FPGA IMPLEMENTATION OF DEEP NEURAL NETWORKS (DNNS) FOR FAST INFERENCE AND ERROR RESILIENCE ANALYSIS

by
Uğur Berk ÇELİK

Submitted to
the Faculty of Engineering and Natural Sciences
in partial fulfillment of the requirements for the degree of
Master of Science

Sabancı Üniversitesi
İstanbul, Türkiye
July 2024

© 2024 by Uğur Berk ÇELİK.
All Rights Reserved.

ABSTRACT

FPGA IMPLEMENTATION OF DEEP NEURAL NETWORKS (DNNS) FOR FAST INFERENCE AND ERROR RESILIENCE ANALYSIS

UĞUR BERK ÇELİK

ELECTRONICS ENGINEERING MSC. THESIS, JULY 2024

Thesis Advisor: Asst.Prof. Ömer Ceylan

Keywords: Deep Learning, Convolutional Neural Networks, Error Injection

Deep learning, a subset of machine learning, has revolutionized numerous fields by its ability to model complex patterns and make highly accurate predictions. As Deep Neural Networks (DNNs) are increasingly deployed in critical applications, ensuring their reliability and robustness becomes paramount. Error resilience refers to a network's ability to maintain acceptable performance despite encountering faults or errors. These errors can stem from various sources, including hardware defects, environmental conditions, and operational stresses. Deploying DNNs on edge devices, such as mobile phones, IoT (Internet of Things), and embedded systems, presents unique challenges due to limited computational power, memory, and energy availability. Exploiting the error resilience of DNNs can significantly enhance energy efficiency for edge devices.

This thesis introduces an error injection framework aimed at evaluating the resilience of convolutional neural networks (CNNs) to bit-level faults. This framework employs a 2D error matrix format to achieve error injection at both the per-bit and per-layer levels. The dimensions of the matrix align with the number of error injection layers (rows) and the quantization bit width (columns). Each element in the matrix represents the bit error rate for a specific layer and bit position, allowing for precise and detailed error simulation.

Real-time error injection simulations, where faults are introduced dynamically during the network's operation, add another layer of computational demand. These simulations must accurately emulate real-time conditions and network responses, often requiring high-frequency processing and low-latency computation.

To efficiently implement this advanced framework, it was deployed on the AMD Xilinx Ultrascale+ SoC using the Vitis AI platform. The framework was further enhanced by

designing a specialized Error Injection IP in Verilog HDL. This IP facilitates the injection of errors into the outputs of convolutional and fully connected layers. The Error Injection IP performs error injection using an XOR gate based on the generated bit errors.
The Error Injection IP includes a Linear Feedback Shift Register (LFSR) to generate random errors, and the filter module is designed to ensure efficient and controlled fault injection.

ÖZET

DERİN SİNİR AĞLARININ (DSA) HIZLI ÇIKARIM VE HATA DAYANIKLILIK ANALİZİ İÇİN FPGA UYGULAMASI

UĞUR BERK ÇELİK

ELEKTRONİK MÜHENDİSLİĞİ YÜKSEK LİSANS TEZİ, TEMMUZ 2024

Tez Danışmanı: Dr.Öğr.Üyesi Ömer Ceylan

Anahtar Kelimeler: A,B,C,D Keywords: Derin Öğrenme, Evrişimsel Sinir Ağı, Hata Ekleme

Derin öğrenme, makine öğrenmesinin bir alt kümesi olarak, karmaşık kalıpları modelleme ve yüksek doğrulukta tahminler yapma yeteneği ile birçok alanda devrim yaratmıştır. DNN'ler giderek kritik uygulamalarda kullanıldıkça, güvenilirlik ve sağlamlıklarını sağlamak birinci derecede önem kazanmaktadır. Hata dayanıklılığı, bir ağın hatalarla veya arızalarla karşılaşmasına rağmen kabul edilebilir performansı sürdürme yeteneğini ifade eder. Bu hatalar, donanım arızaları, çevresel koşullar ve operasyonel stresler dahil olmak üzere çeşitli kaynaklardan kaynaklanabilir. DNN'leri mobil telefonlar, nesnelerin interneti ve gömülü sistemler gibi uçta hesaplama yapan cihazlara dağıtmak, sınırlı hesaplama gücü, bellek ve enerji mevcudiyeti nedeniyle zorluklar sunar. DNN'lerin hata dayanıklılığından yararlanmak, kaynak kısıtlı cihazlarda enerji verimliliğini önemli ölçüde artırabilir.

Bu tez, evrişimsel sinir ağlarının (CNN'ler) bit düzeyindeki hatalara karşı dayanıklılığını değerlendirmeyi amaçlayan bir hata enjeksiyon platformu sunmaktadır. Bu platform, hem bit başına hem de katman başına hata enjeksiyonu sağlamak için 2D hata matrisi formatını kullanır. Matrisin boyutlarını, hata enjeksiyon katmanlarının sayısı (satırlar) ve kuantizasyon bit genişliği (sütunlar) belirler. Matristeki her bir eleman, belirli bir katman ve bit pozisyonu için bit hata oranını temsil eder ve böylece kesin ve ayrıntılı hata simülasyonu sağlar.

Ağın çalışması sırasında dinamik olarak hataların tanıtıldığı gerçek zamanlı hata enjeksiyon simülasyonları, başka bir hesaplama talebi katmanı ekler. Bu simülasyonlar, gerçek zamanlı koşulları ve ağ tepkilerini doğru bir şekilde taklit etmeli ve genellikle yüksek frekanslı işleme ve düşük gecikmeli hesaplama gerektirir.

Bu, verimli bir şekilde uygulamak için, AMD Xilinx Ultrascale+ SoC üzerinde Vitis AI platformu kullanılarak dağıtılmıştır. Çerçeve, Verilog HDL'de tasarlanan özel bir Hata Enjeksiyon IP'si ile daha da geliştirilmiştir. Bu IP, evrişimsel ve tam bağlantılı katmanların çıktısına hataların enjekte edilmesini kolaylaştırır. Hata Enjeksiyon IP'si, gelen aktivasyon verilerini üretilen bit hatalarına dayalı olarak bir XOR kapısı kullanarak hatalar enjekte etmek gibi çeşitli kritik görevleri yerine getirir.
Hata Enjeksiyon IP'sinde, rastgele hatalar üretmek için Doğrusal Geri Besleme Kaydırma Kaydedicisi (LFSR) kullanılır. LFSR çıktılarını kontrolünü sağlamak amacıyla filtre modülü tasarlanmıştır.

ACKNOWLEDGMENT

The successful completion of this thesis has been greatly influenced by the guidance, support, and encouragement of several key individuals, for which I am profoundly thankful.

First and foremost, I would like to express my deepest appreciation to my advisor, Asst. Prof. Ömer CEYLAN. His insightful guidance, patience, and support have been instrumental in shaping this research. I am profoundly grateful for reviewing my work and providing invaluable feedback. His mentorship has played a pivotal role in my academic development.

I would also like to extend my sincere thanks to Prof. Dr. Emre SALMAN, for his expertise and thoughtful advice throughout this process. His contributions have been crucial in refining my research and ensuring its success.

My gratitude also goes to the esteemed members of my jury, Assoc. Prof. Öznur TAŞTAN and Asst. Prof. Atilla UYGUR, for their careful review and constructive feedbacks.

I am also grateful to my lab mates for their constant support and collaboration, which have made this journey both valuable and rewarding.

Lastly, I would like to express my deepest gratitude to my family for their patience and encouragement. Their belief in my abilities has been a constant source of strength and motivation throughout this journey.

To everyone who has supported me along the way, I offer my heartfelt thanks. Your contributions have made this thesis possible.

Dedicated to
my parents Zinnet & Ali Osman, my sisters Elanur, Şevval, my aunt Zeynep and my beloved fiance Şeyma.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

1. INTRODUCTION

With recent advancements in the computing power of digital hardware circuits, such as general-purpose central processing units (CPUs), graphics processing units (GPUs), and field-programmable gate arrays (FPGAs), coupled with the increased availability of massive amounts of data, Deep Learning (DL) has emerged as a state-of-the-art method for solving complex tasks (Schmidhuber, 2014). Deep Learning algorithms, including neural networks with many hidden layers, known as Deep Neural Networks (DNNs), are designed to model and understand complex patterns in labeled data. These algorithms are particularly well-suited for tasks such as image classification (Krizhevsky et al., 2012), natural language processing (Radford et al., 2019), and autonomous systems (Muller et al. (2005)) achieving performance that surpasses human accuracy (Weyand et al., 2016).

The rapid integration of DNNs into various domains, including autonomous systems, healthcare, and finance, has significantly increased the demand for robust and reliable AI systems. However, despite their numerous advantages, DNNs are inherently susceptible to a range of errors that can arise from various sources, such as hardware faults (He et al., 2023), software bugs (Tian et al., 2017), and adversarial attacks (Wang et al., 2023). These vulnerabilities pose serious risks to the performance and reliability of AI systems, making the assessment and enhancement of DNN resilience a critical concern.

The vulnerabilities of DNNs are complex and varied. Hardware faults, which can occur due to manufacturing defects, radiation-induced soft errors, or wear and tear over time, can introduce unexpected behaviors in neural network operations. Additionally, deploying DNNs on edge devices introduces a unique set of challenges due to the inherent constraints of these platforms. Edge devices typically possess limited computational capabilities, restricted memory, and finite energy resources, which can significantly hinder the performance of DNNs. Consequently, optimizing DNN deployment on such devices requires innovative strategies to ensure efficiency and sustainability. Techniques like voltage scaling are often employed to manage power consumption, enabling these edge devices to run complex models without excessive energy depletion.

Neural networks possess a degree of fault tolerance that can handle certain errors without significant performance degradation (Li et al., 2017), (Chen et al., 2018). This property can be advantageous. It enables the design of low-power, error-aware deep learning hardware that can operate efficiently even under less-than-ideal conditions (Choi et al., 2019). However, these issues still necessitate comprehensive evaluation techniques to ensure the robustness and reliability of DNN-based systems.

Neural networks inherently exhibit a level of fault tolerance, allowing them to manage certain errors without substantial degradation in performance (Li et al., 2017), (Chen et al., 2018). This characteristic is particularly beneficial in the context of designing low-power, error-aware deep learning hardware, which can continue to function efficiently even under suboptimal conditions. The ability to maintain functionality despite these errors highlights the potential for developing more energy-efficient and resilient systems (Choi et al., 2019). However, despite this inherent fault tolerance, it remains crucial to employ evaluation methodologies to assess and guarantee the robustness and reliability of systems based on DNNs. Such evaluation is essential to ensure that these systems can consistently perform as expected in real-world scenarios, where unforeseen faults and errors are likely to occur.

Error injection has emerged as a vital technique for assessing the resilience of DNNs (Reagen et al., 2018), (Li et al., 2017) and cryptographic attacks (Barenghi et al., 2012), (Blömer and Seifert, 2003) and (Bae et al., 2011). This method involves deliberately introducing faults into the system to observe how it responds, thereby identifying potential weaknesses and strategies to enhance fault tolerance. Error injection allows researchers to systematically evaluate the impact of various types of faults on DNN performance and cryptographic algorithms, providing insights into the network's robustness under adverse error rates.

Traditional error injection methods, while effective, are often constrained by significant limitations. They are typically time-consuming and computationally intensive, making them impractical for large-scale DNNs and real-time applications. According to (Rathore, 2021), the fault resilience evaluation of EfficienNet-B4 (Tan and Le, 2019) CNN model requires 11.6 hours to 4.5 days, depending on dataset usage. Timing results are observed with a subset of Imagenet dataset (Russakovsky et al., 2015).

To address these challenges, this thesis focuses on the implementation of a fast error injection platform for DNN applications. This platform aims to provide a scalable and efficient solution for resilience evaluation of DNNs, by leveraging advancements in hardware acceleration techniques with FPGA. The proposed platform is designed to support fault injection on datapaths, enabling a comprehensive assessment of DNN resilience while significantly reducing the time required for the evaluation process. This work focuses on datapath error injection instead of memory faults (Reagen et al., 2018) .Unlike PyTorchFI (Mahmoud et al., 2020) and (Narayanan et al., 2023), the proposed error injection platform supports both Tensorflow and PyTorch deep learning frameworks since Vitis AI supports deployment on AMD Xilinx DPU IP for both.

In this thesis, the proposed error injection platform is implemented on the AMD Xilinx Ultrascale+ MPSoC device, a highly versatile and powerful platform for embedded systems. The implementation leverages the advanced capabilities of the Vitis AI (AMD, 4 25) development environment, which facilitates the efficient deployment of compute-intensive layers, particularly convolutional and fully connected layers, within deep neural networks. These layers are crucial for the performance of neural networks and are realized using the AMD Xilinx Deep Learning Processing Unit (DPU) IP core. The DPU is synthesized within the FPGA fabric of the MPSoC, enabling accelerated processing and enhanced parallelism. This integration not only optimizes the computational throughput but also ensures that the platform can handle the demands of high-throughput inference while maintaining the flexibility required for error injection and resilience testing.

To implement the error injection task efficiently, an Error Injection Intellectual Property (IP) core is designed with FPGA resources. This custom IP core is designed to manipulate activation data output from convolutional and fully connected layers within neural networks. By injecting errors at predetermined rates, the IP core enables the simulation of fault conditions that may occur in real-world scenarios. The error injection process is realized through the introduction of controlled bit flips within the activation data, thereby allowing a systematic analysis of network robustness.

The accuracy performance of the AMD Xilinx DPU is evaluated under different CNN models without error injection, including ResNet-18, ResNet-50 (He et al., 2015), MobileNetV2 (Sandler et al., 2018b) and InceptionV3 (Szegedy et al., 2015). These CNN models are evaluated using the Imagenet dataset (Russakovsky et al., 2015).

Furthermore, the proposed fault injection architecture is not limited to traditional neural network applications. It holds significant potential for application within the domain of cryptographic fault injection, where the ability to induce and analyze faults can play a crucial role in assessing the security and robustness of cryptographic algorithms against fault-based attacks.

Error injection performance of the designed platform is measured and compared with CPU performance.

1.1 Motivation

Artificial intelligence (AI) is significantly expanding its range of applications and is becoming increasingly significant across various sectors. It is being extensively utilized in industrial domains such as healthcare, autonomous vehicles, call centers, and numerous other areas.

Edge devices encompass a broad spectrum of hardware, ranging from smartphones to Internet of Things (IoT) sensors and embedded systems. By processing data locally, edge devices can make real-time decisions essential for latency-sensitive applications like autonomous vehicles. Edge computing reduces the necessity of transmitting large data volumes to the cloud for processing. This not only decreases operational costs but also mitigates network congestion, thereby improving system efficiency. Moreover, local data processing on edge devices ensures that sensitive information remains on the device which enhance data security. Additionally, edge devices can operate independent of network connectivity, making them more reliable in environments with unstable or limited internet access.

However, edge devices typically have limited processing power and memory compared to cloud servers. Deep learning models, such as Convolutional Neural Networks (CNNs), require substantial computational resources and often consist of millions of parameters, leading to significant memory demands.Techniques such as model quantization and pruning are actively being explored to address these challenges.

Battery life remains a paramount concern for edge devices, especially for mobile phones and Internet of Things (IoT) sensors, where the continuous operation of deep learning algorithms can lead to rapid battery depletion. The energy demands of these algorithms pose significant challenges, particularly in scenarios where battery life is essential for maintaining the functionality of these devices. Among the various strategies employed, voltage scaling has emerged as an effective technique for reducing power consumption during the deployment of deep neural networks on edge devices. By adjusting the voltage levels in accordance with the computational load, this approach not only conserves energy but also prolongs the operational lifespan of the device, making it a critical component in the development of sustainable and efficient edge computing solutions.

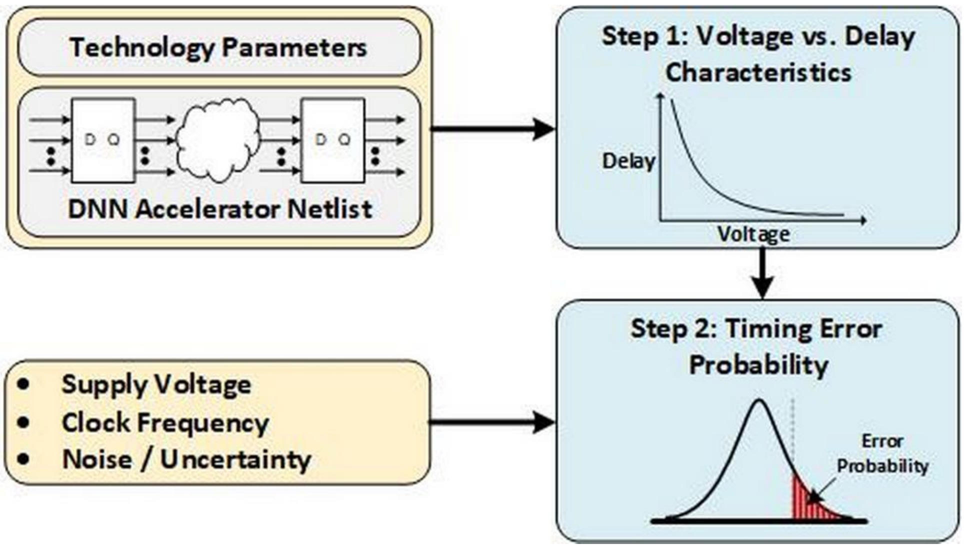


Figure 1.1 Timing error probability model overview

In the context of this research, error probability models have been developed to facilitate voltage scaling across layers of deep neural networks as shown in Figure 1.1. Unlike traditional gate-level simulations, which are computationally expensive and time-consuming, the error probability of a given netlist can be efficiently calculated using these models. The delay characteristics are derived based on variations in voltage, temperature, and process parameters. The probability density function (PDF) of the nominal values of these derived characteristics is obtained, and the error probability is subsequently calculated based on this PDF.

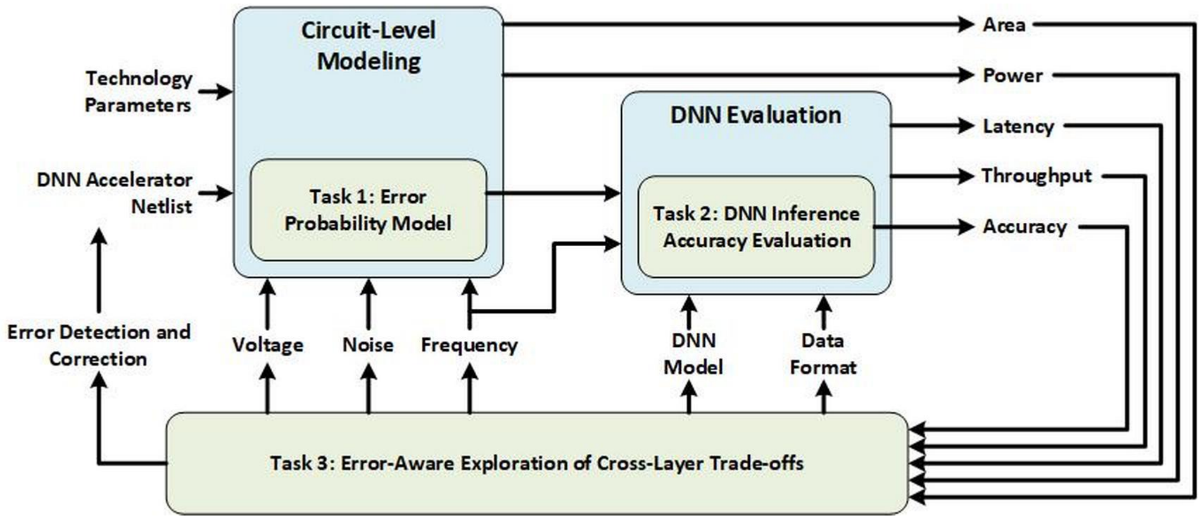

Circuit-Level Modeling
Technology Parameters
DNN Accelerator Netlist
Task 1: Error Probability Model
DNN Evaluation
Task 2: DNN Inference Accuracy Evaluation
Area
Power
Latency
Throughput
Accuracy
Error Detection and Correction
Voltage
Noise
Frequency
DNN Model
Data Format
Task 3: Error-Aware Exploration of Cross-Layer Trade-offs

Figure 1.2 Summary of the PVT aware Probabilistic Timing Error Model

Figure 1.2 illustrates the overall process for evaluating Deep Neural Networks (DNNs). Circuit-level modeling generates error probabilities according to the netlist while accounting for variations in Process, Voltage, and Temperature (PVT). In this thesis, we evaluate the accuracy of a given CNN model by injecting errors derived from the error probability model. These errors provide valuable insights into the model's robustness and the potential impact of hardware-induced inaccuracies.

