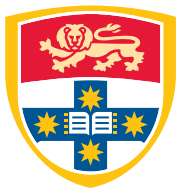
Low Latency and Scalable Machine Learning on FPGA-based System-on-Chip

BINGLEI LOU

Doctor of Philosophy (PhD)



THE UNIVERSITY OF SYDNEY

Supervisor: Prof. Philip H.W. Leong
Associate Supervisor: Dr. David Boland

A thesis submitted in fulfilment of
the requirements for the degree of
Doctor of Philosophy

School of Electrical and Computer Engineering
Faculty of Engineering
The University of Sydney
Australia

4 December 2024

Declaration

I, *Mr. Binglei Lou*, declare that this thesis is submitted to fulfill the requirements for the conferral of the degree *Doctor of Philosophy (PhD)*, from the University of Sydney, is wholly my own work unless otherwise referenced or acknowledged. This document has not been submitted for qualifications at any other academic institution.

Abstract

Machine learning (ML) has proven highly effective in analyzing data, making decisions, and solving complex problems, particularly when deriving explicit mathematical models is difficult. Field-Programmable Gate Arrays (FPGAs) offer a powerful hardware platform for executing ML tasks at the edge, where rapid decision-making and responsiveness are crucial. However, deploying ML methods on customized FPGA hardware for real-time applications involves navigating the trade-offs between latency, accuracy, and flexibility, all while managing the constraints of on-chip resources and memory. The research is centered on three primary objectives: (1) enhancing the accuracy of ML accelerators in an area-efficient manner, (2) integrating these accelerators into a unified system-on-chip (SoC) architecture, and (3) designing adaptable partial blocks within the SoC to respond to varying environmental conditions.

At the circuit level, this thesis introduces LUTEnsemble, a specialized LUT-based architecture optimized for rapid and accurate deep neural network (DNN) inference on FPGAs. This architecture addresses the significant scaling challenges inherent in traditional LUT-based DNN approaches, where LUT resource requirements increase exponentially with the number of inputs. By integrating multiple PolyLUT sub-neurons with adder tree structures and forming an ensemble of sparsely connected sub-nets, LUTEnsemble enhances both neuron connectivity and scalability. In comparison to state-of-the-art solutions, LUTEnsemble establishes new benchmarks for accuracy, latency, and resource efficiency in LUT-based DNN inference on FPGAs.

The system-level discussion centers on the application of FPGA-based neural networks for qubit state measurements in trapped-ion quantum information processing. This thesis examines how the LUTEnsemble and Vision Transformer (ViT) architectures effectively tackle the challenges of achieving both low latency and high accuracy in this context. Significant reductions in system latency are achieved through optimized buffering and interface design

between the electron-multiplying charge-coupled device (EMCCD) camera and the FPGA. The resulting total detection latency for one- and three-qubit tests on the FPGA demonstrated a reduction of 119× and 94×, respectively, compared to the GPU baseline system.

Finally, this thesis explores strategies to enhance the composability of FPGA-based SoCs. A flexible computing framework for FPGA streaming ensemble anomaly detection (fSEAD) tasks is proposed. This framework utilizes multiple partially reconfigurable blocks, known as pblocks, interconnected via an AXI switch. This architecture supports the dynamic composition of these blocks at runtime, enabling the merging and combining of results to form an ensemble. Experimental validation on the PYNQ platform, comparing fSEAD (with ensembles of three sub-detectors: Loda, RS-Hash, and xStream) to an equivalent CPU implementation across four public datasets, achieved speed-ups ranging from 3× to 8×.

This work contributes to the field by advancing FPGA-based ML design across circuit, system, and tool levels, offering novel solutions for real-time, resource-constrained, and reconfigurable applications on edge hardware.

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my main supervisor, Professor Philip Leong. Philip's extensive expertise, connections, and keen research insights in the field of FPGA have been invaluable to my PhD journey. He consistently provided me with intriguing research ideas to explore, even though I only managed to complete a fraction of the many ideas we discussed. Before starting my PhD, I did not have many expectations for receiving sufficient supervision from a busy professor, but Philip always found time in his tight schedule to guide me through the basics of writing, experiments, coding, and presentations. He also offered me a great platform for collaboration, including working with Professor Wayne Luk and Professor George A. Constantinides's team at Imperial College London. I am also grateful to the School of Physics at the University of Sydney for their support throughout my research.

I would also like to sincerely thank my secondary supervisor, David Boland. I have always considered David to be a super smart person. He has a remarkable ability to pinpoint the core issues in my research and always finds the weakest link just when I think everything is settled, helping me to refine and improve my work. On top of that, David is not only a great researcher but also a genuinely kind and supportive person in everyday life.

My gratitude also goes to our lab's postdoc, Richard Rademacher. Despite being younger than him by more than a decade, I am deeply impressed by his enthusiasm and energy for both life and research. Richard not only provided me with practical and effective support, but also brought joy and laughter to the group like an older brother.

I am grateful for the collaborative efforts involving PhD candidate Filip Wojcicki, Dr. Zhiqiang Que, Professor Wayne Luk from the Department of Computing at Imperial College London, and Professor George A. Constantinides from the Department of Electrical and Electronic Engineering at Imperial College London.

I would like to thank my research fellows: Gautham Duddi Krishnaswaroop, Ruilin Wu, Haoyan Qi, Chuliang Guo, Wenjie Zhou, Wenting Xu, Xueyuan Liu, and Carol Li. Every deadline I faced was made more manageable with their companionship and support.

I want to thank my friends for their companionship throughout my time in Sydney. Although I met many people, true friendships have been rare and precious. The joy of traveling, playing basketball, sharing drinks, and having heartfelt conversations with you all has been unforgettable. Although some of you had already left Sydney by the time I wrote this thesis, I sincerely wish you all the best in the future.

I gratefully acknowledge the financial support from the China Scholarship Council, which provided funding for my PhD studies. I am also honored to have received the Faculty of Engineering Career Advancement Award from the University of Sydney in 2024. Furthermore, I deeply appreciate the support from the EPSRC Spatial Computational Learning grant, which not only enabled my research exchange at Imperial College London but also funded my travel to Italy to attend the FPL2024 conference.

Finally, I sincerely thank my family for their constant support and understanding throughout my studies. Despite the distance, they have always cared deeply, with each phone call filled with encouragement, reminding me to take care of myself and eat well.

Authorship Attribution Statement

This thesis contains several chapters with material that was previously published during my PhD candidature under the supervision of Prof. Philip Leong and Dr. David Boland. The ideas and preparation behind each publication are primarily my own work under the oversight of my supervisors, with all assistance that I received, stated and acknowledged below:

- Chapter 3 of this thesis expands on two related works: PolyLUT-Add [1] (published in FPL2024) and LUTEnsemble. This chapter presents comprehensive approaches to enhancing LUT-based FPGA neural networks. The experiment and writing were assisted by discussions with Philip Leong, David Boland, and Richard Rademacher.
- Chapter 4. The idea of applying a Vision Transformer to the qubit detection system was proposed by Philip Leong. The design of the Vision Transformer hardware was a collaborative effort involving PhD candidate Filip Wojcicki, Dr. Zhiqiang Que, and Professor Wayne Luk from the Department of Computing at Imperial College London. The experiments were assisted by Gautham Duddi Krishnaswaroop, Ruilin Wu, and Richard Rademacher.
- Chapter 5 of this thesis is published as fSEAD [2] (TRETS). The experiment and writing were assisted by discussions with Philip Leong and David Boland.

Contents

Declaration ii

Abstract iii

Acknowledgements v

Authorship Attribution Statement vii

Contents viii

List of Figures xi

Chapter 1 Introduction 1

1.1 Motivation and Aims 1

1.2 Contributions 3

1.3 Thesis Structure 5

Chapter 2 Background 6

2.1 Machine Learning 7

2.1.1 Neural Network 7

2.1.2 Anomaly Detection 14

2.1.3 Ensemble Techniques 15

2.2 FPGA Design Workflows 18

2.2.1 Special Case: Dynamic Partial Reconfiguration 20

2.3 FPGA aided Machine Learning on Quantum Computers 22

2.3.1 Trapped-Ion 22

2.3.2 Qubit Measurement 23

2.3.3 Machine Learning aided Solutions 25

2.3.4 Summary 26

Chapter 3 Fast and Accurate Look-up Table based Neural Networks 28
3.1 Related Work 31
3.2 Design 33
3.2.1 Addtree Architecture 33
3.2.2 Ensemble 34
3.2.3 Mixer Feature Mask Layer 34
3.2.4 System Toolflow 38
3.3 Results 38
3.3.1 Datasets 38
3.3.2 Experimental Setup 40
3.3.3 Evaluation of PolyLUT-Add 41
3.3.4 Evaluation of LUTEnsemble 45
3.4 Summary 51

Chapter 4 Low-latency Neural Network Qubit Detection System in Trapped-Ion Quantum Computing 52
4.1 Design 57
4.1.1 FPGA-based Control System 57
4.1.2 Data Flow and Timing Measurements 58
4.1.3 Vision Transformer Accelerator 61
4.1.4 Implementation in Vivado HLS 61
4.2 Experimental Setup 63
4.2.1 Trap ion and EMCCD 63
4.2.2 Latency Measurement Setup 64
4.2.3 Image dataset 67
4.2.4 ViT and LUTEnsemble Model configuration 68
4.3 Results 69
4.3.1 Accuracy Comparison 69
4.3.2 Latency Comparison 70
4.4 Summary 74

Chapter 5 fSEAD: Composable Hardware on FPGA SoC **76**
5.1 Related Work 79
5.1.1 Streaming Anomaly Detection 79
5.1.2 FPGA-based Anomaly Detection Solutions 84
5.2 Design 86
5.2.1 Module Generator 87
5.2.2 DFX Tool Flow 90
5.2.3 Composable Infrastructure 91
5.2.4 FPGA Implementation 94
5.3 Results 96
5.3.1 Test Platform 96
5.3.2 Sub-detectors and Ensemble Accuracy 97
5.3.3 Pblock Assignment and FPGA Implementation 99
5.3.4 Speed, Accuracy and Power Comparison 101
5.3.5 Partial Reconfiguration 107
5.4 Summary 109

Chapter 6 Conclusion **110**
6.1 Future outlook 111

Bibliography **113**

List of Figures

1.1 (a) A trade-off between different hardware architectures: CPU, GPU, FPGA, and ASIC. (b) Data are from Market Research Report (Ref. [11]) 2

1.2 Ensemble Technique 2

1.3 Thesis structure. 3

2.1 The block diagram of MLP. 8

2.2 Block diagram of ViT. 11

2.3 Block diagram of multi-headed self-attention in (a) and self-attention in (b). 13

2.4 Anomaly Detection Case-1. 14

2.5 Anomaly Detection Case-2. 14

2.6 Diagram of the `deep ensemble` neural network. This example improves diversity using E independent feedforward passes of the same input. The final prediction is estimated by combining E predictions on the output. 16

2.7 Diagram of `MIMO` ensemble neural network. E is set to 2 in this case; diversity among this ensemble is guaranteed from 2 compact sparse sub-models inside the single feedforward pass. 17

2.8 Sample Architecture for SEAD Methods. 18

2.9 Basic Premise of Partial Reconfiguration. 21

2.10 Example of FPGA-aided Qubit Detection in Tapped-Ion. The photons scattered by the ions are collected and magnified by optical elements and directed toward an EMCCD camera. These images are then transmitted to an FPGA via the Cameralink protocol [86]. The FPGA processes the images in real time, and an embedded classifier determines the qubit states. 22

2.11 Electronic level structure of $^{171}Yb^{+}$. The 369.5 nm transition is used for Doppler cooling to near the motional ground state and for qubit measurement. The state is measured by detecting photons from scattering from the separate lasers on this

wavelength. The $^2S_{1/2}$ hyperfine splitting (12.6 GHz) defines two measurable qubit states: bright and dark. Quantum gate operations are performed using an off-resonant 355 nm optical transition (not shown). 23

2.12 Detection of photons emitted by ions with an EMCCD camera. Fluorescence at 369.5 nm from a string of ions is collected using an objective lens and focused onto an EMCCD's active sensor area. Frame Transfer allows the image to be transferred to a storage area before being amplified and converted to a digital signal through an ADC. 25

2.13 Histogram of thresholding method. In the training process, We process histograms for the dark (shown in blue) and bright images (shown in orange), and determine an optimal threshold that maximizes the fidelity. Afterward, a measurement is processed by calculating the total number of photon counts in the image and comparing it to the pre-calculated threshold, and it's unavoidable that some samples will be misclassified. 25

2.14 An example of simulated 2-ion crosstalk with 2-D resolution. The integrated pixel brightness along the y-direction (black dots) is fitted with a Gaussian function (red line). One can see an overlap between the signals, demonstrating visible cross-talk between the two ions. 26

3.1 An example of LUT-based DNN on an image classification task. 28

3.2 Comparison between PolyLUT and PolyLUT-Add. (a) For AF inputs, PolyLUT requires $\mathcal{O}(2^{A\beta F})$ LUTs. (b) PolyLUT-Add uses A sub-neurons each with F inputs requiring $\mathcal{O}(A \times 2^{\beta F})$ LUTs and the adder box requires a lookup table with $\mathcal{O}(2^{A(\beta+1)})$ LUTs. Only $A \in \{2, 3\}$ are supported since larger values ($A \geq 4$) would lead the adder box to be an area and latency bottleneck. 30

3.3 Illustration of the LUT-based DNN inference scheme used in LogicNets, PolyLUT, and NeuraLUT. 32

3.4 A single adder tree incorporating A PolyLUT sub-neurons. 33

3.5 The dashed boxes represent pipeline registers. 34

3.6 Architecture of LUTEnsemble with E sub-nets employing random connectivity patterns. 35

3.7 Block diagram of a Mixer Layer. As illustrated, for an output y_i (y_1 and y_{19} are shown in orange), it takes $M = 4$ inputs around x_i (shown in orange and gray), consistent with the structure depicted in Figure 3.6 and includes a random feature mask (in blue) and trainable weighted sum computation (in green) for training. Following training, the Mixer layer is absorbed into LUTEnsemble nodes. 36

3.8 Tool flow of LUTEnsemble. It is inherited from the open-source PolyLUT toolflow [14]. We show the modified elements in red. 39

3.9 Accuracy results on different models. We use Deep($\mathbb{D}$), Wide($\mathbb{W}$) and Add(A) to denote "PolyLUT-Deeper", "PolyLUT-Wider" and "PolyLUT-Add" respectively. 42

3.10 Results showing the trade-off between test error rate and the sum of lookup table entries for $D = 2$. The lookup table entries on the x-axis are the relative size compared to PolyLUT (HDR model in Table 3.2). The black dashed lines depict the Pareto frontier. 46

3.11 Results showing the trade-off between uncertainty (NLL) and the sum of lookup table entries for $D = 2$. The lookup table entries on the x-axis are the relative size compared to PolyLUT (HDR model of Table 3.2). The black dashed lines depict the Pareto frontier. 46

3.12 Density plot of entropy values of networks trained on MNIST and tested on MNIST (as the known dataset) and NotMNIST (as the out-of-distribution dataset). 48

4.1 In the block diagram of ML-aided qubit detection in a trapped ion system, photons scattered by the ions are collected and magnified by optical elements before being directed toward an EMCCD camera. The resulting images are transmitted to the acceleration hardware, such as an FPGA or GPU, via the Cameralink protocol, with the GPU serving as the baseline. The right bar chart illustrates the speedup of the ViT model achieved on a 3-qubit test. The detailed implementation for FPGA and GPU are introduced in Figure 4.2 and Figure 4.10 respectively. 54

4.2 This figure demonstrates the framework of our qubit detection system. The camera is controlled by an ARTIQ system [130], with a master Kasli controller [131] that communicates with sub-modules, such as a DIO card, to trigger the camera and receive qubit detection results from the Sundance FPGA. Kasli also carries a

traditional qubit classifier based on thresholding [132] inside it, which will serve as a baseline for traditional models in the subsequent experimental section for comparison with our proposed ML scheme, which is implemented in the Sundance FPGA. By switching the Selector to connect the camera to Sundance, the ion image is transferred from the camera to FPGA through cameralink protocol, followed by the cameralink deserializer to decode the data packet to the ViT model for real-time classification. The FPGA is controlled by a PCIe driver designed in Linux PC, and a Xilinx Vivado [108] is used to load bitstream to the QSPI Flash on the FPGA board. 56

4.3 This figure describes the flow chart of the proposed FPGA-based qubit detection system. The steps with attached $*$ symbolize that probe signals are available in this step for latency measurements. The detailed timing diagram of `FVAL`, `LVAL`, `tx_done` and `vit_valid`, `vit_data` are illustraed in Figure 4.4. The `FIFO` structure is discussed in Figure 4.5 58

4.4 Image readout and detection signal timing. We refer the reader to Ref. [86] for a specialized description of the CameraLink Protocol. Here, our focus is mainly on the relationships between `FVAL`, `tx_done`, and the `vit` output, with other signal details being omitted for simplicity. 59

4.5 FIFO of Cameralink Deserializer. It serves as a buffer to transfer data from the cameralink interface domain (works in the `pixel_clk`) to the AXIS bus domain works in a different clock source: `m_axis_aclk`). The `wr_en` and `rd_en` are 'write enable' and 'read enable' signals, respectively, with full and empty indicating the FIFO's capacity status. 59

4.6 Hardware optimization for each component inside of the ViT 60

4.7 Resource-efficient matrix multiplication. The operation in the example is $\mathbf{Y} = \mathbf{X} \cdot \mathbf{W}$, with user-defined `reuse_factor`, `block_factor` is calculated by $N\_in \times N\_out$/`reuse_factor`. 61

4.8 The reuse factor is denoted as R in this example, which determines the level of parallel processing. (a). $R = 1$, the process is fully parallelized, allowing all multiplication tasks to be carried out at once with the greatest number of

multipliers. (b). $R = 2$, $\frac{1}{R}$ of the tasks are processed simultaneously and half of the resources are time-division multiplexed. (c). $R = 4$, 4 multipliers are time-division multiplexed, turning the computation into sequential execution. 63

4.9 This figure shows three different experiment setups. (a) is the Thresholding method implemented on Kasli. (b) is our FPGA solution with ViT as the classifier. (c) is the GPU solution; it employs 3 COTS cards with the cameralink crab card to acquire cameralink images, a GPU for accelerating ViT inference, and a COTS GPIO card for ViT label outputs. The cameralink grab card and IO card are both PCIe-rooted for low latency. These three COTS are linked together through PC software: C program for cameralink grab and IO cards, Python with Pytorch for ViT on GPU. 65

4.10 The GPU and FPGA solution are mounted in the same PC main board through the PCIe interface. The former employs 3 COTS cards with the cameralink crab card to acquire cameralink images, a GPU for accelerating ML inference, and a COTS GPIO card for ML label outputs. The dashed arrow indicates the data flow. The Sundance FPGA is only powered and controlled by the PC Mainboard; no data transmission is issued between them. 66

4.11 Comparison of experimental and simulated single ion images. Insets show an example of a simulated and experimental image of a single ion. The simulated dataset is generated such that the resulting histograms closely resemble the statistics of the experimental data. 67

4.12 Latency Measurements from Testbed for 3-Qubit Detection. 73

4.13 ILA Waveform over an entire 10×10 pixel image capture. 73

5.1 Overview of the fSEAD Framework. 87

5.2 An Example of A Hash-based AD Hardware Structure in fSEAD. 88

5.3 An Example of Xilinx Partial Reconfiguration Tool flow. 90

5.4 Composable Topology. 92

5.5 Examples of Combination Scheme. 93

5.6 FPGA Layout. 95

5.7 Placement on FPGA. 95

5.8 Ensemble Performance Measured on Dataset: Cardio. 98

5.9 Multi-threaded CPU Implementation for xStream for HTTP-3 103
5.10 Execution Time Comparison of fSEAD and CPU for Detector: Loda. 103
5.11 Execution Time Comparison of fSEAD and CPU for Detector: RS-Hash. 103
5.12 Execution Time Comparison of fSEAD and CPU for Detector: xStream. 103
5.13 Roofline Model on CPU. 105
5.14 Roofline Model on FPGA. 105
5.15 Example of the Scalability inside Single Partial Block: RP-1 106
5.16 Chip Power Consumption. 107
5.17 System Power Test-bed. 107
5.18 Example of fSEAD Channel with Empty Logic. 108

CHAPTER 1

Introduction

1.1 Motivation and Aims

Machine learning (ML) has emerged as a powerful tool for analyzing data, making decisions, and solving complex problems across a wide range of domains. Moreover, the growing demand for real-time data processing in applications such as financial trading systems [3], communication signal processing [4, 5], autonomous robotic agents [6, 7], data analysis in physics [8], and real-time anomaly detection [9, 10] has driven the deployment of ML models directly on edge devices. In these areas, real-time data processing is crucial to enable rapid decision-making and responsiveness.

Integrating advanced ML capabilities into edge devices often requires significant modifications to existing infrastructure and systems, which adds complexity and increases costs. This challenge underscores the need for more efficient hardware solutions. In recent years, general-purpose accelerators, particularly Graphical Processing Units (GPUs), have been extensively used to enhance workload performance. Additionally, there has been growing interest in hardware specifically tailored for specialized ML applications. In this context, Field-Programmable Gate Arrays (FPGAs) and Application-Specific Integrated Circuits (ASICs) are widely utilized to optimize ML computations and accelerate inferencing at the edge (see Figure 1.1).

Among mainstream edge ML hardware, FPGAs stand out for their energy efficiency, low system latency, and greater hardware reconfigurability compared to CPUs and GPUs, as well as lower development costs than ASICs. However, achieving optimal performance on

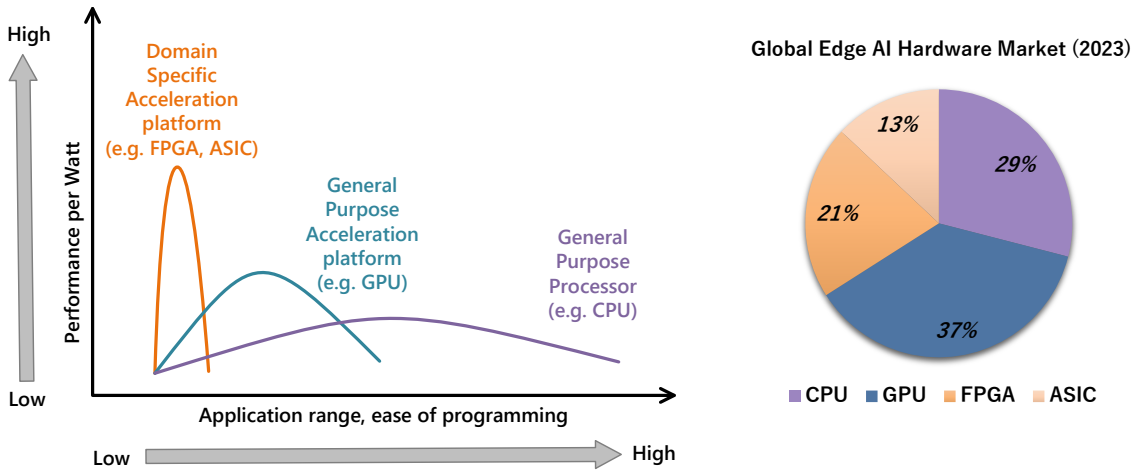


FIGURE 1.1: (a) A trade-off between different hardware architectures: CPU, GPU, FPGA, and ASIC. (b) Data are from Market Research Report (Ref. [11])

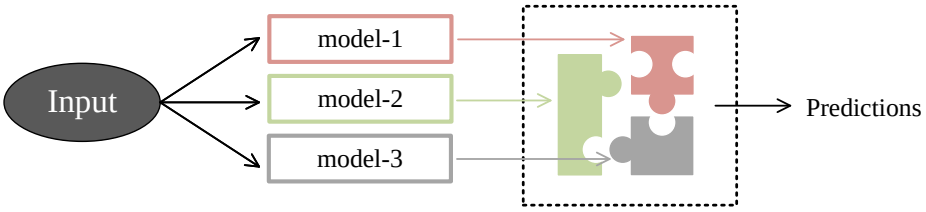


FIGURE 1.2: Ensemble Technique

customized ML tasks with FPGAs poses significant challenges. These include the limited on-chip resources, memory constraints, and the static nature of many FPGA designs, all of which must be carefully balanced when implementing edge ML models on this hardware.

In designing ML accelerators, achieving high accuracy while managing resource constraints is a primary objective. Ensemble methods, which combine multiple weak sub-models into a more accurate overall decision, are frequently employed to address this challenge [12, 13] (see Figure 1.2). These methods not only improve accuracy but also provide scalability, allowing for the integration of an arbitrary number of sub-models depending on available resources and performance requirements.

Integrating ML accelerators with other functional blocks in FPGA-based systems adds further complexity. Effective interaction with external interfaces and peripherals—such as memory, sensors, or communication modules—is crucial for real-time data processing and

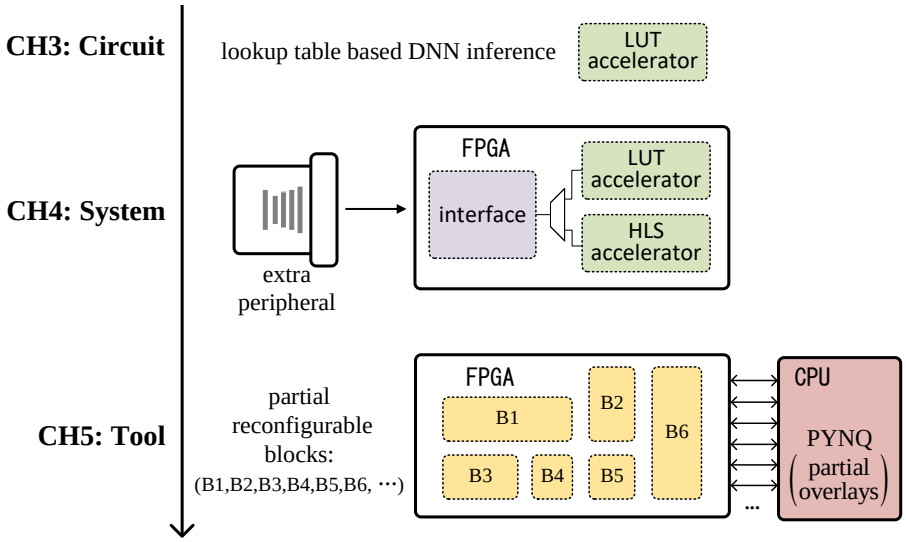


FIGURE 1.3: Thesis structure.

environmental interaction. Although modern GPUs offer higher computational capacity, a well-designed FPGA can outperform a GPU in edge tasks by directly accessing data from peripherals and minimizing buffering delays that often impede CPU and GPU performance.

Additionally, most FPGA-based ML systems are static, which limits their adaptability to evolving requirements. Partial reconfiguration addresses this issue by allowing specific regions of the FPGA to be reprogrammed without affecting the entire system, thereby enhancing hardware reuse and adaptability. This capability underscores the need for designing composable hardware that can dynamically reconfigure to meet varying operational demands and adapt to the evolving needs of complex, real-time systems.

1.2 Contributions

This thesis emphasizes specialized optimization at the circuit, system, and tool level to meet the unique demands of implementing ML-aided tasks on FPGAs. A summary of each contribution is encapsulated in Figure 1.3.

First, to achieve both fast and accurate deep neural network (DNN) inference, we introduce a LUT-based neuron architecture for FPGAs, named LUTEnsemble. Traditional LUT-based

DNNs often sacrifice accuracy to achieve ultra-low latency due to the extreme sparsity needed to keep them within manageable sizes. LUTEnsemble overcomes this challenge to enhance accuracy by integrating multiple PolyLUT [14] sub-neurons using an adder tree structure and assembling them into an ensemble with sparsely connected sub-networks. To the best of our knowledge, for similar accuracy, LUTEnsemble produces the best-reported FPGA latency and area trade-off on the benchmarks tested. Additionally, it provides the advantage of improved uncertainty estimation accuracy.

Next, we address a specific application: qubit state measurements in trapped-ion quantum information processing. This thesis investigates how FPGAs and integrated neural networks can solve challenges in this field by delivering low latency while maintaining high accuracy. Two acceleration approaches are explored: the previously mentioned LUTEnsemble and a new Vision Transformer, designed using high-level synthesis (HLS). These accelerators are tailored for low-latency and high-accuracy solutions, respectively. To handle system latency with an electron-multiplying charge-coupled device (EMCCD) camera as the sensor peripheral, we establish a direct interface between the FPGA and the EMCCD camera, eliminating buffering and interface overheads. This setup results in a total detection latency reduction of $119\times$ and $94\times$ for one- and three-qubit tests compared to the GPU baseline system, which relies on the peripheral component interconnect express (PCIe) for data movement.

Finally, this thesis explores strategies to enhance the composability and scalability of FPGA-based SoCs. We introduce a flexible computing framework for FPGA streaming ensemble anomaly detection (fSEAD) tasks. It incorporates multiple partially reconfigurable regions, called pblocks, each assigned a specific function. These pblocks are interconnected via an AXI switch, enabling arbitrary composition and merging of results at runtime to optimize FPGA resource utilization and accuracy. Leveraging reconfigurable Dynamic Function eXchange (DFX), the detector can adapt to evolving environmental conditions. We validate this approach on PYNQ [15] by comparing fSEAD with an equivalent CPU implementation across four public datasets, achieving speed-ups ranging from $3\times$ to $8\times$.

1.3 Thesis Structure

The main contributions of this thesis have been previously published in the following references:

- [1] Binglei Lou, Richard Rademacher, David Boland, and Philip Leong. "PolyLUT-Add: FPGA-based LUT Inference with Wide Inputs", 34th International Conference on Field-Programmable Logic and Applications (FPL), 2024.
- [2] Binglei Lou, David Boland, and Philip Leong. "fSEAD: A Composable FPGA-Based Streaming Ensemble Anomaly Detection Library." ACM Transactions on Reconfigurable Technology and Systems (TRETS), 2023.

The thesis structure for each chapter is given below:

- Chapter 2 provides background on (1) machine learning: neural networks, anomaly detection, and ensemble methods, (2) FPGA design workflows, and (3) FPGA-aided machine learning on quantum computers.
- Chapter 3 describes a hardware design for a fast and accurate lookup table-based DNN implementation (Ref. [1]).
- Chapter 4 proposes a low latency FPGA control system for real-time machine learning-based qubit detection in trapped-ion quantum computing.
- Chapter 5 investigates approaches to enhance the composability and scalability of FPGA-based hardware design (Ref. [2]).
- Chapter 6 conclude this thesis.

CHAPTER 2

Background

This chapter provides a brief background on machine learning and its hardware implementation. It begins with a discussion of the theory and descriptions of neural networks, anomaly detection, and ensemble methods. Following this, the chapter explores the mainstream workflow of deploying machine learning on FPGAs. Finally, it delves into the background and motivation for using FPGA-aided applications in physics experiments, specifically focusing on real-time machine learning-based qubit detection in trapped-ion quantum computing.

The chapter is organized as follows: In Section 2.1, neural networks and anomaly detection are presented as two representative methods in the machine learning category. Additionally, ensemble methods are introduced as a model augmentation technique to enhance the performance of both neural networks and anomaly detection tasks. In Section 2.2, the workflow of deploying machine learning on FPGAs is discussed, with the tool of dynamic partial reconfiguration discussed as a special case. In Section 2.3, the background of trapped-ion quantum computing and the significant role FPGAs can play in its qubit detection process are briefly introduced as a concrete application. This section highlights how FPGA-based machine learning solutions can meet the low-latency and high-accuracy requirements of this task.

2.1 Machine Learning

2.1.1 Neural Network

This section briefly describes the background of two well-known deep neural network structures. First, the Multilayer Perceptron (MLP) is introduced as a dense neural network. It was chosen because it is the simplest universal approximator, *i.e.*, it can approximate any continuous function [16]. Then, we introduce Vision Transformers (ViTs), which are designed to process structured grid data such as images. ViTs rely on the self-attention mechanism from transformer models, which were originally developed for natural language processing (NLP) tasks [17]. ViTs divide an image into patches and treat each patch as a 'token', similar to words in a sentence. Through layers of self-attention, ViTs can model long-range dependencies and global context, leading to state-of-the-art performance on various image recognition benchmarks [18, 19].

Multilayer Perceptron (MLP)

Given the simplicity of the MLP, both inference and training processes are covered. This section details the Forward and Backward passes, corresponding to neural network inference and training phase respectively, and the Stochastic Gradient Descent (SGD) is discussed as the training optimizer.

Forward path:

Figure 2.1 shows an example of an MLP structure with the colors green, orange, and blue representing the input layer, hidden layers, and output layer, respectively. The MLP has a total of L layers. Each layer implements the vector function $F^l : \mathbb{R}^{N_l} \rightarrow \mathbb{R}^{N_{l+1}}$ (N_l is the inputs number of the l^{th} layer). We denote the weights for the l^{th} layer as $\mathbf{w}^l \in \mathbb{R}^{N_{l+1} \times N_l}$, and the bias $\mathbf{b}^l \in \mathbb{R}^{N_{l+1}}$. The result of the linear transformation of l^{th} layer is described by $\mathbf{z}^l \in \mathbb{R}^{N_{l+1}}$, and $\mathbf{a}^l \in \mathbb{R}^{N_{l+1}}$ is the result of the non-linear transformation applied to $\mathbf{z}^l$. To process the computation, it multiplies the activation inputs of the previous vector $\mathbf{a}^{l-1}$ by

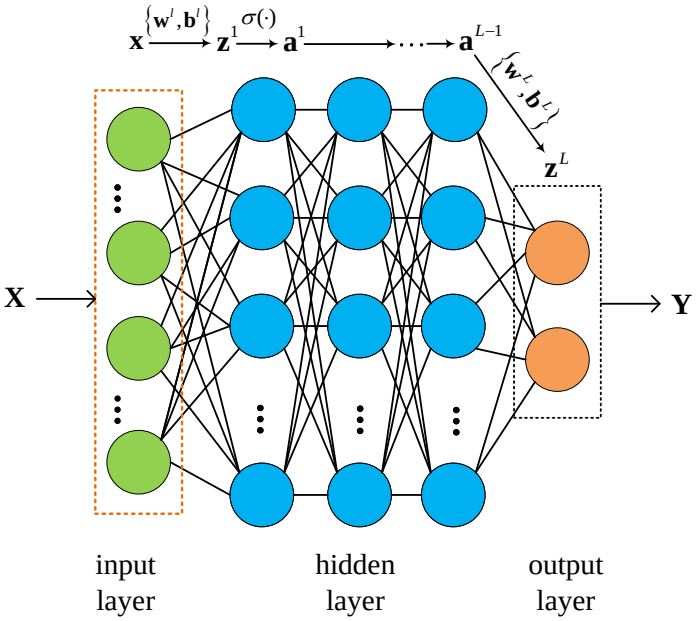


FIGURE 2.1: The block diagram of MLP.

the trained weights for the layer $\mathbf{w}^l$, adding a bias $\mathbf{b}^l$, and passing the result $\mathbf{z}^l$ through a non-linear activation function σ.

