

American University in Cairo

AUC Knowledge Fountain

Theses and Dissertations

Student Research

Winter 1-31-2025

Design and Implementation of UVM-based Verification Framework for Deep Learning Accelerators

Randa Ahmed Hussein Aboudeif
aranda@aucegypt.edu

Follow this and additional works at: <https://fount.aucegypt.edu/etds>



Part of the [Computer and Systems Architecture Commons](#), [Digital Circuits Commons](#), [Electrical and Electronics Commons](#), and the [Hardware Systems Commons](#)

Recommended Citation

APA Citation

Aboudeif, R. (2025). *Design and Implementation of UVM-based Verification Framework for Deep Learning Accelerators* [Master's Thesis, the American University in Cairo]. AUC Knowledge Fountain.
<https://fount.aucegypt.edu/etds/2383>

MLA Citation

Aboudeif, Randa Ahmed Hussein. *Design and Implementation of UVM-based Verification Framework for Deep Learning Accelerators*. 2025. American University in Cairo, Master's Thesis. *AUC Knowledge Fountain*.
<https://fount.aucegypt.edu/etds/2383>

This Master's Thesis is brought to you for free and open access by the Student Research at AUC Knowledge Fountain. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of AUC Knowledge Fountain. For more information, please contact thesisadmin@aucegypt.edu.



Graduate Studies

*Design and Implementation of
UVM-based Verification Framework for
Deep Learning Accelerators*

A THESIS SUBMITTED BY

Randa Ahmed Hussein Aboudeif

TO THE

*Electronics and Communications Engineering Department
School of Sciences and Engineering (SSE)*

SUPERVISED BY

*Prof. Dr. Yehea Ismail
Dr. Mohamed AbdelSalam*

August, 2024

*in partial fulfillment of the requirements for the degree of Master of Science in
Electronics and Communications Engineering*

Declaration of Authorship

I, Randa Ahmed Hussein Aboudeif, declare that this thesis titled, "Design and Implementation of UVM-based Verification Framework for Deep Learning Accelerators" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Randa Aboudeif

Date:

8/8/2024

**The American University in Cairo
School of Sciences and Engineering**

Design and Implementation of UVM-based Verification Framework for Deep Learning Accelerators

A Thesis Submitted by

Randa Ahmed Hussein Aboudeif

To the

Master of Science in Electronics and Communications Engineering


August 8th, 2024

In partial fulfillment of the requirements for the degree of

Master of Science in Electronics and Communications Engineering

Has been approved by

Dr. Yehea Ismail (Advisor)
Professor, Chair of Electronics and Communications Engineering Dept.
Director of Center of Nanoelectronics and Devices
American University in Cairo



Dr. Hassanein Amer (Internal Examiner)
Professor of Electronics and Communications Engineering Dept.
American University in Cairo



Dr. Ahmed Saeed Sayed (External Examiner)
Associate Professor, Electronics and Communications,
FUE.



Dr. Sherif Abdelazeem (Moderator)
Professor of Electronics and Communications Engineering Dept.
Graduate Program Director
American University in Cairo


Graduate Program Director

8/8/2024
Date

Dean

Date

Abstract

Recent advancements in deep learning (DL) have made hardware accelerators, known as deep learning accelerators (DLAs), a preferred solution for numerous high-performance computing (HPC) applications, including speech recognition, computer vision, and image classification. Moreover, the next generation of HPC electronic control unit (ECU) designs requires specialized DLAs for low power consumption and resource efficiency. Deep neural networks (DNNs) demonstrated outstanding performance in learning-enabled autonomous systems (LEAS) for essential autonomous driving functions in these ECUs, like perception.

DLAs are composed of hundreds of parallel processing engines to speed up computations and can gain access to pre-trained networks from the cloud or through on-chip memory to implement the DNN inference process. DLA verification is becoming an important and challenging phase. The requirement from the verification process has been to handle the complex DLA design and to figure out and fix the design issues that appeared due to the large computations involved in each DNN layer mapped on hardware. Moreover, the reliability of DLAs is critical for assessment as they are involved in safety-critical applications, especially with the noticeable increase in sensor faults, adversarial attacks, and hardware functional errors occurring in DLAs, resulting in violations of safety and reliability requirements. Those challenges in DLA design indicate the need for a robust, self-checking, and powerful verification methodology with error injection and detection capabilities to tackle those challenges.

In our thesis, a novel, scalable, reusable, and efficient verification framework for deep learning hardware accelerators using the Universal Verification Methodology (UVM) is introduced. The proposed framework is to create a scalable and reusable UVM verification testbench for testing deep learning accelerators with simulation, emulation, and Field Programmable Gate Array (FPGA) prototyping by running different testing scenarios for convolutional neural networks (CNNs) with multiple configurations. Each test scenario configures the DLA to run a specific CNN and drives the input data, pretrained weight, and bias required for each layer. The proposed framework is applicable to different DNN architectures, e.g., ALEXNet, GoogLeNet, etc. Therefore, the proposed framework makes the DLA verification process faster, easier, and more reliable. It provides large coverage, can hit corner cases, and has complete access to changing the DLA configurations through a generic verification environment with components that can be easily reused to test different DLA designs.

Moreover, the proposed framework has a scalable error injection methodology for testing the trustworthiness of deep learning accelerators with simulation, emulation, and FPGA prototyping. The proposed methodology applies error injection using three mechanisms. Firstly, error injection is applied to the DNN data path by corrupting each DNN layer's feature map, weight, and bias. Secondly, adversarial attacks on the input image are applied to mitigate the perturbation in input received from cameras and other sensors in the physical world. Thirdly, the proposed

methodology applies error injection to the DLA hardware configurations to detect faulty hardware configurations for a DNN that may disrupt the inference process. Moreover, the proposed error injection methodology is reliable and has complete access to the DNN data path between layers and the DLA configurations. In addition to that, it is applicable to different DNN architectures.

The Nvidia deep learning accelerator (NVDLA) is used as an example of a DLA design under verification to prove the robustness of our proposed verification framework. Our verification framework is applicable to run any sophisticated custom and standard CNN architectures as per the CNN sequence for programming the DLA hardware. As a case study, our verification framework is applied to the single convolution layer and the LeNet-5 CNN. The verification framework has less simulation and emulation runtime, as the proposed framework simulation runtime for a single CNN layer is reduced by 17x compared to that of the NVDLA trace-player direct testbench. Moreover, the simulation runtime for the LeNet-5 is reduced by more than 200x using our framework compared to that using the NVDLA software. Similarly, the LeNet-5 CNN emulation runtime for our framework is reduced by more than 10x compared to that for the NVDLA software. Additionally, the proposed framework LeNet-5 CNN test case runtime is reduced by more than 3x for emulation compared to simulation.

Regarding error injection, the cross-layer error injection in the LeNet-5 CNN indicates that the LeNet-5 CNN is more sensitive to the internal layers' multiple values of input data, weight, and bias corruption and less sensitive to single-value corruption in data, weight, and bias propagation between layers. Furthermore, the LeNet-5 CNN accuracy rate decreases with high values of perturbation factor using the fast gradient sign method for input image error injection. Furthermore, compared to multiple weight errors, the convolution layer is more vulnerable to multiple data errors than a single error. The existence of the ReLU activation function in the convolution layer almost masks data and weights single-value errors.

Acknowledgements

Above all, I am truly grateful to the Almighty God for His sustenance, grace, and strength. His kindness has enabled me to succeed in every academic endeavor I have undertaken. I express my gratitude to him for giving me this chance and giving me the capacity to move forward with success. I am appreciative of the gifts of happiness, difficulties, and growth opportunities that I have experienced throughout my life, not just during this research project.

My deepest gratitude goes out to my supervisors, Prof. Dr. Yehea Ismail and Dr. Mohamed AbdelSalam, for their support and for giving me the chance to complete my master's thesis. I value the time and ideas they have contributed to make my work interesting and productive. Their insightful advice, insights, and mentorship inspired me to keep learning more every day. I benefited from their profound insights at different points during my research. Once again, many thanks to them, since without them, this work would have never seen the light as it is today.

I would like to thank my colleague, Eng. Tasneem Awaad, for her support and helpful suggestions on this work. One of the many important contributions to this research has been your support.

I would like to thank Dr. Ahmed Awaad for his support and encouragement over the past few months.

I would like to extend my sincere gratitude to my parents and my brothers, mainly my mother, Prof. Dr. Sayeda Salem, and my brother, Dr. Khaled Aboudeif, for their unwavering support and encouragement in helping me stay inspired to work harder and more productively throughout my life. You have always been an inspiration and a source of strength for me, especially when I had no one to turn to for advice or support.

Contents

Abstract	iv
Acknowledgements	vi
Contents	vii
List of Figures	xi
List of Tables	xiii
List of Abbreviations	xiv
1 INTRODUCTION	1
1.1 State of The Art	1
1.1.1 GPU	1
1.1.2 Deep Learning Hardware Acceleration	2
1.2 Problem Statement and Motivation	3
1.3 Research Objective	4
1.4 Contributions	4
1.5 General Thesis Outline	5
2 BACKGROUND AND RELATED WORK	6

2.1	Background	6
2.1.1	Convolutional Neural Networks	6
2.1.1.1	Convolutional Layer	7
2.1.1.2	Pooling Layer	8
2.1.1.3	Fully Connected Layer	8
2.1.2	Fault Injection for Deep Neural Networks	9
2.1.3	Deep Learning Hardware Accelerators	12
2.1.3.1	Eyeriss Deep Learning Accelerator	12
2.1.3.2	Flex Logix InferX X1 Accelerator	13
2.1.3.3	NVDLA	13
2.1.3.3.1	Small NVDLA	14
2.1.3.3.2	Large NVDLA	14
2.1.3.3.3	NVDLA Hardware Architecture	15
2.1.3.3.4	NVDLA Software	17
2.2	Related Work	18
3	PROPOSED UVM FRAMEWORK FOR DLA VERIFICATION	22
3.1	Input Data and Weight Preparation	24
3.2	UVM Testbench Architecture	24
3.2.1	HDL Top	26

3.2.2	HVL Top	26
3.2.2.1	CNN Test	27
3.2.2.2	CNN Virtual Sequence	27
3.2.2.3	CNN Environment	28
3.2.2.4	CNN Layer Agent	28
3.2.2.5	AXI Environment	29
3.2.2.6	Reference Model	29
4	PROPOSED ERROR INJECTION METHODOLOGY FOR DLA VERIFICATION	30
4.1	DNN Data Path Error Injection	31
4.2	DNN Input Image Error Injection	32
4.3	DLA Hardware Registers Error Injection	33
5	NVDLA CASE STUDY	35
5.1	NVDLA Integration	35
5.2	NVDLA Registers Configuration	36
5.3	NVDLA Data and Weight Memory Mapping	37
5.3.1	Input Image Mapping	38
5.3.2	Feature Data Mapping	38
5.3.3	Weight Mapping	39

6	EXPERIMENTAL RESULTS	40
6.1	Single CNN Convolution Layer Results	40
6.2	LeNet-5 CNN Results	41
6.3	Emulation Analysis	45
6.4	Error injection in CNN Results	45
6.4.1	DNN Data Path Error Injection	46
6.4.2	Input Image Error Injection	47
7	CONCLUSION AND FUTURE WORK	50
7.1	Contributions	50
7.2	Future Work	51
	References	52

List of Figures

2.1	Illustration of a single convolutional layer [13].	8
2.2	Max pooling [12].	9
2.3	Eyeriss deep learning accelerator architecture [25].	13
2.4	Small NVDLA versus large NVDLA system [28].	15
2.5	NVDLA hardware architecture [28].	16
2.6	NVDLA interface [30].	17
2.7	NVDLA system software dataflow [28].	18
2.8	NVDLA integration on RISC-V SoC [32].	19
2.9	The NN accelerator RTL model [35].	20
3.1	The Veloce Strato emulator [37].	22
3.2	The Veloce proFPGA platform [38].	23
3.3	Input data and weight pre-processing.	24
3.4	UVM testbench architecture.	25
3.5	UVM testbench timed HDL and untimed HVL partitioning.	26
4.1	UVM testbench architecture with the error injection methodology.	31
4.2	Input data and weight pre-processing with input image error injection.	33
5.1	NVDLA integration with the UVM testbench architecture.	35

5.2	NVDLA ping-pong register file [51].	37
5.3	Packed feature data [52].	38
5.4	DC one group weight mapping [52].	39
6.1	16-bit binary number [56].	41
6.2	Proposed framework LeNet-5 CNN test case simulation versus emulation wall time.	44
6.3	Proposed framework LeNet-5 CNN test case code coverage analysis.	44
6.4	NVDLA software environment LeNet-5 CNN code coverage analysis.	44
6.5	Single CNN convolution layer data path error injection testing scenarios error rate.	46
6.6	LeNet-5 CNN data path error injection testing scenarios error rate.	47
6.7	The accuracy rate of the LeNet-5 CNN with adversarial images. . .	48

List of Tables

4.1	DNN data path error injection testing scenarios.	32
6.1	The single CNN convolution layer parameters.	40
6.2	The single CNN convolution layer simulation and emulation runtime.	41
6.3	The LeNet-5 CNN parameters.	42
6.4	LeNet-5 output for digit 8 input image.	42
6.5	LeNet-5 output for digit 2 input image.	43
6.6	LeNet-5 simulation and emulation runtime	43
6.7	Emulation analysis summary.	45
6.8	The proposed framework versus the NVDLA software environment .	49

List of Abbreviations

API	Application Programming Interface
AI	Artificial Intelligence
APB	Advanced Peripheral Bus
AVB	Advanced Verification Board
BFM	Bus Functional Model
CNN	Convolutional Neural Network
CPU	Control Processing Unit
DC	Direct Convolution
DNN	Deep Neural Network
DL	Deep Learning
DLA	Deep Learning Accelerator
DRAM	Dynamic Random Access Memory
DUT	Design Under Test
ECU	Electronic Control Unit
EDA	Electronic design automation
FGSM	Fast Gradient Sign Method
FI	Fault Injection
FPGA	Field Programmable Gate Array
GPU	Graphics Processing Unit
KB	Kilo Byte
HA	Hardware Accelerator
HAV	Hardware-Assisted Verification
HDL	Hardware Description Language
HLS	High Level Synthesis
HVL	Hardware Verification Language
HPC	high-performance computing
IoT	Internet of Things
IVM	Image Verification Model
KMD	Kernal Mode Driver
KPIs	Key Performance Indicators

MAC	Multiplication and Accumulation
ML	Machine Learning
NN	Neural Networks
NVDLA	Nvidia Deep Learning Accelerator
1D	one-dimensional
ONNX	Open Neural Network Exchange
OOP	Object-Oriented Programming
OVM	Open Verification Methodology
RGBA	Red Green Blue Alpha
RGB	Red Green Blue
RTL	Register-Transfer Level
SCE-MI	Standard Co-Emulation Modeling Interface
SDCs	Silent Data Corruptions
SoC	System on Chip
SRAM	Static Random Access Memory
SV	SystemVerilog
SWiFI	Software-implemented Fault Injection
TACs	Testbench Acceleration Compilers
TBX	Testbench-Xpress
3D	three-dimensional
TLM	Transaction Level Model
UMD	User Mode Driver
UVM	Universal Verification Methodology
VMM	Verification Methodology Manual

Chapter 1

Introduction

In this chapter, an overview of the thesis is introduced. We begin with an introduction to deep learning hardware acceleration. We state the problem statement, motivations, and objectives of our work. Then we highlight our contributions. Finally, we conclude this chapter with a general thesis outline.

1.1 State of The Art

Deep neural networks (DNNs) have demonstrated exceptional performance in many applications, particularly in learning-enabled autonomous systems (LEAS) for essential autonomous driving tasks like perception implemented in high-performance computing (HPC) electronic control units (ECUs). DNNs may extract complex characteristics from raw data by using deep learning (DL) algorithms on large datasets. This results in an effective input space representation. The groundbreaking research by Geoffrey Hinton in 2012 [1] inspired the development of deep learning since deep neural networks outperformed conventional machine learning (ML), computer vision, and feature engineering algorithms in the classification of images. Artificial intelligence (AI) is currently enabling endless capabilities for both the present and the future. In particular, the DNN architecture laid the groundwork for its current applications in object detection, speech, and face recognition.

1.1.1 GPU

The graphics processing unit (GPU) was first developed as a processor to control the performance of graphics and video on computers. Currently, GPUs have reached high performance and are used for more complex computational tasks. They are general-purpose designs with thousands or hundreds of cores that can simultaneously process thousands or hundreds of threads. Since deep learning algorithms are applied to large amounts of data, they allow for an effective representation of the input space. This requires complex computations and high memory requirements, making it challenging for computing platforms to achieve real-time performance combined with energy efficiency [2]. Consequently, to achieve high throughput and speed up DNNs during the training and inference stages, GPUs are employed as hardware

accelerators. However, their excessive power consumption is frequently costly for systems with limited resources.

1.1.2 Deep Learning Hardware Acceleration

As depicted, GPUs have low predictability of time execution, which is a crucial necessity in safety-critical systems like autonomous vehicles. The Field Programmable Gate Array (FPGA) is a promising GPU substitute. These platforms facilitate the implementation of hardware accelerators (HAs) targeted at specific problems, satisfying the memory and computational requirements for complex DNNs while preserving effective power consumption. Several deep learning applications demonstrated the power efficiency, low latency, and timing predictability of FPGA-based platforms [3].

