the systolic array size. Therefore, the layer is decomposed into smaller *matmul* operations. The more complex the network, the deeper the sequence of *matmul* operations composing each layer. Our method operates at the granularity of individual *matmul* execution, and integrating it as an extension of the multiplication instruction allows us to choose when to apply test mode, reducing its impact on NN real-time response. In general, considering a layer that, when decomposed, requires M *matmul* operations performed in sequence, if we want to execute each *matmul* in testing mode, then the penalty on the overall layer will be 3*M CC.

Therefore, it is up to the programmer to decide whether to execute an entire application (i.e., complete inference) in testing mode, incurring a penalty of 3 CC for each *matmul* but ensuring that every calculation performed is error-free. Alternatively, they can choose to perform only specific computations in testing mode, such as only a few matrices per layer or only the final layer, according to real-time response requirement of the specific application.

From a perspective focused on design choices, employing parallel computation for reference values offers significant advantages in terms of scalability and implementation flexibility. In FPGA implementations, systolic arrays require minimal programmable logic because the MAC units are mapped to DSPs. In the specific instance of this open-source core, the design utilizes less than 10% of the available LUTs and Carry logic within the device. It's important to note that the device used is not a next-generation one, which typically offers a wealth of additional resources, allowing for more extensive

implementation possibilities. As a result, when dealing with larger systolic arrays, designers have the option to choose between utilizing Datapath accumulators or dedicated resources. In our case, we deliberately made use of the resources provided by the datapath and employed accumulators in parallel to reduce hardware overhead. However, the use of parallel adders is also fully compatible, and it does not impact the overall methodology. During the third phase of the approach, as described in Section V-A, the reference values, whether generated by accumulators themselves or by additional units, pass through the accumulators. These values are subsequently compared with those from the systolic array. The rationale behind this data comparison, aimed at identifying fault locations, remains valid in both scenarios.

The adoption of a comparative analysis between values produced by different units in runtime allows the detection of errors induced not only by stuck-at faults but also by more complex scenarios. Indeed, if the checksums produced are not complemented values and the values produced by accumulators differ from those produced by the systolic arrays, then the mismatch between values pinpoints the presence of a fault in the Datapath and will trigger the error detection unit. In this case, the method is not capable of identifying fault location, since it is modeled to target stuck-at, but still, the presence of a fault will be detected in most of the cases. In fact in the following section fault injection campaign targets not only stuck-at but also FPGA fault models. The letter, depending on the configuration memory cell affected may result in different fault scenarios affecting the systolic core such as stuck-at, logic functions alterations, bridge faults, open faults, etc.

As previously mentioned, our proposed method is not able to provide the exact faulty MAC unit, while indicating in which column of the array it is located. Hence, detection performance is lower if compared to the scan-based method [9][10] while providing the benefit of runtime execution, not inducing program interruption, which is not the case for the scan test routine. Being our proposed method in the middle between scan and ABFT, it presents a good tradeoff between the two techniques taking the major benefits of both at lower cost both in hardware resources and intrusiveness on the workload.

From a power consumption perspective, the methods do not increase static power consumption, since it does not allocate additional computational resource. However, both adopting SIMD mode in the accumulators and flowing of complemented data across the units, incurs an increment of dynamic power consumption. However, since the application of the method can be confined to a few *matmul* executions (programmer's choice), the impact on the overall workload may be minimal when compared to complete NN execution.

VI. Experimental Results

We evaluate the fault detection capabilities of the proposed method by implementing a TPU architecture on the Xilinx Zynq 7020 programmable SoC and performing fault injection campaigns. In detail, the TPU module was implemented within the SoC programmable logic and interfaced with the embedded ARM cores through an AXI bus interface. The TPU module has been equipped with an additional debug port capable of transmitting the functional conditions of its resources (faulty or not faulty) once the self-test operations are executed. Since implemented on FPGA, as done by [8] both MACs and accumulators have been mapped to DSP, allowing us to exploit the proposed asymmetric SIMD. To exploit all the available on-chip DSPs the systolic core size is 14 x 14 and the datapath is provided with 14 accumulators.

The fault injection has been deployed targeting the insertion of faults within the SoC bitstream resources controlling the mapped TPU design. The insertion of a faulty bitstream mimics the insertion of permanent faults within the TPU resources depending on the selected bitstream coordinates according to the techniques developed in [24]. Please note that the effective injection within the bitstream is done before the upload of the programmable logic configuration memory (CRAM).

The experimental validation campaigns have been done evaluating 20,000 faults. In order to evaluate the proposed methodology with respect to the execution of typical TPU applications, we developed two benchmark applications: a Convolutional Neural Network and a Multilayer Perceptron Neural Network. The two applications were settled to execute a classification task on the MNIST digit dataset. Each model is executed by exploiting the *tmatmul* instruction to assess the validity of the proposed method.