Specially, for an input $\mathbf{a}^0 = \mathbf{x}$, the first layer is computed as $\mathbf{z}^1 = \mathbf{w}^1\mathbf{x} + \mathbf{b}^1$. The final output, $\mathbf{y}$, is equal to $\mathbf{z}^L$. The forward path is given as Eq. (2.1) to Eq. (2.4).

$$\mathbf{a}^0 = \mathbf{x} \tag{2.1}$$

$$\mathbf{z}^l = \mathbf{w}^l\mathbf{a}^{l-1} + \mathbf{b}^l \tag{2.2}$$

$$\mathbf{a}^l = \sigma\left(\mathbf{z}^l\right) \tag{2.3}$$

$$\mathbf{y} = \mathbf{z}^L \tag{2.4}$$

Backward path:

The weights $\mathbf{w}^l$ and biases $\mathbf{b}^l$ form model parameters θ that are updated during the training process, *i.e.*, $\theta = \{\{\mathbf{w}^l\}, \{\mathbf{b}^l\}\}$. These can be found using gradient descent-based algorithms,

with stochastic gradient descent (SGD) being commonly used one [20, 21]. The back-propagation algorithm computes the error/activation gradient (*i.e.*, the derivative of the cost function with respect to backward activations), using the chain rule from the last to the first layer [22]. This can be achieved through outer products of the weight gradient and forward activations to avoid iterative calculations of intermediate terms when deriving the weight gradient. The process is described mathematically in Eq. (2.5) to Eq. (2.8).

$$\delta^L = \nabla_{\mathbf{a}}\mathbf{C} \odot \sigma'\left(\mathbf{z}^L\right) \tag{2.5}$$

$$\delta^l = \left(\left(\mathbf{w}^{l+1}\right)^T \delta^{l+1}\right) \odot \sigma'\left(\mathbf{z}^l\right) \tag{2.6}$$

$$\frac{\partial \mathbf{C}}{\partial \mathbf{b}^l} = \delta^l \tag{2.7}$$

$$\frac{\partial \mathbf{C}}{\partial \mathbf{w}^l} = \mathbf{a}^{l-1}\delta^l \tag{2.8}$$

Computation of the gradient of the output layer δ^L, requires the element-wise multiplication ($\odot$) of the partial derivatives of the cost function $\mathbf{C}$ with respect to the output activation ($\nabla_{\mathbf{a}}\mathbf{C} = \partial\mathbf{C}/\partial\mathbf{a}^L$), and $\sigma'\left(\mathbf{z}^L\right)$, as given in Eq. (2.5). To back-propagate this to the l^{th} layer, we multiply the transpose of the relevant weight matrix $\mathbf{w}^{l+1}$ by the gradient δ^{l+1}, and then compute its Hadamard product with $\sigma'(\mathbf{z}^l)$, as shown in Eq. (2.6). This is repeated for all layers of the network. Finally, Eq. (2.7) and Eq. (2.8) give the partial derivatives with respect to the weights and biases $\partial\mathbf{C}/\partial\mathbf{w}^l$ and $\partial\mathbf{C}/\partial\mathbf{b}^l$ from δ^l and $\mathbf{a}^{l-1}$.

Training Optimization:

A comprehensive discussion of various neural network optimizers can be found in Ref. [23]. This section introduces stochastic gradient descent as an example of an optimizer. SGD estimates the overall gradient using a randomly chosen mini-batch. To enhance the convergence speed of SGD, one of the most popular optimization mechanisms—momentum—is introduced as well [24]. Here, we discuss the same momentum scheme as implemented in PyTorch [25].

The SGD algorithm begins by randomly selecting m training inputs (m refers to batch size here and is usually small). We label these random training inputs as $X_1, X_2, \ldots, X_m$, and refer to them as a mini-batch of training data. For a given layer-l, the weights and biases are updated according to Eq. (2.11) and Eq. (2.12), where $\mathbf{w}_{t+1}$ and $\mathbf{w}_t$ represent current and previous weight respectively, lr is the learning rate, $\mathbf{v}_{\mathbf{w}_{t+1}}$ and $\mathbf{v}_{\mathbf{b}_{t+1}}$ represents the current 'velocity' vectors for weights an biases, computed by Eq. (2.9) and Eq. (2.10). The partial derivatives of the cost function with respect to the weights and biases for mini-batches of training inputs are averaged to compute the relevant velocity.

$$\mathbf{v}^l_{\mathbf{w}_{t+1}} = \mu * \mathbf{v}^l_{\mathbf{w}^l_t} + \frac{1}{m}\sum_{j=1}^{m}\frac{\partial \mathbf{C}_{X_j}}{\partial \mathbf{w}^l_t} \tag{2.9}$$

$$\mathbf{v}^l_{\mathbf{b}_{t+1}} = \mu * \mathbf{v}^l_{\mathbf{b}^l_t} + \frac{1}{m}\sum_{j=1}^{m}\frac{\partial \mathbf{C}_{X_j}}{\partial \mathbf{b}^l_t} \tag{2.10}$$

$$\mathbf{w}^l_{t+1} = \mathbf{w}^l_t - lr * \mathbf{v}^l_{\mathbf{w}^l_{t+1}} \tag{2.11}$$

$$\mathbf{b}^l_{t+1} = \mathbf{b}^l_t - lr * \mathbf{v}^l_{\mathbf{b}^l_{t+1}} \tag{2.12}$$

Vision Transformer (ViT)

Vision Transformer is a promising neural network for computer vision that is based on Transformers. After the general Transformer architecture was introduced in 2017 [17], it gained widespread attention and has become one of the most promising neural network architectures in the field of Natural Language Processing. The Transformer tries to capture relationships between different words of a text to be analyzed. The ViT architecture, where images are processed without the need for convolution layers, was proposed in 2019 [18] and later empirically evaluated [19]. ViTs capture the relationships between different portions of an image by breaking down input images into a collection of patches that, once transformed into vectors, are seen as words in a normal Transformer. In recent years, the Transformer model has demonstrated superior performance and greater efficiency compared to Convolutional Neural Networks (CNNs) and residual network (ResNet)-based CNNs in image classification tasks [26]. For instance, according to Ref. [19], ViT models pre-trained on the JFT-300M

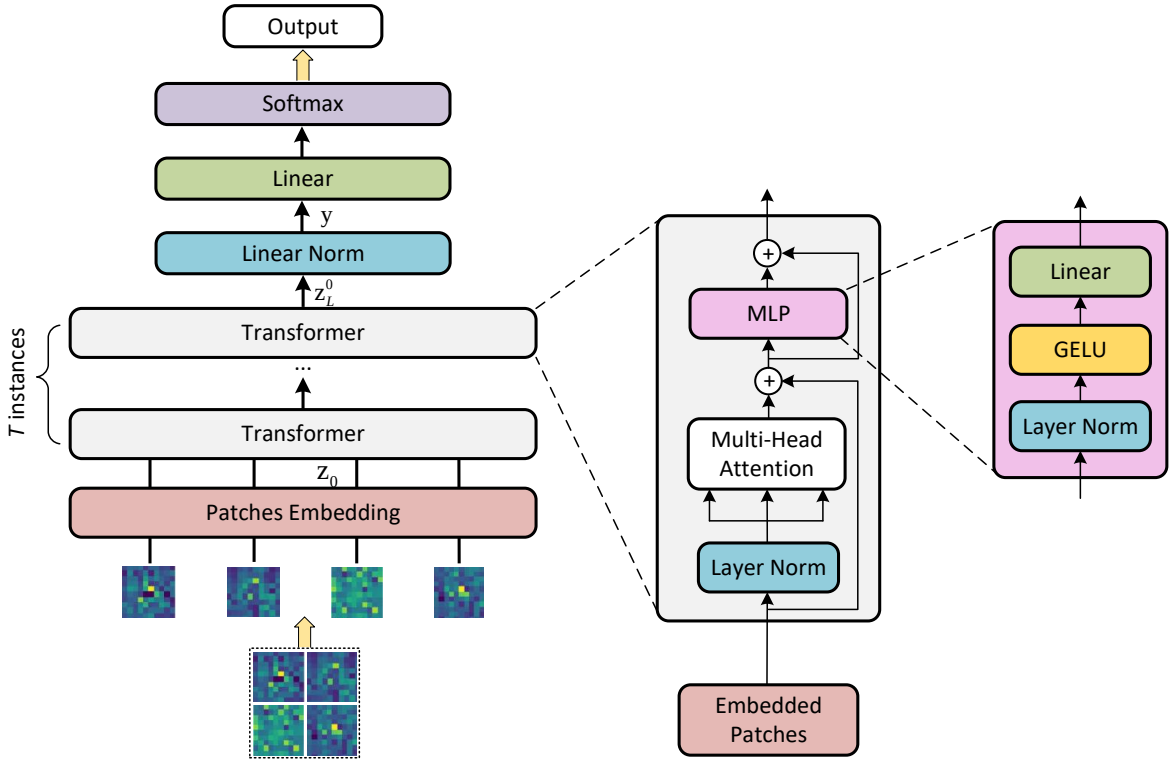


FIGURE 2.2: Block diagram of ViT.

dataset outperform ResNet-based baselines on all tested datasets (*e.g.*, ImageNet, CIFAR-100, and CIFAR-10, *etc.*), while taking substantially less computational resources to pre-train. The test accuracies for model 'ViT-H/14' versus 'ResNet152x4' are reported as follows: 88.55% vs. 87.54% on ImageNet, 94.55% vs. 93.51% on CIFAR-100, and 99.50% vs. 99.37% on CIFAR-10.

In what follows, we detail the working principle and architecture of the ViT, whose block diagram is illustrated in Figure 2.2. The 2D input image of size $\mathbf{x} \in \mathbb{R}^{H\times W}$ is first reshaped into a sequence of flattened 2D patches $\mathbf{x}_p \in \mathbb{R}^{N\times P^2}$, where (H, W) is the resolution of the original image and (P, P) is the resolution of each image patch. $N = HW/P^2$ is the resulting number of patches and also serves as the effective input sequence length for the Transformer. As formulated in Eq. (2.13), we use $\mathbf{x}_p^n \in \mathbb{R}^{P^2}, n = 1, 2, .., N$ to represent N independent patches. The flattened patches are then sent to a single feed-forward layer that contains an embedding matrix $\mathbf{E} \in \mathbb{R}^{P^2\times D}$, where D is a constant latent vector size throughout all subsequent Transformer layers. A learnable embedding $\mathbf{z}_0^0 = \mathbf{x}_{class}$ is also

attached to the sequence of embedded patches to integrate global information from the image for the final classification task. For a general $\mathbf{z}_{layer}^{index}$, $layer$ corresponds to the layer index, and $index$ corresponds to the index of the inside component. Finally, the standard learnable 1D position embedding, $\mathbf{E}_{pos} \in \mathbb{R}^{(N+1)\times D}$, is added to the patch embedding to retain positional information. The resulting sequence of embedding vectors $\mathbf{z}_0$ serves as an input to the following Transformer block.

Each Transformer consists of three major processing elements: Layer Norm (LN), Multi-headed Self-Attention (MSA), and multi-layer perceptron [1]. Additionally, blocks within the Transformer have connections to one another (see Eq. (2.14) and Eq. (2.15)). The final MLP block, also referred to as the MLP head, corresponds to the transformer's output. Categorical labels are provided by a softmax function: $\sigma\left(x_i\right) = e^{x_i}/(\sum_{j=1}^{N} e^{x_j})$, which provides a probability distribution indicating the likelihood that the predicted output belongs to each target label.

$$\mathbf{z}_0 = \left[\mathbf{x}_{class}; \mathbf{x}_p^1\mathbf{E}; \mathbf{x}_p^2\mathbf{E}; \cdots; \mathbf{x}_p^N\mathbf{E}\right] + \mathbf{E}_{pos}, \mathbf{E} \in \mathbb{R}^{P^2\times D}, \mathbf{E}_{pos} \in \mathbb{R}^{(N+1)\times D} \tag{2.13}$$

$$\mathbf{z'}_l = \text{MSA}\left(\text{LN}\left(\mathbf{z}_{l-1}\right)\right) + \mathbf{z}_{l-1} \tag{2.14}$$

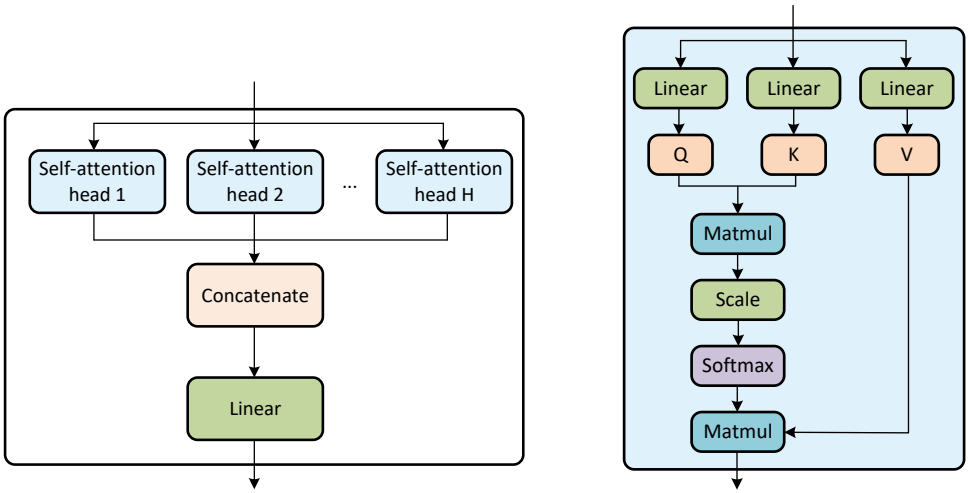
$$\mathbf{z}_l = \text{MLP}\left(\text{LN}\left(\mathbf{z'}_l\right)\right) + \mathbf{z'}_l \tag{2.15}$$

$$\mathbf{y} = \text{LN}\left(\mathbf{z}_L^0\right) \tag{2.16}$$

$$\text{Output} = \text{softmax}(\mathbf{y}\mathbf{W}_y + \mathbf{b}_y\ ) \tag{2.17}$$

The MSA comprises several self-attention (SA) blocks, concatenate, and linear layers (see Figure 2.3(a)). The SA is at the heart of the Transformer and is composed of linear layers, matrix multiplication and a softmax function (see Figure 2.3(b)). The SA is formulated in Eq. (2.18) and Eq. (2.19), where Q, K, and V correspond to $queries$, $keys$, and $values$ respectively. For standard SA, head$_i$, $Q_i, K_i, V_i \in \mathbb{R}^{(N+1)\times D}$ are calculated using matrix

[1] This MLP notation inside ViT is different from the standard MLP discussed in Section 2.1.1. It contains three layers: an LN, Linear, and a Gaussian Error Linear Unit (GELU [27]) non-linearity, which is unformulated in this chapter for simplicity. The LN is formulated as $\text{LN}\left(x_i\right) = \frac{\gamma_i(x_i-\mu_x)}{\sqrt{\sigma_x^2+\epsilon}} + \beta_i$ where μ_x is the mean of the input vector x, σ_x^2 is the variance of the input vector x, ϵ is a small value to avoid division by zero, and γ_i and β_i are the i^{th} trainable scaling and shifting parameters respectively.



(a) MSA (b) SA

FIGURE 2.3: Block diagram of multi-headed self-attention in (a) and self-attention in (b).

multiplications of an input sequence $\mathbf{z} \in \mathbb{R}^{(N+1)\times D}$ with weight matrices $\mathbf{W}_{Qi}, \mathbf{W}_{Ki}, \mathbf{W}_{Vi} \in \mathbb{R}^{D\times D}$ (see Eq. (2.18)). The Attention function is defined in Eq. (2.19), where d is the dimension of the $queries$/$keys$/$values$. It is also important to note that multiple blocks of SA, referred to as $heads$, can be used together. This allows for each head to attend to information relating to a different hidden characteristic that refers to MSA. The MSA, illustrated in Figure 2.3(a), is an extension of SA in which H parallel $heads$ are operated and project their concatenated outputs. The MSA is defined in Eq. (2.20) and Eq. (2.21), where $\mathbf{W}_O \in \mathbb{R}^{H\times D\times D}$.

$$Q_i = \mathbf{z}_{l-1}\mathbf{W}_{Qi},\ K_i = \mathbf{z}_{l-1}\mathbf{W}_{Ki},\ V_i = \mathbf{z}_{l-1}\mathbf{W}_{Vi} \tag{2.18}$$

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d}})V \tag{2.19}$$

$$\text{head}_i = \text{Attention}(Q_i, K_i, V_i) \tag{2.20}$$

$$\text{MSA}(\mathbf{z}_{l-1}) = \text{Concat}(\text{head}_1, ..., \text{head}_H)\mathbf{W}_O \tag{2.21}$$

Finally, the output of the L-th transformer layer, $\mathbf{z}_L$, is processed to produce the final output. The class token $\mathbf{y}$ is usually used for this purpose in classification tasks, where $\mathbf{W}_y \in \mathbb{R}^{D \times K}$, $\mathbf{b}_y \in \mathbb{R}^K$ are learnable parameters, and K is the number of classes.

2.1.2 Anomaly Detection

Anomaly detection is a key machine learning task, which refers to the automatic identification of unforeseen or abnormal samples embedded in a large amount of normal data. It can be used for diverse applications, including fault detection in surveillance systems and financial transactions, network security, aircraft, or potential risks in health data [28, 29].

Figure 2.4 and Figure 2.5 demonstrate two different anomaly detection scenarios. The top side of Figure 2.4 is a time series where anomalies occur where the amplitude is notably higher. The bottom part shows a running anomaly score, with the bars in red indicating an anomalous region of the time series. In Figure 2.5, the normal data are shown in blue, with the outliers being in brown.

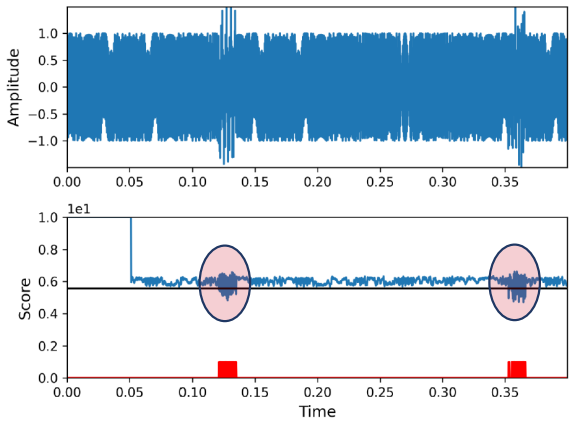


FIGURE 2.4: Anomaly Detection Case-1.

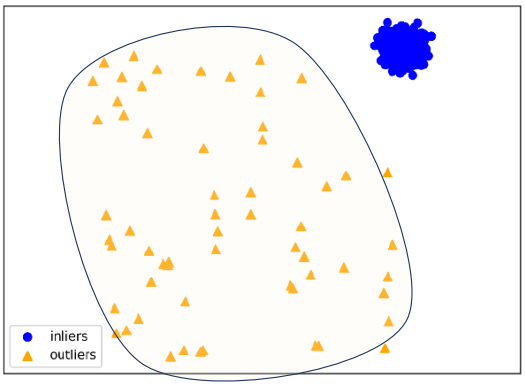


FIGURE 2.5: Anomaly Detection Case-2.

From the data processing perspective, we distinguish between static and streaming anomaly detection types. Static detectors usually operate on a relatively large batch of data before performing information extraction and feature analysis to identify rare items, events, or observations from the general distribution of a population. Representative methods include k-Nearest Neighbors (kNN) [30], Local Outlier Factor (LOF) [31], and Principal Component

Analysis (PCA) [32]; for a more comprehensive collection of static methods, we refer the reader to the SUOD [33]. While batch processing can lead to high throughput and accuracy, it is unsuitable for systems requiring real-time performance since the computing resource requirements and latency increase with batch size.

In contrast, streaming methods only store and process a window of recent instances [29, 34]. This is more amenable to achieving accurate anomaly scores under limited memory, processing, and time constraints. Moreover, the algorithms are designed to facilitate more lightweight and potentially real-time implementations. A group of algorithms that support streaming anomaly detection processing includes, but is not limited to, ensemble-centric methods [35, 36, 37], tree-based methods [38, 39, 40], kernel-methods [29], as well as neural network-based solutions such auto-encoders [41] and adversarial models [42].

Real-time anomaly detection is critical in applications where timely identification of unusual patterns is essential for preventing failures and ensuring security. FPGAs are particularly well-suited for this task due to their low latency and ability to handle high data throughput.

2.1.3 Ensemble Techniques

Ensembles are a class of methods that combine weak sub-models to collectively form a more accurate decision by utilizing diversity in the sub-models [12, 13]. Each sub-model in an ensemble can be data-independent and structure-identical. The first feature makes it naturally amendable for parallel processing, while the latter provides the flexibility to assemble arbitrary numbers of sub-models into an ensemble according to the computational resources and desired accuracy. The ensemble method can boost performance for both neural networks and anomaly detection.

Example: Neural Network

The ensemble has been shown to be able to boost predictive performance, *e.g.*, classification accuracy on ImageNet dataset [43] and Kaggle contests [44]. The neural network-based

ensemble is further investigated in Ref. [45] with `deep ensemble` proposed. Taking image classification as an example, Figure 2.6 shows an inference implementation of `deep ensemble` with an ensemble size of E. The training process (omitted in this figure) initializes the training parameter θ_e of each sub-net randomly, and the data points are also randomly shuffled to generate ensemble diversity. In inference, E copies of a single image are fed into E feedforward passes, each of which predicts the probabilities of a class. Finally, these are combined via averaging to produce the predicted probabilities $p(y|x) = E^{-1} \sum_{e=1}^{E} p_{\theta_e}(y|x, \theta_e)$.

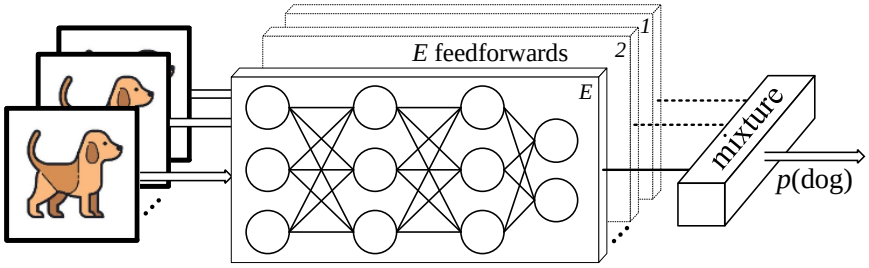


FIGURE 2.6: Diagram of the `deep ensemble` neural network. This example improves diversity using E independent feedforward passes of the same input. The final prediction is estimated by combining E predictions on the output.

Each ensemble member can be trained in parallel without interaction, making this approach suitable for distributed and parallel computing. The major limitation, however, lies in the requirement for multiple feedforward passes, leading to an amplified number of computation delays and memory storage. Additionally, ensuring diversity among the individual neural networks within the ensemble poses another challenge. While random initialization and shuffling can help introduce diversity, they may not always be adequate, especially in complex datasets where subtle patterns may require nuanced representations.

An efficient solution to the above-mentioned multiple feedforward passes problem is the `multi-input multi-output (MIMO)` architecture [46]. This achieves the advantages of ensemble methods with minimal computational overhead. A `MIMO` ensemble allows a single model to accommodate multiple sub-networks, with each being independently trained. This enhances model robustness with minimal additional computational requirements. Following the same application, Figure 2.7 shows an `MIMO` with an ensemble size $E = 2$. The main challenge in `MIMO` ensembles is achieving diversity, *e.g.*, its performance diminished when

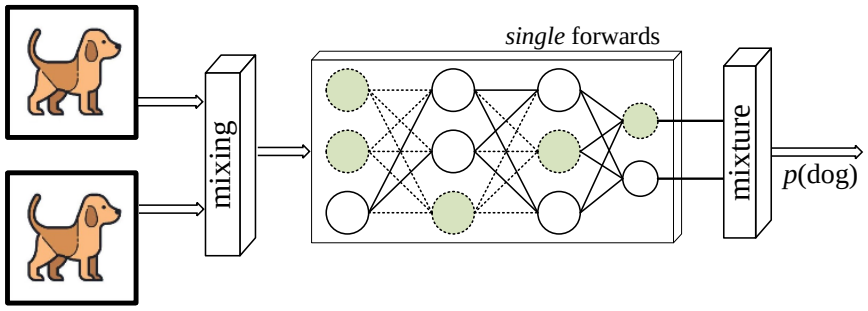


FIGURE 2.7: Diagram of MIMO ensemble neural network. E is set to 2 in this case; diversity among this ensemble is guaranteed from 2 compact sparse sub-models inside the single feedforward pass.

$E > 2$. This necessitates carefully designed augmentation techniques and structural optimizations. For instance, Ref. [47] proposed mixed sample data augmentations—particularly with rectangular CutMix patches—enhancing results by making subnetworks stronger and more diverse. Ref. [48] proposes the addition of an early-exits ensemble to the MIMO architecture with inferred depth-wise weightings to produce multiple predictions for the same input, giving a more diverse ensemble.

Example: Anomaly Detection

An example of applying ensemble to anomaly detection is illustrated in Figure 2.8, which demonstrates an architecture for streaming ensemble anomaly detectors (SEADs), where each sub-detector inside SEAD takes a stream of input data and produces a stream of transformed outputs which indicate the anomaly scores. Averaging is used to compute the final score from all sub-detectors, or a *threshold* can be applied to translate this averaged score to a *Labels* (anomaly or no anomaly). While a simple ensemble could be multiple instances of the same detector, a more powerful ensemble will utilize different detectors.

An easy-to-use scalable library allows one to explore the performance of ensembles, many of which have been developed. Examples of anomaly detection packages include: ELKI Data Mining [49] and RapidMiner [50] in Java; Outliers [51] in R; SUOD [33], PyOD [52] and PySAD [34] in Python. Aside from their differences in programming languages, different libraries are also tailored to different kinds of anomaly detection, *e.g.*, PySAD focuses in

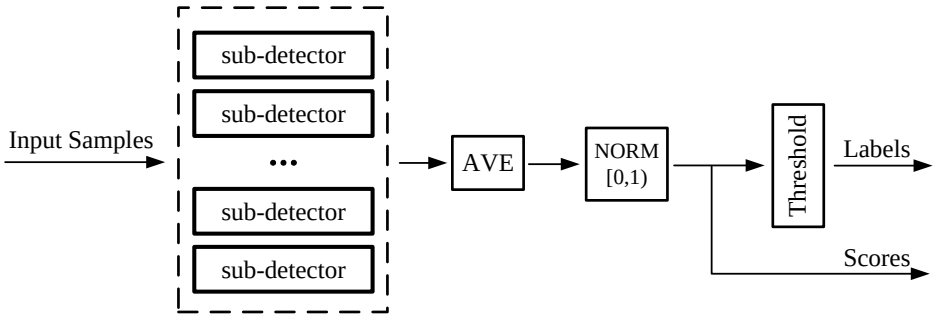


FIGURE 2.8: Sample Architecture for SEAD Methods.

particular on a framework of streaming anomaly detections in Python, whereas only static approaches can be accessed by SUOD and PyOD.

In addition to supporting comprehensive detectors, the libraries provide the ability to combine the output of these models in different ways beyond simply taking the average or maximum across all the base models. A software toolkit *combo* [53] contains more than 15 model combination methods in Python, including basic generic and global methods (like Averaging, Maximization, Weighted Averaging, Feature Bagging *etc.* [54, 55, 56]) and dynamic selection/combination models (such as DCS [57] and LSCP [58] *etc.*). Developers can implement particular combination techniques according to specific system requirements.

2.2 FPGA Design Workflows

Table 2.1 provides a comparison between CPUs, GPUs, and FPGAs [59, 60, 61]. CPUs were originally designed as the processing unit for general-purpose computing tasks, focusing on sequential operations and versatility. GPUs, on the other hand, were initially intended for rendering graphics and images. Over time, they have evolved into powerful parallel processing units and become a popular choice for deep neural networks, thanks to their high performance for data-parallel computations. Although they have specialized architectures, they are supported by standardized tools and libraries like CUDA and OpenCL.

While FPGA architectures were not initially designed with machine learning as a primary focus, well-designed FPGA hardware offers significant advantages in energy efficiency,

TABLE 2.1: Pros and Cons of CPU, GPU and FPGA

Hardware	Speed	Developemnt Proficiency	Energy Efficiency	Hardware Flexibility
CPU	low	high	medium	×
GPU	high	high	low	×
FPGA	high	low	high	✓

latency, and flexibility for specialized machine learning applications. However, FPGAs still lag behind general-purpose hardware like GPUs and CPUs in terms of programming ease. This section reviews the mainstream FPGA programming workflow, highlighting its evolution from low-level to high-level design abstractions over time.

Register-Transfer Level (RTL) languages like VHDL and Verilog were the norm in the early stages. Despite their precise control over hardware, these languages demanded substantial expertise and were time-consuming, which led to the development of High-level synthesis (HLS) tools. HLS tools, such as AMD/Xilinx Vivado HLS [62], emerged as a solution, enabling FPGA designs to be written in higher-level languages like C/C++. Intel also provides tools similar to Vivado HLS, such as the Intel HLS Compiler [63], which supports high-level synthesis using C++ to generate optimized hardware description code. Building on the HLS platform, the $hls4ml$ framework offers a promising approach to bridge the gap between machine learning software frameworks and FPGA HLS tools [64]. $hls4ml$ enables developers to train machine learning models in Python using popular libraries like TensorFlow or PyTorch and then automatically generate synthesizable HLS code. This development marks a significant step towards democratizing FPGA programming, making it more accessible to a broader audience. MATLAB HDL Coder [65] provides similar functionality, allowing MATLAB code to be directly converted to VHDL or Verilog, thereby streamlining the FPGA programming process. OpenCL [66], a parallel programming framework, has been widely adopted for FPGA programming, particularly for accelerating compute-intensive applications. Many FPGA vendors offer OpenCL support, such as Xilinx's SDAccel and Intel's OpenCL SDK.

To further simplify FPGA programming, several new frameworks and tools have emerged, each focusing on enhancing FPGA-based applications through advanced design methodologies. QKeras [67] and QPyTorch [68] are not FPGA-specific but are useful training

frameworks designed for quantizing neural networks. QKeras targets the Keras framework, while QPyTorch is tailored for PyTorch. These tools enable developers to train neural networks with reduced bit precision, optimizing them for FPGA acceleration. Complementing these efforts, Brevitas [69] serves as a versatile PyTorch library focused on training neural networks with arbitrary bit-widths. It offers flexibility in quantization schemes and enables developers to explore various precision levels to balance between model accuracy and resource utilization on FPGAs.

On the other hand, FINN [70] is a framework for designing FPGA accelerators based on binarized neural networks. By leveraging binary quantization, FINN significantly reduces model size and computational complexity, making it well-suited for resource-constrained FPGA deployments. Similarly, DNNBuilder [71] is an automated tool for building high-performance DNN hardware accelerators on FPGAs. It bridges the gap between fast DNN software construction and slow hardware implementation through novel techniques such as high-quality RTL components, fine-grained pipeline architecture, and column-based cache schemes. DNNBuilder enhances throughput, reduces latency, and optimizes FPGA resource utilization, demonstrating significant performance improvements on various DNNs.

These tools provide a comprehensive suite for designing and optimizing machine learning for FPGA deployment.

2.2.1 Special Case: Dynamic Partial Reconfiguration

FPGA technology provides the flexibility to modify a hardware implementation without re-fabrication. Partial reconfiguration (PR) takes this flexibility a step further, allowing dynamic changes to an active design. This requires implementing static logic and multiple Reconfigurable Partitions (RPs) with various Reconfigurable Modules (RMs). The RP is the level of hierarchy within which different RMs can be implemented, and an RM is the netlist or HDL description that is implemented within an RP. Generally, multiple RMs with the same interface exist for a specific RP. Dynamic Function eXchange refers to a Xilinx tool flow that achieves the partial reconfiguration [72, 73].

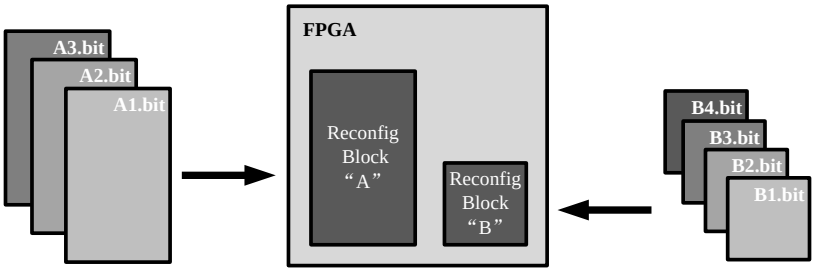


FIGURE 2.9: Basic Premise of Partial Reconfiguration.

Figure 2.9 illustrates the basic premise of partial reconfiguration. The grey area of the FPGA block represents static logic, and the blocks marked as Reconfig Block 'A' and Reconfig Block 'B' represent the RPs. The functionality implemented in Reconfig Block 'A' can be modified by downloading one of several partial BIT files: A1.bit, A2.bit, or A3.bit; similarly, the functions implemented in Reconfig Block 'B' can be modified by one of B1.bit to B4.bit.

Dynamic partial reconfiguration (DPR) on FPGAs has been used for numerous FPGA applications, including video processing [74] and cognitive radio [75]. For FPGA-based neural network implementation, DPR can also play a significant role. One approach involves splitting the NN into chunks consisting of a few layers, which are then executed separately using DPR to upload and process the next chunk. This approach minimizes the resource usage of the accelerator [76]. Another study used the same hardware resources for each convolution layer for lower area consumption, which was executed sequentially. However, the classifier throughput was significantly affected by the DPR overhead [77]. DPR can also be used in low-power applications by adjusting power consumption according to the power level of the battery [78]. Ref. [79] use DPR to select convolutional filters based on class datasets to improve classification performance, although changing the kernel filter size at run time may be inefficient using DPR [80].

Overall, DPR has emerged as a versatile technique for optimizing FPGA-based applications, with its effectiveness depending on the specific problem and design choices. Further research is needed to understand better the trade-offs involved and identify the most effective approaches for each use case.

2.3 FPGA aided Machine Learning on Quantum Computers

Machine Learning is increasingly used in physics experiments to enable powerful representation abilities in data processing and analysis [81, 82, 83, 84, 85]. Figure 2.10 gives an example of ML-aided qubit detection in a trapped ion system. FPGA plays a critical role in accelerating this system through hardware-optimized ML implementation and an efficient image capture interface between ML and its peripherals, such as an Electron-Multiplying Charge-Coupled Device (EMCCD) camera in this context.

Following the example in Figure 2.10, this section is introduced with three parts: trapped ion, qubit measurement, and machine learning solutions.

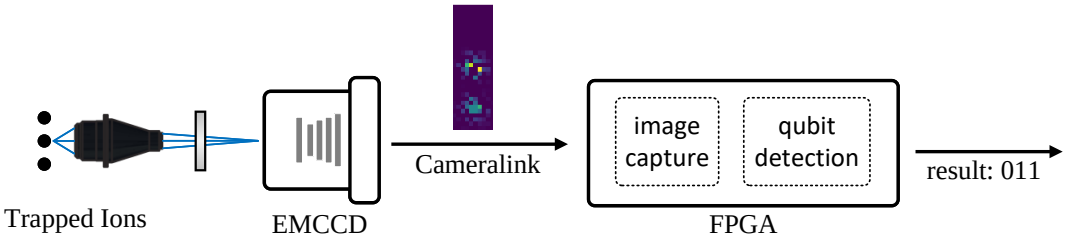


FIGURE 2.10: Example of FPGA-aided Qubit Detection in Tapped-Ion. The photons scattered by the ions are collected and magnified by optical elements and directed toward an EMCCD camera. These images are then transmitted to an FPGA via the Cameralink protocol [86]. The FPGA processes the images in real time, and an embedded classifier determines the qubit states.

2.3.1 Trapped-Ion

Trapped ions are one of the leading platforms for quantum information processing. It leverages the unique properties of ions—atoms with an electric charge due to the loss or gain of electrons—which are confined using electric and magnetic fields.

The fundamental unit of quantum information is the 'quantum bit', or 'qubit', which is the quantum mechanical analog of the classical 'bit'. Qubits are encoded in the electronic states of the trapped ions and may exist in two distinct quantum states, labeled $|0\rangle$ and $|1\rangle$.

