POLITECNICO DI TORINO

Master's Degree in electronic engineering

Master's Degree Thesis

New techniques for assessing and enhancing the reliability of DNNs

Supervisors

Prof. Matteo SONZA REORDA

Prof. Marco LEVORATO

Dr. Juan David GUERRERO BALAGUERA

Candidate

Francesco PESSIA

2024

Summary

Nowadays, machine learning (ML) algorithms are being exploited in a variety of applications, from health care to avionics, from computer vision to natural language processing. Furthermore, an increasing number of tasks exploiting deep learning are being performed on edge devices such as smartphones or drones through new programming paradigms, like dynamic neural networks or split computing [1],[2]. Semiconductors manufacturers produce AI accelerators, such as GPUs, able to provide outstanding power consumption and latency performances, leaving out *reliability*, which is a crucial factor in safety-critical applications, such as autonomous vehicles. In fact, the increasingly miniaturization of systems on chip (produced using 7 *n*m, 10 *n*m, 20 *n*m technology), exposes them to manufacturing defects escaping end-of-production tests or fault activation due to electromigration or aging during in-field utilization. This work aims at evaluating the effects introduced by hardware permanent (*"stuck at"* model) faults in GPU architectures while executing DNN workloads. Furthermore, it proposes new techniques for software-based fault detection and application hardening as well as a new version of an accelerator able to compute them.

Acknowledgements

ACKNOWLEDGMENTS

My sincere regards are devoted to Prof. Sonza for coordinating my activities and sharing his network with me. This experience would have not been possible without your supervision.

A special acknowledgement goes to Prof. Levorato for hosting me in his research laboratory in Irvine. Meeting you and your phD. students has been enlightening.

Greetings to Dr. Balaguera for sharing his knowledge and paper writing skills with me. I would also like to thank you to help improving my interest towards scientific research.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Semiconductor manufacturers produce AI accelerators, such as GPUs, able to provide outstanding power consumption and computational performances, but often neglecting *reliability*, which is a crucial factor in most safety-critical applications, such as autonomous or semi-autonomous vehicles. In fact, new generation systems on chip might be affected, with a not irrelevant probability, by manufacturing defects escaping end-of-production test or fault activation (due to electromigration or aging) during utilization. Nevertheless, *dependability* study over such complex architectures is most of the times problematic due to platforms complexity. On the other hand, to increase the reliability of hardware platforms, several testing techniques allowing fault detection during utilization have been already implemented, such as *Design for Testability, Built-in Self-Test* and *Software-Based Self-Test* (SBST). For instance, an outstanding *Software-Based Self-Test* solution has been designed by NVIDIA researchers able to obtain a fault coverage up to 92.4% in *Tensor Core Units* (TCUs), crucial platforms for deep learning (DL) applications, with a latency of only 100ms [3]. Although providing very high fault coverage, overhead is introduced to monitor hardware integrity reducing application performances by 2% in the automotive domain. Therefore, the mentioned methodologies, despite obtaining optimized test coverage, either require modification of architecture RTL description or block application execution. In this regard, detection procedures, based on *Convolution Neural Networks* (CNN) behavior, need to be devised. Nevertheless, fault injection campaigns at the application level, based on an accurate description of underlying hardware require sensitive simulation time. Therefore, alternative fault injection methodologies, designed to optimize simulation complexity, have been already proposed. Nevertheless, the proposed solutions only target *single-cycle*, *single flip flop* bit flip failures, belonging to the category of *"transient faults"*. For the best of our knowledge, no work has been proposed yet for modeling permanent faults (for the purpose of this work resorting to the *stuck-at fault model*) in TCUs. In this work, we will resort to experimental data gathered from multiple fault injection campaigns to extract patterns in faults behavior and propose a modeling methodology able to accurately replicate the effects of standard injection procedures. Subsequently, exploiting discrepancies in the CNN response with respect to fault-free scenarios, we propose a software-based fault detection technique able to provide a good level of fault coverage (78% of targeted faults) without reducing application performances. To conclude, training similar CNNs with several pooling filters (Average, Max, Median) we discovered that other hardware resources of GPUs (i.e., planar processors) can be exploited to mitigate faults in TCUs, at times compensating their effects. In particular, Median filter-based CNNs provide higher classification accuracy with respect to Average and Max filters, when GPUs are damaged. Nevertheless, Median pooling is very rarely exploited in commercial applications and more used in research. Therefore, to the best of our knowledge, no modern *Systems on Chip* (SoCs) support the execution of this layer. In this work, we developed a new version of NVDLA (NVIDIA deep learning accelerator) able to execute median filters with small kernel shapes (2x2, 3x3) with identical latency of Max and Average pooling.

The final manuscript, structured in chapters, provides: i) an overview about the state of the art with particular emphasis both on modern GPU hardware resources and cutting edge software-based detection techniques; ii) a survey about the environment exploited to perform fault injection campaigns and observations on gathered experimental results, that led us to devise a modeling procedure able to accurately map TCU permanent faults into *bit flip masks*; iii) a Software-based fault detection technique able to monitor the hardware integrity exploiting *LeNet5* behavior and finally iv) some software-based application-aware hardening techniques, together with an hardware accelerator designed to efficiently execute them.

Chapter 2

State of the Art

As the utilization of GPUs is getting popular among edge devices, especially for ML applications [4], reliability analysis mandated by safety protocols (ISO262) must be performed over such complex hardware structure. Semiconductor vendors designed highly parallelised platforms, such as GPUs, to optimize power consumption and latency that are key factors to estimate the performances of edge computing devices [2]. In fact, embedded systems executing computer vision algorithms for self-driving vehicles aim to reduce *Machine Learning* (ML) models inference latency, in order to exploit as much as possible the maximum frame rate of cameras. In order to meet such strict computational requirements, either specifically designed hardware is devised according to ML model parameters, or software-based techniques are exploited to improve model execution efficiency on general purpose SoCs. While the former option is usually implemented on FPGAs, the latter benefits from the high flexibility of general purpose structures, like GPUs.
In this chapter we provide a survey about GPUs structures with particular emphasis on the hardware resources exploited during CNNs computation. In this regard, we provided a short overview of a core called *Streaming Multiprocessor* (SM) that is the basic computational unit of GPUs. We will mainly focus on illustrating the hardware resources embedded in SMs exploited in ML applications (i.e. *Tensor Core Unit*). Starting from a short survey about the GPU architecture, we explain how the workload is generated (i.e, *threads* allocation) and assigned to available hardware resources (*Scheduling Policies*). The illustrated architecture inspired the design of a parametric GPU simulator, equipped with fault injection infrastructures, exploited to perform reliability studies. To conclude a short survey on cutting-edge software-based fault detection techniques is provided as a benchmark to evaluate the inedited proposed solution.

2.1 GPU Architecture

Modern GPUs are composed of several computational units called *Streaming multiprocessors* (SMs). Each SM can include up to 64 general-purpose parallel processing cores to execute integer and floating-point operations and several hardware accelerators for matrix multiplications or highly parallellized operations, such as Special Function Units (SFUs) or Tensor Core Units (TCUs). In particular, TCUs are hardware arrays of Dot-Product-Units (DPUs) comprising several Multiply-and-Add (MAC) cores specially designed to efficiently execute GEMM operations in the DL domain [5], [6]. Through a single (or a minimal set of) assembly instruction for the TCU, a 4x4 matrix multiplication multiply and accumulate operation (e.g., D = AB + C) can be computed. TCUs can support mixed precision operations (input FP16, output FP32) and compressed precision, such as INT4, INT8. In general, most GPU architectures include SMs with at least 2 TCUs. In high-performance class GPUs, such as the Titan V, the number of TCUs rises up to 8. Every SM core is equipped with internal *Cache* memory (L1) and specialized registers to merge cache miss to minimize redundant memory accesses. Furthermore, SMs are organized into clusters to device a scalable design. All SMs in a cluster share a Network on Chip (NoC) and have to compete against each other for memory bandwidth, potentially causing bottleneck effects [7]. *Intra-cluster cohaleshing units* (ICCU) merge data requests from all SMs in a cluster and access an upper level of *Cache memory* (L2) shared across all clusters. The power consumption for GPUs mainly stems from memory access rather than computational units. Therefore, ICCUs might optimize drastically energy consumption if SMs on the same cluster work on similar data. Semiconductor vendors implemented in hardware several scheduling policies to distribute application workload across available computational resources. The performances of schedulers are mainly measured through *Intra-Cluster Locality* (ICL) that stems from redundant data requests without taking into consideration reliability (more about this on hardening techniques chapters).

To conclude, some families of NVDIA GPUs such as *Jetson Xavier*, also comprise NVDLA (NVDIA Deep Learning accelerator) in their architecture.

2.1.1 NVDLA

The NVDIA Deep Learning accelerator is an open source system on chip, designed to optimize the execution of CNNs. It supports a pipeline with up to 13 stages, each computed by a different computational unit. It can support both fusion and independent layer computation. While the former execution mode joins the output of a pipeline stage to the input of next one, the second configuration performs *memory-to-memory* operations. In fact, an *AMBA AXI 4 bus* master/slave interface (DBBIF) provides direct access to an external DRAM (see Fig. 2.1). Furthermore, a host system (*NVDLA wrapper*) leverages on a *Configuration Space Bus* (CSB) interface, based on a valid/request protocol, to accesses and configure NVDLA address space. Through a 32 bits data bus and a 16 bits address, registers are written in order to modify control signals of functional units data path. In order to minimize configuration latency, a ping-pong synchronization has been devised to access the address space. According to this synchronization paradigm, for each computation unit, two identical address spaces are allocated: **consumer** and **producer**. During execution, while the configuration of consumer address space is exploited to drive control signals of data path, the producer address space can be modified through the CSB interface. Alternating producer and consumer address spaces in driving control signals, improves drastically configuration latency and therefore system performances. Some systems may have the need for more throughput and lower latency than the DRAM can provide, and may wish to use a small SRAM as a cache to improve the NVDLA's performance. A secondary

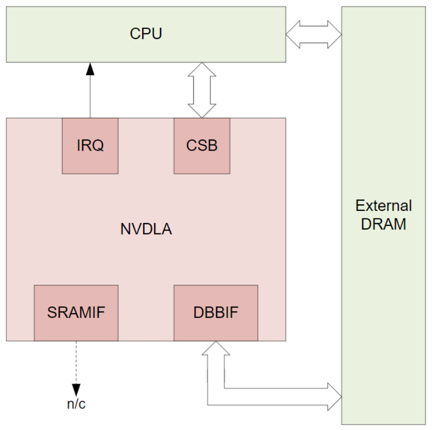

Figure 2.1: Small NVDLA system block diagram

AXI4-compliant interface (SRAMIF) is provided for an optional SRAM to be attached to the NVDLA. Nevertheless, in this work we will resort on the so called *Small NVDLA system* to evaluate functional tests keen to validate our modification on this accelerator. Our new version of NVDLA supports the computation of new operations geared toward alleviation of TCU permanent faults effects. NVDLA supports both half integer (INT8, INT16) and floating point (FP16) precision.

2.2 GEMM Execution on GPUs

Graphics Processing Units (GPUs) are based on the *multiple instructions multiple data* (MIMD) programming paradigm. During CNNs inferences, convolutions and fully-connected layers are mapped into *General Matrix Multiplications* (GEMM) suitable for *single instruction multiple data* (SIMD) programs. Nevertheless, in order to fit convolution layers into general matrix multiplications (GEMMs), whose workload is efficiently executed by GPUs, the *imm2col* algorithm is exploited, increasing memory redundancy and zero sparcity [8] (see Fig. 2.2). On the other hand, linear algebra theorems, implemented in software through cuBLASS libraries for NVDIA GPUs, can now be exploited during GEMM to partition and distribute the computation workload across several parallel cores [8].

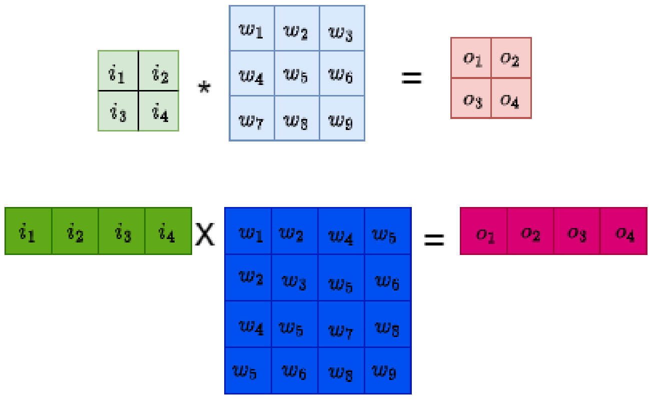

Figure 2.2: Imm2Col algorithm to transform single channel input tensors

During GEMM computation, a single kernel function is exploited to perform dot products between vectors. Therefore, multiple threads are issued, each executing the same kernel function across different sets of data. In order to fit data flow into *Cache memory* levels and exploit as much as possible all available parallel cores, the computation of the GEMM output matrix (also known as tensor), is split in sub-blocks or *"tiles"* [9]. Each *tile* is computed through a set of threads called *Cooperative Thread Array* (CTA). Therefore, each thread executes kernel functions through *multiply and accumulate* (MAC) operations over few data, threads are grouped into warps to cover sub-vector dot products, and finally warps are merged into CTAs to compute *tiles* or sub-matrices of output tensors. Each *tile* is computed through a series of sub-matrices multiply and accumulate operations ($D = AB + C$). Input tensors are sliced, according to tiling size, respectively into $M_x \times K_s \times N_y$ and $M_x \times N_y$ blocks [9]. Tensors blocks are fed to operational units (SM) for computation resorting on an hierarchical memory structure.