New high-performance computing applications, including speech recognition, computer vision, and image classification, besides the HPC ECU designs that require the exploitation of neural network algorithms, need a specific architecture where resource efficiency and low power consumption are essential. The most highly computational element of neural networks (NN) is training, which is often carried out online on a GPU to produce a model that has already been trained. Consequently, pre-trained networks can be accessed from cloud storage, or an on-chip memory linked to hardware accelerators for inference. In general, a physical chip that speeds computations is referred to as a Deep Learning Accelerator (DLA). A convolutional neural network (CNN) is a type of DNN that contains a convolutional layer. Moreover, a CNN may also contain pooling and fully connected layers. Because the convolutional and fully connected layers are computationally intensive, these accelerators are mostly focused on them. These specific layers are mostly composed of multiplication and accumulation (MAC) processes, which are readily parallelizable. As a result, the first implemented hardware inference accelerators carried out parallel processing across a range of processing elements by taking advantage of the intrinsic spatial parallelism of DNNs [4]. These accelerators used data-flow architectures that focused on highly parallel processing paradigms and effectively mapped convolution operations in hardware. Consequently, the customized accelerators outperformed the conventional control processing units (CPUs) and GPUs in terms of efficiency. However, data transfers and memory access were the main bottlenecks for these implementations [5]. Previous designs of accelerators implied that data transmission required more energy than the computations themselves.

As a result, the next stage of inference accelerator development concentrated on energy-efficient designs that minimized memory overheads and increased data reuse. Deep compression techniques like quantization and low precision were first used by new DLA architectures to reduce memory traffic at the expense of accuracy [5]. These compression methods take advantage of the inherent error robustness of neural networks. The currently used DNN algorithms were optimized at both the hardware and algorithmic levels in the current accelerator developments [3]. Therefore, the specific methods produced improved energy efficiency and performance.

1.2 Problem Statement and Motivation

DLA verification is growing in importance and complexity and is becoming more challenging. Handling the complex DLA design and identifying and resolving design issues resulting from the massively parallel MAC operations involved in each DNN layer mapped on hardware have been necessitated by the verification process. Since DLAs are used in safety-critical applications and there has been an apparent increase in sensor failures, adversarial attacks, and hardware functional errors that violate safety and reliability standards, the reliability of DLAs is essential for assessment. Therefore, these challenges associated with DLA design point to the necessity of a strong and effective verification methodology.

Universal Verification Methodology (UVM) is a widely used, highly portable open-source library that relies on contributions from various verification approaches, such as the Verification Methodology Manual (VMM) and the Open Verification Methodology (OVM). The unending capacity for evaluating and verifying any advanced digital structure implementation was demonstrated by the UVM testbench architecture [6]. Any digital design implementation can be verified because of the UVM testbench architecture's efficient and well-proven infrastructure. Verification has evolved into a double-edged sword recently; it takes up the majority of the design cycle time yet is still necessary. The appeal of UVM modularity and reusability of verification components was that they provided a stronger foundation for long-term design and verification [7].

Any UVM testbench can be divided into three main parts: the test, which runs test scenarios outlined in the test plan; the sequence, which serves as the test's stimulus and the input of the Design Under Test (DUT); and the environment, which initiates the sequence and monitors the DUT's response to verify its functionality [8]. Extensive simulation and hardware emulation techniques are required to verify such sophisticated DLA designs, and these approaches are essential for evaluating the correctness and high-quality performance of such complex DLA architectures. These architectural designs are represented in software routine forms in the software-based simulation. Nevertheless, there are a number of drawbacks to this method, including its comparatively slow speed as design complexity rises. However, such architectural designs are expressed in Hardware Description Languages (HDLs) for hardware emulation, which can help run faster and identify design and architectural issues early. Co-simulation combined with the transaction-level methodology (co-emulation) [9] is an extraordinary method of emulation that is exploited by emulation systems such as Veloce Strato [10]. In this method, the testbench's transactor interfaces with the DUT run on the emulator through the testbench test that runs on the host machine. Using just simulated test scenarios to assess and validate different DLA designs has gotten difficult. Therefore, the DLA verification requires hardware acceleration. Emulation speeds up complex test cases so they can run faster and find corner cases.

1.3 Research Objective

This research aims to create a scalable and reusable UVM verification framework for testing the inference functionality of deep learning accelerators with simulation, emulation, and FPGA prototyping. This framework makes the deep learning accelerator verification process faster, easier, and more reliable. The proposed framework in this research has a generic and scalable UVM-based verification testbench. This testbench has the possibility of running different deep neural networks with different architectures and inference parameters on deep learning accelerators to test the inference process with different hardware configurations. It provides complete access to the DLA configuration space, which helps in hitting corner cases and speeding up the detection of functional bugs to achieve large coverage for DLA verification. Moreover, the proposed framework has a scalable error injection methodology for testing the trustworthiness of various DLA designs. The proposed error injection methodology applies error injection using three mechanisms: DNN data path error injection, input image error injection, and hardware configuration registers error injection. This methodology can be applied to each layer of a convolutional neural network to evaluate the robustness and ability of each layer to alleviate the impact of an injected error. The verification environment in this framework is portable across simulation and hardware-assisted verification (HAV) platforms for emulation and FPGA prototyping. Simulation helps to detect functional issues with more visibility in the early phases of the verification process. Furthermore, the co-emulation accelerates the DLA verification process by speeding up the running test cases for complex DNNs so they can run faster and hit corner cases. The Nvidia Deep Learning Accelerator (NVDLA) is used as a case study to prove this objective.

1.4 Contributions

Our main contributions can be summarized as follows:

1. Creating a novel and reusable UVM-based framework for DLA verification.
2. Design a novel, generic, scalable, and reusable UVM testbench to test DLA for simulation and hardware emulation.
3. Develop testing scenarios for single and multiple-layer CNNs for DLA verification.
4. Implementing a novel cross-layer error injection methodology to test the trustworthiness of DLA design with different CNNs using three error injection mechanisms:
 - (a) Applying error injection in the DNN data path by corrupting each DNN layer's feature map, weight, and bias.

- (b) Applying adversarial attacks on the input image to mitigate the perturbation in input received from cameras and other sensors in the physical world.
 - (c) Applying error injection in the DLA hardware configuration registers to detect faulty hardware configurations for a DNN that may disrupt the inference process.
5. Applying the proposed framework to NVDLA as a case study to demonstrate the trustworthiness of our verification framework.
 6. Analyzing and evaluating the verification framework simulation and emulation results for running CNN on DLA.

1.5 General Thesis Outline

This thesis is organized as follows: Chapter 2 provides background on CNN and the DLAs, then discusses the previous efforts in DLA verification.

Chapter 3 explains our scalable and reusable proposed framework for DLA verification.

Chapter 4 demonstrates the error injection methodology that is part of our proposed verification framework.

Chapter 5 illustrates applying the proposed framework to the NVDLA as a case study.

Chapter 6 presents the simulation and emulation experimental results.

Finally, the thesis is concluded in Chapter 7, where some future work is suggested as well.

Chapter 2

Background and Related Work

This chapter provides some background about convolutional neural networks, fault injection for deep neural networks, and deep learning hardware accelerators. Moreover, it discusses related work for different research efforts focused on DLA testing and verification.

2.1 Background

The application of neural networks to nonlinear "cognitive" problems is growing, including computer vision and natural language processing. In the inference phase, these models can learn from a dataset in the training phase and produce predictions with ever-increasing accuracy on new unseen data.

2.1.1 Convolutional Neural Networks

Because of their enormous data-handling capacity, deep neural networks have gained a lot of attention in the literature over the past couple of decades, as they are the most powerful tools available. Recently, deeper hidden layer technology has started to outperform classical methods in several domains, most notably pattern recognition. Convolutional neural networks are among the most often used deep neural networks. It gets its name from convolution, a mathematical linear action involving matrices [11]. A convolutional neural network is a specialized neural network that is specifically used for image detection. CNN consists of multiple layers, including convolution layers, pooling layers, and fully connected layers, to automatically and adaptively learn spatial hierarchies of information through back-propagation. The convolution and pooling layers extract the image features, while the fully connected layer maps the extracted features into the final output, like classification. Since those images have a very high dimension. Therefore, the image is split up into distinct sections from which local features can be extracted. Afterward, those segments are combined [12].

2.1.1.1 Convolutional Layer

A key component of the CNN architecture is the convolution layer, which conducts feature extraction. Typically, this involves combining linear and nonlinear operations, such as the convolution operation and activation function. Convolution is a particular type of linear operation for feature extraction. A matrix of numbers called a kernel is applied across the input, which is another matrix of numbers called a tensor. At each point of the tensor, the element-wise product between each element of the kernel and the input tensor is calculated and added to get the output value at the corresponding position in the output tensor, which is referred to as a feature map, as shown in Figure 2.1 [12]. This process is then repeated using different kernels to create feature maps that each reflect a different attribute of the input tensors; in this way, different kernels may be thought of as distinct feature extractors. Let layer l be a convolutional layer. Then, the input of layer l comprises $m_1^{(l-1)}$ feature maps from the previous layer, each of size $m_2^{(l-1)} \times m_3^{(l-1)}$. In the case where $l = 1$, the input is a single image I consisting of one or more channels. This way, a convolutional neural network directly accepts raw images as input. The output of layer l consists of $m_1^{(l)}$ feature maps of size $m_2^{(l)} \times m_3^{(l)}$ [13]. The i^{th} feature map in layer l , denoted $Y_i^{(l)}$, is computed as follows [13]:

$$Y_i^{(l)} = B_i^{(l)} + \sum_{(j=1)}^{m_1^{(l-1)}} K_{i,j}^{(l)} * Y_j^{(l-1)}, \quad (2.1)$$

where $B_i^{(l)}$ is a bias matrix and $K_{i,j}^{(l)}$ is the filter of the trainable weights of the network with size $2h_1^l + 1 \times 2h_2^l + 1$ connecting the j^{th} feature map in the layer $(l-1)$ with the i^{th} feature map in layer l . To skip a fixed number of pixels, both in the horizontal and vertical directions, before applying the filter again, $s_1^{(l)}$ and $s_2^{(l)}$ skipping factors are used [13]. Then, the size of the output feature maps is given by [13]:

$$m_2^{(l)} = \frac{m_2^{(l-1)} - 2h_1^l}{s_1^{(l)} + 1} \text{ and } m_3^{(l)} = \frac{m_3^{(l-1)} - 2h_2^l}{s_2^{(l)} + 1}, \quad (2.2)$$

A nonlinear activation function is then applied to the convolutional outputs. Currently, the rectified linear unit (ReLU) is the most widely utilized nonlinear activation function [12]. It simply performs the function [12]:

$$f(x) = \max(0, x), \quad (2.3)$$

2.1.1.2 Pooling Layer

A pooling layer offers a standard down-sampling technique that lowers the feature maps' in-plane dimensionality. To introduce a translation invariance to slight shifts and distortions and to cut down on the number of subsequently learnable parameters. It is noteworthy that, unlike filter size, stride, and padding, which are hyperparameters in pooling operations comparable to convolution operations, there is no learnable parameter in any of the pooling layers. Max pooling is the most widely used type of pooling operation. It takes input feature maps, extracts patches from them, outputs the largest value in each patch, and discards all other values, as shown in Figure 2.2 [12].

2.1.1.3 Fully Connected Layer

The final convolution or pooling layer's output feature maps typically get flattened, or converted into a one-dimensional (1D) array of numbers, and then connected to one or more fully connected layers, referred to as dense layers, where each input and each output are connected by a learnable weight. A subset of fully connected layers maps the features that were retrieved by the convolution layers and down-sampled by the pooling layers to the final outputs of the network, which in classification tasks are probabilities for each class. The number of output nodes in the last fully linked layer is typically equal to the number of classes [12].

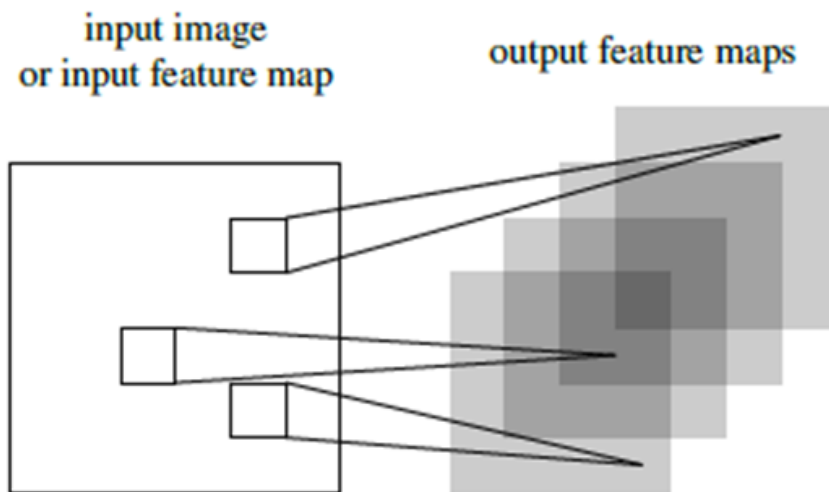


Figure 2.1: Illustration of a single convolutional layer [13].

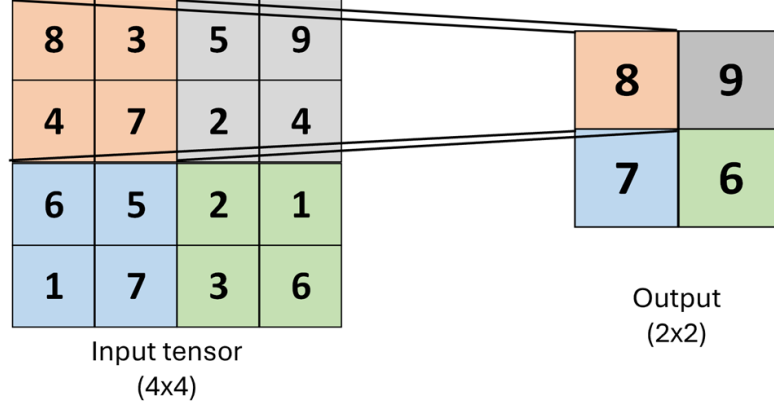


Figure 2.2: Max pooling [12].

2.1.2 Fault Injection for Deep Neural Networks

Hardware faults are classified as either permanent or transient based on fault duration. A hardware failure is a persistent failure that stems from physical defects and remains active until fixed. However, a transient fault has a short period of activity. External factors are the primary cause of transient faults, commonly referred to as soft errors [14]. A temporary malfunction in a system might lead to an application’s output being distorted or the system crashing. Error mitigation is also required for DNN systems to meet specified reliability objectives since many of them are safety-critical and soft errors can have catastrophic consequences. As with autonomous cars, a minor error might cause an object to be misclassified and for the car to respond incorrectly. A truck, for example, might be misclassified under soft error. The DNN in the car should identify the approaching object as a transportation truck, just as in the case of no errors, and the brakes should be deployed promptly to prevent a collision. However, if a slight error happens in the DNN, the truck can be mistakenly identified as a bird, and fast-moving vehicles may have a delay in the braking system’s engagement, which could lead to an accident. This is a major issue because it results in the vehicles’ functional safety being violated, as per ISO 26262 requirements [15].

There are some concerns to consider while implementing an effective fault injection methodology because evaluating the impact of a fault differs from measuring a typical fault injection. There are three primary fault categories [16]:

- Masked faults: The fault-free and faulty DNN outputs are the same for the user.
- Benign faults: The user accepts a faulty output even when it differs from a fault-free output.
- Malignant fault: The user rejects the defective output.

The fault allowance is decided by the DNN, the chosen metric, and the error threshold. For example, in autonomous driving object detection algorithms, a fault

could cause a minor deviation from the desired behavior; in this case, it would be considered a benign fault if the pedestrians were still identified correctly. Conversely, if the error significantly alters the behavior of the application, the fault is then classified as malignant, much like when a major failure occurs and an object cannot be positively identified. For example, when a car doesn't stop and collides with a pedestrian.

Many DNN fault models have been proposed by academics, for example:

- TensorFI, a Software-implemented Fault Injection (SWiFI) tool, is presented in [17], where faults are injected into the data-flow graph utilized in TensorFlow [18] applications. TensorFlow operator outputs can have hardware and/or software errors injected into them using TensorFI, and the impact of these errors on the machine learning application can be examined. TensorFI's primary benefit over conventional SWiFI frameworks is that it works directly with the TensorFlow operators and graphs, making its results accessible to developers. Since many external libraries rely on the structure and semantics of the TensorFlow graph, and since TensorFlow operators are not available once the graph has been constructed, these libraries shouldn't be modified.

TensorFI thus tackles these challenges by first duplicating the TensorFlow graph and then using Python to create a Fault Injection (FI) graph that mirrors the original one. With the exception of the ability to insert faults based on the configuration parameters provided, the operators in the FI graph function similarly to the original TensorFlow operators. Furthermore, the original TensorFlow graph's performance is unaffected because the FI graph is only called during fault injection. Furthermore, additional libraries that rely on the structure and semantics of the graph can keep functioning since the TensorFlow graph is left unchanged other than the addition of the FI graph.

TensorFI is then assessed using a wide range of FI configurations (e.g., fault types, error rates) on 12 ML applications in TensorFlow, including DNN applications. Notable variations were discovered as a result of various configurations and individual ML applications. However, because the tool's TensorFlow operators during fault injections are modeled in Python and do not benefit from the optimizations and low-level implementation of TensorFlow, the fault injection tool introduced considerable performance overheads.

- The design and the technical underpinnings of PyTorchFI, a runtime perturbation open source tool for DNNs created within the well-known PyTorch [19] framework, are presented in [20]. With PyTorchFI, users can perform NN perturbations in weights and/or neurons in neural networks during the convolutional operations of DNNs. As a result, it makes it possible to investigate how various perturbations emerge and propagate at the application level. In addition to that, a straightforward and user-friendly framework is offered for carrying out perturbations at runtime that has been tested on both CPUs and GPUs.