Before experiments, the qubit is initialized into a known state $|0\rangle$. After quantum computations, the state of the qubit is measured using state-dependent fluorescence. Detailed in

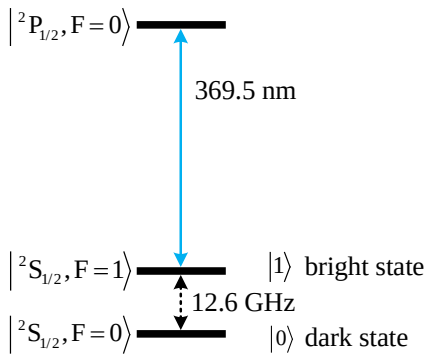


FIGURE 2.11: Electronic level structure of $^{171}Yb^{+}$. The 369.5 nm transition is used for Doppler cooling to near the motional ground state and for qubit measurement. The state is measured by detecting photons from scattering from the separate lasers on this wavelength. The $^2S_{1/2}$ hyperfine splitting (12.6 GHz) defines two measurable qubit states: bright and dark. Quantum gate operations are performed using an off-resonant 355 nm optical transition (not shown).

Figure 2.11, the target experiment ion of this thesis is illuminated with a 369.5 nm detection laser beam that drives the $|^2S_{1/2}, F = 1\rangle$ to $|^2P_{1/2}, F = 0\rangle$ transition (see Figure 2.11). The ion will interact with the detection beam and fluoresce if the qubit is in the $|1\rangle$ state. However, if the qubit is in the $|0\rangle$ state, the ion will not fluoresce since the detection beam is off-resonant. Therefore, the qubit's state can be inferred from the number of emitted photons.

2.3.2 Qubit Measurement

Qubit state measurements are an integral part of quantum information processing. Quantum systems are highly susceptible to errors from decoherence [87, 88], and the error correction system must detect and correct them before they propagate and accumulate above a certain threshold. This is a key ingredient to realizing a fault-tolerant quantum computer [89, 90] and involves error syndrome measurements of qubits followed by in-sequence, conditional corrective operations. For these strategies to be effective, the underlying qubit state measurements must be accurate to ensure the correct identification of errors and rapid enough to outpace the decoherence times of qubits. Another driver for fast qubit detection is that classification latency should be minimized so as not to limit the quantum computer's total

clock speed. This requirement imposes two challenges for qubit measurement per single-shot operation: high fidelity and low latency.

TABLE 2.2: Related works with reported qubit detection latency

Reference	Ion-trap Experiment	Measurement	Detector	readout latency$*$	detection latency	Total latency
Myerson *et al.* [91]	Ca^+	PMT	Maximum Likelihood	-	145 μs	-
Halama *et al.* [92]	Be^+	EM-CCD	Thresholding	2 ms	400 μs	-
Burrell *et al.* [93]	Ca^+	EM-CCD	Thresholding	-	400 μs	-
Pino *et al.* [94]	$Yb-Ba$	Q-CCD	Thresholding	-	120 μs	133.35 ms
Jeong *et al.* [85]	Yb^+	EM-CCD	ResNet*	-	-	-

$*$: The readout latency includes frame grabber latency and cameralink data movement latency.
*: No latency aspect reported for the ResNet-based solution.

In the literature, a reasonable qubit detection latency is at the microsecond (μs) level, while the coherence time could be the second (s) level [95, 96]. Table 2.2 presents the qubit detection latency in modern trap-ion computers with different, where 120-400 μs were reported for different setups. In particular, Ref. [94] also reported the total time of 133.35 ms for an 'N=6 QV Circuit' example, with qubit detection consumes 120 μs.

In this thesis, fluorescence from ions is collected onto an EMCCD camera [97] which converts the photons into a digital signal (see Figure 2.12). This camera has a W$\times$ H = 512$\times$ 512 'image area', where each pixel has a single photon sensitivity. The EMCCD operates by Frame Transfer, so images captured on the exposed sensor are transferred to a storage area by moving rows of pixel charges downwards. To read out an image, rows of charges are shifted vertically into a readout register and then shifted horizontally into an amplifier and an analog to digital converter (ADC) [97, 98].

After gathering the digital-format fluorescence information, the following classification aims to determine the most likely state of an ion. A common classification method uses thresholding (*e.g.*, Ref. [92, 93, 94]), in which the total number of photons collected during a measurement is compared to a pre-optimized threshold (see Figure 2.13). Classification on EMCCD cameras using thresholding is achieved by defining regions of interest (ROI) around each ion, in which pixels are summed together, and only the total photon count is recorded [99, 100]. However, the fidelity worsens due to cross-talk between ROIs (see Figure 2.14).

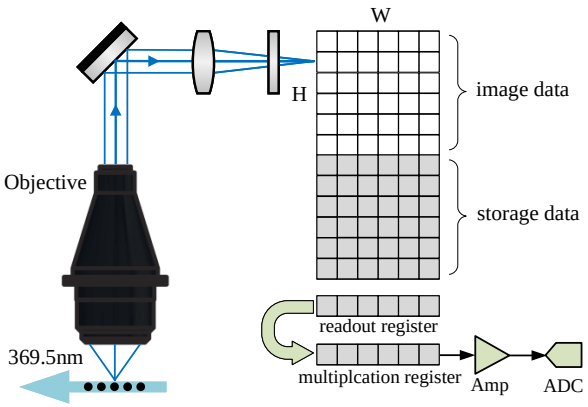


FIGURE 2.12: Detection of photons emitted by ions with an EMCCD camera. Fluorescence at 369.5 nm from a string of ions is collected using an objective lens and focused onto an EMCCD's active sensor area. Frame Transfer allows the image to be transferred to a storage area before being amplified and converted to a digital signal through an ADC.

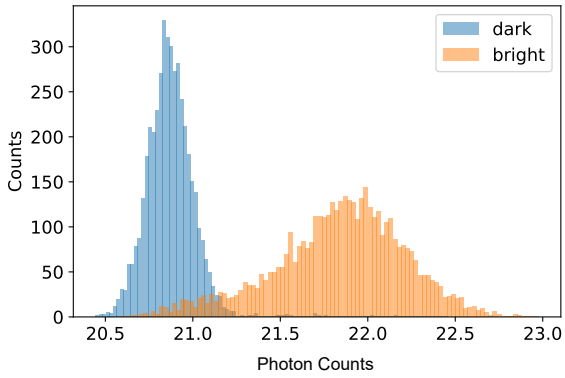


FIGURE 2.13: Histogram of thresholding method. In the training process, We process histograms for the dark (shown in blue) and bright images (shown in orange), and determine an optimal threshold that maximizes the fidelity. Afterward, a measurement is processed by calculating the total number of photon counts in the image and comparing it to the pre-calculated threshold, and it's unavoidable that some samples will be misclassified.

2.3.3 Machine Learning aided Solutions

In recent years, machine learning has been applied to various applications in quantum computing, showing promising results and opening up new avenues for exploration and development [81, 82, 83, 84]. For instance, Ref. [81] leverages control theory and machine learning to predict and suppress qubit decoherence. It introduces a time-division-multiplexed approach and predictive feedback, significantly improving qubit phase stability. Moreover, various

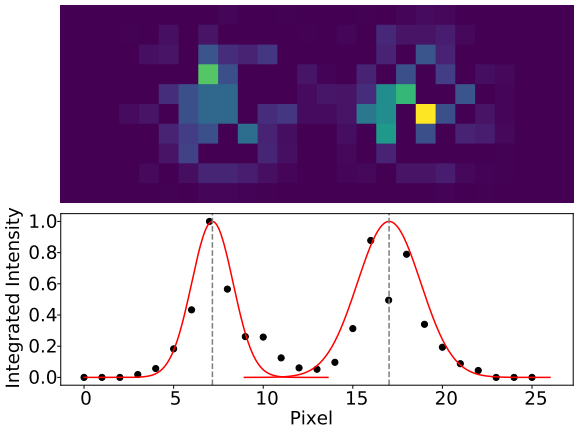


FIGURE 2.14: An example of simulated 2-ion crosstalk with 2-D resolution. The integrated pixel brightness along the y-direction (black dots) is fitted with a Gaussian function (red line). One can see an overlap between the signals, demonstrating visible cross-talk between the two ions.

machine learning algorithms for state estimation and forward prediction of qubit dynamics under non-Markovian dephasing are analyzed in [82] where the autoregressive Kalman Filter approach outperforms its Fourier-based counterpart, underscoring the importance of algorithm choice in qubit state prediction. In the field of trapped-ions, a neural network was employed to reduce excess micromotion, thereby enhancing the ion's coherence time [83]. Machine learning for computer vision has also been applied to trapped ions. Several neural networks for computer vision, such as time-stamped neural network (TSNN) and recurrent neural network (RNN), were used to improve ion state classification [84]. Particularly for the qubit detection task, Jeong *et al.* [85] applies a residual network based CNN model to improve 4-qubit state measurements using an EMCCD and achieves a fidelity error reduction of $\sim$50% compared to the maximum likelihood method [101] (shown in Table 2.2). However, their analysis was done offline and did not consider latency. This limits its suitability for fast quantum error correction.

2.3.4 Summary

To enhance the precision of qubit analysis in an EMCCD-based detection system, integrating machine learning techniques for 2D image classification can fully leverage the improved spatial resolution provided by EMCCD technology. The neural network models (MLP and

ViT) and ensemble techniques discussed in Section 2.1 are utilized as the 2D-image classifier in the following chapters of this thesis. These approaches show significant promise for advancing qubit detection accuracy. However, to achieve effective real-time qubit detection, both the image readout process and the subsequent ML-based detection algorithm must meet stringent latency requirements. For example, modern trapped ion systems report readout latencies of approximately 2 ms and detection latencies between 120 and 400 μs. However, as will be discussed in Chapter 4, our commercial off-the-shelf (COTS) GPU-based testbed (NVIDIA GeForce RTX 2080 Ti) exhibits a total latency of 212-215 ms, with a ViT-based data analysis latency of 2.8-2.9 ms, and the remainder attributed to readout latency. This performance is not comparable to modern trapped ions.

We propose that an FPGA-based ML solution, as illustrated in Figure 2.10, has the potential to meet the high fidelity and low latency requirements necessary for qubit measurement and single-shot operation. However, this system design presents several challenges. Firstly, the ML accelerator must be optimized to be fast and accurate on the target FPGA. Secondly, readout latency should be addressed by interfacing the EMCCD directly to the FPGA's onboard cameralink interface, eliminating buffering and interface overheads. This allows the ML accelerator to process the data from the EMCCD efficiently. Additionally, scalability and composability are also essential in balancing the trade-offs between latency, area, and detection accuracy in dynamic system environments.

CHAPTER 3

Fast and Accurate Look-up Table based Neural Networks

This chapter describes an FPGA implementation of neural networks based on the lookup table (LUT). It investigates how to improve the connectivity of LUT-based FPGA inference in a scalable way, therefore enhancing performance. The presentation is based on the background given on neural networks and ensemble in Chapter 2 and expands on two related works: PolyLUT-Add [1] (published in FPL2024) and LUTEnsemble. This chapter primarily focuses on LUTEnsemble, which includes comprehensive approaches to enhancing LUT-based FPGA neural networks. PolyLUT-Add, introduced as a preliminary study and special case of LUTEnsemble, illustrates the contribution of wider inputs for higher connectivity.

Deep neural networks (DNNs) have been shown to provide powerful feature extraction and regression capabilities and are widely employed across a spectrum of applications, including image classification for autonomous driving [6], data analysis in particle physics [8], and network intrusion detection for cybersecurity [9]. Field-Programmable Gate Arrays (FPGAs)

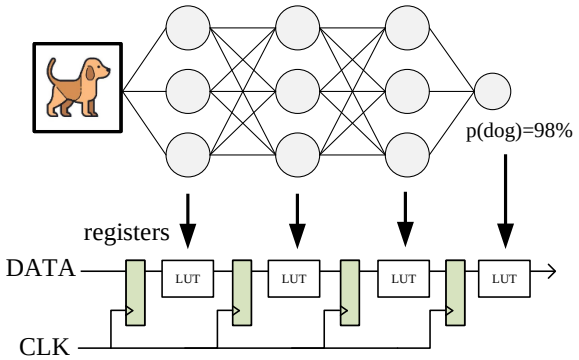


FIGURE 3.1: An example of LUT-based DNN on an image classification task.

provide a unique implementation platform for deploying DNNs, with significant advantages over other technologies, particularly in real-time inference tasks.

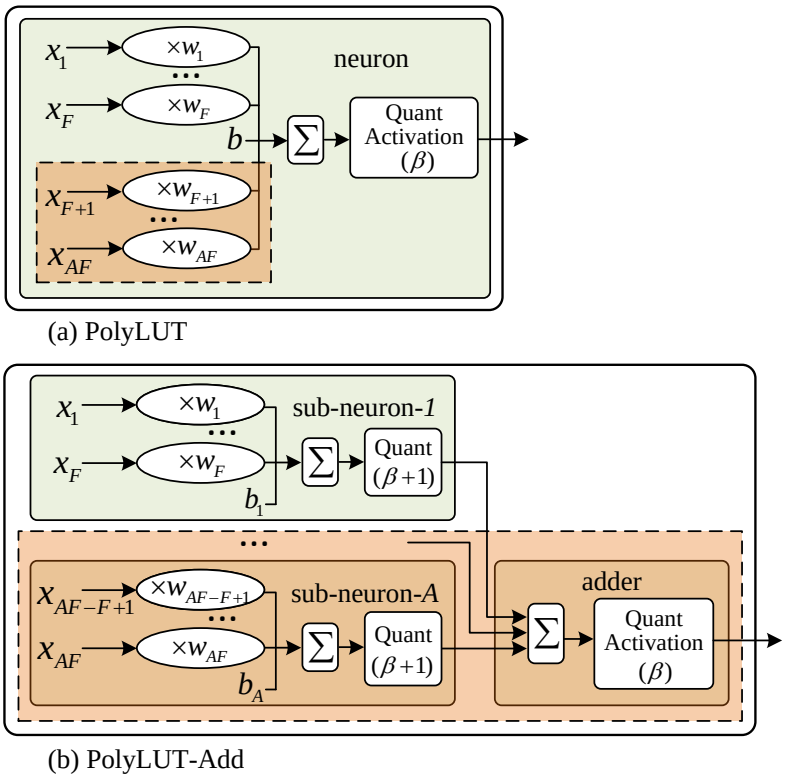
Lookup Table (LUT) based DNNs use the primitive elements of FPGAs to achieve high area efficiency and ultra-low latency (an example is shown as Figure 3.1) Compared with Binary Neural Networks (BNNs [102]), which utilize 1-bit quantization to replace multipliers with simple XNOR gates, LUT-based neurons further optimize FPGA resource utilization by using LUTs as direct inference operators that can implement arbitrary Boolean functions. Examples of accelerators published using this approach include LUTNet [103], LogicNets [104], NullaNet [105], PolyLUT [14] and NeuraLUT [106].

Even though LUT-based DNNs offer ultra-low latency for various benchmarks, *e.g.*, 20 ns inference latency is achievable [14], they suffer from low accuracy due to being constrained to having extremely sparse connectivity. This is due to the exponential scaling of LUT usage with the number of inputs, restricting LUT-based DNNs to architectures with small fan-in. Denoting the fan-in to be F and the wordlength of each input as β, this requires $\mathcal{O}(2^{\beta F})$ LUTs for each output bit. Referring to Figure 3.2(a), if the fan-in is increased $A\times$ this would become $\mathcal{O}(2^{A\beta F)}$, an exponential increase.

To address this problem and significantly improve accuracy, this chapter introduces LUTEnsemble, an extension of the PolyLUT framework [1] with multi-level optimizations.

First, for each neuron computation, we combine A copies of PolyLUT sub-neurons via an arbitrary adder tree structure to increase neuron fan-in. PolyLUT-Add, a special case of this LUTEnsemble contribution, combines A copies of PolyLUT sub-neurons via a single A-input adder to increase neuron fan-in, as demonstrated in Figure 3.2(b). This approach has LUT demand $\mathcal{O}(A \times 2^{\beta F} + 2^{A(\beta+1)})$ with an extra bit used for each sub-neuron intermediate output to avoid overflow.

Second, we incorporate E sub-nets with random initial connectivity patterns to form an ensemble. Ensembles are a class of methods that combine weak sub-models to form an improved model by utilizing diversity [12, 13]. As LUT-based DNN sub-nets have inherent diversity, the accuracy, and scalability can be further improved.



(a) PolyLUT

(b) PolyLUT-Add

FIGURE 3.2: Comparison between PolyLUT and PolyLUT-Add. (a) For AF inputs, PolyLUT requires $\mathcal{O}(2^{A\beta F})$ LUTs. (b) PolyLUT-Add uses A sub-neurons each with F inputs requiring $\mathcal{O}(A \times 2^{\beta F})$ LUTs and the adder box requires a lookup table with $\mathcal{O}(2^{A(\beta+1)})$ LUTs. Only $A \in \{2, 3\}$ are supported since larger values ($A \geq 4$) would lead the adder box to be an area and latency bottleneck.

Lastly, for datasets with strong spatial correlation, *e.g.*, image classification, we propose a customized Mixer feature mask layer. This approach improves networks with extreme sparsity and has a marginal area and latency overhead.

The contributions of this work can be summarized as:

- At the algorithmic level, we introduce LUTEnsemble that: (1) Applies a Mixer layer to the input, serving to increase the receptive field; (2) combines A PolyLUT sub-neurons using an adder tree to form an *adder tree neuron*; (3) incorporates E sub-nets with random connectivity to the output of the Mixer layer, forming an ensemble.

- We further test the hypothesis that LUTEnsemble has the added advantage of improving uncertainty estimation accuracy. Our results show that LUTEnsemble significantly outperforms PolyLUT and PolyLUT-Add in terms of negative log-likelihood on MNIST data and on out-of-distribution (OOD) data by quantifying the accuracy of an MNIST-trained network on the NotMNIST dataset.
- To the best of our knowledge, for similar accuracy, LUTEnsemble and PolyLUT-Add produce the best-reported FPGA latency and area trade-off on the tested benchmarks. The PolyLUT-Add is open-sourced for reproducible research. Source code and data to reproduce our results are available from Github [1].

3.1 Related Work

TABLE 3.1: Related Work of LUT-based DNN Methods

Method, (year)	**Sparsity**	**LUT representation**
LUTNet [103], (2019)	pruning of high-precision model	logic expansion from XNOR
NullaNet [105], (2019)	lossy truth table sampling	linear + activation neurons
LogicNets [104], (2020)	a prior fixed sparsity	linear + activation neurons
PolyLUT [14], (2023)	a prior fixed sparsity	multivariate polynomials + activation neurons
NeuraLUT [106], (2024)	a prior fixed sparsity	nonlinear sub-networks

Wang *et al.* introduced LUTNet, the first FPGA inference scheme optimized for LUTs [103]. It pruned a high-precision trained network and subsequently performed logic expansion to map the network operations onto k-input LUTs, thereby leveraging the strengths of FPGA architectures. LogicNets [104] and NullaNet [105] adopted a different approach by quantizing the inputs and outputs of each neuron and encapsulating the neuron's computation (*i.e.*, densely connected linear and activation functions) in a truth table. This method enumerated all possible combinations of a neuron's inputs and determined the corresponding outputs based on the neuron's weights and biases. By replacing popcount operations with Boolean expressions, significant computational savings were made. For a DNN model with total L layers, each implementing the transfer function $F^l : \mathbb{R}^{N_l} \rightarrow \mathbb{R}^{N_{l+1}}$ (N_l is the number of neurons in the l^{th} layer). As illustrated in Figure 3.3, a maximum of F inputs are randomly selected from N_l nodes of l^{th} layer to connect to each neuron of the $(l+1)^{th}$ layer. This

[1]PolyLUT-Add: https://github.com/bingleilou/PolyLUT-Add

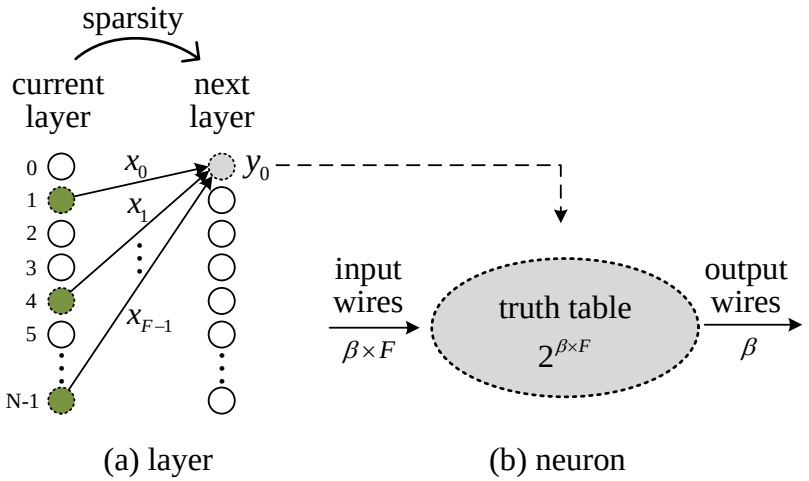


FIGURE 3.3: Illustration of the LUT-based DNN inference scheme used in LogicNets, PolyLUT, and NeuraLUT.

forms a sparse connection between the layers that can be directly implemented using LUTs in an FPGA. Additionally, the bit width of each neuron is quantized as β. Therefore, the transfer function mapping an input vector $[x_0, x_1, \ldots, x_{F-1}]$ to the output node y_0 can be implemented using βF inputs in hardware, and hence requires $\mathcal{O}(2^{\beta F})$ LUTs. The constraint $F \ll N_l$ is applied to limit the size of the lookup table, but this usually results in a degradation of accuracy.

To improve the accuracy of LogicNets, several works are proposed and can be split into two categories: (1) improve the representation ability of each LUT. (2) improve the fan-in of each neuron. PolyLUT [14] and NeuraLUT [106] proposed by Andronic *et al.* utilize the former approach. Specifically, PolyLUT enhanced accuracy and reduced the number of required layers by introducing piecewise polynomial functions. NeuraLUT maps entire sub-nets to a single LUT, enabling a deeper NN and better accuracy. As the sub-nets are absorbed within the LUT, the NN topology and precision within a partition do not affect the number of entries in the lookup tables generated. In contrast, the proposed PolyLUT-Add in this chapter can be categorized in the latter approach, addressing the fan-in bottleneck of LUT-based DNNs with an extension of PolyLUT (see Figure 3.2(b)).

To further optimize LUT-based DNN at the algorithmic level, the ensemble method shows good potential to boost performance in a scalable manner. Ensembles are a model augmentation technique to enhance the performance of machine learning tasks, *e.g.*, classification accuracy on the ImageNet [43] dataset and Kaggle contests [107] can be improved using ensembles. The neural network-based ensemble is further investigated in Ref. [45] with `deep ensemble` proposed. As discussed in Chapter 2.1.3, while traditional ensembles rely on carefully designed parameter initialization and training procedures to extend diversity between sub-nets, LUT-based DNNs have a natural diversity as each sub-net can be randomly initialized with different, sparse connectivity.

3.2 Design

3.2.1 Addtree Architecture

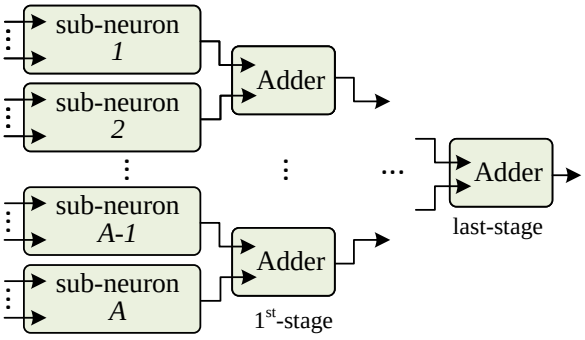


FIGURE 3.4: A single adder tree incorporating A PolyLUT sub-neurons.

As illustrated in Figure 3.4, the adder tree structure introduced in LUTEnsemble builds on the PolyLUT-Add concept (see Figure 3.2(b)) to achieve better scalability with larger A and to prevent exponential growth. The lookup table in each layer can be implemented either as a single combinatorial module or in a pipelined manner. Various pipeline strategies can be employed based on specific requirements (see Figure 3.5). Any of the registers can be removed to map multiple layers as combinational logic in a cascade.

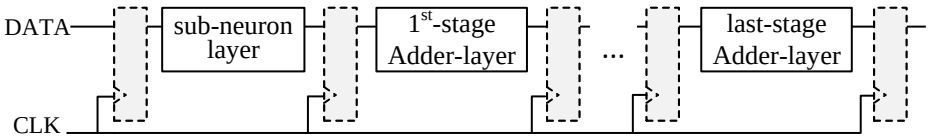


FIGURE 3.5: The dashed boxes represent pipeline registers.

3.2.2 Ensemble

In LUTEnsemble, an ensemble is built based on the `deep ensemble` approach [45] and extended to support model connectivity pattern $\mathcal{C}$ which enhances its diversity, *i.e.*, each sub-net can be randomly initialized with different sparse connectivity. Figure 3.6 illustrates the ensemble. Furthermore, Algorithm 1 describes the LUTEnsemble Training and Inference process. For DNN training, random initialization of $\mathcal{C}$ and distinct random shuffling of the training data points were deployed to increase diversity. The ensemble can be considered a uniformly weighted mixture model that combines E individual predictions.

In practice, the sparsity pattern is controlled by using different random seeds. For example, with an ensemble size of 3, the seed configurations could be set as (base seed, base seed + 1, base seed + 2).

While some randomly generated sparsity patterns may perform poorly, strategically selecting sparsity patterns could provide additional benefits beyond purely random selection. This optimization is left for future work.

3.2.3 Mixer Feature Mask Layer

For image-related applications, high degrees of sparsity (*e.g.*, a fan-in = 6 in PolyLUT [14]) can result in significant information loss. To address this issue, we introduce a Mixer layer to improve connectivity from inputs to subsequent layers. As shown in Figure 3.7, the Mixer layer involves two main steps: feature masking and computation.

The Mixer layer does not change the input dimension. Given an input vector $\mathbf{x} = [x_1, x_2, \ldots, x_{N_0}]$, a feature mask selects M out of N_0 inputs. For the output node y_i, $i = 1, 2, ..., N_0$, this results

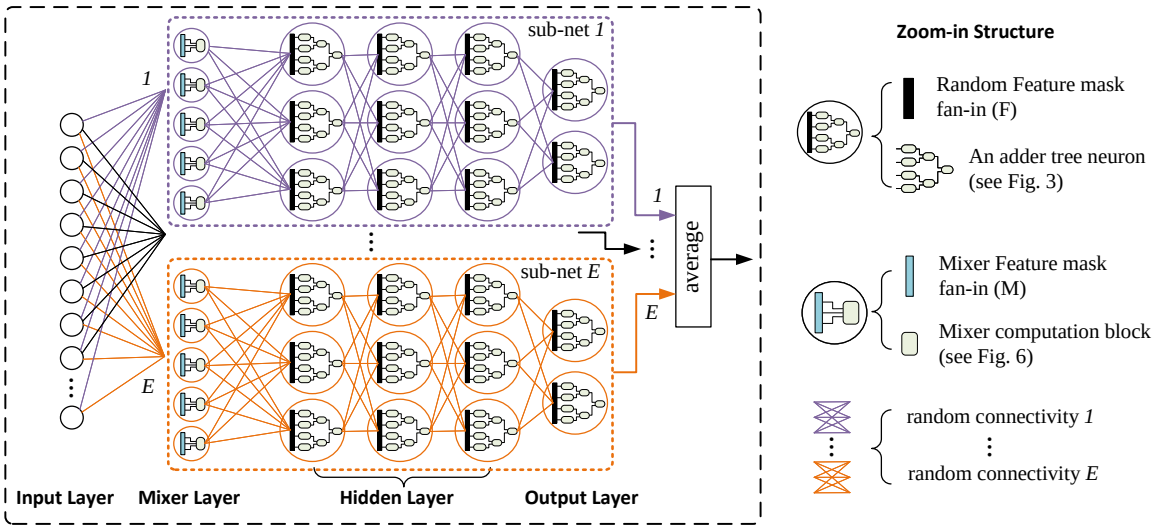


FIGURE 3.6: Architecture of LUTEnsemble with E sub-nets employing random connectivity patterns.

Algorithm 1 Training and Inference procedure for LUTEnsemble with Ensemble Size of E

Input: A neural network initializing a connectivity pattern ($\mathcal{C}$) and parameter distribution (θ) over the ensemble (E), *i.e.*, $p_{\mathcal{C},\theta}(y|x)$

Initialize $\{\mathcal{C}_e, \theta_e, train\_loader_e\}_{e=1}^{E}$ randomly

Initialize $test\_loader$

Training Procedure:

for $e = 1$ *to* E **do**

  Sample data point n_e from $train\_loader_e$

  Train e^{th} sub-net with n_e

  Minimize the loss function $\mathcal{L}_e$

end

Inference Procedure:

The input data from $test\_loader$ is fed into each sub-net and combines the predictions $p(y|x) = E^{-1} \sum_{e=1}^{E} p_{\mathcal{C}_e,\theta_e}(y|x, \mathcal{C}_e, \theta_e)$

in a masked feature vector $\mathbf{x}'_i$ containing M selected input samples. The feature mask selection is illustrated in Algorithm 2. It operates by selecting the corresponding pixel and neighboring spatial pixels, ensuring comprehensive spatial information transmission. For instance, for output node y_i, x_i is always selected in its mask vector $\mathbf{x}'_i$. The rest of the $M-1$ neighboring pixels around x_i are selected in a way that incorporates randomness, enhancing the robustness of the feature selection. In the example of Figure 3.7, the masks selected for output nodes y_1 and y_{19} are $\mathbf{x}'_1 = [x_1, x_2, x_6, x_7]$ and $\mathbf{x}'_{19} = [x_{18}, x_{19}, x_{20}, x_{24}]$ respectively.

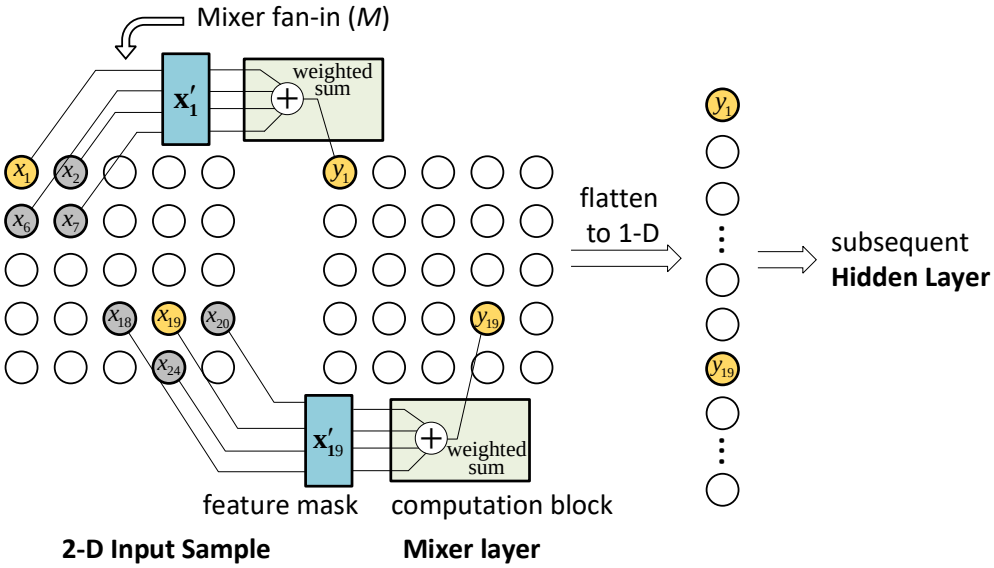


FIGURE 3.7: Block diagram of a Mixer Layer. As illustrated, for an output y_i (y_1 and y_{19} are shown in orange), it takes $M = 4$ inputs around x_i (shown in orange and gray), consistent with the structure depicted in Figure 3.6 and includes a random feature mask (in blue) and trainable weighted sum computation (in green) for training. Following training, the Mixer layer is absorbed into LUTEnsemble nodes.

Subsequently, a weight matrix $\mathbf{W} \in \mathbb{R}^{N_0 \times M}$ is applied to compute the output vector $\mathbf{y}$. Each element y_i of the output vector $\mathbf{y}$ is computed as a weighted sum of the masked features: $y_i = \mathbf{W}_i \mathbf{x}'_i$. The matrix $\mathbf{W}$ is initialized to $1/M$ to ensure uniform importance at the start. It is also trainable, allowing for optimization throughout the learning process. We expect that while keeping the fan-in F of the subsequent layer unchanged, each neuron will be able to receive input information from a broader receptive field.

We observe that the Mixer layer process is similar to and inspired by the operation of the 2-D convolutional layer in the convolutional neural networks (CNNs) with stride=1 [26]. The difference is that the kernel-size constrained area of a convolutional layer is a regular 2D rectangle. In contrast, the fan-in of the Mixer is arbitrary and randomly selected from the surrounding area, providing greater flexibility. Additionally, the weights of the convolutional layer are shared, whereas the weights of the Mixer are independent, allowing each node to have a customized weight allocation. Finally, a key difference is that the node selection in the

Algorithm 2 FeatureMask-2D

Input: Input size: N_0, Mixer fan-in: M, seed, 2D image size (height,width)
Output: mask: $\mathbf{x}'_i, i = 1, 2, ..., N_0$

Set random seed for each sub-net of an ensemble
Function `get_neighbors`$(h, w, radius)$**:**
    Initialize $neighbors$ as an empty list
    for $dh \leftarrow -$*radius* **to** *radius* **do**
        for $dw \leftarrow -$*radius* **to** *radius* **do**
            $nh \leftarrow h + dh$
            $nw \leftarrow w + dw$
            if $0 \leq nh <$ *height and* $0 \leq nw <$ *width* **then**
                Append (nh, nw) to $neighbors$
            end
        end
    end
    return $neighbors$
for $i \leftarrow 1$ **to** N_0 **do**
    $(h, w) \leftarrow$ divmod$(i,$ width$)$
    $radius \leftarrow 1$
    $\mathbf{x}'_i \leftarrow get\_neighbors(h, w, radius)$
    while *len*$(\mathbf{x}'_i) < M - 1$ **do**
        $radius \leftarrow radius + 1$
        $\mathbf{x}'_i \leftarrow get\_neighbors(h, w, radius)$
    end
    if *len*$(\mathbf{x}'_i) > M - 1$ **then**
        $\mathbf{x}'_i \leftarrow$ random$(\mathbf{x}'_i, M - 1)$
    end
    Append (h, w) to $\mathbf{x}'_i$
end
return $\mathbf{x}'$

Mixer layer does not involve padding, ensuring that the processing of edge nodes does not exceed the boundaries.

The Mixer layer is designed specifically for image-type inputs. Similar solutions for datasets lacking a well-defined notion of neighborhood are left for future work.

3.2.4 System Toolflow

Figure 3.8 illustrates the tool flow. Like PolyLUT, the training is conducted offline using PyTorch [25], after which the resulting weights are utilized to construct the LUT for each neuron. These LUTs are then used to produce Register Transfer Level (RTL) files in Verilog, capturing the Boolean expressions derived from the neurons. The RTL is generated through a module generator built on standard Python FILE operations. The final step involves synthesizing the LUT-based DNN design onto hardware using the AMD/Xilinx Vivado tool [108].

By integrating Brevitas [69] with PyTorch, we enable quantization-aware training of DNNs. First, we adapted the network implementation to accept arbitrary A, E, M and F, β, D as hyper-parameters. After training, the model's weights are converted into lookup tables. This conversion process starts by using the quantized states from the trained model to determine the input data range for each neuron. For every neuron, we generate all possible input combinations based on β, F, and M. These combinations are then processed by their respective computational units to produce the corresponding outputs. The input-output pairs are finally compiled to form the values for the lookup table. For the truth-table implementation, the `(* rom_style = "distributed" *)` directive is utilized in the RTL to instruct the tool to infer the LUT ROMs.