2.3 Scheduling Protocols

In this work, the role of 5 frequently implemented scheduling policies has been evaluated in case of permanent faults in TCUs:

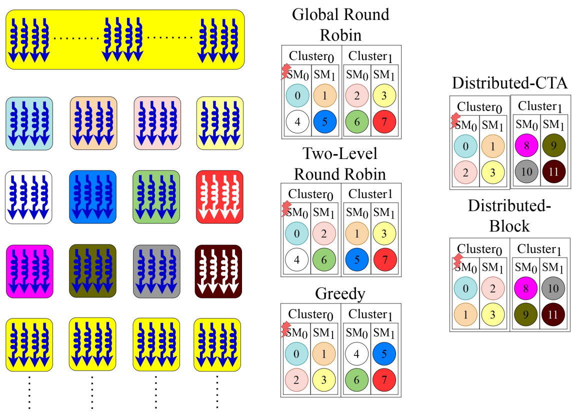

Figure 2.3: Scheduler static allocation of a pool of 16 CTAs across a hypothetical GPU comprising 4 SMs with a 2 CTAs wide *buffer*, divided on 2 clusters

1. **Two Level Round Robin**: CTAs are distributed on SMs of different clusters until every cluster has at least one SM with a CTA allocated (see Fig. 2.3). Subsequently, this process is repeated as long as all SMs on GPU have been provided with a CTA. According to data request arrival time, additional CTAs are allocated to SMs following the same paradigm.

2. **Global Round Robin**: CTAs are allocated across a new cluster if and only if all the SMs of a previously scheduled cluster have at least one CTA. Once all SMs of GPU have data to compute execution is triggered and once more according to data request arrival time new threads array are scheduled

3. **Greedy**: unless SMs support multiple allocation through *data buffers*, this scheduler behaves exactly as Global Round Robin. Otherwise, in order to move allocation to a new cluster, all buffers of all SMs in current scheduled cluster must be filled completely.

4. **Distributed Schedulers**: while *Greedy* or *Round Robin* might generate unbalanced workload across parallel processing units for applications with limited amount of CTAs, schedulers designed following the paradigm of *distributed allocation* ensure that workload is equally distributed across clusters. In fact, before scheduling is initiated, spatially continuous CTAs are divided into *pools* of uniform size. Furthermore, due to continuity, CTAs in the same *pool* tend to reuse same data, improving number of mergeable *Cache miss*. This behaviour results in limited number of redundant data request, improving ICL. In fact according to literature, *distributed schedulers* have the best performances in terms of power consumption [7]. Nevertheless, optimized level of ICL might reduce performances of these schedulers while evaluating their *reliability*. Two

commonly devised *distributed* schedulers are:

i) Distributed CTA: CTAs belonging to same *pool* are allocated across SMs until all of them have been scheduled. Then, if SMs support multiple allocation through *buffers*, the schedulers resumes allocation until all buffers are full. Once execution is triggered, additional CTAs are scheduled according to data request arrival time

ii) Distributed Block: unless SMs are provided with *buffers* it behaves exactly as Distributed CTA, otherwise allocation on buffer belonging to a SM is prioritized with respect to scheduling across SMs on the cluster. This procedure improves data locality of CTAs computed by the same SMs, increasing number of merged *Cache miss* of L1 *Cache memory.*

2.4 Fault Detection Techniques

Tech companies raised serious concerns about hardware faults impact both on training and inference of DNNs. The probability of fault occurrences in big data centres, running training scripts across multiple GPUs is not negligible (few cores over several thousands server machines [10], [11]). Google has conducted a deep study assessing impact of transient faults during training [12]. In particular, they underlined the necessity of good detection techniques in order to accredit training accuracy divergence to hardware. This necessity stems from the incredible amount of time spent by their engineers to review training scripts looking for fictitious software bugs. They suggest to exploit gradient history, calculated according to Adam optimization algorithm, comparing its value against a threshold. In addition, they discovered that majority of faults express their effect generally after two training iterations. Therefore, they suggest to save training status accordingly in order to restore the simulation from a valid configuration in case of abrupt training accuracy divergence caused by transient fault. Nevertheless, this detection methodology requires labelled inputs to calculate loss function, making it unsuitable for in field utilization. Furthermore, *NVDIA* researchers conducted a study to analyse and asses the impact of permanent faults on Object Detection Algorithms such as YoloV5 [13]. To mitigate simulation complexity, they modelled faults by applying single *bit flip* masks on the weights, emulating the effects of *stuck-at faults* of memory cells. In order to improve stability of YoloV5 execution, they suggest to implement hardware logic to sustain information redundancy such as Parity bits, Hamming code. Once concurred that a fault is present, the damaged portion of hardware usually is detected and isolated though *Build-in Self-test* (BIST) or *Software-based self-Test* (SBST). A study [14] has presented a methodology to generate a SBST for *NVDLA* exploiting *Deep Neural Networks* (DNNs) architecture. They suggest to map Test Vectors (TVs), generated through *ATPG* from RTL descriptions of open source *NVDLA* SoC, into a fictitious CNN, exploiting fixed network architecture and CNNs predicable execution procedure. Nevertheless, the ATPG was able to generate TVs only across the logic in charge of convolution computation due to the limited amount of sequential logic. Different strategies should be adopted to test other units such as control units that have complex sequential logic. This elaborate will present a new software-based technique to detect if faults are present and quantifies its performances in terms of detection accuracy. Furthermore, the new methodology is designed to run in parallel with the application, only slightly increasing its computation latency. Nevertheless, this procedure will only establish hardware integrity, eventually triggering additional tests to identify and isolate damaged circuitry.

Chapter 3

Fault Injection

Dependability evaluation of modern hardware structures requires clever investigation techniques producing reliable numeric results without relying on exhaustive test. In fact, according to the granularity of *hardware* description (i.e, high-level, RT-level, gate-level model) the computation time required by fault injection campaigns might became not negligible. Evaluating the effects of hardware faults during DNN inference can be computationally prohibitive without generating representative error models with low computation cost. Some studies have tried to asses performances degradation of CNNs caused by hardware faults through applying *bit flip* masks on weights [13]. Nevertheless, this procedure mainly targets *"stuck-at"* faults in memory cells, that can be easily detected through information redundancy (Hamming code) and eventually identified and isolated through BIST. Modelling the effects of TCUs *stuck-at faults* exploiting low computation error models such as *bit flip masks* is the goal of this fault injection campaign (FIC). Subsequently the generated error model will be exploited to implement *software*-based detection procedures on this crucial hardware resource for GEMM.

3.1 Fault Examination Methodology

To study the effects of permanent faults in TCUs we leverage on several GPU emulators equipped with fault injection infrastructures to perform reliability evaluations. Through gathered experimental results, we propose a modeling procedure to map faults behaviour in an alternative low computationally cost injection methodology (*bit flip masks*). The proposed injection procedure performs fault injections at the application level in acceptable amount of time (e.g, few hours instead of 10,000 days) without losing the accuracy of classical injection methodology. Through a model able to inject hardware-aware errors, detection procedures as well as hardening techniques, based on discrepancies of CNNs behaviour w.r.t. fault free scenario, can be proposed and evaluated.

3.1.1 GPUs Emulation and Injection

In order to evaluate the *reliability* of different scheduling policies, as well as modeling faults behavior, several GPU structures have been studied (see Table 3.1). For each GPU structure and for each scheduler, a FIC is performed targeting a single SM identified as *faulty*.

GPU name	clusters	SMs per cluster
Jetson AGX 32	2	7
Jetson AGX 64	2	8
Jetson Nano	1	1
Jetson TX2	1	2
Jetson Xavier	1	7

Table 3.1: GPUs structures evaluated by the fault injection campaign

GEMM operations have been computed through an instruction-accurate TCU model (*PyOpenTCU* [15]) able to execute tiles (16x16) of output tensor. Furthermore, the TCU model includes a *Fault Injector* (FI), able to inject *stuck at* faults on data-path. In addition, behavioural scheduler algorithms have been designed to allocate GEMM computations across the available SMs. Therefore, parametric GPUs emulators have been designed able to compute CNNs workload. Tuning computation resources (i.e, SMs, TCUs per SM, scheduler) through a configuration file, the user can arbitrarily target any commercial GPU for reliability evaluations. The TCU high-level model description comprises 86,000 locations accessible for fault injection campaigns. A *Statistical Fault Injection campaign* (SFI) has been performed with a confidentiality level of *95%* and error margin of *1%* injecting only 8,600 faults [16]. For each injected fault a seed matrix multiplication is calculated and *tiles* are allocated to SMs according to scheduling policy under examination. The output tensor, calculated by the faulty GPU, is compared against fault-free scenario and key information about fault spatial and scalar effects are stored. The methodology has been implemented exploiting high-level programming language (*Python*) modules:

1. **Golden**: this module is in charge of calculating the fault-free scenario. It randomly generates normalized activation ($a \in (0, 1]$) for seed GEMM and calculates *golden values* by leveraging on a *'Mock'* scheduler that allocates tiles only to one TCU emulator lacking FI. All tensors are stored in files for post processing by other modules

2. **Fault List generator**: in this work we only injected *stuck at* faults (no transient faults) uniquely identified through an integer identifier associated to a fault in the TCU data path by FI. This module generates a list of randomly selected fault identifiers (FIDs) from exhaustive test pool as well as configuring target GPU structure for FIC.

3. **Injector**: this module performs a single fault injection. The scheduler behavioural algorithm is triggered to allocate CTAs to TCUs of SMs. In order to optimize simulation complexity, only tiles allocated to faulty TCUs are computed leveraging on *pyOpenTCU* module. CTAs scheduled to reliable resources are computed exploiting matrix multiplication algorithms provided by *numpy* class. In addition, to further compress FIC complexity, this module has been designed to support multiprocessing execution. Hence, multiple fault injections, each associated to a process, can be executed in parallel. Through a *Queue* (see Ap. A.0.1), *faulty* GEMMs are provided to an additional process executing Validator module that extracts faults features.

4. **Validator**: the execution of this module, is triggered as soon as the Fault list generator module is completed. As soon as data have been pushed into the *Queue*, comparisons against fault free scenario extrapolate faults features. In particular the absolute coordinates (x, y) of corrupted GEMM elements are exploited to model faults spatial propagation Furthermore, the mean absolute error (MAE), defined as the average of absolute errors of corrupted elements, and mean relative error (MRE) are computed for fault classification (e.g. critical faults (CF) not observed (NO) faults). To conclude, the hexadecimal representation (IEEE 754 float16 parallelism) of corrupted elements and expected golden values are stored to model hardware-aware errors able to correctly replicate faults scalar effects.

3.1.2 Data Evaluation

Spatial Propagation

To replicate different faults behavior, it is imperative to determine which elements of faulty GEMMs are going to be effected by damages in TCU data-path. Subsequently, data corruption, exploiting low computation error models, will only occur in forecast GEMM elements. Corrupted elements position in output tensor mainly depends on 4 factors: i) GPU target structure, ii) scheduler allocation, iii) the fault location and iv) data capability to activate the faults. Therefore, storing absolute coordinates of corrupted elements in fault injection campaign (FIC) executing a study case GEMM, is not going to correctly generalize the fault spatial propagation. One of our publications has shown that spatial propagation of faults strongly depends on

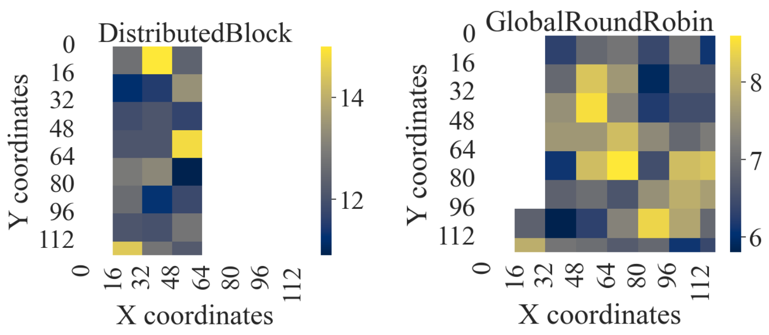

Figure 3.1: CTA allocation probabilities for GEMM with output matrix dimensions(120 x 120) for Distributed Block and Global Round Robin schedulers on *Jetson AGX 32*

different levels of Intra-cluster locality (ICL) associated to scheduling policies [17]. In particular, for GPU structures comprising more than one cluster and leveraging on distributed scheduler for CTAs allocation, the observed effects of TCU faults are localized. Furthermore, Distributed Schedulers can increase up to 6 % the corruption effects of TCU permanent faults with respect to *Round Robin* Schedulers in large GPUs such as *Jetson AGX32* [17]. In order to quantitatively evaluate the region of interest (ROI) of fault effects propagation we proposed to leverage on CTA corruption probability (eq. 3.1) :

$$\begin{aligned}
&P_{CTA}(Scheduler, CTA_{id}) = \frac{\sum_{id=0}^{\#faults-1} Corruption(Scheduler, CTA_{id})}{\#faults} \\
&Corruption(Scheduler, CTA_{id}) = \begin{cases} 1 : CTA_{id} \in CTA_{allocation}(Scheduler) \\ 0 : others \end{cases} \\
&CTA_{allocation}(Scheduler) = Scheduler.allocate(FaultySM)
\end{aligned} \tag{3.1}$$