As a further demonstration of PyTorchFI's versatility, several use cases are presented, including (1) reliability analysis of CNNs, (2) reliability analysis

of object detection networks, (3) resiliency analysis of models intended to withstand adversarial attacks, (4) training error-resilient models, and (5) an initial investigation of PyTorchFI’s use for DNN interpretability.

Because there is only one check per layer in PyTorchFI, the implementation has a very low overhead. There isn’t any overhead if there are no defined perturbations. Additionally, PytorchFI operates at the native speed of silicon because error modeling doesn’t require code instrumentation. This makes it possible to explore the enormous state space more quickly, which is essential for comprehending parts of faults in safety-critical systems that arise in the real world. For the purpose of modeling high-level perturbations and comprehending their impact at the system level, PyTorchFI operates at the application level of DNNs. Lower-level perturbation models, on the other hand, including register-level errors, cannot be represented at this level unless they can be mapped to single- or multiple-bit flips.

- In [21] a novel fault injection framework is introduced, which is called PyTorchALFI (Application Level Fault Injection for PyTorch) based on PyTorchFI. PyTorchALFI provides an efficient way to define randomly generated and reusable sets of faults to inject into PyTorch models, defines complex test scenarios, enhances data sets, and generates test Key Performance Indicators (KPIs) while tightly coupling fault-free, faulty, and modified NN. The introduced framework focuses on efficiently incorporating fault injection into the regular software development cycle, especially for safety-critical NN applications built with PyTorch. Moreover, how the framework could be integrated into existing PyTorch projects and how it is configured for different use cases are demonstrated.
- A methodology for the reliability analysis of CNNs is presented in [22], using an error simulation engine and validated error models taken from an extensive fault injection approach. These error models are characterized by the patterns of output corruption resulting from faults in the CNN operators, thus bridging the gap between fault injection and error simulation and utilizing the advantages of both approaches. Furthermore, this framework is easy to use, controllable, flexible, and fast, while combining the accuracy of architecture-level and application-level fault injection. In terms of speed and the ability to reproduce the effects of faults on the final CNN output, this methodology achieved higher accuracy than the TensorFI functional error simulator [23] and the innovative SASSIFI fault injection environment [24] for CNNs accelerated onto GPUs.

In this work, the proposed error injection methodology models both the application level and the architecture level perpetuation using the three proposed error injection mechanisms: DNN data path error injection, DNN input image error injection, and hardware configuration registers error injection for testing the resilience of DNN running on deep learning accelerators. The proposed methodology has a cross-layer error injection at different layers in the DNN. Moreover, it did not require any type of additional processing, which maintains the system’s performance compared to other fault injection mechanisms. Hence, the proposed framework did not add any extra time.

2.1.3 Deep Learning Hardware Accelerators

A DNN accelerator is designed using one of two properties shared by all DNN algorithms: (1) Strong temporal and spatial localities in the data within and across each feature map allow the data to be stored and reused strategically; (2) each feature map's MAC operations have highly sparse dependencies that allow them to be computed in parallel. To benefit from the second property, DNN accelerators use spatial architectures, which consist of massively parallel processing engines that each compute MACs. Hence, a DNN accelerator consists of a global buffer along with an array of processing engines. In addition to a DRAM connection from which data is transferred [14].

2.1.3.1 Eyeriss Deep Learning Accelerator

Eyeriss [25] is an accelerator that utilizes two key techniques to provide cutting-edge accuracy with minimal consumption of energy in the system, including Dynamic Random Access Memory (DRAM) in real-time. Firstly, to minimize data movement and support various shapes, efficient dataflow and supporting hardware (spatial array, memory hierarchy, and on-chip network) are needed; secondly, data statistics are used to minimize energy through zeros skipping to avoid needless reads and computations. In addition to that, data compression is used to reduce off-chip memory bandwidth, which is the most expensive data movement.

The accelerator's architecture and memory hierarchy are shown in Figure 2.3. To improve data movement, temporal reuse of loaded data is facilitated by buffering the input image data (Img), filter weights (Filt), and partial sums (Psum) in a shared 108 KB Static Random Access Memory (SRAM) buffer. Memory traffic and computation can overlap when image data and filter weights are read from DRAM into the buffer and streamed into the spatial computation array. Even with the memory link operating at a lower clock frequency than the spatial array, the system is still able to achieve excellent computational efficiency due to the presence of streaming and reuse. The partial sums that are produced by the spatial array's computation of the inner products between the image and the filter weights are sent to the buffer, where they may subsequently be compressed and rectified before being sent to the DRAM. With run-length-based compression, the average image bandwidth is lowered by 2x. Partial sums are saved in the buffer and then restored to the spatial array to provide configurable support for image and filter sizes that do not fit entirely within the spatial array. The number of these "passes" required to complete the computations for a particular layer depends on the buffer size and spatial array.

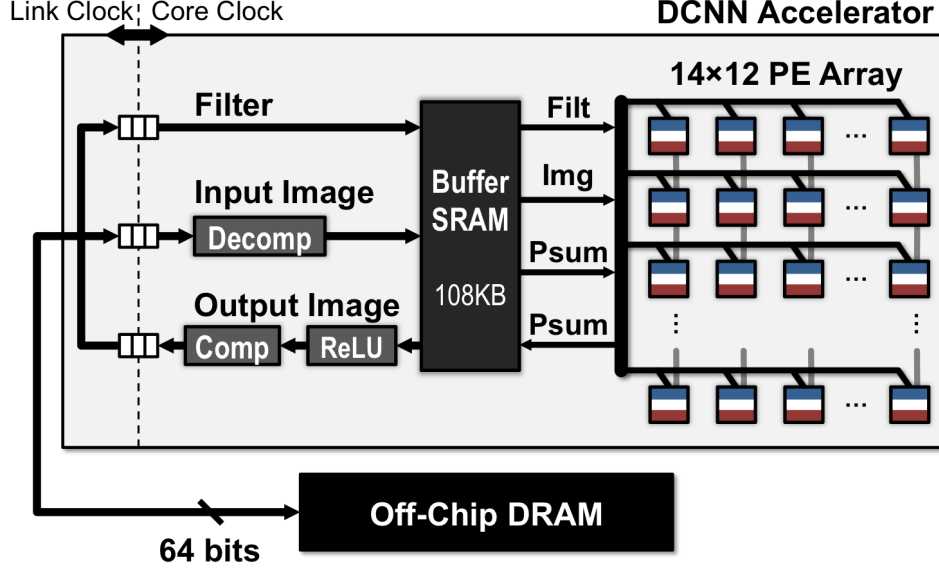


Figure 2.3: Eyeriss deep learning accelerator architecture [25].

2.1.3.2 Flex Logix InferX X1 Accelerator

The Flex Logix InferX X1 [26] accelerator is a system that combines software and parallel hardware that can be adjusted per the needs of any given algorithm to fully leverage parallelism. For edge servers and industrial vision systems, the InferX X1M board is suitable for processing huge models and mega-pixel images at batch = 1 for object detection and other high-resolution image processing tasks. It can also support Yolov5. The software tools offered by Flex Logix include a utility to move trained Open Neural Network Exchange (ONNX) models to the X1M, a basic runtime framework to facilitate inference processing on Linux and Windows, and a driver with external APIs to set up and deploy models. Additionally, X1M offers high computational density and performance, giving customers' applications greater flexibility.

2.1.3.3 NVDLA

NVDLA is a configurable deep learning accelerator part of Nvidia's Xavier System on Chip (SoC) [27]. It is used to carry out deep learning inference operations. It accelerates convolution, activation, and pooling, which are the building elements involved in every CNN. To meet inference's computational requirements, NVDLA provides an open and standardized design. Since it maintains separate and programmable components, the modular and scalable architecture of NVDLA is appropriate for a wide range of applications. Furthermore, it is an open-source and Verilog-based source code that supports popular deep-learning frameworks like TensorFlow and Caffe. Inference acceleration is made simple and flexible with NVDLA hardware. It is easily scalable for a wide range of applications and performance levels, from smaller and cost-sensitive Internet of Things (IoT) devices

to larger performance-focused IoT devices. Furthermore, NVDLA can be set up as a small or large system, as shown in Figure 2.4, to satisfy particular power, cost, or area requirements [28].

2.1.3.3.1 Small NVDLA

The small-NVDLA system shown in Figure 2.4 is suitable for cost-sensitive IoT devices, AI, and automation-oriented systems for which cost, area, and power are the main concerns. Area, power, and cost reduction are achieved through the NVDLA configurable resources. The complex NN models can be split and their load complexity can be decreased by pre-compiling and performance-optimizing. This allows for a scaled-down NVDLA implementation where models use less storage and require less time for system software to load and process. Since these specially designed systems usually only handle one task at a time, high system performance when using the NVDLA as part of the system is not a strong concern. Hence, the second memory interface option is not part of the systems that use the small-NVDLA system. The lack of a fast memory path is unlikely to have a significant effect when overall system performance is not a priority [28].

2.1.3.3.2 Large NVDLA

If high performance and versatility are the main priorities, the large-NVDLA system is a better choice. Performance-oriented systems need to retain a high degree of flexibility because they can execute inference on a wide variety of network topologies. Additionally, rather than serializing inference operations, these systems can handle multiple jobs concurrently. Therefore, inference procedures shouldn't take up a lot of the host's processing capacity. A second memory interface for a specialized high-bandwidth SRAM was incorporated into the NVDLA hardware shown in Figure 2.4 as an optional feature to meet these demands. The NVDLA uses this SRAM as a cache and has it connected to a fast-memory bus interface port [28].

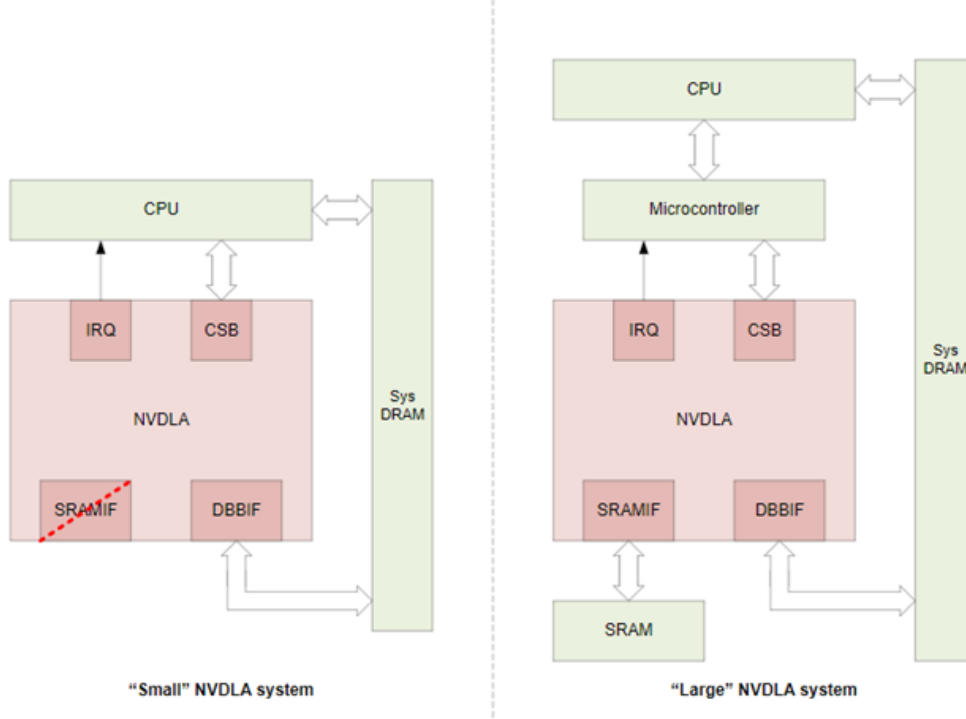


Figure 2.4: Small NVDLA versus large NVDLA system [28].

2.1.3.3.3 NVDLA Hardware Architecture

The NVDLA architecture shown in Figure 2.5 consists of different components that operate separately from each other, supporting specific operations integral to inference on deep neural networks [29]:

- **Convolution core:** It is responsible for high-performance convolutions. Two types of data are used in its operation: the input data, such as the image that needs to be classified, and the weights that were previously trained online. NVDLA supports different kinds of convolutions, which are direct, image-input, Winograd, and batching convolutions. The convolution engine is a five-stage pipeline consisting of Convolution DMA (CDMA), Convolution Buffer (CBUF), Convolution Sequence Controller (CSC), Convolution MAC (CMAC), and Convolution Accumulator (CACC).
- **Activation Engine (Single Data Point Operations, SDP):** It is responsible for performing both linear and nonlinear activation functions. The non-linear functions have various supports for ReLU, Parametric Rectified Linear Unit (PReLU), and Sigmoid, while other linear functions are primarily scaling functions.
- **Pooling Engine:** It is used for pooling operations. Various types of pooling, including average, min, and max pooling, are supported.
- **Local Response Normalization Core (Cross-channel Data Processor, CDP):** It

can be used to apply Local Response Normalization, a kind of normalization function.

- Data Reshape Core (RUBIK): It is capable of transforming data formats in several ways, including contraction, slicing, joining, and splitting.
- Bridge DMA: It transfers information between the high-performance memory interface if used and the system DRAM.

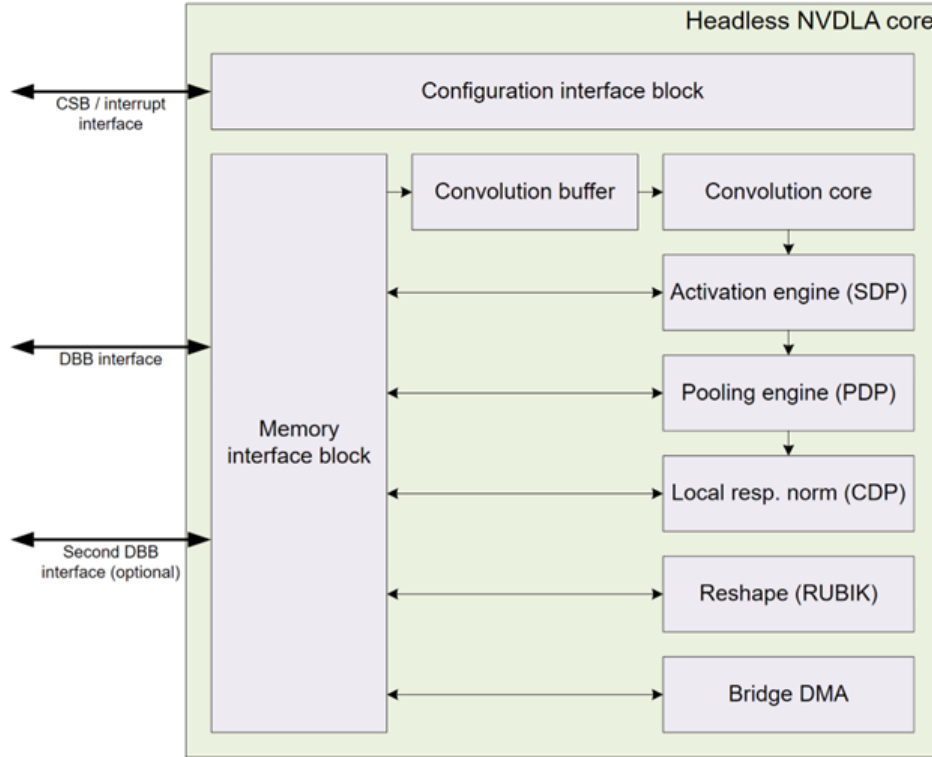


Figure 2.5: NVDLA hardware architecture [28].

Additionally, NVDLA has three main interfaces, as shown in Figure 2.6 [30]:

- Data Backbone (DBB) interface: It is an AXI interface used to connect the NVDLA with the external memory,
- Configuration Space Bus (CSB) interface: It is a control bus that a CPU uses to access the NVDLA configuration registers.
- Interrupt interface: It is a single bit that is asserted upon completion of a task or error.

Furthermore, the System Data Bus (SRAM) is a high-bandwidth interface that is optionally used for an external SRAM connection.

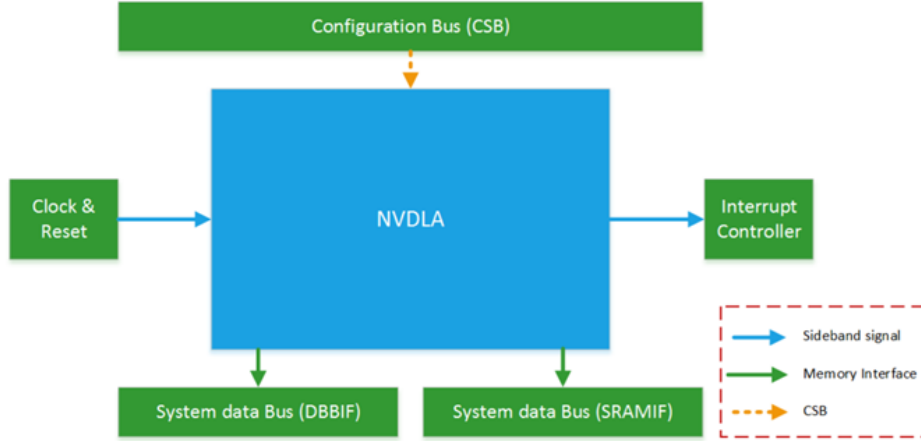


Figure 2.6: NVDLA interface [30].