The obtained results demonstrate detection capabilities with an average rate of 94.6% for the two NN models. Additionally, the detection latency was found to be low, as the system is capable of identifying faults and notifying the host PC from the first NN layer execution. However, it should be noted that the proposed method cannot achieve a 100% detection rate on FPGA design due to the higher complexity of the fault model. As mentioned before, a single bitflip in CRAM may induce stuck-at as well as bridge, conflict, and open faults, which are not directly addressed by the proposed approach. However, independently from faults type, if the effects result in a mismatch between checksums, it will be detected. Indeed, during the injection campaign, the fault location was random and the fault effect unpredictable, but still we were able to detect more than 90% of faults. Therefore, compared to the method described in [8], which relies on multiple checksum computation and processing, the detection rate is lower as they report obtaining around 97% detection rate, while in [7] they do not provide actual statistics. Nevertheless, as shown in Table I, the overhead induced by our method is negligible compared to ABFT approaches while introducing the ability of fault diagnosis. The tradeoff between performance, area, functionalities, and detection capabilities is evident.

In order to compare our method with scan-based solutions, we performed a stuck-at based fault simulation by injecting stuck-at faults in all MAC units, inputs, and outputs resources. Our detection system demonstrated a fault coverage of 100% similar to [9] [10] but with a drastically lower computing overhead. We observed, that our approach requires the application of 4 times less test patterns versus previous approaches, without PE architectural modifications.

We evaluated the area overhead by comparing the equivalent ASIC gate count for LUT-based implementation, as reported in [25], and in order to be comparable with [10], we also configured the systolic array size to 256 x 256, mapping MACs to LUTs. As can be seen in Table II, our method has an area overhead of 0.31%. Overall, the experimental results proved that with minor modifications to the architecture, it is possible to

monitor the conditions of the computational resources in real-time without interrupting the inference execution. The proposed method also exhibits flexibility, as it is capable of detecting the presence of stuck-at faults as well as other types of faults with a fair detection rate while maintaining reduced overhead in terms of performance and area.

When considering mission-critical applications such as avionics or automotive the fault scenario and requirement may be different and strictly dependent on target technology. If we consider SRAM-based implementation, the bitflip fault model we adopted to assess the effectiveness of the methods holds. In fact, high-energy particles typically induce single event upset in CRAM which leads to structural changes in the Datapath, as we tested in our fault injection campaign where we obtained 94.6% detection rate. However, we evaluated only the case of one fault per time, while multiple-bit upset can occur, affecting multiple resources of the device at the same time. This scenario will be deeper explored in future work. On the other hand, considering ASIC implementation, another kind of error apart from stuck-at faults may be induced by transient propagation as analyzed in [3]. In this case, if a weight resource samples the transient, our proposed method will detect the data corruption, for the reasoning explained in Section V-A (case if $a_j = not(\ a_j^*)$), while the effects on the input will be addressed in the future.

TABLE I
ARCHITECTURAL OVERHEAD CONSIDERING N X N SYSTOLIC ARRAY IN ABFT METHODS

	[7]	[8]	**RunSAFER**
PEs	N	0	**0**
Checksums	N adders	2N+1 adders, 1 MAC	**0**
Detection Comparators	N	N+2	**N**

TABLE II
COMPARISON WITH SCAN METHODS FOR SYSTOLIC ARRAY 256x56

	[9]	[10]	**RunSAFER**
Required Test Patterns	11	12	**3**
Area Overhead	NA	5.25%	**0.31%**
Fault Coverage SA	100%	100%	**100%**

TABLE III
FUNCTIONALITY COMPARISON

	[7]	[8]	[9]	[10]	**RunSAFER**
Detection	✓	✓	✓	✓	✓
Diagnosis	X	X	✓	✓	✓
Runtime	✓	✓	X	X	✓

VII. CONCLUSIONS

In this work, RunSAFER is proposed as a runtime fault detection methodology suitable for systolic array-based accelerators. The proposed method attempts to combine the characteristics of scan methods and ABFT to achieve real-time fault detection with reduced hardware overhead and limited timing penalities, regardless of systolic array size. Its feasibility has been proved by embedding the methodology as custom extension of matmul instruction in a TPU open-source core, while its validity has been proved through fault injection campaign. The experimental results proved 100% fault coverage for stuck-at faults and a four times speed up versus state-of-the-art solutions when running complex neural network applications.

REFERENCES

[1] N. P. Jouppi et al., "In-datacenter performance analysis of a tensor processing unit", ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA), 2017, pp. 1-12.

[2] R. L. R. Junior et al., "Reliability of Google's Tensor Processing Units for Convolutional Neural Networks", 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks - Supplemental Volume (DSN-S), 2022, pp. 25-27.

