HAL open science

Hardening a Neural Network on FPGA through Selective Triplication and Training Optimization

Wilfread Guilleme, Youri Helen, Rémy Priem, Angeliki Kritikakou, Cedric Killian, Daniel Chillet

To cite this version:

Wilfread Guilleme, Youri Helen, Rémy Priem, Angeliki Kritikakou, Cedric Killian, et al.. Hardening a Neural Network on FPGA through Selective Triplication and Training Optimization. RADECS 2024 - RADiation and its Effects on Components and Systems Conference, Sep 2023, Toulouse, France. pp.1-4. hal-04542185

HAL Id: hal-04542185

https://inria.hal.science/hal-04542185v1

Submitted on 11 Apr 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Hardening a Neural Network on FPGA through Selective Triplication and Training Optimization

Wilfread Guillemé*, Youri Helen†, Rémy Priem†, Angeliki Kritikakou*, Daniel Chillet* and Cédric Killian*
*Univ. Rennes, IRISA, INRIA, France †DGA, France

***Abstract*—The proposed approach identifies SEU sensitive flip-flops and optimizes the training phase of a neural network. With this information, selective triplication is applied to improve the reliability with limited resource overhead on an FPGA device.**

***Index Terms*—FPGA, Neural Network, SEU, TMR, VHDL.**

I. Introduction

EMBEDDED systems in critical applications have constraints in ensuring safe and reliable operation. They must be able to withstand harsh environments. The effects of ionizing radiation can cause errors in electronic circuits, compromising system operation. One example is given by the Single Event Upset (SEU) [1], which inverts a bit contained in a flip-flop. Modern electronic architectures are more and more sensitive to radiation, due to the decrease of the size of transistors [2]. Furthermore, radiation impact becomes worst with the increase of particles at higher altitude [3]. Meanwhile, Neural Networks (NNs) have proven to give very good results for critical applications, such as object recognition in images/videos, satellite, aerospace, and autonomous driving. Therefore, the protection of NN becomes crucial in critical modern embedded systems, as SEU can lead to prediction errors of the model, jeopardising its operation.

However, NNs comes at the price of an extremely large algorithmic complexity, thus requiring enormous computational power. To address these limitations, hardware accelerators dedicated to NNs are designed, able to perform calculations in parallel, accelerating the calculation time. Moreover, their custom design for a specific model allows complex calculations to be performed with satisfactory energy efficiency. Such hardware accelerators can be implemented on FPGAs. Flash technology is preferable to SRAM technology, since the configuration is stored in non-volatile memories, which are less sensitive to the effects of radiation. In order to improve the reliability of the accelerator, typical approaches for fault protection can be used, such as Triple Modular Redundancy (TMR) [4] and information redundancy by Hamming [5] and BCH [6] error correcting codes. However, such approaches are designed for traditional hardware, leading to prohibited overhead when applied to NNs.

To achieve reliable NNs with low overhead, we exploit an interesting property of NNs, i.e, they are intrinsically resilient to computational faults and approximations [7]. Exploiting this property, we are able to detect the sensitive flip-flops and protect only the most sensitive elements, saving hardware resources. More precisely, the proposed method characterizes the sensitivity of the flip-flops through fault injection based on their impact on the NN output, taking into account the set of inputs and the iterations considered during training. Then, selective protection is applied, where only the most sensitive flip-flops are triplicated. Our use-case is a multilayer perceptron [8] performing digit recognition, which is written in Python, implemented in VHDL and programmed on an FPGA target.

II. Related work

Several studies focus on the implementation of NNs on FPGAs to enhance their energy efficiency through parallelism and specialisation. For instance, a NN hardware implementation and its back-propagation training phase, to decrease the error rate over learning iterations, is presented in [9]. In the context of autonomous vehicles, the impact of radiation on an SRAM-based FPGA is investigated and a hardening technique is proposed, masking up to 40% of faults with 8% additional overhead, by protecting the most sensitive layer of the NN [10]. An error correction approach using Hamming codes and protection by triplication on parameters, such as weight and bias of neurons, against SEUs in different NN architectures is presented in [11].

Compared to the existing approaches, this work applies selective triplication of flip-flops to all neurons across all layers, coupled with the optimization of the training phase for a NN hardware accelerator.

III. Proposed methodology

A. Selective protection

This section presents our methodology for selective fault tolerance, based on fault injection at neuron level and detection of sensitive flip-flops, and thus, sensitive layers. The overview of the sensitivity analysis is described by Algorithm 1. The approach is based on sequentially injecting faults at all the flip-flops of a neuron, for all neurons, considering different inputs in a dataset. The output is a table describing the sensitivity of each flip-flop.

