Image Test Libraries for the on-line self-test of functional units in GPUs running CNNs

Image Test Libraries for the on-line self-test of functional units in GPUs running CNNs

A. Ruospo*, G. Gavarini*, A. Porsia*, M. Sonza Reorda*, E. Sanchez*, R. Mariani†, J. Aribido† and J. Athavale†
*Politecnico di Torino, DAUIN, Torino, Italy
†NVIDIA, US

***Abstract*—The widespread use of artificial intelligence (AI)-based systems has raised several concerns about their deployment in safety-critical systems. Industry standards, such as ISO26262 for automotive, require detecting hardware faults during the mission of the device. Similarly, new standards are being released concerning the functional safety of AI systems (e.g., ISO/IEC CD TR 5469). Hardware solutions have been proposed for the in-field testing of the hardware executing AI applications; however, when used in applications such as Convolutional Neural Networks (CNNs) in image processing tasks, their usage may increase the hardware cost and affect the application performances. In this paper, for the very first time, a methodology to develop high-quality test images, to be interleaved with the normal inference process of the CNN application is proposed. An Image Test Library (ITL) is developed targeting the on-line test of GPU functional units. The proposed approach does not require changing the actual CNN (thus incurring in costly memory loading operations) since it is able to exploit the actual CNN structure. Experimental results show that a 6-image ITL is able to achieve about 95% of stuck-at test coverage on the floating-point multipliers in a GPU. The obtained ITL requires a very low test application time, as well as a very low memory space for storing the test images and the golden test responses.**

***Index Terms*—Deep Neural Networks, Reliability, On-line Self-test, Fault Injection, Functional Safety**

I. Introduction

The ever-increasing adoption of AI-based solutions in modern systems labeled as safety-critical is requiring the academic and industrial communities to increase their efforts to guarantee higher reliability for these products. Among the AI-systems, those based on Convolutional Neural Networks (CNNs) are among the most used for their outstanding capabilities in tasks like image processing and classification. In the last years, many standards have been proposed to guide the adoption of different mechanisms to face reliability issues. For example, the ISO 26262 standard is commonly followed in the automotive industry. Similarly, new standards are being released concerning the functional safety of AI systems (e.g., ISO/IEC CD TR 5469). Among the possible solutions adopted in the different fields, on-line testing strategies based on functional methods have been incorporated as a common solution by industry sectors such as the automotive one [1]. In these cases, the on-line test of the processor core and the related peripherals is performed through the periodic execution of Software Test Libraries (STLs) composed of a set of assembly programs able to thoroughly excite the processor core and detect possible permanent faults. Adopting STL solutions allows the system to perform the test on-line, and does not require any hardware overhead since it only needs a suitable memory space for saving the test libraries.

In the literature, STLs have been proposed as an effective safety mechanism to test in the field systems such as Graphics Processing Units (GPUs), widely used to accelerate AI applications [2], [3]. However, devising an STL requires a large amount of manual and semi-automatic work, since no EDA tools are available for their generation. In particular, the execution of specific STLs interleaving CNN inferences may jeopardize the strive for performance maximization [4].

Recently, an in-field testing solution for testing deep learning (DL) accelerators has been suggested by [5]. As a case study, they exploit NVDLA, an open-source Nvidia's DL accelerator. Their technique resorts to combinational Automatic Test Pattern Generation (ATPG) to generate functional test patterns to detect permanent faults in both computational and logic units. These patterns consist of sets of $\{\text{input, weight}\}$ pairs and are mapped to DNN test programs depending on the specific dataflow algorithm. For each ATPG pattern, a single DNN is created. However, the execution of specific DNN test programs (e.g., that can be more than 6k only for the computational units) involves a non-negligible time for memory transfers, and, above all, can be performed only during dead times of the systems, i.e., boot or reset. Additionally, the total test storage can require, in some cases, up to 600 MB only for the computational units.

This research work describes a method to develop high-quality test stimuli *in the form of test images* to be used during the normal CNN inference process. This requires finding a set of input pixels that, convolved with a set of fixed/known CNN weights, can detect with high test coverage (TC) permanent faults affecting the target hardware unit. The idea comes mainly from the following observation: when a CNN is deployed in the field, the trained version is loaded, and the weights never change. The *same* CNN can be exploited to periodically process carefully-developed test images to test on-line specific hardware units. A comparison mechanism is then adopted to possibly alert for the presence of a fault.