3.3 Results

3.3.1 Datasets

The datasets used to evaluate the proposed LUTEnsemble and its PolyLUT-Add special case are shown as follows:

(1) *Handwritten Digit Recognition*: In timing-critical sectors such as autonomous vehicles, healthcare and medical imaging, and real-time object tracking, low-latency

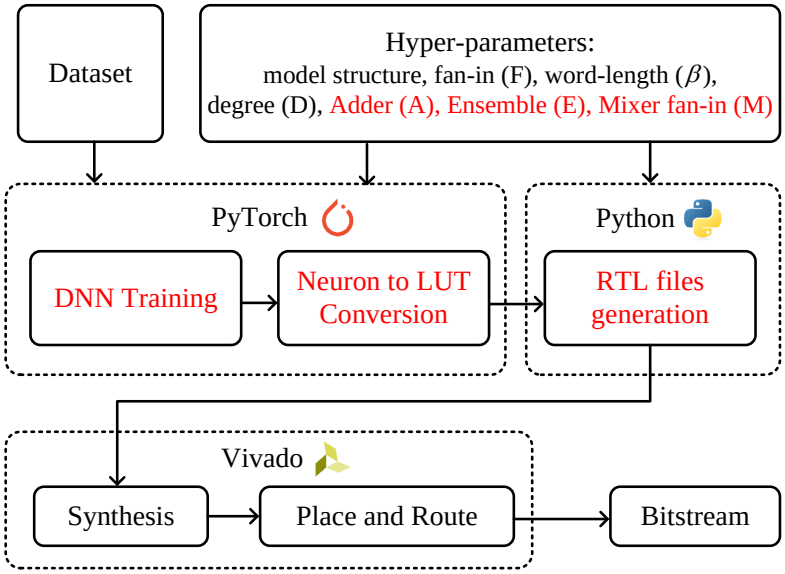


FIGURE 3.8: Tool flow of LUTEnsemble. It is inherited from the open-source PolyLUT toolflow [14]. We show the modified elements in red.

is crucial. These applications underscore the necessity for swift and accurate decision-making, where even minimal delays can have significant repercussions. Unfortunately, there is no public dataset specialized for low-latency image classification tasks so the MNIST [109] is utilized to compare our work with other LUT-based DNNs. MNIST is a dataset for handwritten digit recognition tasks with 28×28 pixels as input and 10 classes as outputs.

(2) *Jet Substructure Classification*: Real-time decision-making is often important for physics experiments such as the CERN Large Hadron Collider (LHC). Jet Substructure Classification (JSC) is one of its applications that requires high-throughput data processing. Prior works [110, 8, 111, 112] employed neural networks on FPGA for this task to provide real-time inference capabilities. We also use the JSC dataset formulated from Ref. [8] to evaluate our work, with the dataset having 16 substructure properties as input and 5 types of jets as outputs. FPGA-based classification in this task must be pipelined to manage a data rate of 40 MHz while ensuring the response latency remains under microsecond.

TABLE 3.2: Model setups used to evaluate different datasets.

Dataset	Model name	Neurons per layer	β	F	D	A
MNIST	HDR	256, 100, 100, 100, 100, 10	2	6	1, 2	2, 3
Jet Substructure	JSC-XL[1]	128, 64, 64, 64, 5	5	3	1, 2	2
Jet Substructure	JSC-M Lite	64, 32, 5	3	4	1, 2	2, 3
UNSW-NB15	NID Lite[2]	686, 147, 98, 49, 1	3	5	1	2

1: Remarks: $\beta_i = 7$, $F_i = 2$
2: Remarks: $\beta_i = 1$, $F_i = 7$

(3) *Network Intrusion Detection*: In the field of cybersecurity, the swift detection and mitigation of network threats are important for the preservation of digital infrastructure integrity (*e.g.*, fiber-optic throughput can reach 940 Mbps). Prior works have used FPGAs to accelerate DNNs, enabling real-time Network Intrusion Detection Systems (NIDS) with high accuracy and enabling privacy on edge devices [14, 104, 113]. The UNSW-NB15 dataset [114] was used as the benchmark for our evaluation process. It has 49 input features and binary classification (bad or normal).

The aforementioned datasets were used for evaluations of the PolyLUT [14]. The UNSW dataset [114] used to test PolyLUT and PolyLUT-Add was omitted in experiments for NeuraLUT [106] as well as LUTEnsemble because its training convergence is very sensitive to the initial random seed.

3.3.2 Experimental Setup

As a foundation for our experiments and to ensure consistency in evaluation, our setup closely follows the PolyLUT [14]. Table 3.2 lists its neural network configurations for three datasets. Using AdamW as the optimizer [25], we trained the smaller models/datasets: (JSC-M Lite, NID Lite) for 1000 epochs, and used 500 epochs for (JSC-XL and HDR). The mini-batch size is set to 1024 and 128 for (JSC-XL and JSC-M Lite, NID-Lite) and MNIST, respectively.

For hardware evaluation, we target the `xcvu9p-flgb2104-2-i` FPGA part as done in LogicNets, PolyLUT, and NeuraLUT. Also, the projects are compiled using Vivado 2020.1 with `Flow_PerfOptimized_high` settings and are configured to perform synthesis in

the `Out-of-Context` (`OOC`) mode. Default values were used as the Place & Route settings for our experiments.

3.3.3 Evaluation of PolyLUT-Add

To evaluate the performance of the wider input enabled by the additional adder layer, we use PolyLUT-Add as a special case with a single-stage adder tree in this section.

Based on the setup in Table 3.2, our newly introduced A is set to $A \in \{2, 3\}$ for models (HDR and JSC-M Lite) with small truth table ($2^{\beta F}$), and $A = 2$ is used for JSC-XL. The polynomial degree $D = 1$ and $D = 2$ correspond to linear and quadratic representations, respectively. We utilize $D \in \{1, 2\}$ to evaluate the performance of PolyLUT-Add. We comment the case $A = 1$ is identical to PolyLUT, and $A = 1, D = 1$ corresponds to LogicNets. We also note that the training convergence of the UNSW dataset is sensitive to the initial random seed, and hence, multiple trials were necessary before a result with good accuracy was achieved. As an exception, we therefore apply $A = 2$ and $D = 1$ to evaluate the NID Lite model.

PolyLUT-Add vs. Deeper and Wider PolyLUT (small D)

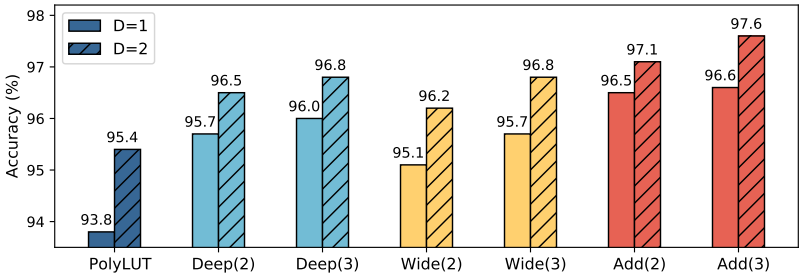
We first present results comparing PolyLUT-Add with PolyLUT in configurations with the same polynomial degree. Three configurations were tested:

(1) **Original PolyLUT**: This serves as the baseline network.

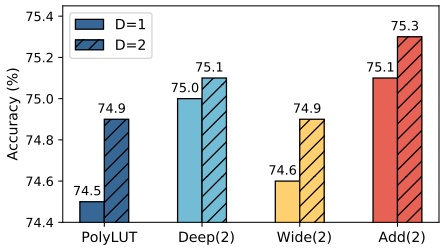
(2) **PolyLUT-Deeper**: This explores the impact of increasing network depth. We denote the depth factor as $\mathbb{D}$. Then $\mathbb{D}\times$ the number of layers is applied to models in Table 3.2. For example, for JSC-M Lite, if $\mathbb{D} = 2$, the hidden layer is doubled, meaning the neurons per layer becomes (64,64,32,32,5).

(3) **PolyLUT-Wider**: This examines the impact of a wider network model. We denote the width factor as $\mathbb{W}$. Then $\mathbb{W}\times$ the number of neurons per layer are applied to models in Table 3.2. Once again, for JSC-M Lite, if $\mathbb{W} = 2$, the neurons per layer becomes (128,64,5).

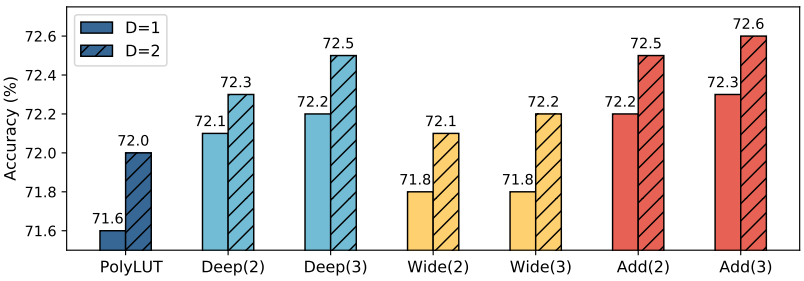
Figure 3.9 shows the accuracy of the configurations with parameter settings detailed in Table 3.2. PolyLUT-Add achieves the highest accuracy against all baselines on all datasets for both the linear ($D = 1$) and non-linear ($D = 2$) cases.



(a) HDR



(b) JSC-XL



(c) JSC-M Lite

FIGURE 3.9: Accuracy results on different models. We use Deep($\mathbb{D}$), Wide($\mathbb{W}$) and Add(A) to denote "PolyLUT-Deeper", "PolyLUT-Wider" and "PolyLUT-Add" respectively.

Optimizing for Accuracy

In terms of accuracy and hardware, Table 3.3 shows that for $A = 2$, PolyLUT-Add achieved accuracy improvements of 2.7%, 0.6% and 2.3% over PolyLUT on the MNIST, Jet Substructure classification and Network Intrusion Detection benchmarks respectively. However, this required a 2-3$\times$ increase in LUT size. We also evaluate the performance of simply increasing

TABLE 3.3: Comparison of accuracy and hardware results (post-synthesis) between PolyLUT and PolyLUT-Add ($\mathbb{D} = 1$, $\mathbb{W} = 1$). The Verilog Generation time was measured on a desktop with Intel(R) Core(TM) i7-10700F @2.9GHz and 64GB memory.

Models	**Degree** D	**Model**	**Fan-in** $(F \times A)$	**Acc(%)↑**	**Lookup table entries↓**	**LUT (% of 1182240)**	**FF (% of 2364480)**	F_max **(MHz)**	**Latency (cycles)**	**RTL Gen. (hours)**
HDR	1	PolyLUT	6	93.8	2^{12}	3.43	0.12	378	6	1.40
			10	96.1	$2^{12} \times 256$		–			–
		PolyLUT-Add	6×2	96.5	$2^{12} \times 2 + 2^6$	12.69	0.12	378	6	3.00
			6×3	**96.6**	$2^{12} \times 3 + 2^9$	20.67	0.12	378	6	4.40
	2	PolyLUT	6	95.4	2^{12}	6.62	0.12	378	6	1.40
			10	97.3	$2^{12} \times 256$		–			–
		PolyLUT-Add	6×2	97.1	$2^{12} \times 2 + 2^6$	19.78	0.07	378	6	3.00
			6×3	**97.6**	$2^{12} \times 3 + 2^9$	31.36	0.07	378	6	4.50
JSC-XL	1	PolyLUT	3	74.5	2^{15}	19.55	0.07	235	5	2.10
			5	74.9	$2^{15} \times 1024$		–			–
		PolyLUT-Add	3×2	**75.1**	$2^{15} \times 2 + 2^{12}$	50.10	0.07	235	5	5.17
	2	PolyLUT	3	74.9	2^{15}	37.40	0.07	235	5	2.30
			5	75.2	$2^{15} \times 1024$		–			–
		PolyLUT-Add	3×2	**75.3**	$2^{15} \times 2 + 2^{12}$	89.60	0.07	235	5	5.24
JSC-M Lite	1	PolyLUT	4	71.6	2^{12}	0.97	0.01	646	3	0.16
			7	72.1	$2^{12} \times 512$		–			–
		PolyLUT-Add	4×2	72.2	$2^{12} \times 2 + 2^8$	2.62	0.01	488	3	0.35
			4×3	**72.3**	$2^{12} \times 3 + 2^{12}$	4.33	0.01	363	3	0.63
	2	PolyLUT	4	72.0	2^{12}	1.51	0.01	568	3	0.16
			6	72.5	$2^{12} \times 512$		–			–
		PolyLUT-Add	4×2	72.5	$2^{12} \times 2 + 2^8$	4.29	0.01	440	3	0.34
			4×3	**72.6**	$2^{12} \times 3 + 2^{12}$	6.57	0.01	373	3	0.64
NID Lite	1	PolyLUT	5	89.3	2^{15}	6.86	0.15	529	5	4.09
			8	91.0	$2^{15} \times 512$		–			–
		PolyLUT-Add	5×2	**91.6**	$2^{15} \times 2 + 2^8$	21.41	0.15	529	5	8.76

−: Data for very high fan-in settings has been omitted due to CUDA memory limitations on the GPU and memory on the FPGA.

PolyLUT's fan-in, F. This has a lookup table consumption of 256-1024× for similar accuracy, showing that PolyLUT-Add can improve model accuracy without an excessive impact on LUT size. Furthermore, it's noteworthy that the RTL Generation time cost also correlates with the lookup table size; it follows that a direct increase in fan-in would incur exponentially higher RTL Generation time costs.

We conducted additional experiments with PolyLUT-Add using the setup in Table 3.4. This utilizes lower F compared with the PolyLUT setup in Table 3.2. $A = 2$ is used for all models (which are denoted as "HDR-Add2", "JSC-XL-Add2", "JSC-M Lite-Add2", "NID-Add2"). We also reduced the layer sizes in the DNN model for the UNSW-NB15 dataset. These configurations were found to reduce area whilst maintaining comparable accuracies. Optimization of these parameters may further improve results for specific applications.

TABLE 3.4: Model setups for smaller F of PolyLUT-Add.

Dataset	Model name	Neurons per layer	β	F	D	A
MNIST	HDR-Add2	256, 100, 100, 100, 100, 10	2	4	3	2
Jet Substructure	JSC-XL-Add2[1]	128, 64, 64, 64, 5	5	2	3	2
Jet Substructure	JSC-M Lite-Add2	64, 32, 5	3	2	3	2
UNSW-NB15	NID-Add2[2]	100, 100, 50, 50, 1	2	3	1	2

1: Remarks: $\beta_i = 7$, F_i = 1
2: Remarks: $\beta_i = 1$, F_i = 6, $\beta_o = 2$, F_o = 7

TABLE 3.5: Comparison results with prior works. PolyLUT-Add uses smaller F and D (see Table 3.4), whereas PolyLUT uses larger D, F for accuracy.

Dataset	Model	Accuracy↑	LUT	FF	DSP	BRAM	F_max(MHz)↑	Latency(ns)↓
MNIST	**PolyLUT-Add (HDR-Add2, D=3)**	**96%**	**14810**	**2609**	**0**	**0**	**625**	**10**
	PolyLUT (HDR, D=4) [14]	**96%**	70673	4681	**0**	**0**	378	16
	FINN [70]	**96%**	91131	-	**0**	5	200	310
	`hls4ml` [64]	95%	260092	165513	**0**	**0**	200	190
Jet Substructure	**PolyLUT-Add (JSC-XL-Add2, D=3)**	75%	**36484**	**1209**	**0**	**0**	**315**	**16**
	PolyLUT (JSC-XL, D=4) [14]	75%	236541	2775	**0**	**0**	235	21
	Duarte *et al.* [8]	75%	88797*		954	**0**	200	75
	Fahim *et al.* [112]	**76%**	63251	4394	38	**0**	200	45
Jet Substructure	**PolyLUT-Add (JSC-M Lite-Add2, D=3)**	**72%**	**895**	**189**	**0**	**0**	**750**	**4**
	PolyLUT (JSC-M Lite, D=6) [14]	**72%**	12436	773	**0**	**0**	646	5
	LogicNets [104]	**72%**	37931	810	**0**	**0**	427	13
UNSW-NB15	**PolyLUT-Add (NID-Add2, D=1)**	**92%**	**1649**	830	**0**	**0**	**620**	**8**
	PolyLUT (NID-Lite D=4) [14]	**92%**	3336	686	**0**	**0**	529	9
	LogicNets [104]	91%	15949	1274	**0**	5	471	13
	Murovic *et al.* [113]	**92%**	17990	**0**	**0**	**0**	55	18

$*$: Paper reports "LUT+FF"

Table 3.5 shows the results and comparisons with prior works. Notably, PolyLUT applied $D = 4$ for HDR, JSC-XL and NID Lite models and $D = 6$ for the JSC-M Lite model, while PolyLUT-Add used smaller D. For comparable accuracy, the proposed PolyLUT-Add achieved a LUT reduction of 4.8×, 6.5×, 13.9× and 2.0× for the MNIST, JCS-XL, JSC-M Lite and UNSW-NB15 benchmarks respectively.

Finally, we studied latency with comparable accuracy. The removal of intermediate pipeline registers between the sub-neuron layer and its subsequent adder layers, as shown in Figure 3.5, was implemented to minimize the number of clock cycles. Compared with PolyLUT, this approach achieved a 1.6×, 1.3×, 1.2×, and 1.2× decrease for the four benchmarks, respectively. These significant reductions are attributed to lower polynomial degree D, and lower F.

3.3.4 Evaluation of LUTEnsemble

This section evaluates the performance of LUTEnsemble with MNIST classification as a case study. The model setup follows the PolyLUT work (see Table 3.2). Three configurations were tested:

(1) **PolyLUT**: This is the baseline network [14] parameterized by the width factor W, which serves to increase the number of activation units in each layer by the factor W.

(2) **PolyLUT-Add**: This is the PolyLUT-Add network [1] (introduced in the previous section) parameterized by the number of sub-neurons A.

(3) **LUTEnsemble**: This is configured with $A = 4$ (two-stage adder tree) and a smaller fan-in of $F = 5$ to control resource growth (the other models use $F = 6$ as shown in Table 3.2). This is parameterized by the ensemble size E.

Test Accuracy and Uncertainty Estimation Performance

Figure 3.10 first presents the trade-off between test error rate (1 - test accuracy) and the sum of lookup table entries for three models with a polynomial degree of $D = 2$, while varying hyper-parameters (W, A, $E \in \{2, 3, 4\}$). Notably, LUTEnsemble achieves a more efficient Pareto frontier compared to PolyLUT and PolyLUT-Add in terms of accuracy. Specifically, PolyLUT-Add has significantly larger lookup table entries as A increases. The x-axis in Figure 3.10 and the following Figure 3.11 represents the number of entries in the lookup tables, which directly correlates with the area of the synthesized hardware and impacts the timing closure after place-and-route. Therefore, it is used as an indicator of hardware overhead.

In this chapter, negative log-likelihood (NLL) is introduced as an additional evaluation metric. NLL is a common metric used for measuring the quality of in-domain uncertainty of deep learning models [45]. NLL compares the predicted probability distribution and the observed data, with lower values indicating better predictive uncertainty. DNNs often struggle to accurately quantify predictive uncertainty despite good accuracy on supervised

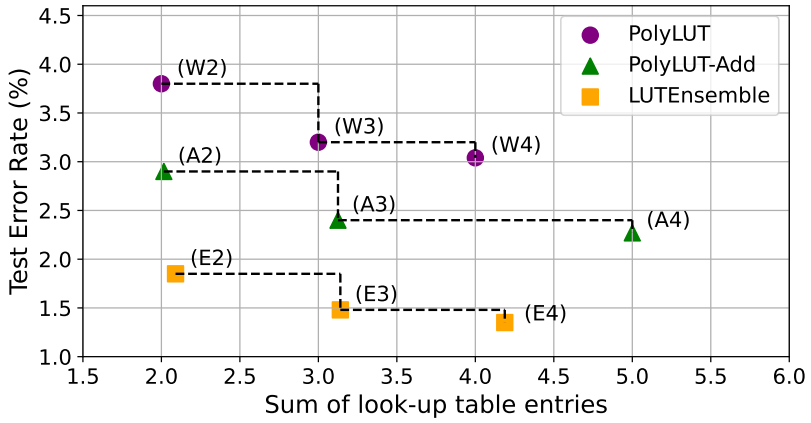


FIGURE 3.10: Results showing the trade-off between test error rate and the sum of lookup table entries for $D = 2$. The lookup table entries on the x-axis are the relative size compared to PolyLUT (HDR model in Table 3.2). The black dashed lines depict the Pareto frontier.

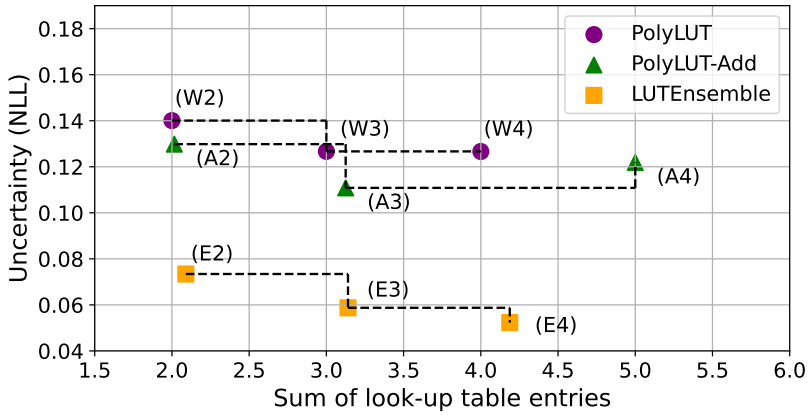


FIGURE 3.11: Results showing the trade-off between uncertainty (NLL) and the sum of lookup table entries for $D = 2$. The lookup table entries on the x-axis are the relative size compared to PolyLUT (HDR model of Table 3.2). The black dashed lines depict the Pareto frontier.

learning benchmarks. This is crucial in many mission-critical applications such as autonomous driving [115], medical diagnostics [116], and meteorological forecasting [117].

Figure 3.11 shows the results of several configurations of our networks comparing NLL and lookup table entries. From the figure, it can be clearly seen that LUTEnsemble has an improved Pareto frontier over the other models.

We further assess the uncertainty of OOD samples from classes not encountered during training. The networks are trained on its standard MNIST dataset train/test split. Besides evaluating the network on the standard MNIST test set, we also test its performance on an additional set comprising unknown classes—the test portion of the NotMNIST dataset [2]. Though the image dimensions in NotMNIST align with MNIST, its content significantly differs, featuring alphabetic characters instead of digits.

We calculate the entropy of the probability distribution over the 10 classes of MNIST and NotMNIST. Given the logits $\mathbf{z} = (z_1, z_2, \ldots, z_{10})$ from the neural network, we apply the softmax function to obtain the class probabilities $P(z_i) = e^{z_i}/(\sum_{j=1}^{10} e^{z_j})$. The entropy H is then computed using the natural logarithm (Eq. (3.1)):

$$H(\mathbf{P}) = -\sum_{i=1}^{10} P(z_i) \ln P(z_i) \tag{3.1}$$

Figure 3.12 gives the density plot of entropy values of networks tested on MNIST and NotMNIST. While the ground truth for true entropy distribution is not accessible, it anticipates that predictions for OOD samples will tend toward a uniform distribution, reflecting higher entropy due to increased uncertainty. Conversely, for known classes, the predictive distribution is expected to concentrate more on the actual targets, indicating lower entropy due to higher certainty. For known classes in Figure 3.12, PolyLUT-Add and LUTEnsemble all exhibit low entropy, aligning with expectations. For unknown classes, Figure 3.12(a) shows that the LUTEnsemble effectively handles OOD samples with apparent higher entropy. However, the OOD density peak of PolyLUT-Add in Figure 3.12(b) is still centered in the same area as the known dataset, which incidents overconfident wrong prediction. Such overconfident predictions, especially when incorrect, pose significant risks in practical applications involving a mix of known and unknown classes.

[2]Accessed from: http://yaroslavvb.blogspot.co.uk/2011/09/notmnist-dataset.html

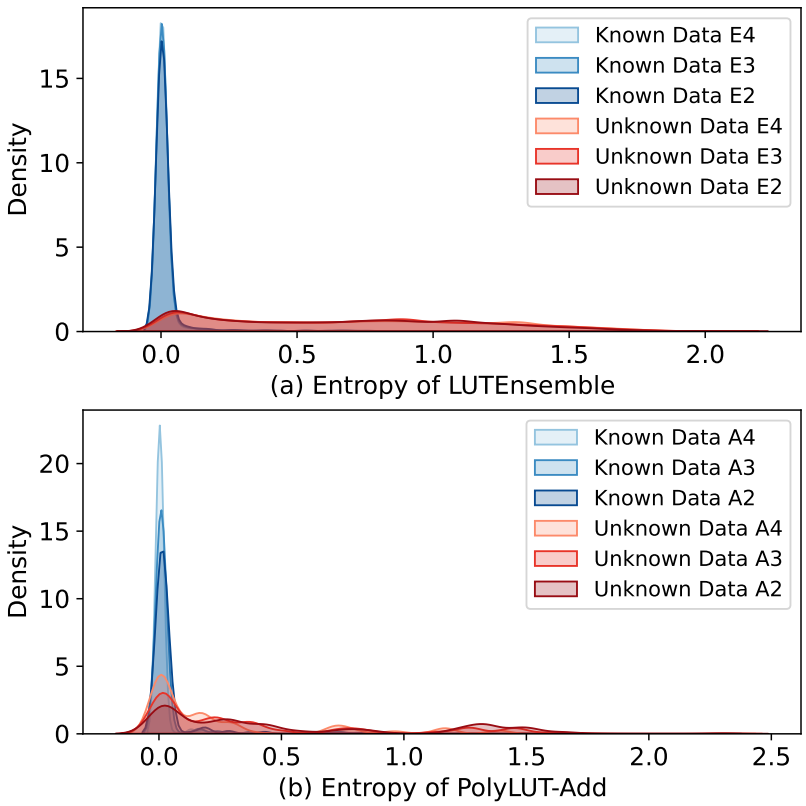


FIGURE 3.12: Density plot of entropy values of networks trained on MNIST and tested on MNIST (as the known dataset) and NotMNIST (as the out-of-distribution dataset).

Mixer Layer Performance

The input features of the JSC and NID datasets have no spatial correlation. Then, the Mixer is only tailored for image classification tasks and tested on the MNIST dataset. Table 3.6 shows the accuracy of PolyLUT and LUTEnsemble on $D = 1$ and $D = 2$ cases. The Mixer fan-in $M \in \{2, 3, 4\}$ are tested. The PolyLUT and LUTEnsemble models are tested on the HDR model in Table 3.2. As an ablation study, we also set LUTEnsemble with $A = 1$ and $E \in \{2, 3, 4\}$ to evaluate the Mixer layer's performance on different ensemble sizes.

For the base PolyLUT, compared to the setup without a Mixer layer, an accuracy improvement of 1.16% and 1.28% is achieved at the cost of only a 3.33% and 0.84% increase in lookup table entries for $D = 1$ and $D = 2$ setups, respectively. Results in the ensemble setting also

TABLE 3.6: Performance of Mixer layer. 'Base' denotes the original PolyLUT and $E \in \{2, 3, 4\}$ is configurated for LUTEnsemble models.

Models	Accuracy with D=1				Accuracy with D=2			
	w/o M	M=2	M=3	M=4	w/o M	M=2	M=3	M=4
Base	93.76	94.77	94.67	**94.95**	95.40	96.53	**96.68**	96.66
E2	95.95	96.27	96.57	**96.60**	97.24	97.25	97.77	**97.91**
E3	96.49	96.93	97.06	**97.19**	97.68	98.09	98.05	**98.23**
E4	96.92	97.07	97.34	**97.47**	97.87	98.24	98.26	**98.39**

For M=2,3,4 the lookup table entries is increased by 0.21%, 0.84%, 3.33%; the latency is increased by 1 clock cycle.

TABLE 3.7: Model setups for LUTEnsemble.

Dataset	Model name	Neurons per layer	β	F	D	A	E
MNIST	HDR-A4E2M3[1]	256, 100, 100, 100, 100, 10	2	3	3	4	2
Jet Substructure	JSC-HC-A4E2[2]	128, 64, 64, 64, 5	3	2	3	4	2

1: Remarks: $M = 3$, LUT in the last two layers (100,10) are combined in single pipeline registers.
2: Remarks: $\beta_{input} = 9$, $F_{input} = 1$, $A_{input} = 1$.

show an improvement in accuracy, which increases with the ensemble size. Moreover, as expected, a higher M results in better accuracy in most cases.

Overall Comparison with Related Works

We conducted additional experiments with LUTEnsemble using the configuration in Table 3.7. The frequency and the area are collected from the Vivado post Place & Route timing and resource report. Latency is obtained by multiplying the depth of the model's pipeline registers by frequency. An adder tree structure with $A = 4$ (two-stage adder tree) and an ensemble of $E = 2$ are used for all models. In particular, the Mixer fan-in $M = 3$ is applied to the MNIST task. We donate the model names "HDR-A4E2M3" and "JSC-HC-A4E2" for MNIST and Jet Substructure classification (high accuracy) benchmarks. Notably, even though this LUTEnsemble model setup requires three combinational logic stages (one sub-neuron layer and two adder layers) in the datapath as shown in Figure 3.4, they are implemented in the same pipeline stage and do not introduce additional clock cycles. Moreover, to compensate for the 1 extra clock cycle of the Mixer layer, the LUT logic in the last two layers (the last hidden layer and the output layer) is also combined between the same pipeline registers.

TABLE 3.8: Performance comparison with prior works.

Dataset	Model	Accuracy↑	LUT	FF	DSP	BRAM	F_max(MHz)↑	Latency(ns)↓
MNIST	**LUTEnsemble (HDR-A4E2M3, D=3)**	**98%**	26225	8319	**0**	**0**	468	13
	PolyLUT-Add (HDR-Add2, D=3) [1]	96%	**14810**	**2609**	**0**	**0**	**625**	**10**
	NeuraLUT (HDR-5L) [106]	96%	54798	3757	**0**	**0**	431	12
	PolyLUT (HDR, D=4) [14]	96%	70673	4681	**0**	**0**	378	16
	FINN [70]	96%	91131	-	**0**	5	200	310
	`hls4ml` [64]	95%	260092	165513	**0**	**0**	200	190
Jet Substructure	**LUTEnsemble (JSC-HC-A4E2, D=3)**	75%	**18810**	1906	**0**	**0**	**450**	**11**
	PolyLUT-Add (JSC-XL-Add2, D=3) [1]	75%	36484	**1209**	**0**	**0**	315	16
	NeuraLUT (JSC-5L) [106]	75%	92357	4885	**0**	**0**	368	14
	PolyLUT (JSC-XL, D=4) [14]	75%	236541	2775	**0**	**0**	235	21
	Duarte *et al.* [8]	75%	88797*		954	**0**	200	75
	Fahim *et al.* [112]	**76%**	63251	4394	38	**0**	200	45

$*$: Paper reports "LUT+FF"

These configurations were found to reduce area while maintaining comparable accuracies. Optimization of these parameters may further improve results for specific applications.

Table 3.8 shows the results and comparisons with prior works. Notably, PolyLUT applied $D = 4$ for MNIST, Jet Substructure classification model; we maintain smaller $D = 3$, which is the same as PolyLUT-Add.

In MNIST experiments, the proposed LUTEnsemble achieved 2% accuracy improvements compared with related works, followed by a LUT reduction of 2.7× and latency reduction of 1.2× compared with PolyLUT. The increased flip-flop (FF) consumption is due to the increased pipeline registers required by the higher fan-in policy, which is not the bottleneck of the target FPGA, consuming only 0.35% of the total 2364480 FFs.

For similar accuracy on the Jet Substructure classification, LUTEnsemble achieves the best balance between accuracy, LUT, and latency over others, in particular, compared with PolyLUT, a reduction of $12.6\times$ and $1.9\times$ decrease in LUT and latency, respectively.

3.4 Summary

In this chapter, we introduce PolyLUT-Add and its enhancement, LUTEnsemble. PolyLUT-Add first demonstrates enhanced neuron connectivity in PolyLUT by using an additional lookup table block to combine A PolyLUT sub-neurons. It serves as a preliminary study and a special case of LUTEnsemble, which includes more comprehensive optimizations. These techniques mitigate scalability issues associated with conventional implementations and significantly improve efficiency.

Specifically, we demonstrate that by utilizing a configuration of $A = 2$, PolyLUT-Add with a lower polynomial degree D and fan-in F is sufficient to achieve comparable accuracy to PolyLUT. On the MNIST, Jet Substructure Classification, and Network Intrusion Detection benchmarks, PolyLUT-Add reduced LUT consumption by factors of 2.0-13.9 with a 1.2-1.6 times decrease in latency.

For LUTEnsemble, we further demonstrate that it achieves a LUT reduction of 2.7 times and a 1.2 times decrease in latency for a 2% accuracy improvement on MNIST compared to PolyLUT. For the Jet Substructure Classification, LUTEnsemble achieves similar accuracy with a LUT reduction of 12.6 times and a 1.9 times decrease in latency. Finally, we show that LUTEnsemble excels in uncertainty estimation, achieving superior negative log-likelihood performance.

CHAPTER 4

Low-latency Neural Network Qubit Detection System in Trapped-Ion Quantum Computing

---

This chapter focuses on a practical application: qubit state measurements in trapped-ion quantum information processing. It explores how FPGAs and integrated neural networks can address the challenges in this field by achieving low latency while maintaining high accuracy. The LUTEnsemble proposed in Chapter 3 is applied as a low-latency MLP solution, and a Vision Transformer (ViT) DNN accelerator is proposed in this chapter for higher accuracy. The content of this chapter builds upon the background of qubit detection in Chapter 2.

Quantum algorithms have the potential for new and highly efficient processing of large data, including super-polynomial speedup of search [118], cryptography [119], and random walk [120] problems, along with many others [121]. However, the implementation of the apparatus itself is challenging [122]. Trapped ion quantum computers are one of several promising platforms [123, 124], where in a typical blade-style Paul trap, individual atoms are held in space by radio frequency(RF) electric fields [125]. When ionized, they behave like hydrogenic atoms, with energy levels like those for Ytterbium shown in Figure 2.11. The qubit state of each atom is the electronic sublevel of the valence electron. Tight confinement of ions in the chain also experiences Coulomb repulsion, and exchange motional energy to form the multi-qubit operations required to produce entangled states [126].

Qubit state measurements are an integral part of quantum information processing. With trapped ions, this can be performed via a state-dependent fluorescence technique [123, 124]. A laser excites a transition and induces photon scattering if the qubit is in the $|1\rangle$ state; however, the laser is off-resonant from any transition if the qubit is in state $|0\rangle$ and no photons

are scattered. Measuring the number of photons registered on an Electron-Multiplying Charge-Coupled Device (EMCCD) camera provides a way to determine the qubit state, and this technique has the ability to spatially resolve multiple trapped ions [127, 99, 128].

However, quantum systems are highly susceptible to errors from decoherence [87, 88], and the error correction system must detect and correct them before they propagate and accumulate above a certain threshold. This is a key ingredient to realizing a fault-tolerant quantum computer [89, 90] and involves error syndrome measurements of qubits followed by in-sequence, conditional corrective operations. For these strategies to be effective, the underlying qubit state measurements must be accurate to ensure the correct identification of errors and rapid enough to outpace the decoherence times of qubits. Another reason for driving fast qubit detection is that classification latency should be minimized so as not to limit the quantum computer's total clock speed.

