Resilience-Performance Tradeoff Analysis of a Deep Neural Network Accelerator

Salvatore Pappalardo[1], Annachiara Ruospo[2], Ian O'Connor[1], Bastien Deveautour[3], Ernesto Sanchez[2], Alberto Bosio[1]
[1,3]*Univ Lyon, ECL, INSA Lyon, CNRS, UCBL, CPE Lyon, INL, UMR5270, 69130 Ecully, France*
[2]*Politecnico di Torino, Dip. di Automatica e Informatica, Torino, Italy*
Email: [1]{name.surname}@ec-lyon.fr, [2]{name.surname}@polito.it, [3]{name.surname}@cpe.fr

***Abstract*—Nowadays, Deep Neural Networks (DNNs) are one of the most computationally-intensive algorithms because of the (i) huge amount of data to be transferred from/to the memory, and (ii) the huge amount of matrix multiplications to compute. These issues motivate the design of custom DNN hardware accelerators. These accelerators are widely used for low-latency safety-critical applications such as object detection in autonomous cars. Safety-critical applications have to be resilient with respect to hardware faults and Deep Learning (DL) accelerators are subjected to hardware faults that can cause functional failures, potentially leading to catastrophic consequences. Although DNNs possess a certain level of intrinsic resilience, it varies depending on the hardware on which they are run. The intent of the paper is to assess the resilience of a systolic-array-based DNN accelerator in the presence of hardware faults, in order to identify the architectural parameters that may mainly impact the DNN resilience.**

***Index Terms*—DNN Hardware accelerators, Fault Injection, Reliability**

I. Introduction

Many different applications such as image segmentation, natural language processing and object detection and recognition are implemented with Deep Neural Networks (DNNs). These models require a huge quantity of computationally expensive matrix multiplication operations that, associated with nonlinear activation functions and the huge amount of data to be transferred from/to the memory, represent a performance bottleneck [1]. In order to satisfy performance and power budget constraints of real-time safety-critical systems, dedicated Deep Neural Network (DNN) hardware accelerators are widely used [2]. These architectures are usually implemented as systolic arrays: a grid of synchronous Processing Elements (PEs) that can perform iterative algorithms with regular data dependencies [3]. Even though systolic architectures were invented in the 80s, they have been broadly used in recent years [2], [4]. For example, in 2016 Google announced their Tensor Processing Unit (TPU) [5], an application specific integrated circuit used for accelerating DNNs inferences.

Nevertheless, hardware is prone to failure due to manufacturing process (latent defects, variation in parameters), silicon aging, environmental stress (vibrations, heat, humidity) or even Single Event Upsets caused by ionization. These faults can cause operational error, potentially leading to functional failures and therefore fatal consequences, especially for safety-critical systems [6]. Estimating the resilience of such hardware is mandatory for designing fault-tolerant hardware.

The research community is moving toward understanding the resilience of hardware accelerators exploited in DL applications. In the literature, many DL accelerators, such as [7]–[9], have been proposed and many efforts have been done to assess their fault resilience (e.g., [1], [10]). As an example, the authors of [10] used the Eyeriss accelerator to assess the fault tolerance of different CNNs. The objective of these studies was to find the most critical parameters (e.g. the layer) of the CNN model. In [11], the authors investigated the fault tolerance of Google's TPUs and used the results to design a custom fault-tolerant TPU. The idea consisted in the static mapping between weights and Multiply and Accumulate (MAC) units to determine which weights to prune. Another popular accelerator is the Nvidia's opensource NVDLA [8]. Reference [12] shows how to exploit Automatic Test Pattern Generation (ATPG) for in-field testing the occurrence of permanent faults in the architecture, promising to improve the fault tolerance of DNNs running on it.

The goal of this paper is to assess the resilience of a custom DNN accelerator (based on systolic arrays) by varying architectural parameters. The resilience analysis is done by injecting stuck-at faults and comparing faulty outputs with respect to fault-free outputs. During the injection campaign, we varied the size of the systolic array and its bit-width. Interestingly, we found out that the resilience depends not only on the deployed DNNs and the dataset, but notably on the hardware architecture.

The rest of the paper is organized as follows. Section II presents previous works in the literature. Section III presents the custom hardware and its ecosystem. Section IV describes the setup of the different performed experiments and section V shows the obtained results. Finally, section VI concludes the paper.