In this work, we describe a method to generate a set of images for the on-line test of the multipliers in a GPU, which have a relevant role in convolutional operations, and convolutional layers in CNNs account for more than 90% of the total operations [6]. This set of images constitutes the Image Test Library (ITL). Experimental results reveal that with a reduced set of test images, our technique achieves about 95% single stuck-at test coverage for all GPU multipliers. As a case study, two ITLs have been developed for two CNNs: ResNet20 and DenseNet121. In addition, we exper-

imentally demonstrate that the developed ITLs can propagate the effect of the faults up to the software level. Although the current paper focuses on NVIDIA GPUs, the method can be extended to other GPU architectures as well. The rest of the paper is organized as follows: section II provides the reader with some background knowledge about convolutional operations in GPUs. Section III describes the proposed approach. Next, Section IV reports on the experimental results. Finally, Section V draws conclusions and future directions.

II. Background

A. NVIDIA's Graphics Processing Units

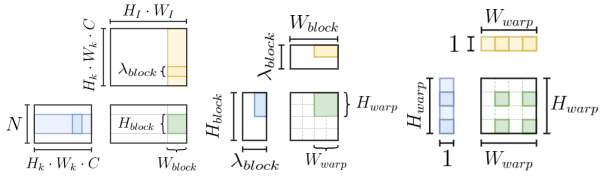
Among all the hardware accelerators used for efficiently running CNNs, GPUs are one of the most popular choices. Recent NVIDIA's GPUs feature a number of *Graphics Processing Clusters* (GPCs), each composed of multiple *Streaming Multiprocessors* (SMs). The SMs constitute the core of the accelerator, as they are in charge of most of the operations performed by the device. A single SM is composed by a group of *processing blocks*, each with a dedicated *warp scheduler*. Typically, a processing block is composed of multiple *CUDA cores*. At the software level, a GPU needs to handle hundreds of threads concurrently. For this reason, GPUs use a *Single-Instruction, Multiple-Threads* (SIMT) execution model. Each group of processes, namely a *thread block*, is assigned to an SM and is then partitioned in groups of parallel threads called *warps*. Subsequently, as an SM executes a thread block, each of its warps is dispatched to a warp scheduler, which issues instructions to the CUDA cores of the corresponding processing block in a SIMT fashion. While the mappings between thread blocks, warps and threads depend on the software implementation (with some hardware constraints), the mapping between threads and CUDA cores depends exclusively on the device.

B. Scheduling a convolutional algorithm

The core operation of CNNs is the convolution, corresponding to a function that, given an input tensor $I \in \mathbb{R}^{H_I \times W_I \times C}$ and a weight tensor $K \in \mathbb{R}^{H_k \times W_k \times C \times N}$ returns an output tensor $O \in \mathbb{R}^{H_O \times W_O \times N}$. Input and output tensors are also referred to as Input Feature Map (IFMAP) and Output Feature Map (OFMAP), respectively. This operation is often not executed directly: modern libraries use equivalent algorithms to efficiently implement the convolution operation, such as General Matrix Multiplication (GEMM), Winograd or Fast Fourier Transform (FFT). The most efficient algorithm for convolution, especially for large inputs, is GEMM. In GEMM, a convolution is reduced to a matrix multiplication, as show in figure Fig. 1a.

Understanding how GEMM works is crucial to correlate an element of the OFMAP with the specific core that computes its value. This correlation depends on two levels of scheduling:

- *Thread-core mapping*: the mapping between a software thread and a hardware core. It depends on the architecture of the device.
- *Workload-thread mapping*: the mapping between an element of the OFMAP and a software thread. It depends on the software implementation of the GEMM algorithm. NVIDIA's cuDNN and CUTLASS are two libraries that offer different implementations and, possibly, different mappings.



(a) Threadblock (b) Warp (c) Thread

Fig. 1: CUTLASS implementation of GEMM algorithm

In this paper, we refer to the *dataflow algorithm* as the ensemble of the convolutional algorithm, the thread-core mapping and the workload-thread mapping. While cuDNN and CUTLASS differ in their workload-thread mapping, they have a similar approach: they both split the computation of the output matrix into products of sub-matrices called *tiles*. Hence, knowing to which thread a certain tile is assigned means knowing the workload-thread mapping. Unfortunately, details about the implementation of the mapping between threads and tiles is not publicly available for the cuDNN version of GEMM. However, for CUTLASS, it is known [7]:

- *Threadblock-level GEMM*: Each threadblock computes a tile $H_{block} \times W_{block}$ of the output matrix (Fig. 1a);
- *Warp-level GEMM*: To each warp inside a block a portion $H_{warp} \times W_{warp}$ of a threadblock-level tile (Fig. 1b) is assigned;
- *Thread-level GEMM*: Each thread computes a certain number of elements of the warp-level tile. In particular, to take advantage of the SIMT architecture, the elements within a thread are tiled in a 2D structure, as shown in Fig. 1c

III. Proposed ILT Generation Method

This research work presents a method to develop test images for the on-line self test of multipliers in GPUs. The overall idea relies first on an ATPG-based approach (highly effective for regular structures like GPU functional units) to find out a set of suitable input values at the functional unit level, and then on transforming them into a test image. By processing these images using the CNN architecture and its weights (in particular, those of the first convolutional layer), it is possible to obtain a high fault coverage for the targeted unit. This methodology is described in details in Section III-A. Therefore, the produced test images can be executed in the field, by leveraging the same CNN, and by alternating "normal" inferences with the ITL self-test images (without moving or loading new weights in memory). In Section III-B, a methodology to validate the effectiveness of the test images is presented.

A. Image Test Library (ITL) Generation

The overall idea behind the generation of test images is illustrated in Fig. 2 and follows three stages: (i) dataflow algorithm extraction, (ii) ATPG-based patterns generations, and (iii) self-test images generation.

(i) Dataflow algorithm extraction: The goal of this phase is to derive how the different operations are scheduled on a specific GPU architecture, and how convolutions are performed. In other words, *to find a correlation between input pixels, CNN weights, and individual multipliers.* In GPUs, the dataflow algorithm is fixed in the architectural specification: it can be accessed by profiling the execution of the software and by tracking operations.

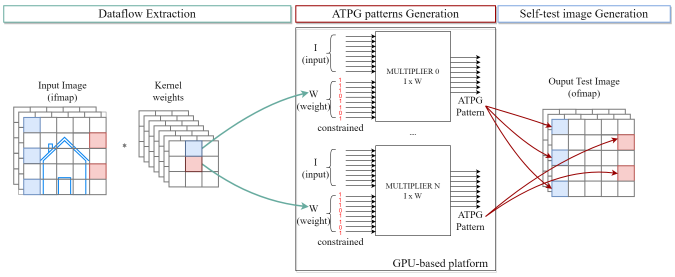


Fig. 2: Graphic representation of the proposed method to generate ITLs.

This correlation is necessary to find out the set of fundamental operations that each multiplier executes. The reader should note that by knowing the workload-thread mapping, the proposed solution can generate ITLs for every module for which we know the thread-core mapping. Details and exact figures on the specific software and device-dependent implementation are given in Section IV.

(ii) ATPG-based pattern generation: Once the dataflow algorithm is known, an ATPG process is set up to find out the collection P_c of input-weight pairs $\langle i, w \rangle$ that maximize the test coverage of the multipliers of a core c. The parameter w corresponds to the real trained weights W of the CNN. To this end, they are put as constraints for the ATPG generation. So, the resulting test patterns depend on both the actual CNN's weights and input values generated by the ATPG. For the on-line testing, such carefully-crafted $\langle i, w \rangle$ values are fed to the multiplier unit by means of suitable images composing the ITL. It is worth underlining that the ATPG process is executed only on the targeted module, and the obtained test patterns only relate to its inputs.

(iii) Self-test images generation: After the ATPG process, the ITL is built using the process described in Algorithm 1. The first step (line 3) is to reverse the mappings described in stage (i), associating each core with pairs of indices $\langle i_{idx}, w_{idx} \rangle$ of elements processed by that core. Here, i_{idx} is the index of an element of the input feature map I (i.e., $I(i_{idx}) = i$), while w_{idx} is the index of an element of the weight tensor W (i.e., $W(w_{idx}) = w$). When dealing with a convolution, a core reuses the same weight for different inputs. For this reason, given a weight index w_{idx} and a core c, a list of the associated input feature map indices $I_{idx}(w_{idx}|c)$ is built (line 4).