This requirement imposes two challenges for qubit measurement per single-shot operation: high fidelity and low latency.

Our study addresses these challenges by implementing the LUTEnsemble and Vision Transformer (ViT [18, 19]) on a customized FPGA. In the literature, a reasonable qubit detection duration is at the microsecond (μs) level, while the coherence time could extend to the second (s) level [95, 96]. Table 4.1 presents the qubit detection latency in modern trapped-ion computers, with reports ranging from 120-400 μs for different setups. In particular, Ref. [94] reported a total time of 133.35 ms for an 'N=6 QV Circuit' example, with qubit detection consuming 120 μs.

By utilizing the LUTEnsemble model as the qubit detection method, its mean measurement fidelity (MMF) error on three-qubit (2.7%) demonstrates a 4.22$\times$ improvement over thresholding. The 24 ns achievable detection latency is ultra-low and more than sufficient for modern detection latency, as reported in Table 4.1.

For a better-balanced trade-off between latency and fidelity, the ViT model was designed to demonstrate further improved fidelity over thresholding, LUTEnsemble, and convolutional neural network (CNN) approaches. For the three-qubit test, the mean measurement fidelity

TABLE 4.1: Related works with reported qubit detection latency

Reference	Ion-trap Experiment	Measurement	Detector	readout latency∗	detection latency	Total latency
Myerson *et al.* [91]	Ca^+	PMT	Maximum Likelihood	-	145 μs	-
Halama *et al.* [92]	Be^+	EM-CCD	Threshold	2 ms	400 μs	-
Burrell *et al.* [93]	Ca^+	EM-CCD	Threshold	-	400 μs	-
Pino *et al.* [94]	$Yb^+ - Ba^+$	Q-CCD	Threshold	-	120 μs	133.35 ms
Our work	Yb^+	EM-CCD	LUTEnsemble	1.25 ms	24 ns	-
			Threshold/ViT	1.25 ms	16-40 μs	-

∗: The readout latency includes frame grabber latency and cameralink data movement latency.

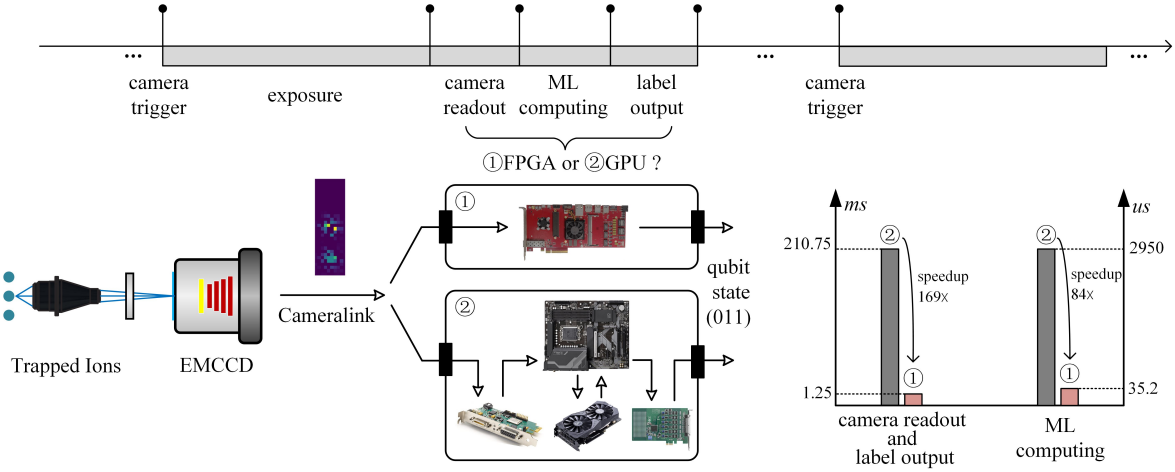


FIGURE 4.1: In the block diagram of ML-aided qubit detection in a trapped ion system, photons scattered by the ions are collected and magnified by optical elements before being directed toward an EMCCD camera. The resulting images are transmitted to the acceleration hardware, such as an FPGA or GPU, via the Cameralink protocol, with the GPU serving as the baseline. The right bar chart illustrates the speedup of the ViT model achieved on a 3-qubit test. The detailed implementation for FPGA and GPU are introduced in Figure 4.2 and Figure 4.10 respectively.

(MMF) error (1.5%) is $7.60\times$, $1.80\times$, and $1.13\times$ lower than thresholding, LUTEnsemble and CNN, respectively. Moreover, the number of parameters in the ViT is considerably smaller than in the CNN. The proposed ViT-aided solution achieves a measured detection latency of 16-40 μs, which is comparable to modern trapped-ion systems. The following chapter primarily focuses on ViT, with the LUTEnsemble discussed as a baseline in the experiment section.

Figure 4.1 shows the overview of this work with ① representing the proposed FPGA qubit detection system on the ViT model and ② as its functionality-equivalent GPU baseline. The $^{171}Yb^{+}$ ions are used in our trapped ion experiment, with its scattered photons collected by an EMCCD camera. Given that the typical EMCCD cameras (*e.g.*, Andor iXon 897 [97]) output an image through the Cameralink protocol [86] and acts as a serializer, the FPGA detailed in Figure 4.2 serves as an integrated System-on-chip (SoC), encapsulating a well-defined workflow to (1) deserialize the image from cameralink, (2) process the image classification in a ViT accelerator, (3) output results (qubit states) to the next stage. Specifically, (1) and (2) are operated in a pipelined manner through an AXI-Stream (AXIS) bus [129] for efficient data processing. A Low voltage Complementary Metal Oxide Semiconductor (LVCMOS) Input/Output (IO) interface is used in step (3) to minimize output latency.

To the best of our knowledge, the work of Jeong *et al.* [85] is the closest to ours. It applied a residual network (ResNet)-based CNN model to improve 4-qubit state measurements using an EMCCD and achieved a fidelity error reduction of $\sim$50% compared to the maximum likelihood method [101]. Furthermore, their work demonstrated that a machine-learning model can improve qubit detection accuracy on real experimental images. However, their analysis was done offline and did not consider latency. This limits its suitability for fast quantum error correction. This motivates us to extend their study in two ways: (a). use a state-of-the-art ViT model to pursue further accuracy improvement; (b). integrate the qubit detection system with a ViT accelerator on an FPGA for low-latency implementation in real-time.

The main contributions of this chapter are:

- To the best of our knowledge, our work is the first machine learning-aided trapped-ion detection system that is embedded in a low-latency FPGA control system and uses 2D camera input images.
- We applied the LUTEnsemble proposed in Chapter 3 to provide an ultra-low latency qubit detection solution and then proposed an innovative hardware-efficient ViT architecture with a customizable accuracy-latency tradeoff facilitated by the $hls4ml$ framework.

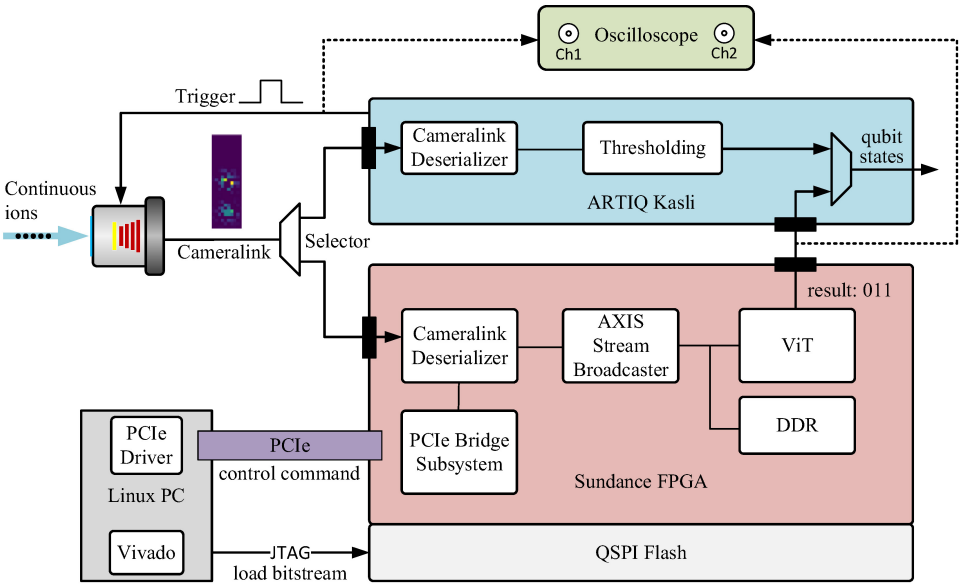


FIGURE 4.2: This figure demonstrates the framework of our qubit detection system. The camera is controlled by an ARTIQ system [130], with a master Kasli controller [131] that communicates with sub-modules, such as a DIO card, to trigger the camera and receive qubit detection results from the Sundance FPGA. Kasli also carries a traditional qubit classifier based on thresholding [132] inside it, which will serve as a baseline for traditional models in the subsequent experimental section for comparison with our proposed ML scheme, which is implemented in the Sundance FPGA. By switching the Selector to connect the camera to Sundance, the ion image is transferred from the camera to FPGA through cameralink protocol, followed by the cameralink deserializer to decode the data packet to the ViT model for real-time classification. The FPGA is controlled by a PCIe driver designed in Linux PC, and a Xilinx Vivado [108] is used to load bitstream to the QSPI Flash on the FPGA board.

- Using the synthetic dataset that closely resembles the statistics of experimented data, the LUTEnsemble and ViT achieve accuracies that are 2.9-8.7% and 3.1-9.9% greater than existing thresholding methods, respectively.
- Experimentally validated latency measurements of ViT solution show a substantial speed-up; the total detection latency for one- and three-qubit tests on the FPGA was 1.78$\pm$0.02 ms and 2.29$\pm$0.02 ms, a 119$\times$ and 94$\times$ reduction over the GPU baseline system (211.95$\pm$17.72 ms and 214.70$\pm$10.87 ms). Moreover, the ViT latency accounts for only 0.89-1.53% of the total detection latency, so the bottleneck

now lies in the EMCCD and transmission via Cameralink. Thus, this work could help improve the performance of next-generation qubit detection systems.

4.1 Design

4.1.1 FPGA-based Control System

As depicted in Figure 4.2, the ion-trap experiment and the camera are controlled by an Advanced Real-Time Infrastructure for Quantum Physics (ARTIQ) system, a leading control system for quantum information experiments [130]. The ARTIQ hardware consists of a master controller (Kasli [131]) that communicates with sub-modules, such as a Digital IO (DIO) card to trigger the camera and a frame-grabber card for image data transfer. A thresholding method [132] is also implemented on the Kasli as a conventional qubit detection baseline. As Kasli is not designed for large-scale computational tasks, and does not have sufficient logic resources for the ML accelerator, an auxiliary Sundance FPGA board [133] is employed as our cameralink interface and ViT implementation platform.

The EMCCD camera is configured prior to experiments. Trapped ion fluorescence captured on this camera is sent to an FPGA through the Cameralink protocol, which has a low-latency interface and simple protocol for rapid decoding of the data. [86]. Inside Sundance FPGA, a Deserializer block, provided by Sundance [133], deserializes the Low Voltage Differential Signalling (LVDS) inputs of the Cameralink interface. On the output side of the IP, the reconstructed data after deserialization is composed of two parallel dataflows, achieved by an AXIS Broadcaster [129]. The first dataflow connects the Cameralink Deserializer output to Double Data Rate (DDR) memory for buffering of the captured images. The second dataflow is connected to the ViT for real-time classification. To achieve the lowest latency transfer to ARTIQ Kasli, we use a direct LVCMOS IO interface to send out the classification results.

Notably, programming the Cameralink Deserializer is necessary to configure the resolution and Cameralink mode (Base, Medium, or Full) [86]. For easy-to-use programming, we therefore design a Peripheral Component Interconnect express (PCIe) Driver in a Linux PC

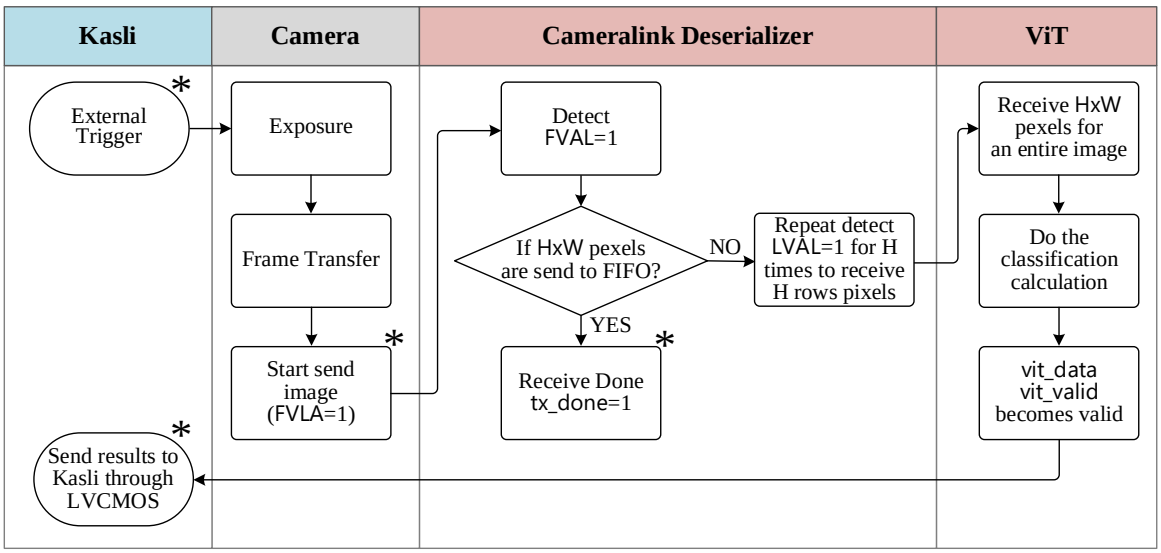


FIGURE 4.3: This figure describes the flow chart of the proposed FPGA-based qubit detection system. The steps with attached $*$ symbolize that probe signals are available in this step for latency measurements. The detailed timing diagram of `FVAL`, `LVAL`, `tx_done` and `vit_valid`, `vit_data` are illustraed in Figure 4.4. The `FIFO` structure is discussed in Figure 4.5

that passes configuration commands to the Cameralink Deserializer through a PCIe interface (see Figure 4.2). Moreover, we utilize an off-chip Quad serial peripheral interface (QSPI) Flash chip to store the FPGA programming file (bitstream), thereby establishing a boot sequence as follows: (a). the Linux PC powers up, (b). the FPGA automatically loads the bitstream from the QSPI Flash, (c). the PC then detects and initializes the PCIe peripheral, and (d). the user can send control commands to configure the FPGA through our PCIe driver. The FPGA only needs to be reloaded with new bitstreams from Vivado via the joint test action group (JTAG) interface when internal logic modifications are required, *e.g.*, an updated ViT accelerator.

4.1.2 Data Flow and Timing Measurements

Following the framework in Figure 4.2, the flow chart of the proposed FPGA qubit system is shown in Figure 4.3. Kasli starts the camera with an external trigger, and then ions' fluorescence is collected by the exposed sensor. The 'Frame Transfer' block transfers the collected analog image to a digital format. The resolution of a single digital 2D-image is

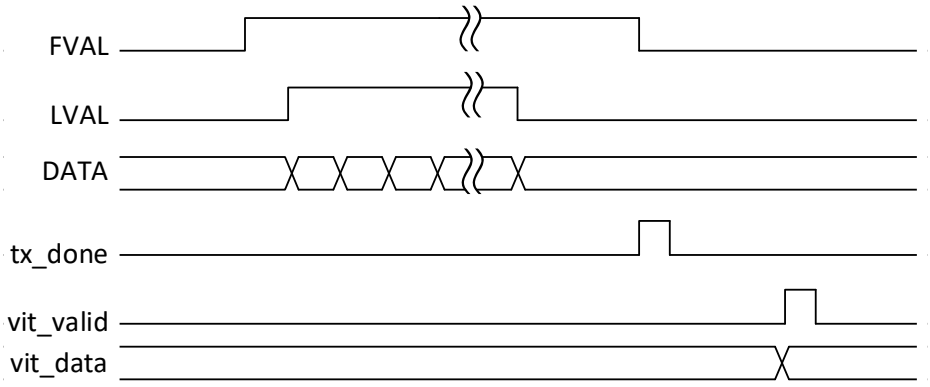


FIGURE 4.4: Image readout and detection signal timing. We refer the reader to Ref. [86] for a specialized description of the CameraLink Protocol. Here, our focus is mainly on the relationships between `FVAL`, `tx_done`, and the `vit` output, with other signal details being omitted for simplicity.

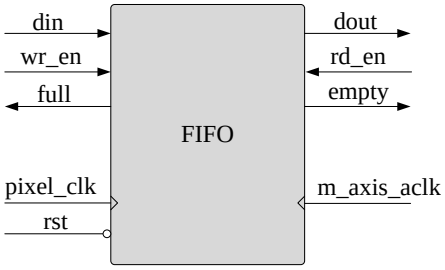


FIGURE 4.5: FIFO of Cameralink Deserializer. It serves as a buffer to transfer data from the cameralink interface domain (works in the `pixel_clk`) to the AXIS bus domain works in a different clock source: `m_axis_aclk`). The `wr_en` and `rd_en` are 'write enable' and 'read enable' signals, respectively, with full and empty indicating the FIFO's capacity status.

defined as $(H \times W)$, where H and W are the height and width respectively, so $(H \times W)$ pixels are streamed to the FPGA using the cameralink protocol. The Cameralink deserializer on the FPGA receives the pixels sequentially via an internal first-in, first-out (FIFO) memory. The FIFO implements a double-buffered streaming approach to ensure continuous data flow for processing $(H \times W)$-sized images. Specifically, by the time the final pixel of an image is written into the FIFO's read buffer, the preceding $(H \times W - 1)$ pixels have already been seamlessly transferred from the FIFO's send buffer into the internal cache of the ViT accelerator, ensuring no delay in data availability for processing. Finally, after an entire image is assembled in the ViT accelerator, it starts to process the classification computation, and

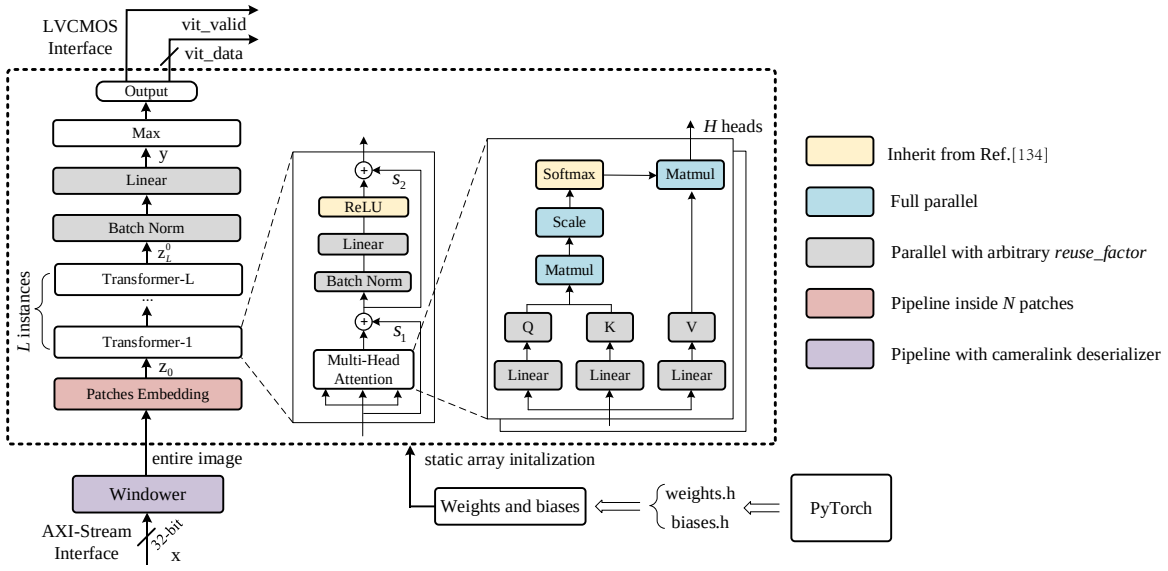


FIGURE 4.6: Hardware optimization for each component inside of the ViT

when the computation is completed, its output is placed in `vit_data` and the `vit_valid` signal is set to `HIGH`. This forms the interface with the Kasli controller.

A detailed timing diagram of the Cameralink image acquisition and ViT data output is given in Figure 4.4, where Frame Valid (`FVAL`) is `HIGH` for a valid frame and Line Valid (`LVAL`) is `HIGH` for a valid line. Both `FVAL`, `LVAL`, and `DATA` are provided by the EMCCD camera, which acts as the master. The first rising edge of the `FVAL` signal represents the start of the image data transmission to the FPGA. The `tx_done` is an internal status signal within the Cameralink deserializer and is calculated by comparing the received pixel counter with the predefined image resolution. A `tx_done` signal of `HIGH` indicates that the last pixel of an image has been completely received and stored in the FIFO (as shown in Figure 4.5).

In order to evaluate the latency of the entire system, `FVAL`, `tx_done`, and `vit_valid` are also connected to the FPGA's output IO as debug signals so that their timing can be measured on an external oscilloscope (steps that can be directly probed are highlighted with a $*$ in Figure 4.3).

Our qubit detection system is highly scalable due to a clear division between logical processing, handled by ARTIQ, and hardware acceleration via the Sundance FPGA. Its modular design

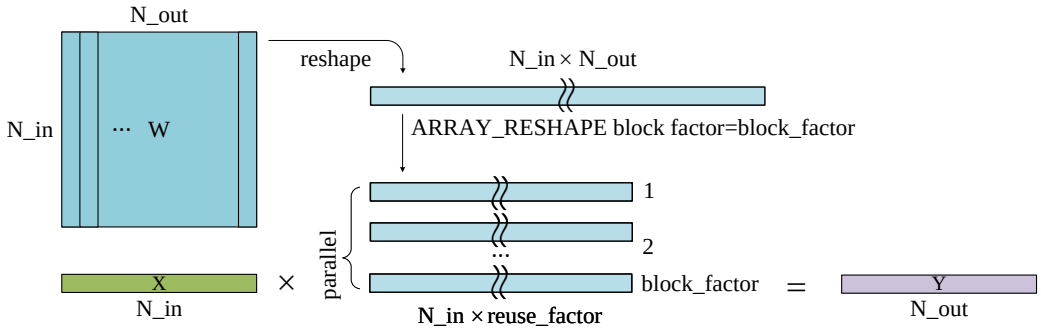


FIGURE 4.7: Resource-efficient matrix multiplication. The operation in the example is $\mathbf{Y} = \mathbf{X} \cdot \mathbf{W}$, with user-defined `reuse_factor`, `block_factor` is calculated by $N\_in \times N\_out$/`reuse_factor`.

allows each component to function independently, enabling the optimization or replacement of individual parts without affecting the system's overall performance.

4.1.3 Vision Transformer Accelerator

Our proposed ViT model builds upon prior Transformer work by Ref. [134] for jet-tagging physics experiments with various simplifications and optimizations that were added to make it practical for the FPGAs. These simplifications are: (1) transformer size, *i.e.*, the number of transformer layers is reduced to 1, and self-attention latent dimensions are reduced from 128 to 16, (2) the original SiLU activation function was simplified to ReLU to allow for easier FPGA mapping, and (3) layer normalization is replaced by batch normalization.

We further upgrade this Transformer block to support our qubit detection vision task, that is, turn it into a Vision Transformer as formulated in Chapter 2.

4.1.4 Implementation in Vivado HLS

In contrast to the standard approach of directly designing our ML model in a hardware description language (HDL) such as VHDL or Verilog, high-level synthesis from an equivalent C program was used. This approach has advantages for design productivity and verification. The C program transforms from its equivalent PyTorch Python model using the Xilinx Vivado

high-level synthesis (HLS) tool (v2020.1) [135] and then encapsulated into a single HDL program intellectual property (IP) block. Post-synthesis verification between the generated HDL program and the source C program is conducted to guarantee functional consistency. Finally, this HDL IP then passes through Xilinx Vivado Design Suite (v2020.1) for system integration and bitstream generation [108].

Deploying ViT on edge devices requires extended parallelism within the constraints of limited resource usage. The implementation in Ref. [134] fully partitioned all matrix elements and parallelized all `for`-loops. This approach is suitable for data center applications on large FPGAs (an XCU250 FPGA) but overly aggressive for edge devices. We adopted a matrix operation accelerator to achieve a favorable resource and latency tread-off on our Sundance FPGA platform.

Figure 4.7 provides an example of our approach to parameterize the degree of parallelism degree for the matrix-vector multiplication $\mathbf{X} \cdot \mathbf{W} = \mathbf{Y}$. First, the weights $\mathbf{W}$ of size (N_in, N_out) are flattened in a column-major order to obtain a one-dimensional vector with length $N\_in \times N\_out$. Subsequently, the `HLS ARRAY_RESHAPE` pragma is used in Xilinx Vivado Tool [135] to reshape the matrix in a block factor mode, where the `block_factor` is calculated by dividing $N\_in \times N\_out$ by a user-defined `reuse_factor` (the `reuse_factor` is illustrated in Figure 4.8 and should be chosen as a number that can be evenly divided [136]). The `reuse_factor` ranges from 1 to N_in, hence the `block_factor` ranges from $N\_in \times N\_out$ to N_out. After the `block_factor` division, the weights $\mathbf{W}$ become independent storage units, and parallelism is achieved through the `HLS UNROLL` pragma. By applying this approach, we can more effectively manage the trade-off between computational latency and resource usage.

Figure 4.6 illustrates the ViT accelerator. We initially implemented the model using PyTorch and trained the network on a graphics processing unit (GPU). The trained weights and biases are converted to static arrays in C. These are then mapped to on-chip memory and placed onto FPGA hardware via the HLS tool. The next stage of this design is the Windower, which accepts pixel samples from the input AXI-stream interface and uses a shift register to assemble a full image of past inputs to the next stage. Pipeline data transfer is adopted

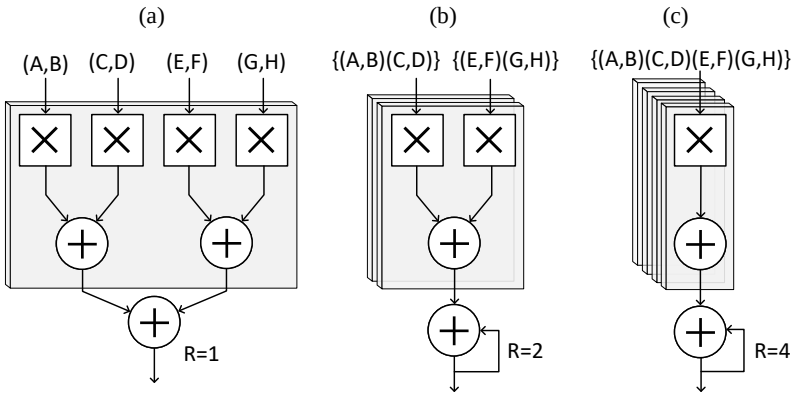


FIGURE 4.8: The reuse factor is denoted as R in this example, which determines the level of parallel processing. (a). $R = 1$, the process is fully parallelized, allowing all multiplication tasks to be carried out at once with the greatest number of multipliers. (b). $R = 2$, $\frac{1}{R}$ of the tasks are processed simultaneously and half of the resources are time-division multiplexed. (c). $R = 4$, 4 multipliers are time-division multiplexed, turning the computation into sequential execution.

between the Windower and cameralink deserializer. The different layers of the ViT are shown in different colors, corresponding to the type of parallelism employed in our implementation. The Patches Embedding marked in red uses a pipelining scheme over the N-sized patches. The modules with high resource consumption and latency, marked in grey, all utilized an arbitrary `reuse_factor` to allow their performance to be controlled.

4.2 Experimental Setup

4.2.1 Trap ion and EMCCD

The system is designed for a macroscopic radio-frequency Paul trap, using $^{171}\mathrm{Yb}^{+}$ as the trapped ion set. This was chosen for its qubit properties and experimental ease-of-use [137]. In such an experiment, the $|0\rangle$ and $|1\rangle$ qubit states are encoded in the $|^{2}\mathrm{S}_{1/2}, \mathrm{F} = 0\rangle$ and $|^{2}\mathrm{S}_{1/2}, \mathrm{F} = 1\rangle$ energy levels of the valence electron, respectively (see Figure 2.11).

During experiments, the qubit is initialized into a known state $|0\rangle$. Following this, the qubit is manipulated by driving transitions between the $|0\rangle$ and $|1\rangle$ states that are separated by 12.6

GHz. In practice, this can be achieved by driving microwave or laser fields. Precise control of the applied electromagnetic field's parameters enables arbitrary operations on qubits, allowing for single and multi-qubit gates that are necessary for quantum computations [125].

After computations, the state of the qubit is measured using state-dependent fluorescence by scattering photons. The ion is illuminated with a 369.5 nm detection laser beam that drives the $|^2\mathrm{S}_{1/2}, \mathrm{F} = 1\rangle$ to $|^2\mathrm{P}_{1/2}, \mathrm{F} = 0\rangle$ transition (see Figure 2.11). If the qubit is in the $|1\rangle$ state, the ion will interact with the detection beam and fluoresce. However, if the qubit is in the $|0\rangle$ state, the ion scatters the minimal number of photons since the detection beam is far off-resonant. The state of the qubit can, therefore, be inferred from the number of emitted photons.

In this work, an Andor 987 EMCCD camera [97] is employed to collect the fluorescence from $^{171}\mathrm{Yb}^+$ ions and then converts the photons into a digital signal. The ARTIQ system controls the EMCCD camera with a Kasli Controller v1.1 [130, 131].

4.2.2 Latency Measurement Setup

We compare latency performance using three different setups (see Figure 4.9):
(a). Thresholding implemented on Kasli.
(b). ViT implemented on FPGA.
(c). ViT implemented on GPU.

In the baseline system, where the thresholding is an integral part of the physical trapped-ion system, the EMCCD image is transferred to Kasli via Cameralink, and the frame-grabber performs region-of-interest (ROI) summing. The total photon counts are then transferred to the Kasli, which performs thresholding. Figure 2.13 explicit the thresholding methods.

In contrast, the proposed FPGA qubit detection system uses the Sundance SOLAR EXPRESS 120 (SE120) containing a Xilinx XCZU11EG chip on which the ViT accelerator is implemented. Images from the EMCCD camera are captured using a Cameralink FPGA Mezzanine Card Module [133].

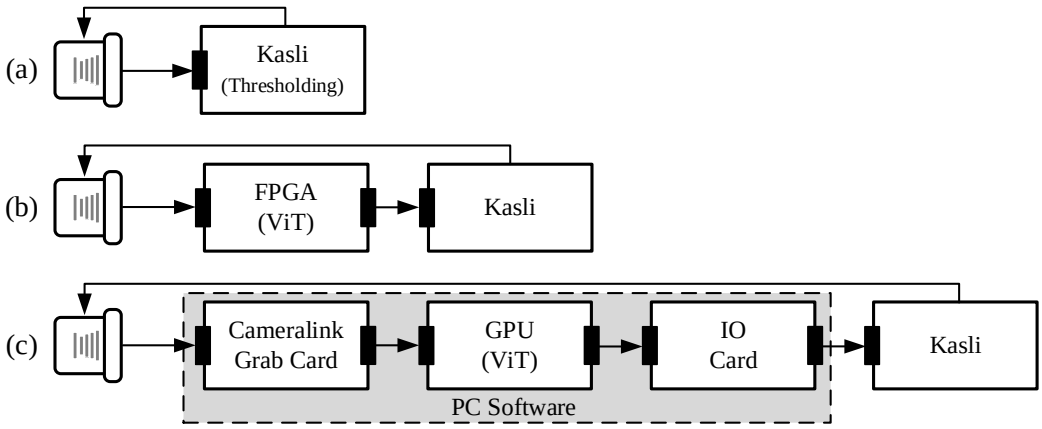


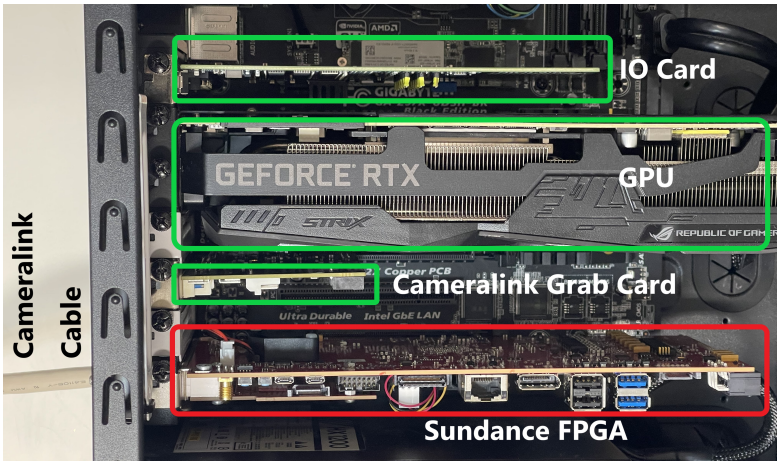
FIGURE 4.9: This figure shows three different experiment setups. (a) is the Thresholding method implemented on Kasli. (b) is our FPGA solution with ViT as the classifier. (c) is the GPU solution; it employs 3 COTS cards with the cameralink crab card to acquire cameralink images, a GPU for accelerating ViT inference, and a COTS GPIO card for ViT label outputs. The cameralink grab card and IO card are both PCIe-rooted for low latency. These three COTS are linked together through PC software: C program for cameralink grab and IO cards, Python with Pytorch for ViT on GPU.

The accelerator platform in the machine learning domain can usually be either FPGA or GPU. In order to fairly validate the latency speed-up of our proposed FPGA system, we also create a GPU acceleration solution for the qubit detection task.

Commercial GPU products do not have a cameralink interface or a user-programable IO interface. To allow the GPU to process image data from the EMCCD and send out classification results to the Kasli controller in the lowest latency, we assembled the 3-card solution in Figure 4.9 (c).

Figure 4.10 demonstrates the board's setups and datapath in our GPU and FPGA experiments. The GPU and FPGA tests are on the same PC mainboard for consistent evaluation; the Cameralink Calble is either connected to the GPU or FPGA setups for independent tests.

We use a commercial off-the-shelf (COTS) card to receive cameralink images, PyTorch with GPU for ML inference, and a COTS GPIO card for label outputs. Teledyne Dalsa Aquarius Base CL x1 is used as the cameralink grab card [138], an NVIDIA GeForce RTX 2080 Ti GPU is located in the middle, followed by the Advantech PCIE-1751-B card as the GPIO card [139].



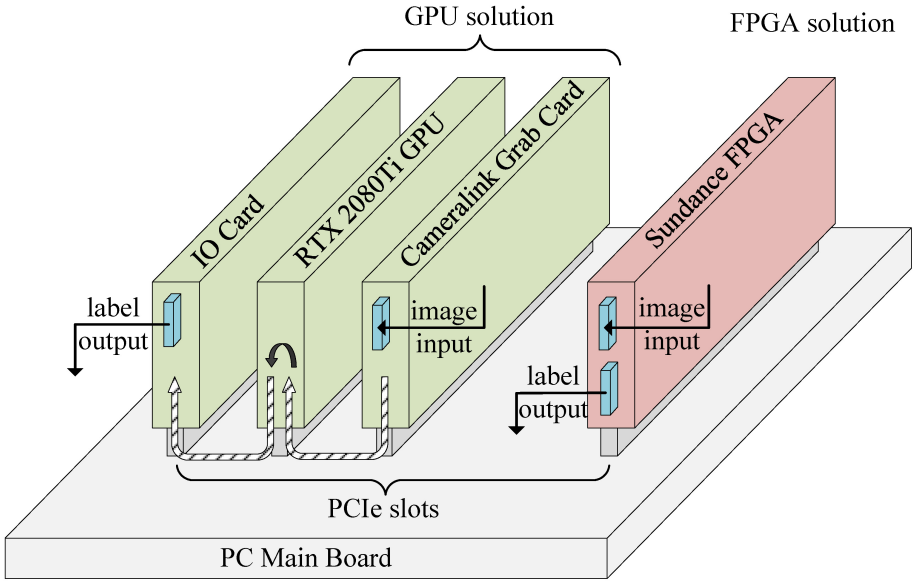


FIGURE 4.10: The GPU and FPGA solution are mounted in the same PC main board through the PCIe interface. The former employs 3 COTS cards with the cameralink crab card to acquire cameralink images, a GPU for accelerating ML inference, and a COTS GPIO card for ML label outputs. The dashed arrow indicates the data flow. The Sundance FPGA is only powered and controlled by the PC Mainboard; no data transmission is issued between them.