To perform the sensitivity analysis, the first step is to design a fault injection mechanism at neuron level. Figure 1 illustrates the functionality of an artificial neuron, which is based on a combinatorial logic block, that performs the multiplication-accumulation operation between the inputs X_i and the weights W_i, with a bias B added. The result is stored in a register before being sent to the activation function. To inject a fault at the neuron level, we need to be able to insert the impact of

Algorithm 1 Sensitivity analysis for NN flip-flops

$i = 0$ Input Selection
$n = 0$ Neuron Selection
$f = 0$ Flip-Flop Selection

for $i = 0...Num_Inputs$ **do**
  for $n = 0...Num_Neurons$ **do**
    for $f = 0...Num_FlipFlops$ **do**
      $Original_NN = Calculate_NN(i)$;
      $Fault_Injection(n, f)$;
      $Extended_NN = Calculate_NN(i)$;
      if $Original_NN \neq Extended_NN$ **then**
        $Sensitive_Flip_Flops(n, f) + +$;
      end if
    end for
  end for
end for
$Calculate_sensitivity_level()$;

a SEU to the neuron. This is achieved through the mechanism illustrated in Figure 2. A multiplexer, controlled by the "Fault Injection" signal, is used to select the flip-flop, where the fault will be injected, and an inversion gate is used to change the value to be stored.

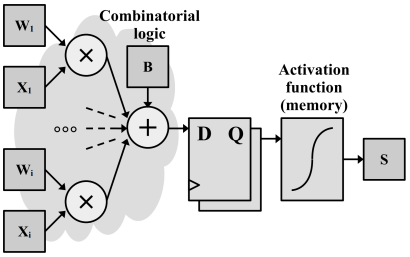

Fig. 1. Functional diagram of an artificial neuron.

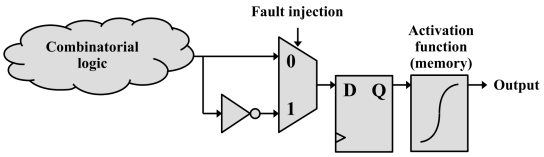

Fig. 2. Fault injection mechanism at the neuron level.

As shown in Figure 3, the original NN is extended with the above fault injection mechanism for all neuron flip-flop. Then, the extended NN and the original NN are executed in parallel. The original model performs a fault-free execution, providing the golden results, whereas the extended model is executed under faults. The prediction results, obtained by the two NNs, are compared. If the predictions, provided by the models for all inputs, are the same, the flip-flop, where the fault is injected, did not cause any model prediction error. Therefore, it is considered as non-sensitive. However, if the outputs of the two NNs are different for an identical input data, the selected flip-flop has caused a prediction error of the model, and thus, it is considered as a sensitive one.

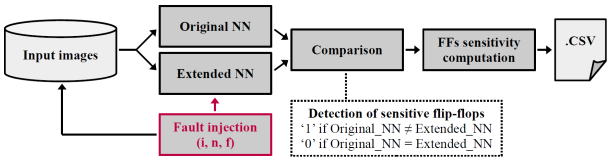

Fig. 3. Hardware implementation of sensitive flip-flop detection.

After the fault injection campaign, we obtain histogram with how many times a flip-flop has been considered sensitive during the fault injection campaign. The last step it to compute the sensitivity level for each flip-flop by calculating the ratio of the number of times a flip-flop has been detected as sensitive to the total number of inputs. For instance, a flip-flop detected as sensitive on all inputs during sensitivity analysis will have a sensitivity level of 1, while a flip-flop detected as sensitive on half of the inputs, it will have a sensitivity level of 0.5. The sensitivity levels are used to prioritize the flip-flops to be protected, i.e., the higher the sensitivity level of a flip-flop, the higher its priority.

B. Training phase optimisation

The NN may have different fault tolerance at different moments during training, as it depends on the characteristics of the input data and the learning parameters. We explore this characteristic by applying the aforementioned sensitivity analysis in different moments during training with the goal of identifying the most fault tolerant NN. In this way, the proposed approach identifies the NN with the maximization inherited resiliency, and thus, minimizes the resource overhead of selective triplication.

IV. Experimental evaluation

A. Use-case

Our use-case is a multilayer perceptron model that recognizes handwritten numbers. Figure 4 presents the architecture of the studied multilayer perceptron. The input layer of the model is size 64. The model consists of an intermediate layer with 64 neurons and an output layer with 10 neurons to represent the numbers from 0 to 9.