II. Background

Fault injection is a widely used testing technique to understand the behavior of the system under test in faulty scenarios. It is based on the realization of controlled experiments to evaluate the system behavior in the presence of artificial faults. Many surveys and books discuss fault injection in detail [6],

979-8-3503-3277-3/23/$31.00 ©2023 IEEE

[13]. Different techniques exist for performing fault injections and are usually classified depending on the abstraction level.

Some papers propose the injection at software level [14]–[16]. The most investigated aspects include the fault location and the execution platform. These techniques provide results with useful insights and are generally used because they are extremely cheap to conduct, both in terms of time and resources. The main drawback is that there is no information about the hardware that executes the computations.

To this end, the research community recently took interest into another well known direction: hardware-level fault injections. It usually consists of simulating the faulty hardware and comparing the faulty output with the fault-free one. Although these techniques require more resources and time, they are also more flexible and accurate than their software counterpart. For example, when injecting at software level, it is not possible to take into account underlying hardware — a systolic array in our case — let alone how its size affects the fault propagation. Recently, many authors have exploited these techniques [1], [17], [18]. Their results carry more information about actual real-world faults, even though it implies targeting a specific hardware. Therefore, a comprehensive resilience assessment of the systems can be obtained only when explicitly considering the hardware platform running the neural network.

Another method consists in injecting faults in manufactured hardware using ions or laser beam, and then analyze its behavior. An example of this technique is given in [19]. This method was disregarded for this paper since it is costly, requires manufactured hardware, and there is no control over the injection.

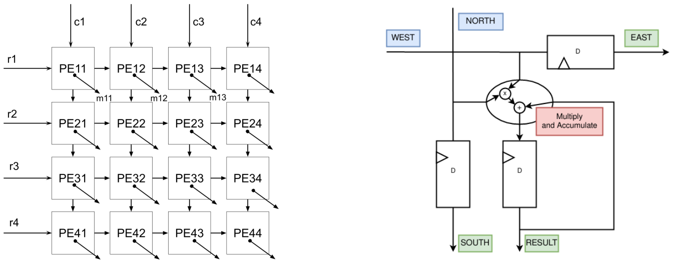
III. Resilience Framework

In this section, we briefly describe the designed hardware architecture with its ecosystem and the fault injection framework developed to perform the injection campaigns.

A. Architecture

As [20] points out, there are three main types of systolic arrays: **(i) Weight stationary**, in which the weights are "fixed" at each Processing Element (PE) and the accumulation operation is performed through the columns; **(ii) Output stationary**, in which both weights and activations flow in the array in perpendicular directions, while the PEs perform the MAC operation using an accumulation register; **(iii) No local reuse**, similarly to the first type, the partial sum flows through the columns, but in this case the weights have a dedicated bus.

We developed a custom systolic array of the output stationary type. Figure 1a shows the general architecture. The weights flow "vertically" (i.e. from the NORTH to the SOUTH of each PE) while the activations flow "horizontally". Figure 1b shows a single PE with its inputs (NORTH and WEST) and its outputs (EAST, SOUTH and RESULT). Our implementation is written in VDHL and is effortlessly configurable in order to easily perform experiments.



(a) Custom systolic array

(b) Single PE internals

Fig. 1: Systolic array architecture

B. HW-to-DNN mapping

Even though the same architecture can be used for fully connected layers, in this work, only convolutional layers are considered. A DNN convolutional layer can be defined by the following triplet:

$$Conv = C@N \times N \tag{1}$$

where C is the number of channels (and thus different convolutions) composing the layer and N is the size of the convolution. The proposed systolic array is mapped to a single channel, and it is able to compute a single convolution at a time with an $O(N)$ complexity (i.e., linear w.r.t. to one dimension of the convolution). Please note that the goal of our systolic array is not to accelerate the execution of the DNN, but to simulate the execution of a single channel at the RTL level. In other words, we intend to have the smallest RTL description able to run the channel and perform a fault injection.

The developed systolic array can perform a single convolution at a time, therefore, the same array has to be reused for different channels, otherwise multiple arrays are needed. For a single convolution, each pass is performed by a single PE, so that each PE contains the value of a single pixel of the output.

There are two scenarios: (i) the array is larger or equal than the convolution output, in which case some PEs may stay idle and do nothing; (ii) the array is smaller than the input. This second scenario is more interesting, since it requires hardware re-usage. Indeed, in this case the image is processed in chunks, which have the same size as the array. This also means that the same faulty PEs computes different parts of the input image, effectively injecting multiple faults in the output.