The propagation probability is calculated for each CTA_{id} of GEMM and quantifies the likelihood of a scheduler to allocates that CTA to faulty SM workload (.allocate method). Through dividing the number of faults corrupting CTA_{id} by total injected faults, scheduler role in faults spatial propagation is evaluated. Schedulers with very efficient ICL such as Distributed Block are characterized by localized ROI ($P_{CTA}(CTA_{id}) > 0$, see Fig. 3.1). The corruption localization stems from the allocation procedure. In fact, distributed schedulers statically divide spatially continue CTAs into the same *pool*, scheduled to a cluster. Hence only a localized portion of the output tensor is going to be executed by the cluster comprising faulty SM. On the other hand, Round Robin schedulers evenly distribute CTAs, corrupting a larger portion of faulty tensor. In fact, only the CTAs statically allocated at the beginning of execution to reliable SMs have no corruption probability (see Fig. 3.1). Aforementioned, while this patterns is only visible for

GPU structures with more than one cluster(*Jetson AGX 32*, *Jetson AGX 64* see Tab. 3.1), for smaller GPUs such as *Jetson Nano* there is no relevant difference between examined schedulers. Furthermore, for every scheduling policy we noticed that in ROIs, CTAs probability of corruption is evenly distributed, precluding the possibility of exploiting these probabilities to model Schedulers behaviour. In fact, scheduler allocation is determined only at run time according to *data request arrival times.* Therefore, the same scheduler allocation can change from GEMM to GEMM.

On the other hand, though gathered experimental results, we noticed that only CTAs executed by faulty SM comprised elements discrepant with respect to fault free scenario. Therefore, we propose to model faults spatial behavior leveraging on scheduler behavioral algorithm to determine CTAs allocation. Data corruption is performed only on CTAs elements scheduled to faulty TCU. Nevertheless, TCU comprise multiple *Dot Product Units* (DPU) and, according to fault location, only one DPU is damaged at the time. Hence, only the elements computed by faulty DPU are going to be corrupted. According to this reasoning, we propose to leverage on corruption probabilities to determine the CTA elements computed by faulty DPU. As a first step, we collapsed all corrupted elements positions (x,y) into a coordinate system bounded by tiling size (x', y') exploiting reminder operator (see eq. 3.2).

$$\begin{aligned} x' &= x\%M_x \\ y' &= y\%N_y \end{aligned} \tag{3.2}$$

Afterwards, we evaluated corruption probabilities P(x',y') for each element of CTA (see eq. 3.3 where g and f are respectively the golden and faulty output tensor). Through *Propagate* function, called recursively for all corrupted entrances (x,y), it is possible to evaluate how often a coordinate (x', y') is injected. Therefore, dividing the corruption frequency of each element by the total scheduled CTAs to faulty TCU, a set of probability is computed to model fault behaviour.

$$\begin{aligned} &P(x',y') = \frac{\sum_x \sum_y Propagate(x,y,x',y')}{\#CorruptedCTA} \\ &Propagate(x,y,x',y') = \begin{cases} 1 : x' == (x\%M_s) \wedge y' == (y\%N_s) \\ 0 : otherwise \end{cases} \\ &\forall x,y : g(x,y) \neq f(x,y) \\ &\forall x' \in [0, M_s), \forall y' \in [0, N_s) \end{aligned} \tag{3.3}$$

As expected, independently to studied fault, the majority of coordinates (x', y') are associated to null propagation probability. In general, up to two coordinates are associated to a corruption probability of about 90% and standard deviation about 20%.

To conclude, we proposed to determine the elements of output tensor effected by each fault through:

i) Performing Tiling to slice output tensor in blocks $N_s \times M_s$

ii) Call scheduler behavioral algorithm to determine CTAs allocation

iii) Determine absolute coordinates of corrupted elements through corruption probabilities and scheduler allocation.

The proposed spatial propagation modelling procedure is flexible with respect to input tensors dimensions. Notably, CNNs inference is calculated through several layers with different number of parameters. Therefore, the number of CTAs to schedule and compute is a peculiarity of each convolution layer. Modeling spatial propagation at CTA level device an injection methodology feasible for any CNNs.

Data Corruption

Aforementioned, the goal of this FIC is to emulate fault behaviour through *bit flip masks.* This error modelling strategy, provides both a flexible procedure to interpolate experimental results and a relatively low computation cost injection procedure. Therefore, in order to asses and reproduce the scalar effects introduced by every fault, *bit flip* probabilities have been calculated for each bit in *IEEE* 754 float 16 standard (see eq. 3.4).

$$\begin{gathered} P_b = \frac{\sum_{x,y} Flipped(x,y,b)}{\sum_{x,y}\sum_{i=0}^{n} Flipped(x,y,i)} \\ Flipped(x,y,i) = ([g(x,y) \oplus f(x,y)] \wedge 2^i) >> i \\ \forall x,y : g(x,y) \neq f(x,y), \forall b \in [0,n) \end{gathered} \tag{3.4}$$

Flipped function receives as input a bit position (i) and a corrupted coordinate (x,y). It compares hexadecimal representation of golden and faulty elements through XOR operator in order to underline corrupted *bits.* Subsequently it inspects if input bit (i) has been injected. Exploiting this function recursively across all injected coordinates and bit positions, it is possible to asses which bits are more likely to be effected by fault under examination. To leverage on *bit flip probabilities* to device corruption masks, it is mandatory to design a mask generation algorithm aware of numbers parallelism. In fact, while for integer numbers *bit flip probabilities* alone can correctly replicate faults scalar effects, in floating point 16 numbers, bits are clustered in *Sign*, *Exponent* and *Mantissa.* Hence additional metrics are necessary. As expected, while critical faults have a *bit flip* probability distribution shifted towards exponent bits, not observed faults have a probability distribution shifted towards least significant bits of mantissa.

Nevertheless, it is imperative to distinguish between exponent corruption caused by *"stuck at"* faults in exponent bits with respect to exponent corruption caused by normalization of MAC (see Ap. A.0.2) operations processing corrupted mantissas. In fact, while in former case the fault generate huge mean absolute error, the latter compensate the corruption of exponent through several bit flips in mantissa. Therefore, without distinguish between these two corner cases the error introduced by fault might be overestimated. Probability of exponent corruption (PEC) is calculated as ratio between faulty elements with at least one bit flip in the exponent over the total number of elements effected by the fault (see eq. 3.5). Through this probability it is possible to easily identify the corner cases illustrated above since while for the former PEC is high, the latter is characterized by low corruption probability. In fact, normalization procedure strongly depends on input data.

$$\begin{aligned} &PEC = \frac{\sum_x \sum_y ExpCorruption(x,y)}{\#corruptedentrances} \\ &ExpCorruption(x,y) = \begin{cases} 1 : ((f(x,y) \oplus g(x,y)) \wedge 0x7C00) \neq 0 \\ 0 : otherwise \end{cases} \\ &\forall x,y : f(x,y) \neq g(x,y) \end{aligned} \tag{3.5}$$

Additional information relevant to Mask generation algorithm (see Sec. 3.2) are extrapolated from experimental data for each fault :

- Bits that are always flipping: as the logic AND of *bit flip* masks of all corrupted coordinates (see eq. 3.6)
- Average number of bit flips in Mantissa when the exponent is not corrupted (AB_fME_g)
- Average number of bit flips in the exponent (AB_fE)
- Average number of bit flips in Mantissa if exponent is corrupted (AB_fME_c)

$$\begin{cases} \prod(g(x,y) \oplus f(x,y)) \\ \forall x,y : g(x,y) \neq f(x,y) \end{cases} \tag{3.6}$$

3.2 Error Model

3.2.1 Model Generation

Exploiting all information obtained during FIC results analysis, a methodology has been hypothesized to replicate faults behaviour through *bit flip* masks. The error models (*Effective Application-level Error Modeling of Permanent Faults on AI Accelerators* publication proposal) consist of *bit flip masks* spatially distributed in the output tensor. Pairs of coordinates to inject and masks to apply directly on golden values, are associated to each fault to effectively reproduce faults effects. The coordinates are expressed in the CTA coordinate system (see sec. 3.1.2). The CTA coordinates are mapped back into absolute coordinates according to scheduler allocation which in turn depends on target GPU structure. Therefore, for each GEMM, the scheduler behavioural algorithm is called and data corruption will only occur on elements whose computation is scheduled to faulty hardware. The masks are generated following the flow illustrated in Fig. 3.2.

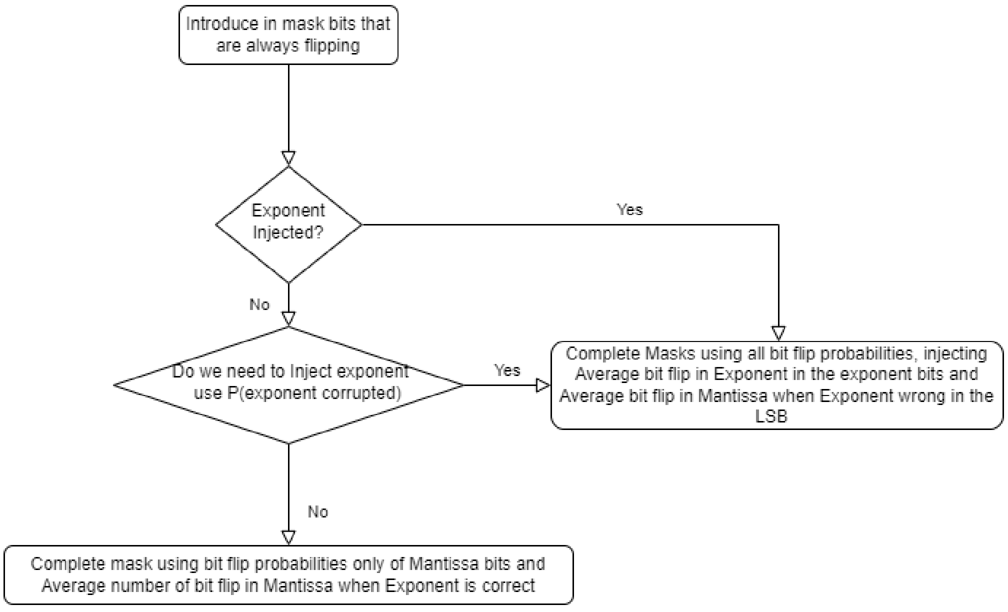
Introduce in mask bits that are always flipping
Exponent Injected?
Yes
No
Do we need to Inject exponent use P(exponent corrupted)
Yes
Complete Masks using all bit flip probabilities, injecting Average bit flip in Exponent in the exponent bits and Average bit flip in Mantissa when Exponent wrong in the LSB
No
Complete mask using bit flip probabilities only of Mantissa bits and Average number of bit flip in Mantissa when Exponent is correct

Figure 3.2: Flow Chart describing masks generation algorithm

First of all, *bits that are always flipping* are introduced in the mask. Then if exponent has been already injected or the fault is characterized by high PEC the masks is "completed" exploiting both average bit flips in exponent and average bit flips in mantissa. Otherwise, the masks is completed only using average bit flip in mantissa . According to averages rounded values, bits are introduced inside mask by randomly generating their position using *bit flip probabilities.* The averages are rounded up or down according to outcome from random event generator with probabilities proportional to fractional magnitude (see Tab. 3.2).