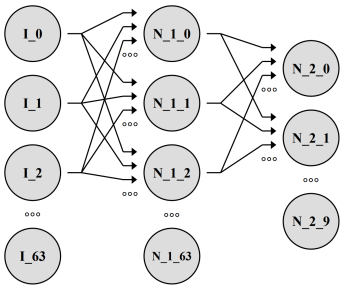

Fig. 4. Use-case: Multilayer perceptron model.

The multilayer perceptron is developed with Keras/TensorFlow using a modified MNIST database.

The original dataset consists of 70,000 handwritten digit images, 60,000 of which are used for training the model and 10,000 for testing. These images, originally of size 28x28 in shades of gray were modified into 8x8 images in black and white for FPGA resource reasons, they constitute the 64 inputs of the NN. During training, the weights and biases of the model are estimated.

Note that, the data and parameters of the Python model are real and signed numbers, instantiated in single or double precision floating point format, encoded on 32 bits and 64 bits respectively. Such an encoding is not optimal for an FPGA implementation, where the hardware resources are much more limited. In this case, the data can be represented by the SFIXED type, which represents a signed real number and allows a customized data format. In the SFIXED type, the most significant bit is the sign, the following bits correspond to the integer part and the last bits to the decimal part. The SFIXED format used in our implementation consist of 10 bits, with 1 bit for the sign, 4 bits for the integer part and 5 bits for the decimal part, as shown in Figure 5.

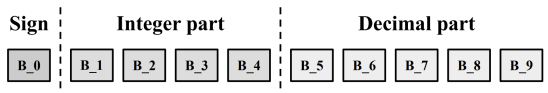

Fig. 5. SFIXED data format studied.

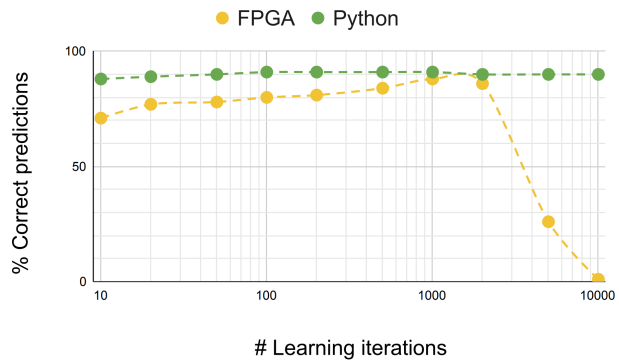

Fig. 6. Comparison of performance between FPGA and Python models.

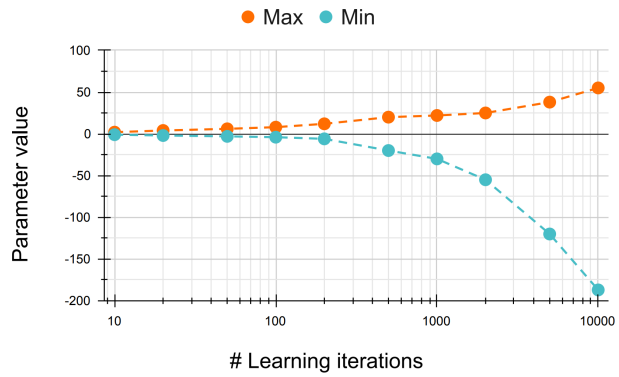

Fig. 7. Divergence of neural network parameters over learning iterations.

B. Experimental results

In this section, we first present the accuracy comparison between the Python and the FPGA models. Then, we show the sensitivity analysis results for a given number of training iterations. Last, we analyse the impact of the training iterations in the model protection and the area overhead introduce from the proposed selective protection.

1) Comparison of FPGA and Python models: Initially, we compare the accuracy of the Python model and the FPGA model based on the SFIXED format with respect to the training iterations. As shown in Figure 6, the FPGA model has a reduced number of correct predictions than the Python model, when the number of training iterations is low. However, as the training progresses, the FPGA model converges to the performance of the Python model, reaching up to 85% correct predictions, where the Python model obtained 90%. Beyond 2,000 training iterations, the FPGA model experienced a significant decrease in accuracy, which is not the case for the Python model.

The observed decrease in accuracy is related to the divergence of the NN parameters during the training phase, as shown in the Figure 7. The parameter values reach levels that the SFIXED format can no longer represent, which explains the accuracy decrease of the FPGA model. On the other hand, up to about 2000 training iterations, the FPGA model improves its accuracy, as the parameter values fully utilize the scale of the SFIXED format.