Obviously, the larger the array, the higher the throughput of the architecture.

C. Fault injection framework

In order to perform the injections, a custom cross-layer framework was used. Figure 2 shows a high level representation of the framework. The basic idea is to simulate the first layer in hardware, injecting the fault, and then passing the simulation output as the input of the second layer using traditional DNN techniques (such as Tensorflow or PyTorch). We used QuestaSim for hardware simulations and injections and then a C-compiled version of the network, written using the n2d2 library [21]. This approach allowed quick inferences without sacrificing the flexibility of hardware-level injections. Furthermore, the framework supports parallel injections, reducing even more the required fault injection campaign time.

IV. Experiments

A. DNN model

The neural network model used for the experiments is the LeNet-5 [22] architecture. The model is composed by 7 layers: the first four are convolutional, while the last three are fully connected. The convolutional layers' sizes are 6@28x28, 6@14x14, 16@10x10 and the last is 16@5x5. The network was trained on the MNIST dataset [22], obtaining 99% accuracy when testing 10'000 images from the validation set. Note that we define as "accuracy" of the neural network the capability to correctly classify the input picture; the accuracy is computed by using the top-1 score [22].

We implemented only the first layer of this network, since it was shown to be the most critical [14]. It is composed by 6 2D-convolutions (or channels). It accepts 32x32 images in input and produces 6 28x28 activations. The last layer outputs 10 values corresponding to the probability of the input to be classified with one of the corresponding label.

Since the first layer is composed by 6 channels, our architecture is composed by 6 systolic arrays that concurrently compute the 6 different convolutions.

B. Setup

In this section, we will describe the setup used for our experiments.

We carried out three sets of experiments, varying three parameters:

1) **Precision** of the network - it describes whether the network is implemented with 16 or 8-bit values;
2) **Injection point** - we injected both the activations and the weights, we will refer to weight-injection experiments with the letter 'w' and to activations-injection (or input-injection images, since it is the first layer) with the letter 'i';
3) **Array size** - as explained before, it corresponds to the physical size of the array. We varied this parameter among three values: 28, 14 and 7.

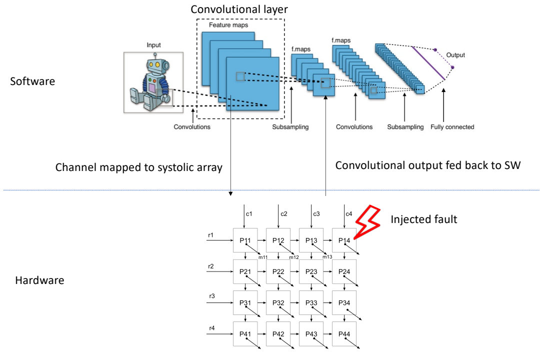


Fig. 2: Fault Injector framework

Given these three parameters we could refer, for example, to the experiment 8w7 referring to the experiment with an 8-bit network implemented on a 7x7 systolic array and injecting in the weights inputs. Note that when using the symbol X, we refer to the entire set of experiments varying that parameter. As an example, experiments 8wX are all the experiments with 8bit network and injecting the weights, while experiments XwX are all the experiments where the injections are made in the weights.

The complete list of the performed experiments is then 8w28, 8w14, 8w7, 16w28, 16w14, 16w7, 8i28, 8i14 and 8i7.

C. Fault model

The targeted fault model is stuck-at faults injected during the simuation. This fault can be interpreted in two ways: either (i) the fault is transient and only affects the array during the computation of the first layer, or (ii) the fault is permanent but the array is only used for computing the first layer (because of some constraint) while the rest of the network is processed by different hardware.

In a given simulation, only one fault is injected. It is defined in function of three parameters:

- PE: this parameter describes which PE is injected, among all the possible PEs in the architecture;
- Bit: it is the bit that will be modified by the fault. It depends on the PE data precision (i.e., bit-width),
- $Polarity$: whether the fault is a stuck-at-0 or a stuck-at-1.