In order to reduce simulation complexity, error models have been generated only for *Jetson AGX32* supporting *Two Level Round Robin* scheduler. Unfortunately, although spatial propagation has been generalized and can be exploited to inject any CNN, we hypothesized that masks faults replication strongly relies on data. Therefore a *refinement procedure* is performed to isolate the masks able to correctly

Algorithm 1 Error model generation and refinement algorithm.

```
Input:$MxM_{faultfree}$, $MxM_{faulty}$, $faultIDs$, $P_b$,$PEC$,$AB_fME_g$,$AB_fE$,$AB_fME_c$
Output:$errormodels$
 1: function MaskGen($P_b$,$PEC$,$AB_fME_g$,$AB_fE$,$AB_fME_c$)
 2:     if random($PEC$) then
 3:         $mask = random(P_b, AB_fE, AB_fME_c)$
 4:      else
 5:         $mask = random(P_b, AB_fME_g)$
 6:     end if
 7:     return $mask$
 8: end function
 1: $errormodels = []$
 2: for all $faultID$ in $faultIDs$ do
 3:     $MxM_{mask} = []$
 4:     for all $element$ in $MxM_{faultfree}$ do
 5:         $mask$ =MaskGen($P_b$,$PEC$,$AB_fME_g$,$AB_fE$,$AB_fME_c$ )
 6:         $element_m = element \oplus mask$
 7:         $MxM_{mask} \leftarrow element_m$
 8:     end for
 9:     $MAE_{mask} = MAE(MxM_{mask}, MxM_{faultfree})$
10:     $MAE_{faulty} = MAE(MxM_faulty, MxM_{faultfree})$
11:     if $1/Th < MAE_{faulty}/MAE_{mask} < Th$ then
12:         $errormodels_{refined} \leftarrow [mask, element_{pos}]$
13:     end if
14: end for
15: return $errormodels$
```

Fractional magnitude	Probability Rounding Up	Probability Rounding Down
< 0.2	0%	100 %
< 0.4	25%	75%
< 0.6	50 %	50 %
< 0.8	75%	25%
>= 0.8	100 %	0 %

Table 3.2: Rounding procedure of averages to complete masks

replicate faults behavior. The error models generation and refinement procedure relies on an iterative process (see Alg. 1). As a first step, for every fault, pairs of coordinates and masks are generated through data analysis performed on gathered results of a first FIC. A second fault injection campaign is performed randomly generating seed GEMM for each fault. The MAEs associated to each fault are stored (MAE_{faulty}). At this stage, golden values are corrupted through *bit flip*

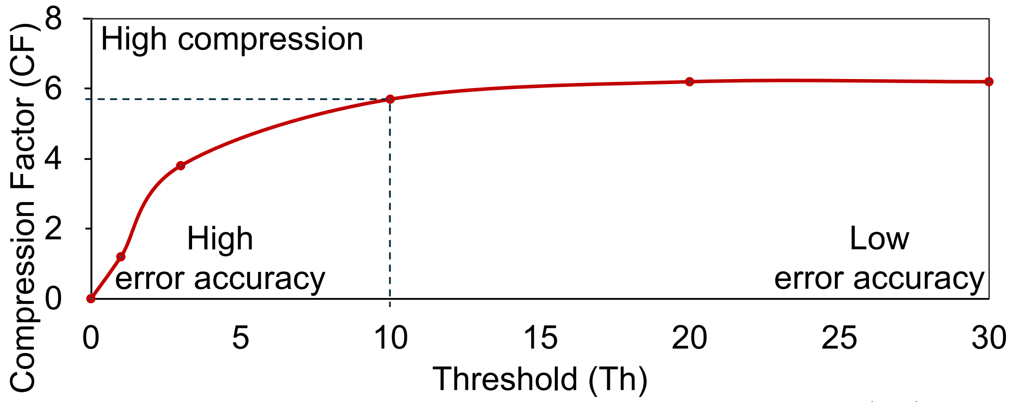

Figure 3.3: Experimental relation between *Clustering Threshold* (Th) and *Compression Factor* (CF) for the shape-wise error model from the 100X GEMM.

masks are their effects are evaluated (MAE_{mask}). Through comparing different injection methodologies effects (see Alg. 1, line 11), error models are refined to isolate masks able to properly reproduce faults scalar effects. Verified masks are characterized by good replication accuracy through ensuring a MAE with same order of magnitude with respect to standard injection procedures. The verification threshold can be tuned according to required replication accuracy (see Tab. 3.3). According to verification threshold only a portion of the *bit flip masks* were able to pass refinement stage. Nevertheless, up to 66% of faults have been modeled with high accuracy (Th to 3). Furthermore, by increasing the verification threshold to 10, 93% of studied faults have been modeled.

Th	1	3	5	10	20
Bit-flip-masks (%)	8.96	66.37	81.25	93.0	100.0

Table 3.3: Refined bit flip masks according to verification threshold

"Unmodellable" faults injected both high and low absolute error on faulty elements. In fact, the introduced minimum absolute error was at least two order of magnitude lower with respect to maximum absolute error, complicating faults behaviour replication. Refined error models are clustered in order to further improve computation complexity required by fault injection campaign at the application level. Error models are grouped according to faults spatial propagation and scalar effects. In particular, faults collapsed on the same cluster are characterized by identical spatial propagation and MAEs respecting a clustering threshold (we imposed MAEs with same order of magnitude). The clustering threshold can be tuned to trade-off error accuracy with compression factor (see Fig. 3.3).

Nevertheless, at this stage, seed GEMMs are still generated with fixed tensor dimensions. We supposed that masks replication accuracy is enhanced if generated through fault injection campaign results obtained starting from a seed GEMMs with shapes similar to GEMM to inject. In particular, according to GEMM size and tiling, the number of sub-matrix multiply and accumulate operations per tile changes. Since these operations are recursively computed by the same faulty TCU, we supposed a strong correlation between the numeric effects of faults and the GEMM shape. Therefore, size aware error models need to be generated for each fault to correctly inject CNNs. Multiple FICs have been performed with different set of input tensors (see Ap. A.0.3). For each golden value set, faults are modeled into error models. Furthermore, error models undergo refinement and clustering procedures.

3.2.2 Model Validation

Several test layers have been exploited to evaluate replication effectiveness of different error models when varying GEMM tensor shapes (see Tab. 3.4).

CNN Layer	Inputs Shape	$Mac/entrance$
LeNet5	6x75x576	75
ResNet18 RB1	64x147x12544	147
ResNet18 RB2	64x576x12544	576
YoloV5	32x27x102400	27
MobileNetV2	3x3x12544	9

Table 3.4: Validation convolution layers. Shapes are expressed in AxBxC where B is the common dimension of GEMM input tensors. RB stands for *residual block*.

For each validation layer, a fault injection campaign is performed using *Jetson AGX32* emulator injecting only one fault per cluster. The MAEs are calculated for each fault for validation (see Ap. A.0.3). We propose to rely on vector cross-correlation coefficient to evaluate replication accuracy of masks generated from different golden values. This statistical indicator is calculated exploiting MAEs of faults injected both through masks and classical reliability procedures. A high level of cross correlation indicates a stronger replication of faults through a set of bit flip masks for that particular test layer. Comparing several levels of cross correlation associated to the same test layer underlines the set of bit flip masks with stronger replication accuracy. Surprisingly, the supposed relationship between replication accuracy of masks and GEMM shape has not been fully proved by experimental data. In fact, according to set of golden values (see Ap. A.0.3) exploited for error models generation and test layers shapes (see Tab. 3.4), masks obtained through 100x100x100 golden GEMM should have been enhancing faults behaviour replication for *LeNet5*, *Yolov5*, *MobileNetV2* and *ResNet18 Residual*

Block 1 convolutions. On the other hand, due to similar MAC operations per entrance, masks obtained from 200x200x200 were supposed to optimize faults behaviour recreation in *ResNet18 Residual Block 2.* Masks from 300x800x300 and 400x1200x400 golden GEMMs should have been characterized by low cross correlation regardless to the test layer.

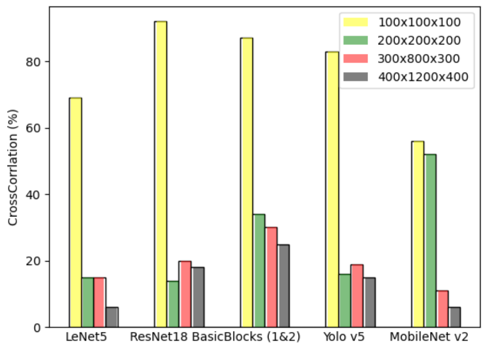

Figure 3.4: Different Levels of Cross Correlations for every test layer according to masks generated from different set of golden values

As shown in the histogram above (see Fig. 3.4), the *bit flip masks* obtained by 100x100x100 GEMM, enhanced faults numeric effect replication for every test layer. The histogram above is characterized by clusters of bars, associated to a test layer. Each bar in the same cluster, represents a cross correlation associated to a particular group of error models extrapolated starting from a seed GEMM. From left to right we have respectively error models devised through FIC using: 100x100x100, 200x200x200, 300x800x300 and 400x1200x400 input tensors. As expected, cross correlations between errors introduced from masks extrapolated though 300x800x300 and 400x1200x400 GEMMs and fault injection campaigns is relatively low for every test layer. Surprisingly, faults replication of 200x200x200 GEMM error models, is poor for almost every test layer. In fact, this set of *bit flip masks* is characterized by a cross correlation larger than 50% only when tested against *MobileNetV2* convolution. To conclude, cross correlations have been calculated only taking into consideration masks injecting a MAE with at least two order of magnitude similarity with respect to fault injection (e.g $0.01 < MAE_{mask}/MAE_{fault} < 100$). Otherwise, the error model is discarded, reducing maximum reproducible fault coverage for fault injection campaign at the application level. According to test layer, the amount of error models failing this test was between 5-11 %. The failing rate is almost a peculiarity of validation layer and seems to not depend on the error models seed GEMM shape (see Tab. 3.5)

To conclude, error models generated and refined from 100x100x100 seed GEMM

Failing Rate	Golden GEMM	Test Layer
5.6 %	100x100x100	*LeNet5*
5.54 %	200x200x200	*LeNet5*
10.81 %	100x100x100	*ResNet18 BB1*
10.17 %	200x200x200	*ResNet18 BB1*
9.84 %	300x800x300	*ResNet18 BB1*

Table 3.5: Failing rate levels

FIC replicate fault behaviour with decent levels of cross correlation against all validation layers. On the other hand, these error models are characterized by slightly higher failing rate. Overall, their performances in terms of replication effectiveness enabled, fault injection campaign at the application level. Therefore,for this particular golden values set, from the 8,600 injected faults, 93.17 % has been modelled and refined, into 1330 clusters (error models). Testing obtained error models against *LeNet5* validation layer , 91% of studied faults are also reproducible through masks with decent level of emulation robustness against input tensor dimensions. This enabled the possibility to perform a fault injection campaign at the application level with a fault coverage comprising 7905 faults. On the other hand, we strongly underline that results obtained in following chapters are characterized by uncertainty.

Chapter 4

Fault Detection

4.1 The Problem

Fault injection campaign at the application level relying on an accurate description (*gate* level) of underling hardware require sensitive time. For example, 10,000 days are necessary to inject only 1,000 faults in *LeNet5* simulated with the *FlexGripPlus* GPU model[18]. Furthermore, the simulation complexity remains a substantial issue regardless to the hardware description level. In fact, about 4 days are necessary for SFI campaign on *LeNet5* corrupting one inference per fault through high-level description of *Jetson AGX32*. Currently, Fault Injection (FI) campaigns are often conducted at the application level, to assess the resilience of DNNs with respect to hardware defects, through corrupting the *synapses* (i.e., random bit-flips on the weights) or the *neurons* (i.e., random bit-flips on the feature maps) [13]. However, such FI approaches are hardware-agnostic, meaning that they do not take into account the underlying hardware. Any proposed detection technique based on the CNN response either does no consider permanent faults [12] or are hardware-agnostic [13]. Furthermore, none of the available detection procedures are both suitable for in-field utilization and application-aware. For example, SBST devised for TCU [3], despite providing outstanding fault coverage, is application-agnostic, meaning does not consider the application behavior for detection but only rely on the target hardware structure. Such approach is hardware-specific and not flexible w.r.t. design modifications required by new generation platforms. In addition, being application-agnostic, it requires to block the application itself to conduct reliability analysis, degrading performances. Furthermore, a specific SBST needs to be designed for each computational unit, increasing integrity check analysis time for complex hardware platforms. In this work we propose an application-aware fault detection technique able to provide good fault coverage without reducing application performances. We propose to monitor the application behavior and associate unexpected responses to damages in the underlying hardware. Leveraging on the application misbehavior, a flexible detection procedure can be devised able to monitor multiple hardware resources (i.e, planar data processor, TCU) in parallel. Nevertheless, in order to evaluate the proposed fault detection procedure, it is mandatory to monitor the application response both when varying the inputs and the faults. For example, in computer vision tasks, leveraging on CNNs, the behavior of hidden layer neurons strongly depends on the input video. This huge *injection space* requires sensitive simulation time and clever investigation techniques. In this regard, through Error Modeling we proposed an alternative **hardware-aware** injection methodology that despite losing accuracy w.r.t. traditional FI procedures, speeds up execution by 225X. In fact, while about 8h are required to conduct a SFI campaign with 100x100x100 seed GEMM through *pyOpenTCU*, error models can be injected in 2,26 min. Therefore, few days long simulations compute multiple inferences (about 250) of *LeNet5* for each studied fault. This enabled the possibility to explore with a decent level of freedom the injection space and propose reliable software-based detection methodologies that can trace back the cause (i.e. application performances degradation) to the effect (i.e. error model). Therefore, the utilization of SBST can be limited to either booting procedures or unexpected application behavior, improving application performances. Followig our approach, safety standard protocols can be respected without drastically reducing application performances.

The results of this chapter have been obtain injecting *Jetson AGX 32* error models. In particular, the masks are generated and refined starting from a 120x120x120 seed GEMM (14,400 MAC per entrance). Furthermore, a cross correlation coefficient of 75% has been achieved w.r.t. classical injection methodologies while validating against *LeNet5* convolution layers.