2) Sensitivity analysis: Table I presents the sensitivity results of the neuron flip-flops and the multilayer perceptron layers, applying the sensitivity analysis of Section III-A. We performed fault injection in all 740 flip-flops of the model, considering 20 images. The model has performed 1,000 training iterations with a size of batch equal to 32. The obtained results show that the flip-flops containing the sign bits are the most sensitive ones. This is due to the fact that the inversion of the sign or a high value bit drastically changes the input data for the activation function and strongly impacts the output result of the neuron. Furthermore, we observe that the output layer is much more sensitive than the intermediate layer. The reason is that the output layer consists of a smaller number of neurons which define the prediction result of the model.

TABLE I
SENSITIVITY LEVEL OF DIFFERENT FLIP-FLOPS.

	Sign	Integer part				Decimal part				
Neuron	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8	B_9
N_1_0	0.1	0.05	0.1	0	0.05	0.05	0	0	0	0
N_1_1	0.2	0	0.05	0	0	0	0	0	0	0
N_1_2	0.25	0.05	0	0	0	0	0	0	0	0
N_1_3	0.1	0.05	0.05	0.05	0	0	0	0	0	0
...										
N_1_63	0.1	0.1	0.05	0	0	0	0	0	0	0
N_2_0	0.55	0	0.05	0	0	0	0	0	0	0
N_2_1	0.4	0.1	0.1	0	0.1	0	0	0	0	0
N_2_2	1	0.05	0.1	0	0	0	0	0	0	0
N_2_3	0.9	0.1	0.15	0.05	0.05	0	0	0	0	0
N_2_4	0.6	0.1	0.05	0	0	0	0	0	0	0
N_2_5	0.95	0.1	0.05	0.05	0.05	0.05	0	0	0	0
N_2_6	0.5	0.05	0.05	0.05	0	0	0	0	0	0
N_2_7	0.8	0.05	0	0.1	0.1	0	0	0	0	0
N_2_8	0.85	0.15	0.1	0.1	0.1	0.05	0	0	0	0
N_2_9	1	0	0.1	0	0	0	0	0	0	0

3) Training phase optimisation: The sensitivity analysis, and thus, the NN fault tolerance depends on the training phase. To explore this impact, we have generated several NNs,

considering different number of training iterations. Figure 8 depicts the number of errors detected during the sensitivity analysis of the NN at different training iterations. The grey curve corresponds to the error detection using the training images, while the purple curve represents the error detection using the validation images. Based on the obtained results, the fault tolerance of the NN increases with the number of training iterations. Furthermore, the NN is more fault tolerant on training images than on validation images. Therefore, the model is more resilient when it is exposed to an environment for which it is designed. Note that, the most resilient NN model is the one with 1,000 training iterations.

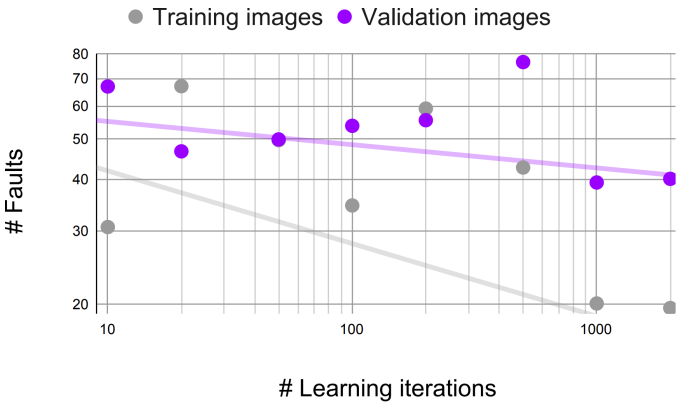

Fig. 8. Number of faults detected by the test scenario.

Last, but not least, Figure 9 compares the number of flip-flops among the original, and thus, unprotected model, the proposed approach based on selective triplications and a typical Triple Modular Redundancy (TMR) method, considering different number of training iterations. The unprotected model consist of 74 artificial neurons each composed of 10 flip-flops, leading to 740 flip-flops. The TMR approach applies triplication of all flip-flops, leading to 2,220 flip-flops. The proposed approach significantly reduces the number of required flip-flops, while keeping the accuracy. For instance, 1,200 flip-flops are required for the model that underwent 1,000 training iterations, leading to a 45% decrease compared to the typical TMR approach.