The space of possible faults is then 3-dimensional, and its size depends on both the array size and the bit-width:

$$\text{FaultSpace} = \underbrace{2}_{\text{Polarity}} \times \underbrace{K}_{\text{Bit}} \times \underbrace{6 \times M}_{\text{PE}} \tag{2}$$

where K is the bit-width and M is the number of PEs per systolic array. Note that all of our experiments are performed injecting "internal" PEs. This means that in the computation of M, the first row (when injecting in the weights, first column otherwise) has to be excluded. For example, for experiments XX7 M is $7 \times 6 = 42$.

The size of the faults space varies depending on the specific experiment. In order to obtain statistically relevant results, we used the method described in [23]. The number of injections per experiment is determined by equation 3:

$$n = \frac{N}{1 + e^2 + \frac{N-1}{t^2 - p(1-p)}} \tag{3}$$

where N is the size of the fault space (as computed with equation 2; e corresponds to the margin error (i.e. the error on the estimation of the experiments) and we assumed a value of 1%; p is the estimated probability of a fault resulting in a failure (we assumed 50%); t is the cut-off point and depends on a confidence level, we set $t = 2.58$ corresponding to a 99% confidence level; finally n is the minimum number of injections to perform to have the desired characteristics (in our case, 1% error margin with 99% confidence level).

Table I shows the number of injected faults per each experiment. Due to the relatively small size of the space for experiment 8w7 and 8i7, we performed an exhaustive injection in every PE.

TABLE I: Fault injected per experiment

Experiment	fault space size	injected faults
8w28, 8i28	72,576	14,000
8w14, 8i14	17,472	8,600
8w7, 8i7	4,032	4,032
16w28	145,152	15'000
16w14	34,944	12,000
16w7	8,064	5,500

The faults were generated randomly. Each fault was applied to 100 different simulations, because each simulation was done with a different randomly selected input (from the validation set). The total number of simulations for each experiment is 100 times what indicated in Table I.

D. Fault Injection for Deep Neural Networks

Even though DNNs can be seen as a software executed on a given hardware, there are some peculiarities that have to be considered to set up an effective fault injection campaign.

The fault impact has to be measured differently compared to classical fault injection. It is especially important for us to evaluate the **functional impact** of each fault. For this reason, we follow a categorization similar to [14].

The output of the faulty DNN is compared with a fault-free execution of the network with the same input. We defined two main categories:

- **Benign** faults: the output of the faulty DNN is either exactly the same (**masked** sub-category), or it is different, but the classification is correct and the confidence of the top-1 item is higher than the fault free (**good** sub-category).
- **Malignant** faults: there are three cases: the classification is correct but the top-1 probability is smaller (i) within 5% of the fault free (**accept** sub-category) or (ii) more (**warning** sub-category); worst case scenario, (iii) the input is misclassified (**critical** sub-category).

V. Results

In this section, we will overview the most significant results of our experiments.

A. General Resilience

In this subsection, we will show how the different faults affected the architecture in general. Table II shows the percentage of each fault category for experiments 8wX.

The data show a tiny amount of critical faults (consistently less than 1%), which is desirable, as it is the worst category of faults. Another important point is related to the percentage of masked faults. Indeed, it is the most frequent class of faults. Obviously, when injecting a single fault the effect can be small, but we need to consider that it can be amplified as it is propagated.