2.1.3.3.4 NVDLA Software

Software support is provided by NVDLA to facilitate the user’s interaction with the hardware. Both the compilation tools and the runtime environment comprise the two categories of NVDLA-related software [28]:

- **Compilation tools:** A parser and compiler are the main components. The compiler is in charge of generating an order of hardware layers that is optimized for a particular NVDLA setup to improve performance by lowering model size, load, and run times. Parsing and compiling are the two fundamental steps in the compilation process. The parser can read a Caffe model that has already been trained and produce an "intermediate representation" of a network that can move on to the next compilation stage, as shown in Figure 2.7. The compiler creates a network of hardware layers based on the hardware configuration of an NVDLA implementation and the parsed intermediate representation. These steps can be performed on the device that has the NVDLA implementation and could be done offline. It’s crucial to understand the precise hardware setup of an NVDLA implementation since it helps the compiler produce the right layers for the features that are offered. Depending on the convolution buffer size available, this could involve, for instance, dividing convolution operations into several smaller mini-operations or choosing between various convolution operation types (such as Winograd convolution or basic convolution). This stage is also in charge of assigning memory regions for weights and quantizing models to lower precision, such as 8- or 16-bit integers or 16-bit floating points. A list of operations for various NVDLA configurations can be produced using the same compiler tool.
- **Runtime environment:** This refers to the setup where a model is executed on NVDLA-compatible hardware. It can be split into two sections:
 - **User Mode Driver (UMD):** This is the primary user-mode software interface. The neural network is parsed, and then the compiler builds the network layer by layer into an NVDLA loadable file format, as shown

in Figure 2.7. After loading this loadable, the user mode runtime driver sends the inference job to the kernel mode driver.

- Kernel Mode Driver (KMD): It consists of drivers and firmware where the NVDLA layer operations are scheduled, and each functional block is configured by programming the NVDLA registers.

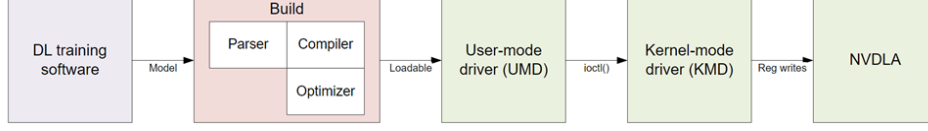


Figure 2.7: NVDLA system software dataflow [28].

The API for UMD is standard and may be used to process loadable images, execute inference, and bind input and output tensors to memory locations. An "NVDLA loadable" image is a type of stored representation of the network that is used to begin runtime execution. In the loadable, every functional block in the NVDLA implementation is represented by a "layer" in the software; each layer contains details about the dependencies of the blocks it uses, the tensors it uses as inputs and outputs in memory, and the particular configuration of each block for a particular operation. KMD schedules each action by utilizing a dependency graph that connects the layers. The format of an NVDLA loadable is standardized for both UMD and compiler implementations. An inference job is received by KMD's main entry point, which then chooses one of the several possible jobs for scheduling and sends it to the core engine scheduler. The core engine scheduler is in charge of managing NVDLA interrupts, scheduling layers on a block-by-block basis, and updating any dependencies associated with that layer in response to a task from a preceding layer being completed. When a layer is ready to be scheduled, the scheduler takes information from the dependency graph. This helps the compiler schedule layers optimally and prevents performance discrepancies caused by different KMD implementations.

2.2 Related Work

The previous efforts were based on directly testing an accelerator's inference function by running a specific DNN.

In [31], an Image Verification Model (IVM) is proposed for the verification of SoC vision-based accelerators. The proposed IVM is a reusable model to verify an accelerator block output. The IVM can be configured to meet the varying requirements of the different accelerators being verified. Additionally, it can be programmed to monitor and analyze some needed parameters, such as throughput, latency, frame rate, and resolution. The IVM proposed is developed in SystemVerilog (SV) and is compliant with UVM. Nevertheless, this model is specific to vision-based accelerators.

Shanggong et al. in [32] run the LeNet-5 CNN with INT8 format on NVDLA, which is integrated on RISC-V SoC as shown in Figure 2.8. The NVDLA’s CSB interface is connected through an adapter to the ARM Advanced Peripheral Bus (APB) that is used in the SoC to connect the NVDLA to the RISC-V core. Moreover, an AXI4-compliant DBB interface is used for communication between the NVDLA and an external SoC DRAM. Additionally, the LeNet-5 CNN is implemented in the C language. However, this work is specific to a single CNN and is done with a low-precision INT8 format that doesn’t provide enough testing for the NVDLA. Furthermore, the communication time between the RISC-V core and the NVDLA is much higher than using a native UVM testbench code, which affects the overall system speed during the verification process.

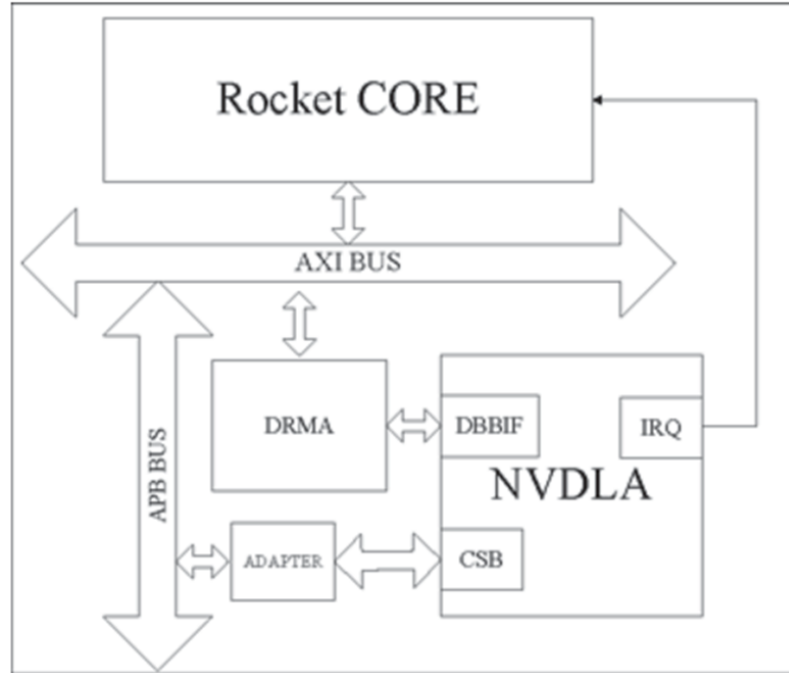


Figure 2.8: NVDLA integration on RISC-V SoC [32].

In [33], the NVDLA hardware information is configured in the virtual verification platform provided by Nvidia, and a Linux kernel image is built to run the software environment of NVDLA. Then, a LeNet network model project of a Caffe framework is built to test and verify the function of the NVDLA compiling part of the Caffe framework network models and implement the deep learning inference on the NVDLA hardware. Similarly, this effort is done for a specific CNN instruction-by-instruction execution that consumes much time compared to that for the UVM testbench, mainly in the early phases of the testing process.

An implementation is proposed in [3, 34] for the NVDLA architecture as an SoC design and prototype on an FPGA platform to run on-device inference of CNN. At which building an embedded SoC architecture accomplishes relevant communication between the host processor and NVDLA core. Furthermore, a run-time execution environment to load and process compiled neural networks in the NVDLA system is implemented. Then build an inference setup for the small NVDLA hardware on the

Zynq Ultrascale+ FPGA for executing AlexNet as a benchmark for evaluating the implemented system.

The resilience of the Register-Transfer Level (RTL) design of the neural network accelerator is investigated in [35] using a High-Level Synthesis (HLS)-based methodology. The severity of RTL NN components against different types of errors is demonstrated by an extensive vulnerability analysis conducted on the different components of the RTL NN accelerator against both transient and permanent faults. This is done by applying fault injection into the data registers of the RTL neural network accelerator, as shown in Figure 2.9. Furthermore, a low-overhead technique that recovers corrupted bits without requiring redundant data is proposed to mitigate faults with high efficiency.

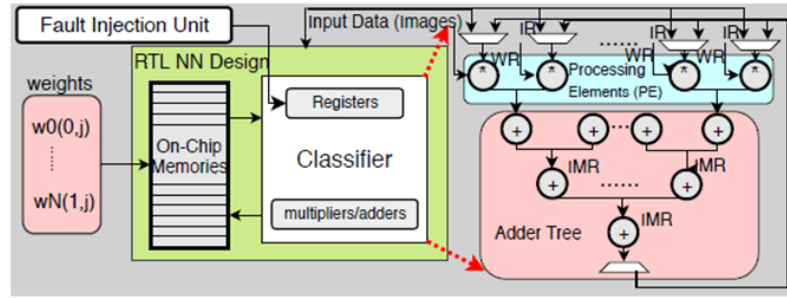


Figure 2.9: The NN accelerator RTL model [35].

In [14], a canonical model of the DNN accelerator hardware is used to modify a DNN simulator and give a fault injection technique for four popular neural networks for image recognition. Furthermore, a large-scale study on fault injection is conducted for the faults occurring in the accelerators' data path. Then, depending on the neural network's architecture, data types, layers' positions, and types, the error propagation behaviors are categorized. In addition to that, two cost-effective error protection techniques are proposed to reduce the rate of Silent Data Corruptions (SDCs). The first technique is symptom-based detectors implemented in software, and the second technique is selective latch hardening implemented in hardware.

In [36] a fault-tolerant technique for deep learning accelerators in ImageNet applications is proposed. The proposed technique detects faults with a shadow register and corrects the error in the next accumulation step without suspending the existing pipeline or requiring data replay. As no stall or flushing operation is required by the proposed correction mechanism, there is no penalty on throughput performance. Moreover, the proposed technique is compared with other mitigation techniques, including bit masking, zeroing on error, and sign-bit masking, on an Intel FPGA-based DNN accelerator.

Furthermore, for convolutional layers in systolic array RTL DNN accelerators at channel granularity, a cross-layer fault injection methodology is proposed in [16] by simulating the RTL Channel Under Test (ChUT) execution. Then, to determine the impact of the injected faults at the application level, the DNN execution is completed using the faulty outputs from the RTL simulation at the software level. This framework has parallel capabilities and can reduce fault injection time. The

DNN execution is then finished using the faulty outputs from the RTL simulation at the software level to determine the impact of the injected faults at the application level. This framework runs faster in terms of fault injection time and has parallel capabilities.

As depicted, none of these efforts use UVM, which would have provided more modularity, scalability, and reusability compared to other verification methodologies.

In this work, the proposed framework provides a generic and scalable UVM-based verification methodology that is much more efficient in terms of running speed and adaptability for different DLA designs for both simulation and emulation for various DNN architectures. Moreover, the proposed error injection methodology is scalable and reliable across different DLA designs and DNN architectures. As the proposed error injection methodology did not add any extra simulation or emulation runtime. In addition, it did not add any performance overhead compared to other fault injection mechanisms. Furthermore, the proposed framework can be portable across the various HAV platforms to accelerate the verification process.

Chapter 3

Proposed UVM framework for DLA Verification

This chapter presents the proposal of the UVM-based framework to verify the inference process of the DNNs on DLAs with different designs. The proposed framework is to create a scalable and reusable UVM verification testbench for testing deep learning accelerators by running different test scenarios for convolutional neural networks with multiple configurations. Each test scenario configures the DLA to run a specific CNN and drives the inputs, pre-trained weights, and bias required for each layer. The proposed framework makes the verification process faster, easier, and more reliable. It provides large coverage, can hit corner cases, and offers complete access to the DLA configurations. The proposed framework is portable across simulation and HAV platforms for emulation and FPGA prototyping. Those HAV platforms, including emulation and prototyping platforms, are used in SoC verification to accelerate the verification testbenches to run faster and hit more corner cases. Figure 3.1 shows an example of an emulation platform, which is the Veloce Strato emulator [37]. Moreover, Figure 3.2 shows an example of an FPGA prototyping platform, which is the Veloce proFPGA platform [38].



Figure 3.1: The Veloce Strato emulator [37].

Hardware simulation models the behavior of the SoC or system-level design using the Electronic Design Automation (EDA) tool setup. Moreover, hardware simulation is low-cost and efficient at the early stage of the verification process to run different testing scenarios for debugging and system stability [39]. On the other hand, the main advantage of hardware emulation is speeding up the verification process and maintaining system reliability, including full functional verification and virtual prototyping for a SoC, which maintains the system validation, the accuracy of functional behavior, and the functional coverage of every system component in

the design. A design description expressed in HDL is the foundation of the hardware emulator [10], which is an array of FPGAs. The design is converted into a gate-level netlist using an RTL compiler/synthesis tool. A crystal chip—which might be referred to as an FPGA—has the design mapped onto it with different technologies. An Advanced Verification Board (AVB) would be used to distribute a large design across numerous multiple-crystal chips.



Figure 3.2: The Veloce proFPGA platform [38].

A hardware emulator contains many AVBs, as shown in Figure 3.1, to implement and validate large-scale DLAs to solve the FPGA limited resources issues. This facilitates the DLA design integration with the UVM environment to benefit from emulation speed-up. The hardware emulation shows enhanced execution over the conventional programming test simulators. Although the core programming simulation strategy is still used mainly at the early verification stages, the hardware emulator’s specific hardware has custom integrated circuit innovation enhanced for hardware emulation’s requirements. Soft models need to pass through the synthesis process even though they are aggregated into hardware. The UVM testbenches rely strongly on Object-Oriented Programming (OOP) techniques. They use SV classes and constructs that cannot be synthesized and cannot be modeled or implemented on a hardware emulator; instead, they are implemented and assessed on a software simulator. The power and reuse benefits of test environments based on UVM can be preserved while maximizing the execution benefits of emulation by combining hardware emulation for these modules that can be synthesized with a software tool for these classes that cannot be synthesized [40]. Using hardware emulation to implement a test case in which the software-based simulator keeps running in addition to the hardware emulator poses unique challenges:

- Constructs that are time-planned, such as clock synchronization or other delays, that block the performance of the hardware emulation should be removed. Moreover, the verification environment’s hierarchical objects should be able to overcome any obstacles between the RTL and the testbench. For example, transactor components such as drivers and monitors ought to be included in both timed and untimed modules, as well as modules that are located in their separate spaces.

- Every transactor's untimed part is put inside the Hardware Verification Language (HVL) space to be a proxy to the corresponding timed parts in the HDL space that are implemented as a Bus Functional Model (BFM), which acts as an interface connection. The HDL BFM and the related testbench proxy must be interfaced at the Transaction Level Model (TLM) to communicate between the hardware emulator and the software-based simulator.
- The synthesizable RTL code that will be represented on the hardware emulator should be kept fully separate from the non-synthesizable verification environment that has been built. The HDL for DUT and the Hardware Verification Language (HVL) for verification are the two primary and significant sets of chains that are required [41]. Consequently, the software-based simulator and the hardware emulator (which deals in pin wiggles format instead of transaction objects) are connected via a physical communications link.

3.1 Input Data and Weight Preparation

The proposed framework uses a CNN Caffe model that has the CNN pre-trained weight and bias and a .prototxt file containing the details of the CNN model architecture to extract the CNN weight and bias for each layer using Python-based scripts. After that, the extracted weight, and bias, in addition to the input data, are then remapped according to the DLA requirements. Then, precision conversion is done on the remapped data based on the DLA precision configurations. Hence, the remapped and precision converted data is saved inside an input package in the CNN testbench to be used by the UVM verification environment during the testing process, which is then transmitted to the DLA as shown in Figure 3.3.

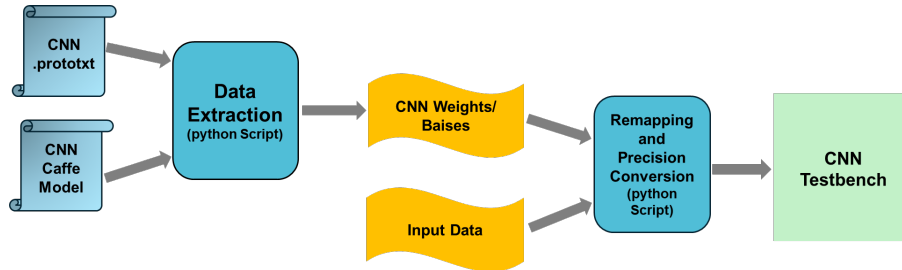


Figure 3.3: Inupt data and weight pre-processing.

3.2 UVM Testbench Architecture

The proposed verification testbench shown in Figure 3.4 is a UVM-based testbench established for testing CNN inference on the deep learning accelerators for simulation and emulation. As illustrated, hardware acceleration has become essential in SoC verification. Complex test cases can be accelerated with emulation to run faster

3.2.1 HDL Top

The HDL Top has the DUT instance and its connections with the UVM testbench virtual interfaces as it is timed and synthesizable. Moreover, the HDL Top is the hardware part running on the emulator. The implemented transactors in Figure 3.5 allow the transaction-level UVM testbench part to access the RTL DUT. Transactors are split into proxies (as part of the HVL side running on the simulator) and BFM s, which are AXI-based interfaces (as part of the HDL side running on the HAV platform). The implemented AXI-based interfaces make it easier to integrate different DLA RTL designs. Furthermore, the virtual interfaces are registered in the UVM configuration database so that other testbench components can have access to them, as they are connected to the DUT as follows:

- **AXIMaster IF:** A virtual AXI interface responsible for sending the DUT configurations on the AXI bus.
- **AXISlave IF:** A virtual AXI interface responsible for read/write operations on the DUT DRAM interface used for CNN data, weight, and bias.
- **INTR IF:** It is responsible for detecting interrupts sent by the DLA DUT.