2. BACKGROUND

This segment introduces the terminology employed throughout the dissertation while explaining the concepts of Computatonal Architectures, FPGA and SoC architecture, Deep Neural Networks, Vitis AI, Quantization and Error Injection.

2.1 Computational Architectures

In the modern computing area, several key architectures—namely Central Processing Units (CPUs), Graphics Processing Units (GPUs), Field-Programmable Gate Arrays (FPGAs), Application-Specific Integrated Circuits (ASICs), and System on Chip (SoC)—are employed to meet various computational needs. Each architecture offers unique advantages in terms of performance, flexibility, energy efficiency, and scalability, making them suitable for different types of applications.

2.1.1 Central Processing Units (CPUs)

The Central Processing Unit (CPU) is the principal component of a computer that performs most of the processing inside the system. Often referred to as the ”brain” of the computer, the CPU executes instructions from programs by performing basic arithmetic, logic, control, and input/output (I/O) operations. Modern CPUs are characterized by a number of key features, including multiple cores, high clock speeds, and advanced instruction sets. Multi-core CPUs, in particular, have become standard in contemporary computing, allowing for parallel processing and the execution of multiple tasks simultaneously.

2.1.2 Graphics Processing Units (GPUs)

Graphics Processing Units (GPUs) have become pivotal in modern computing, evolving far beyond their initial role of rendering images and video in gaming and entertainment applications. Originally designed to accelerate the creation of images intended for output to a display, GPUs are now extensively used for a wide range of general-purpose computing tasks, particularly those involving parallel processing. GPUs are designed for parallel processing, enabling them to handle multiple tasks simultaneously. This capability is rooted in their architecture, which comprises thousands of smaller, more efficient cores engineered for handling multiple tasks concurrently. This parallelism makes GPUs especially suitable for computationally intensive tasks such as deep learning, scientific simulations, and data analytics.

2.1.3 System on Chips (SoCs)

System on Chips (SoCs) with integrated FPGA combine the flexibility of FPGAs with the convenience of a complete computing system on a single chip. An SoC typically includes a CPU, memory, and peripherals alongside the FPGA fabric. This integration allows for a versatile platform where the CPU handles general-purpose tasks while the FPGA accelerates specific functions, providing a balance between flexibility, performance, and ease of use.

2.1.4 Application-Specific Integrated Circuits (ASICs)

Application-Specific Integrated Circuits (ASICs) are custom-designed semiconductor devices tailored for specific tasks, providing a high level of performance and efficiency. ASICs are optimized for particular functions, making them a crucial component in various high-performance and specialized systems. ASICs have become a cornerstone in industries where speed, efficiency, and reliability are paramount. Their design process involves creating a circuit that performs a dedicated function, which can significantly reduce power consumption and increase processing speed. As a result, ASICs can deliver superior performance in specific applications, such as telecommunications, automotive systems, and consumer electronics.

2.1.5 Field-Programmable Gate Arrays (FPGAs)

FPGAs provide a balance between performance and flexibility by allowing users to reconfigure the hardware to match specific tasks. This adaptability makes FPGAs ideal for use in areas such as signal processing, cryptography, and machine learning. FPGAs are often chosen for applications where the benefits of hardware customization outweigh the advantages of fixed-function hardware.

2.1.6 Comparison

CPUs, GPUs, FPGAs, and ASICs each offer distinct advantages in modern computing. CPUs are highly flexible and broadly applicable, GPUs excel at parallel processing tasks, FPGAs provide customizable hardware solutions, and ASICs deliver unmatched efficiency for specific tasks. The selection of an appropriate architecture depends on the specific requirements of the application, such as the need for flexibility, computational power, and energy efficiency.

CPUs are most effective in applications requiring strong single-thread performance and are ubiquitous in devices ranging from personal computers to large-scale servers. However, CPUs may not be as optimized for highly parallel or specialized tasks compared to other architectures.

In contrast to CPUs, GPUs are composed of a large number of smaller cores that work together to process data concurrently, making them highly efficient for parallel tasks. However, GPUs are less effective for tasks that are not easily parallelizable and still demand strong single-thread performance.

ASICs, unlike FPGAs, are hardwired during manufacturing, which makes them extremely power-efficient and fast for their designated functions. However, this high degree of specialization comes at the cost of flexibility; once produced, the functionality of an ASIC cannot be altered.

FPGAs, unlike CPUs, ASICs, and GPUs, which have fixed architectures, can be programmed to optimize their hardware layout for specific applications, making them highly efficient for tasks requiring customized hardware acceleration. However, the reprogrammable nature of FPGAs demands specialized expertise and tools, and they generally consume more power compared to ASICs.

The main advantage of an SoC with FPGA lies in its ability to perform heterogeneous computing, where different types of processors work together to optimize performance and power efficiency for complex applications.

Each of the computational architectures discussed above has distinct advantages and disadvantages, with well-defined areas where each excels. Given that this thesis aims to implement high-throughput fault resilience analysis of Deep Neural Networks, the AMD Xilinx SoC computation architecture has been selected.

2.2 FPGA and SoC ARCHITECTURE

Field-Programmable Gate Arrays (FPGAs) have emerged as crucial components in modern computing systems, offering flexible and efficient solutions for multiple applications. FPGA programming plays a vital role in unleashing the full potential of these devices, enabling customized hardware accelerators, real-time signal processing, and high-performance computing capabilities with parallel processing features. FPGA programming involves designing and implementing custom hardware circuits on FPGA devices. Unlike traditional processors, FPGAs allow users to configure hardware functionality at a register transfer level (RTL), providing paralleled flexibility and performance optimization opportunities. HDLs (Hardware Description Language) such as Verilog and VHDL are commonly used for FPGA programming. These languages enable developers to describe the behavior and structure of digital circuits, including logic gates, registers, and complex datapaths.

Additionaly , the advent of High-Level Synthesis (HLS) has revolutionized FPGA programming by enabling developers to design hardware at a higher level of abstraction, significantly simplifying the development process.

High-Level Synthesis (HLS) refers to the process of converting high-level programming languages, such as C, C++, or OpenCL, into hardware description code that can be deployed on an FPGA. This approach allows developers to leverage familiar software programming paradigms while still targeting the parallelism and performance benefits of FPGA hardware. HLS tools automate the generation of HDL code from high-level language descriptions, enabling a broader range of engineers and developers to engage in FPGA design without the steep learning curve associated with traditional HDL programming.

FPGAs consist of an array of configurable logic blocks (CLBs) as shown in Figure 2.1, that can be programmed to perform various logic functions. Each CLB typically contains lookup tables (LUTs) for implementing combinational logic such as AND,OR and NAND gate, flip-flops for sequential logic, and multiplexers for routing signals. Interconnects are the pathways that connect logic blocks, input/output pins, and other components within the FPGA. FPGAs utilize programmable routing resources such as switch matrices and routing tracks to establish connections based on the design requirements. In modern FPGAs, DSP (Digital Signal Processor) blocks provide high-speed processing capabilities for arithmetic operations such as multiplication and addition, Block RAM with low power consumption, high-speed capability, and transievers for high-speed data communication like PCIe. In Figure 2.1, an example of the FPGA structure with components are shown. FPGAs inherently support parallel processing due to their parallel architecture with multiple logic blocks operating simultaneously. This parallelism leads to high-throughput performance, particularly in tasks that benefit from parallel execution, such as signal processing and machine learning.

System on Chip (SoC) with FPGA, such as the AMD Xilinx Ultrascale+ MPSoC, combines the flexibility of FPGAs with the processing power of traditional processors, creating a highly versatile and powerful platform, as shown in Figure 2.2. The AMD Xilinx Ultrascale+ MPSoC integrates several components, such as ARM Processors, and Programmable Logic (PL) to enhance its functionality. The Ultrascale+ MPSoC includes quad-core ARM Cortex-A53 processors for high-performance applications and dual-core

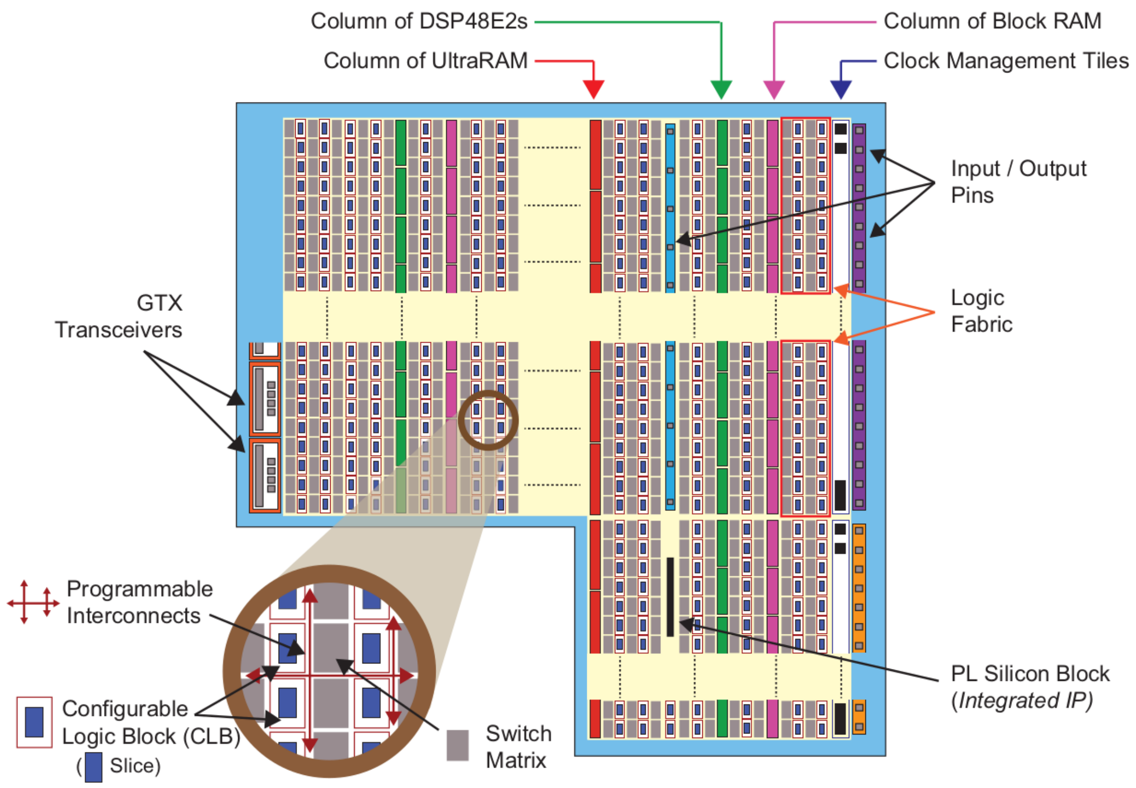


Figure 2.1 FPGA Example with components (Louie, 2021) .

ARM Cortex-R5 real-time processors for deterministic real-time tasks. This combination allows the SoC to handle a wide range of computing tasks efficiently.

Integrating Field Programmable Gate Arrays (FPGAs) with Central Processing Units (CPUs) is becoming increasingly common in modern computing systems. A key component facilitating this integration is the Advanced eXtensible Interface (AXI), specifically the AXI-4 protocol. As part of the ARM Advanced Microcontroller Bus Architecture (AMBA) (ARM Ltd., 2022), AXI-4 interconnects between CPUs and FPGAs as shown in Figure 2.2, enabling efficient communication and data transfer for AMD Xilinx SoC devices. The AXI-4 protocol is designed to support the demands of high-performance designs. It consisted of separate address/control and data channels, making data handling flexible and efficient. A defining characteristic of AXI-4 is its burst-based transactions, which facilitate the transfer of large data blocks from CPU memory with minimal overhead. This capability is particularly advantageous for applications requiring significant data movement between CPUs and FPGAs, such as digital signal processing and machine learning.

The AXI-4 interface has five independent channels: read address, read data, write address,

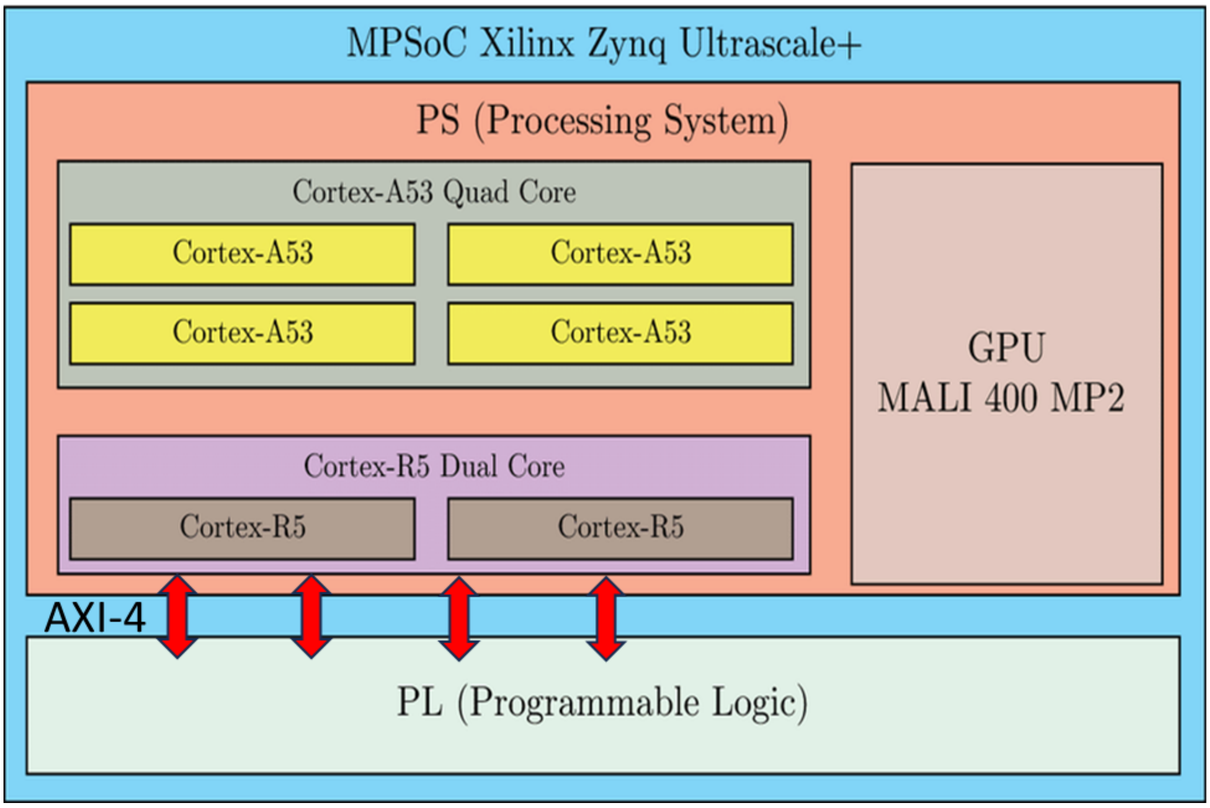


Figure 2.2 System-on-Chip Overview.

write data, and write response. The independent operation of these channels permits concurrent read and write operations. Those channels are:

- *Read Address Channel*: Carries the address of the data to be read from memory, including signals for address, burst type, burst length, and control information.
- *Read Data Channel*: Transports the data read from memory along with signals indicating data validity and any errors during the read operation.
- *Write Address Channel*: Conveys the address of the data to be written to memory, similar to the read address channel, with signals for address, burst type, and burst length.
- *Write Data Channel*: Transfers the data to be written to memory, along with control signals indicating data validity.
- *Write Response Channel*: Provides feedback on write operations, including signals indicating completion status and any errors encountered.

The AXI-4 protocol comprise a handshake mechanism to controls data flow between the CPU and FPGA. Each channel uses valid and ready signals to manage data transfers.

The master asserts the valid signal when data is available, and the slave asserts the ready signal when it is prepared to receive the data. This handshake mechanism ensures that data is transferred only when both parts are ready, preventing data loss and ensuring reliable communication. AXI-4 Ports of AMD Xilinx Ultrascale+ MPSoC are shown in Figure 2.2.

Figure 2.3 shows a write transfer sequence realized with AXI-4 memory mapped interface.

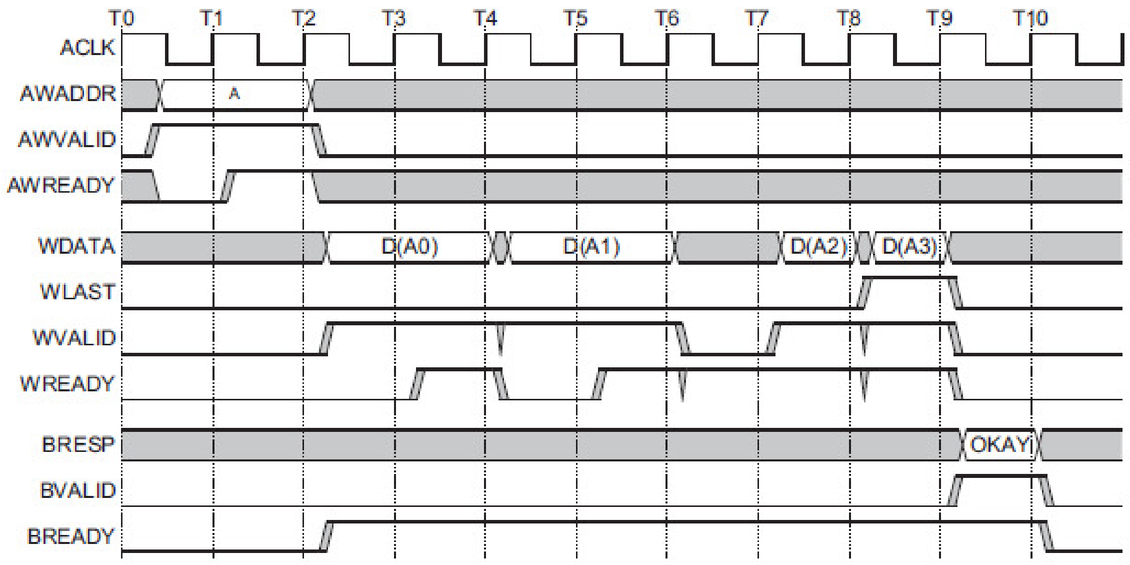


Figure 2.3 AXI-4 Write Transfer Timing Diagram (ARM Ltd., 2022).

2.3 Deep Neural Networks (DNNs)

Neural network is a computational paradigm inspired by the human brain. These networks detect patterns and make data-driven decisions through a process akin to human learning. The fundamental architecture of a deep neural network (DNNs) comprises multiple layers: an input layer, one or more hidden layers, and an output layer as shown in Figure 2.4. The hidden layers, where the actual computations are performed, are key to the network's ability to learn and generalize from the input data.
A neural network operates in two main phases: training and inference. During the training phase, the network learns from a large dataset specifically collected for the task. This phase involves adjusting the model parameters to minimize errors and improve accuracy.

The inference phase, on the other hand, invokes the trained model to make predictions or decisions based on new, unseen data.