4.2 Binary Decision Tree as a Fault Detector

Error models enabled fault injection emulation at DNN level. Several inferences can be computed and corrupted to determine which faults of TCU if activated can generate failure at the application level. As a study case, we focused our effort on designing software based detection technique to identify with high accuracy the faults degrading LeNet5 performance in terms of *classification accuracy loss*. Leveraging on *PyTorch* libraries, it has been possible to customize convolution and fully connected layers. In fact, exploiting *imm2col* algorithm, input tensors are reshaped, multiplied, corrupted and reconstructed. According to scheduler allocation, hidden layers neurons are injected with error models. A binary classifier provide versatile algorithm able to generate decent level of detection accuracy with moderate computation cost. These characteristics made it a very suitable candidate for detection since its execution can be performed in parallel with application without increasing drastically computation latency. Nevertheless, a data set generation methodology is mandatory to train it. We propose to exploit statistical information about DNN hidden layers activations to establish weather or not the application is being executed by faulty hardware (F/NF). In particular, according to classification output, several information about the layer under examination (l) have been extracted such as sum of activations, number of weak activations, maximum activation position and value and more (see Tab. 4.1). These information have been collected across several inferences (250 for each error model) both in fault free scenarios and injecting error models.

Weak	Sum	Std	PSNR	Max	MaxPos(A)	Class(o)	F/NF
$a_l < 0.05$	$\sum a_l$	$\sigma(a_l)$	(see eq.4.1)	$\max(a_l)$	c,w,h	[0,10)	1/0

Table 4.1: Fault detector data set features. The leftmost features are exploited by binary decision tree to generate F/NF as a prediction that is compared against the corresponding value in dataset to estimate detection accuracy.

$$\begin{gathered} MA_o(c,w,h) = \textstyle\sum_{i=0}^{N} a_l(c,w,h)/N \\ \forall o \in MNIST_{class} \\ PSNR(A_l, MA, o) = -10\log_{10}\left(\frac{\sum_{c,w,h}^{C,W,H}(A_l(c,w,h) - MA_o(c,w,h))^2)}{C*W*H}\right) \end{gathered} \tag{4.1}$$

The *peak signal to noise ratio* (PSNR) feature has been calculated between hidden layers activations (A_l) and *"mean activation"* tensor (MA). For each possible classification output (o), a MA is computed running N *fault free* inferences and calculating the average activation of each neuron across them (see eq. 4.1). During data set features calculation, according to predicted output, eventually different to CNN *fault free* classification, PSNR is calculated to quantify distortion with respect to DNN average behavior [19]. During in filed utilization, the presented detection methodology will generate a prediction with a computation complexity of $\mathcal{O}(C \times W \times H)$. In fact, assuming single image batches, and a system with enough memory to store all *mean activation* tensors, the features required for prediction can be computed in parallel. Nevertheless, in order to find optimal position in DNN in terms of detection accuracy, for each hidden layer a new data set must be generated increasing simulation time consistently especially for deep CNNs. In addition to features listed in Tab. 4.1, each vector of data set is associated to the error model injected while computing it to trace back the effect to the cause. The *fault free* inferences have been associated to an error model wit empty *bit flip* masks. Subsequently, error models have been clustered, using K-means unsupervised learning algorithm, according to application performances degradation. Application degradation has been measured through *classification accuracy loss*, as the discrepancy between LeNet5 fault free classification accuracy and classification precision while injecting the errors. The overall idea is to partition the data set according to faults criticality at application level. Nevertheless, the majority of faults in fault coverage ($\sim$ 90%) tend to introduce low to none classification accuracy loss ($< 5\%$, see Fig. 4.1)

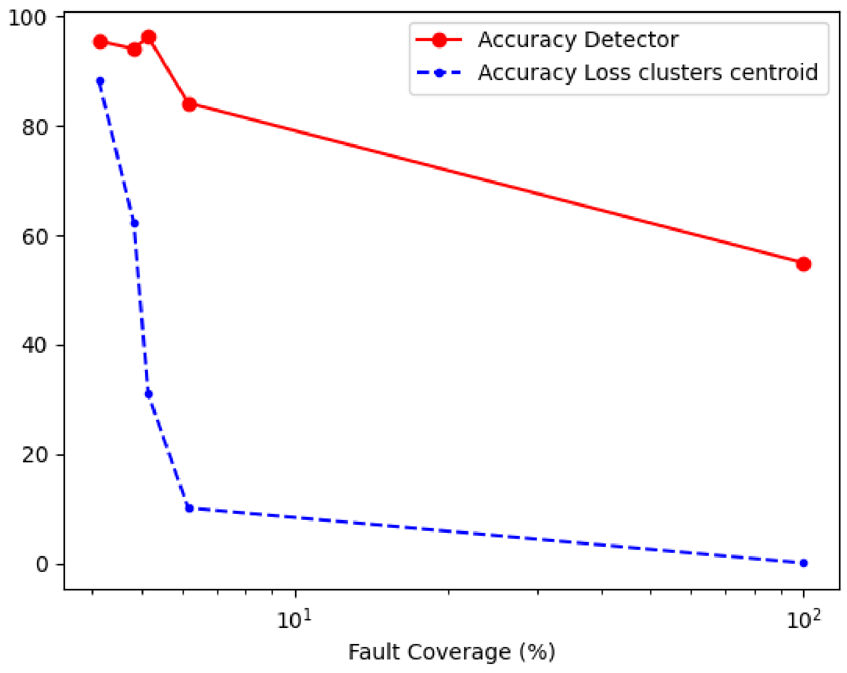

Figure 4.1: Detection accuracy of fault detector across different error models clustered according to classification accuracy loss

Nevertheless, as illustrate in Fig. 4.1, the binary decision tree, trained through a data set comprising only features of application critical faults, have a detection accuracy larger than 80%. The detection accuracy is obtained locating the binary classifier at output of second convolution layer of LeNet5. In addition, considering *masked faults* [12], [20] contribution in training data set, decreases detection accuracy down to 54%. In fact, faults injecting low *mean absolute error*, tend to have neurons activation statistics very similar to fault free scenario. Although always balancing training data set between fault free and faulty scenarios, when considering also *masked faults* in detection, number of false positive increased drastically during validation. In fact, although overall detection accuracy is about

54%, the binary classifier was able to identify 78% of fault coverage. Therefore, a low level of classification precision is caused by a large number of false positives.

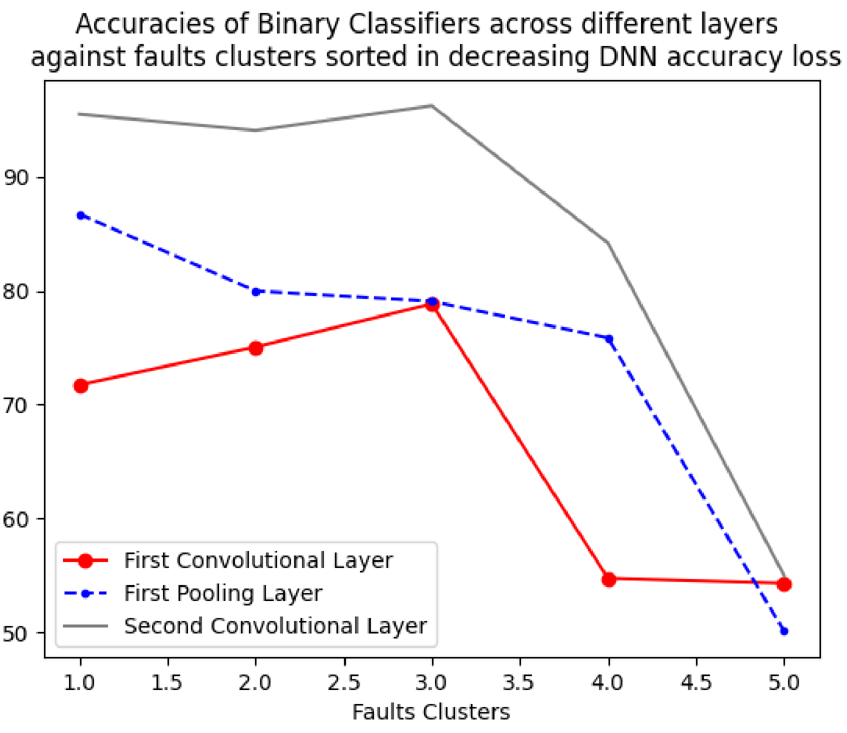

Figure 4.2: Detection Accuracy of Fault Detector across different layers of LeNet5

Repeating the presented detection methodology across 3 different layers of LeNet5 provided us the optimal position for the binary classifier in terms of detection accuracy (see Fig. 4.2). In fact, the activations at the output of second convolution layer seems to enhance detection accuracy with respect to other layers neurons. This suggested that positioning the classifier deep in CNNs improves classification outcome. Therefore, neurons activation discrepancy with respect to *fault free* inferences is proportional to number of faulty convolutions. In fact, convolution layers are recursively computed by same faulty GPU, hence at each iteration new neurons will be injected and already corrupted neurons will propagate, generating an *avalanche* effect. Nevertheless, pooling layers reduces tensor dimensions, decreasing the number of exploitable neurons for detection. Therefore, locating fault detector too close to fully connected layers reduces its detection performances.

Entropy-based triggering procedure has been hypothesized to enable fault detection. Through training a triggering threshold when classification was uncertain, application performances can be further boosted by calling detection procedures only when strictly necessary. This methodology has been already exploited in *Deep-One* Classifiers [21]. According to their work, it was possible to train a CNN to map an input image into a *n-th* dimensional circle. If an input image was mapped inside that circle it meant that DNN was trained with similar images, otherwise the input was an *anomaly*. Furthermore, they suggest to trigger this classifier every single time in-field application output was not certain (*classification entropy*). Nevertheless, we obtained a very optimized *classification entropy* (always 0.0) both during fault free inferences and injecting our error models. In fact, data set simplicity (MNIST) and very high fault free accuracy ($\sim$ 99%) might have caused certain misclassifications. Therefore, it is has not been possible in our study case to exploit *Entropy* of neurons at *SoftMax* layer output to trigger fault detection. On the other hand, we propose to exploit a polling methodology to periodically trigger fault detector to ensure hardware integrity.

Chapter 5

Hardening Techniques

5.0.1 Motivation

Although faults can compromise application performances, suitable countermeasures have been already proposed to avoid catastrophic consequences for in field applications [22]. Some hardware-software solutions have been implemented to guarantee tasks execution despite faults [23]. Nevertheless, the majority of fault protection procedures still strongly rely on modification of hardware description and are application-agnostic (i.e triple modular redundancy [24]). Software-based solutions always proved to be the most preferred approach from companies to ensure reliability. In this chapter we propose an application-aware hardening solution based on pooling layers computation. The proposed solution is suitable to mitigate faults in TCU executing DNN applications. Nevertheless, despite being a software-based solution, its computation is not optimized through specialized libraries or hardware accelerator in current GPU versions.The proposed solution, although improving resilience of applications executed by damaged TCUs, it can reduce GPU performances. Therefore, we devised a customized version of an accelerator already embedded in multiple GPU devices, able to support the proposed layer computation with identical latency of traditional pooling layers.

5.1 Pooling Techniques for Hardening *LENET5* Execution by NVIDIA GeForce GTX 1500 Ti

The idea behind this experiment set up is to study the overall distortion introduced by hardware faults in Tensor core units, while propagating their effect through different layers. While TCUs oversee the execution of dot products in convolutions, activation layers and pooling layers are calculated by different hardware resources in GPUs (i.e, planar data processor), hence they could be exploited to mitigate errors.

5.1.1 Experiment Set Up

CNN Structure

In order to obtain reliable numeric results, the golden values of convolution have been generated starting from inputs and weights feeding the first convolution layer of LetNet5 when an image from MNIST data set classified as numeric 0 is processed. Nevertheless, after convolution layer computation is completed, ReLu activation has been replaced by a Normalization Layer and a Tanh activation. This network design was able to calculate mean square errors necessary to obtain Peak Signal to Noise Ratio (PSNR) even when faults were corrupting tensor elements to float16 infinite (or nan) leveraging on Tanh layer. Furthermore, a by channel normalization layer was crucial due to gradient vanishing problems introduced by hyperbolic tangent. Therefore, a matrix multiplication of T1 (size 6 x 25) by T2 (size 25 x 784) is calculated to perform Convolution, the output tensor is reconstructed T3(6 x 28 x 28), a by channel normalization is performed(T4), followed by tanh activation(T5) and a pooling layer (T6). The channel normalization shrinks data in the range between 0 and 1 (see eq. 5.1)

$$\begin{cases} T_o[c,w,h] = (T_i[c,w,h] - min(T_g[c,:,:])) \div (max(T_g[c,:,:]) - min(T_g[c,:,:])) \\ \forall c,w,h \in T_i \end{cases} \tag{5.1}$$

Minimum and maximum values required for normalization in each *channel (c)*, are selected from golden values. Three different pooling layers are fed by same tensor produced after activation layer and PSNR is calculated w.r.t fault free scenario to study which between Average, Max and Median pooling was generating lowest distortion.