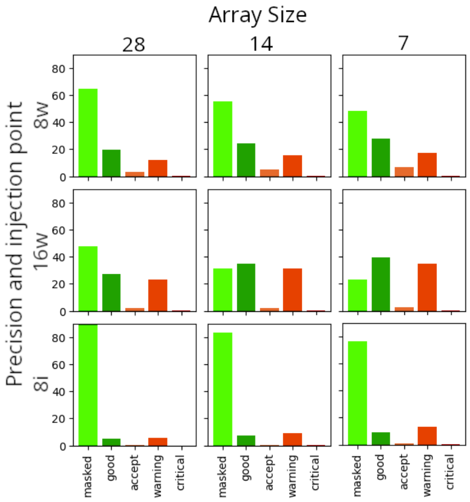


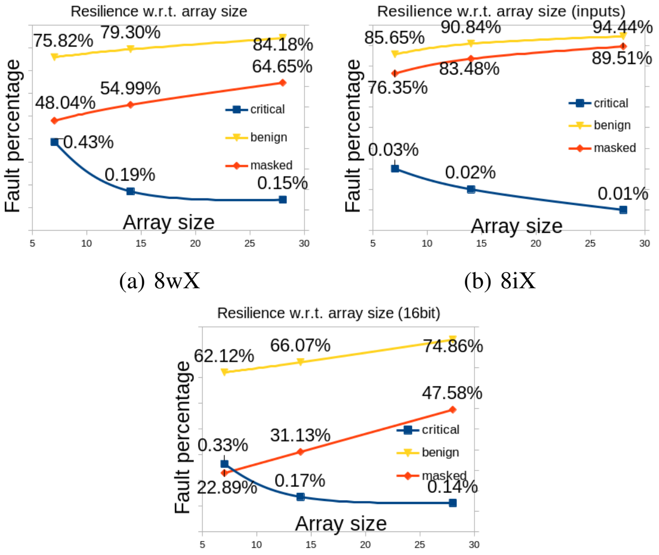
Fig. 3: General resilience performance per experiment. Each graph shows the percentage of each category of fault.

TABLE II: Fault classification in experiments 8wX

Experiment	8w28	8w14	8w7
Masked	64.65%	54.99%	48.04%
Good	19.53%	24.31%	27.78%
Accept	3.48%	4.78%	6.67%
Warning	12.19%	15.73%	17.07%
Critical	0.15%	0.19%	0.43%

Figure 3 summarizes the experimental results. It is possible to see that experiments 8iX show the highest resilience. This behaviour underlines a good disturbance rejection when corrupting values of the input image. Furthermore, the resilience of the 8bit hardware is higher than that of the 16-bit one. As the figure shows, in the former case, the masked faults are always at least 50% of the total, while in the latter the portion of masked faults drops to around 20% in 16w7. This result shows that faults in a 16-bit architecture are more critical than those in 8-bit architectures.

Another common trend is the reduction of masked faults with the reduction of the array size from 28 down to 7. A higher percentage of critical faults identifies a less resilient architecture, since it implies that a greater number of faults are likely to produce a mis-prediction. On the other hand, the higher the number of benign faults, the more resilient the architecture. Figure 4 shows the relationship between the array size and the resilience of the network. The curves are cubic interpolations of the points. It is possible to see that, while the percentage of benign faults grows linearly, the critical faults decrease in an exponential fashion, even though they always stay under 1%.



(a) 8wX

(b) 8iX

(c) 16wX

Fig. 4: Size-reliability tradeoff in the different experiments sets. The triangles designate benign faults, the rhombi designate masked faults, while the squares designate critical faults.

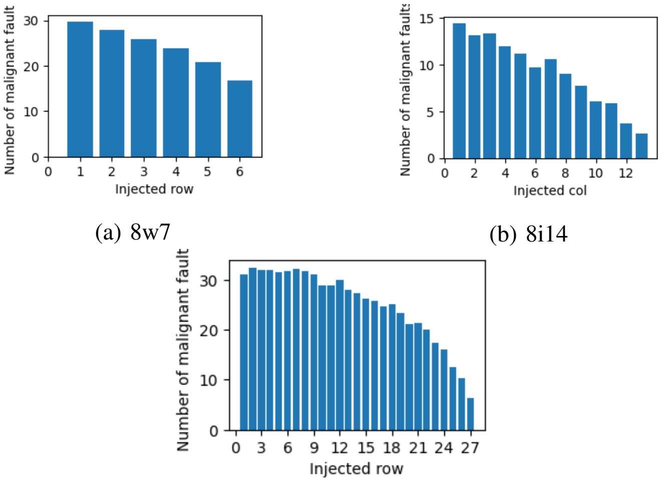
B. Position-based resilience

As explained before, a single injection produces multiple faults, since faulty values are propagated forward. Figure 5 shows the number of malignant faults with respect to the injected columns and rows. They all show a decreasing trend, even though experiments 8w7 and 16w28 show the number of malignant faults per each injected **row**. While experiment 8i14 shows the number of malignant faults with respect to each injected **column**. It can be seen that the "higher" the row (or the column) the more critical the fault is, since the faulty value affects more PEs.

It should be noted that this behavior is peculiar to this array type. In facts, it results from the "output-stationarity" of the array, since the weights are propagated perpendicularly to the activations. This is also why XwX experiments show this trend in the rows, while XiX experiments show the same trend in the columns.