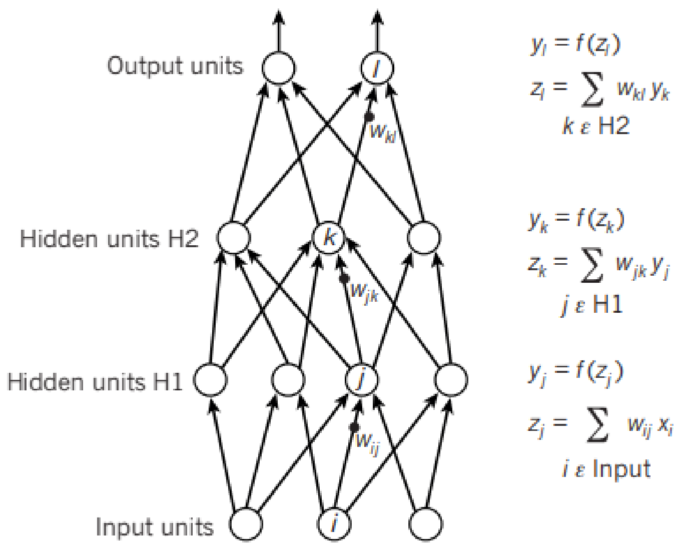


Figure 2.4 Basic Neural Network Architecture (LeCun et al., 2015).

Deep Neural Networks, based on neural networks with many hidden layers, have become synonymous with deep learning. These networks can include tens of hidden layers, enabling them to model complex patterns and representations in data. The depth of these networks allows them to learn more complex features.

One notable type of DNN is the Convolutional Neural Network (CNN), which is particularly effective for computer vision tasks. Unlike traditional computer vision methods, CNNs automate feature extraction, making them highly efficient for image recognition. This is achieved through convolutional layers that apply filters to the input data, extracting relevant features without requiring manual intervention. Although the concept of CNNs dates back to the 1980s, (LeCun et al., 1989), their substantial advancement and widespread adoption have occurred in recent years, notably catalyzed by seminal works like AlexNet (Krizhevsky et al., 2012), a pioneering deep neural network that significantly influenced the field of deep learning. CNNs operate through a hierarchical approach to image processing, employing specialized layers called convolution,pooling, and activation functions.

Convolutional layers extracts features from input images. These layers employ filters to capture patterns such as edges, textures, and shapes. Pooling layers are utilized to decrease the spatial dimensions (width and height) of the feature maps while preserving essential information such as max pooling and average pooling. This reduction aids in lowering the computational complexity of the network by decreasing the number of pa-

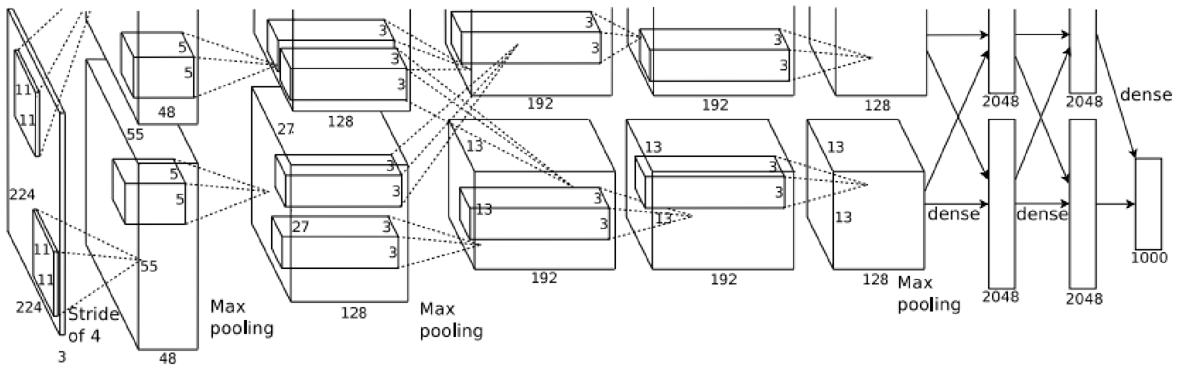


Figure 2.5 AlexNet CNN Architecture (Krizhevsky et al., 2012) with 5 Convolution, 3 Max Pooling and 2 Fully Connected layers.

rameters and computations required in subsequent layers. Activation functions introduce non-linearities into the network, enabling it to learn and represent complex relationships within the data. Without non-linear activation functions, the network would be restricted to learning linear transformations, thus impeding its ability to capture intricate patterns and features present in the data.

At the end of CNN models, fully connected layers are employed for classification, which involves assigning input data to specific categories or classes. The output layer of the fully connected network typically utilizes softmax activation to produce probabilities corresponding to each class.

As an example, the AlexNet (Krizhevsky et al., 2012) CNN architecture consists of eight layers, five of which are convolutional layers followed by three fully connected layers as shown in Figure 2.5. The first layer applies a convolution operation to the input image using a set of filters, capturing low-level features such as edges and textures. Subsequent convolutional layers progressively learn more complex features, with each layer's filters capturing increasingly abstract patterns in the data. A notable feature of AlexNet is the use of Rectified Linear Units (ReLU) as activation functions. ReLUs introduce non-linearity to the model, enabling it to learn complex patterns more effectively than traditional activation functions like sigmoid or tanh. Additionally, AlexNet employs max-pooling layers after some of the convolutional layers, which downsample the spatial dimensions of the feature maps, reducing the computational load and providing translational invariance to the network. Another key aspect of AlexNet is its use of dropout (Srivastava et al., 2014) in the fully connected layers. Dropout is a regularization technique that prevents overfitting by randomly "dropping out" a fraction of the neurons during training, forcing the network to learn more robust features.

The implementation of DNNs, particularly CNNs, has been significantly enhanced by the use of Field Programmable Gate Arrays (FPGAs) and Graphics Processing Units (GPUs). These devices are well-suited for the parallel processing demands of CNNs, enabling faster and more efficient computations. The parallel structure of FPGAs and GPUs allows for the simultaneous processing of multiple data streams, which is essential for handling the large volumes of data typically associated with deep learning tasks.
In summary, neural networks represent a powerful and flexible approach to pattern recognition and decision-making. The advancement of DNNs, particularly through architectures like CNNs and the utilization of FPGAs and GPUs, has opened new avenues for research and application in various fields, including computer vision, natural language processing, and beyond. The ability of these networks to learn from data, adapt to new information, and perform complex tasks underscores their importance in the ongoing development of artificial intelligence technologies.

2.4 Deep Neural Networks with FPGA

Deep Neural Networks have revolutionized in various fields, including computer vision (Krizhevsky et al., 2012), natural language processing (Radford et al., 2019), and autonomous systems Muller et al. (2005), due to their ability to learn and model complex patterns from vast amounts of data. As DNNs, particularly Convolutional Neural Networks (CNNs), become more sophisticated and resource-intensive (Krizhevsky et al., 2012), there is a growing need for hardware accelerators that can meet the performance and power efficiency requirements of these applications. Field-Programmable Gate Arrays (FPGAs) have emerged as a promising platform for DNN acceleration due to their reconfigurable architecture, which allows for customized and optimized implementations of Deep Learning (DL) algorithms.
The flexibility of FPGAs extends to their ability to implement custom dataflow architectures tailored to the specific requirements of DNN models. By designing custom hardware pipelines for convolutional, pooling, and activation layers, FPGAs can achieve higher performance and lower latency. Furthermore, the use of high-level synthesis (HLS) tools has simplified the development process for FPGA-based accelerators, allowing designers to describe hardware functionality using high-level programming languages such as C or C++ (Liang et al., 2012). This has significantly reduced the development time and effort

required to implement DL algorithms on FPGAs.
Most FPGA platforms tailored for CNN tasks consist of two main components: the processing system (PS) and the programmable logic (PL) (Qiu et al., 2016).
The PS encompasses the CPU and DDR memory, playing a pivotal role in orchestrating the overall operation of the system. The primary function of the PS is to facilitate data flow and manage the Programmable Logic (PL), ensuring that computationally intensive tasks associated with Convolutional Neural Networks (CNNs) are executed efficiently within PL part. To achieve this, the PS is responsible for preparing the necessary instructions and storing CNN model parameters within the DDR memory. Furthermore, the PS undertakes the critical tasks of input image pre-processing and output post-processing, thereby preparing the data for the CNN model and subsequently managing the results generated by the model. This division of responsibilities allows the PS to ensure seamless integration between the CPU and the PL, optimizing the performance of CNN tasks.
In PL, Block RAMs (BRAMs) play a crucial role in buffering both input and output data. This strategic use of BRAMs is essential for minimizing the overall memory footprint, particularly given that Dynamic Random Access Memory (DRAM) often emerges as a critical bottleneck in high-performance computing systems. To further enhance computational efficiency, Digital Signal Processors (DSPs) are leveraged within the PL to execute operations that are computationally intensive and require high throughput. These operations primarily include convolutions and fully connected layers, which are dominated by multiply-accumulate (MAC) operations. To carry out these complex calculations, the PL is responsible for fetching the necessary instructions, input data, and parameters of the CNN from memory. Once these calculations are completed, the resulting output data is written back to the CPU memory, ensuring seamless integration between the processing stages and efficient utilization of the system's resources.

Another device used in Deep Learning tasks is called the Graphical Processing Unit (GPU). A typical GPU consists of multiple streaming multiprocessors (SMs), each housing numerous smaller processing units known as CUDA cores (in NVIDIA GPUs) or stream processors (in AMD GPUs) as shown in Figure 2.6. These cores are designed to execute thousands of threads concurrently, making GPUs highly efficient for parallel processing tasks. The SMs also include special function units (SFUs) for executing complex mathematical operations and texture units for handling texture mapping and filtering tasks.

GPUs, originally designed for rendering graphics in video games, are highly parallel processors capable of performing thousands of simultaneous operations. This parallelism makes GPUs particularly well-suited for the workloads associated with deep learning,

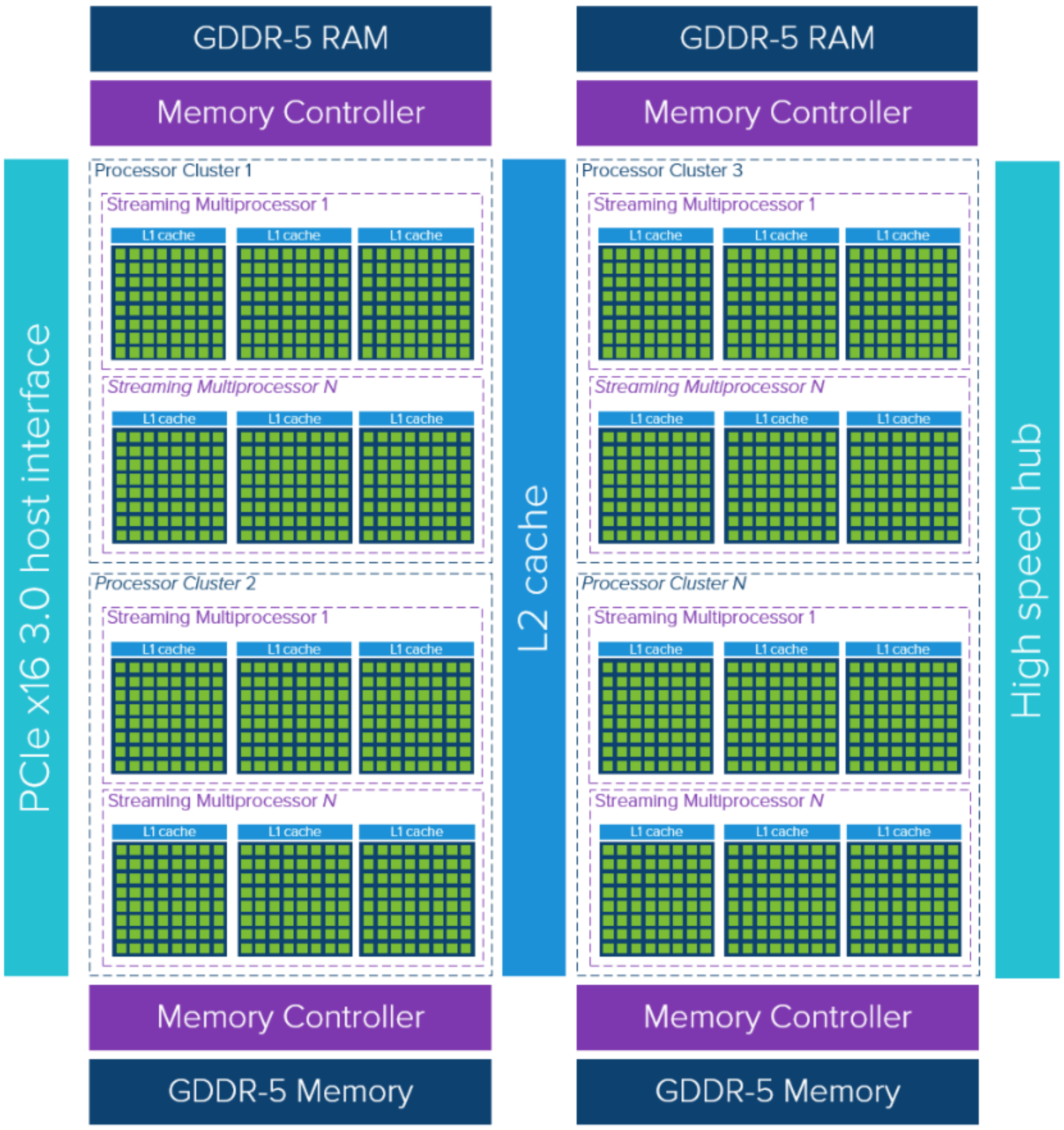


Figure 2.6 GPU Architecture (VMware, 2023).

where operations on large matrices and vectors can be distributed across many cores. As a result, GPUs can significantly accelerate the training and inference processes of deep neural networks compared to CPUs.

One of the primary advantages of GPUs in deep learning is their ability to handle high data throughput. Deep learning models, particularly CNNs and recurrent neural networks (RNNs), require substantial data to train effectively (Yazdanbakhsh et al., 2015). GPUs facilitate this by leveraging their memory bandwidth and parallel processing capabilities to manage and process large datasets efficiently. This capability reduces the time required to train models, enabling researchers and practitioners to iterate more quickly and improve model performance.

In most deep learning papers such as (Krizhevsky et al., 2012), GPU devices were used to train deeper DNN models .When considering the implementation of DNNs, a critical trade-off exists between using FPGAs and GPUs, primarily due to the significant parallel computing capabilities of GPUs. GPUs are renowned for their ability to handle extensive parallel processing tasks, which is particularly advantageous for the computationally intensive operations required by DNNs. However, this performance comes at a cost: GPUs are typically power-hungry devices. In contrast, FPGAs offer a more power-efficient alternative, making them a preferable choice in applications where power consumption is a critical factor. FPGAs can operate effectively at lower supply voltages, which is advantageous in environments where power availability is limited or where energy efficiency is paramount. Therefore, while GPUs might provide superior raw computational power, the energy efficiency of FPGAs presents a compelling case for their use in certain DNN applications, especially those with strict power constraints. The comparative analysis of performance and performance per watt for FPGAs with different clock speeds and Nvidia Titan X GPU is depicted in Figure 2.7. As illustrated in Figure 2.7(a), when the frequency estimate for the Stratix 10 is moderately increased to 500 MHz, its performance surpasses that of the GPU. In terms of energy efficiency, Figure 2.7(b) demonstrates that FPGAs consistently deliver superior performance per watt compared to GPUs. Specifically, the Arria 10 FPGA achieves a higher performance per watt ratio relative to the GPU, with the Stratix 10 exhibiting even greater improvements in this metric.

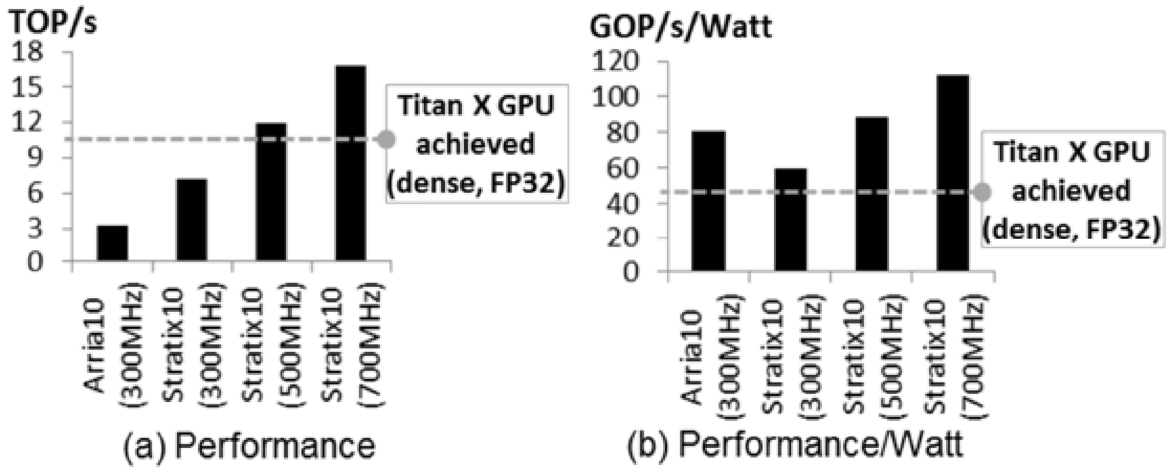
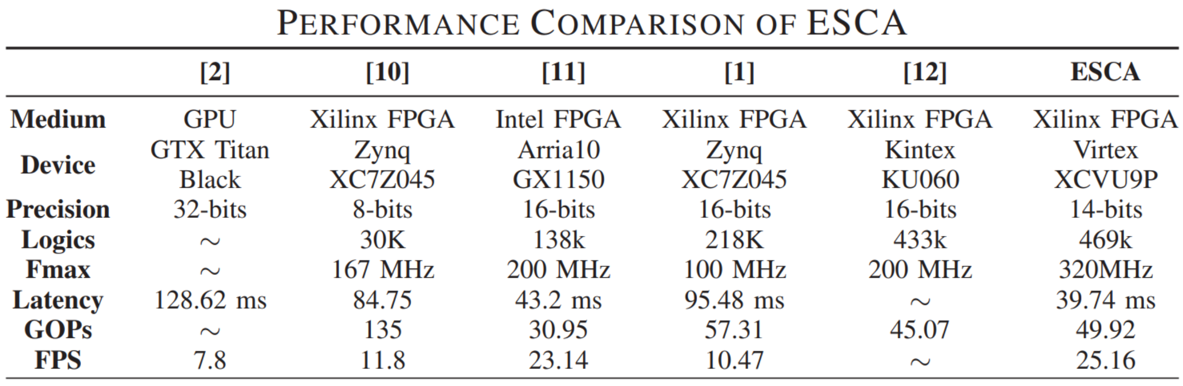


Figure 2.7 FPGA vs GPU Comparion (Guo et al., 2018).