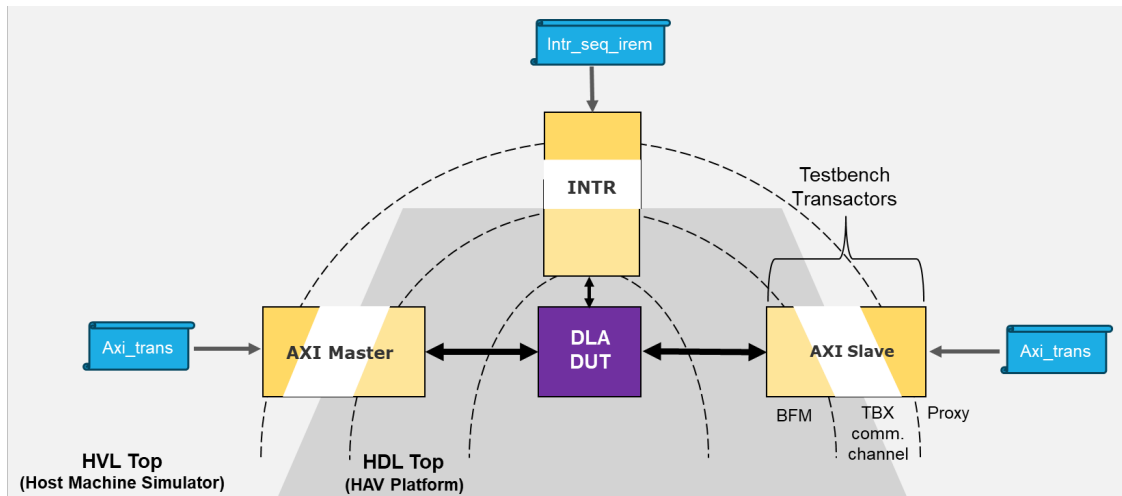


Figure 3.5: UVM testbench timed HDL and untimed HVL partitioning.

3.2.2 HVL Top

HVL Top is responsible for running the UVM test, which is the testbench part running on the host machine simulator, as it is dynamic, class-based, behavioral, and untimed.

3.2.2.1 CNN Test

The CNN test class is extended from the `uvm_test` class, which is the top-level UVM component in the UVM testbench. The CNN test checks and verifies the DUT functionality by running the testing scenarios created in the verification plan. This test class includes the CNN environment and the AXI environment. At the test level, we could pass the environment configuration by creating a handle for the `env_config` shown in Figure 3.4 and assigning all needed arguments for the testbench agents and other configurations. Also, declare all virtual sequences, virtual sequencers, and other components that are needed for the entire test, as shown in Figure 3.4. Furthermore, the test runs by calling the task `run_test()` in the HVL top.

3.2.2.2 CNN Virtual Sequence

A UVM sequence is an object that contains a behavior for generating a stimulus. UVM sequences are not part of the component hierarchy. They are UVM objects that can be transient or persistent. A UVM Sequence instance can come into existence for a single transaction; it may drive stimulus for the duration of the simulation or anywhere in between. UVM sequences can operate hierarchically with one sequence, called a parent sequence, invoking another sequence, called a child sequence. To operate, each UVM sequence is eventually bound to a UVM sequencer. Moreover, multiple UVM sequence instances can be bound to the same UVM sequencer [44].

The `cnn_vseq` is a virtual sequence that is extended from the `uvm_sequence`, which is implemented and started by a virtual sequencer as shown in Figure 3.4. The `cnn_vseq` virtual sequence starts the `cnn_layer_seq` sequence that is responsible for constructing and sending the `cnn_layer_seq_item` transaction that is a UVM transaction containing the DLA hardware configurations that are specific for each CNN layer in addition to the CNN layer data, weight, and bias. Then this `cnn_layer_seq_item` transaction is sent to the `cnn_layer` agent. Moreover, the `cnn_vseq` is a base sequence that could be extended to many child CNN sequences. Each one is responsible for running a certain CNN by setting the configurations of each hardware block required by each layer of that CNN. In addition to that, the input data, weight, and bias for each CNN that are pre-loaded through the Python-based scripts are retrieved from the input package to be sent to the DUT as part of the `cnn_layer_seq_item` transaction with the hardware configurations.

For example, if we want to run a CNN with multiple layers, then a new sequence will be extended from the `cnn_vseq` base sequence, and then it can have the retrieved input data, weight, and bias for each layer, along with the configurations of each block in the hardware related to each CNN layer. For example, the convolution pipeline for convolution layers, the planar data engine for the POOL layers, and so on. Then, this flow could be extended to multiple layers. Therefore, the proposed approach is scalable to run different convolutional neural networks by adding new sequences for each network.

3.2.2.3 CNN Environment

The UVM environment is a higher-level verification and hierarchical component that groups other interrelated verification components. The CNN environment is extended from the UVM environment. The CNN environment consists of two levels of abstraction:

1. The `cnn_layer` agent.
2. The three UVM agents: the `reg_file` agent, the `mem` agent, and the `intr` agent.

In addition to the Data Scoreboard and the `env_config` class that has the agents' configurations as shown in Figure 3.4.

3.2.2.4 CNN Layer Agent

The `cnn_layer` agent Figure 3.4 is extended from the `uvm_agent`. The `cnn_layer` agent is the highest abstraction level agent that has the CNN layer driver, which is extended from the `uvm_driver`. The CNN layer driver receives the `cnn_layer_seq_item` transaction and maps the included hardware configurations inside it to the DLA hardware blocks' registers specified for each layer as register read/write operations included in the `reg_file_seq_item` transaction, which is also a UVM transaction to be transmitted to the middle level of abstraction `reg_file` agent. Moreover, the CNN layer data, weight, and bias received in the `cnn_layer_seq_item` transaction are transmitted via the CNN layer driver according to the memory address configurations included in the `mem_seq_item` UVM transaction as memory read/write operations to the middle level of abstraction `mem` agent. The transaction flow is summarized as follows:

- The `cnn_layer` agent driver in Figure 3.4 receives a high-level `cnn_layer_seq_item` transaction that contains CNN layer-specific configurations, data, weight, and bias, and then this driver converts this `cnn_layer_seq_item` transaction into two different mid-level transactions:
 - `reg_file_seq_item` that contains the register's address and its configuration value for register read/write operation related to the required DLA CNN configuration. This transaction is then received by the `reg_file` agent's driver using the `cnnlayer2regfile` UVM sequence. This sequence acts as a linking sequence to send the `reg_file_seq_item` transaction to the `reg_file` agent. Then this transaction is converted to the `axi` transaction (which has a lower level of abstraction) to be sent to the DUT through the `AXIBusMaster` driver.
 - `mem_seq_item` that contains the memory address and value for memory read/write operations related to CNN data and weight. This transaction

is then received by the mem agent’s driver using the `cnnlayer2mem` UVM sequence. This sequence also acts as a linking sequence to send the `mem_seq_item` transaction to the mem agent. Similarly, this transaction is then converted to the `axi_transaction` (which also has a lower level of abstraction) to be sent to the DUT through the `AXIBusSlave` driver.

- Furthermore, the interrupt received from the DLA DUT through the middle level of abstraction `intr` agent is handled through the `intr_seq_item`, and then forwarded to the `cnn_vseq` through the CNN layer driver in the CNN layer agent.

Additionally, the CNN layer driver is connected to the data scoreboard using a UVM analysis port to send the received CNN layer output to the data scoreboard for comparison and checking.

3.2.2.5 AXI Environment

The mentioned middle-level agents send the CNN data, weight, and bias with the DLA hardware registers’ configurations as memory and register read/write operations to be sent by the AXI environment agent on the DLA DUT AXI interface. The AXI environment is extended from the UVM environment. It consists of the `axiBus` UVM agent, which contains the `AXIBusMaster Driver` that stimulates the DUT by driving AXI transactions for DLA register read/write operations, and the `AXIBusSlave Driver` that stimulates the DUT by driving AXI transactions for memory read/write operations, as shown in Figure 3.4.

3.2.2.6 Reference Model

The implemented reference model in this framework, as shown in Figure 3.4 is a Python-based script that runs the CNN model using network architecture and inference parameters received from the `cnn_vseq` sequence. The CNN model is implemented using the Keras Sequential API [44]. Keras is the high-level TensorFlow platform API. It provides an approachable, highly productive, and efficient interface for solving modern deep-learning problems.

After inference, the output predictions are transmitted to the Data scoreboard. The implemented UVM data scoreboard receives each CNN layer output data from the CNN layer driver. It then compares the output data received from the DUT for the last CNN layer with the output data received from the reference model and then checks the resilience of the inference process.

In summary, this chapter explains in detail the implementation of the proposed framework and how it works to test the DLA design for a CNN inference process. This work has been published in [45]. The next chapter demonstrates the error injection methodology as part of the proposed framework.

Chapter 4

Proposed Error Injection Methodology for DLA Verification

This chapter presents the proposed error injection methodology, which is part of our UVM verification framework mentioned in Chapter 3. The proposed error injection methodology is implemented to test the trustworthiness of deep learning accelerators. It can detect errors at an early stage during the DLA verification process with simulation, emulation, and FPGA prototyping. The proposed methodology applies error injection using three mechanisms, as shown in Figure 4.1: (1) applying error injection in the DNN data path by corrupting the DNN layers; feature map, weight, and bias for each DNN layer; (2) applying adversarial attacks on the input image to mitigate the perturbation in input received from cameras and other sensors in the physical world; and (3) applying error injection in the DLA hardware configurations to detect faulty hardware configurations for a DNN that may disrupt the inference process.

In [46], a sequence item-based method for injecting errors using UVM is presented. There have been two different architectures shown: the proactive master and the reactive slave. A proactive master is a testbench component that injects the stimuli it requires at the intended time by the testbench. While a testbench component known as a reactive slave only injects stimulus in response to requests from the DUT. Furthermore, Schwartz et al. investigated injecting errors using sequence modification, transaction modification, or driver error injection [47]. A class extension can be used for transaction error injection either through direct injection or with the help of randomization constraints added to the extended class. Sequences are the best component to modify when it comes to choosing which error to inject. As in sequences, transactions can typically be created and then corrupted. In the case of driver error injection, all types of design inputs are accessible for the injected errors. Therefore, the sequence modification error injection method is used in our framework. It provides flexibility and reusability for choosing which error to inject and in which field in the implemented transaction according to the intended error injection scenario.

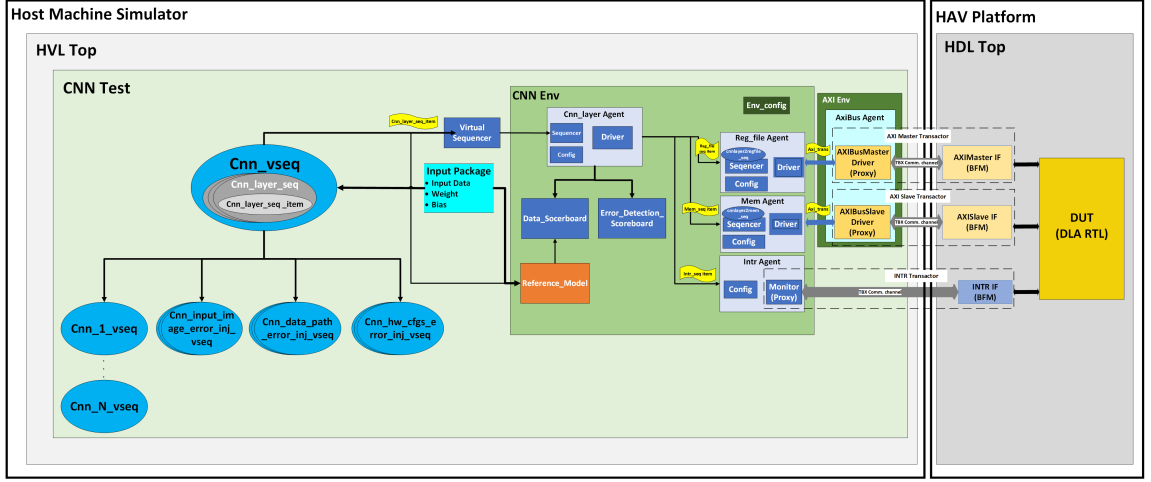


Figure 4.1: UVM testbench architecture with the error injection methodology.

4.1 DNN Data Path Error Injection

Error injection for the DNN data path is done in this framework by extending a new sequence for error injection from the `cnn_vseq` base sequence, which is a UVM virtual base sequence as shown in Figure 4.1. As demonstrated in Chapter 3, this virtual sequence's responsibility is to run the `cnn_layer_seq` sequence, which is used for running a CNN on the DLA. The `cnn_layer_seq` sequence constructs the `cnn_layer_seq_item` transactions that include the DLA hardware configurations, data, weight, and bias specific for each CNN layer. Then these transactions are transmitted using the virtual sequencer to the `cnn_layer_agent`, as shown in Figure 4.1. After that, the `cnn_layer_agent` splits down these received transactions to a lower level of abstraction to be transmitted to the DLA DUT using the `reg_file_seq_item` transaction through the `reg_file_agent` for the DLA configuration registers read/write operations that are sent to the DLA DUT AXI interface from the `AXIBusMaster Driver` which is part of the `AXIBus agent` in AXI environment, and using the `mem_seq_item` transaction through the `mem Agent` for the memory read/write operations that are sent to the DLA DUT DRAM AXI interface from the `AXIBusSlave Driver` which is also part of the `AXIBus agent` in the AXI environment for CNN data, weight, and bias transmission.

Furthermore, the error injection is done by randomly injecting errors for single and multiple values in the CNN internal layers' input data, weight, and bias included in the `cnn_layer_seq_item` transactions for each CNN layer. This is done at different positions in the CNN (at different layers) to study the error propagation in the CNN and the resilience of each layer. Moreover, to investigate the effect of the error injection in each aspect of the DLA inference process in terms of the CNN robustness and the output predictions. Table 4.1 lists the proposed data path error injection testing scenarios that are applicable for any CNN and scalable for different CNN architectures. After that, the DLA DUT response is checked in the Data scoreboard to be compared to the reference model output to study the impact of the injected errors on the DLA DUT CNN output predictions.

Table 4.1: DNN data path error injection testing scenarios.

Testing scenarios	Features to test
1. Single value feature map error injection	Single random incorrect data value injected in a randomly chosen single or multiple CNN layers.
2. Multiple values feature map error injection	Multiple random incorrect data values injected in a randomly chosen single or multiple CNN layers.
3. Single value weight error injection	Single random incorrect weight value injected in a randomly chosen single or multiple CNN layers.
4. Multiple values weight error injection	Multiple random incorrect weight values injected in a randomly chosen single or multiple CNN layers.
5. Single value bias error injection	Single random incorrect bias value injected in a randomly chosen single or multiple CNN layers.
6. Multiple values bias error injection	Multiple random incorrect bias values injected in a randomly chosen single or multiple CNN layers.

4.2 DNN Input Image Error Injection

The error injection is done on the input images in the proposed methodology by generating adversarial examples from the original input data to test the CNN’s robustness. An adversarial example is a sample of input data that has been slightly modified to intentionally make a CNN misclassify it. CNN still fails, even if these changes are so minor that a human observer does not even notice them. Adversarial examples cause security concerns because they could be used to attack deep learning systems, even if the adversary is unable to access the underlying model [48].

The Fast Gradient Sign Method (FGSM) [48] attack is used in the proposed error injection mechanism to generate the adversarial examples. A Python-based script is used to implement this method on the input image during data preparation, as demonstrated in Chapter 3, to generate the adversarial image to be used during the inference process as shown in Figure 4.2. The neural network’s gradients are used by the fast gradient sign method to generate an adversarial example. The method takes an input image and creates a new image that maximizes the loss using the loss gradients with respect to the input image. We refer to this newly created image as the adversarial image. The FGSM method can be explained using the following

equation [49]:

$$\mathbf{adv_img} = \mathbf{in_img} + \epsilon \cdot \text{sign}(\nabla_{\mathbf{in_img}} J(\theta, \mathbf{in_img}, in_lbl)), \quad (4.1)$$

where:

- $\mathbf{adv_img}$: Adversarial image
- $\mathbf{in_img}$: Original input image.
- in_lbl : Original input label.
- ϵ : A perturbation factor.
- θ : Model parameters.
- $\nabla_{\mathbf{in_img}} J(\theta, \mathbf{in_img}, in_lbl)$: The gradient of the loss function $J(\theta, \mathbf{in_img}, in_lbl)$ with respect to the input $\mathbf{in_img}$.

In this method, gradients are calculated with respect to the input image. This is carried out to create an image that maximizes the loss. This is done by calculating the contribution of each pixel in the image to the loss value and then adding a perturbation correspondingly. This method is pretty fast because it is easy to determine how each input pixel contributes to the loss using the chain rule and determining the required gradients. Hence, the goal is to deceive an already-trained model using these adversarial generated images. Therefore, the `cnn_vseq` base sequence is extended to implement a new sequence to run the CNN on the DLA using the generated adversarial images, as shown in Figure 4.1. The DLA DUT inference output is then verified in the data scoreboard compared to that received from the reference model. To evaluate the input image perturbation impact on the CNN output predictions for different perturbation factor values.