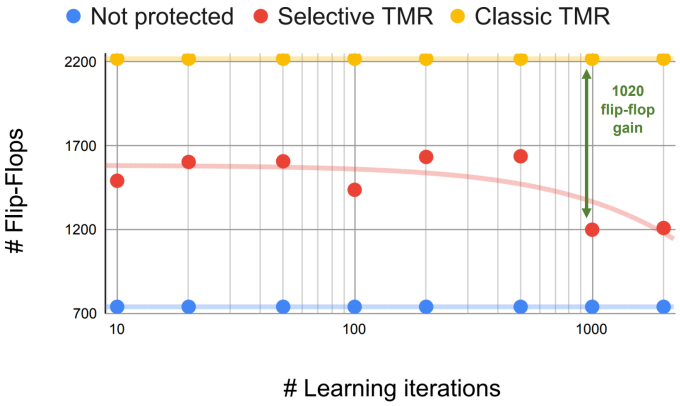

Fig. 9. Cost in flip-flops of selective and classic triplication.

V. Conclusion and Discussion

This work presents a novel approach to protect a neural network accelerated on an FPGA, based on sensitivity analysis and selective triplication of the most sensitive flip-flops combined with optimization of the training phase. Through sensitivity analysis, we are able to identify the most critical flip-flops and prioritize their protection. As a result, not only the protection efficiency is increased, but also the power consumption and area overhead is reduced, making it an efficient protection approach for NN requiring high reliability in harsh environments, such as in space and aviation domains. The obtained results show that the fault sensitivities of NN layers are not identical, with the last layer being the most sensitive. It significantly reduces the hardware cost of protecting the NN model, while maintaining its protection level.

As future work, we will explore how to leverage the proposed approach for large neural networks, where the number of flip-flops is significantly increased. To deal with this issue, we will explore fault injection mainly on the flip-flops that are expected to be sensitive to faults based on the used format, e.g., flip-flops containing the sign and the most significant bits (MSBs), and combine the proposed approach with statistical fault injection.

Acknowledgments

This work is supported by the *Direction Générale de l'Armement*, financed by the *Agence de l'Innovation de Défenses* and the national institute of research in digital sciences and technologies *"INRIA"*.

References

[1] P. E. Dodd and L. W. Massengill, "Basic mechanisms and modeling of single-event upset in digital microelectronics," *IEEE Transactions on nuclear Science*, vol. 50, no. 3, pp. 583–602, 2003.

[2] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers, "The impact of technology scaling on lifetime reliability," in *International Conference on Dependable Systems and Networks, 2004*. IEEE, 2004, pp. 177–186.

[3] S. P. Assurance, "Techniques for radiation effects mitigation in aasic and fpgas handbook," Technical Report. ESA Requirements and Standards Division, Tech. Rep., 2016.

[4] M. Berg and K. A. LaBel, "Verification of triple modular redundancy (tmr) insertion for reliable and trusted systems," in *2016 MRQW Microelectronics Reliability and Qualification Working Meeting*, no. GSFC-E-DAA-TN29375, 2016.

[5] K. Furutani, K. Arimoto, H. Miyamoto, T. Kobayashi, K. Yasuda, and K. Mashiko, "A built-in hamming code ecc circuit for drams," *IEEE Journal of Solid-State Circuits*, vol. 24, no. 1, pp. 50–56, 1989.

[6] R. Chien, "Cyclic decoding procedures for bose-chaudhuri-hocquenghem codes," *IEEE Transactions on information theory*, vol. 10, no. 4, pp. 357–363, 1964.

[7] A. Bosio, P. Bernardi, A. Ruospo, and E. Sanchez, "A reliability analysis of a deep neural network," in *2019 IEEE Latin American Test Symposium (LATS)*, 2019, pp. 1–6.

[8] M. W. Gardner and S. Dorling, "Artificial neural networks (the multilayer perceptron)—a review of applications in the atmospheric sciences," *Atmospheric environment*, vol. 32, no. 14-15, pp. 2627–2636, 1998.

[9] S. Li, K. Choi, and Y. Lee, "Artificial neural network implementation in fpga: A case study," in *2016 International SoC Design Conference (ISOCC)*. IEEE, 2016, pp. 297–298.

[10] F. Libano, B. Wilson, J. Anderson, M. J. Wirthlin, C. Cazzaniga, C. Frost, and P. Rech, "Selective hardening for neural networks in fpgas," *IEEE Transactions on Nuclear Science*, vol. 66, no. 1, pp. 216–222, 2018.

[11] Z. Yan, Y. Shi, W. Liao, M. Hashimoto, X. Zhou, and C. Zhuo, "When single event upset meets deep neural networks: Observations, explorations, and remedies," in *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2020, pp. 163–168.