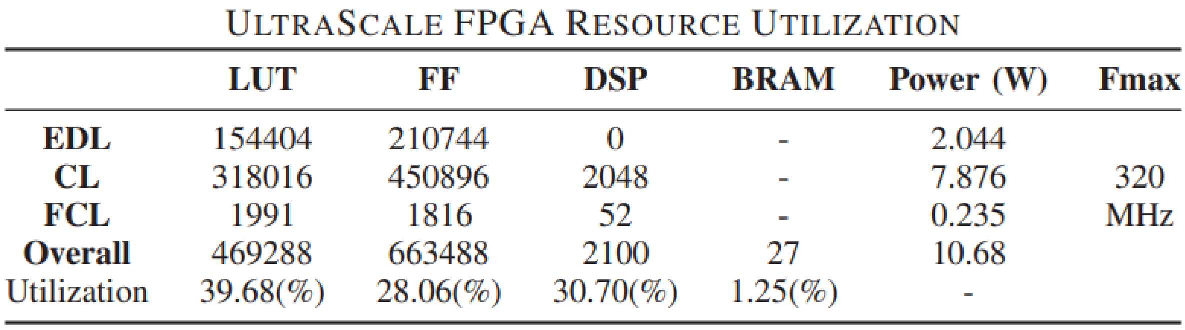
Moreover, the remarkable performance of GPUs stems from their capability to handle large batches of images concurrently. This parallel processing efficiency is highly advantageous in many scenarios. However, some applications, such as video streaming, necessitate a different approach. In these instances, each frame is processed individually due to the stringent latency requirements. The timely processing of each frame is crucial to maintaining the overall performance and responsiveness of the application. Thus, while batch processing is beneficial in general image processing tasks, frame-by-frame processing is essential for applications where immediate feedback and low latency are critical (Guo et al., 2018),(Shawahna et al., 2019). Research conducted by (Bhowmik et al., 2021) demonstrates that FPGAs exhibit superior performance compared to GPUs in time-critical computer vision applications, particularly in terms of frames per second (FPS) while requires 10.68 Watt power, as illustrated in Figure 2.8 and Figure 2.9.



PERFORMANCE COMPARISON OF ESCA

	[2]	[10]	[11]	[1]	[12]	ESCA
Medium	GPU	Xilinx FPGA	Intel FPGA	Xilinx FPGA	Xilinx FPGA	Xilinx FPGA
Device	GTX Titan Black	Zynq XC7Z045	Arria10 GX1150	Zynq XC7Z045	Kintex KU060	Virtex XCVU9P
Precision	32-bits	8-bits	16-bits	16-bits	16-bits	14-bits
Logics	~	30K	138k	218K	433k	469k
Fmax	~	167 MHz	200 MHz	100 MHz	200 MHz	320MHz
Latency	128.62 ms	84.75	43.2 ms	95.48 ms	~	39.74 ms
GOPs	~	135	30.95	57.31	45.07	49.92
FPS	7.8	11.8	23.14	10.47	~	25.16

Figure 2.8 FPGA vs GPU Comparion with ESCA (Bhowmik et al., 2021).



ULTRASCALE FPGA RESOURCE UTILIZATION

	LUT	FF	DSP	BRAM	Power (W)	Fmax
EDL	154404	210744	0	-	2.044	
CL	318016	450896	2048	-	7.876	320
FCL	1991	1816	52	-	0.235	MHz
Overall	469288	663488	2100	27	10.68	
Utilization	39.68(%)	28.06(%)	30.70(%)	1.25(%)	-	

Figure 2.9 ESCA Resource and Power Requirement (Bhowmik et al., 2021).

2.5 Quantization

Deep Learning quantization is a technique used to reduce the precision of parameters in Neural Networks, without significantly compromising their performance. This technique is valuable for deploying deep learning models on edge devices, such as mobile phones and IoT devices, where memory and computational resources are limited. Pre-trained CNN models retain their parameters in 32-bit IEEE-754 floating-point format. The quantization process converts these parameters from 32-bit IEEE-754 floating-point format to a fixed-point format with a specified number of bits. This conversion reduces memory usage and enables faster computation (Courbariaux et al., 2016).

There are 2 types of quantization methods extensively used:

2.5.1 Post-Training Quantization (PTQ)

Post-Training Quantization (PTQ) is implemented subsequent to the full training of the model. This approach involves transforming the parameters of the trained model into lower-precision formats, such as converting from 32-bit floating-point representations to 8-bit integers. The primary advantage of PTQ lies in its simplicity, as it can be applied without the need for retraining or access to the original training data. However, this simplicity often comes with a trade-off; the reduction in precision can lead to a degradation in model accuracy, particularly in cases where the model is sensitive to small numerical variations. As a result, the deployment of PTQ requires careful consideration of the balance between computational efficiency and the performance impact on the model's.

2.5.2 Quantization-Aware Training (QAT)

Quantization-Aware Training (QAT) is a method that simulates the effects of quantization throughout the training process of a neural network. By incorporating quantization into both the forward and backward passes, the model is able to learn how to compensate for the errors that arise due to reduced numerical precision. This adaptation process allows the network to adjust its parameters in response to the quantization effects, ultimately leading to a more robust model. As a result, QAT generally yields higher accuracy than Post-Training Quantization (PTQ), as the model is better equipped to handle the challenges posed by lower precision during inference. This advantage is particularly significant in scenarios where maintaining model performance.

In this thesis, the potential degradation in accuracy due to quantization is not a primary concern, as the focus is centered on assessing the resilience of deep learning applications under various conditions. To facilitate this evaluation, the PTQ method is employed. The implementation of PTQ in this study utilizes a linear quantization algorithm, which is outlined in Algorithm 1 (8). Linear quantization, recognized for its simplicity and direct approach, is among the most fundamental techniques for reducing the precision of neural network parameters while maintaining an acceptable level of performance.

Algorithm 1 Quantization Algorithm

```
1: Input: Floating-point weights $\mathbf{W}$, quantization bits $b$
2: Output: Quantized weights $\mathbf{Q}$
3: Initialize $\mathbf{Q}$ as an empty array
4: for each weight $w$ in $\mathbf{W}$ do
5:     Determine the quantization scale $s$ based on $b$
6:     Quantize $w$ using $s$: $q = \text{round}(w * 2^s)/2^s$
7:     Store $q$ in $\mathbf{Q}$
8: return $\mathbf{Q}$
```

2.6 AMD Xilinx Vitis AI

Vitis AI is a comprehensive software development platform developed by AMD Xilinx to facilitate the deployment of artificial intelligence (AI) inference on AMD Xilinx hardware platforms, including Field-Programmable Gate Arrays (FPGAs) and Adaptive Compute Acceleration Platforms (ACAPs) (AMD, 4 25). By providing an end-to-end solution, Vitis AI bridges the gap between software development and hardware acceleration, enabling developers to leverage the performance and flexibility of AMD Xilinx devices without requiring extensive knowledge of hardware design. This platform integrates a range of tools and libraries, offering a streamlined workflow from model development to deployment.

Vitis AI supports several major deep learning frameworks, including TensorFlow, PyTorch, and Caffe as shown in Figure 2.10. This broad support allows developers to continue using their preferred frameworks for model development leveraging the powerful inference capabilities of AMD Xilinx hardware. The seamless integration with these popular frameworks ensures that models can be easily imported into the Vitis AI toolchain, where they can be quantized, compiled, and deployed with AMD Xilinx DPU IP.

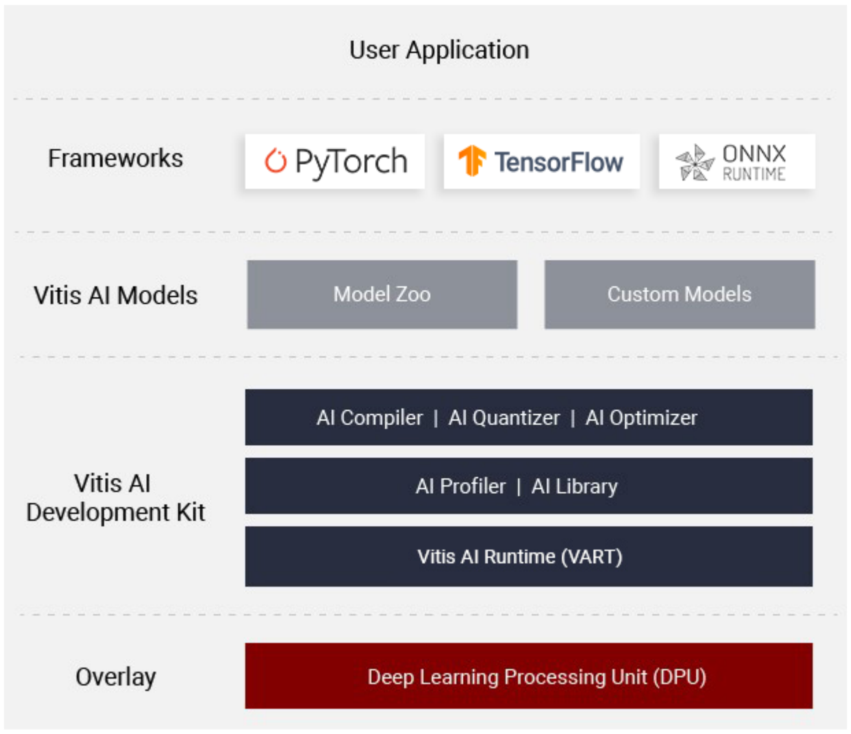


Figure 2.10 Vitis AI Overlay (AMD, 2024).

2.6.1 Vitis AI Model Zoo

The Vitis AI includes Model Zoo, a repository of pre-trained models. These models span a variety of AI applications such as image classification, object detection, and natural language processing. The models in the Vitis AI Model Zoo are created and optimized for performance, making it easier for developers to deploy AI solutions on AMD Xilinx devices. By providing these pre-optimized models, Vitis AI significantly reduces the time and effort required to develop and deploy AI applications.

2.6.2 Vitis AI Quantizer

Quantization is a crucial step in deploying AI models on hardware, as it reduces the computational and memory requirements by converting pre-trained 32-bit floating-point parameters and activations to lower-precision formats, typically 8-bit fixed point integers as shown in Figure 2.11. The Vitis AI Quantizer is a tool that automates this process, ensuring minimal loss in accuracy while significantly enhancing performance and efficiency on AMD Xilinx devices. The quantizer supports both post-training quantization and quantization-aware training.

2.6.3 Vitis AI Compiler

The Vitis AI Compiler is designed to optimize and compile deep learning models for execution on AMD Xilinx hardware. It takes models trained in popular deep learning frameworks and translates them into an efficiently executable format on AMD Xilinx devices. The compiler performs various optimizations, such as layer fusion, data layout transformations, and hardware-specific tuning, to maximize the performance of the deployed model. The Vitis AI Compiler supports a variety of deep learning frameworks, ensuring compatibility and ease of integration into existing AI development workflows. Before compilation, the trained DNN model shall be quantized into 8-bit precision with the Vitis AI Quantizer.

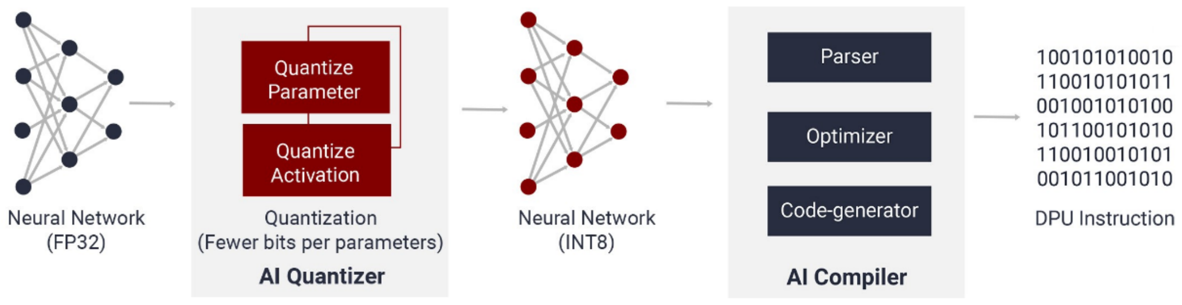


Figure 2.11 Vitis AI Compilation Process Overview (AMD, 4 25)

2.6.4 Vitis AI Profiler

Profiling is a critical step in deep learning applications development, particularly when targeting FPGA platforms. FPGAs offer a high degree of flexibility and parallelism, which can be used to accelerate deep learning workloads. However, achieving optimal performance requires a deep understanding of the computational and memory requirements of the neural network. Profiling provides the necessary insights into how the network performs on the hardware. Vitis AI Profiler is an integral tool within the Vitis AI development environment, designed to optimize and accelerate the deployment of deep learning applications on AMD Xilinx hardware platforms such as FPGAs and adaptive SoCs. Vitis AI Profiler provides detailed insights into the performance of neural networks, enabling developers to identify bottlenecks and optimize their designs for maximum efficiency.

2.6.5 Xilinx Intermediate Representation (XIR)

Xilinx Intermediate Representation (XIR) is a intermediate-level representation of deep learning models designed for execution on AMD Xilinx's acceleration platforms. XIR abstracts the details of the model, encapsulating the neural network's structure, parameters, and computational graph in a format that can be easily parsed and optimized for hardware execution.
The primary objective of XIR is to bridge the gap between high-level neural network frameworks (such as TensorFlow, PyTorch, and Caffe) and low-level hardware-specific implementations. By converting models from their native formats into XIR, developers can leverage AMD Xilinx's tools and libraries to optimize and deploy these models on the Data Center accelerator and SoCs efficiently.
An XModel file is a serialized representation of a deep learning model in the Xilinx Intermediate Representation format. It encapsulates the entire computational graph of the model, including layers, weights, biases, and other parameters necessary for inference. XModel files are generated during the model compilation process, where a trained neural network from a high-level framework is converted into the XIR format. This process involves various optimization steps, such as layer fusion and pruning, to tailor the model for efficient execution on AMD Xilinx hardware.
The structure of an XModel file is hierarchical, reflecting the organization of the neural

network it represents. It consists of the following key components:

2.6.5.1 Subgraph

In the context of Xilinx Intermediate Representation (XIR), a subgraph is defined as a distinct computational segment within the broader neural network architecture. This subgraph can be conceptualized as a smaller, self-contained graph embedded within the larger model graph, encapsulating a specific sequence of operations, often referred to as 'ops.' The primary function of these subgraphs is to simplify and streamline the process of model execution by allowing the network to be divided into more manageable and modular components. This modularity not only facilitates efficient computation but also enhances the scalability and flexibility of neural network models, particularly in hardware-accelerated environments such as FPGAs. By organizing the model into subgraphs, developers can optimize and execute specific portions of the network independently, thereby improving overall performance and resource utilization.

2.6.5.2 Op (Operation)

In the context of Vitis AI, an operation (or 'op') in the Xilinx Intermediate Representation (XIR) corresponds to a distinct computational task within the neural network. These tasks include fundamental operations such as convolution, pooling, activation, or matrix multiplication, which collectively form the building blocks of DNNs. Each operation is defined by its type, input tensors, output tensors, and a set of associated parameters, such as kernel size, stride, and padding in the case of a convolution operation.

2.6.5.3 TensorBuffer

TensorBuffers are used to represent the input and output data of each op within the XModel. They encapsulate information about the data type, shape, and memory address of the tensors. TensorBuffers are critical for managing the flow of data through the computational graph during inference.

The organization of XModel files allows for efficient parsing and execution of DNN models by the AMD Xilinx DPU. Each subgraph can be processed independently, and ops within each subgraph can be executed in a sequence determined by the computational dependencies. Figure 2.12 is an example XModel computation graph portion with three different subgraph. Circled with red color is assigned to CPU subgraph which ops will be executed on CPU part since DPU does not support tanh activation function. Blue circled subgraphs are assigned to DPU which the first one realizes convolution and the last one pooling operation. Each ops and subgraphs has its own tensorbuffer to manage dataflow between subgraphs and ops within subgraphs.

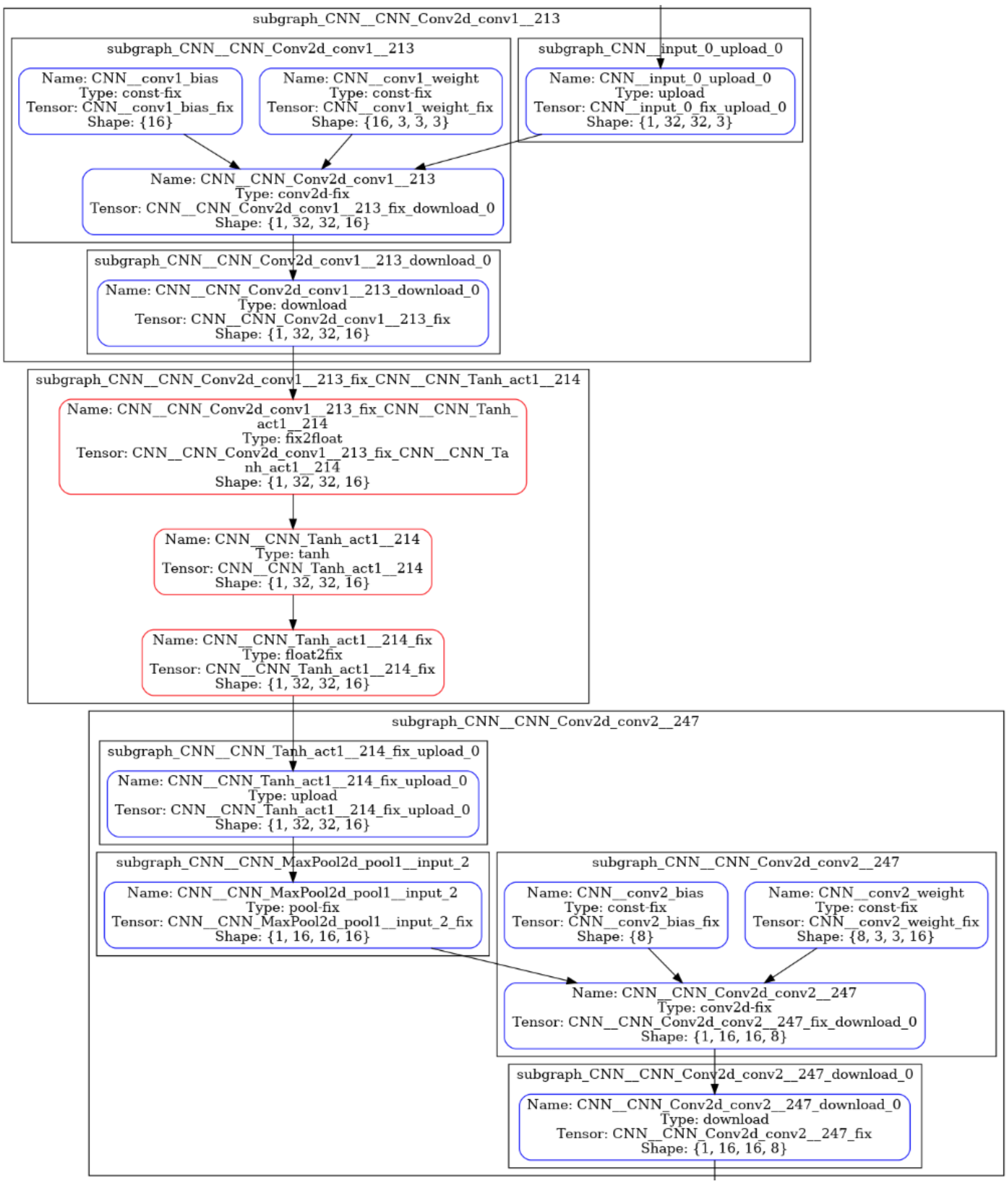

subgraph_CNN__CNN_Conv2d_conv1__213
subgraph_CNN__CNN_Conv2d_conv1__213
subgraph_CNN__input_0_upload_0
Name: CNN__conv1_bias Type: const-fix Tensor: CNN__conv1_bias_fix Shape: {16}
Name: CNN__conv1_weight Type: const-fix Tensor: CNN__conv1_weight_fix Shape: {16, 3, 3, 3}
Name: CNN__input_0_upload_0 Type: upload Tensor: CNN__input_0_fix_upload_0 Shape: {1, 32, 32, 3}
Name: CNN__CNN_Conv2d_conv1__213 Type: conv2d-fix Tensor: CNN__CNN_Conv2d_conv1__213_fix_download_0 Shape: {1, 32, 32, 16}
subgraph_CNN__CNN_Conv2d_conv1__213_download_0
Name: CNN__CNN_Conv2d_conv1__213_download_0 Type: download Tensor: CNN__CNN_Conv2d_conv1__213_fix Shape: {1, 32, 32, 16}
subgraph_CNN__CNN_Conv2d_conv1__213_fix_CNN__CNN_Tanh_act1__214
Name: CNN__CNN_Conv2d_conv1__213_fix_CNN__CNN_Tanh_act1__214 Type: fix2float Tensor: CNN__CNN_Conv2d_conv1__213_fix_CNN__CNN_Tanh_act1__214 Shape: {1, 32, 32, 16}
Name: CNN__CNN_Tanh_act1__214 Type: tanh Tensor: CNN__CNN_Tanh_act1__214 Shape: {1, 32, 32, 16}
Name: CNN__CNN_Tanh_act1__214_fix Type: float2fix Tensor: CNN__CNN_Tanh_act1__214_fix Shape: {1, 32, 32, 16}
subgraph_CNN__CNN_Conv2d_conv2__247
subgraph_CNN__CNN_Tanh_act1__214_fix_upload_0
Name: CNN__CNN_Tanh_act1__214_fix_upload_0 Type: upload Tensor: CNN__CNN_Tanh_act1__214_fix_upload_0 Shape: {1, 32, 32, 16}
subgraph_CNN__CNN_MaxPool2d_pool1__input_2
Name: CNN__CNN_MaxPool2d_pool1__input_2 Type: pool-fix Tensor: CNN__CNN_MaxPool2d_pool1__input_2_fix Shape: {1, 16, 16, 16}
subgraph_CNN__CNN_Conv2d_conv2__247
Name: CNN__conv2_bias Type: const-fix Tensor: CNN__conv2_bias_fix Shape: {8}
Name: CNN__conv2_weight Type: const-fix Tensor: CNN__conv2_weight_fix Shape: {8, 3, 3, 16}
Name: CNN__CNN_Conv2d_conv2__247 Type: conv2d-fix Tensor: CNN__CNN_Conv2d_conv2__247_fix_download_0 Shape: {1, 16, 16, 8}
subgraph_CNN__CNN_Conv2d_conv2__247_download_0
Name: CNN__CNN_Conv2d_conv2__247_download_0 Type: download Tensor: CNN__CNN_Conv2d_conv2__247_fix Shape: {1, 16, 16, 8}

Figure 2.12 Example XModel portion (AMD, 2023).