GPU Architecture

The GPU simulator has been configured to perform reliability study over *NVIDIA GeForce GTX 1500 Ti* platform. The studied GPU structure comprises 2 clusters each containing 5 Streaming multiprocessors (SM), each handling 2 CTA in the buffer during scheduling. In order to provide an initial estimation of which *pooling layer* compensates TCU permanent faults effects, classical fault injection campaign has been conducted leveraging on *pyOpenTCU*. In fact, due to limited number of CTAs in LeNet5 convolution and propagating faults through few CNN layers, the simulation time required by classical fault injection campaign was lower than

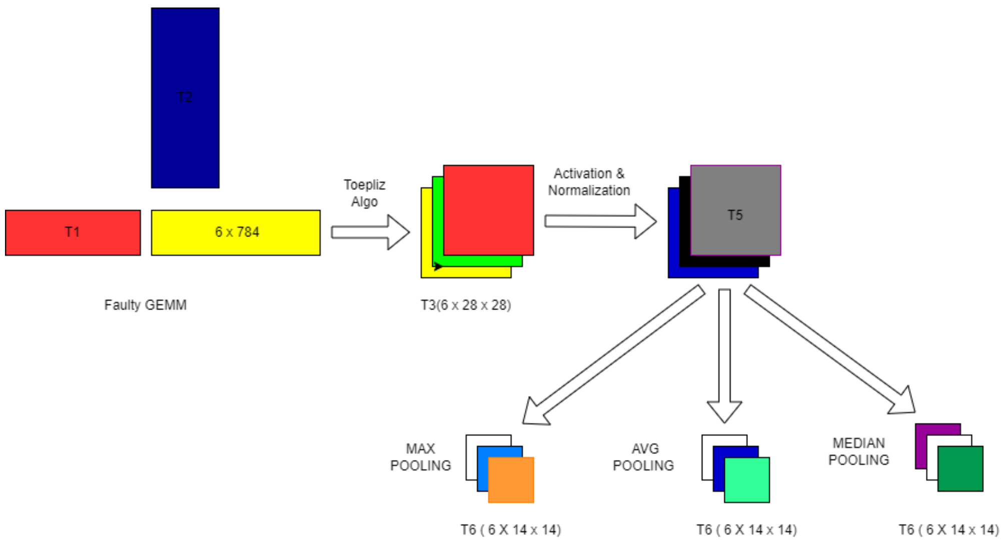

Figure 5.1: TCU faults propagation across different layers

few days. Furthermore, only application critical faults have been injected during this evaluation to further mitigate computation complexity (about 10% of studied faults of previous Chapters).

5.1.2 Pooling Size = 2

The tensor generated by faulty convolution propagates in the network described previously and feeds different pooling layers, respectively Median Average and Max, with kernel size and stride equal to 2 and no zero padding. Peak signal to noise ratio is calculated against golden values to measure distortion with respect to fault free scenario. When the pooling filters were mitigating fault effect completely, the *PSNR* was saturated to 150 dB for the sake of representation.

Cutting the graphs (Fig. 5.2) vertically 3 intersections (counting intersecting with x-axis) are generated, representing values of measured distortion at output of pooling layers while propagating the same fault. The *PSNRs* have been sorted according to distortion introduced by Max Pooling by hypothesizing a lower resilience with respect to other studied pooling algorithms. Independently with respect to scheduling policy, it can be established that the Median filter mitigates better the effect of permanent faults with respect to Average and Max pooling layers. The stronger numeric stability is indicated by a lower level of distortion with respect to fault free scenario measured through an higher *PSNR* with respect to Average Pooling. Furthermore, sometimes Median Pooling removes the effect of the fault completely, as shown by pikes to 150 db. According to studied scheduling polices, Median Pooling provides better performances with respect to Average Pooling in

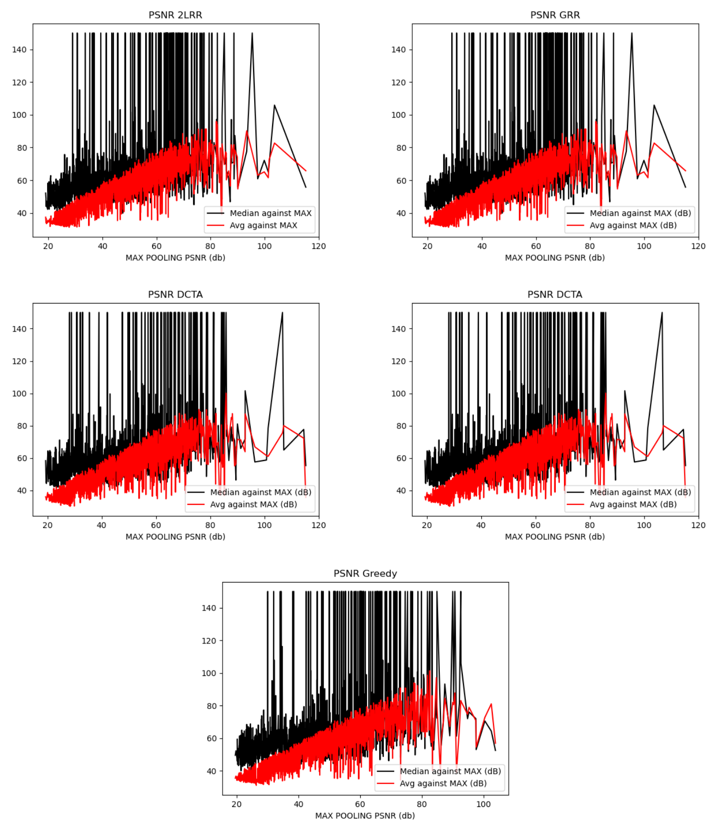

Figure 5.2: Peak Signal to Noise Ratios for each Pooling technique with *GEMM* performed with different schedulers

the range between 65 % to 70 % of injected faults (Tab. 5.1). Furthermore, data suggests that *Greedy* scheduling policy provides better performances in term of *reliability* with respect to other policies.

SCHEDULING POLICY	*MED VS AVG*	*AVERGE PSNR MED (dB)*
Two-Level-Round Robin	67.85 %	57.70166764680866
Global Round Robin	67.85 %	57.70166764680866
Greedy	71.3 %	60.5917261645620
Distributed CTA	68.95 %	59.540405686236554
Distributed Block	66.6 %	59.42639558354495

Table 5.1: Evaluating Med Pooling performances against Avg Pooling for each fault, pooling size = 2

Greedy, exploited together with Median Pooling might be the best solution in order to amortize faults effect when pooling size is small. In fact due to decent ICL, this scheduling policy is characterized by the best balance between number of pooling filters comprising at least one faulty element and number of filters having more.

5.1.3 Pooling Size = 3

Data seems to suggest that increasing kernel size improves the performance of Median algorithm. In fact, by increasing the number of activation feeding the filter, the probability of compromising the median value is lower. Furthermore, Distributed CTA and Distributed block now have a better average distortion than Greedy. Global Round Robin and Two-Level Round Robin seem to have worst performances with respect to remaining scheduling policies regardless to pooling size.

SCHEDULING POLICY	*MED VS AVG*	*AVERGE PSNR MED (dB)*
Two-Level-Round Robin	71.45 %	65.60639909447866
Global Round Robin	71.45 %	65.60639909447866
Greedy	77.85 %	70.86309969900103
Distributed CTA	78.4 %	73.42546600567651
Distributed Block	72.21 %	72.2155914689443

Table 5.2: Evaluating Med Pooling performances against Avg Pooling for each fault pooling size = 3

5.2 LeNet5 Injected Though Error Models

Although providing an initial evaluation of resilience with respect to TCU permanent faults of several pooling filters, the results illustrated in section above must be supported by data obtained from a fault injection campaign at the application level. Nevertheless, to mitigate simulation time, faults have been injected though error models instead of TCU emulator. Therefore, the data illustrated in this section are subjected to uncertainty introduced by the injection model. In particular, exploiting the *bit flip* masks, generated through multiple fault injection campaigns performed using *Jetson AGX 32* emulator, several LeNet5 CNNs performances have been evaluated while corrupting hidden layers neurons. In fact, three customized CNNs have been studied, each exploiting a different filter in pooling layers (e.g. Avg, Max, Med). The models have been trained using MNIST images though 12 epochs. Fault free classification accuracy has been measured for each one of trained CNN exploiting a validation data set (see Tab. 5.3).

F/NF	AVG LeNet5	Max LeNet5	Median LeNet5
Fault Free	97.8 %	98.6 %	97.7 %
Faulty	82.88 %	83.48 %	95.99 %

Table 5.3: Filters performances

Although the CNNs are characterized by similar fault free detection accuracy, the performances change drastically when injecting error models. In fact, for each inference of validation data set, an error model, associated to a group of permanent faults, has been injected. Comparing the *faulty predictons* against image labels, a new classification accuracy has been measured. Classification accuracy of CNN trained exploiting median filters seems almost unchanged during fault injection campaign. Therefore, its numeric stability and resilience can be firmly established. Nevertheless, median filters require a sorting algorithm whose computation complexity ($\mathcal{O}(nlog(n))$ is larger than average and max filters. Therefore, in order to implement on silicon this sophisticated filter, the kernel sizes must be bounded (at the most 3x3). To conclude as a future reference, for applications exploiting pooling layers with filters size larger than 3x3, we suggest to implement a MIN/MAX filtering procedure. Median pooling algorithm numeric stability stems form a pooling procedure that propagates only one neuron for each kernel. Having the loss function dependent only on one activation for each pooling kernel and avoiding selecting maximum value, ensures strong numeric stability . Therefore, performing min operations between neurons on the same row of pooling filter and subsequently finding the maximum across the obtained minimums, will ensure the stability condition mentioned previously. Nevertheless, the numeric stability of such filtering algorithm has not been evaluated rigorously in this work but only hypothesized.

5.3 MEDIANVDLA

Pooling layers are executed by the *Planar Data Processor* (PDP) in NVDLA SoC through at most 2 pipeline stages. In fact, while fusion mode only requires one stage (PDP), in independent mode an additional pipeline stage (PDP RDMA) is necessary to write DRAM. Through CSB interface an external CPU can modify the embedded address spaces of PDP unit. According to the content of the *POINTER* register, the write operations from CPU are converged either to **consumer** or **producer** address space. By default, CSB inputs modify consumer address space. Once the operation is enabled, consumer address space configuration drives control signals in PDP core data path. Simultaneously, POINTER register is modified by hardware to converge CSB signals to producer address space. Therefore, the external CPU can start configuring next operation control signals without waiting for past operations to ultimate, hence optimizing configuration latency. Once the CPU completes the write operations required by future layer execution, a polling strategy is exploited to monitor the status register. As soon as the data path completes the execution of a pooling layer, the status register is updated and if additional operations have been scheduled, consumer and producer address spaces are interchanged. The implemented configuration protocol can be referred as *ping-pong* synchronization.

5.3.1 Address Space

In this section the address space of PDP is illustrated as well as the modifications required to implement median pooling.

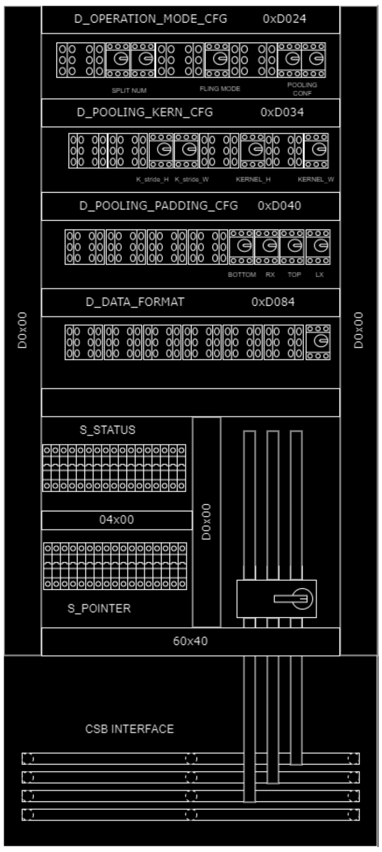

Figure 5.3: Address space of PDP

D_OPERATION_MODE_CFG 0xD024

- Pooling Method[1 : 0] :
  - i) "00" : Avg pooling
  - ii) "01" : Max pooling
  - iii) "10" : Min pooling
  - iv) "11" : Med pooling
- Unused[3 : 2] : zero hardwired
- fling mode[4]
- Unused[8 : 6] : zero hardwired
- Split Num[16 : 9]
- Unused [31 : 17] : zero hardwired

Pooling Method bits can now support a new configuration ("11") to compute Median Pooling. Through the introduced configuration, the output of median pooling core can propagate either to DBBIF interface in independent layer computation or to next pipeline stage in fusion mode.

D_POOLING_KERN_CFG 0xD034

- Kernel width [3 : 0]
- Unused bits [7 : 4]
- Kernel height [11 : 8]
- Unused bits [15 : 12]
- Kernel stride in width direction [19 : 16]
- Kernel stride in height direction [23 : 20]

The bits of this control register are used to configure the pooling kernel shape and strides. The maximum kernel dimension are bounded by data path bus parallelism (more in Sec. 5.3.2). Median pooling supports square kernel shapes not larger than 3x3.