Given the list of suitable input positions for each core and weight, it is possible to reconstruct the images. The general algorithm consists of two nested loops: the outermost one cycles over the available cores, while the innermost one operates on the ATPG-generated pairs $\langle i, w \rangle$ associated to a specific core c. For each $\langle i, w \rangle$ pair, a list of input feature map indices $I_{idx}(w_{idx}|c)$ associated to the weight index w_{idx} of the element w of the weight tensor W is selected (lines 7-8). The result of this process is a collection of suitable positions where to put the inputs i, associated with weight w, returned by the ATPG processes. In line 9, the index of position is selected among all the *free* positions (i.e., not occupied by another pattern), across all the already-generated images. If a free space is not found, a new image is generated and a new position is chosen (lines 10-13). Finally, the input value i is assigned to the selected position (line 14).

Algorithm 1 Self-test Image Generation

Inputs: ATPG-patterns - Patterns for each core
Outputs: ITL - List of test images $[I_{fmap} \in \mathbb{R}^{H_I \times W_I \times C}]$

```
1:  ITL ← []
2:  for core ← 0 to n_cores do
3:      inputs ← MAP-CORE-TO-INPUTS(core)
4:      GROUP-BY-WEIGHT(inputs)
5:      patterns ← ATPG-patterns[core]
6:      for pattern, weight in patterns do
7:          w_idx ← GET-WEIGHT-INDEX(weight)
8:          positions ← inputs[w_idx]
9:          I_fmap, i_free ← FIND-EMPTY-POS(ITL, positions)
10:         if I_fmap = nil then
11:             I_fmap ← APPEND-NEW-IMAGE(ITL)
12:             i_free ← positions[0]
13:         end if
14:         I_fmap[i_free] ← pattern
15:     end for
16: end for
17: return ITL
```

B. ITL Validation

To validate the adoption of ITLs for on-line testing, it is fundamental to highlight their ability to (i) excite hardware faults of the targeted functional module, and (ii) let the faults propagate at the software level (where their occurrence is checked).

To study the cross-level propagation of hardware faults of a target unit, it is necessary to perform architectural-level fault simulations and, for each injected fault, check if the fault is propagated to the software level. In the literature, many FI tools perform architectural-level injections, by instrumenting the source code to corrupt all instruction set architecture registers (e.g., SASSIFI [8]), or by performing dynamic and selective code instrumentation (e.g., NVBinFI [9]), or with hybrid SASSIFI/Tensorflow solutions (i.e., CLASSES [10]). They all consider only the registers, PIs, and POs of functional units. To the best of our knowledge, only [11] propagates at the software level the impact of permanent faults in functional units, and not only registers, PIs and POs. They combine software profiling with gate-level microarchitectural fault simulation to build syndrome tables, a collection of fault syndromes. These are used during the execution of the CNN to support the code instrumentation and to propagate the error effects. However, one single hardware fault may produce multiple error syndromes during the CNN execution, and therefore the size of syndrome tables may be non-negligible.

The proposed idea stems from a mathematical observation. Let us consider the inputs (I and W) and the output (O) of a multiplier. In the presence of a fault affecting it, the product IxW may yield a faulty output $\widehat{O}$, that is: $I * W = \widehat{O}$

However, this fault can also be thought of as a faulty input ($\widehat{I}$ or $\widehat{W}$) entering a *golden* multiplier and producing the *same* faulty output $\widehat{O}$. Knowing the value of $\widehat{O}$ that derives from a fault affecting the multiplier, it is possible to obtain the respective faulty input ($\widehat{I}$ or $\widehat{W}$), that corresponds to the same fault without injecting

it. Assuming a *golden* multiplier, the *same* fault can be seen as:

$$\widehat{I} = \frac{\widehat{O}}{W}, \quad or \quad \widehat{W} = \frac{\widehat{O}}{I} \tag{1}$$

Furthermore, if a *faulty* multiplier performs J multiplications, there will be J corrupted outputs $\widehat{O}_j$ for $j \in [1, .., J]$. This is equivalent to having J multiplications executed by a *golden* multiplier with a set of corrupted inputs $\widehat{I}_j$ (or weights $\widehat{W}_j$), for $j \in [1, .., J]$.

In this work, we propose a methodology to perform very accurate software FIs by applying faulty inputs $\widehat{I}$ to the CNN which exactly correspond to specific hardware faults internal to the targeted functional unit. This approach has two main advantages: it combines the accuracy of the gate-level microarchitectural simulation with the speed of software FIs, and it allows us to experimentally demonstrate that the proposed self-test images (ITLs) can excite permanent faults inside functional units while propagating the effects up to the OFMAP of the first CNN's layer. The impact of hardware faults is not simulated by performing complex and costly multi-level simulation environments, but only launching the inference of *faulty images that exactly reflect a precise hardware fault within the first layer*.