All three cards equip the PCIe point-to-point host interface, which allows simultaneous image acquisition, ML inference, and label output with little intervention from the host CPU. These three cards are mounted in a host PC main board with Intel(R) Core(TM) i7-4790 @3.6GHz and 16GB memory.

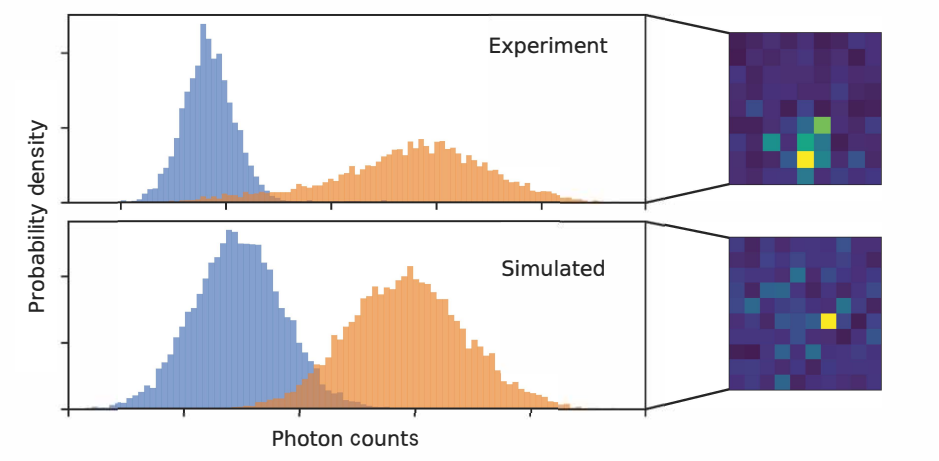


FIGURE 4.11: Comparison of experimental and simulated single ion images. Insets show an example of a simulated and experimental image of a single ion. The simulated dataset is generated such that the resulting histograms closely resemble the statistics of the experimental data.

4.2.3 Image dataset

The neural networks are trained and evaluated on a simulated dataset of ion images. Images are generated in three steps. First, the ideal ion fluorescence is modeled by a Gaussian distribution with standard deviation $\sigma_x = \sigma_y = 0.4$ pixels and an amplitude of 35. Second, multiplicative Poissonian noise is added to the ion image with $\lambda = 0.5$. Finally, Gaussian background noise is added to the entire image with mean $\mu_{bg} = 50$ and standard deviation $\sigma_{bg} = 20$. As shown in Figure 4.11, all values above were chosen such that the statistics of the resulting histograms in simulated data correspond to experimentally calibrated values. We further verified that a Gaussian could approximate the spatial distribution of the ions' fluorescence by investigating experimental images.

A close match between this approach to simulating experimental data alongside the error reduction achieved by using an ML model on actual experimental images has been shown in Ref. [85]. In this work, we focus on minimizing and measuring latency, with ML re-training/testing on real images from a physical ion trap experiment left as future work.

Two datasets of 100,000 (a realistic size) images each are generated for 1 ion (10×10) and 3 ions (12×24). In the latter case, the ions are separated by 5 pixels, reflecting typical spacings

in experiments. The simulated images are divided into training and testing datasets with a ratio of 9:1. Moreover, the imager in the qubit detection system can collect a single image in 10 ms, which guarantees the collection of the dataset for such size practical.

4.2.4 ViT and LUTEnsemble Model configuration

The ViT classifier's model size is ($L = 1$, $H = 8$, $D = 16$), where L is the number of transformer layers, H is the heads number, and D is the self-attention latent dimensions. The patch size is set to $P = 5$ for 1-qubit classification and $P = 6$ for 3-qubit classification. We use a batch size of 128 and a learning rate $lr = 0.05$ for the SGD optimiser [23] with a Cross Entropy Loss as its loss function [23]. The ViT is trained and tested using Pytorch [23]. This ViT and its set of hyper-parameters provided good detection performance while still producing an efficient hardware implementation. Further optimization of these hyper-parameters may result in an improved solution for specific system requirements.

With low-precision, fixed-point representation has shown its advantages over floating-point computations in terms of operational complexity and latency due to the absence of complex mantissa and exponent conversions [140], the `ap_fixed`$\langle N, I \rangle$ type available in Vivado HLS [135] is used for all non-integer operations in our ViT design. This format has integer and fraction fields that use I and F bits, respectively, with $N = I + F$, where I determines the range of numbers that can be represented and F controls the precision. The reported results were achieved using `ap_fixed`$\langle 16, 8 \rangle$ for ViT and CNN parameters.

We configure LUTEnsemble with the previously evaluated model setup (HDR-A4E2M3, D=3) with the best accuracy and area performance for the MNIST classification task in Chapter 3 to the following qubit detection task evaluation.

TABLE 4.2: Comparison of three detection models in accuracy and parameters

Ion Number	Module	Error (%)↓	Parameters↓
1-Qubit (10×10)	Threshold	3.7	-
	LUTEnsemble	0.8	9212
	CNN	0.7	271666
	ViT	**0.6**	6704
3-Qubit (12×24)	Threshold	11.4	-
	LUTEnsemble	2.7	9296
	CNN	1.7	272056
	ViT	**1.5**	7622

4.3 Results

4.3.1 Accuracy Comparison

In this section, we evaluate the accuracy achieved by our ViT model for 1- and 3-ion images. Our ARTIQ system has an integral Thresholding model, which is also evaluated as a benchmark. We also introduce the ResNet-based CNN model as an ML solution benchmark, with its effectiveness validated in Ref. [85]. For the edge-device task, we selected ResNet20-based CNN (the smallest architecture variant of standard ResNet CNN) as our benchmark [141].

$$\bar{F} = \frac{1}{2^n} \sum_i p\left(measured\ i | prepared\ i\right) \tag{4.1}$$

The mean measurement fidelity (MMF) is employed as fidelity comparison, as defined in Ref. [142], which is defined as Eq. (4.1) where n is the number of ions, and the summation index i spans across all possible combinations of n-qubit states. The measured i and prepared i represent predicted output from the qubit classifier and provided label, respectively. The classification error is therefore calculated as (1 - $\bar{F}$)

The MMF error for the 1-qubit and 3-qubit datasets obtained with thresholding, LUTEnsemble, CNN, and ViT are reported in Table 4.2. We also report the number of trainable parameters calculated from the `count_parameters()` function in PyTorch [23]. Parameters are not reported for the thresholding method as it uses the histogram discrimination procedure [132]. For a 1-qubit, the classification error of machine learning techniques, LUTEnsemble, CNN,

TABLE 4.3: FPGA resource consumptions constrained by the target clock frequency: 250 MHz. Look-up-table (LUT), Digital Signal Processing (DSP) Elements, Block RAM (BRAM), and Flip Flops (FF) are used to indicate the main FPGA programmable logic resources. ViT implemented under 1-qubit and 3-qubit sets consumes 4054 and 8797 clock cycles, respectively.

Module	LUT	DSP	BRAM	FF
Cameralink	1513	0	18	2097
Util	0.5%	0%	1.5%	0.35%
LUTEnsemble (1-Qubit)	25500	0	0	7275
ViT (1-Qubit)	69395	467	207	48113
Util	23%	15%	17%	8%
LUTEnsemble (3-Qubit)	26065	0	0	8860
ViT (3-Qubit)	152021	600	272	76568
Util	50%	20%	22%	12%

and ViT, is lower than the thresholding error by 2.9%, 3.0%, and 3.1%, respectively. The number of parameters of ViT is smaller than that of CNN. In the 3-qubit scenario, the benefits of the machine learning models become more apparent, as LUTEnsemble, CNN, and ViT yield MMF errors that lower the threshold method by 8.7%, 9.7%, and 9.9%. The MMF error of the ViT is 1.13×, 1.80×, and 7.6× lower than CNN, LUTEnsemble, and thresholding, respectively, with its parameters considerably smaller than CNN. Notably, the parameter count of LUTEnsemble is not a strong directive metric for its model complexity since the LUT-based DNN is also constrained by the lookup table size of each neuron.

4.3.2 Latency Comparison

The FPGA design is running on the `m_axis_aclk` (see Figure 4.5). Operating under a frequency constraint of 250 MHz (4 ns), the resource consumption of synthesized hardware is shown in Table. 4.3. Around 20% and 50% look-up-table (LUT) is consumed by 1-qubit and 3-qubit ViT and 4054 clock cycles (16.22 μs) and 8797 clock cycles (35.19 μs) are utilized respectively. The LUTEnsemble serves at a latency and area-tailored solution, with 6 clock cycles (24 ns) and a smaller resource consumption achievable.

By integrating the generated FPGA hardware into the qubit detection system (see Figure 4.2), the system's latency is experimentally determined by measuring the duration between various

probed signals on an external oscilloscope. The EMCCD camera's exposure time is set to 1 ms, and the image size is 10×10 and 12×24, representing 1- and 3-qubit detection, respectively. The Cameralink transmission frequency is set to its maximum frequency of 17 MHz [97]. The camera is externally triggered by a signal originating from the ARTIQ system. This signal also acts as the starting point for latency measurements. We individually measure (`FVAL`, `tx_done` and `vit_valid`) by mapping them to an external IO pin. A 2.5 GHz oscilloscope is used for dual-channel edge detection. The results in this section are achieved from 20 times repeated tests.

- `FVAL`: $\mathrm{T_{Ch2}}(\texttt{FVAL}) - \mathrm{T_{Ch1}}(\texttt{trigger})$
- `tx_done`: $\mathrm{T_{Ch2}}(\texttt{tx\_done}) - \mathrm{T_{Ch1}}(\texttt{trigger})$
- `vit_valid`: $\mathrm{T_{Ch2}}(\texttt{vit\_valid}) - \mathrm{T_{Ch1}}(\texttt{trigger})$

The Sundance FPGA supports customized interest-of-area crops from a full 512×512 image. Then, the 10×10 and 12×24 area is cropped by the camera and then transmitted to the FPGA-based ViT. In contrast, in the GPU test, the most fitted crop scales for the target image size supported by the cameralink grab card (Teledyne Dalsa Aquarius Base CL x1) are 16×16 and 24×24, which is received firstly by the cameralink grab card and then further cropped to 10×10 and 12×24 in Pytorch environment before sending to the ViT on GPU. In light of this, our FPGA solution offers superior transmission efficiency for customized ROI cropping compared to the GPU approach, supporting more optimized crops directly from a full 512×512 image.

The results of the FPGA-based and GPU-based ViT latency measurements are shown in Table. 4.4. The `FVAL`: 1.41$\pm$0.01ms is measured on our FPGA probed IO signal on different image sizes from 10×10 to 512×512 which validates that this duration is independent of the transferred image size and only determined by the cameralink protocol. Then, we safely assume the GPU solution shares the same value as FPGA, as there is no accessible probed IO on the GPU solution for direct measurement.

The `tx_done` measured on FPGA is conducted on target image sizes 10×10 and 12×24, which is 1.75$\pm$0.01 ms and 2.25$\pm$0.02 ms respectively, which indicates that this duration is

TABLE 4.4: Latency measurement results of FPGA- and GPU-based qubit detection system (Unit: ms). The mean and stand deviation are calculated from 20 times repeated tests. The FPGA-based ViT latency (0.016 ms and 0.035 ms) listed here is from the Vivado hardware report, which consumes fixed clock cycles (4054 for 1-qubit and 8797 for 3-qubit) in repeated executions. The internal `tx_done` signal is not accessible on the GPU solution. In comparison, the FPGA system offers much lower latency and higher certainty (lower standard deviation). FPGA system achieves 119× and 94× speedup than GPU solution for 1-qubit and 3-qubit test respectively with the ViT accelerator on FPGA is 178× and 84× faster than that on GPU. Another interesting part is that the ML accelerator on GPU only takes ∼1.3% of the entire qubit detection duration with ∼98% latency consumed on the PCIe-rooted interface.

Platform	FPGA		GPU	
Ion number	1-Qubit	3-Qubit	1-Qubit	3-Qubit
`FVAL`	1.41±0.01	1.41±0.01	1.41±0.01	1.41±0.01
`tx_done`	1.75±0.01	2.25±0.02	-	-
`vit_valid`	1.78±0.02	2.29±0.02	211.95±17.72	214.70±10.87
ViT model	0.016	0.035	2.85±0.13	2.95±0.11
Util	0.89%	1.53%	1.34%	1.37%

corresponding to the actual transmitted image size. Unfortunately, the precise value for the GPU solution is immeasurable because of the absence of probed `tx_done` signal.

The overall system latency indicated by `vit_valid` and separate ViT latency shows that the FPGA system achieves 119× and 94× speedup than the GPU solution for 1-qubit and 3-qubit tests, respectively, with the ViT accelerator on FPGA 178× and 84× faster than that on GPU.

Notably, the developer only has access to measure the ViT inference latency separately on GPU through the `time()` function in the Python environment, the rest of latency (includes PCIe-rooted interface latency, data movement latency and operating system-related latency) are summarily measured together. This transmission time is influenced by factors such as the operating system's workload and scheduling mechanism, hence this latency is in-deterministic and shows a big deviation.

Using the 3-qubit results as an example, we also include a comparison with the baseline methods (Thresholding on Kasli) in Figure 4.12.

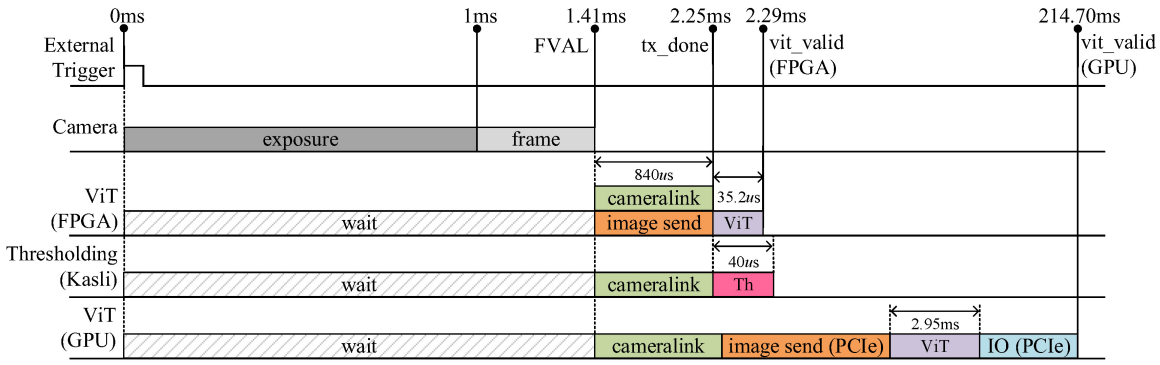


FIGURE 4.12: Latency Measurements from Testbed for 3-Qubit Detection.

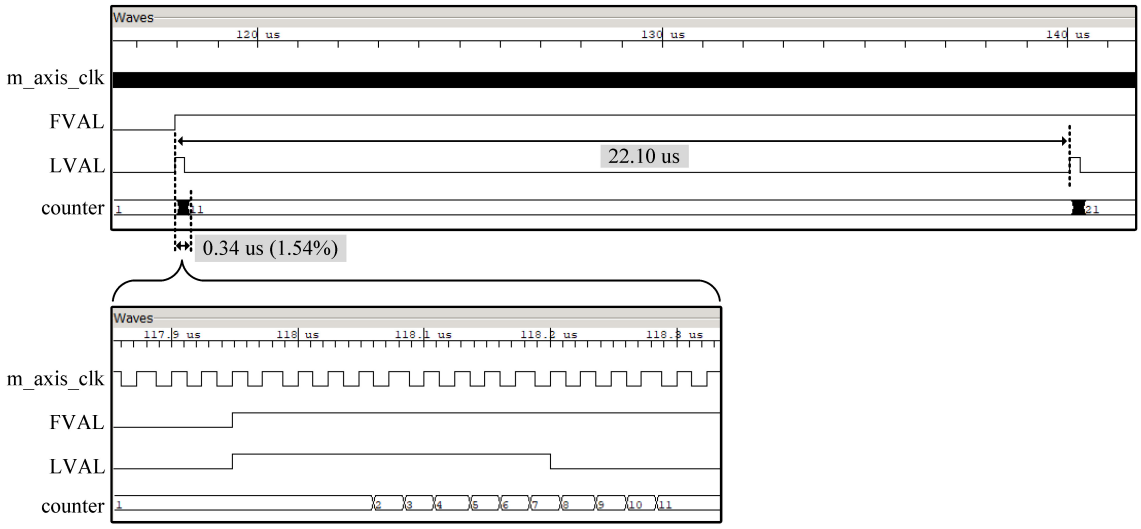


FIGURE 4.13: ILA Waveform over an entire 10×10 pixel image capture.

From these results, the ViT achieves latency comparable to the thresholding baseline. The overall detection latency is dominated by the EMCCD camera's frame period (0.41 ms) and the Cameralink data streaming (840 μs). Further insights are gained by looking at the image send step, which is the duration cost by moving data from the cameralink bus to the ViT accelerator. For FPGA, it is almost parallel (only 1 pixel later) with the cameralink data streaming thanks to the pipeline design (see Section. 4.1.3). In GPU implementation, it is transferred in a serial model because the image send operation has to start until one single image has been fully received from the cameralink card and transferred to memory.

To analyze the logic behavior of the internal signals within the FPGA, we employed the Integrated Logic Analyzer (ILA [143]) to record critical internal signals. With 1-qubit as an example, Figure 4.13 shows the waveform obtained by ILA, captured in real-time at 250 MHz (AXIS clock: `m_axis_clk`), and reflects the actual data transmission. It can be seen that, for a 10×10 image, `FVAL` remains `HIGH` until the full image is received, and `LVAL` is valid during each line of data transmission. We only show single `LVAL` duration for clear illustration purposes. The counter indicates the number of received pixels inside the ViT. It accepts a new row of 10 pixels each time `LVAL` is valid until it successfully receives 100 pixels forming a complete image, after which it resets to `HIGH`.

The entire line reading operation only accounts for 1.54% of the interval between two `LVAL`s. This is because no matter how many pixels will be transmitted, there are always 512-pixel time slots consumed per line [97]. This observation provides insight for future improvements; that is, lower latencies could be achieved from other cameras with more efficient readout modes.

4.4 Summary

This chapter presents a low-latency neural network-based qubit detection system and its real-time FPGA implementation for trapped ion quantum information processing. First, we use two machine learning models (ViT and LUTEnsemble) to infer the state of 2D images and use a simulated dataset that closely resembles the statistics of experimental data. The LUTEnsemble, tailored for latency and area optimization, shows improved one- and three-qubit fidelities over thresholding [132] methods in an ultra-low latency (24 ns), significantly surpassing modern trap-ion system requirements (120-400 μs). The ViT demonstrates better-balanced latency and accuracy tread-off, which improved one- and three-qubit fidelities over thresholding, LUTEnsemble and CNN [85]. On the three-qubit test, the MMF error of the ViT is 1.13×, 1.80×, and 7.6× lower than CNN, LUTEnsemble, and thresholding, respectively, with its parameters considerably smaller than CNN. The ViT's latency (16-35

μs) is comparable in modern trap-ions, making it an optimal solution for systems with more available resources and a higher demand for detection accuracy.

Second, we reduce the entire system latency by integration with a cameralink interface and acceleration of the ML model on an FPGA. We experimentally compare the latency of our ViT solution with that of a GPU. As our approach is highly parallel, avoids transfers between multiple cards, and minimizes buffering, total detection latency is two orders of magnitude lower than the GPU.

Interestingly, as for the proposed FPGA system, the ViT's latency only accounts for 0.89-1.53% of the detection system's latency. The bottleneck is currently the EMCCD camera's digitization of the image and transmission via Cameralink. Further reduction in latency will, therefore, require an improved camera readout interface.

Future work will extend this architecture in two key directions. First, we will re-train the ViT and LUTEnsemble using experimentally collected images, complete with all imperfections, and then evaluate this FPGA-based qubit detection system in a physical trap-ion experiment.

Second, we will enhance the system's scalability by introducing partial reconfiguration. The current workflow relies on the PCIe driver in a Linux PC to pass configuration commands to the Cameralink Deserializer through a PCIe interface (see Figure 4.2). Whenever the ML accelerator is updated, a new bitstream must be loaded to refresh the FPGA logic. This process requires rebooting the Linux PC to detect and initialize the PCIe peripheral, which can be inconvenient for tasks running on the same PC. We believe partial reconfiguration could mitigate this inconvenience. Furthermore, this challenge has also motivated the general composable framework design discussed in the following chapter.

CHAPTER 5

fSEAD: Composable Hardware on FPGA SoC

Chapter 4 highlighted the inconvenience of updating partial functions within an FPGA, which could require refreshing the entire bitstream or even rebooting the PC, leading to interruptions in ongoing tasks. Inspired by this challenge, this chapter investigates approaches to enhance the composability and scalability of FPGA-based SoCs to overcome these issues. A general ensemble model implementation serves as the case study. To accomplish this, we investigate a flexible computing architecture through the lens of FPGA streaming ensemble anomaly detection (fSEAD) tasks. We propose a framework comprising multiple partially reconfigurable blocks, termed pblocks, which are interconnected via an AXI switch and facilitate arbitrary composition before merging and combining results at run-time to create an ensemble. We validate this approach by comparing fSEAD to an equivalent CPU implementation across four standard datasets, yielding speed-ups ranging from $3\times$ to $8\times$.

The presentation of this chapter is based on background given on ensemble and anomaly detection in Chapter 2 and expands on work previously published in [2].

Ensembles are a class of methods that pool weak detectors to form a more accurate combination [13]. Over a number of decades, they have proven to be an excellent methodology that utilizes the diversity of weak detectors to reach a better overall decision than the individual ones [12].

Anomaly detection is a key machine learning (ML) task and refers to the automatic identification of unforeseen or abnormal samples embedded in normal data [28, 29]. Applications of anomaly detection include fault detection surveillance systems [144], fraud detection in financial transactions [145], intrusion detection for network security [146], monitoring of

sensor readings in aircraft [147] and discovery of potential risks or medical problems in health data with predictive maintenance [148].

When data is susceptible to concept drift [149], anomaly detectors can be utilized in a streaming fashion, with the detector being updated in an online manner. Streaming techniques store and process a window of recent instances. Streaming ensemble anomaly detectors (SEADs) achieve high accuracy under limited memory, processing, and time constraints because ensembles contribute high accuracy and robustness, and real-time processing is enabled via streaming algorithms.

Existing anomaly detection libraries have been developed for CPUs. These include unsupervised, supervised, heterogeneous approaches such as SUOD [33] and PyOD [52]. These libraries provide a single, well-documented application programming interface (API), making it easy to compare and compose different algorithms. In particular, PySAD introduces a framework for streaming ADs in Python [34] with a common interface. Using the aforementioned AD libraries, a software-based model combination toolkit: $combo$ was presented in reference [53], allowing anomaly detectors to be combined in Python. CPU-based SEAD libraries allow programmers to switch detector types or combine multiple detectors to boost performance for specific scenarios.

Implementing customized anomaly detection algorithms in hardware is desirable to achieve higher performance, lower power, and lower latency [150, 151, 10]. However, most designs do not have the same flexibility or customizability as software-based approaches. This is one of the main challenges for reconfigurable computing, and while we do not solve the problem, we propose an approach with considerably more flexibility than conventional FPGA designs. To the best of our knowledge, no composable FPGA implementations of ADs have been published to date.

In this chapter, to enable AD on FPGA with higher flexibility and scalability, we propose fSEAD, a composable and low latency FPGA-based SEAD library. fSEAD has two components. The first is a high-level synthesis (HLS) based module generator that converts three state-of-the-art SEAD algorithms (Loda [35], RS-Hash [36], and xStream [37]) into

optimized sub-detector-level paralleled FPGA entities which store all parameters in on-chip memory. The second component is a composable hardware framework that enables online switching and dynamic data routing between IP cores at run-time through reconfigurable Dynamic Function eXchange (DFX) and the arbitrary routable AXI4-Stream Switches. This allows new functionality to be introduced to the design at run-time. Composability is gained through the combination of coarse-grained reconfigurability of sub-detectors and switchability, which facilitates their flexible interconnection. While run-time reconfiguration of the FPGA introduces tens or hundreds of milliseconds in overhead for large FPGAs [152, 153], this is not a concern for fSEAD as this is only done when fSEAD is idle.

The main contributions of this chapter are:

- The first FPGA-based ML system that allows complex and more powerful ADs to be created from simple blocks without recompilation.
- The creation of hardware implementations for three streaming ensemble anomaly detectors (Loda, RS-Hash, and xStream) and demonstration of how they can be integrated within our framework. These implementations are created from an HLS-based generator for FPGA instances, with a GCC-based alternative that creates multi-threaded CPU versions for comparison. New detectors can be written in C and Python and are easily integrated into this library.
- A composable framework that utilizes multiple reconfigurable regions connected via two AXI switches. In the implementation, high frequency is achieved through floorplanning pblocks surrounding the fSEAD infrastructure, minimizing routing delay.
- We use PYNQ partial overlays invoked in Python for executing AD functions [15], allowing an easy-to-use interface for composing and comparing different ADs.
- This research uses publicly available datasets and our design has been made open source to facilitate reproducible research [1].

[1] fSEAD: https://github.com/bingleilou/fSEAD.

The remainder of the Chapter is organized as follows. We start in Section 5.1 with an introduction of the SEAD background and a formal definition of the framework that we deployed in the FPGA. In Section 5.2, we describe the design of the proposed fSEAD from four aspects: Module Generator, DFX Tool Flow, Composable Infrastructure and its FPGA Implementation. Then, the results of the experiment and performance are shown in Section 5.3. Finally, Section 5.4 concludes the work.

5.1 Related Work

This section introduces anomaly detectors for streaming data, followed by the related work of FPGA-based anomaly detection applications in this domain. Backgrounds of the ensemble can be found in Chapter 2.

5.1.1 Streaming Anomaly Detection

Anomaly detection is a key machine learning (ML) task, which refers to the automatic identification of unforeseen or abnormal samples embedded in a large amount of normal data [28, 29]. From the perspective of processing data, we distinguish between two anomaly detection types: static and streaming.

Static detectors usually operate on a relatively large batch of data before performing information extraction and feature analysis to identify rare items, events, or observations from the general distribution of a population. Representative methods include k-Nearest Neighbors (kNN) [30], Local Outlier Factor (LOF) [31], and Principal Component Analysis (PCA) [32]; for a more comprehensive collection of static methods, we refer the reader to the SUOD [33]. While batch processing can lead to high throughput and accuracy, it is not suitable for systems that require real-time performance since the computing resource requirements and latency increase with batch size.

In contrast, streaming methods only store and process a window of recent instances [29, 34]. This is more amenable to achieving accurate anomaly scores under limited memory,

TABLE 5.1: Block Diagram of SEAD Methods

Anomaly Detectors	**Blocks**			
	Projection	***Core***	***Sliding-Window***	***Score***
Loda	prj-loda	histogram	$1 \times W$	$-\log_2(c/W)$
RS-Hash	prj-rshash	CMS	$w \times W$	$-\log_2(1 + \min\{c_1, \ldots, c_W\})$
xStream	prj-xstream	CMS	$w \times W$	$-\log_2(1 + \min\{2^1 c_1, \ldots, 2^W c_W\})$

[1] W: the length of the sliding-window.
[2] w: the number of hash functions in the count-min sketch (CMS).
[3] c: the count of histogram or hash code of CMS

processing, and time constraints. Moreover, the algorithms are designed to facilitate more lightweight and potentially real-time implementations. A group of algorithms that support streaming anomaly detection processing includes, but is not limited to, ensemble-centric methods [35, 36, 37], tree-based methods [38, 39, 40], kernel-methods [29], as well as Neural Network-based solutions such auto-encoders [41] and adversarial models [42].

In this work, we select a set of three state-of-the-art and representative anomaly detectors for our HLS module generator: Loda (Light-weight Online Detector of Anomalies) [35], RS-Hash [36], and xStream (Outlier Detection in Feature-Evolving Data Streams) [37]. These are briefly explained below. We denote $X = \{\vec{x}_i\}$ as the input dataset and dimension d of each sample, then $\vec{x}_i \in \mathbb{R}^d$ is the ith input sample of the data stream, and R is the ensemble size.

Loda is a projection-based histogram detector, RS-Hash uses a randomized subspace hashing algorithm, and xStream is a density-based detector. Although the three techniques are based on different principles, the algorithms can be expressed as a composition of the following standardized blocks: *Projection*, *Core*, *Sliding-window*, and *Score*. Table 5.1 summarizes the main functional blocks of these three SEAD methods.

The purpose of *projection* is to reduce the dimensionality of a set of points while retaining the essence of the original data. Randomness between sub-detectors to improve diversity in an ensemble is also introduced at this step. It is also the most computationally expensive step. The *core* block is the cornerstone of each method: Loda is based on the histogram; RS-Hash and xStream approaches make use of the count-min sketch (CMS) [154], in which

Algorithm 3 Loda

Input: Streaming input signal $X = \vec{x}_i \in \mathbb{R}^d$; ensemble size R; data dimension d; sliding-window length W.
Output: Streaming anomaly value $Score$.

#pragma `HLS DATAFLOW`
// ❶WINDOWER:
for $dim = 1, 2, \ldots, d$ **do**
    A shift register to produce an entire sample: X with d features.
end
// ❷ENSEMBLE:
for $r = 1, 2, \ldots, R$ **do**
    // ❸PROJECTION:
    for $dim = 1, 2, \ldots, d$ **do**
        #pragma `HLS PIPELINE`
        $prj\_X \leftarrow prj\_X + X * loda\_prj$ (random projection)
    end
    // ❹HISTOGRAM:
    $idx \leftarrow (prj\_X - loda\_\min)/(loda\_\max - loda\_\min)$
    $v \leftarrow sliding\text{-}window[idx]$
    // ❺SLIDING-WINDOW:
    Update $sliding\text{-}window$.
    // ❻SCORE:
    $score(r) \leftarrow \log_2(v)$
end
// ❼SCORE AVERAGING:
$Score \leftarrow \frac{1}{R} \sum_{r=1}^{R} score(r)$
return $Score$

w pair-wise independent hash tables are used. These three methods all operate over a *sliding-window* of the input data, which is received in a streaming fashion. The difference is that the histogram-based Loda only uses a 1-row table as a sliding window, whereas the CMS-based methods allow the sliding window with a w-row table ($w \geq 1$). The *score* block calculates the negative log-likelihood, so the less likely a sample is, the higher the anomaly value.

Algorithm 3 to Algorithm 5 present the pseudo-code for Loda, RS-Hash, and xStream, respectively. Algorithm 6 introduces the hash function, which is applied in RS-Hash and xStream. Clearly, each function has the streaming input x and the streaming output $Score$.

Algorithm 4 RS-Hash

Input: Streaming input signal $X = \vec{x}_i \in \mathbb{R}^d$; ensemble size R; data dimension d; sliding-window length W; hash functions number in CMS: w.

Output: Streaming anomaly value $Score$.

#pragma `HLS DATAFLOW`
// ❶WINDOWER:
for $dim = 1, 2, \ldots, d$ **do**
    A shift register to produce an entire sample: X with d features.
end
// ❷ENSEMBLE:
for $r = 1, 2, \ldots, R$ **do**
    // ❸PROJECTION:
    for $dim = 1, 2, \ldots, d$ **do**
        #pragma `HLS PIPELINE`
        $norm\_X \leftarrow$ normalize $X[\text{dim}]$ to the range of [0,1]
        $prj\_X \leftarrow (norm\_X + rshash\_alpha[r][\text{dim}])/rshash\_f[r]$
    end
    // ❹HASH-FUNCTION:
    for $row = 1, 2, \ldots, w$ **do**
        #pragma `HLS UNROLL`
        $prj\_hash[row][\text{dim}] \leftarrow prj\_X$
        $hash\_value[row] \leftarrow Jenkins(key = prj\_hash[row], len = d, seed = row)$
        (see **Algorithm 6** for details of $Jenkins$)
        $v_{row} \leftarrow sliding\text{-}window[hash\_value[row]]$
        // ❺SLIDING-WINDOW:
        Update $sliding\text{-}window$.
    end
    $min\_v \leftarrow \min\{v_1, v_2, ..., v_w\}$
    // ❻SCORE:
    $score(r) \leftarrow \log_2(min\_v)$
end
// ❼SCORE AVERAGING:
$Score \leftarrow \frac{1}{R} \sum_{r=1}^{R} score(r)$
return $Score$

The ensemble can be divided into seven parts. The ❶WINDOWER block uses a shift register to assemble samples $x \in \mathbb{R}^1$ into an entire vector $\vec{x}_i = \{x_1, x_2, ..., x_d\} \in \mathbb{R}^d$, ❷ENSEMBLE is a big *for* loop with R independent iterations. Each iteration can be regarded as a sub-detector with a unit of 1. Each sub-detector executes the functional modules of ❸PROJECTION,

Algorithm 5 xStream

Input: Streaming input signal $X = \vec{x}_i \in \mathbb{R}^d$; ensemble size R; data dimension d; sliding-window length W; hash functions number in CMS: w; projection size: K.
Output: Streaming anomaly value $Score$.

#pragma `HLS DATAFLOW`
// ❶WINDOWER:
for $dim = 1, 2, \ldots, d$ **do**
    A shift register to produce an entire sample: X with d features.
end
// ❷ENSEMBLE:
for $r = 1, 2, \ldots, R$ **do**
    // ❸PROJECTION:
    for $dim = 1, 2, \ldots, d$ **do**
        #pragma `HLS PIPELINE`
        for $k = 1, 2, \ldots, K$ **do**
            #pragma `HLS UNROLL`
            $prj\_X[k] \leftarrow prj\_X[k] + X[\text{dim}] * xstream\_prj[\text{dim}][k]$
        end
    end
    // ❹HASH-FUNCTION:
    for $row = 1, 2, \ldots, w$ **do**
        #pragma `HLS UNROLL`
        $prj\_hash[row] \leftarrow perbins(prj\_X)$
        $hash\_value[row] \leftarrow Jenkins(key = prj\_hash[row], len = K, seed = row)$
        (see Ref. [37] and **Algorithm 6** for details of $perbins$ and $Jenkins$)
        $v_{row} \leftarrow sliding\text{-}window[hash\_value[row]]$
        // ❺SLIDING-WINDOW:
        Update $sliding\text{-}window$.
        $score_{row} \leftarrow \log_2(v_{row}) + row$
    end
    // ❻SCORE:
    $score(r) \leftarrow \min\{score_1, score_2, ..., score_w\}$
end
// ❼SCORE AVERAGING:
$Score \leftarrow \frac{1}{R} \sum_{r=1}^{R} score(r)$
return $Score$

❹CORE (Histogram for Loda, and Hash Function for RS-Hash and xStream) ❺SLIDING-WINDOW and ❻SCORE sequentially. The execution of these modules is time-dependent, that is, the start of the latter one must wait for the execution of the former to complete. Finally,

Algorithm 6 Jenkins Hash Function

Input: The string: key, string length: len and random seed: $seed$
Output: The hash code of input string.

```
hash ← seed;
for i = 1, 2, ..., len do
    #pragma HLS PIPELINE
    hash ← hash + key[i]
    hash ← hash + (hash << 10)
    hash ← hash ⊕ (hash >> 6)
end
hash ← hash + (hash << 3)
hash ← hash ⊕ (hash >> 11)
hash ← hash + (hash << 15)
hash_code ← hash%MOD
return hash_code
```

❼SCORE AVERAGING is used for producing the final ensemble anomaly score by averaging the outputs of all sub-detectors.