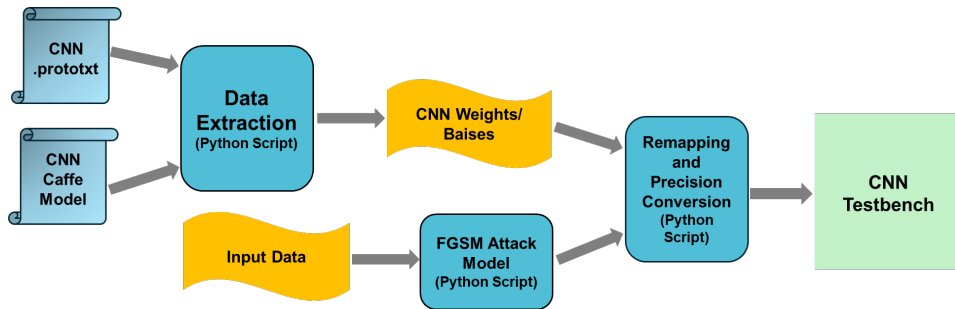


Figure 4.2: Input data and weight pre-processing with input image error injection.

4.3 DLA Hardware Registers Error Injection

Error injection for the DLA hardware configurations register space is done in this framework by extending a new sequence from the `cnn_vseq` base sequence for DLA

configurations error-injecting as shown in Figure 4.1. The hardware configurations of the DLA for a specific CNN included in the `cnn_layer_seq_item` transaction are randomly corrupted such that:

- Changing the configurations related to data, weight, and bias memory addresses.
- Changing the feature map parameters for different CNN layers. For example, changing the input or output data width, height, or number of channels.
- Changing the weight kernels' parameters. For example, changing kernels' width, height, or number of channels.

In addition to that, the `Error_Detection` scoreboard shown in Figure 4.1 is implemented to check and report the DLA behavior in case of incorrect or unexpected hardware configuration during the inference process for a running CNN. The implemented error detection scoreboard is connected to the CNN layer driver through a UVM analysis port.

In summary, the proposed error injection methodology illustrated in this chapter is scalable and reliable for cross-layer error injection and is applicable to any CNN architecture. It provides large coverage, can hit corner cases, and has complete access to the DNN data path between layers and the DLA configurations. Moreover, it is portable across simulation and HAV platforms for emulation and FPGA prototyping. This work has been published in [50]. The next chapter showcases applying the proposed framework on the NVDLA as a case study.

Chapter 5

NVDLA Case Study

In this chapter, the proposed framework is used to verify the NVDLA as a case study. The proposed framework tests running a CNN on the NVDLA by applying a specific testing scenario for a CNN architecture with the required NVDLA hardware blocks' configurations for each CNN layer according to the layer's parameters and then sending the required input, pre-trained weight, and bias to the NVDLA through its DRAM interface.

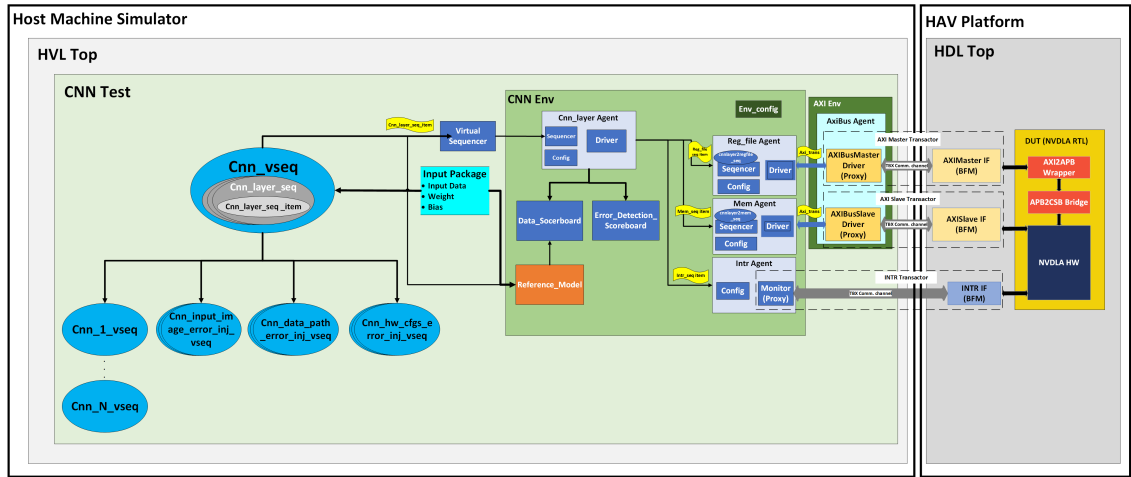


Figure 5.1: NVDLA integration with the UVM testbench architecture.

5.1 NVDLA Integration

To integrate the NVDLA DUT in the UVM testbench, the APB2CSB bridge shown in Figure 5.1 is used to make the NVDLA CSB interface that is used to access the NVDLA configuration registers compatible with the Advanced Peripheral Bus (APB) interface. Then, the AXI2APB wrapper is used to convert from the APB interface to the AXI interface and vice versa; the AXI2APB wrapper acts as a bridge between the two buses to be then connected to the AXIMaster virtual interface. Furthermore, the NVDLA DBB interface is then connected directly to the AXISlave virtual interface, while the NVDLA interrupt signal is connected to the testbench through the INTR virtual interface, as shown in Figure 5.1.

5.2 NVDLA Registers Configuration

The NVDLA register configurations are done in this framework in the CNN layer driver inside the `cnn_layer` agent according to the hardware configurations for each CNN layer in the received `cnn_layer_seq` item transaction as demonstrated in Chapter 3 and Chapter 4. To hide the reprogramming latency that may occur when the hardware is idle between two consecutive hardware layers while waiting for reprogramming after sending a "done" interrupt to the testbench. The NVDLA relies on the idea of ping-pong register programming for per-hardware-layer registers. The CNN layer driver configures the NVDLA registers for each hardware block (subunit) and then sends these configurations as a register read/write operation to the NVDLA DUT with the help of the `reg_file` driver and the `AXIBusMaster` driver, as mentioned before. Most NVDLA subunits have two sets of registers; the testbench can program the second group in the background while the subunit operates using the configuration from the first register set, and then the testbench sets the second group's "enable" bit when it is finished. The hardware will move to the second group if the second group's "enable" bit has already been set after processing the layer defined by the first register set and clearing its "enable" bit. Hardware becomes idle until programming is finished if the "enable" bit for the second group is not yet set. Next, the procedure is repeated, with the first group now functioning as the "shadow" group to which the testbench writes in the background and the second group as the active group. With the help of this method, the hardware may easily switch between active layers without wasting any cycles on testbench configuration [51].

The NVDLA core is implemented based on a series of pipeline stages. All or part of the hardware layers are handled by each pipeline stage. These are the pipeline's stages [51]:

- Convolution DMA (CDMA).
- Convolution Buffer (CBUF).
- Convolution Sequence Controller (CSC).
- Convolution MAC Array (CMAC).
- Convolution Accumulator (CACC).
- Single Data Processor, Read DMA (SDP_RDMA).
- Planar Data Processor (PDP).
- Planar Data Processor, Read DMA (PDP_RDMA).

The convolution core pipeline consists of the first five pipeline stages mentioned above; all of these pipeline stages, except the CBUF and the CMAC, create hardware layer layers through linked ping-pong buffers [51].

As shown in Figure 5.2, the ping-pong technique is considered in each pipeline stage's register file. Three register groups form each register file implementation: a separate non-shadowed group called a "single register group" makes up the third register group, while the two ping-pong groups (duplicated register groups 0 and 1) share the same addresses. The CONSUMER register value shows which register the data path is sourcing from, whereas the PRODUCER register field in the POINTER register is used to specify which of the ping-pong groups is to be accessible from the CSB interface. Both pointers choose group 0 by default. The names of registers indicate which register set they are part of; if a register's name begins with D_, it is part of a duplicated register group; if not, it is part of a single register group. The ping-pong groups' registers serve as hardware layer configuration parameters that are configured by the testbench based on the CNN layers' parameters. An enable register field exists in each group, and it is set by the testbench and cleared by hardware. Before setting the enable bit, the testbench sets all other fields in the group; this indicates that the hardware layer is prepared for execution. Until the hardware layer finishes executing, all writes to register groups that have the enable bit set will be silently dropped. At that point, the hardware will clear the enable bit [51].

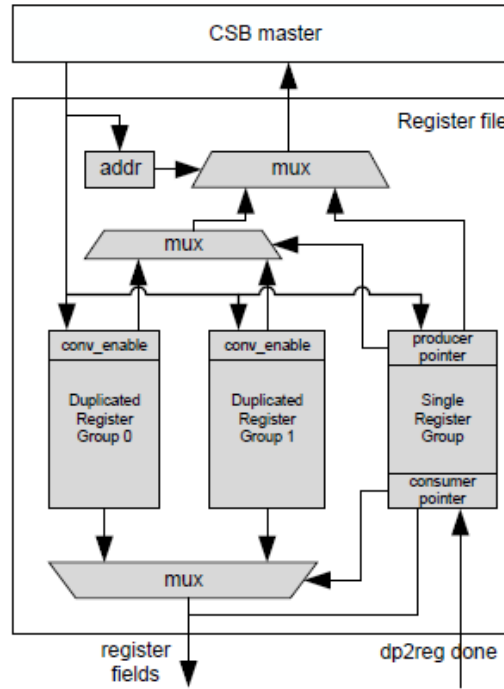


Figure 5.2: NVDLA ping-pong register file [51].

5.3 NVDLA Data and Weight Memory Mapping

To run a CNN on NVDLA, the image data, weight, and bias need to be mapped into the memory's address space. The data is then loaded into the CBUF by the CDMA for computation. For this reason, the previously mentioned process of

extracting the bias and weight in Chapter 3 and Chapter 4 is followed according to the NVDLA configurations. The data of each layer in the CNN is mapped, and then precision conversion is done, and the resulting mapped and precision converted data is written in an input package that the testbench can read, as shown in Figure 5.1.

5.3.1 Input Image Mapping

To improve MAC utilization, input image data convolution is used in this architecture for a CNN input layer. The mapping of the image data done by the convolution buffer is implemented such that each input pixel line component and the left and right padding are compactly stored in CBUF entries. For example, in a Red Green Blue Alpha (RBGA) image, the elements are mapped in the following order: R -> G -> B -> A, while in a Red Green Blue (RGB) image, the elements are mapped in the following order: R -> G -> B [52].

5.3.2 Feature Data Mapping

The feature data format that NVDLA supports with Direct Convolution (DC) is the one utilized for the other internal CNN layers. As shown in Figure 5.3, the feature data elements of one layer are grouped into a three-dimensional data cube with width (W), height (H), and channel (C) size. The data cube is divided into 1x1x32 byte tiny atom cubes in order to do the memory mapping. Such that, data may be added to the channel end if the original data is not 32byte aligned in the C direction. Moreover, the atom cube mapping at surface or line boundaries may occur in-compactly or adjacently. However, their alignment is always 32-byte. Next, a compact mapping of every small cube in the feature data cube is performed [52].

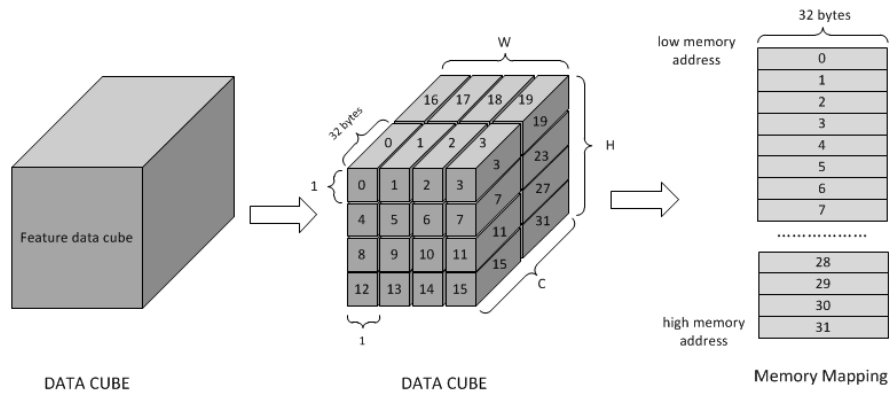


Figure 5.3: Packed feature data [52].

5.3.3 Weight Mapping

The weights in this framework are not compacted. The width, height, channel, and number of kernels are the four dimensions of the original weight data. A group of three-dimensional (3D) data cubes can be formed using these dimensions. One data cube is referred to as a kernel. Grouping the kernels is necessary before mapping. Four types of weight data are supported by the NVDLA. These are the weights for image input, weights for deconvolution, weights for Winograd convolution, and weights for DC. Moreover, sparse compression and channel post-extension are the two weight options that can be used to enhance DLA performance. In DC mode, a single weight group has 16 kernels if the weights are in the FP16 or INT16 format. Each kernel should be divided into $1 \times 1 \times 64$ -element small cubes. Each small cube is 128 bytes for FP16 or INT16 format. Furthermore, the last group may have fewer kernels if the total number of kernels is not a multiple of 16. Figure 5.4 shows the weight mapping for DC mode within a single group. Once the first weight group has been mapped, the second weight group is also compactly mapped. Then zeros must be added to each mapped weight for 128-byte alignment; otherwise, any zeros should not be added in between the group boundaries. Similar to weight for DC, weight mapping is used for input image convolution as well. The key difference is that input image weight mapping requires a prior extra channel extension step, which is mandatory as the first step for the input image weight mapping. Its main concept is based on converting all the weights that are in the same line to a single channel, then applying the same mapping steps of weight for DC [52].

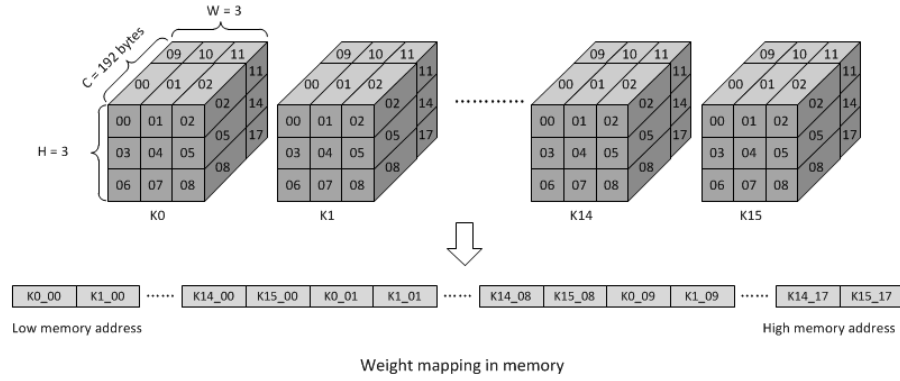


Figure 5.4: DC one group weight mapping [52].

This chapter evaluated the proposed framework's scalability and reusability by verifying the NVDLA as a case study. This work has been published in [45, 50]. The next chapter discusses the experimental results.

Chapter 6

Experimental Results

This chapter illustrates the experimental results of using the proposed framework to verify the NVDLA inference function by running different CNN testing scenarios with and without the error injection methodology for simulation and emulation. The proposed framework is applicable to sophisticated custom and standard CNN architectures as per the CNN sequence for programming the NVDLA core. The NVDLA DUT is integrated with the UVM testbench through AXI-based interfaces, as demonstrated in Chapter 5. Each running testing scenario configures the NVDLA hardware blocks required for each CNN layer according to the layer’s parameters and then sends the required input data, pre-trained weight, and bias to the NVDLA through its DRAM interface. The tool used for simulation is the QuestaSim simulator [53]. Moreover, the platform used for emulation is Veloce Strato [10].

6.1 Single CNN Convolution Layer Results

The first testing scenario runs a single CNN convolution layer on the NVDLA with the inference parameters shown in Table 6.1 for simulation and emulation. The NVDLA supports the following three data types: FP16, INT16, and INT8. To improve calculative performance, INT16 precision is used in this testing scenario. Moreover, the ReLU activation function is applied in this convolution layer. Hence, the NVDLA is configured according to the mentioned parameters.

Table 6.1: The single CNN convolution layer parameters.

Layer	Input	Filters No.	Filter size	Stride	Output
Convolution	8*8*32	16	3*3*32	1	1*1*16

The simulation output is then compared with the trace-player direct testbench test case supplied with the NVDLA release with the same inference parameters [54]. The comparison showed that they have the same simulation output. Furthermore, by comparing the simulation runtime, it is shown in Table 6.2 that the proposed framework simulation runtime for a single CNN layer is reduced by 17x compared to that of the trace-player direct testbench. In addition to that, the same single CNN convolution layer test case is run on the emulation platform with the runtime value shown in Table 6.2.

Table 6.2: The single CNN convolution layer simulation and emulation runtime.

CNN	Simulation runtime	Total emulation runtime
Proposed framework single CNN convolution layer test case	20.48 us	3.12s
NVDLA direct testbench test case single CNN convolution layer	355.23 us	-

6.2 LeNet-5 CNN Results

The second testing scenario is running LeNet-5 CNN on the NVDLA. LeNet-5 CNN was proposed by LeCun in 1998 [55], which was successfully applied to handwritten digit recognition. LeNet-5 consists of the following layers: an input layer, two convolution layers each followed by a pooling layer, two fully connected layers, and an output layer. The architecture and the parameters used for inference are mentioned in Table 6.3 [32]. The NVDLA is configured according to the LeNet-5 parameters. The dataset used for testing is the MNIST handwritten digit dataset. The grayscale 28*28 images are used as input data for LeNet-5. To improve calculative accuracy and performance, FP16 precision is considered in this testing scenario. Therefore, the weights with floating point precision are converted to FP16 format. The precision conversion from a 16-bit binary to a 16-bit floating point and vice versa is done in this framework according to the IEEE-754-2008 standard using the below equation, at which each variable is reflected in Figure 6.1 [56]:

$$(-1)^{sign} * 2^{(exponent-15)} * 1.fraction_2. \quad (5.1)$$

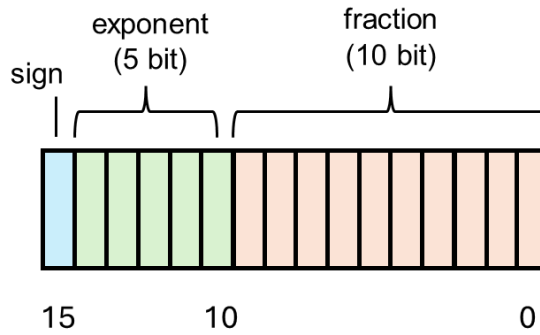


Figure 6.1: 16-bit binary number [56].