The generation of faulty images that corresponds to injecting a specific fault within a multiplier is described in Algorithm 2. First, the fault is injected at low level in the multiplier (line 2). Then, for each operation performed by the multiplier during the convolution, its input weight $W[op]$ and the low-level faulty output $\widehat{O}$ are collected (line 4-5). These values are used to compute the faulty input $\widehat{I}[op]$ (line 6). Finally, the list of all the input elements (one for each operation) is converted to images following the same logic of Algorithm 1 (line 8).

To inject faults at application-level using the images generated in Algorithm 2, it is necessary to combine the information of the list of images in a single faulty OFMAP. Therefore, given a fault f_c affecting core c, the first step to fault simulate a layer l, is to generate the set of faulty images I_{f_c} as described in Algorithm 2.

Algorithm 2 Faulty Images for a fault in an HW multiplier.

Inputs:
- MULx - Selected multiplier;
- fault - A stuck-at fault of MULx;
- ITL - Image test library for a specific CNN;
- Operations - Pairs of ⟨input,weight⟩ multiplications performed by MULx during the convolution.
- n_op - Number of Operations

Outputs: FImg - List of faulty images for a single HW fault.

1: FImg ← []; $\widehat{I}$ ← []; W ← [];
2: MULX-INJECT(fault)
3: **for** op ← 0 **to** n_op **do**
4: $\widehat{O}$ ← MULX-MULTIPLY(Operations[op])
5: W[op] ← GET-WEIGHT(Operations[op])
6: $\widehat{I}$[op] ← $\frac{\widehat{O}}{W}$
7: **end for**
8: FImg[fault] ← PATCH-ITL($\widehat{I}$, W, ITL)
9: MULX-CLEAN(fault)
10: **return** FImg

The reader should note that the OFMAP of the first layer contains values that are not only computed by core c. For this reason, the application of a mask $M(c)$ (that depends only on the core c) to the output of the layer is required. An element of this mask is set to 1 if the corresponding element in the OFMAP is computed by core c, 0 otherwise. As such, the faulty output of the layer $l(i)$, for a clean input image i can be computed as the sum of element-wise multiplications:

$$l_{f_c}(i) = l(i) \cdot (1 - M(c)) + \sum_{i_{f_c}}^{\in I_{f_c}} l(i_{f_c}) \cdot M(c) \tag{2}$$

a) Software-level Observability: With the developed ITL, the TC achieved by executing the test images is observed at the output of the single multipliers. However, during the on-line self test, we want to fix the observability point at the software level. As a consequence, for each self-test image, the respective golden OFMAP of the first layer is stored (we refer to as *signature-ofmap*), and it is compared to the actual one on-line: if they differ, a warning is raised.

IV. Experimental Setup and Results

The effectiveness of the approach has been experimentally validated by using the scheduler and the architectural details of the NVIDIA Jetson Nano, which includes a 128-core NVIDIA Maxwell GPU. The proposed ITLs have been developed to detect permanent faults affecting the multipliers of the targeted GPU. This work does not take into account neither hardware-level mechanisms [12] nor OS-level mechanisms [13] used to guarantee the safety of the GPUs. For the sake of reproducibility, an open source unit from OpenCores was exploited [14]. This unit is a IEEE-754 compliant, single-precision, and signed 32-bit floating-point (FP32) multiplier. The RTL design has been synthetized with the 45nm NangateOpenCell Library [15] and a frequency of 50 MHz. The Synopsys TetraMAX tool has been used for the ATPG process. The synthetized gate-level unit features a total of 12,510 stuck-at faults.

We developed two ITLs, for two different CNNs (ResNet20 and DenseNet121) trained and tested on CIFAR-10 by using PyTorch. The ResNet20's first layer performs a convolution of stride 1 between a 32x32x3 input image and 16 filter weights of size 3x3x3, for a total of 432 FP32 weights. On the other side, the first convolutional layer of DenseNet121 has stride 2 and convolves a 32x32x3 input image with 64 filter weights of size 7x7x3 (9,408 FP32 weights).

a) ITL generation: The first step of the proposed method corresponds to extracting, for each GPU core, a list of all the weight-input pairs processed by every multiplier. This is possible by extracting the dataflow algorithm, as explained in Section III. In our case, we fixed the *convolutional algorithm* to be GEMM, and we observed the *thread-core mapping* through tracking operations. Note that changing this mapping could change the fault propagation and affect the TC of the targeted units. To have a knowledge of the *workload-thread mapping*, we added a PyTorch function to force the usage of the GEMM implementation provided by CUTLASS, since it is publicly available. Furthermore, we extended this implementation to (i) at the warp level, have output tiles with the same width as the weight matrix (i.e., the number of

TABLE I: Details about the ATPG process.