The serial combination of these four blocks (❸❹❺❻) constitutes a base sub-detector. We use R parallel executions of the base sub-detector with a different starting seed or hash and average their scores to form an *ensemble*. For a CPU implementation, R sub-detectors are processed sequentially. Details of HLS directives used in the pseudo-code are described in Section 5.2.1.

5.1.2 FPGA-based Anomaly Detection Solutions

This section reviews other anomaly detection techniques and literature on FPGA-based anomaly detection. A powerful class of anomaly detectors utilizes tree-based structures, with examples including iForest [38], HS-Tree [40], and RS-Forest [39]. The Isolation Forest (iForest) approach builds an ensemble of "Isolation Trees" (iTrees) for the dataset, and anomalies are the points with shorter average path lengths on the iTrees. HS-trees are similar to Isolation Forest, but decision rules within tree nodes are generated randomly. RS-Forest takes a further step by using multiple fully randomized space trees to tackle the streaming

detection problem from the density estimation aspect, which is more efficient as it leverages the common operations shared by the prediction and model update processes.

Kernel methods and data-centered models have also been proposed as online anomaly detectors. Support Vector Regression (SVR) with Gaussian kernel-based online novelty detection on temporal sequences is presented by Ma *et al.* [29]. High-performance FPGA implementations of online kernel methods were demonstrated in references [155, 156]. Based on static data-centered LOF methods [31], an incremental LOF algorithm appropriate for detecting outliers in data streams is proposed in [157], which provides equivalent detection performance as the iterated static LOF algorithm while requiring significantly less computational time. Das *et al.* proposed a system based on feature extraction and Principal Component Analysis (PCA) [150]. Their FPGA implementation could support data at over 20 Gbps. Pang *et al.* [155] present a high-performance FPGA implementation that achieves improvements in execution time, latency, and energy consumption by factors of 5, 5, and 12, respectively, over CPU and digital signal processor (DSP) implementations for the online kernel method [29]. Hayashi and Matsutani [151] offload the Local Outlier Factor (LOF) calculation to an FPGA-based Network Interface Card for online anomaly filtering. This leads to throughput improvements of up to 10x on the anomaly filtering over a software-based execution.

Neural Network models have also been proposed for streaming anomaly detection. Moss *et al.* [10] introduce an FPGA accelerated Neural Network-based anomaly detector based on an auto-encoder for processing of physical-layer radio-frequency (RF) signals. This design processes continuous 200 MS/s complex inputs, producing anomaly classifications at the same rate, with a latency of 105 ns, improving at least 4 orders of magnitude over a conventional approach using a software-defined radio.

While there are published FPGA implementations of random forests [158, 159, 160], kernel and neural network-based FPGA-based accelerators, these are all fully-customized hardware designs for a specific algorithm. Moreover, they typically utilize most of the available FPGA resources, meaning it is difficult to implement such detectors in a partial region of an FPGA. The focus of this work is on the development of a flexible library. Unlike the many comprehensive and easy-to-use anomaly detection libraries released on different software

platforms [49, 50, 51, 33, 52, 34], we believe this is the first flexible FPGA library for anomaly detection in the literature.

Although we are unaware of any publications describing runtime reconfigurable anomaly detection libraries, dynamic reconfiguration on FPGAs has been used for other FPGA applications. The most similar to this work is Wilson, who proposed a real-time video processing pipeline that utilizes the dynamic reconfigurable aspects of FPGA [74]. Their work used 11 reconfigurable regions, allowing for multiple custom run-time configurations, and they adopted partial bitstreams with PYNQ overlays to ease software development. Our work is different in that it uses AXI switches to dynamically switch anomaly detectors and model combinations between reconfigurable regions and employs a flexible module generator to create the regions themselves.

5.2 Design

This section provides a high-level description of the fSEAD system, followed by the module generator for creating integrated anomaly detection IPs. Next, we introduce the DFX workflow we abstracted to create a partial reconfiguration project in the Xilinx Vivado environment. We then describe the interconnection scheme, which provides a high degree of flexibility and enables IP modules to be composed at run-time, allowing applications to be efficiently accelerated without regeneration of bitstreams. Finally, the FPGA implementation is discussed.

The system framework is illustrated in Figure 5.1. It consists of four software and hardware components. *fSEAD_gen* in the upper-left corner of Figure 5.1 is a Python-based module generator. It takes parameterized function entities and produces Vivado HLS modules. The lower-left corner of Figure 5.1 shows the interface for pblocks, which take streaming inputs and produce streaming outputs. These then are encapsulated as multiple unique IPs and wrapped within AXI streaming interfaces and an AXI-Lite controller. All parameters required by the IP modules are stored in on-chip memory (OCM) for performance. The lower-right sub-figure shows multiple pblock regions. Multiple sub-detector IPs are arranged in a spatially

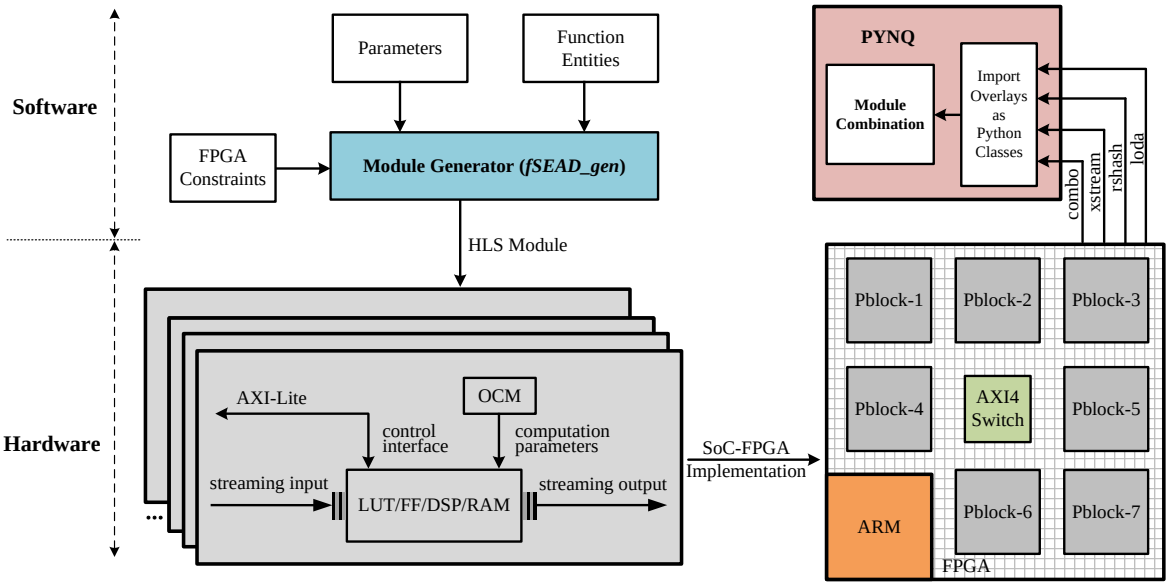


FIGURE 5.1: Overview of the fSEAD Framework.

parallel fashion within each pblock. Each pblock is implemented by many reconfigurable modules (RMs) and can be customized at run-time.

5.2.1 Module Generator

The module generator allows customization of the underlying sub-detectors for latency optimization and resource utilization exploration.

The *fSEAD_gen* module generator, written in Python, takes as inputs the anomaly detector parameters, data type, precision, function description, target dataset, and testing set. It produces a standalone C program suitable for synthesis via HLS as output. The parameters, interface, and optimization directives are all embedded in the C program. A compact sub-detector C instance is formed by combining the ❸PROJECTION, ❹CORE, ❺SLIDING-WINDOW and ❻SCORE parts of Algorithm 3-5. Arbitrary numbers of sub-detectors, specified by the user to the module generator, are integrated parallel to form an ensemble. A self-verifying test bench compares the program with golden results from the original Python description. Thus, programming errors introduced by the generated ensemble program can be quickly identified in the simulation step. For synthesis, compiler directives such as `DATAFLOW`, `ARRAY`

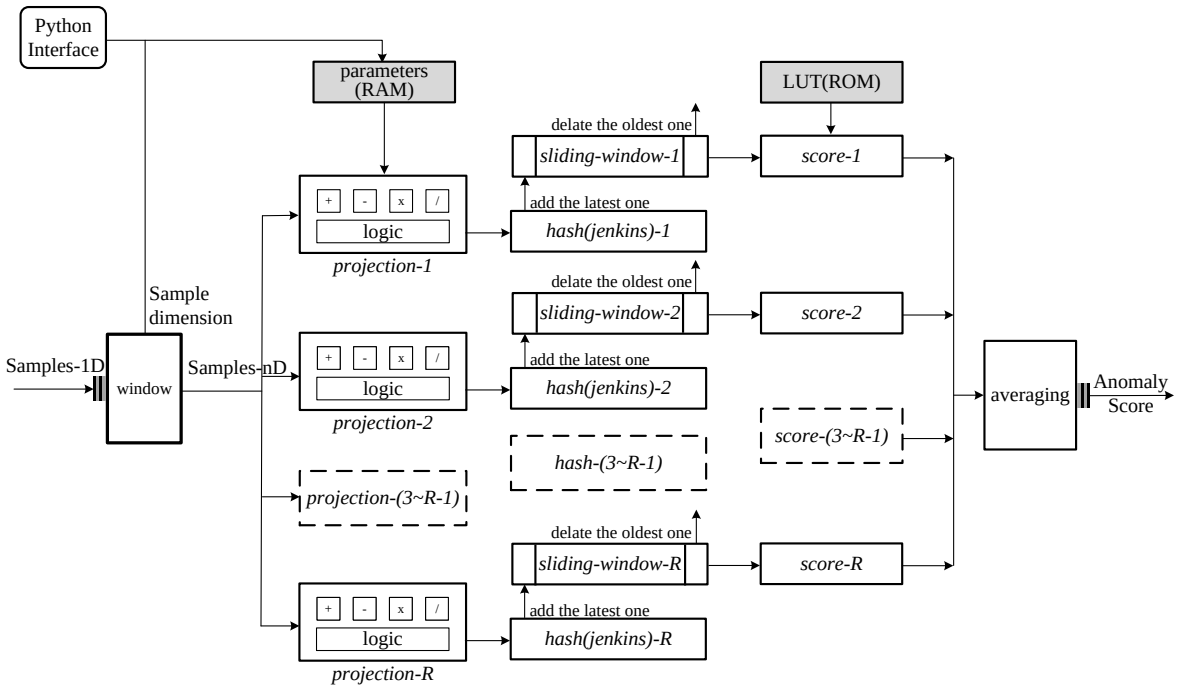


FIGURE 5.2: An Example of A Hash-based AD Hardware Structure in fSEAD.

`PARTITION`, `UNROLL`, and `PIPELINE` are used to aid the translation of the C description to a spatially parallel RTL circuit.

In Algorithm 3 to Algorithm 5, we use the `DATAFLOW` pragma only in the top layer (line-1 of each algorithm) to enable task-level pipelining. This allows functions and loops to execute concurrently and achieve sub-detector parallelism. `UNROLL` is used to create multiple instances of the loop body in the RS-Hash and xStream algorithms to exploit spatial parallelism. This enables all hash functions inside the CMS table (line-16 in Algorithm 4 and line-18 in Algorithm 5) to execute concurrently and similarly accelerates the projection of size K for xStream (line-12 in Algorithm 5). `PIPELINE` reduces the initiation interval (II, the number of clock cycles before the function can accept new input data) by allowing overlapped execution of operations within a loop or function. It is used in the $projection$ loop body of all detectors (line-10 of Algorithms 3 to 5) and the for loop in the Jenkins hash function (line-3 of Algorithm 6). $II = 1$ is achieved in all the loop bodies with `PIPELINE` for our implementation. The result of all these optimizations is a highly parallel design that balances resource utilization and latency.

fSEAD_gen currently supports three types of detectors with arbitrary ensemble sizes and coefficients (Loda, RS-Hash, and xStream). All HLS directives are manually designed for the target FPGA and then embedded in *fSEAD_gen*. *fSEAD_gen* can automatically generate optimizations for different ensemble sizes, such as hyper-parameters and other configurations, without modifying the existing HLS directives. However, developers maintain the flexibility to adapt the current HLS directive to find more optimized solutions for a specific area-latency trade-off requirement. New detectors can be written in C/Python and easily integrated using existing detectors as examples. In this case, specific optimized HLS directives have to be manually re-designed for new members to the fSEAD library.

Figure 5.2 shows an ensemble of hash-based detectors with a more complex hardware implementation than Loda. For simplicity, this figure highlights the parallel sub-detectors generated by the `DATAFLOW` pragma instead of the detailed circuits from the lower level by `PIPELINE` and `UNROLL` pragmas. This represents the main acceleration of fSEAD: task-level parallelism for sub-detectors operating concurrently.

Input data is streamed into the IP on the left interface and outputs the real-time anomaly score on the right side. Input samples are windowed to assemble a batch with target dimension: d (refer to ❶WINDOWER in Algorithm 4 and Algorithm 5). Computation of all the streams occurs concurrently, before a final reduction step (refer to ❷ENSEMBLE, in Figure 5.2 this is averaging). Sub-detector level parallelism is achieved by applying the `HLS DATAFLOW` directive on the top function. In this way, the maximum ensemble size is only limited by resources available in the target FPGA. Inside each sub-detector, a small area and low latency are the goals. We compute each row of a CMS table using independent hash functions in parallel (refer to ❹HASH-FUNCTION). `PARTITION` directives are used to divide a large RAM into smaller storage units, increasing the available parallelism of accesses. This is advantageous because even with dual-port RAM, only one read and one write operation can be completed in one clock period. Notably, since the usage of `PARTITION` significantly affects the resource consumption of LUTs or FFs on FPGA, it is only used for a specific dimension of the target array. This guarantees the functions of `DATAFLOW`, `PIPELINE`, and `UNROLL` directives. All HLS directives above enable a balanced optimization inside an

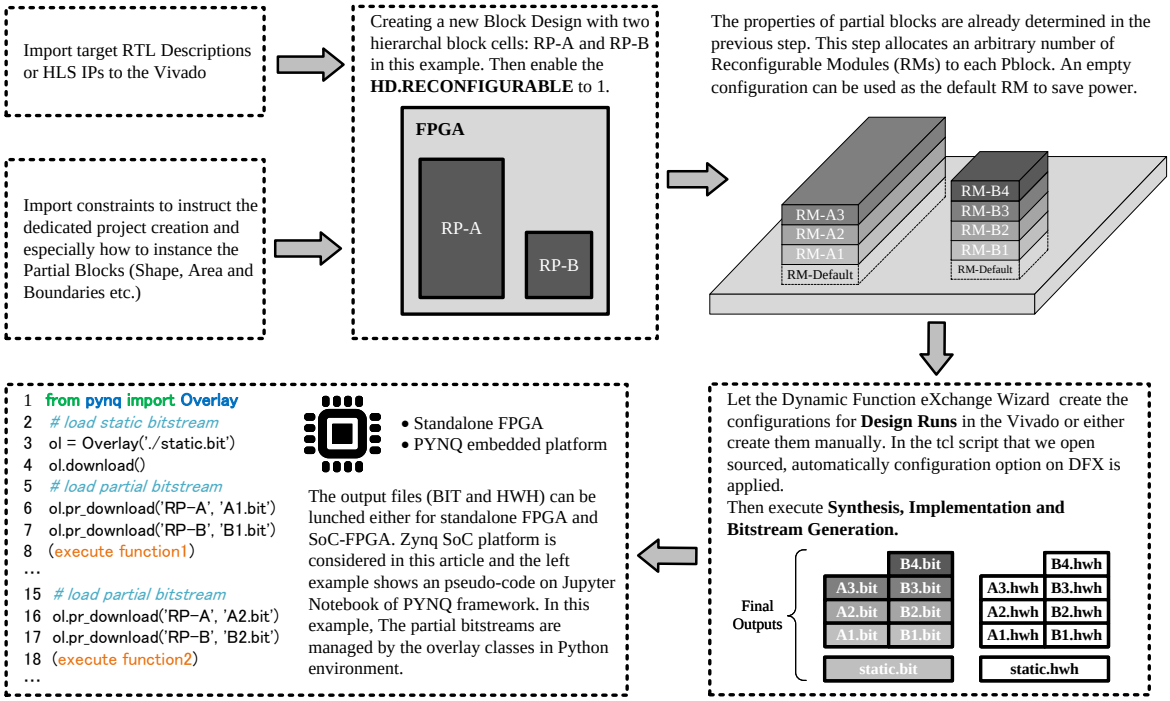


FIGURE 5.3: An Example of Xilinx Partial Reconfiguration Tool flow.

ensemble AD instance, which is independent of the partial-block-level model combination scheme introduced in the following Section 5.2.3.

We constrain the ❸PROJECTION module using `PIPELINE` instead of `UNROLL` inside each sub-detector to avoid high resource cost with little throughput benefit. To compute the logarithm required for the negative log-likelihood score of Table 5.1, a W-deep lookup table with 32-bit representation is used for the window size of W.

5.2.2 DFX Tool Flow

DFX is an important tool to endow fSEAD with partial reconfigurability. Its customized workflow for fSEAD is introduced in this section. Continuing on the basis of the example in Figure 2.9. Figure 5.3 shows an example of how two reconfigurable areas are mapped to Reconfigurable Partitions (RPs) using our tool flow. It should be noted that in addition to the dynamic RMs for each Partial block (*e.g.*, the RM-A1 to RM-A3 for RP-A and RM-B1 to

RM-B4 for RP-B), a default RM can also be assigned for each pblock. The logic of the RM-Default will be first active when the static.bit is downloaded, which brings it a recommended way of setting the default RM to an empty logic to save power before this pblock is configured by the dedicated RM of users.

In the last step (the lower left sub-figure) in Figure 5.3, we show that standalone FPGA or Xilinx PYNQ platforms are able to support the fSEAD execution. In this example, we select the PYNQ as the final execution platform. The bitstreams and a hardware hand-off (HWH) file are used as overlays by PYNQ to automatically identify the Zynq system configuration and IP, including static regions and partial blocks.

5.2.3 Composable Infrastructure

The Xilinx DFX tool [72] supports partial reconfiguration of RMs in an FPGA while the rest of the device remains operational. There is no limitation in the number of RMs supported. In fSEAD, each pblock is a unique RM configuration with its own BIT file. By downloading the appropriate one, the hardware functionality can be customized at run-time. In summary, pblocks are either (1) Ensembles of homogeneous sub-detectors implemented in a pblock and (2) Combination modules that aggregate heterogeneous pblock output streams.

An AXI4 switch acts as a router and orchestrates data movements between pblocks. Inputs can be routed from multiple input streams to any pblock, with outputs routed to remaining pblocks and back to a host processor.

Figure 5.4 shows the composable topology proposed in fSEAD. The grey regions are pblocks. Blue blocks symbolize Direct Memory Access (DMA) controllers for data transfer. Two AXI4-Stream switches [161] enable dynamic routing from a Slave port to a Master port, *e.g.*, for switch-1, S1 in the lower left is the first Slave interface, and M14 in the lower right symbolizes the 14th Master interface. Master and Slave interfaces are symmetric and point-to-point, so Master output signals can connect directly to Slave input signals. Any number of external modules can be daisy-chained together. The modules can be used for many different tasks, such as buffering, data transformation, or routing. Multiple switches

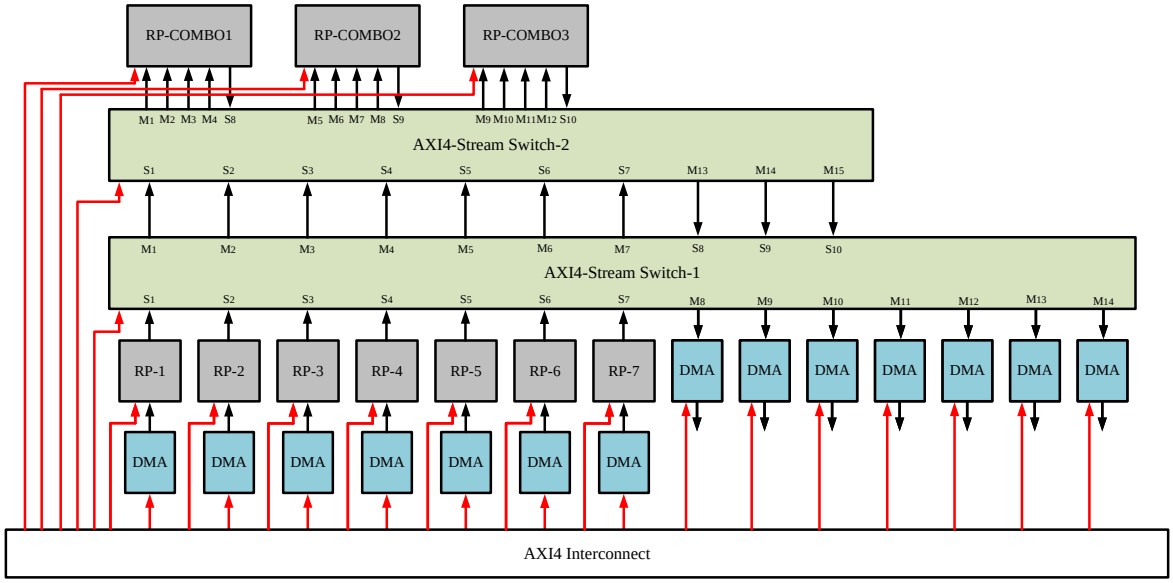
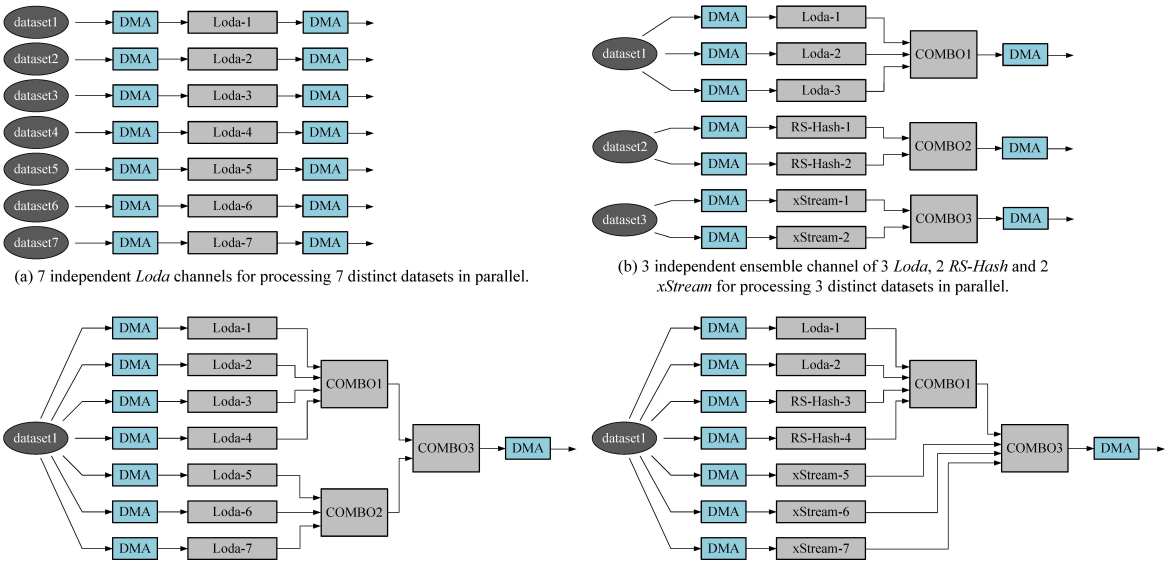


FIGURE 5.4: Composable Topology.

are used since each Xilinx AXI4-Stream Switch IP switch only supports a maximum of 16 Slave ports and 16 Master ports. The cascades of two or more switches allow an arbitrary number of pblocks to be interconnected. The black lines depict AXI4-Stream connections, and the red lines are AXI-Lite bus connections. These are connected and controlled by the AXI4 Interconnect at the bottom.

In our prototype implementation, seven independent pblocks are available for implementing anomaly detectors (shown as RP-1 to RP-7 in Figure 5.4. Each pblock has one AXI4-Stream input connected to a fixed DMA and one output interface connected to a Slave port of AXI4-Stream Switch-1. Multiple sub-detectors can be placed in a single pblock, each with different parameter settings. For example, 35 Loda sub-detectors fit in a single pblock. The combo pblocks at the top of Figure 5.4 are responsible for aggregation. Each is equipped with four input ports and one output port, all connected to Switch-2.

Table 5.2 shows the currently supported combination methods implemented in fSEAD for continuous (score) and discrete (label) targets. The label targets are ‘0’, meaning not an anomaly, and ‘1’ for anomaly. General and global (GG) methods [58, 54, 55, 56] are used by popular machine learning libraries such as scikit-learn [162], XGBoost [163] and LightGBM [164] for aggregation of sub-detector class results. In fSEAD, for sub-detectors



(a) 7 independent *Loda* channels for processing 7 distinct datasets in parallel.

(b) 3 independent ensemble channel of 3 *Loda*, 2 *RS-Hash* and 2 *xStream* for processing 3 distinct datasets in parallel.

(c) Applying all resources on FPGA on single type of detector: *Loda* to get the maximum ensemble size for single dataset.

(d) Combination of 3 ensemble detectors: 3 *Loda*, 2 *RS-Hash* and 2 *xStream* for single dataset.

FIGURE 5.5: Examples of Combination Scheme.

with continuous outputs, we implemented three representative GG methods, namely Averaging (*GG_A*), Maximum (*GG_M*), and Weighted Average (*GG_WA*). To combine label outputs, two commonly used approaches: or (a class is true if any sub-detector outputs true for that label) and voting (the class with most votes is chosen as the output) are applied. In this work, we always use averaging to combine anomaly scores and the or-gate technique to combine labels. Of course, this can be customized by the user.

Routing through the AXI switches is configured via the AXI-Lite interface. A register is used for each master-to-slave connection. After the registers have been configured, the interconnection is determined. Unused master or slave interfaces are disabled. When a slave interface is connected to multiple masters, only the lowest numbered one is used, *e.g.*, if both Master-1 and Master-3 are configured to connect to Slave-2, then Master-1 wins the arbitration, and Master-3 is disabled. Thus, effectively, only one connection between each master and slave is made.

Figure 5.5 shows four example configurations of our composable topology. Figure 5.5(a) shows the simplest case where seven parallel Loda pblocks (each containing multiple sub-detectors) are used to analyze seven different datasets. Each streaming channel implements a

TABLE 5.2: Combination Methods for Scores and Labels

Output	Combination Methods	Equation
score	Averaging	$combo = (score_1 + score_2 + \cdots + score_N)/N$
score	Maximization	$combo = max(score_1, score_2, \cdots, score_N)$
score	Weighted Average	$combo = (w_1 * score_1 + w_2 * score_2 + \cdots + w_N * score_N)/N, (\sum_{i=1}^{N} w_i = 1)$
label	Or	$combo = (label_1 \| label_2 \| \cdots \| label_N)$
label	Voting	$combo = voting(label_1, label_2, \cdots, label_N)$

unique and independent anomaly detection application. Switch-1 is configured to directly route RP-1 to RP-7 to seven output DMA channels. This configuration only requires Switch-1, so connections to Switch-2 and the combo pblocks are disabled. The case in Figure 5.5(b) implements three independent anomaly detection applications. It exploits all pblocks and two of the Switches. RP-1, RP-2, and RP-3 implement a Loda ensemble, and their outputs are routed to the inputs of COMBO1. The output is the final score, which is sent to the host via DMA. The pblocks (RP-4, RP-5) and (RP-6, RP-7) generate RS-Hash and xStream ensembles for two different datasets. Figure 5.5(c) is an example that only operates on a single dataset and a single type of anomaly detector, namely Loda. It uses all the pblocks to implement a maximally parallel, homogeneous ensemble. Finally, Figure 5.5(d) is similar to Figure 5.5(c) but incorporates three different types of detectors: Loda, RS-Hash, and xStream.

The configurations are not limited to the four cases just described. By providing different pblocks and switch configurations, a multitude of customized anomaly detection functionalities can be implemented in a run-time configurable manner.

5.2.4 FPGA Implementation

This section continues the example of Figure 5.5(c) as a concrete example to describe physical implementation on a Zynq UltraScale+ ZCU111 Evaluation Board with XCZU28DR-2FFVG1517E RFSoC FPGA.

Figure 5.6 and Figure 5.7 show the FPGA layout and placement, respectively. The floorplan is based on two considerations: (1) Seven AD-pblocks occupy the main FPGA resources to ensure that the fSEAD has sufficient resources to implement parallel ensembles and complex

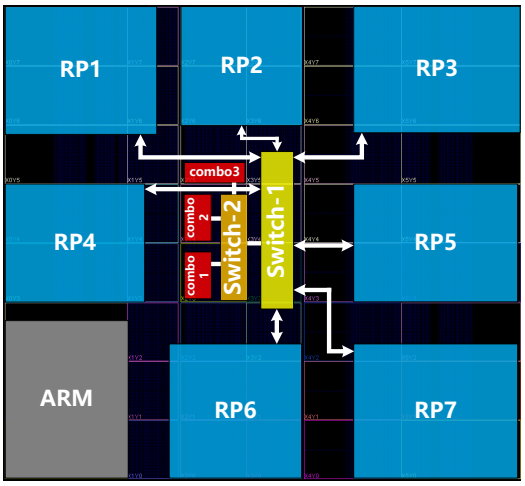


FIGURE 5.6: FPGA Layout.

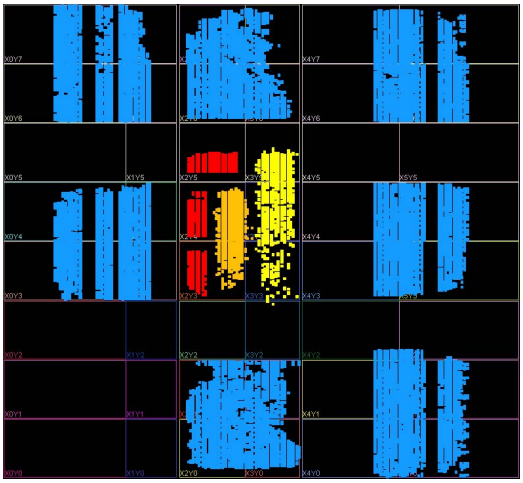


FIGURE 5.7: Placement on FPGA.

detectors. In comparison, the combo modules, switches, and DMA units are not resource-intensive and are reflected in the layout; (2) Although long interconnections are needed because of the distributed pblocks, this is alleviated through AXI bus-level pipelining [165]. This technique isolates the path between the master and slave with registers while maintaining an AXI4 protocol-compliant pipeline stage. The register slice is implemented as a two-deep FIFO buffer that supports bus communication without generating unnecessary idle cycles. It serves to achieve timing closure by reducing the critical path.

We place Switch-1 (yellow) in the center of the FPGA with Switch-2 (orange) adjacent since they two communicate directly with each other. Switch-1 is assigned a larger area than Switch-2 as it connects to seven pblocks (in blue), while Switch-2 is smaller to prevent blocking routes from RP-1 and RP-4 to Switch-1. As shown in Figure 5.6, the nearest routing channel from RP-1 to Switch-1 is only a very narrow programmable slot (white line). The three combo-pblocks (red) only connect to Switch-2. The floorplan for the above-mentioned blocks is compact and sits in the middle layout of FPGA.

The seven AD-pblocks (in blue) occupy the remaining resources. It is important to note that we did not issue placement area constraints for DMAs, AXI4-Interconnect, and the other static components. Instead, we allow these components to be placed by the Vivado tool to minimize routing delay. The spaces between the colored regions are available for any remaining static

logic. Furthermore, during the Partial Reconfiguration process, the DFX Decoupler [72] allows users to isolate the logic being configured until it is done and the new logic reset.

5.3 Results

The aforementioned techniques were implemented, and the results are presented in this section. In Section 5.3.1, we introduce the development environment and test platform. In Section 5.3.2, we demonstrate why larger ensembles with more sub-detectors are desirable and how performance is boosted in terms of accuracy from the model combination. We then discuss the resource use of the static regions and available resources in each of the FPGA partial regions in Section 5.3.3. Furthermore, we show the performance gains of our architecture over a CPU in Section 5.3.4. Finally, in Section 5.3.5, we analyze the performance characteristics of partial reconfiguration and more general features of the fSEAD architecture.

5.3.1 Test Platform

We utilize the Xilinx Zynq UltraScale+ ZCU111 (xczu28dr-2ffvg1517e) board to evaluate the fSEAD library. *fSEAD_gen* is used to generate three anomaly detectors (Loda, RS-Hash, and xStream) with HLS and GCC targets used for FPGA and multi-threaded CPU implementation, respectively. The HLS module is synthesized to RTL using the Xilinx Vivado HLS tool (v2020.1) and then passed through Xilinx Vivado Design Suite (v2020.1). We then deploy the architecture on the FPGA using the PYNQ framework. The GCC compiled versions of Loda, RS-Hash, and xStream as CPU benchmarks are tested on desktop with Intel(R) Core(TM) i7-10700F @2.9GHz and 64GB memory for performance comparisons. The g++ compiler uses flags '-Wall –std=c++14 -g -O3', with '-lpthread' applied for multi-threaded optimization. The flag '-ftree-vectorize' is turned on by default under '-O3' to enable automatic vectorization for further optimization.

The anomaly detection performance evaluation was conducted on four publically available datasets: Cardio, Shuttle, HTTP-3, and SMTP-3, with their main attributes summarised in Table 5.3. Cardio and Shuttle are also used in SUOD [33]. HTTP-3 and SMTP-3 were derived

TABLE 5.3: Datasets

Datasets	Sample Length	Dimension	Outliers	%Outliers
Cardio	1831	21	176	9.61
Shuttle	49097	9	3511	7.15
SMTP-3	95156	3	30	0.03
HTTP-3	567498	3	2211	0.40

from the KDD-cup 99 [166] dataset and are 3 feature variants of the 41 feature full datasets, first used in [40]. The sample number n varies from 1831 (Cardio) to 567498 (HTTP-3), and the dimensionally ranges from 3 to 21. Cardio proportionally has the largest number of anomalies (9.61%), and only 0.03% of samples in SMTP-3 were in the anomaly category. In addition to the four datasets above, others are available, including those in the ODDS Library [167], UCI repository [168], and DAMI Datasets [169] for future research.

Accordingly to the SEAD structure in Figure 2.8, the output scores of each detector are first normalized to [0,1). A sample with a higher score indicates a higher probability that this sample belongs to the anomaly category. Furthermore, with the anomaly percentage, or named contamination rate that the users know in advance, a threshold can be determined to translate anomaly scores to binarized anomaly labels. For example, if the threshold is 0.8, then inputs with anomaly scores of 0.9 and 0.5 would be assigned to labels '1' (anomaly) and '0' (normal), respectively. The standard metric used for evaluating anomaly detectors is the Area Under the Curve (AUC) of the Receiver Operating Characteristics (ROC) curve. It can be used for scores or labels and is described in detail in [170]. We adopt this metric to analyze the accuracy of our implementations.

5.3.2 Sub-detectors and Ensemble Accuracy

We experimented to show the effectiveness of increasing the number of sub-detectors for each of the three supported methods in fSEAD.