D_POOLING_PADDING_CFG 0XD040

- Left Padding [2 : 0]
- Unused [3]
- Top Padding [6 : 4]
- Unused [7]
- Right Padding [10 : 8]
- Unused [11]
- Bottom Padding [14 : 12]
- Unused[31 : 15]

D_DATA_FORMAT 0xD084

- Data Precision [1 : 0]:
  i) "00" : INT8
  ii) "01" : INT16
  iii) "10" : FP16
- Unused [31 : 2] : Zero hardwired

OTHER REGISTERS

- D_OP_ENABLE : this register is exploited to trigger execution with current address space configuration
- S_POINTER : select which group of register is accessed by CSB Master
- D_DATA_CUBE_IN_WIDTH : input tensor width -1
- D_DATA_CUBE_IN_HEIGHT : input tensor height -1
- D_DATA_CUBE_IN_CHANNEL : number of channels of input tensor -1
- D_DATA_CUBE_OUT_HEIGHT : output tensor height -1
- D_DATA_CUBE_OUT_WIDTH : output tensor width -1

For additional information about the PDP address space or NVDLA address space in general please refer to *hardware manual* in their official web site :
http : //nvdla.org/hw/v1/hwarch.html#pdp.

5.3.2 Planar Data Processor Data Path

Through strobes, it has been possible to reverse engineer the data flow of *Planar data processor* exploiting the scripts provided in *verif/traces/traceplayer* directory. Behavioral simulations leverage on *Verilator* tool to translate Verilog RTL description of the core into C++ code. The *Data Flow Graphs* (DFGs) of max pooling are reconstructed for two kernel configurations (see Fig. 5.4). Additional simulations have been conducted to analyse data flow behavior against input data parallelism (INT8/INT16). The operators of DFGs are executed in two different sub-cores : *PDP core cal1d and cal2d.* Operations between neurons in same rows are executed in cal1d unit and outcome is fed to cal2d unit through a series of FIFOs.

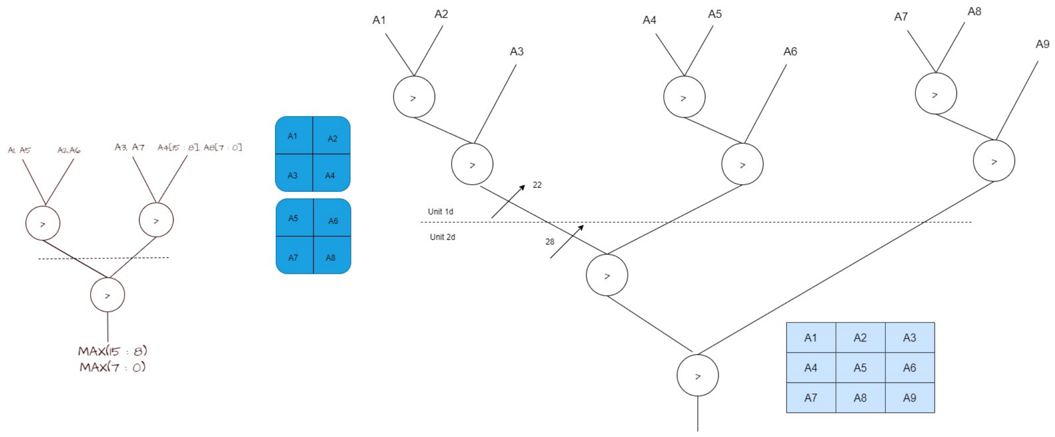

Figure 5.4: DFGs for several kernel configurations (2x2,3x3) and different data formats. Leftmost DFG represents the data flow for 2x2 kernel exploiting INT8 parallelism. On the right, the DGF for 3x3 kernel with INT16

As shown in DFGs, while connections in *cal1d* unit are on 22 bits, sign extension increases the parallelism up to 28 bits in *cal2d* unit. This design decision handles overflow during average pooling execution with INT16 parallelism. Interestingly, when configuring data parallelism to INT8 (see Fig. 5.4 leftmost DFG), the operators handle two data in each edge of DFG. In fact, through concatenating two INT8 in the same 22 bits bus, throughput is improved by doubling data processed per clock cycle. To conclude, sporadically, some operations against null operands are executed to clear/fill buffers.

5.3.3 Median Core

In order to exploit identical synchronization of control signals in *pdp* data path, the median core unit has been designed as a fully combinational circuit. Unfortunately, although performing partial sorting of neurons at each operator, all inputs must propagate from *cal1d* to *cal2d* unit. In fact, the operators need to determine the sorting position of more than 50% of the inputs before dropping any neurons. In this regard, operators in *call1d* unit mainly execute sorting and concatenation of neurons. On the other hand, concatenation strongly depends on the architecture parallelism and number parallelism. Therefore, the utilization of median core is limited and does not have the same configuration flexibility of Max/Avg pooling. Without changing drastically the data path of *pdp*, it has been possible to guarantee correct execution of median pooling layers with square kernel size not larger than 3x3 and data precision up to INT8. This design constrain strongly depends on *cal1d* and *cal2d* parallelism (respectively 22 and 28 bits). In addition, Median core operators cannot support multiple data per edge as Max core when parallelism is low (INT8). Therefore, although configuring the parallelism to INT16 in the address space, data are expressed on 8 bits, lowering throughput.

Kernel 2x2

When configuring kernel shape to square 2x2 filter, operators in *call1d* unit perform:

i) MSBs drop: since input data are on 16 bits but information is only coded on 8 bits, the sign bits are dropped

ii) Sorting: signed comparison to perform partial sorting. Data are partially sorted in order to reduce on chip area and improve hardware reusability.

iii) Packing: sorted data are packed into 16 bits. Concatenation is performed taking into consideration sorting output. Hence devising a packing unpacking procedure that propagates neurons order without additional sequential logic. For example, when activation a_1 is bigger then activation a_2, a_1 is packed in the least significant bits (i.e, 7 downto 0) and a_2 in the most significant bits (15 downto 8).

Results from *call1d* sub-core are propagated to *call2d* in order to extrapolate median points through:

i) Unpacking: at the input of *call2d* operator, 4 INT8 neurons are fed through 2 packets of 16 bits.

ii) Median point computation: the median value is obtained through a series of comparisons of presorted inputs to determine the penultimate neuron (Median point). In details, the larger neurons computed by *cal1d* unit are compared to obtain the minimum between them. In addition, the maximum over the presorted minimums is computed. Finally, the median value is obtained as the minimum of the results of previous comparisons. Therefore with a critical path of two INT8 comparators, the median point is extrapolated.

Due to 22 bits parallelism, INT16 and FP16 parallelism are not handled. In fact, concatenation would have required 32 bits busses.

Kernel 3x3

The maximum handled filter size is imposed by bus parallelism. In fact, for this particular configuration, up to three INT8 neurons must be encoded on 22 bits in the *call1d* unit. Therefore in order to support median sorting algorithm with up to 9 neurons per kernel, a lossless compression methodology has been designed to pack the inputs of *call1d* sub-unit and communicate them to *call2d* core. The devised lossless compression technique exploits the sorted position of neurons to encode MSBs in a more compacted form. Two operators are executed in *cal1d* for this particular kernel configuration. While the first operator can simply perform sorting/concatenation (as explained in Sec. 5.3.3), the second has to perform compression through:

i) Unpacking and MSBs drop: through flag bits (bits 21 downto 16) written while packing data in the previous operator, the design is able to recognize which between the inputs comprises two neurons without leveraging on sequential circuitry. The bus comprising two data is unpacked. At the same time MSBs of the other bus are dropped.

ii) Sorting: the 3 INT8 numbers are sorted.

iii) LSBs Packing: the 5 least significant bits of each neuron are packed according to partially sorted order.

iv) MSBs Compression: the Most significant bits (3 bits for each activation for a total of 9) are fed to a LUT that encodes them on 7 bits (see Fig. 5.5).

The idea behind this compression is to exploit the packed order of least significant bits to pay as a compression cost the order of MSBs. While decoding, the shuffled MSBs are re-associated to their suffixes to reconstruct INT8 data with the same sorting order of packed LSBs. Therefore, since MSBs order is irrelevant for data reconstruction, equivalent permutations can be collapsed on the same representation (code). During decoding, the code is used as a key to access the **Compression LUT** to obtain back shuffled MSBs. Through dropping equivalent permutations of the same MSBs (see example in Sec. A.0.4) , it has been possible to compress 9 bits to 7 respecting bus parallelism without losing valuable information. Through a python simulation, all possible combinations of 3 signed INT8 are fed to a behavioral description of the proposed compression methodology and losless has been ensured.

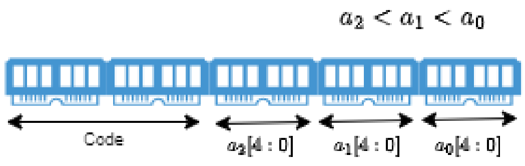

Figure 5.5: Data packing at the end of second level of operations in Median3x3 DFGs

The 22 bits packets of 3 INT8 numbers are fed to *cal2d* unit that through 2 operators (latest executed operators in Max pooling DFG for 3x3 kernel configuration) extrapolates median point. Firstly two packets, each comprising 3 neurons are fed to a operator that:

i) Unpack & Reconstruct : for each input bus, unpacks the least significant bits and codes. Access the LUT through codes to decode MSBs. Associates shuffled MSBs to their relative LSBs in order to reconstruct neurons with same sorting order of packed least significant bits.

ii) Partial Sorting: through a comparison based, binary decision tree, 6 neurons, pre-sorted in groups of 3, are ordered with an overall latency of 1 INT8 comparison

iii) Extremes Dropping: the lower activation and biggest activation can be dropped at this stage since they are not going to be the median point. In fact, at this stage about 66% of the inputs have been sorted and data can be dropped.

iv) Compression and Packing: 4 INT8 neurons must be packed on 28 bits bus. Furthermore, requiring at least one bit as a flag to recognize the number of neurons comprised in the bus, the overall data need to be represented using 27 bits. The 4 LSBs of each neuron are packed and the MSBs are fed to a second LUT that compresses 16 bits into 12. Nevertheless, 12 bits code and 16 bits packed LSBs and 1 flag bit end up exceeding the output bus parallelism. Therefore, the lowest activation is rounded to the nearest even in order to save one bit, generating a potential loss. Nevertheless, this design decision stems from the huge LUT size required to encode MSBs 5 bits wide to enhance compression performances.

If one among the inputs has an active flag, the following operator is executed to obtain median value:

i) Unpacking & Decoding: both busses are unpacked and codes are exploited to access LUTs to decode 4 and 3 neurons respectively

ii) Median tree : through an additional comparator based binary decision tree, median value is computed with a latency of 1 INT8 comparison

iii) Sign extension: since the data format is imposed on INT16 parallelism, the sign of median value is extended

5.3.4 Functional Simulations

All code is open source and available on GitHub at:
https://github.com/fpessia/MEDIANVDLA/tree/main

Exploiting *Verilator* to convert RTL description of NVDLA SoC, designed through verilog, into C++ code, behavioural simulations are performed to ensure correct functionality. First of all, short stand alone tests have been performed both on *call1d* and *call2d* modifications (Median core itself). Through terminal interface ("*$display*"), it has been possible to monitor I/O of median core. Two stand alone tests ensure correct functionality of DFGs executed both in 2x2 (*verif/traces/traceplayer/pdp_med2x2_pooling_int_8/Tb_Kernel2*) and 3x3 kernel configuration (*verif/traces/traceplayer/pdp_med3x3_int8/Tb_Kernel3*). Subsequently the *MEDIANVDLA* SoC undergoes a full functional test executing a Median Pooling Layer with fictitious input tensor. Exploiting a Python script (*LayerIOgeneration.py*) a random input tensor is generated and fed to a Median Filter. The golden values are stored in binary files (*input_feature_map.dat* & *output_feature_map.dat*) with memories organized in *Little Endianess* and *Surface Packed*. Through Perl scripts, *NVDIA* engineers provided a simple text based programming language to drive CSB and DBB interface during C++ simulations. Therefore, through so called *traces*, the address space can be modified as well as DRAM content.

```bash
#!/bin/bash

# Cloning GitHub repo
git clone git@github.com:fpessia/MEDIANVDLA.git
#running configuration tree script
cd MEDIANVDLA
make
# Computing golden values
cd verif/traces/traceplayer/pdp_med2x2_pooling_int8
python LayerIOgeneration.py
# Compiling Soc with Verilator V5.019
#configured from source code (Ubuntu 20.04)
cd ../../../../
./tools/bin/tmake -build verilator
#Running simulation script 2x2 kernel conf
cd verif/verilator
```

```
make run TEST=pdp_med2x2_pooling_int8
#or 3x3
#make run TEST=pdp_med3x3_int8
```

Listing 5.1: Bash script for 2x2 or 3x3 median functional tes

A trace (*./pdp_med2x2_pooling_int8/input.txt*) has been programmed to drive control signals to execute a Median Pooling layer with following parameters:

i) I : 64x8x8

ii) K : 2x2

iii) Stride : 1x1

iv) O : 64x7x7

In order to run this functional test, all bash commands listed above (see Lising 4.1) must be executed. Furthermore, by simply changing *pdp_med2x2_pooling_int8* into *pdp_med3x3_int8* in all of its occurrences it is possible to run a test with the following layer parameters:

i) I : 64x8x8

ii) K : 3x3

iii) Stride : 1x1

iv) O : 64x6x6

The random neurons of input tensor are all even integers for 3x3 kernel configuration, in order to perform an error free simulation. Nevertheless, when also taking into consideration odd numbers a small error up to one unit can be introduced.