2.7 Deep Processing Unit

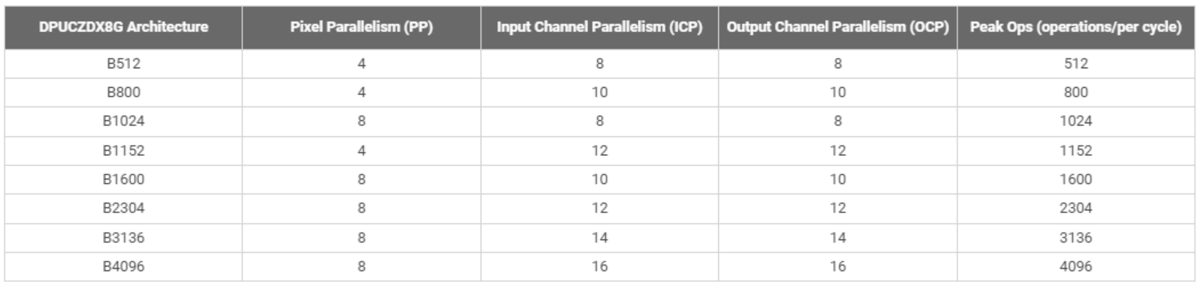
The AMD Xilinx Deep Learning Processing Unit (DPU) is a programmable logic IP core optimized for accelerating the inference of DNN models. It is specifically designed to execute the computationally intensive operations of neural networks, such as convolutions, pooling, activation functions, and fully connected layers. By offloading these operations to the DPU, developers can achieve significant performance improvements and power savings compared to running deep learning models on general-purpose CPUs or GPUs.
The characteristics of Convolutional Neural Network (CNN) layers can vary significantly. For instance, ResNet-18 (He et al., 2016) incorporates traditional convolutional layers, whereas MobileNet V2 Sandler et al. (2018a) uses depth-wise convolutional layers. To accommodate these differences, the AMD Xilinx DPU is equipped with specialized commands that enable it to implement various CNN models. These commands compile models created in an environment called Vitis AI. Vitis AI converts deep learning architectures defined in Python deep learning frameworks such as PyTorch, or TensorFlow into an XModel file. This XModel file contains the necessary instructions for layer implementation and the associated weight parameters.
The AMD Xilinx DPU IP is a crucial component of the Vitis AI development stack, offering a comprehensive suite of tools for deploying AI models on AMD Xilinx platforms. Its architecture and functionality are highly configurable, allowing for customization to meet the specific requirements of various applications and hardware environments. The AMD Xilinx DPU IP specific parameters enable optimal management of FPGA resources for the best performance on the target device. These parameters include the number of DPU cores, DPU architecture, RAM usage, channel augmentation, arithmetic logic unit (ALU) parallelism, and softmax enable.
Certain DPU parameters impact the Vitis AI compilation process. Based on these parameters, Vitis AI generates DPU instructions to execute the target CNN on the dedicated DPU. After DPU IP is compiled with Vivado or Vitis, the arch.json file is generated by tools that includes DPU parameters. This arch.json file is specified during the compilation step. The parameters that affects Vitis AI compilition process are given as follows:

- *DPU Cores*: Specifies the number of DPU cores that will operate in parallel.
- *DPU Architecture*: Defines the structure and organization of the convolution compute unit within each DPU core. AMD Xilinx introduces three types of parallelism related to multiply and accumulate (MAC) operations in convolution: pixel parallelism (PP), input channel parallelism (ICP), and output channel parallelism

(OCP). The convolution operation speed is determined by the equation:

$$PP \times ICP \times OCP \times 2 \tag{2.1}$$

Eq 2.1 calculates the number of MAC operation per clock cycle. Figure 2.13 shows parallelism numbers for different DPU Architectures. In this work, B4096 architecture is selected.



DPUCZDX8G Architecture	Pixel Parallelism (PP)	Input Channel Parallelism (ICP)	Output Channel Parallelism (OCP)	Peak Ops (operations/per cycle)
B512	4	8	8	512
B800	4	10	10	800
B1024	8	8	8	1024
B1152	4	12	12	1152
B1600	8	10	10	1600
B2304	8	12	12	2304
B3136	8	14	14	3136
B4096	8	16	16	4096

Figure 2.13 Parallelism for Different Convolution Architectures (AMD, 2020).

- *RAM Usage*: Offers two options: low RAM usage and high RAM usage. This parameter affects block RAM usage within the FPGA. When the DPU executes CNN models, weights, biases, and intermediate feature maps are stored in the FPGA block RAM to maximize the performance. Depending on the DPU architecture and RAM usage, block RAM utilization varies.

The operation of the Deep Processing Unit (DPU) also necessitates the involvement of the Application Processing Unit (APU) as shown in Figure 2.14. The APU is responsible for handling interrupts, which are essential for coordinating the transfer of data. This coordination ensures that data is efficiently moved between the various components, maintaining the smooth and effective functioning of the DPU.

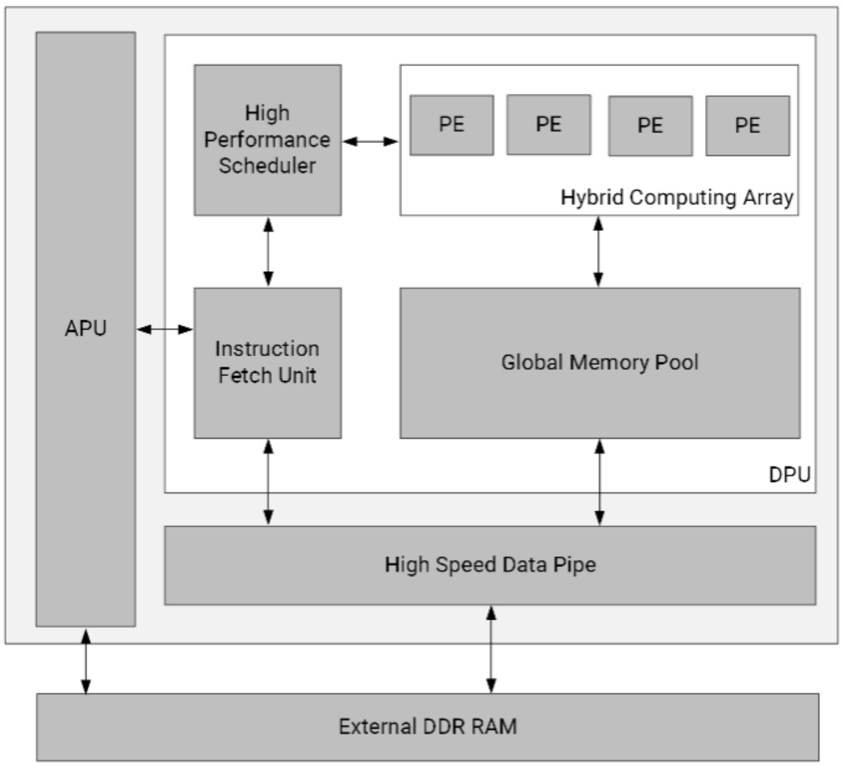


Figure 2.14 DPU Top-Level Block Diagram (AMD, 2020).

2.8 Error Injection

In this thesis, the error injection method is adopted from Rathore's (Rathore, 2021) PyTorch framework and improved to enable detailed error injection simulation. Additionally, in this thesis, a new error injection method is designed to inject errors for both per-bit and per-layer errors. This framework for analyzing DNN error resilience involves several key steps, including model setup, quantization, error injection, and inference evaluation. The framework is implemented in PyTorch and takes pre-trained neural network models as input. The models are quantized to a fixed-point precision, with weights and forward activations quantized based on user-defined bit precisions. The quantized models are then duplicated, and a `QuantandError` layer is inserted after each convolutional and fully-connected layer to facilitate error injection. Bit-level errors are introduced in the quantized activations by randomly flipping data bits at specified error rates. This process is implemented using error tensors that indicate the bit positions with errors. The error tensors are applied to the original data tensors, resulting in faulty data tensors used for inference evaluation. The inference accuracy of the faulty models is evaluated to quantify error resilience of given DNN model. The framework calculates the Top-1 and Top-5 accuracy to determine the impact of errors on network performance.

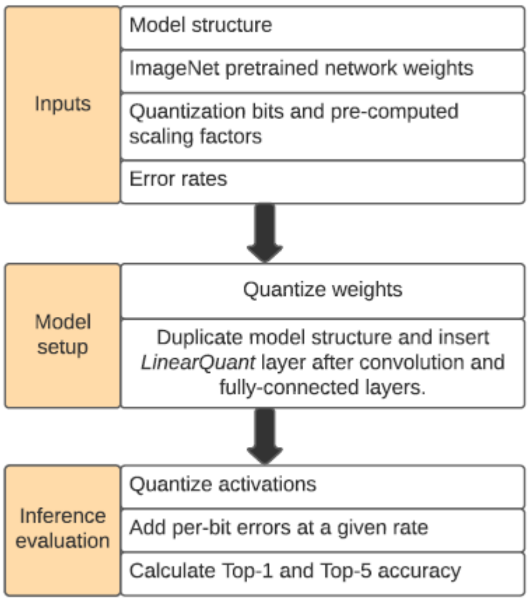

Inputs
Model structure
ImageNet pretrained network weights
Quantization bits and pre-computed scaling factors
Error rates
Model setup
Quantize weights
Duplicate model structure and insert *LinearQuant* layer after convolution and fully-connected layers.
Inference evaluation
Quantize activations
Add per-bit errors at a given rate
Calculate Top-1 and Top-5 accuracy

Figure 2.15 DNN error resilience evaluation framework (Rathore, 2021).

The proposed methodology for quantifying the error resilience of DNNs involves several key steps. This section describes the framework's implementation, including the input parameters, model setup, error injection process, and inference evaluation as shown in Figure 2.15. The framework requires several inputs to configure the error resilience analysis:

- *Quantization Parameters*: Defines the bit precision for quantizing weights and activations. This precision can be different for each layer, allowing for fine-tuned analysis.
- *Error Rates*: Specifies the rate at which bit-level errors are injected. Different error rates can be applied to various layers and bits to study their impact.
- *Error Locations*: Indicates where the errors should be injected.

The neural network model is prepared for quantization and error injection. The setup process includes the following steps:

- *Weights*: Pre-trained weights are quantized to a fixed-point format as defined by the quantization parameters.
- *Activations*: Activations are computed in floating-point format and then quantized

to fixed-point format. This is achieved by duplicating the original network and adding a QuantandError layer after each convolutional and fully-connected layer.

For error injection, bit-level errors are injected into the quantized activations by bit flipping. Errors are modeled by generating an error tensor that identifies bit positions with errors. This tensor is applied to the original data tensor to produce a faulty data tensor.

3. Proposed Metodology

Two different approaches were introduced to evaulate the fault resilience of deep neural network models. The first approach was PyTorch framework on CPU added to the Error Injection framework mentioned in Section 2.8. The second approach was designed for fast error injection on the AMD Xilinx UltraScale+ MPSoC using Vitis AI. In this section, both approaches will be explained.

3.1 Error Injection with Pytorch Framework on CPU

In this framework, the error injection process is designed to be highly flexible and precise, allowing for both per-bit and per-layer error control. The framework utilizes a 2D format for error rates, allowing for a detailed and precise fault injection across the neural network layers. This section provides an in-depth explanation of the framework, its structure, and the error injection process. The framework core is the 2D error matrix, where the number of rows are equal to the number of `QuantandError` layers in the neural network, and the number of columns are equal to the quantization bit width. Each row of the matrix corresponds to a specific `QuantandError` layer, while each element within a row corresponds to the bit error rate for a specific bit position in that layer.

Matrix structure is explained with the following:

- *Rows*: Each row in the 2D matrix represents a *QuantandError* layer in the neural network. For instance, if there are four `QuantandError` layers in the network, there will be four rows in the error matrix. The first row corresponds to the first `QuantandError` layer, the second row to the second `QuantandError` layer, and so on.
- *Columns*: Each column in the matrix represents a specific bit position within the quantized data. The columns range from the least significant bit (LSB) on the far right to the most significant bit (MSB) on the far left. If the quantization bit width is 8, there will be eight columns in the matrix.
- *Elements*: Each element in the matrix is a value representing the bit error rate for that particular bit in the respective layer. This allows for precise control over which bits are affected by errors and to what extent. The value of each element is a probability between 0 and 1, indicating the likelihood of a bit-flip occurring at that position.

The error injection process is implemented as follows:

1.1 *Initialization*: The framework takes the 2D error matrix as an input containing the error rates. The dimensions of the matrix should be set according to the number of `QuantandError` layers and the quantization bit width. Users can specify different error rates for each bit position in each layer.

1.2 *Layer-wise Processing*: During the forward pass of the neural network, each `QuantandError` layer processes its input data sequentially. For each layer, the corresponding row in the error matrix is retrieved. This row contains the error rates for all bit positions in that layer.

1.3 *Bit-wise Error Injection*: For each bit position in the quantized activations, the error rate specified in the matrix element is applied. Random bit-flips are introduced based on this error rate, simulating faults in the respective bit positions.

Users can define different error rates for different layers and bit positions, allowing for customized fault tolerance testing. This flexibility is particularly useful for exploring various fault scenarios and for optimizing the network's resilience to errors.

3.1.1 Example Implementation

Consider a neural network with four `QuantandError` layers and a quantization bit width of 8 bits. The 2D error matrix for this setup would have 4 rows and 8 columns. An example matrix might look like this:

$$\begin{bmatrix} 0.001 & 0.002 & 0.001 & 0.005 & 0.003 & 0.004 & 0.002 & 0.007 \\ 0.002 & 0.001 & 0.003 & 0.004 & 0.002 & 0.005 & 0.003 & 0.002 \\ 0.003 & 0.002 & 0.001 & 0.002 & 0.003 & 0.004 & 0.005 & 0.001 \\ 0.004 & 0.003 & 0.002 & 0.001 & 0.005 & 0.002 & 0.001 & 0.004 \end{bmatrix}$$

In this matrix:

- The first row corresponds to the first `QuantandError` layer, with specific error rates for each bit position. For instance, the element at the first row and the first column (0.001) represents the error rate for the MSB of the first `QuantandError` layer, while the element at the first row and the last column (0.007) represents the error rate for the LSB of the same layer.
- The second row corresponds to the second `QuantandError` layer etc.

3.2 Error Injection with FPGA

To rapidly assess the fault resilience of deep neural network models, a novel error injection platform has been developed using the AMD Xilinx UltraScale+ MPSoC. This particular hardware platform was chosen due to its seamless compatibility with the Vitis AI framework, which is essential for efficient deployment and execution of AI models on AMD Xilinx FPGA-based systems. The core component of this platform is an Error Injection Intellectual Property (IP) block, designed using Verilog HDL. This IP block is responsible for injecting faults into the outputs of both convolutional and fully connected layers, ensuring high-throughput performance during the error resilience evaluation process.

Custom operator with Vitis AI flow was implemented in order to realize Error Injection IP next to each convolution and fully connected layers. During run-time, Xilinx Run Time (XRT) (AMD, 2024d) library was used to manage Error Injection IP with software for decleration of parameters and control of dataflow. The overview flow of Error Injection with FPGA is shown in Figure 3.1.

This section of the Thesis dedicated to Error Injection with FPGA is organized into three detailed subsections. The first subsection covers the creation of the Vitis AI models, outlining the steps involved in preparing the neural network for deployment on the FPGA. The second subsection delves into the design of the run-time software, which manages the execution of the models and the error injection process. Finally, the third subsection provides an in-depth exploration of the Error Injection IP, explanation of its architecture.

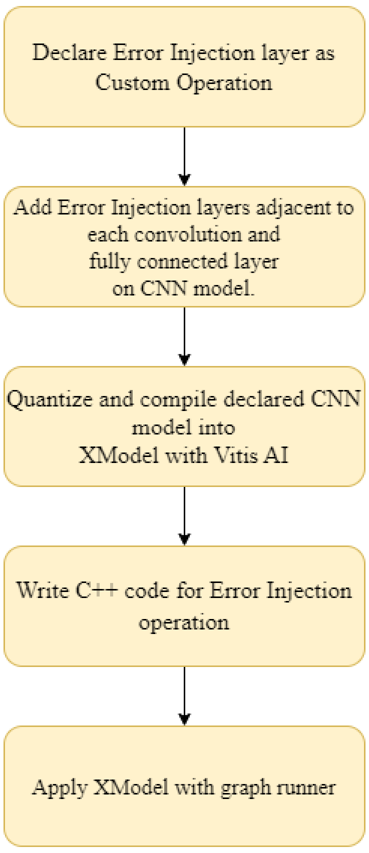

Declare Error Injection layer as Custom Operation
Add Error Injection layers adjacent to each convolution and fully connected layer on CNN model.
Quantize and compile declared CNN model into XModel with Vitis AI
Write C++ code for Error Injection operation
Apply XModel with graph runner

Figure 3.1 Error Injection with SoC Overview.

3.2.1 Vitis AI Model Creation

Vitis AI manages the compilation process of convolutional neural network (CNN) models, allowing them to run efficiently on the AMD Xilinx DPU IP within the FPGA portion of the system-on-chip (SoC). These models, whether custom-built or pre-trained, are defined in widely used deep learning frameworks such as PyTorch Paszke et al. (2017) or TensorFlow (Abadi et al., 2015).

When the DPU IP does not support all the layers of a CNN model, it becomes necessary to manually implement these unsupported layers using a custom operator (custom op) flow. This approach creates a sub-graph to execute the custom layer on the CPU. For operations such as error injection, which is a custom layer, a custom operator is declared in CNN model architecture.