CNN	Num. of weights	Selected Weights	Num. ATPG patterns	Test Coverage
ResNet-20	432	144	128	93.58
DenseNet-121	9,408	576	135	94.28

filters/output channels N) and (ii) at the thread-level, force each thread to compute a whole output row (i.e., N output elements, 1 for each output channel). For ResNet20, with a threadblock size of 128 threads, given an input matrix of size 1024×27 and a weight matrix of size 27×16, we obtain 8 threadblock tiles of size 128×16, warp tiles of size 32×16, and thread tiles of size 1×16. This means that a single core is in charge of computing $16 \cdot 8$ elements of the OFMAP, performing $3,456$ multiplications.

The modified *workload-thread* mapping guarantees that each core processes all the weights at least once: since the constraints are the same, it is possible to launch one ATPG process for all the cores. Depending on convolution parameters, the convolution algorithm and the GPU architecture, some cores may always multiply some weights by 0-padding inputs. As we have no control over padding values, we should select only weights that are multiplied by a non-padding input element at least one time by each core. That is, we must exclude from the list every pair containing a weight that in at least one core is only multiplied by padding.

Details about the ATPG process are given in Table I. The second column reports the total number of weights used to perform the convolution of the first layer. For both CNNs, each core uses all the weights at least once during the first convolution. As for DenseNet121, since the amount of weights is substantial, we kept only 32 filters instead of 64 and selected a 3x3 region around the central element of each channel, to keep the process as close to ResNet20 as possible. By removing weights that in some cores are exclusively multiplied by the 0-padding ($\frac{2}{3}$ of all weights for ResNet20, $\frac{1}{3}$ of reduced weights for DenseNet121), we obtained the final list of candidate weights (Column 3$^{\text{rd}}$, Table I). An ATPG process was set up by putting as constraints the selected weights, and, for every one, a single ATPG pattern was found. Some weights did not originate patterns able to increase the TC. The TetraMAX process took about 0.17s of CPU time to find a single pattern. The number of final ATPG patterns is given in Column 4$^{\text{th}}$, and the final TC was equal to 93.58% for ResNet20, and 94.28% for DenseNet-121. Due to the constrained weights, the 3.89% and 4.01% were classified as ATPG Untestable, respectively. Clearly, these percentages are lower than the ones achieved in [5] on NVDLA's computational units. Indeed, their ATPG process did not place any constraints, modifying not only the input values but also the actual weights of the NN. For this reason, it is not suitable for on-line testing (we intentionally use real CNN's weights to alternate on-line inferences of "normal" images with self-test ones).

The ATPG patterns are then used to reconstruct the self-test images, as described in Algorithm 1. The reconstruction process requires knowing which pixel in the input image are processed by which core: this information is retrieved during the dataflow algorithm extraction. At the end of this procedure, we obtained 6 self-test images for ResNet20 and 8 for DenseNet121. Samples from the real ITLs are illustrated in Fig. 3a and Fig. 3b.

Next, starting from these ITLs, we performed a logic simulation resorting to Modelsim® HDL Simulation, by simulating the exact $\{$input, weight$\}$ pairs of the obtained images entering into each of the core's multipliers. For each test image, 128 (the total number of multipliers) value change dump (VCD) files have been collected. These VCDs have been used to run gate-level fault simulations with TetraMAX, to compute the exact TC that each self-test image achieves on each core's multipliers. We observed that the actual TC was always higher than the one computed at the end of the ATPG process. Indeed, apart from the selected weights (Column 3$^{\text{rd}}$, Table I), each core executes more operations, including different multiplications by zero (the 0-padding). It means that the proposed ITL generation guarantees a minimum TC value among all the GPU cores (the exclusion of those weights that, in some cores, multiply the 0-padding, avoid penalising individual cores).