Core Limitations

As already mentioned in previous subsections, the main drawback of *MEDIANVDLA* design is limited configurability. Nevertheless, while configuring kernel size larger than those handled, the execution doesn't crash but performs MIN/MAX filtering (mentioned in Sec. 5.2). The same strategy has been adopted when dealing with a parallelism different than INT16 (e.g. FP16). An additional limitation is caused by buffer filling/cleaning. In fact, although comparison against null operators doesn't effect computation of MAX pooling layers, for what concerns MIN and MEDIAN pooling this dataflow can corrupt some neurons. A zero discard mechanism is implemented in MEDIAN core to neglect null operands. Nevertheless, being unable to distinguish between null operands and zero activations, this version of the core doesn't support zero padding. Furthermore, zero activations must be represented in the following format 0xFF00 in order to not being discarded by the core.

Chapter 6

Conclusion

This work explored and analysed the criticality of hardware faults for applications relying on matrix multiplications such as CNNs. Through modeling TCU permanent faults behavior in errors we propose a different approach to perform fault injection at the application level targeting massive architectures. Relying on classical injection procedures, faults have been modelled through a scalar and spatial characterization. In order to produce hardware-aware errors, multiple fault injection campaigns have been exploited for feature extraction. Unfortunately, experimental data suggest that an universal representation of TCU permanent faults through *bit flip masks* is not obtainable. In fact, errors magnitude depends both on fault location (*hardware*) and GEMM size (*application*). In this regard, size-aware errors have been modelled. Our approach has shown good reproduction capabilities (up to 92% correlation) optimizing drastically simulation complexity (225× injection time gain). We firmly believe that our is one of the first steps towards a different approach in the field of reliability analysis. Although Moore's slow is slightly slowing down, *System on Chips* will surely comprise an increasing number of components due to parallelization in the future. Classical fault injection will slowly became obsolete to asses the effects of hardware faults in applications. With our contributions to the field we hope to have clarified the criticality of new procedures to map/extrapolate faults features into errors. In our work we propose to leverage bit flip masks, but alternatives need to be considered. For example, saturation masks (OR masks, AND masks, or a combination of both) can be exploited to model those faults that have been labelled as "unmodellable" in our approach (about 8% of strudied faults for seed GEMM 100x100x100). In addition, we point out the necessity to support Median Pooling in modern SoCs. CNNs trained through this pooling algorithm have shown outstanding application stability. For *LeNet5* trained through Median filters, the *classification accuracy loss* is marginal (about 2%) with respect to performance degradation of *LeNet5* trained with Avg and Max pooling (more than 20%). Our version of NVDLA is not meant to be a final product but a first step towards more sophisticated designs supporting this keen operation for safety critical applications. Machine Learning is growing in popularity in our every day life and we would be unconscious to neglect safety.

Appendix A

Additional Informations

A.0.1 Queue

The *Queue* is a Python class of *multiprocessing* module. This class provides an interface for different process data sharing. In fact, for each issued process, a different address space is allocated. Although variables are characterized by identical names of father process, their content is associated to a different memory location in child process. Therefore, data sharing is either performed through files, global variables (not suggested by literature, requires locks) or FIFOs (*Queue*). In the fault injection campaign, a process, running Validator module, waits until data are pushed into the FIFO and extracts them for processing. On the other hand, Injector module, pushes data into the FIFO as soon as it completes GEMM. Through this paradigm it is possible to enhance computation performances, exploiting multiple CPUs during FIC.

A.0.2 MAC Normalization

MAC (*Multiply & accumulate*) operations are executed though a multiplication and a sum between 3 floating point numbers. In fact, two inputs (from now on A and B) are multiplied and the result is summed with an accumulator (C). Normalization can either occur during multiplication or addition. In particular, in order to support overflow for FP16 multiplications, 20 bits are allocated to store the multiplication between 10 bits mantissas. Normalization is performed in order to represent mantissa multiplication result on 10 bits, updating the exponent accordingly. Furthermore, normalization process can occur also during floating point sum in order to obtain two operands with identical exponent to perform mantissa addition. Nevertheless, while the former normalization can propagate faults in mantissa into corrupting the exponent, the latter does not.

A.0.3 Error Models Golden Values

Bit flip masks have been generated starting from multiple fault injection campaigns with several input tensor shape (see Tab. A.1).

Input tensor dimension	MAC/operand
100x100x100	100
200x200x200	200
300x800x300	800
400x1200x400	1,200

Table A.1: Set of golden values. The AxBxC format indicates the shape of input tensors with B being common dimension

A.0.4 Compression example

Input1	Input2	Code
00	00	0
00	01	1
00	10	2
00	11	3
01	00	1
01	01	4
01	10	5
01	11	6
10	00	2
10	01	5
10	10	7
10	11	8
11	00	3
11	01	6
11	10	8
11	11	9

Table A.2: Compression example for 1 bit inputs

This subsection aims at providing a simple example of how LUT in Median core compresses MSBs in a lossless code. For example, lets consider a simple 2 busses 2 bits wide. All 16 possible combinations (listed in Tab. A.2) are compressed in only 10 codes since identical permutations are characterized by identical compression.

Through increasing number of inputs and bus size, the compression methodology performances improve and bits can be saved.

Bibliography

[1] Yizeng Han, Gao Huang, Shiji Song, Le Yang, Honghui Wang, and Yulin Wang. *Dynamic Neural Networks: A Survey.* 2021. arXiv: `2102.04906 [cs.CV]` (cit. on p. ii).

[2] Amir Erfan Eshratifar, Amirhossein Esmaili, and Massoud Pedram. *BottleNet: A Deep Learning Architecture for Intelligent Mobile Cloud Computing Services.* 2019. arXiv: `1902.01000 [cs.DC]` (cit. on pp. ii, 3).

[3] Saurabh Hukerikar and Nirmal Saxena. «Runtime Fault Diagnostics for GPU Tensor Cores». In: *2022 IEEE International Test Conference (ITC).* 2022, pp. 524–528. DOI: `10.1109/ITC50671.2022.00065` (cit. on pp. 1, 23).

[4] Jongmin Jo, Sucheol Jeong, and Pilsung Kang. «Benchmarking GPU-Accelerated Edge Devices». In: *2020 IEEE International Conference on Big Data and Smart Computing (BigComp).* 2020, pp. 117–120. DOI: `10.1109/BigComp48618.2020.00-89` (cit. on p. 3).

[5] Md Aamir Raihan, Negar Goli, and Tor Aamodt. *Modeling Deep Learning Accelerator Enabled GPUs.* 2019. arXiv: `1811.08309 [cs.MS]` (cit. on p. 4).

[6] Brent Ralph Boswell et al. *Generalized acceleration of matrix multiply accumulate operations.* U.S. Patent 10,338,919. July 2019 (cit. on p. 4).

[7] Lu Wang Xia Zhao David Kaeli Zhiying Wang Member and Lieven Eeckhout. «Intra-Cluster Coalescing and Distributed-Block Scheduling to Reduce GPU NoC Pressure». In: 14 (Aug. 2015), p. 15 (cit. on pp. 4, 7).

[8] Vivienne Sze Yu-Hsin Chen Tien-Ju Yang and Joel S. Emer. *Efficient Processing of Deep Neural Networks: A Tutorial and Survey.* 2017. arXiv: `1703.09039 [cs.CV]` (cit. on p. 6).

[9] Jianyu Huang Chenhan D. Yu Robert A. van de Geijn. «Implementing Strassen's Algorithm with CUTLASS on NVIDIA Volta GPUs». In: (Aug. 2018), p. 22 (cit. on p. 6).

[10] Peter H. Hochschild, Paul Turner, Jeffrey C. Mogul, Rama Govindaraju, Parthasarathy Ranganathan, David E. Culler, and Amin Vahdat. «Cores That Don't Count». In: *Proceedings of the Workshop on Hot Topics in Operating Systems*. HotOS '21. Ann Arbor, Michigan: Association for Computing Machinery, 2021, pp. 9–16. ISBN: 9781450384384. DOI: `10.1145/3458336.3465297`. URL: `https://doi.org/10.1145/3458336.3465297` (cit. on p. 8).

[11] Harish Dattatraya Dixit, Sneha Pendharkar, Matt Beadon, Chris Mason, Tejasvi Chakravarthy, Bharath Muthiah, and Sriram Sankar. *Silent Data Corruptions at Scale*. 2021. arXiv: `2102.11245 [cs.AR]` (cit. on p. 8).

[12] Yi He Mike Hutton Steven Chan Robert de Grujil Rama Govindaraju Nishant Patil Yanjing Li. «Understanding and Mitigating Hardware Failures in Deep Learning Training Accelerator Systems». In: (June 2023), p. 16 (cit. on pp. 8, 23, 26).

[13] Atieh Lotfi, Saurabh Hukerikar, Keshav Balasubramanian, Paul Racunas, Nirmal Saxena, Richard Bramley, and Yanxiang Huang. «Resiliency of automotive object detection networks on GPU architectures». In: *2019 IEEE International Test Conference (ITC)*. 2019, pp. 1–9. DOI: `10.1109/ITC44170.2019.9000150` (cit. on pp. 8, 10, 23).

[14] Yi He, Takumi Uezono, and Yanjing Li. «Efficient Functional In-Field Self-Test for Deep Learning Accelerators». In: *2021 IEEE International Test Conference (ITC)*. 2021, pp. 93–102. DOI: `10.1109/ITC50571.2021.00017` (cit. on p. 8).

[15] Robert Limas Juan David Balaguera Josie E. Rodriguez Condia Matteo Sonza Reorda Sierra. «Analyzing the Impact of Different Real Number Formats on the Structural Reliability of TCUs in GPUs». In: *2023 IFIP/IEEE 31st International Conference on Very Large Scale Integration (VLSI-SoC)*. 2023, pp. 1–6 (cit. on p. 11).

[16] R. Leveugle A. Calvez P. Maistri P. Vanhauwaert. «Statistical Fault Injection: Quantified Error and Confidence». In: () (cit. on p. 11).

[17] Robert Limas Sierra Juan-David Guerrero-Balaguera Francesco Pessia Josie E. Rodriguez Condia Matteo Sonza Reorda. «Analyzing the Impact of Scheduling Policies on the Reliability of GPUs Running CNN Operations». In: *to be appeared in 42nd IEEE VLSI Test Symposium* (2024) (cit. on p. 13).

[18] Josie E. Rodriguez Condia, Boyang Du, Matteo Sonza Reorda, and Luca Sterpone. «FlexGripPlus: An improved GPGPU model to support reliability analysis». In: *Microelectronics Reliability* 109 (2020), p. 113660. ISSN: 0026-2714. DOI: `https://doi.org/10.1016/j.microrel.2020.113660`. URL: `https://www.sciencedirect.com/science/article/pii/S0026271419307978` (cit. on p. 23).

[19] Onur Keleş, M. Akın Yılmaz, A. Murat Tekalp, Cansu Korkmaz, and Zafer Dogan. *On the Computation of PSNR for a Set of Images or Video*. 2021. arXiv: `2104.14868 [eess.IV]` (cit. on p. 25).

[20] Yi He, Prasanna Balaprakash, and Yanjing Li. «FIdelity: Efficient Resilience Analysis Framework for Deep Learning Accelerators». In: *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2020, pp. 270–281. DOI: `10.1109/MICRO50266.2020.00033` (cit. on p. 26).

[21] Philipp Liznerski, Lukas Ruff, Robert A. Vandermeulen, Billy Joe Franks, Marius Kloft, and Klaus-Robert Müller. *Explainable Deep One-Class Classification*. 2021. arXiv: `2007.01760 [cs.CV]` (cit. on p. 28).

[22] Juan Pimentel and Jennifer Bastiaan. «Characterizing the Safety of Self-Driving Vehicles: A Fault Containment Protocol for Functionality Involving Vehicle Detection». In: *2018 IEEE International Conference on Vehicular Electronics and Safety (ICVES)*. 2018, pp. 1–7. DOI: `10.1109/ICVES.2018.8519488` (cit. on p. 29).

[23] Weichen Liu, Jiang Xu, Xuan Wang, Yu Wang, Wei Zhang, Yaoyao Ye, Xiaowen Wu, Mahdi Nikdast, and Zhehui Wang. «A Hardware-Software Collaborated Method for Soft-Error Tolerant MPSoC». In: *2011 IEEE Computer Society Annual Symposium on VLSI*. 2011, pp. 260–265. DOI: `10.1109/ISVLSI.2011.48` (cit. on p. 29).

[24] Shubu Mukherjee. *Architecture design for soft errors*. Morgan Kaufmann, 2011 (cit. on p. 29).