The definition of the custom layer must be integrated within the deep learning framework. In this work, the PyTorch framework is employed. Since it is aimed to apply errors to the outputs of convolutional and fully connected layers, the error injection layer must be declared adjacent to each layer. While defining this custom operation, its functionality must be incorporated within the CNN model. This requirement ensures that Vitis AI adjusts the quantization process by the custom operation.
Since error injection on activation data occurs at runtime, the custom operation for the Error Injection layer passes the incoming data to the next layer as defined in the layer structure. Once the custom model is compiled with the custom operator, the software is built to implement the custom operation while executed on runtime. This software will then allow the input data to be processed on the Error Injection IP.

3.2.2 Run-time Software Design

This section delves into the run-time software design utilized for implementing fast DNN error injection.

Vitis AI compiles defined CNN layers into `subgraphs` that the Deep Processing Unit can handle efficiently. The initial step involves transforming the input CNN model layers into the Xilinx Intermediate Representation format. To do this, Vitis AI applies various optimizations, dividing it into subgraphs based on whether operations can be executed on the DPU or the CPU. These `subgraphs` includes operators based on the CNN model definition. The Vitis AI library incorporates operators to compute these subgraphs. Ultimately, the optimized graph, along with all required information and instructions are serialized into a compiled XModel file.

The graph runner is a runtime function designed to execute XModel files generated by the Vitis AI. As shown in Figure 3.2 it orchestrates the execution of the layers defined in the

XModel, ensuring that each layer is processed sequentially and manages data flow between these layers. The graph runner supports the integration of custom layers, allowing for specialized processing as needed by the application. When executing an XModel, the graph runner loads the XModel file, which contains a serialized representation of the neural network architecture, including all the layers and their configurations. For each layer, the graph runner invokes the corresponding execution code. For standard layers (e.g., convolutional, pooling), it uses pre-defined implementations. For custom layers, such as our Error Injection layer, it calls the user-provided C++ code. If the defined layers supported by DPU, then it is assigned to DPU device to executed on FPGA. The graph runner handles the movement of data between layers, ensuring that inputs and outputs are correctly passed along the computational graph.

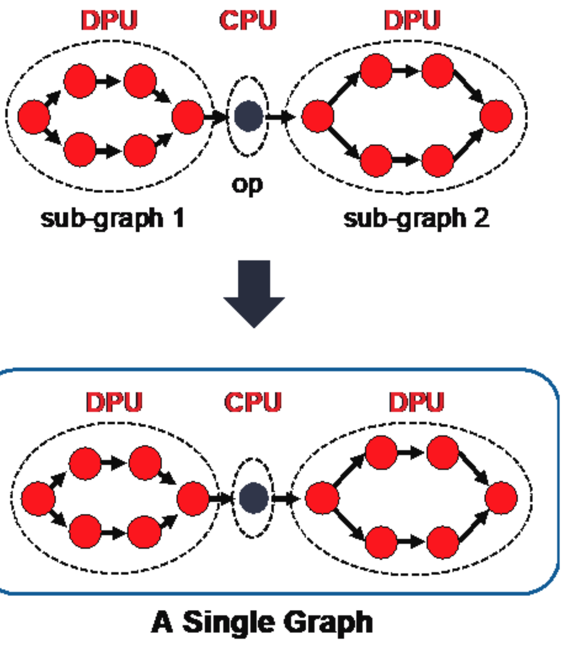


Figure 3.2 Graph Runner Explanation (AMD, 2024).

The Error Injection layer is a custom layer integrated into the XModel file. This layer is responsible for injecting faults into the DNN model during runtime, using the Error Injection IP designed with Verilog. When the graph runner reaches the Error Injection layer in the XModel, it invokes the custom software code associated with this layer.

This custom software code uses the Xilinx Run Time library. XRT is a runtime library that provides the necessary interfaces for interacting with FPGA-based accelerators. It supports data movement, kernel execution, and device management, enabling seamless integration between software applications and FPGA hardware. XRT provides APIs to transfer data between the host memory and the FPGA device.

In the custom software code for the Error Injection layer, the parameters associated with the layer are first written to specified Error Injection IP memory addresses. Subsequently, a buffer object is created and populated with input data. After the Error Injection IP completes its process, the same buffer object is updated with the output data. The buffer object can hold a maximum size of 16,384 bytes. For large activation data, such as the first activation layer in ResNet-18 (He et al., 2015), which involves a 112x112x64 tensor, the data must be processed in iterations. Once the error injection operation is completed, the output data is sent to the input tensor of the next subgraph without data-flow management using the graph runner function.

3.2.3 Error Injection with FPGA

A novel fault injection platform has been developed to assess the fault tolerance of DNNs with high throughput. This platform leverages the parallel architecture of the FPGA to simultaneously execute DNN operations and error injection processes. The Arm Cortex-A53 processor primarily supports the execution of DNN layers on the AMD Xilinx DPU IP by running Vitis AI software functions. Furthermore, the XRT library is used to transmit activation data, which is subsequently subjected to error injection via the Error Injection IP.

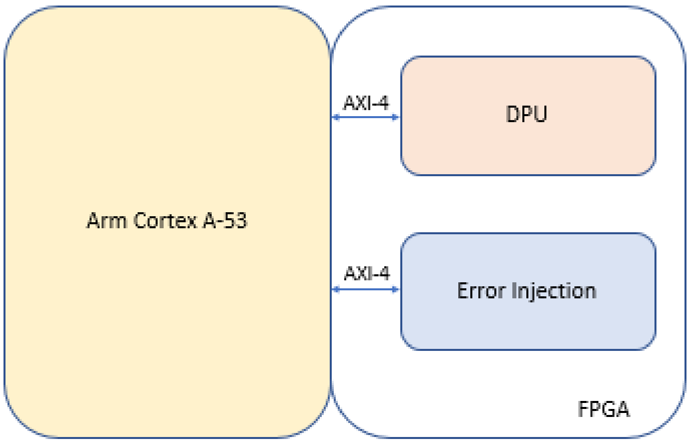


Figure 3.3 Error Injection with FPGA Overall Architecture

3.2.4 Error Injection IP

To implement error injection on specified convolutional and fully connected layers activations with given error rates, an Error Injection IP (Intellectual Property) using Verilog HDL is designed. This IP begins by converting the incoming activation data from a 32-bit floating point format to a 32-bit fixed point format. After this conversion, the 32-bit data is further quantized into an 8-bit signed integer format. This is necessary because Vitis AI supports only 8-bit quantization, making it imperative for the Error Injection IP to realize the 8-bit quantization process.

During the quantization process, Error Generator produces defined errors per bit, which are then injected into the quantized data using an XOR gate. This mechanism effectively implements bit flips on specified bits according to the generated errors per bit. Following error injection, the data is converted back to the 32-bit floating point format and written back to DDR memory via the AXI-4 interface, as shown in Figure 3.4.

The Error Injection IP requires several parameters, such as the scaling factor for data quantization and the error rate per bit. Initially, these parameters are declared within the CPU and subsequently written to the Error Injection IP. An error generation IP is incorporated within the Error Injection IP to generate the desired error per bit amount.

A true random number generator (TRNG) for error generation is not required in this application. Instead, a Linear Feedback Shift Register (LFSR) is employed to generate random errors. The random numbers produced by the LFSR pass through a filter within the error generation IP to control the generated errors per bit.

This methodology ensures that the error injection process is precise and effective, adhering to the constraints and requirements of the Vitis AI framework. By meticulously managing the conversion, quantization, and error injection processes, the designed Error Injection IP achieves the desired functionality while maintaining compatibility with existing CNN architectures and hardware interfaces.

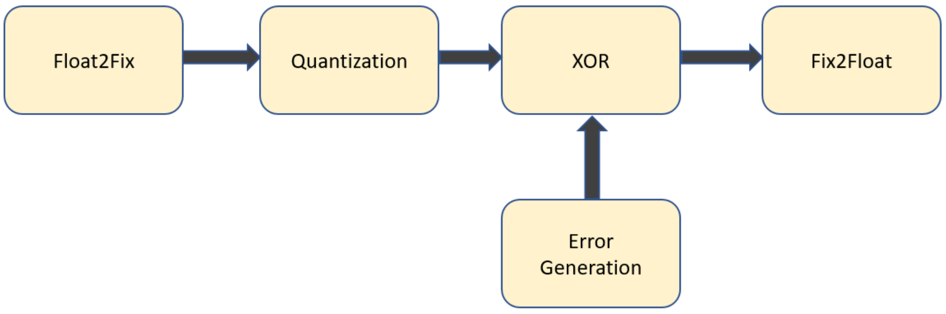


Figure 3.4 Error Injection Dataflow

3.2.4.1 Error Generation

The Error Generation module is designed to produce error according to given per-bit error rates. At the onset of error injection for each layer, a designated quantity of errors must be given as input into a specific register through software control. Each bit has its own error register that maintains the total count of errors. Once the error injection process commences, the error generation IP module accesses the error ratio and generates errors based on given error rates.

To introduce random errors, the system employs linear feedback shift registers (LFSR). In this context, a true random number generator is deemed unnecessary, opting instead for a simple yet effective LFSR configuration. Within the Error Generation IP module, a filter module is incorporated to regulate the output of the LFSR. The filter component effectively refines the LFSR output in accordance with the error ratio. It employs logic AND and logic OR gates at each bit position in the LFSR output as shown in Figure 3.5. If the desired error is generated but the LFSR continues to produce an error bit for a specific location, the filter converts it to a non-error bit using an AND gate. Conversely, if an insufficient number of errors is generated, the filter IP generates an error bit for that particular bit location. This algorithm ensures the targeted quantity of errors is successfully generated.

The error generation process works parallel with the quantization step. While incoming data is quantized, errors also are produced.

Algorithm 2 Error Generation Algorithm

```
 1: Input: Error Ratio for 8-bit ER, Total Iteration Iter
 2: Output: Error per bit Err
 3: Initialize Iter_temp ← Iter
 4: Initialize each element of AND filter ANDF to 1
 5: Initialize each element of OR filter ORF to 0
 6: for $i$ ← Iter to 0 do
 7:     Generate 8-bit LFSR output LFSR
 8:     for $f$ ← 0 to 7 do
 9:         if ER[$f$] == 0 then
10:             ANDF[$f$] ← 0
11:         else if ER[$f$] == $i$ then
12:             ORF[$f$] ← 1
13:             ER[$i$] ← ER[$i$] − 1
14:         else if LFSR[$f$] == 1 then
15:             ER[$f$] ← ER[$f$] − 1
16:         Err[$f$] ← (LFSR[$f$] $\wedge$ ANDF[$f$]) $\vee$ ORF[$f$]
17: return Err
```

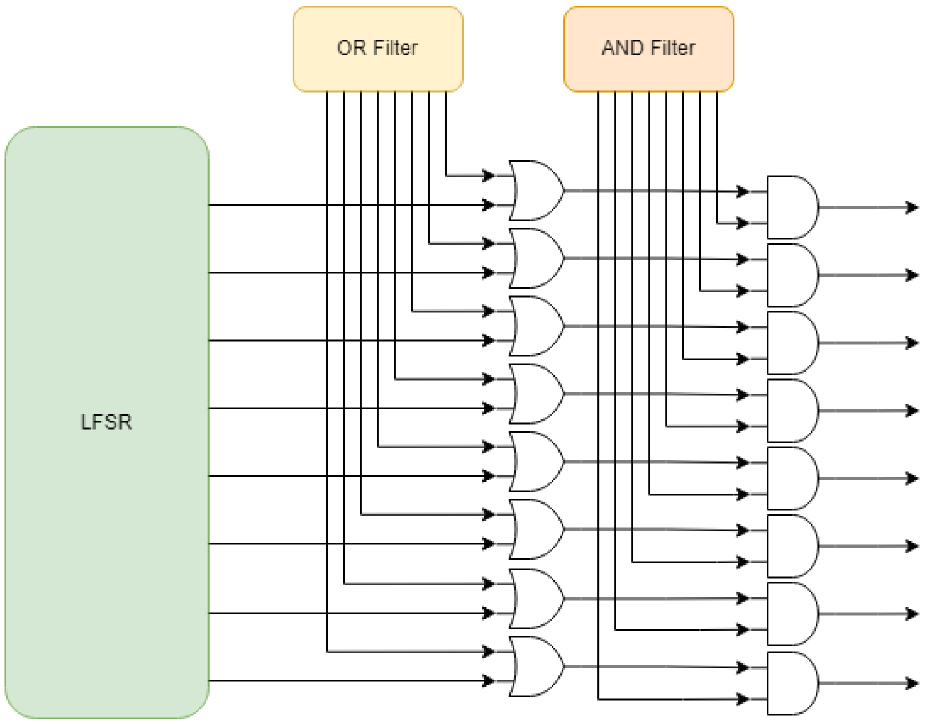


Figure 3.5 Error Generation Architecture

3.2.4.2 Quantization

Vitis AI supports only 8-bit quantization for AMD Xilinx Ultrascale+ MPSoC devices. To this end, intermediate activation data should be quantized into an 8-bit fixed point data format. Since the incoming data format is a 32-bit floating point, floating point to fixed point conversion must be realized before quantization. To this end, 32-bit floating-point data is converted into 32-bit fixed-point format with 8-bit integer and 24-bit fraction. This is realized by shifting the mantissa part of floating-point data by the value of (128-exponent). If the exponent is greater than 128 then the shift left operation is realized visa versa. After that according to the scaling factor, 32-bit fixed point data is quantized into 8-bit fixed point. The scale factor is used to decide the fraction bit width of data. This scaling factor is declared during the quantization process on Vitis AI. Algorithm 1 is a pseudo code for a quantization scheme that is realized on Error Injection IP. Then Quantized value passed through the XOR gate with the generated error. If bit 1 is generated for this bit, then bit-flip will be realized. Otherwise bit value will be the same. Finally, error injection was realized, and fixed-point to floating-point conversion was realized on the FPGA part. This process is realized pipelined with prior processes.

In fixed-point to floating-point conversion, the first bit is checked with value bit 1. To the bit index of the detected bit, the exponent part is calculated. The rest of the least significant bits are declared as the most significant bits in the mantissa part.

3.2.4.3 AXI-4 Read and Write

To enable data transfer between Error Injection IP and CPU, AXI-4 memory mapped (MM) interface is implemented in Error Injection IP. Since the read and write channels of the AXI-4 interface are separate, Error Injection IP can start writing data transfer while reading activation data from CPU memory.

4. RESULT AND COMPARISON

In this section, the inference speed performance without error injection for the DPU is compared with that of the CPU. The error injection speed per layer of the proposed Error Injection with the System-on-Chip (SoC) platform was measured using Vitis AI Profiler (2.6.3) for each layer. Subsequently, the optimization of the Error Injection platform is discussed.

4.1 DPU Performance without Error Injection

Several tests were conducted to evaluate the performance of the designed platform. The primary test measures the classification performance between CPU and SoC platforms in frames per second (FPS). FPS shows the total number of classified images per second. Two different devices were selected for this test. For the CPU, an AMD Ryzen 9 7900X with a clock speed of 4.70GHz and 12 cores was chosen. For the SoC, a Kria K26 SOM was used. This SoC includes the AMD Xilinx Ultrascale+ MPSoC, featuring a Quad-core Arm Cortex-A53 MPCore^{TM} with a clock speed of 1.5 GHz and an FPGA. Within the FPGA, the AMD Xilinx DPU IP core was implemented with a B4096 architecture with 4096 operations per cycle (2.13), running at 200 MHz for the AXI Interface and 400 MHz for the internal DSP clock speed. For benchmark models, ResNet-18, ResNet-50 (He et al., 2015), MobileNetV2 (Sandler et al., 2018b) and Inception V3 (Szegedy et al., 2015) CNN models were selected. The CPU-based CNN tasks were executed using the PyTorch deep learning framework.

Table 4.1 Accuracy and Inference Speed of Neural Networks running on CPU

Model	Accuracy (%)	Inference Performance (FPS)
ResNet-50	76.13	35,4
ResNet-18	70.04	101
MobileNetV2	71.88	70.1
InceptionV3	77.21	33,5

Table 4.2 Accuracy and Inference Speed of Neural Networks running on AMD Xilinx DPU

Model	Accuracy (%)	Inference Performance (FPS)
ResNet-50	72.18	101
ResNet-18	68.03	237
MobileNet V2	67.65	344
InceptionV3	73.27	68

Table 4.3 Accuracy Degredation and Performance Gain AMD Xilinx DPU

Model	Accuracy Degredation(%)	Performance Gain
ResNet-50	3.95	3
ResNet-18	2.01	2.34
MobileNet V2	4.23	4.9
InceptionV3	3,94	2

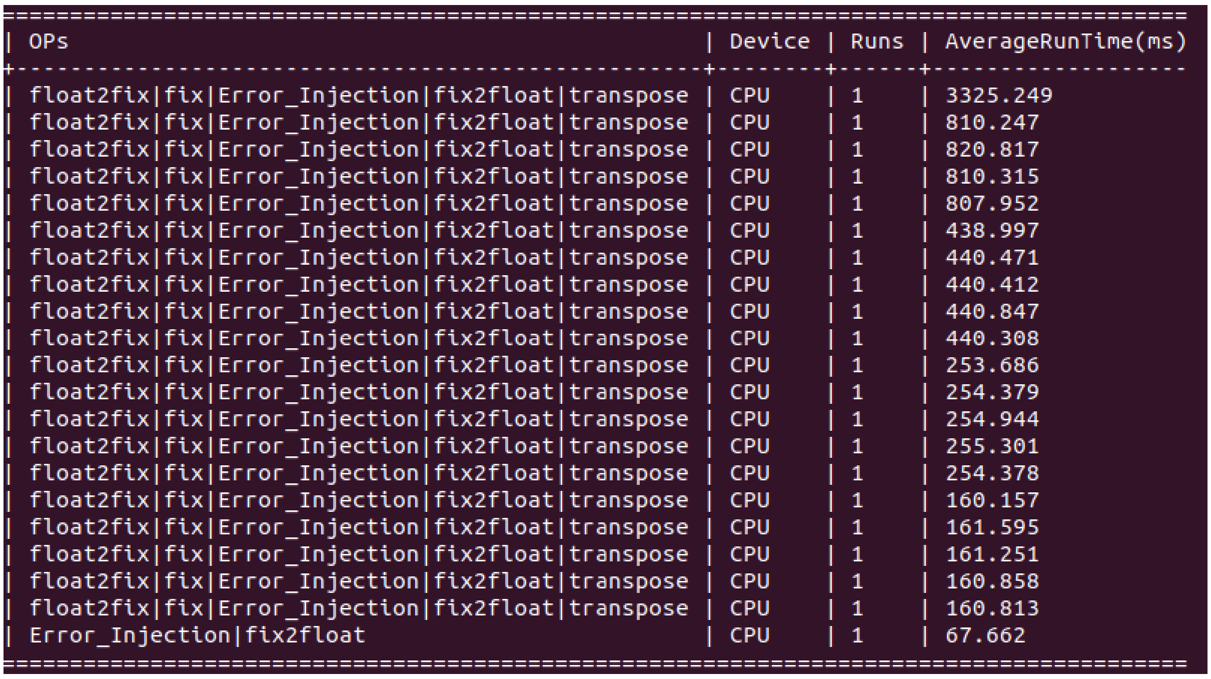
As shown in Tables 4.1 and 4.2, AMD Xilinx DPU is better in terms of inference speed. Since AMD Xilinx DPU have depth-wise convolution architecture, it shows better performance on MobileNet V2 then others while CPU shows less inference speed. While CPU shows similiar performance on ResNet-50 and InceptionV3, AMD Xilinx DPU shows better performance on ResNet-50 other than Inception V2. With these performance gains, there is a accuracy degredation becuse of quantization process. Also pre-procesing and post-processing effects inference accuracy. In Table 4.3 performance improvement and accuracy difference of DPU compared to CPU is shown. In this study, pre-processing and post-processings are realised using OpenCV C++ library (Itseez, 2015). Since this work accelerate Error Injection process with FPGA, accuracy degredations because of software manner is not concerned.