Table II reports details of the two ITLs, in terms of (i) number of images, (ii) average TC over all the 128 cores, (iii) time required to run the ITLs and compare the golden signature-ofmap with the computed one, and the (iv) total storage required to support our self-test approach. As for the self-test time, it is worth underlining that the time required to run the inference of 6 CIFAR10 images is 0.35 ms for ResNet20, and 0.36 ms for DenseNet121 (images are run in a batch of 8 images: batches are always a power of 2). Furthermore, considering the memory space needed to store the ITLs, we sum up the space to store the self-test images (e.g., for ResNet20, 6 test images multiplied by 32x32x3x4 bytes), together with the space needed to store the golden test responses, i.e., the signature-ofmap for each image (e.g., 6 test images multiplied by 32x32x16x4 bytes). For the sake of completeness, Tables III and IV compare the TC of the proposed ITLs (in each warp) with the ones obtained by running the inference of checkerboard images, random images, and CIFAR10 images. An example of checkerboard images is given in Fig 3c; they seek to reproduce the well-known testing technique of applying specific checkerboard test patterns in assembly programs (e.g., 0xa5a5a5a5). Then, the same quantity of test images was selected for each type of ITL (the proposed, checkerboard, random, and CIFAR10), and gate-level fault simulations for each core in each warp have been performed. Each test image was fault simulated separately, and at the end, the 6 or 8 fault lists have been merged through TetraMAX. The final value is reported as the average over the 32 CUDA cores' multipliers in each warp. As emerging, TC values oscillate depending on the amount of 0-padding assigned to each core, but it is interesting to note that the proposed ITL's values are always higher than ones obtained at the end of the ATPG process (Table I). As shown in Tables III and IV, the main advantage of the proposed test images consists in the achieved test coverage: it is $\sim$13%, $\sim$9%, and $\sim$10% higher than the checkerboard, random, and CIFAR10 ITLs, respectively. It means that with a very low number of inferences, it is possible to cover about the 95% of stuck-at faults of the GPU's multipliers, without modifying the current CNN or undertaking costly memory load operations.

b) ITL Validation: Instead of performing low-level fault injections and propagating faulty values at the software level, we propose to inject hardware faults affecting the multipliers as a set of carefully modified images which mimic the same faulty output of the multiplier (without injecting it at gate level). Permanent faults affecting MUL_0 have been considered, and the faulty images

TABLE II: Details of the ITLs developed for testing on-line the FP32 multiplier.

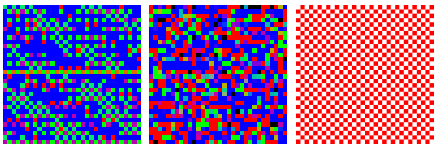
Proposed ITLs	Num. of images	Avg. TC [%]	Self-test time [ms]	Memory Space for storing the ITL [kB]
ResNet-20	6	94.74	0.35	467
DenseNet-121	8	95.46	0.36	623

TABLE III: ResNet-20: comparing the proposed ITL with Checkerboard, Random, and CIFAR10 images.

ITL type	Num. of images	Avg. Test Coverage on FP32 mul. [%]			
		warp0	warp1	warp2	warp3
Proposed ITL	6	94.72	94.76	94.76	94.72
Checkerboard ITL	6	81.37	81.12	81.12	81.46
Random ITL	6	85.23	84.73	85.01	85.13
CIFAR10 ITL	6	84.42	84.85	84.56	84.61

have been created by following Algorithm 2 for ResNet20. MUL_0 executes a total of 3,456 multiplications by using all weights more than once. Of all the weights in the first layer (16 filters of size 3x3x3), MUL_0 is responsible for performing 216 multiplications per filter. This means that in total, we get 16 patches (3,456/216) containing faulty values for the first layer only. These patches must be overlaid on each of the ITL images. Therefore, in total, each individual stuck-at fault corresponds to 6*16 faulty images.

To compute the $\widehat{O}$ and the $\widehat{I}$, a Modelsim HDL simulation was performed by injecting the stuck-at faults that have been marked as detected at the end of the gate-level fault simulation. Then, given $\widehat{I}$ and the respective ITL, the faulty images have been created (Algorithm 2). Finally, inferences on the faulty images have been performed with a PyTorch simulation, without changing the ResNet20 CNN model. To verify that the faulty ITLs can propagate the multiplier's faults up to the first convolution OFMAP, Eq. 2 was used to check for differences between tensors. They all produced a difference in the OFMAPs: all the detected faults (after the gate level simulation) are propagated and observed through the tensors (OFMAPs) of the first convolutional layer.



(a) (b) (c)

Fig. 3: Samples from the final ITLs: DenseNet121 (3a), ResNet20 (3b), and checkerboard (3c).