C. Bit-wise resilience

We counted the number of malignant faults per each bit. Figure 6 shows the representative data for experiments 8XX, where bit 1 is the MSB. Figure 6b shows a clear descendent trend. It is important to underline that the values are 8-bit **unsigned** integers. It is easy to perceive that the less important a bit is, the less critical it is. Furthermore, more than 70% of the malignant faults are produced by a stuck-at-1 fault. A similar trend is shown in Figure 6a. The real difference is that the values in this case are two's complement **signed** integers, with bit 1 being the sign bit. In this case, the most critical bit is actually bit 2, but we noticed another trend: bit 1 is most critical with stuck-at-0 faults. Indeed, 67% of the faults in bit 1 comes from stuck-at-0 injections. In this case, this means that 1s are more common than 0s in bit 1 of the weights. Remember that when injecting a 0 in a bit whose value is always 0, there is no fault. Furthermore, we have to consider that this data is relative to the 7x7 array, thus the array is reused 16 times for each processed image.



(a) 8w7

(b) 8i14

(c) 16w28

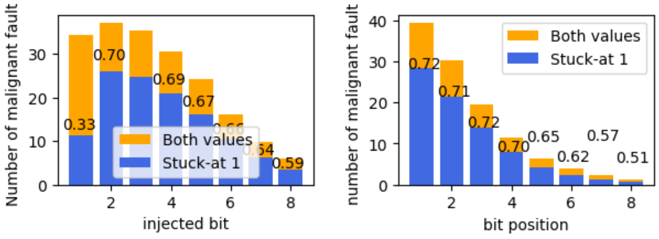
Fig. 5: Figures 5a and 5c show the number of malignant faults with respect to the injected row. The other figure shows the number of malignant faults with respect to the injected column.

Figure 7 shows the data for all the three experiments with 16bit representation. Data do not follow the same trend as the other experiments, and the criticality is especially correlated with the size of the array. Indeed, even if Figure 7a is basically the same as 6a, the percentage of stuck-at-1 faults in bit 1 is 46%, which is considerably higher than the 33% of the other experiment. Furthermore, Figures 7b and 7c show that bits 2 to 8 are equally critical. Additionally, there is not much difference between stuck-at-1 and stuck-at-0 injections for these bits. The figure shows that about half of the faults are caused by stuck-at-1 injections.

The main reason for this behavior has to be searched in the reuse of the array, as said before. This means that not only the resilience depends on the used dataset, but also in the way the data flows, which depends on the architecture. With this observation in mind, it would be interesting to investigate how different data-sets affect the behavior of the architecture, and that might be a direction for a future work.

VI. Conclusions

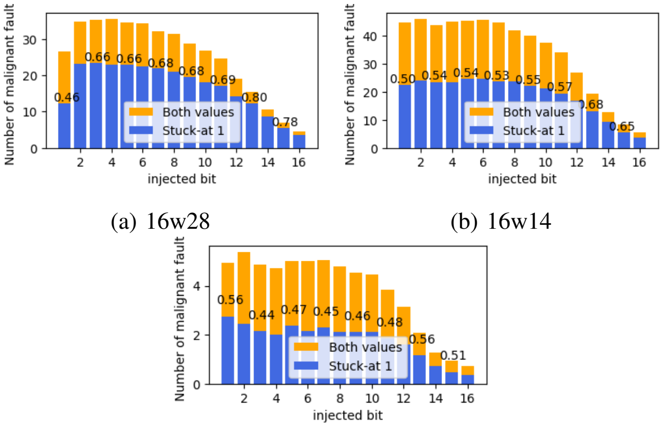
In this paper we showed how array size, precision and disturbances affect the resilience of the system. We concluded that, in general, neural networks seem to be highly resilient to disturbances in the input image, but not in weights. We also showed a negative correlation between array size and resilience, since the latter drops when the former increases. Furthermore, we underlined how higher precision can be actually worse for network resilience. Our experiments showed



(a) 8w7

(b) 8i7

Fig. 6: In yellow, number of malignant fault per bit normalized with respect to the number of injection per that bit. In blue, only the stuck-at-1 part of those faults. The superimposed number corresponds to the percentage of the blue bar w.r.t the yellow one.



(a) 16w28

(b) 16w14

(c) 16w7

Fig. 7: In yellow, number of malignant fault per bit normalized with respect to the number of injection per that bit. In blue, only the stuck-at-1 part of those faults.

that the 8-bit network is more resilient, masking a higher percentage of injected faults. This result might be further investigated in future works.