4.2 Error Injection with FPGA Performance

An Error Injection IP was designed to implement error injection. Since the incoming data to the Error Injection IP is in floating-point format after the CNN model is compiled to an XModel file, a quantizer module was designed in the FPGA to convert the 32-bit floating-point data to an 8-bit fixed-point integer using Algorithm 1 (8). While the incoming data is being quantized, errors for the incoming data are generated in parallel. Once the quantization process is completed, the error is ready to be injected into the data. After the Error Injection IP was implemented on the generated model, the overall performance was measured on the AMD Xilinx SoC.

4.2.1 Error Injection Performance

After Error Injection IP is implemented with DPU, overall performance is measured. With the ResNet-18 CNN model, the designed platform performance is measured with 0.1 frames per second (FPS). Vitis AI Profiler is used to identify the platform bottleneck. The output of the Vitis AI profiler is given in Figure 4.1.

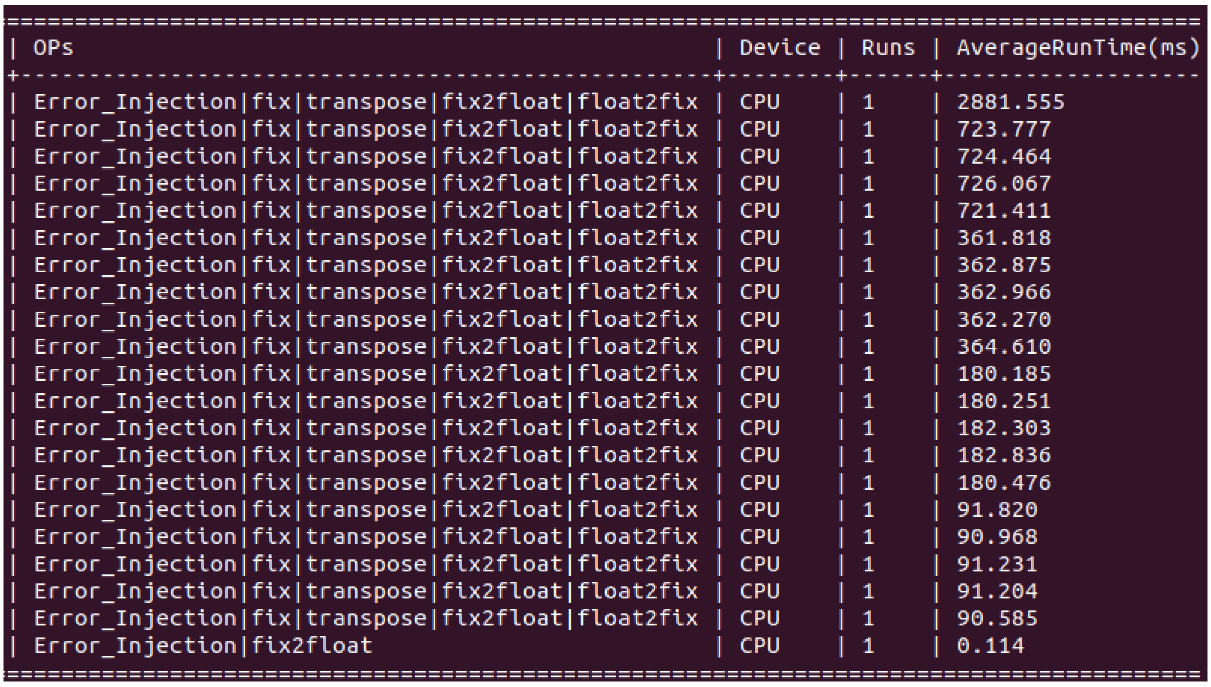


```
==========================================================================================
| OPs                                                 | Device | Runs | AverageRunTime(ms)
+-----------------------------------------------------+--------+------+-------------------
| float2fix|fix|Error_Injection|fix2float|transpose | CPU    | 1    | 3325.249
| float2fix|fix|Error_Injection|fix2float|transpose | CPU    | 1    | 810.247
| float2fix|fix|Error_Injection|fix2float|transpose | CPU    | 1    | 820.817
| float2fix|fix|Error_Injection|fix2float|transpose | CPU    | 1    | 810.315
| float2fix|fix|Error_Injection|fix2float|transpose | CPU    | 1    | 807.952
| float2fix|fix|Error_Injection|fix2float|transpose | CPU    | 1    | 438.997
| float2fix|fix|Error_Injection|fix2float|transpose | CPU    | 1    | 440.471
| float2fix|fix|Error_Injection|fix2float|transpose | CPU    | 1    | 440.412
| float2fix|fix|Error_Injection|fix2float|transpose | CPU    | 1    | 440.847
| float2fix|fix|Error_Injection|fix2float|transpose | CPU    | 1    | 440.308
| float2fix|fix|Error_Injection|fix2float|transpose | CPU    | 1    | 253.686
| float2fix|fix|Error_Injection|fix2float|transpose | CPU    | 1    | 254.379
| float2fix|fix|Error_Injection|fix2float|transpose | CPU    | 1    | 254.944
| float2fix|fix|Error_Injection|fix2float|transpose | CPU    | 1    | 255.301
| float2fix|fix|Error_Injection|fix2float|transpose | CPU    | 1    | 254.378
| float2fix|fix|Error_Injection|fix2float|transpose | CPU    | 1    | 160.157
| float2fix|fix|Error_Injection|fix2float|transpose | CPU    | 1    | 161.595
| float2fix|fix|Error_Injection|fix2float|transpose | CPU    | 1    | 161.251
| float2fix|fix|Error_Injection|fix2float|transpose | CPU    | 1    | 160.858
| float2fix|fix|Error_Injection|fix2float|transpose | CPU    | 1    | 160.813
| Error_Injection|fix2float                          | CPU    | 1    | 67.662
==========================================================================================
```

Figure 4.1 Vitis AI Profiler Output

In the custom operator flow of Vitis AI, additional layers are added before and after the custom operator. These layers include a fix2float layer that converts the DPU output from an 8-bit fixed-point format to a 32-bit floating-point format. Subsequently, a transpose layer is applied to take the matrix transpose of input data. Following this, the data undergoes quantization through a Fix layer. However, the output of the Fix layer is in a 32-bit floating-point format. The data is then passed to the custom operator. After the custom operator processing is completed, the specified layers are applied again in reverse order. Notably, instead of the initial fix2float layer, a float2fix layer is used. This layer converts the 32-bit floating-point data back into an 8-bit integer format. The data flow of these layers is shown in Figure 4.3.

To identify bottlenecks, the Error Injection IP in the error injection layer was removed. Instead, the incoming data was transmitted directly to the subsequent layer. With this modification, the model was re-evaluated using the Vitis AI Profiler. Following this, the output of the Vitis AI Profiler is as shown in Figure 4.2.



```
==================================================================================
| OPs                                             | Device | Runs | AverageRunTime(ms)
+-------------------------------------------------+--------+------+-------------------
| Error_Injection|fix|transpose|fix2float|float2fix | CPU    | 1    | 2881.555
| Error_Injection|fix|transpose|fix2float|float2fix | CPU    | 1    | 723.777
| Error_Injection|fix|transpose|fix2float|float2fix | CPU    | 1    | 724.464
| Error_Injection|fix|transpose|fix2float|float2fix | CPU    | 1    | 726.067
| Error_Injection|fix|transpose|fix2float|float2fix | CPU    | 1    | 721.411
| Error_Injection|fix|transpose|fix2float|float2fix | CPU    | 1    | 361.818
| Error_Injection|fix|transpose|fix2float|float2fix | CPU    | 1    | 362.875
| Error_Injection|fix|transpose|fix2float|float2fix | CPU    | 1    | 362.966
| Error_Injection|fix|transpose|fix2float|float2fix | CPU    | 1    | 362.270
| Error_Injection|fix|transpose|fix2float|float2fix | CPU    | 1    | 364.610
| Error_Injection|fix|transpose|fix2float|float2fix | CPU    | 1    | 180.185
| Error_Injection|fix|transpose|fix2float|float2fix | CPU    | 1    | 180.251
| Error_Injection|fix|transpose|fix2float|float2fix | CPU    | 1    | 182.303
| Error_Injection|fix|transpose|fix2float|float2fix | CPU    | 1    | 182.836
| Error_Injection|fix|transpose|fix2float|float2fix | CPU    | 1    | 180.476
| Error_Injection|fix|transpose|fix2float|float2fix | CPU    | 1    | 91.820
| Error_Injection|fix|transpose|fix2float|float2fix | CPU    | 1    | 90.968
| Error_Injection|fix|transpose|fix2float|float2fix | CPU    | 1    | 91.231
| Error_Injection|fix|transpose|fix2float|float2fix | CPU    | 1    | 91.204
| Error_Injection|fix|transpose|fix2float|float2fix | CPU    | 1    | 90.585
| Error_Injection|fix2float                         | CPU    | 1    | 0.114
==================================================================================
```

Figure 4.2 Vitis AI Profiler Output without Error Injection IP

As shown in Figure 4.1 and Figure 4.2, even after the removal of the Error Injection IP, the runtime decreases from 11 seconds to 9 seconds. In the 11-second version, both the additional layers and the Error Injection IP are executed. In contrast, the 9-second version executes only the additional layers. Based on this assessment, the additional layers added by Vitis AI account for 88 percent of the runtime.

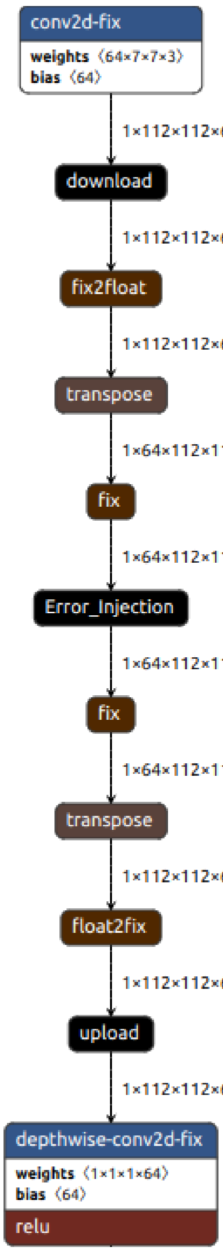


Figure 4.3 Vitis AI XModel Custom Operator portion

4.2.2 XModel and Error Injection IP Optimizations

As explained in Section 4.2.1, the layers added by Vitis AI for custom op flow, creates bottlenecks. Moreover, these layers are not required for error injection task. To address this issue, subgraphs were optimized using AMD Xilinx Intermediate Representaion (XIR) library. With this method, bottleneck layers were removed from the subgraphs using the XIR library. Additionally, optimizations of the Error Injection IP were implemented with the newly formed subgraphs.

4.2.2.1 Subgraph Optimization

The XIR library was utilized to remove the extra layers added by Vitis AI. This library allows for the manipulation of Operators (Ops) and TensorBuffers within the XModel. For XModel optimization, the Subgraphs within the XModel file are first read using the deserialize function. Subsequently, the subgraphs are ordered according to the data flow. The Ops in the Subgraphs assigned to the CPU are then arranged. Each CPU subgraph contains Fix2float and Float2fix layers, which facilitate data conversion between the CPU subgraph and the DPU subgraph.

For XModel optimization, Fix2float and Float2fix Ops are placed adjacently in an array. All subgraphs within the XModel are scanned to ensure these layers are stored in the array. Then, the subgraphs, including the DPU subgraphs, are scanned again. During this scan, the Float2fix Op corresponding to the input tensor name of each upload Op is identified. Each operator in the XModel has its own input and output TensorBuffers, each with a name, data type, and data size. The identified Float2fix Op and subsequent operators, except Fix2float, are removed. The output tensor of the Fix2float layer is then connected to the input of the upload operator. During this process, the data structure of the output tensor of the Fix2float operator is changed to an 8-bit fixed-point format from a 32-bit floating-point format. Finally, the code for all fix2float operators is modified. With the new code, the 8-bit fixed-point data from the DPU is transmitted to the Error Injection IP. Subsequently, the error-injected outputs are sent back to the DPU. The optimized version of the subgraph shown in Figure 4.3 is presented in Figure 4.5. All of the explained steps to optimize the computation graph are realized on SoC using the XIR C++ library. An overview of subgraph optimization is shown in Figure 4.4.

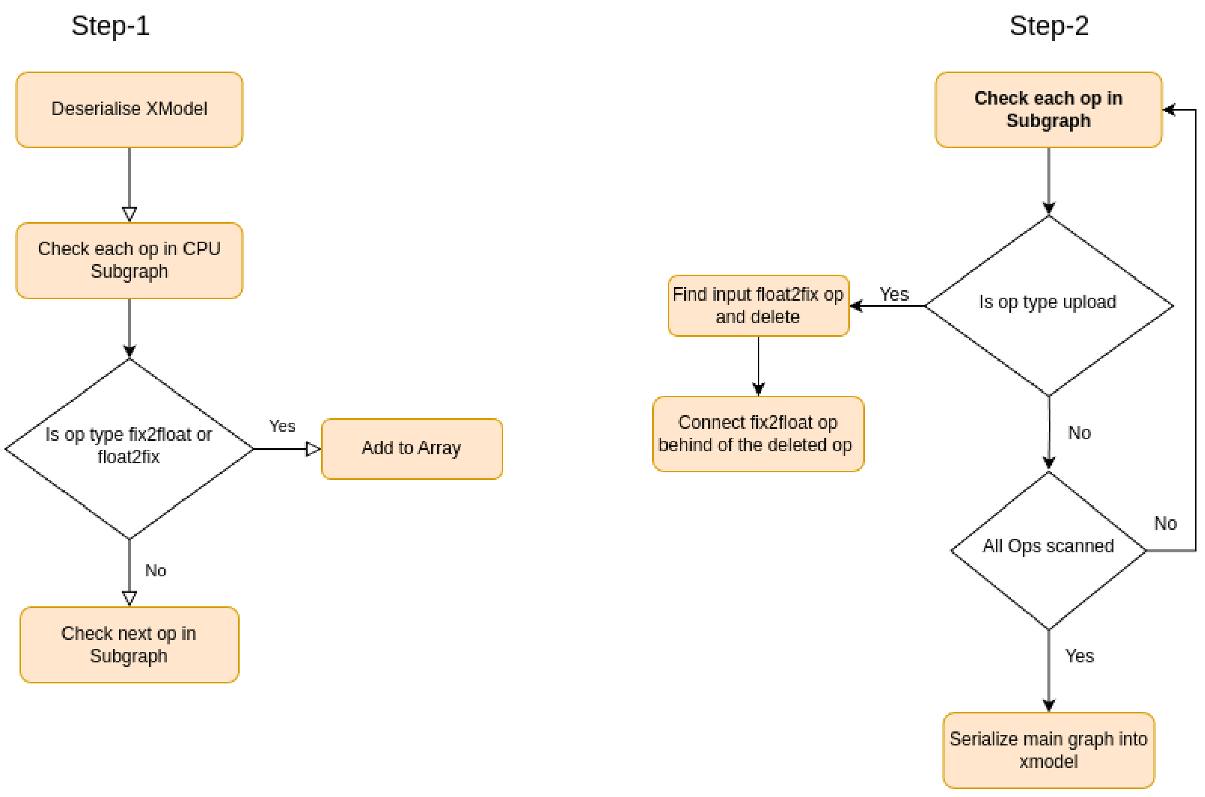

Step-1
Deserialise XModel
Check each op in CPU Subgraph
Is op type fix2float or float2fix
Yes
Add to Array
No
Check next op in Subgraph
Step-2
Check each op in Subgraph
Is op type upload
Yes
Find input float2fix op and delete
Connect fix2float op behind of the deleted op
No
All Ops scanned
No
Yes
Serialize main graph into xmodel

Figure 4.4 Subgraph Optimization Overview.

4.2.2.2 Error Injection IP Optimization

The modifications made to the XModel subgraph improve the Error Injection IP. In the new XModel file, the Error Injection subgraphs input data is in 8-bit fixed-point integer format. This change has enabled the elimination of the float2fix, quantization, and fix2float processes within the Error Injection IP. In the new Error Injection IP, errors are generated with the incoming data, and a direct gate-level XOR operation is applied. This change has eliminated the six clock cycles previously required for these processes. Since error generation now takes three clock cycles and data is processed in 8-bit fixed-point format instead of 32-bit floating-point format, the throughput of the Error Injection IP has increased eightfold. The data flow of the new Error Injection IP is shown in Figure 4.6. The new AXI-4 data transfer format is also shown in Figure 4.7.

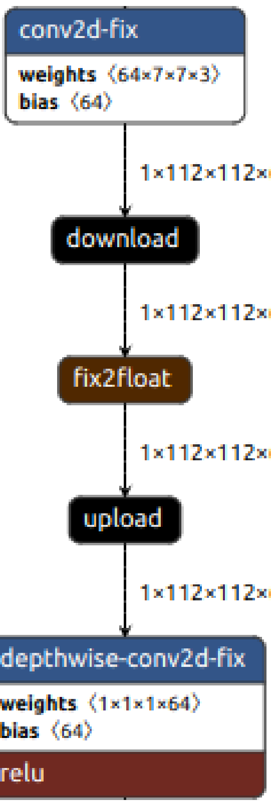


Figure 4.5 Optimized Subgraph.