V. Conclusions and Future Works

This paper describes a method to develop test images able to detect on-line the occurrence of stuck-at faults in GPU's multipliers. We showed that with a very reduced set of images it is possible to cover about 95% of permanent stuck-at faults at the cost of a very low self-test time and a very low memory space for storing the ITLs. Future work will include the extension of the method to other units of a GPU. The main remark the reader may raise is that, in GPUs, thread-core and workload-thread mappings are not always publicly available. Extracting the dataflow algorithm can be done

TABLE IV: DenseNet-121: comparing the proposed ITL with Checkerboard, Random, and CIFAR10 images.

ITL type	Num. of images	Avg. TC on FP32 mul. [%]			
		warp0	warp1	warp2	warp3
Proposed ITL	8	95.45	95.47	95.47	95.45
Checkerboard ITL	8	81.69	81.58	81.22	81.68
Random ITL	8	86.01	85.38	85.4	85.91
CIFAR10 ITL	8	84.88	85.39	85.53	85.07

in specific cases (e.g., CUTLASS) and by performing profiling steps. Moreover, in this work, the developed ITLs are specific to multiplier units. In the future, we plan to extend the technique to other computational and logic units. A final consideration is related to the observability point: comparing the ofmaps within the CNN (after the first layer in our case), require considering the CNN as a white box. The best solution might be to consider the CNN as a black box (i.e. fix the observability point at the output), but, in this case, a thorough study of fault propagation must be carried out to take into account the CNN's intrinsic masking ability. We intend to follow this direction in delivering ITLs.

References

[1] P. Bernardi, R. Cantoro, S. De Luca, E. Sánchez, and A. Sansonetti, "Development flow for on-line core self-test of automotive microcontrollers," *IEEE Transactions on Computers*, vol. 65, no. 3, pp. 744–754, 2016.

[2] S. Di Carlo *et al.*, "A software-based self test of cuda fermi gpus," in *2013 18th IEEE European Test Symposium (ETS)*, 2013, pp. 1–6.

[3] J. E. Rodriguez Condia *et al.*, "Using STLs for effective in-field test of GPUs," *IEEE Design Test*, pp. 1–1, 2022.

[4] A. Ruospo, D. Piumatti, A. Floridia, and E. Sanchez, "A suitability analysis of software based testing strategies for the on-line testing of artificial neural networks applications in embedded devices," in *2021 IEEE 27th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, 2021, pp. 1–6.

[5] Y. He, T. Uezono, and Y. Li, "Efficient functional in-field self-test for deep learning accelerators," in *2021 IEEE International Test Conference (ITC)*, 2021, pp. 93–102.

[6] G. Desoli *et al.*, "14.1 a 2.9tops/w deep convolutional neural network soc in fd-soi 28nm for intelligent embedded systems," in *2017 IEEE International Solid-State Circuits Conference (ISSCC)*, 2017, pp. 238–239.

[7] "Cutlass," https://github.com/NVIDIA/cutlass, accessed: 2022-12-15.

[8] S. K. S. Hari, T. Tsai, M. Stephenson, S. W. Keckler, and J. Emer, "Sassifi: An architecture-level fault injection tool for GPU application resilience evaluation," in *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2017, pp. 249–258.

[9] T. Tsai, S. K. S. Hari, M. Sullivan, O. Villa, and S. W. Keckler, "Nvbitfi: Dynamic fault injection for GPUs," in *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2021, pp. 284–291.

[10] C. Bolchini, L. Cassano, A. Miele, and A. Toschi, "Fast and accurate error simulation for CNNs against soft errors," *IEEE Transactions on Computers*, pp. 1–14, 2022.

[11] J. E. R. Condia *et al.*, "A multi-level approach to evaluate the impact of GPU permanent faults on CNN's reliability," in *2022 IEEE International Test Conference (ITC)*, 2022, pp. 278–287.

[12] "NVIDIA Xavier Achieves Industry First with Expert Safety Assessment," https://blogs.nvidia.com/blog/2020/05/20/xavier-achieves-industry-first-safety-assessment/, [Online; accessed 23-December-2022].

[13] "Safe Travels: NVIDIA DRIVE OS Receives Premier Safety Certification," https://blogs.nvidia.com/blog/2022/12/16/nvidia-drive-os-tuv-sud-safety-certification/, [Online; accessed 23-December-2022].

[14] "Opencores, floating point adder and multiplier," https://opencores.org/projects/fpuvhdl, accessed: 2022-12-15.

[15] "Open-cell library," https://si2.org/open-cell-library/, accessed: 2022-12-22.