Table 6.3: The LeNet-5 CNN parameters.

Layer	Input	Filter	Stride	Output
Convolution 1	28*28*1	5*5*1*20	1	24*24*20
Pooling 1	24*24*20	2*2	2	12*12*20
Convolution 2	12*12*20	5*5*20*50	1	8*8*50
Pooling 2	8*8*50	2*2	2	4*4*50
Fully Connected 1	4*4*50	4*4*50*50	1	1*1*50
Fully Connected 2	1*1*50	1*1*50*10	1	1*1*10

The simulation and emulation results are then compared to those when running the same LeNet-5 CNN with the NVDLA software environment [57]. The comparison showed that both have almost similar output accuracy rates for a regression run over a random MINST dataset testing samples. Table 6.4 and Table 6.5 show LeNet-5 output examples for the proposed framework and the NVDLA software environment for two different input images. However, the simulation runtime for the proposed framework is reduced by more than 200x compared to that using the NVDLA software, as shown in Table 6.6. Similarly, the emulation runtime for the proposed framework is reduced by more than 10x compared to that using the NVDLA software, as shown in Table 6.6.

Table 6.4: LeNet-5 output for digit 8 input image.


Input image	
Proposed framework LeNet-5 CNN test case	<pre>[m_data_sb] CNN OUTPUT LABEL = 8 [m_data_sb] CNN REF MODEL OUTPUT LABEL = 8 [m_data_sb] ***** CORRECT CNN OUTPUT LABEL: 8 ***** [cnn_data_scoreboard] ***** CNN Data compare has been done *****</pre>
NVDLA software environment LeNet-5 CNN	<pre>Test pass root@m3ulcb:/home/mynvdla# cat output.dimg 0 0 0 0 0 0 0 0 1 0 root@m3ulcb:/home/mynvdla#</pre>

Table 6.5: LeNet-5 output for digit 2 input image.


Input image	
Proposed framework LeNet-5 CNN test case	<pre>[m_data_sb] CNN OUTPUT LABEL = 2 [m_data_sb] CNN REF MODEL OUTPUT LABEL = 2 [m_data_sb] ***** CORRECT CNN OUTPUT LABEL: 2 ***** [cnn_data_scoreboard] ***** CNN Data compare has been done *****</pre>
NVDLA software environment LeNet-5 CNN	<pre>Test pass root@m3ulcb:/home/mynvdla# cat output.dimg 0 0 1 0 0 0 0 0 0 root@m3ulcb:/home/mynvdla#</pre>

Table 6.6: LeNet-5 simulation and emulation runtime

CNN	Simulation runtime	Total emulation runtime
Proposed framework LeNet-5 CNN test case	0.144 ms	19.42s
NVDLA software environment LeNet-5 CNN	32.91 ms	210.98s

The mentioned runtime values for the proposed framework are for running the CNN on the NVDLA after the data preparation phase is done using Python-based scripts whose running time is insignificant. The reason behind this reduction in runtime is that the NVDLA software environment is running on a virtual platform with instruction-by-instruction execution, which is more realistic. However, the proposed UVM framework is a native code execution that is running very fast on a host machine, it also has larger coverage for running any CNN, hitting corner cases, and provides faster and easier debugging. Therefore, the best practice is to start the testing and verification process of a DLA using the proposed framework until the design stability is reached, then proceed with the realistic software execution.

In addition to that, the simulation and emulation wall time are measured on the same host machine for the LeNet-5 CNN proposed framework test case. Figure 6.2 shows that the LeNet-5 CNN test case runtime is reduced by more than 3x for emulation compared to simulation. Accordingly, accelerating the UVM testbench for emulation speeds up the testing process, particularly for operating complicated DNN architectures and when the complexity of the DLA design increases.

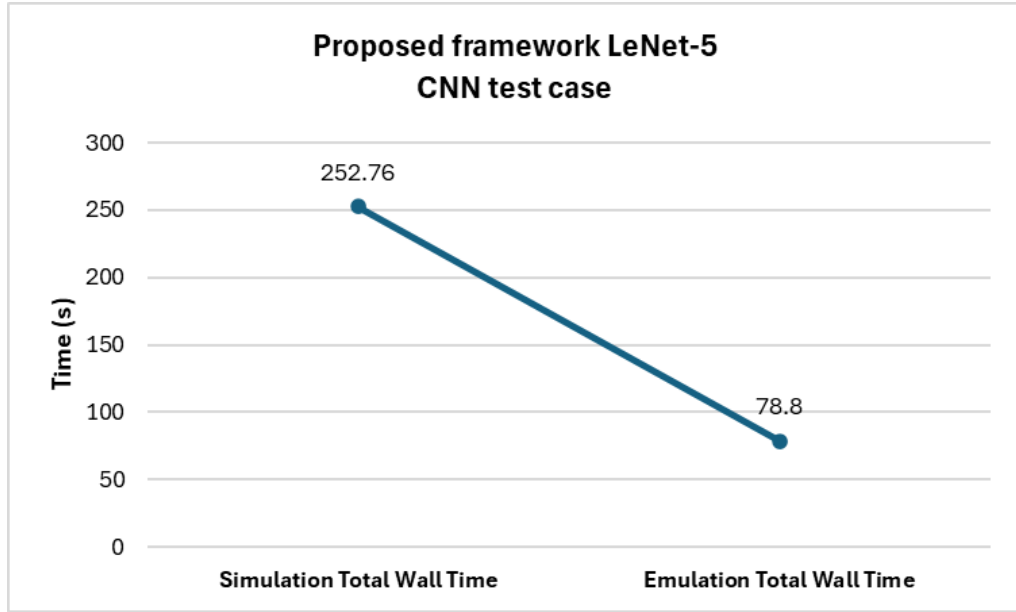


Figure 6.2: Proposed framework LeNet-5 CNN test case simulation versus emulation wall time.

Code coverage is collected by the simulation tool for both the verification testbench and the NVDLA software. The code coverage includes statement coverage, branch coverage, and conditional coverage. These measurements assess the execution of the code from different perspectives. The NVDLA's total code coverage for the LeNet-5 CNN is 45% for both the verification testbench and the NVDLA software, as shown in Figure 6.3 and Figure 6.4. The reason for this is due to the specific hardware configurations sent to the NVDLA DUT for the NVDLA registers according to the LeNet-5 CNN-specific architecture and inference parameters. This causes the execution of RTL code statements depending on the type of running hardware layers and the CNN architecture complexity, which may or may not consume all the DLA hardware blocks to achieve 100% code coverage.

Instance ↑	Branches	Conditions	Expressions	FSM States	FSM Transitions	Statements	Toggles	Total
Search...	Search...	Search...	Search...	Search...	Search...	Search...	Search...	Search...
Total	55.54%	27.6%	10.7%	68%	35.41%	80.59%	39.04%	45.27%
nvdla_mainTop	-	-	-	-	-	-	68.62%	68.62%
u_partition_a	51.51%	25.57%	21.6%	-	-	94.58%	16.93%	42.04%
u_partition_c	53.63%	16.88%	8.27%	80%	43.58%	83.58%	13.89%	42.83%
u_partition_ma	58.2%	61.09%	48.7%	-	-	77.01%	57.9%	60.58%
u_partition_mb	54.93%	60.44%	41.99%	-	-	75.03%	51.55%	56.79%
u_partition_o	56.78%	4.58%	6.57%	20%	0%	82.23%	10.62%	25.82%
u_partition_p	55.39%	3.84%	9.07%	-	-	83.91%	17.88%	34.02%

Figure 6.3: Proposed framework LeNet-5 CNN test case code coverage analysis.

Instance ↑	Branches	Conditions	Expressions	FSM States	FSM Transitions	Statements	Toggles	Total
Search...	Search...	Search...	Search...	Search...	Search...	Search...	Search...	Search...
Total	55.65%	28.57%	10.8%	68%	35.41%	80.63%	39.12%	45.45%
nvdla_mainTop	-	-	-	-	-	-	69.85%	69.85%
u_partition_a	51.51%	25.57%	21.58%	-	-	94.58%	16.93%	42.03%
u_partition_c	53.6%	16.89%	8.24%	80%	43.58%	83.56%	13.92%	42.83%
u_partition_ma	58.18%	61.09%	48.64%	-	-	76.98%	57.97%	60.57%
u_partition_mb	54.91%	60.44%	41.95%	-	-	75.02%	51.66%	56.8%
u_partition_o	57.5%	8.17%	6.98%	20%	0%	82.53%	10.76%	26.56%
u_partition_p	55.33%	3.83%	9.02%	-	-	83.89%	17.89%	33.99%

Figure 6.4: NVDLA software environment LeNet-5 CNN code coverage analysis.

6.3 Emulation Analysis

The emulation analysis shown in Table 6.7 indicates that the proposed framework has a much lower software time and a much lower number of TBX clock cycles, which is due to the optimization done in the proposed framework for interaction with the hardware DLA DUT and the testbench running as software on the host machine simulator either for the register space configurations or for the memory read and write operations. Moreover, there is a simple increase in the hardware area, mainly the Look Up Tables (LUTs) and Registers on the emulation platform used with the proposed framework compared to the NVDLA software, due to the testbench synthesizable added XRTL (Synthesizable SV RTL superset) hardware interface.

Table 6.7: Emulation analysis summary.

CNN	Proposed framework single CNN convolution layer test case	Proposed framework LeNet-5 CNN test case	NVDLA software environment LeNet-5 CNN
Total runtime	3.12s	19.42s	210.98s
Software time	3.11s	18.11s	200.01s
Hardware time	0.01s	0.05s	10.97s
Communication time	0s	1.27s	0s
Total number of TBX clocks	1290599	19081974	210489636
Total number of LUTs	5600812	5600812	5588806
Total number of registers	1563794	1563794	1539950

6.4 Error injection in CNN Results

The implemented error injection testing scenarios are applied to the single CNN convolution layer and the LeNet-5 CNN as a case study. The proposed error injection methodology is applicable to any CNN architecture by extending a new CNN sequence.

6.4.1 DNN Data Path Error Injection

The implemented error injection testing scenarios are applied for running the single CNN convolution layer test case on the NVDLA by randomly injecting errors in weight and the layer's input data (feature map) during the inference for both simulation and emulation. Figure 6.5 shows the simulation and emulation regression results for the convolution layer with the different random error injection testing scenarios. The convolution layer is more sensitive to multiple data errors than to multiple weight errors. Moreover, single data and single weight errors are almost masked in the convolution layer due to the presence of the ReLU activation function.

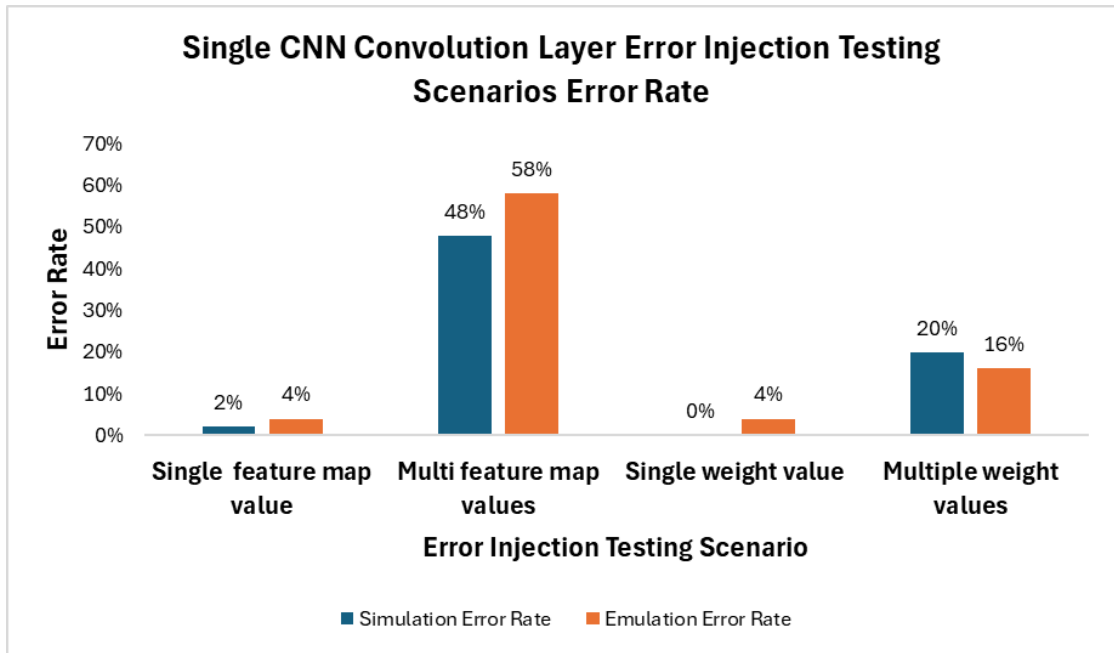


Figure 6.5: Single CNN convolution layer data path error injection testing scenarios error rate.

Furthermore, error injection testing scenarios are implemented for LeNet-5 CNN running on the NVDLA by randomly injecting errors in different layers in weight, bias, and the internal layers' input data (feature maps between layers) during the inference for both simulation and emulation. The error injection is done by inserting random and incorrect single or multiple values of weight, bias, or internal layers' input data for a randomly chosen layer. Simulation and emulation regression are run for the implemented error injection testing scenarios over random MINST dataset testing samples.

As demonstrated in Figure 6.6, the results indicate that in the LeNet-5 CNN, the majority of single-value input data errors within the internal layers are masked and do not impact the output predictions of the final layer. Moreover, the LeNet-5 CNN is sensitive to the multiple values in internal layers' input data errors. However, some of them are masked by the pooling layers if they are injected into the convolution layers. Furthermore, the LeNet-5 CNN masks most of the single-value errors in weight. However, some errors in the second convolution layer were propagated and

corrupted the last layer output predictions as the ReLU activation function hardware configuration is disabled in this convolution layer. Moreover, the LeNet-5 CNN is sensitive to multiple-value weight errors, mainly if injected in the convolution layers. For the bias, the LeNet-5 CNN masks most of the single-value errors. However, the LeNet-5 CNN is sensitive to the multiple-value bias errors and propagates them as they corrupt the output predictions, except for those injected in the fully connected layers.

In summary, the LeNet-5 CNN is sensitive to the internal layers' multiple values of input data corruption and weight corruption more than that of bias corruption, and less sensitive to single value corruption in data, weight, and bias propagation between layers.

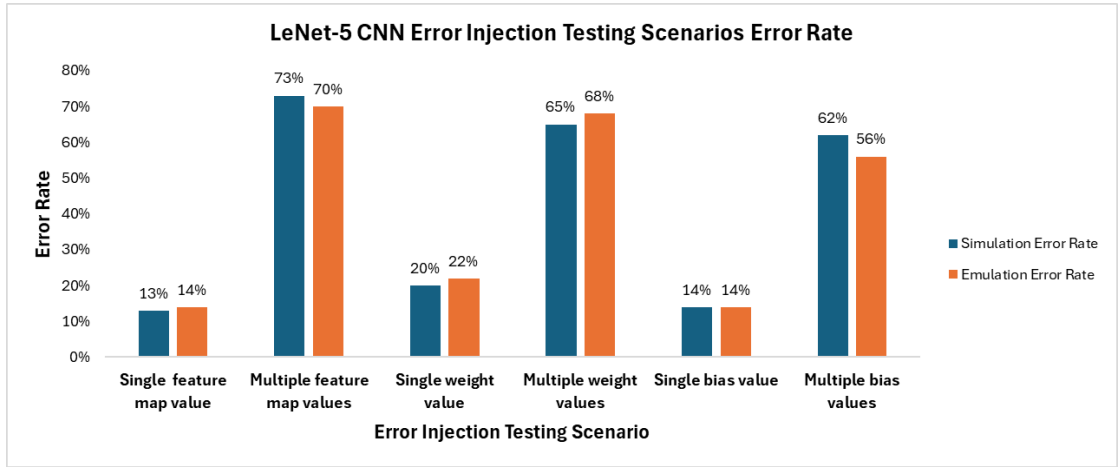


Figure 6.6: LeNet-5 CNN data path error injection testing scenarios error rate.

6.4.2 Input Image Error Injection

The implemented testing scenario for input image error injection is applied to the LeNet-5 CNN test case running on the NVDLA, using adversarial images as input images for simulation and emulation. Figure 6.7 shows the regression results for this test case over random MINST dataset samples with different perturbation factor values. The LeNet-5 CNN accuracy rate decreases with high values of ϵ as the fast gradient sign method adds noise scaled by ϵ to the input image.