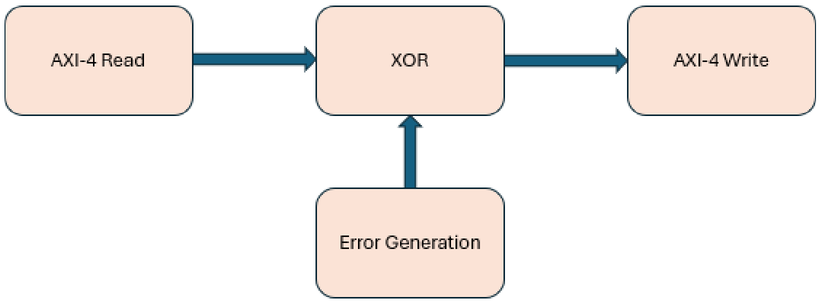


Figure 4.6 Optimized Error Injection IP

4.2.2.3 Optimization Results

After the optimizations, error injection was performed on both the SoC and the CPU, and the results were evaluated as follows: The error injection is executed at 4 FPS on the ResNet-18 model regarding the CPU. In contrast, the platform designed on the AMD Xilinx Ultrascale+ SoC lagged, achieving only 0.6 FPS. Although the optimizations significantly accelerated the error injection process, the desired speed has not yet been reached. The reason for this issue is that the Error Injection IP reads the activation data

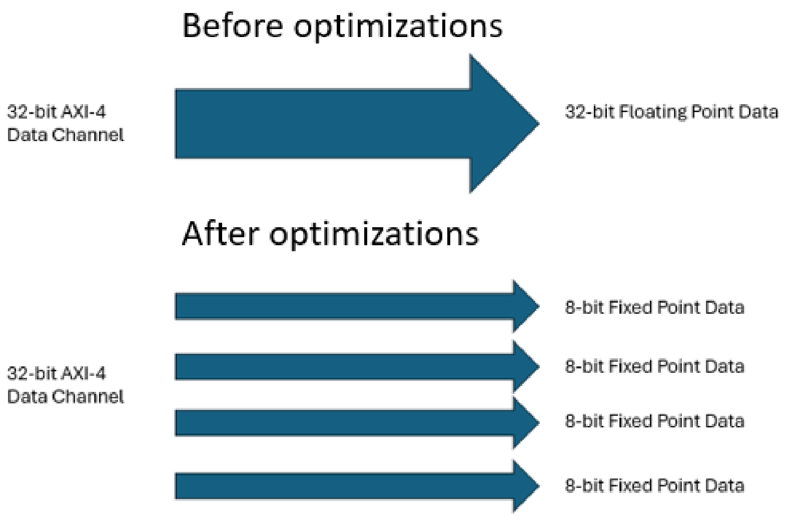
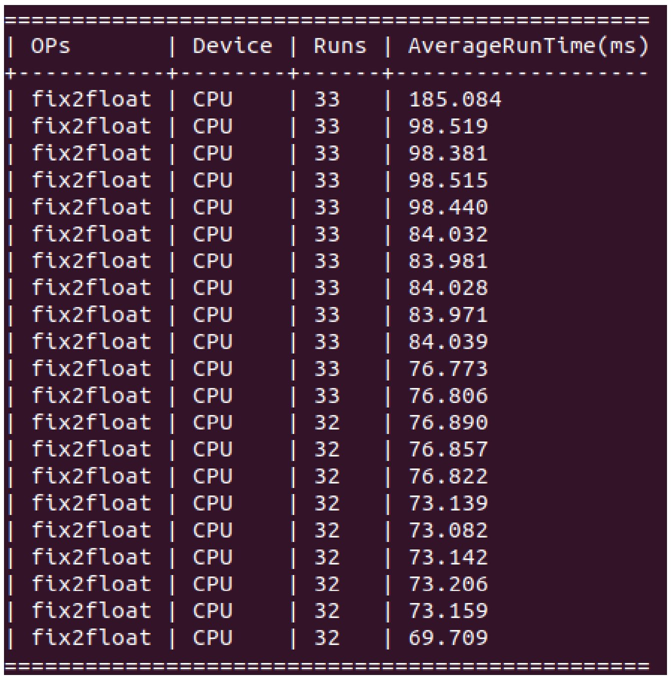


Figure 4.7 Before and After Single Data Transfer of Optimized Error Injection IP

from the CPU memory. Reading each activation data from the CPU memory results in reduced performance. Vitis AI Profiler analysis of the optimized error injection platform is given in Figure 4.8.



```
================================================
| OPs       | Device | Runs | AverageRunTime(ms)
+-----------+--------+------+-------------------
| fix2float | CPU    | 33   | 185.084
| fix2float | CPU    | 33   | 98.519
| fix2float | CPU    | 33   | 98.381
| fix2float | CPU    | 33   | 98.515
| fix2float | CPU    | 33   | 98.440
| fix2float | CPU    | 33   | 84.032
| fix2float | CPU    | 33   | 83.981
| fix2float | CPU    | 33   | 84.028
| fix2float | CPU    | 33   | 83.971
| fix2float | CPU    | 33   | 84.039
| fix2float | CPU    | 33   | 76.773
| fix2float | CPU    | 33   | 76.806
| fix2float | CPU    | 32   | 76.890
| fix2float | CPU    | 32   | 76.857
| fix2float | CPU    | 32   | 76.822
| fix2float | CPU    | 32   | 73.139
| fix2float | CPU    | 32   | 73.082
| fix2float | CPU    | 32   | 73.142
| fix2float | CPU    | 32   | 73.206
| fix2float | CPU    | 32   | 73.159
| fix2float | CPU    | 32   | 69.709
================================================
```

Figure 4.8 Vitis AI Profiler Output for optimized error injection

5. CONCLUSION AND FUTURE WORK

In this thesis, we present two distinct error injection platforms designed to induce bit-flips in the activation data of Convolutional Neural Networks (CNNs). The first platform implements error injection on a Central Processing Unit (CPU) using the PyTorch Deep Neural Network (DNN) framework. In this method, errors are defined as a two-dimensional matrix, allowing for detailed and precise error injection within defined CNN models.

The second platform utilizes the AMD Xilinx Ultrascale+ MPSoC device, with the primary objective of achieving more efficient error injection compared to the CPU, particularly for CNNs with numerous layers. In this platform, the Field-Programmable Gate Array (FPGA) component incorporates both the AMD Xilinx Deep Learning Processing Unit (DPU) IP and a custom Error Injection IP. These components are responsible for executing CNN layer computations and performing error injection on the activation data. To implement error injection, a novel Error Generation architecture is proposed, which is capable of generating a specified number of errors per bit.

Since error injection is a custom operation not natively supported by the Vitis AI framework, it is necessary to define a custom operator for the error injection layer to correctly process the activation data. In Chapter 4, the performance of error injection using the Vitis AI custom operator is evaluated for a pre-trained ResNet-18 model. When the error injection layer is defined as a custom layer, Vitis AI introduces additional layers before and after the error injection layer, which incurs significant overhead due to their inefficient processing of activation data via for loops.

Graph optimizations were applied to both the Xmodel file and the Error Injection IP to mitigate this inefficiency. Subsequently, the new performance gains were demonstrated, and a comparative analysis of FPGA and CPU performance differences in CNN inference and error injection was conducted.

Figure 5.1 outlines the steps for implementing Convolutional Neural Networks (CNNs) on an FPGA using AMD Xilinx DPU IP. This process encapsulates the AMD Xilinx DPU Vitis flow.

In the initial phase of the Vitis AI implementation, the procedure for creating a custom platform tailored to the target board is comprehensively outlined in AMD's documentation (AMD, 2021). This process is essential for compiling Vitis projects specifically for the AMD Xilinx Ultrascale+ MPSoC. It is imperative to ensure that the version of the AMD Xilinx DPU is compatible with the corresponding versions of Vitis and Vitis AI (AMD, 2024c). Additionally, the PetaLinux version within the Sysroot file must align with the version of AMD Xilinx Vitis being utilized.

In the subsequent step, it is necessary to define the reference clock frequency for the AXI-4 interface and the operating frequency for the DPU DSP. These specifications should be based on the number of DPU cores selected in an earlier step. The DSP's operating frequency must be twice that of the AXI-4 interface. Furthermore, at this stage, an AXI-4 master interface port should be designated for each DPU core.

When targeting the AMD Xilinx Ultrascale+ Kria System on Module (SoM) in the sixth step, it is not required to repeatedly burn a new SD card image after each compilation. The Kria SoM includes a library known as 'xmutil,' which facilitates dynamic modifications to the FPGA bitstream while the FPGA remains powered on, using the FPGA binary file compiled through the Vitis project. For more detailed information, Knitter (Knitter, 2023) can be followed.

In contrast to Vitis and Vivado, Vitis AI operates within a Docker environment, necessitating basic knowledge of Docker for its installation and use. For detailed installation instructions, refer to AMD's documentation (AMD, 2024a).

Vitis AI allows for detailed control over the quantization process, enabling fine-tuning through various parameters (AMD Documentation, 2024). The accuracy of the quantized model can be evaluated, with adjustable parameters such as data type, bit width, and method. However, it is important to note that the AMD Xilinx DPU IP does not support all models quantized using Vitis AI. For instance, a CNN model intended for the AMD Xilinx DPU IP must be quantized to 8-bit precision, and the data type parameter must be set to 'int'. By default, the quantization parameters are configured to be compatible with the AMD Xilinx DPU IP. During quantization, the batch size is also specified as a parameter; however, the DPU IP implemented on the AMD Xilinx Ultrascale+ MPSoC supports only a batch size of 1. Consequently, models quantized with a batch size greater than 1 cannot be compiled.

Pre-processing and post-processing tasks for the CNN model should be performed on the processor side. Typically, post-processing involves converting the output from an 8-bit integer format to a 32-bit floating-point format and performing the softmax calculation.

For additional guidance on the procedures related to Vitis AI, the relevant AMD documentation (AMD, 2024b) should be followed.

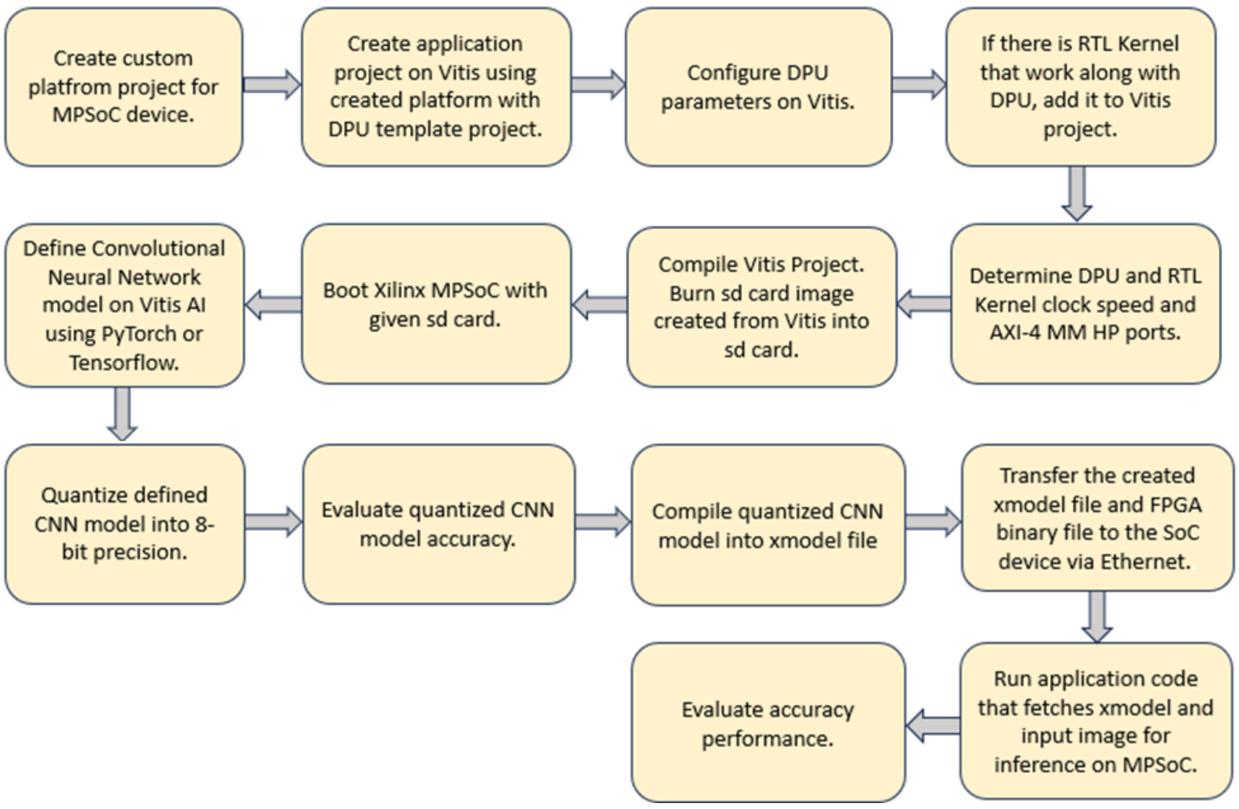

Create custom platfrom project for MPSoC device.
Create application project on Vitis using created platform with DPU template project.
Configure DPU parameters on Vitis.
If there is RTL Kernel that work along with DPU, add it to Vitis project.
Determine DPU and RTL Kernel clock speed and AXI-4 MM HP ports.
Compile Vitis Project. Burn sd card image created from Vitis into sd card.
Boot Xilinx MPSoC with given sd card.
Define Convolutional Neural Network model on Vitis AI using PyTorch or Tensorflow.
Quantize defined CNN model into 8-bit precision.
Evaluate quantized CNN model accuracy.
Compile quantized CNN model into xmodel file
Transfer the created xmodel file and FPGA binary file to the SoC device via Ethernet.
Run application code that fetches xmodel and input image for inference on MPSoC.
Evaluate accuracy performance.

Figure 5.1 Steps to realize Convolutional Neural Networks with AMD Xilinx DPU

5.1 Future Work

In this thesis, an Error Injection IP solution was designed to realize error injection processes with high throughput alongside the AMD Xilinx DPU IP. However, the AMD Xilinx DPU IP and the Error Injection IP operate independently. Specifically, the output of the AMD Xilinx DPU IP is written to CPU memory, and afterward, the Error

Injection IP reads this output using the AXI-4 interface to perform the error injection process. The results demonstrated in this thesis indicate that this method is not sufficiently efficient. It is necessary to process the data on the FPGA and perform the error injection there to achieve high performance from PyTorch's software running on the processor. A future direction for this thesis is to enable the Error Injection IP to work in tandem with Deep Learning Compute Engines, such as the AMD Xilinx DPU, and to compare its performance with the software running on the processor.

BIBLIOGRAPHY

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

AMD (2020). *Zynq DPU Product Guide (PG338)*. Accessed: 2024-07-16.

AMD (2021). *Vitis Platform Creation Tutorial*. AMD. Accessed: 2024-08-15.

AMD (2023). Vitis ai tutorials: Pytorch subgraphs. Accessed: 2024-07-18.

AMD (2024). Introduction on graphrunner with vitis ai 1.4. Accessed: 2024-07-16.

AMD (2024a). Vitis ai: A comprehensive ai inference development platform on xilinx devices. https://github.com/Xilinx/Vitis-AI. Accessed: 2024-08-23.

AMD (2024b). Vitis ai quick start tutorial for mpsoc. https://xilinx.github.io/Vitis-AI/3.0/html/docs/quickstart/mpsoc.html#pytorch-tutorial. Accessed: 2024-08-15.

AMD (2024c). Vitis ai version compatibility. Accessed: 2024-08-23.

AMD (2024d). Vitis xrt. https://www.xilinx.com/products/design-tools/vitis/xrt.html#overview. Accessed: 2024-08-17.

AMD (2024-04-25). Vitis ai. https://www.xilinx.com/products/design-tools/vitis/vitis-ai.html. Accessed on Date.

AMD Documentation (2024). *Quantization Strategy Configuration*. Accessed: 2024-08-23.

ARM Ltd. (2022). *AMBA® AXI and ACE Protocol Specification*. Version 2022.

Bae, K., Moon, S., Choi, D., Choi, Y., Choi, D.-s., and Ha, J. (2011). Differential fault analysis on aes by round reduction. In *2011 6th International Conference on Computer Sciences and Convergence Information Technology (ICCIT)*, pages 607–612.

Barenghi, A., Breveglieri, L., Koren, I., and Naccache, D. (2012). Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures. *Proceedings of the IEEE*, 100(11):3056–3076.

Bhowmik, P., Hossain Pantho, J., Mandebi Mbongue, J., and Bobda, C. (2021). Esca: Event-based split-cnn architecture with data-level parallelism on ultrascale+ fpga. In

2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), pages 176–180.

Blömer, J. and Seifert, J.-P. (2003). Fault based cryptanalysis of the advanced encryption standard (aes). In Wright, R. N., editor, *Financial Cryptography*, pages 162–181, Berlin, Heidelberg. Springer Berlin Heidelberg.

Chen, C.-Y., Choi, J., Gopalakrishnan, K., Srinivasan, V., and Venkataramani, S. (2018). Exploiting approximate computing for deep learning acceleration. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 821–826.

Choi, W., Shin, D., Park, J., and Ghosh, S. (2019). Sensitivity based error resilient techniques for energy efficient deep neural network accelerators. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6.

Courbariaux, M., Hubara, I., Soudry, D., El-Yaniv, R., and Bengio, Y. (2016). Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830*.

Guo, K., Sui, L., Qiu, J., Yu, J., Wang, J., Yao, S., Han, S., Wang, Y., and Yang, H. (2018). Angel-eye: A complete design flow for mapping cnn onto embedded fpga. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(1):35–47.

He, K., Zhang, X., Ren, S., and Sun, J. (2015). Deep residual learning for image recognition. *CoRR*, abs/1512.03385.

He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778.

He, Y., Hutton, M., Chan, S., De Gruijl, R., Govindaraju, R., Patil, N., and Li, Y. (2023). Understanding and mitigating hardware failures in deep learning training systems. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA. Association for Computing Machinery.

Itseez (2015). Open source computer vision library. https://github.com/itseez/opencv.

Knitter, W. (2023). Vitis acceleration flow on kv260 vitis platform. https://www.hackster.io/whitney-knitter/vitis-acceleration-flow-on-kv260-vitis-platform-c3537e. Accessed: 2024-08-23.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In Pereira, F., Burges, C., Bottou, L., and Weinberger, K., editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc.

LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *Nature*, 521(7553):436–444.

LeCun, Y., Boser, B. E., Denker, J. S., Henderson, D., Howard, R., Hubbard, W., and Jackel, L. D. (1989). Backpropagation applied to handwritten zip code recognition. 1(4).

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

Liang, Y., Rupnow, K., Li, Y., Min, D., Do, M. N., and Chen, D. (2012). High-level synthesis: productivity, performance, and software constraints. *JECE*, 2012.

Louie, A. T. (2021). *Exploring Zynq MPSoC: With PYNQ and Machine Learning Applications*. Independently published.

Mahmoud, A., Aggarwal, N., Nobbe, A., Vicarte, J. R. S., Adve, S. V., Fletcher, C. W., Frosio, I., and Hari, S. K. S. (2020). Pytorchfi: A runtime perturbation tool for dnns. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 25–31.

Muller, U., Ben, J., Cosatto, E., Flepp, B., and Cun, Y. (2005). Off-road obstacle avoidance through end-to-end learning. In Weiss, Y., Schölkopf, B., and Platt, J., editors, *Advances in Neural Information Processing Systems*, volume 18. MIT Press.

Narayanan, N., Chen, Z., Fang, B., Li, G., Pattabiraman, K., and DeBardeleben, N. (2023). Fault injection for tensorflow applications. *IEEE Transactions on Dependable and Secure Computing*, 20(4):2677–2695.

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. (2017). Automatic differentiation in pytorch.

Qiu, J., Wang, J., Yao, S., Guo, K., Li, B., Zhou, E., Yu, J., Tang, T., Xu, N., Song, S., Wang, Y., and Yang, H. (2016). Going deeper with embedded fpga platform for convolutional neural network. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '16, page 26–35, New York, NY, USA. Association for Computing Machinery.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. (2019). Language models are unsupervised multitask learners.

Rathore, M. (2021). *Exploring the Accuracy vs Energy Efficiency Trade-offs in Error-Aware Low Voltage DNN Accelerators*. Phd dissertation, Stony Brook University, Stony Brook, New York.

Reagen, B., Gupta, U., Pentecost, L., Whatmough, P., Lee, S. K., Mulholland, N., Brooks, D., and Wei, G.-Y. (2018). Ares: A framework for quantifying the resilience of deep neural networks. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6.

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., and Fei-Fei, L. (2015). ImageNet Large

Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252.

Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., and Chen, L.-C. (2018a). Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520.

Sandler, M., Howard, A. G., Zhu, M., Zhmoginov, A., and Chen, L. (2018b). Inverted residuals and linear bottlenecks: Mobile networks for classification, detection and segmentation. *CoRR*, abs/1801.04381.

Schmidhuber, J. (2014). Deep learning in neural networks: An overview. *CoRR*, abs/1404.7828.

Shawahna, A., Sait, S. M., and El-Maleh, A. (2019). Fpga-based accelerators of deep learning networks for learning and classification: A review. *IEEE Access*, 7:7823–7859.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958.

Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., and Wojna, Z. (2015). Rethinking the inception architecture for computer vision. *CoRR*, abs/1512.00567.

Tan, M. and Le, Q. V. (2019). Efficientnet: Rethinking model scaling for convolutional neural networks. *CoRR*, abs/1905.11946.

Tian, Y., Pei, K., Jana, S., and Ray, B. (2017). Deeptest: Automated testing of deep-neural-network-driven autonomous cars. *CoRR*, abs/1708.08559.

VMware (2023). Exploring gpu architecture. Accessed: 2024-07-21.

Wang, Y., Sun, T., Li, S., Yuan, X., Ni, W., Hossain, E., and Poor, H. V. (2023). Adversarial attacks and defenses in machine learning-powered networks: A contemporary survey.

Weyand, T., Kostrikov, I., and Philbin, J. (2016). Planet - photo geolocation with convolutional neural networks. *CoRR*, abs/1602.05314.

Yazdanbakhsh, A., Park, J., Sharma, H., Lotfi-Kamran, P., and Esmaeilzadeh, H. (2015). Neural acceleration for gpu throughput processors. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, page 482–493, New York, NY, USA. Association for Computing Machinery.

6. APPENDIX

https://github.com/ubercelik/DNN-Fault-Injection