[3] E. Vacca, et al. "A Comprehensive Analysis of Transient Errors on Systolic Arrays", 26th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS), 2023, pp. 175-180.

[4] Kundu et al., "Toward Functional Safety of Systolic Array-Based Deep Learning Hardware Accelerators", in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2021, vol. 29, no. 3, pp. 485-498.

[5] K. T. Chitty-Venkata et al., "Impact of Structural Faults on Neural Network Performance", 2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP), 2019, pp. 35-35.

[6] K. Cui et al., "A Diagonal Checksum Algorithm-Based Fault Tolerance for Parallel Matrix Multiplication" 2020 Eighth International Symposium on Computing and Networking Workshops (CANDARW), 2020, pp. 218-223.

[7] M. Safarpour et al., "Algorithm Level Error Detection in Low Voltage Systolic Array" in IEEE Transactions on Circuits and Systems II: Express Briefs, Feb. 2022, vol. 69, no. 2, pp. 569-573.

[8] F. Libano, et al., "Efficient Error Detection for Matrix Multiplication with Systolic Arrays on FPGAs", in IEEE Transactions on Computers, Aug. 2023, vol. 72, no. 8, pp. 2390-2403.

[9] J. Kim, et al., "ZOS: Zero Overhead Scan for Systolic Array-based AI accelerator", 2022 19th International SoC Design Conference (ISOCC), 2022, pp. 360-361.

[10] H. Lee, et al., "STRAIT: Self-Test and Self-Recovery for AI Accelerator", in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Sept. 2023, pp. 3092-3104 .

[11] J. Fuhrmann, " Implementierung einer Tensor Processing Unit mit dem Fokus auf Embedded Systems und das Internet of Things", 2018, http://hdl.handle.net/20.500.12738/8527

[12] V. S. S. Nair, et al., "Efficient techniques for the analysis of algorithm-based fault tolerance (ABFT) schemes", in IEEE Transactions on Computers, April 1996, vol. 45, no. 4, pp. 499-503.

[13] J. Yuen et al., "VLSI design and implementation of a self-testing systolic array chip for signal processing", 1992 IEEE International Symposium on Circuits and Systems (ISCAS), 1992, pp. 375-378 vol.1.

[14] C. . -I. H. Chen et al., "A self-testing and self-diagnostic systolic array cell for signal processing", 1991 Proceedings, International Conference on Wafer Scale Integration, 1991, pp. 75-81.

[15] J. J. Zhang et al., "Fault-Tolerant Systolic Array Based Accelerators for Deep Neural Network Execution", in IEEE Design & Test, Oct. 2019, vol. 36, no. 5, pp. 44-53.

[16] K. T. Chitty-Venkata et al., "Model Compression on Faulty Array-based Neural Network Accelerator", IEEE 25th Pacific Rim International Symposium on Dependable Computing (PRDC), 2020, pp. 90-99.

[17] Y. Zhao, et al., "FSA: An Efficient Fault-tolerant Systolic Array-based DNN Accelerator Architecture," IEEE 40th International Conference on Computer Design (ICCD), 2022, pp. 545-552.

[18] K. Huang, et al., "Algorithm-based fault tolerance for matrix operations" IEEE Transactions on Computers (Spec. Issue Reliable & Fault-Tolerant Comp.), 33 (1984), pp. 518-528.

[19] A. Roy-Chowdhury, et al., "Algorithm-based fault location and recovery for matrix computations", 24th International Symposium on Fault-Tolerant Computing, 1994, pp. 1239-1247.

[20] G. Bosilca, et al., "Algorithm-based fault tolerance applied to high performance computing",Journal of Parallel and Distributed Computing, 2009, vol. 69, pp. 410-416.

[21] L. Chen, et al.,"Extending checksum-based ABFT to tolerate soft errors online in iterative methods," 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS), 2014, pp. 344-351.

[22] J. Sommer, et al., "DSP-Packing: Squeezing Low-precision Arithmetic into FPGA DSP Blocks", 2022 32nd International Conference on Field-Programmable Logic and Applications (FPL), 2022, pp. 160-166.

[23] A. Anderson, et. al, "High-Performance Low-Memory Lowering: GEMM-based Algorithms for DNN Convolution," IEEE 32nd International Symposium on Computer Architecture and High Performance Computing, 2020, pp. 99-106.

[24] E. Sanchez et al."Effective emulation of permanent faults in ASICs through dynamically reconfigurable FPGAs," 24th International Conference on Field Programmable Logic and Applications (FPL), 2014, pp. 1-6.

[25] https://blogs.synopsys.com/breakingthethreelaws/2015/02/part-deux-how-many-asic-gates-does-it-take-to-fill-an-fpga