Finally, we showed that the resilience heavily depends on both the dataset and the architecture, since varying even just the array size, the resilience was greatly affected. Indeed, injecting a smaller systolic array (especially in experiments performed with a 16-bit architecture) resulted in more malignant faults than injecting a bigger one.

References

[1] S. Kundu, S. Banerjee, A. Raha, S. Natarajan, and K. Basu, "Toward functional safety of systolic array-based deep learning hardware accelerators," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 29, no. 3, pp. 485–498, 2021.

[2] E. Dupuis, S. Filip, O. Sentieys, D. Novo, I. O'Connor, and A. Bosio, "Approximations in deep learning," in *Approximate Computing Techniques*, pp. 467–512, Springer International Publishing, 2022.

[3] L. Wanhammar, "8 - dsp architectures," in *DSP Integrated Circuits*, Academic Press Series in Engineering, pp. 357–385, Burlington: Academic Press, 1999. Section 8.7.

[4] Y. Chen, Y. Xie, L. Song, F. Chen, and T. Tang, "A survey of accelerator architectures for deep neural networks," *Engineering*, vol. 6, no. 3, pp. 264–274, 2020.

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

[6] G. D. Natale, D. Gizopoulos, S. D. Carlo, A. Bosio, and R. Canal, eds., *Cross-Layer Reliability of Computing Systems*. Institution of Engineering and Technology, Oct. 2020.

[7] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *Proceedings of the 43rd Annual International Symposium on Computer Architecture*, (Seoul, South Korea), p. 367–379, 2016.

[8] nvidia, "Nvdla," 2021.

[9] J. J. Zhang, K. Basu, and S. Garg, "Fault-tolerant systolic array based accelerators for deep neural network execution," *IEEE Design Test*, vol. 36, no. 5, pp. 44–53, 2019.

[10] G. Li, S. K. S. Hari, M. Sullivan, T. Tsai, K. Pattabiraman, J. Emer, and S. W. Keckler, "Understanding error propagation in deep learning neural network (dnn) accelerators and applications," in *SC17: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–12, 2017.

[11] J. J. Zhang, T. Gu, K. Basu, and S. Garg, "Analyzing and mitigating the impact of permanent faults on a systolic array based neural network accelerator," in *2018 IEEE 36th VLSI Test Symposium (VTS)*, pp. 1–6, 2018.

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

[13] M. Eslami, B. Ghavami, M. Raji, and A. Mahani, "A survey on fault injection methods of digital integrated circuits," *Integration*, vol. 71, pp. 154–163, 2020.

[14] A. Ruospo, E. Sanchez, M. Traiola, I. O'Connor, and A. Bosio, "Investigating data representation for efficient and reliable convolutional neural networks," 2021.

[15] Z. Chen *et al.*, "Tensorfi: A flexible fault injection framework for tensorflow applications," in *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, (Coimbra, Portugal), pp. 426–435, IEEE, Oct. 2020.

[16] A. Mahmoud *et al.*, "PyTorchFI: A runtime perturbation tool for DNNs," in *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*.

[17] B. Salami, O. S. Unsal, and A. C. Kestelman, "On the resilience of RTL NN accelerators: Fault characterization and mitigation," in *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, (Lyon, France), pp. 322–329, IEEE, 2018.

[18] A. Ruospo, A. Balaara, A. Bosio, and E. Sanchez, "A pipelined multi-level fault injector for deep neural networks," in *2020 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pp. 1–6, 2020.

[19] F. Libano, P. Rech, and J. Brunhaver, "On the reliability of xilinx's deep processing unit and systolic arrays for matrix multiplication," in *2020 20th European Conference on Radiation and Its Effects on Components and Systems (RADECS)*, pp. 1–5, 2020.

[20] V. Sze, Y.-H. Chen, T.-J. Yang, and J. Emer, "Efficient Processing of Deep Neural Networks: A Tutorial and Survey," 2017. Publisher: arXiv Version Number: 2.

[21] CEA-LIST, "N2D2." [Online]. Available: https://github.com/CEA-LIST/N2D2.

[22] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, pp. 2278–2324, Nov. 1998.

[23] A. Ruospo, G. Gavarini, C. De Sio, J. D. Guerrero, L. Sterpone, M. Sonza Reorda, E. Sanchez, R. Mariani, J. Aribido, and J. Athavale, "Assessing convolutional neural networks reliability through statistical fault injections," in *2023 Design, Automation Test in Europe Conference*, 2023, In press.