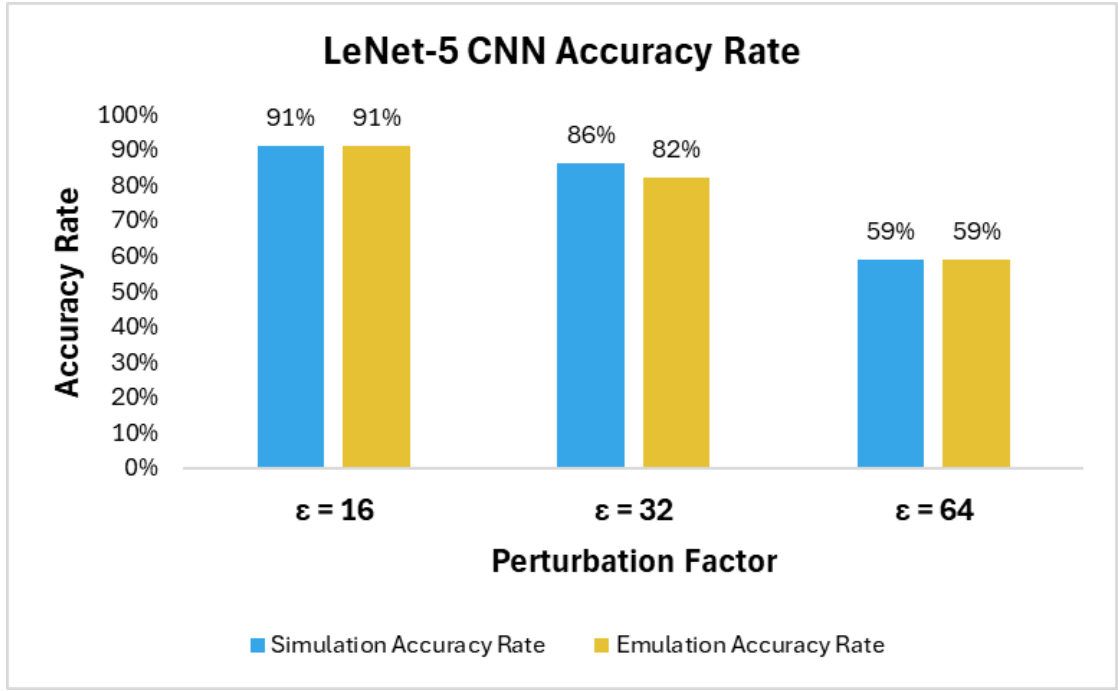


Figure 6.7: The accuracy rate of the LeNet-5 CNN with adversarial images.

As shown in Figure 6.6, Figure 6.5, and Figure 6.7, some of the error injection testing scenarios running on the emulator have higher error rates compared to those running for simulation. This is because emulation tends to hit more bugs compared to simulation, as it executes the design in a more realistic and faster environment and operates at a lower level of abstraction. This combination leads to a higher likelihood of uncovering bugs that might not be exposed during simulation. However, the trade-off is that while emulation can detect more bugs, simulation often provides better tools for visibility and debugging.

The proposed error injection methodology is faster compared to other frameworks introduced in the literature that are based on software instruction-by-instruction execution because the proposed UVM framework is a native code execution that runs very fast on a host machine. The proposed error injection methodology didn't consume any extra simulation or emulation runtime for the running CNNs on the DLA. Moreover, the proposed error injection methodology did not add any performance overhead, as no extra processing is done in the proposed error injection mechanisms.

Moreover, this framework has a larger coverage for running any CNN, hitting corner cases, and is faster and easier to debug. The proposed framework added more visibility during NVDLA testing and debugging, allowing direct and full access to the NVDLA register space for configuration with less runtime. Table 6.8 summarizes the main differences between the proposed framework and the NVDLA software environment.

Table 6.8: The proposed framework versus the NVDLA software environment

Performance aspects	The proposed framework	NVDLA software environment
Run time speed	✓	-
Coverage (running different CNNs)	✓	-
Hitting corner cases	✓	-
Visibility (accessing and changing DLA configurations during runtime)	✓	-
Realisticity	-	✓
Error injection capability	✓	-

In summary, this chapter shows the outstanding performance of the proposed framework against the NVDLA trace player direct test cases, and the NVDLA software environment. These results have been published in [45, 50]. The next chapter concludes the proposed work and discusses the future work of this research.

Chapter 7

Conclusion and Future Work

This chapter lists all contributions that have been made in this research work and provides future work for this thesis.

7.1 Contributions

The main contribution and outcomes of this thesis are as follows:

1. Implementing a novel verification framework for verifying deep learning accelerators' inference function using a generic and scalable UVM environment to run convolutional neural networks with different inference parameters for simulation and emulation.
2. Implementing a novel UVM-based error injection methodology as part of the proposed verification framework to test the trustworthiness of complex DLA designs for each CNN layer mapped on hardware, mainly in the presence of data corruption either due to hardware faults or input perturbations.
3. The implemented error injection methodology added more flexibility and scalability as it introduced cross-layer error injection in the DNN.
4. The NVDLA is used as a case study of the DLA design under verification to prove the robustness of our proposed verification framework with the error injection methodology.
5. Verifying the NVDLA by running a single convolution layer CNN and running the LeNet-5 CNN with and without error injection testing scenarios as an example for CNNs; however, the proposed framework is applicable for any CNN architecture.
6. Comparing the NVDLA direct testbench single CNN convolution layer test case results with those of our framework. And comparing the NVDLA software environment's LeNet-5 CNN results with those of our framework.
7. The implemented verification framework simulation runtime for a single CNN layer is reduced by 17x compared to that of the NVDLA trace-player direct testbench. Moreover, the simulation runtime for the LeNet-5 is reduced by more than 200x using the implemented framework compared to that using the NVDLA software. Similarly, the LeNet-5 CNN emulation runtime for the

implemented framework is reduced by more than 10x compared to that for the NVDLA software. This reduction in simulation and emulation runtime helps to handle complex DLA designs, mainly in figuring out and fixing the design issues that may appear due to the complex computations involved in each CNN layer mapped on hardware. In addition to that, the LeNet-5 CNN test case runtime is reduced by more than 3x for emulation compared to simulation. It causes the testing process to speed up, particularly when testing complex DNN architectures and when the complexity of the DLA design increases.

8. The error injection experimental results for running a single CNN convolution layer and running the LeNet-5 CNN on NVDLA show that a CNN is more sensitive to the internal layers' multiple values of input data, weight, and bias corruption compared to a single value corruption in data, weight, and bias propagation between layers.

7.2 Future Work

Our future work would be to implement testing scenarios to run other more complex CNNs with and without defense mechanisms on the NVDLA to check the system's stability and analyze the resilience of such CNNs against faults and attacks. The proposed verification framework was tested on simulation and hardware emulation; we wish to expand that to the FPGA prototyping system as well to achieve better performance.

References

- [1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” *Advances in neural information processing systems*, vol. 25, 2012.
- [2] C.-s. Oh and J.-m. Yoon, “Hardware acceleration technology for deep-learning in autonomous vehicles,” in *2019 IEEE International Conference on Big Data and Smart Computing (BigComp)*, IEEE, 2019, pp. 1–3.
- [3] G. Cesarano, “FPGA implementation of a deep learning inference accelerator for autonomous vehicles,” Ph.D. dissertation, Politecnico di Torino, 2018.
- [4] S. Chakradhar, M. Sankaradas, V. Jakkula, and S. Cadambi, “A dynamically configurable coprocessor for convolutional neural networks,” in *Proceedings of the 37th annual international symposium on Computer architecture*, 2010, pp. 247–257.
- [5] M. Horowitz, “1.1 computing’s energy problem (and what we can do about it),” in *2014 IEEE international solid-state circuits conference digest of technical papers (ISSCC)*, IEEE, 2014, pp. 10–14.
- [6] Y.-N. Yun, J.-B. Kim, N.-D. Kim, and B. Min, “Beyond UVM for practical SoC verification,” in *2011 International SoC Design Conference*, IEEE, 2011, pp. 158–162.
- [7] *Universal Verification Methodology (UVM) 1.2 User’s Guide, 2015.* https://www.accellera.org/images/downloads/standards/uvm/uvm_users_guide_1.2.pdf (accessed Aug. 16, 2024).
- [8] S. Sutherland and T. Fitzpatrick, “UVM Rapid Adoption: A Practical Subset of UVM,” *Proceedings of DVCon*, 2015.
- [9] S. Hassoun, M. Kudluga, D. Pryor, and C. Selvidge, “A transaction-based unified architecture for simulation and emulation,” *IEEE transactions on very large scale integration (VLSI) systems*, vol. 13, no. 2, pp. 278–287, 2005.
- [10] *Veloce CS system - IC emulation and prototyping, Siemens Digital Industries Software.* <https://eda.sw.siemens.com/en-US/ic/veloce/> (accessed Aug. 16, 2024).
- [11] S. Albawi, T. Abed Mohammed, and S. ALZAWI, “Understanding of a Convolutional Neural Network,” Aug. 2017.
- [12] R. Yamashita, M. Nishio, R. K. G. Do, and K. Togashi, “Convolutional neural networks: An overview and application in radiology,” *Insights into imaging*, vol. 9, pp. 611–629, 2018.
- [13] D. Stutz and L. Beyer, “Understanding Convolutional Neural Networks,” 2014.

- [14] G. Li *et al.*, “Understanding error propagation in deep learning neural network (DNN) accelerators and applications,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017, pp. 1–12.
- [15] *ISO 26262-1:2018, ISO*. <https://www.iso.org/standard/68383.html#lifecycle>.
- [16] S. Pappalardo, A. Ruospo, I. O’Connor, B. Deveautour, E. Sanchez, and A. Bosio, “A fault injection framework for AI hardware accelerators,” in *2023 IEEE 24th Latin American Test Symposium (LATS)*, IEEE, 2023, pp. 1–6.
- [17] Z. Chen, N. Narayanan, B. Fang, G. Li, K. Pattabiraman, and N. DeBardeleben, “TensorFI: A flexible fault injection framework for tensorflow applications,” in *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, IEEE, 2020, pp. 426–435.
- [18] M. Abadi *et al.*, “TensorFlow: A system for Large-Scale machine learning,” in *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, 2016, pp. 265–283.
- [19] A. Paszke *et al.*, “Pytorch: An imperative style, high-performance deep learning library,” *Advances in neural information processing systems*, vol. 32, 2019.
- [20] 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)*, IEEE, 2020, pp. 25–31.
- [21] R. Gräfe, Q. S. Sha, F. Geissler, and M. Paulitsch, “Large-scale application of fault injection into PyTorch models—an extension to pytorchFI for validation efficiency,” in *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks-Supplemental Volume (DSN-S)*, IEEE, 2023, pp. 56–62.
- [22] C. Bolchini, L. Cassano, A. Miele, and A. Toschi, “Fast and accurate error simulation for CNNs against soft errors,” *IEEE Transactions on Computers*, vol. 72, no. 4, pp. 984–997, 2022.
- [23] Z. Chen, N. Narayanan, B. Fang, G. Li, K. Pattabiraman, and N. DeBardeleben, “TensorFI: A flexible fault injection framework for tensorflow applications,” in *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, IEEE, 2020, pp. 426–435.
- [24] S. K. S. Hari, T. Tsai, M. Stephenson, S. W. Keckler, and J. Emer, “SASSIFI: An architecture-level fault injection tool for GPU application resilience evaluation,” in *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, IEEE, 2017, pp. 249–258.
- [25] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, “Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks,” *IEEE journal of solid-state circuits*, vol. 52, no. 1, pp. 127–138, 2016.
- [26] *Home | Flex Logix – Reconfigurable Computing Flex Logix*, [lex-logix.com](https://flex-logix.com/), Jun. 10, 2019. <https://flex-logix.com/> (accessed Aug. 16, 2024).
- [27] *Nvidia Jetson xavier series*, NVIDIA, <https://www.nvidia.com/enus/autonomous-machines/embedded-systems/jetson-agx-xavier> (accessed Aug. 16, 2024).

- [28] *NVDLA primer - NVDLA Documentation*, [nvdla.org. http://nvdla.org/primer.html](http://nvdla.org/primer.html) (accessed Aug. 16, 2024).
- [29] *Hardware Architectural Specification - NVDLA Documentation*, [nvdla.org. http://nvdla.org/hw/v1/hwarch.html](http://nvdla.org/hw/v1/hwarch.html) (accessed Aug. 16, 2024).
- [30] *Integrator's Manual - NVDLA Documentation*, [nvdla.org. http://nvdla.org/hw/v2/integration_guide.html](http://nvdla.org/hw/v2/integration_guide.html) (accessed Aug. 16, 2024).
- [31] M. Khan, V. K. Kodavalla, and A. U. Sameer, "Verification Methodology of Vision Based Hardware Accelerators in System-on-Chip," in *2020 International Conference on Industry 4.0 Technology (I4Tech)*, IEEE, 2020, pp. 39–45.
- [32] S. Feng, J. Wu, S. Zhou, and R. Li, "The implementation of LeNet-5 with NVDLA on RISC-V SoC," in *2019 IEEE 10th International Conference on Software Engineering and Service Science (ICSESS)*, IEEE, 2019, pp. 39–42.
- [33] G. Zhou, J. Zhou, and H. Lin, "Research on NVIDIA deep learning accelerator," in *2018 12th IEEE International Conference on Anti-counterfeiting, Security, and Identification (ASID)*, IEEE, 2018, pp. 192–195.
- [34] S. Ramakrishnan, "Implementation of a deep learning inference accelerator on the FPGA.," 2020.
- [35] 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)*, IEEE, 2018, pp. 322–329.
- [36] W. Liu and C.-H. Chang, "A forward error compensation approach for fault resilient deep neural network accelerator design," in *Proceedings of the 5th Workshop on Attacks and Solutions in Hardware Security*, 2021, pp. 41–50.
- [37] *Veloce Strato CS Emulation Platform for IC Verification*, Siemens Digital Industries Software, 2024. <https://eda.sw.siemens.com/en-US/ic/veloce/strato-hardware/> (accessed Aug. 16, 2024).
- [38] *MIPS leverages Siemensâ€™TM Veloce proFPGA platform*, Siemens Digital Industries Software, 2023. <https://newsroom.sw.siemens.com/en-US/mips-veloce-profpga/> (accessed Aug. 16, 2024).
- [39] *Emulation and simulation; invaluable tools for IC verification - Verification Horizons*, [blogs.sw.siemens.com, Dec. 06, 2016. https://blogs.sw.siemens.com/verificationhorizons/2016/12/06/emulation-and-simulation-invaluable-tools-for-ic-verification/](https://blogs.sw.siemens.com/verificationhorizons/2016/12/06/emulation-and-simulation-invaluable-tools-for-ic-verification/) (accessed Aug. 16, 2024).
- [40] V. Rousseau, Anoop, G. Nagrecha, V. Kankariya, V. Mamidi, and D. Hurakadli, "Techniques for robust transaction based acceleration," in *DVCON, Bangalore, India*, 2016.
- [41] V. Billa and S. Haran, "Trials and tribulations of migrating a native UVM testbench from simulation to emulation," in *DVCON, Bangalore, India*, 2017.

- [42] M. K. A. M. Saad, “An Automatic Generation of NoC Architectures: An Application-Mapping Approach,” M.S. thesis, Computer and Systems Engineering, faculty of engineering, Ain Shams University, 2020.
- [43] *SCE-mi (standard Co-Emulation Modeling Interface)*, Accellera Systems Initiative. <https://www.accellera.org/downloads/standards/sce-mi> (accessed Aug. 16, 2024).
- [44] *TensorFlow, Keras / TensorFlow Core / TensorFlow, TensorFlow*, 2019. <https://www.tensorflow.org/guide/keras> (accessed Aug. 16, 2024).
- [45] R. Aboudeif, T. A. Awaad, M. AbdelElSalam, and Y. Ismail, “UVM Based Verification Framework for Deep Learning Hardware Accelerator: Case Study,” in *International Conference on Electrical, Computer and Energy Technologies, Sydney, Australia*, 2024.
- [46] J. Montesano and M. Litterick, “UVM sequence item based error injection,” *SNUG Ottawa*, 2012.
- [47] K. Schwartz and T. Corcoran, “Error injection: When good input goes bad,” in *Proc. Design Verification Conf.(DVCon US 2017)*, 2017.
- [48] A. Kurakin, I. J. Goodfellow, and S. Bengio, “Adversarial examples in the physical world,” in *Artificial intelligence safety and security*, Chapman and Hall/CRC, 2018, pp. 99–112.
- [49] I. J. Goodfellow, J. Shlens, and C. Szegedy, “Explaining and harnessing adversarial examples,” *arXiv preprint arXiv:1412.6572*, 2014.
- [50] R. Aboudeif, T. A. Awaad, M. AbdelElSalam, and Y. Ismail, “Trustworthiness Evaluation of Deep Learning Accelerators Using UVM-based Verification with Error Injection,” in *DVCON Europe*, 2024 (accepted).
- [51] *Hardware Architectural Specification - NVDLA Documentation*, [nvdla.org. http://nvdla.org/hw/v1/hwarch.html](http://nvdla.org/hw/v1/hwarch.html) (accessed Aug. 16, 2024).
- [52] *In-memory data formats - NVDLA Documentation*, [nvdla.org. http://nvdla.org/hw/format.html](http://nvdla.org/hw/format.html) (accessed Aug. 16, 2024).
- [53] *Questa advanced verification, siemens digital industries software*. <https://eda.sw.siemens.com/en-us/ic/questa/> (accessed aug. 16, 2024).
- [54] *NVDLA Open Source Hardware, version 1.0, GitHub*, Jan. 25, 2023. <https://github.com/nvdla/hw> (accessed Aug. 16, 2024).
- [55] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.
- [56] IEEE, “IEEE Standard for Floating-Point Arithmetic, IEEE Std 754-2019 (Revision of IEEE 754-2008),” Institute of Electrical and Electronics Engineers New York, 2019.
- [57] *software manual - NVDLA documentation*, [nvdla.org. http://nvdla.org/sw/contents.html](http://nvdla.org/sw/contents.html) (accessed Aug. 16, 2024).