For different ensemble sizes, the AUC of detectors in fSEAD and the variance of AUC are evaluated over 10 executions with different random seeds. The hyper-parameters were set to the values in Table 5.4, these being chosen to give an accurate and efficient hardware

TABLE 5.4: Hyper-Parameter Set for the Three Detectors

Detector	*window size*	*Bins*	*CMS-w*	*CMS-MOD*	*K*
Loda	128	20	1	-	-
RS-Hash	128	-	2	128	-
xStream	128	-	2	128	20

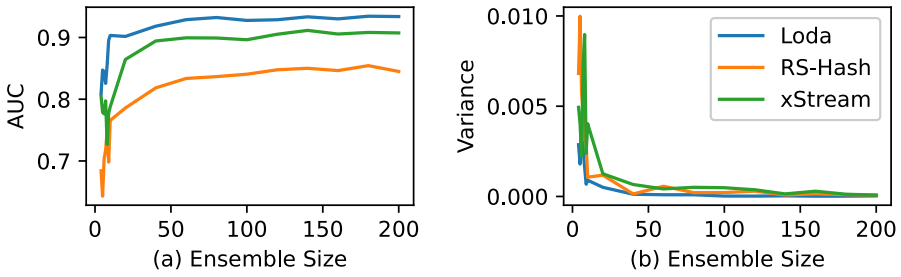


FIGURE 5.8: Ensemble Performance Measured on Dataset: Cardio.

implementation. We refer the reader to the relevant papers regarding selecting these hyper-parameters [35, 36, 37].

Figure 5.8 shows ensemble sizes in the range [3, 200] for Loda, RS-Hash, and xStream respectively. For simplicity, only the Cardio dataset is used to exhibit AUC performance. Figure 5.8(a) represents the mean AUC, and sub-figure (b) shows the variance.

For all detectors, AUC increases with increasing ensemble size before converging to a maximum; the AUC variance shows a decreasing trend before converging to a minimum. This translates to increasing the number of sub-detectors, generally leading to a more reliable result, which follows the underlying principle of ensembles [12, 13].

Furthermore, Table 5.5 shows results for different heterogeneous combinations of detectors over our four benchmarks. The rows in this table are divided into two main parts, mean and variance of the AUC, with the columns divided into AUC of Score and Label results. A, B, and C in the second row indicate three detectors (Loda, RS-Hash, and xStream, respectively), and the numbers indicate the number of pblocks used. The number of sub-detectors we use for each A, B, and C pblock is 35, 25, and 20, respectively; details of this value will be described in Section 5.3.3. For example, A7, B7, and C7 indicate all seven pblocks are utilized for a single type of detector. The fourth one, C223, represents a combination of two pblocks

TABLE 5.5: Model Combination Comparison

AUC	Model	Score									Label								
		A7	**B7**	**C7**	**C223**	**C232**	**C322**	**C331**	**C313**	**C133**	**A7**	**B7**	**C7**	**C223**	**C232**	**C322**	**C331**	**C313**	**C133**
Mean	cardio	**0.933**	0.850	0.907	0.897	0.898	0.891	0.899	0.889	0.900	0.659	0.618	0.646	0.711	**0.721**	0.705	0.708	0.706	0.715
	shuttle	0.991	0.990	0.986	0.990	0.991	0.990	**0.992**	0.990	0.991	0.927	0.950	0.933	0.974	0.970	0.970	0.972	0.974	**0.976**
	smtp3	0.847	**0.856**	0.834	0.848	0.849	0.848	0.851	0.851	0.850	0.743	0.717	0.500	0.757	0.755	**0.770**	0.767	0.765	0.737
	http3	0.993	**0.995**	**0.995**	**0.995**	**0.995**	**0.995**	**0.995**	**0.995**	**0.995**	0.595	0.510	0.512	0.595	**0.620**	0.604	0.584	0.600	0.594
Variance ($\times 10^{-3}$)	cardio	**0.04**	0.2	0.05	0.07	0.06	0.09	0.06	0.23	0.05	0.13	0.14	0.09	**0.05**	0.15	0.14	0.15	0.31	0.21
	shuttle	0.005	**0.000**	0.092	0.001	0.002	0.001	**0.000**	0.002	0.001	0.41	0.06	3.04	0.03	0.07	0.06	**0.01**	0.03	0.02
	smtp3	**0.01**	0.05	1.75	0.08	0.03	0.06	0.04	0.04	0.02	0.62	**0.000**	**0.000**	1.96	1.95	0.93	0.89	1.03	0.71
	http3	0.0011	**0.0001**	0.0003	0.0003	**0.0001**	**0.0001**	**0.0001**	**0.0001**	0.0002	5.01	0.13	**0.12**	8.32	7.76	5.46	3.0	2.46	4.82

assigned to Loda, two pblocks for RS-Hash, and the last three pblocks occupied by xStream. We note that this experiment did not cover all possible combinations. However, the goal of this work is to provide a hardware framework that supports arbitrary model combinations; how to get the best combination scheme for a given benchmark is dataset-dependent and beyond the scope of this work. The best results for each dataset are in bold.

For the Cardio dataset, Table 5.5 shows that Loda always achieves the best Score AUC mean (0.933) and lower variance (0.00004) than RS-Hash, xStream or any listed combination strategy. However, in the label test, all combined labels can get a higher AUC than any single detector, albeit with higher variance.

In general, it can be seen that Loda, RS-Hash, and xStream perform differently for different datasets, and there is no single "best" detector or "ideal" combination. Combined strategies are not always superior to a single detector. However, a combined detector typically yields more reliable performance. For labels, the combined detector always returns a higher AUC. We believe this is because if any detector indicates an anomaly, the combined result is also an anomaly. This reduces the possibility of missing an anomaly and introducing a higher variance. We believe the strong dataset dependence coupled with the difficulty of finding the best combination of sub-detectors justifies the need to create a runtime reconfigurable framework that can support multiple detectors and multiple instances of each detector.

5.3.3 Pblock Assignment and FPGA Implementation

Table 5.6 shows a breakdown of resource utilization for the blocks in Figure 5.6. RP-1 to RP-7 account for most of the FPGA resources and are used for sub-detector implementation.

TABLE 5.6: Resource Partition of FPGA Blocks

Blocks	LUT	DSP	BRAM	FF
RP-1	6.73%	4.49%	6.67%	6.73%
RP-2	8.57%	7.54%	8.52%	8.57%
RP-3	6.24%	6.46%	6.39%	6.24%
RP-4	6.72%	4.49%	6.67%	6.72%
RP-5	6.24%	6.46%	6.39%	6.24%
RP-6	8.74%	8.24%	8.15%	8.74%
RP-7	7.32%	7.30%	7.22%	7.32%
RP-combo1	0.72%	0.56%	0.74%	0.72%
RP-combo2	0.59%	0.84%	0.83%	0.59%
RP-combo3	0.59%	0.84%	0.83%	0.59%
Switch-1	3.46%	4.49%	2.96%	3.46%
Switch-2	1.81%	0.98%	0%	1.82%
DMAs	2.25%	0%	1.30%	0.48%
DFX-Decouplers	0.04%	0%	0%	0.008%
AXI-InterConnect	0.67%	0%	0%	0.58%
AXI-SmartConnect	2.41%	0%	0%	1.61%
ALL	62.5%	52.69%	56.67%	60.42%

TABLE 5.7: Resource of 35 Loda, 25 RS-Hash, and 20 xStream for Cardio

Detector	LUT	DSP	BRAM	FF
Loda-35	16783(63.4%)	122(44.2%)	**54.5(79.0%)**	11478(21.7%)
RS-Hash-25	**23732(89.6%)**	68(24.6%)	50(72.5%)	14012(26.5%)
xStream-20	**23908(90.3%)**	80(29.0%)	60(87.0%)	12617(23.8%)
RP-3	**26480**	**276**	**69**	**52960**

The resource distribution of these blocks is not uniform since the floorplan was manually created to pass the Design Rule Check (DRC) due to the nonuniform resources located on the target FPGA (ZCU111). The resources for blocks to combine the results, noted as combo-pblocks, are minimal, this being LUT (0.63%), DSP (0.75%), BRAM (0.80%), and FF (0.63%). Overall, the partial reconfigurable blocks and two static switch blocks utilize 57.73% LUT, 52.69% DSP, 55.37% BRAM, and 57.74% FF resources. The remaining area implements the remaining interface, including DMAs, AXI-Interconnect, DFX-decoupler, *etc.*

An important issue affecting achievable parallelism is how many sub-detectors can be assigned to a pblock. We determine this number by using the smallest pblock (RP-3) as the target and calculate the maximum ensemble size for Loda, RS-Hash, and xStream. This exercise leads

TABLE 5.8: Accuracy and Execution Time Comparison between fSEAD and CPU for Detector: Loda

Dataset	AUC-S(CPU)	**AUC-S(FPGA)**	AUC-L(CPU)	**AUC-L(FPGA)**	Ex Time(CPU)	**Ex Time(FPGA)**	Speed-up
Cardio	0.9310	0.9311	0.6447	0.6412	13 ms	4.63 ms	2.81×
Shuttle	0.9923	0.9914	0.9490	0.9432	147 ms	34.23 ms	4.29×
SMTP-3	0.8501	0.8506	0.7666	0.7499	222 ms	39.31 ms	5.65×
HTTP-3	0.9937	0.9936	0.6415	0.6336	1396 ms	228.25 ms	6.12×

TABLE 5.9: Accuracy and Execution Time Comparison between fSEAD and CPU for Detector: RS-Hash

Dataset	AUC-S(CPU)	**AUC-S(FPGA)**	AUC-L(CPU)	**AUC-L(FPGA)**	Ex Time(CPU)	**Ex Time(FPGA)**	Speed-up
Cardio	0.8546	0.8524	0.6310	0.6274	15 ms	4.87 ms	3.08×
Shuttle	0.9915	0.9910	0.9543	0.9560	168 ms	35.80 ms	4.69×
SMTP-3	0.8525	0.8513	0.7166	0.7166	260 ms	39.63 ms	6.56×
HTTP-3	0.9944	0.9944	0.5065	0.5067	1490 ms	228.29 ms	6.53×

TABLE 5.10: Accuracy and Execution Time Comparison between fSEAD and CPU for Detector: xStream

Dataset	AUC-S(CPU)	**AUC-S(FPGA)**	AUC-L(CPU)	**AUC-L(FPGA)**	Ex Time(CPU)	**Ex Time(FPGA)**	Speed-up
Cardio	0.9229	0.9222	0.6467	0.6435	18 ms	4.82 ms	3.73×
Shuttle	0.9914	0.9905	0.9688	0.9680	250 ms	40.62 ms	6.15×
SMTP-3	0.8077	0.8076	0.7167	0.7167	366 ms	50.99 ms	7.18×
HTTP-3	0.9947	0.9948	0.5067	0.5069	2460 ms	297.85 ms	8.26×

to an ensemble size of $R = 35$ for Loda, $R = 25$ for RS-Hash, and $R = 20$ for xStream in each AD-pblock. The resources required are shown in Table 5.7. Thus, if we utilize all seven AD-pblocks to implement a homogeneous type of detector, the current configuration supports a maximum of 245 Loda, 175 RS-Hash, or 140 xStream sub-detectors.

5.3.4 Speed, Accuracy and Power Comparison

We achieve 80%-90% logic use of all seven partial blocks in fSEAD on the ZCU111 board for homogeneous detect implementations of Loda (245 sub-detectors), RS-Hash (175 sub-detectors) and xStream (140 sub-detectors). This configuration was configured using the PYNQ environment to test fSEAD performance. The FPGA was operated at a clock rate of 188 MHz. We also implemented a multi-threaded GCC version of the detectors with the same parameters and scale as fSEAD for comparison.

The detector system in fSEAD is configured with the topology shown in Figure 5.5(c). An averaging-based combination and an OR-Gate-based label combination techniques were used. The `ap_fixed<32, 16, AP_TRN, AP_WRAP>` type available in Xilinx Vivado HLS [62] was used for all inner non-integer operations. This has a word length of 32-bit, with 16 bits representing the integer part and 16 bits representing the fraction. Convergent rounding (`AP_TRN`) and saturating arithmetic (`AP_WRAP`) were used. To facilitate the use of the float32 type in the NUMPY library [171] for streaming data transfer with each module on the FPGA, all fSEAD IP interfaces are converted to float32. This does not cause a significant increase in resource consumption.

Tables 5.8 through 5.10 show the performance comparison of Loda, RS-Hash, and xStream detectors in terms of AUC, and execution time on the ZCU111 board and CPU, respectively. The execution time test uses the *time()* function in Python to measure the time from the start of the input DMA transfer to when all data is obtained from the output DMA. The GCC implementation was executed on a multi-threaded CPU on the target PC, and the *time* command in $bash$ was used to measure execution time. The pthread library is used to support the multi-threaded C implementation. Since each sub-detector in the ensemble is data-independent, we equally distribute the same number of sub-detectors to each CPU thread. The sub-score computed by multiple threads requires a synchronization operation at the end of each sample to compute the average score of the ensemble. The $pthread\_mutex\_lock$ and $pthread\_mutex\_unlock$ functions are placed between different threads to guarantee the streaming mode execution. The target CPU has 8 cores and 16 hardware threads; we tested switching the thread number from 1 to 16. Figure 5.9 shows the speedup results with different numbers of threads on the longest test case: xStream for HTTP-3, 4-thread always gets the best speedup. We believe this is due to synchronization overheads introduced by the $mutex$ scheme limiting performance when the number of threads exceeds 4 ($mutex$ is called in every streaming execution). Based on this result, all the following CPU experiments results are measured from a 4-thread configuration.

Regarding accuracy, very similar scores were obtained on CPU and FPGA platforms. This indicates the `ap_fixed<32,16>` variable type used can provide sufficient accuracy as

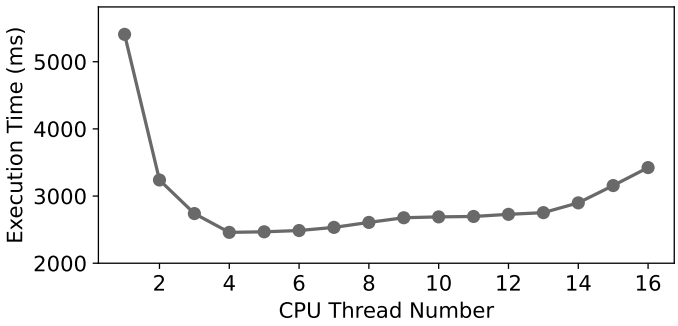


FIGURE 5.9: Multi-threaded CPU Implementation for xStream for HTTP-3

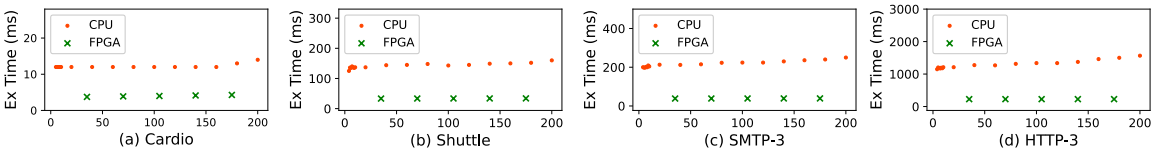


FIGURE 5.10: Execution Time Comparison of fSEAD and CPU for Detector: Loda.

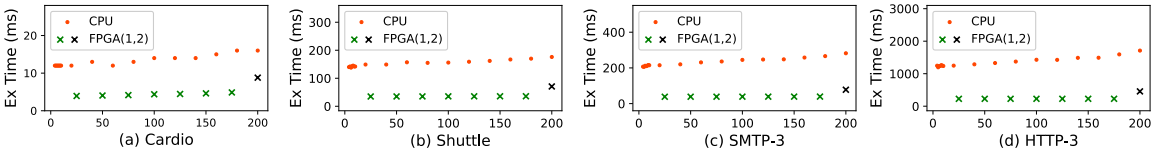


FIGURE 5.11: Execution Time Comparison of fSEAD and CPU for Detector: RS-Hash.

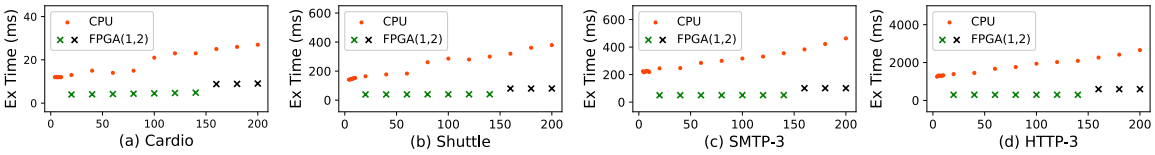


FIGURE 5.12: Execution Time Comparison of fSEAD and CPU for Detector: xStream.

the float32 variable type of CPU implementation. The minor differences are due to the accumulation of errors during the cumulative calculation of the accuracy differences generated by each `ap_fixed` and float32 type sub-detector. Future developers may wish to explore minimizing resource consumption further by reducing precision.

On the smallest dataset, Cardio, it can be observed that the FPGA obtains a speed-up of 2.81 for Loda, 3.08 for RS-Hash, and 3.73 times for xStream. The speedup ratio increases as

TABLE 5.11: Operation Number of fSEAD

Detectors	Operation Number
Loda	$OP = N * (2Rd + 7R + 2)$
RS-Hash	$OP = N * (5Rdw + 4Rd + 11Rw + R + 2)$
xStream	$OP = N * (2Rdk + 5Rdw + 15Rw + 2R + 2)$

[1] N: the length of the input dataset.
[2] d: the dimension of input dataset.
[3] R: the ensemble size.
[4] w: the hash functions number in CMS.
[5] k: the projection size of xStream.

TABLE 5.12: GOPS Comparison of CPU and fSEAD

GOPS	**CPU**				**fSEAD**			
	Cardio	**Shuttle**	**SMTP-3**	**HTTP-3**	**Cardio**	**Shuttle**	**SMTP-3**	**HTTP-3**
Loda	1.690	2.049	1.402	0.776	4.748	8.789	7.924	4.748
RS-Hash	6.772	6.353	4.197	4.331	20.858	29.797	27.533	28.282
xStream	15.427	11.050	6.623	5.878	57.544	67.959	47.554	48.551

the size and dimension of the dataset increases. On the largest dataset, HTTP-3, the three detectors achieve speedups of 6.12, 6.53, and 8.26, respectively. XStream achieves the highest speedup of 8.26 for dataset HTTP-3. For smaller datasets, the transfer time from the Linux OS-based host ARM processor to the FPGA becomes the bottleneck. As fSEAD is optimized for large datasets, this is not a significant issue.

It is worth mentioning that since the ensemble on fSEAD is based on sub-detector-level parallelism, its latency is not significantly affected by the ensemble size in the case that FPGA resources are sufficient to implement the required ensemble in parallel. In contrast, the ensemble implementation on GCC uses a *for* loop to iterate the computation of each sub-detector, so its execution time increases proportionally with the increase of the ensemble size. Figures 5.10 to Figures 5.12 show the execution times of the three detectors on the CPU and FPGA for multiple sets of experiments and different ensemble sizes. The red dots indicating CPU execution time show a linear increase with ensemble size. The green crosses indicate the execution time on FPGA, while the black cross results from two FPGA executions. In all cases, significant speed-ups are achieved over the CPU and limited only by FPGA resources.

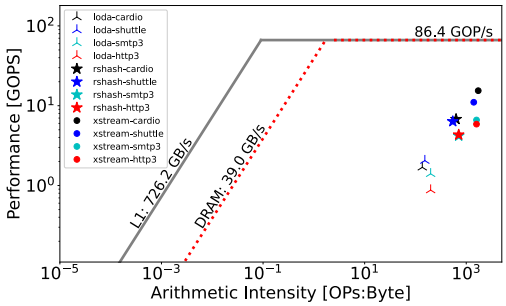


FIGURE 5.13: Roofline Model on CPU.

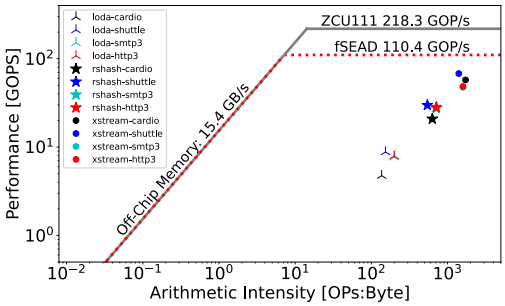


FIGURE 5.14: Roofline Model on FPGA.

We also use the roofline model to estimate how our target anomaly detector methods have been optimized. This describes an application's achieved performance and arithmetic intensity against the machine's maximum achievable performance in terms of memory bandwidth and peak computational performance. The CPU roofline model is measured on the Intel Advisor 2022 [172]. Billions of operations per second (GOPS) is used as the metric for Performance (y-axis in Figure 5.13), and operations (OPs) per byte is used for Arithmetic intensity (x-axis in Figure 5.13). The L1/DRAM bandwidth roofline represents the maximum amount of bytes that can get read or written for a given arithmetic intensity. For FPGA roofline models, we use the method in Ref. [173] to calculate the roofline chart in Figure 5.14. The arithmetic intensity (x-axis) is the number of arithmetic operations performed for each byte read or written to an off-chip memory (we assume 13.4 GB/s off-chip memory bandwidth [174]). The performance (y-axis) is calculated in GOPS.

For the three detectors, Table 5.11 shows the expressions we used to estimate the total number of operations to execute a target dataset. Using this and the execution time from Table 5.8 to Table 5.10, we compute the GOPS in Table 5.12. We use the highest GOPS (67.959 GOPS of xStream for Shuttle) to estimate the compute-bound performance for the target FPGA (ZCU111) and fSEAD structure, respectively. The xStream for Shuttle occupies 132391 LUTs, 476 DSPs, and 79485 FFs, which takes up 31.13% of the total FPGA resources and 61.57% of the fSEAD partial blocks resources. From this, we calculate the compute-bound performance for FPGA and fSEAD as 218.3 GOP/s and 110.4 GOP/s, respectively.

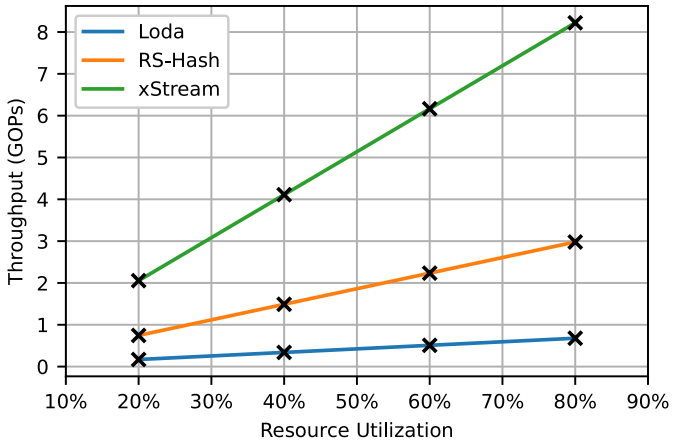


FIGURE 5.15: Example of the Scalability inside Single Partial Block: RP-1

While no algorithms reach the boundary of the roofline, xStream is closer to the computational boundary than Loda and RS-Hash, as seen in Figure 5.13 and Figure 5.14. We believe this is a function of the algorithms: the current three detectors are not extremely computationally intensive, but xStream has more vector and matrix multiplication operations among the current AD library.

Figure 5.15 shows the scalability of a single pblock, RP-1, using the Cardio dataset with 20% to 80% resource utilization for Loda, RS-Hash, and xStream. Working at the same 188 MHz clock frequency, the throughput of three detectors can be seen to vary linearly with the resource utilization of RP-1. This linear scalability enables one to quickly identify potential ensembles that will fit on a given FPGA and estimate their throughput. One could then perform software experiments, such as those in Table 5.5, to identify which of these potential ensembles provides the best accuracy for the target dataset.

The power consumption of fSEAD is separated into chip power and system power; Figure 5.16 and Figure 5.17 show the measured chip and system power consumption from Xilinx Vivado Tool [175] and EcoFlow RIVER Max Portable Power Station [176] respectively. In Figure 5.17, the idle system power measured on EcoFlow is 30 Watts. The working system power (35 Watts) is gained by configuring the PYNQ to invoke all pblocks for xStream with the biggest dataset HTTP-3, which matches the dynamic power of 5.232 Watts, as demonstrated in Figure 5.16. Intel(R) Core(TM) i7-10700F CPU power is measured with the powerstat

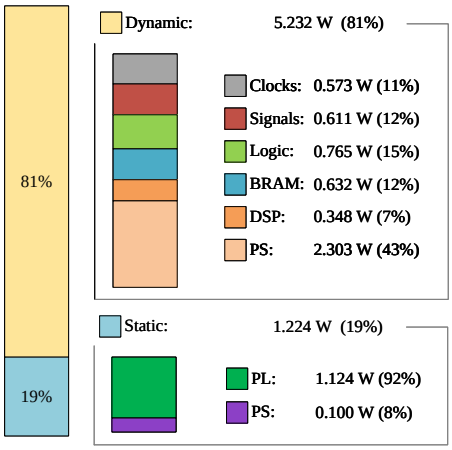


FIGURE 5.16: Chip Power Consumption.

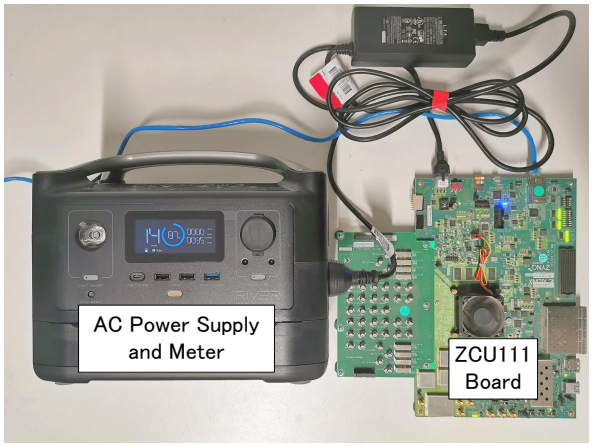


FIGURE 5.17: System Power Test-bed.

command using Running Average Power Limit (RAPL) domains. 120 samples of power measurements with 0.5-second steps are averaged over a one-minute time period. The CPU idle power is 7.90 Watts, and the CPU working power for xStream, which has the biggest dataset, HTTP-3, is 51.23 Watts. The dynamic CPU power (43.33 Watts) is more than 8× higher than the fSEAD dynamic power consumption on ZCU111 FPGA (5.232 Watts).

5.3.5 Partial Reconfiguration

Although the reconfiguration of each partial region in fSEAD can often be done when the system is idle, we measure the overhead of partial reconfiguration in the PYNQ framework. The horizontal axis of Table 5.13 shows all pblocks, and the vertical axis shows the direction of reconfiguration for bitstream downloads. For example, $Function \rightarrow Identity$ indicates that the configured logic is $Function$ which is overwritten with $Identity$. We have chosen a common function module: $Loda\_Cardio$ and $Averaging$ for (RP-1 to RP-7) and (COMBO1 to COMBO3), respectively, as the $Function$ module. $Identity$ is a design where the input is simply passed to the output. The objective is to evaluate the impact of the hardware logic size of the target bitstream on the reconfiguration time cost. Referring to the size of each pblock provided in Table 5.6 (all seven AD-pblock resources are larger than those of COMBO1 to COMBO3; among the seven AD-pblocks, RP-3 has the least programmable resources and RP-6 the most; while COMBO1 occupies the largest area of the three COMBO blocks).

TABLE 5.13: Partial Reconfiguration Time Cost (Unit is ms) of the Different Pblocks

BIT download directions	RP-1	RP-2	RP-3	RP-4	RP-5	RP-6	RP-7	COMBO1	COMBO2	COMBO3
$Function \rightarrow Identity$	607.8	606.1	604.5	606.1	608.9	**609.6**	609.5	587.2	582.7	**579.8**
$Identity \rightarrow Function$	606.3	611.3	607.2	606.0	606.9	608.1	607.5	582.9	580.1	581.9

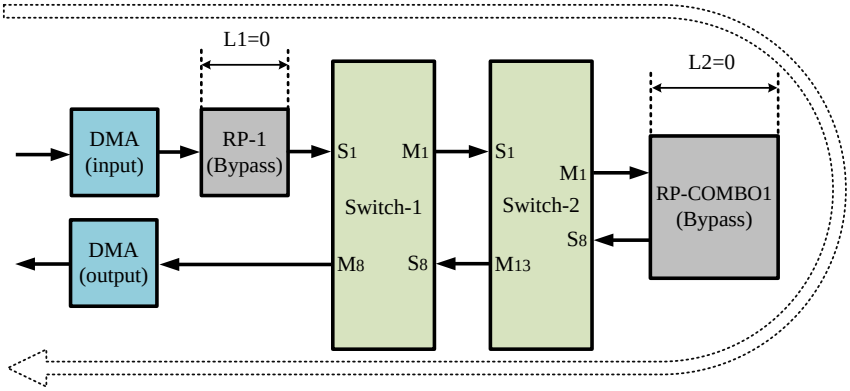


FIGURE 5.18: Example of fSEAD Channel with Empty Logic.

The latencies reported in Table 5.13 are the total partial reconfiguration latency. This will be a function of the size and location of the region, as well as the size of the partial bitstream and the chosen FPGA. Since we do not have the ability to modify the chosen FPGA, in this section, we explore how these other factors impact reconfiguration time. The results in Table 5.13 show that the pblock with larger area takes a little more time to reconfigure, *e.g.*, RP-6 takes 609.6 ms for updating $Loda\_Cardio$ with $Identity$. The pblock with the smallest area uses the shortest time *i.e.*, COMBO3 used 579.8 ms to reconfigure $Loda\_Cardio$ to $Identity$. In addition, the complexity of the target bitstream has a slight impact on the reconfiguration time. Apart from RP-2, RP-3, and COMBO3, a general trend indicates that the simpler the logic of the target bitstream (in this case, $Identity$), the lower the reconfiguration time.

While fSEAD is focused on ensemble-centric anomaly detection, different real-life detectors in the literature can be incorporated into this library for higher flexibility and salable combinations. Moreover, it can potentially be used for more general machine-learning applications by adding different modules. To evaluate the default latency of fSEAD interconnections, since this is a key metric that could affect design decisions, Figure 5.18 illustrates a data path where each pblock simply copies its input to output (the "Bypass" in Figure 5.18 is the same as $Identity$ in Table 5.13). The execution time, which is a measure of system latency overhead,

is **0.80 ms**. Thus, for pblocks with latency L1 on the left and L2 on the right, the maximum latency of the system is $\approx$ **0.80+L1+L2 (ms)**. We also measure the shorter DMA latency where the data path includes DMA (input), L1, Switch-1, and DMA (output); the latency is **0.77 ms**. This reflects that the default system latency is dominated by the Linux OS-based PYNQ framework rather than by the routing latency of switches.

Compared to static hardware, the overhead introduced by the proposed dynamic framework includes the system time required to update partial blocks, additional interconnection latency, and the achievable maximum clock frequency (affected by the manually partitioned pblocks). The first two aspects are discussed in Table 5.13 and Figure 5.18, while the last aspect is not quantified in this thesis and is left for future work.

5.4 Summary

In this Chapter, we proposed a flexible computing architecture consisting of multiple partially reconfigurable regions, called pblocks, interconnected via the AXI-Streaming Switches. The scheme enables parallel constructions of repeating blocks to be easily scaled to fill the FPGA. We also demonstrated a concrete application of this architecture to streaming anomaly detection using ensembles (fSEAD). fSEAD allows complex and more powerful anomaly detectors to be composed of simple pblocks in an arbitrary fashion. Careful floorplanning of the pblocks and switches, together with bus-level pipelining, minimizes routing delay and allows timing closure to be achieved. Through a number of experiments involving ensembles of three sub-detectors (Loda, RS-Hash, and xStream), we demonstrate a 3 to 8$\times$ speedup compared with a CPU.

CHAPTER 6

Conclusion

This thesis explored the development of specialized FPGA-based solutions aimed at enhancing ML tasks for edge applications. The research focused on three key areas: (1) improving the accuracy of ML accelerators in an area-efficient manner, (2) integrating these accelerators into a cohesive SoC architecture, and (3) composing partial blocks within the SoC to adapt to varying environmental conditions. These topics are thoroughly discussed across three chapters: circuit-level accelerators in Chapter 3, system-level applications in Chapter 4, and composable tools in Chapter 5.

First, for ultra-low latency tasks in the nanosecond range, we introduced the LUTEnsemble architecture, a novel approach to DNN inference on FPGAs. Traditional LUT-based DNNs face accuracy challenges due to the exponential growth of lookup table consumption. LUTEnsemble addresses this by enhancing the connectivity of LUT-based DNNs. The results prove that the ensemble technique and sub-neural adder tree structures could provide a scalable approach to improved accuracy. Compared with state-of-the-art works, LUTEnsemble sets new benchmarks for LUT-based DNN inference on FPGAs. However, a gap remains between LUT-based DNNs and fully connected DNNs; existing limitations and potential solutions are discussed in future work.

Second, the thesis explored the application of FPGAs in qubit detection within trapped-ion quantum information processing systems. The LUTEnsemble was integrated into this system as a 2D-image classifier. Additionally, for a more accuracy-tailored solution in the microsecond range, we proposed a Vision Transformer (ViT) accelerator built on the $hls4ml$ framework, which incorporates several simplifications and optimizations to make it practical for FPGA implementation. It supports an arbitrary reuse factor to parameterize the degree of

parallelism in matrix-vector multiplication, allowing customization of the balance between latency and area for edge-device tasks. Both LUTEnsemble and ViT outperform the traditional thresholding classifier; for instance, the MMF error of LUTEnsemble is 4.22× lower than thresholding in the three-qubit test. ViT further outperforms with 1.13× and 7.6× lower error rates than CNN and thresholding, respectively. Compared to a GPU-based ViT, the total detection latency for one- and three-qubit tests on the FPGA showed reductions of 119× and 94×, respectively. Interestingly, the ViT latency accounts for only 0.89-1.53% of the total detection latency, with the bottleneck now lying in the EMCCD and transmission via Cameralink. Thus, this work could contribute to enhancing the performance of next-generation qubit detection systems.

Finally, the research extended to developing a composable tool for FPGA-based SoCs, with a focus on partial reconfiguration. The proposed framework consists of multiple partially reconfigurable blocks, referred to as pblocks, which are interconnected via an AXI switch. This setup allows for arbitrary composition of these blocks at runtime, facilitating the merging and combining of results to form an ensemble. The study demonstrates how the ensemble technique could enhance hardware scalability within a tool. Through a series of experiments involving ensembles of three sub-detectors (Loda, RS-Hash, and xStream) across four public datasets, we observed a 3 to 8× speedup compared to a CPU. These results also highlight that timing closure for this framework can be effectively achieved through pblocks and switches floorplanning combined with bus-level pipelining.

6.1 Future outlook

While the FPGA-based machine learning hardware presented in this thesis offers significant advancements, several research directions remain to enhance its practicality in real-world applications.

In the LUT-based DNN inference framework introduced in Chapter 3, the current implementation is limited to supporting the MLP architecture. A critical direction for future research is to extend LUT-based implementations to more advanced DNN models, such as CNNs,

Transformer [17], and the emerging Kolmogorov-Arnold Networks (KAN) [177]. Achieving this will not only require optimizations at the DNN model/architectural/circuit level but also necessitates corresponding improvements in the associated tools, such as quantization tools (like Brevitas [69]) and model development platforms (like PyTorch [25]), to fully support these advanced models. Furthermore, the sparsity patterns of sub-models in the ensemble are currently selected randomly without targeted optimization. We believe that a selective ensemble approach, which strategically chooses sparsity patterns, could offer advantages over the current random selection method. Moreover, for non-image input data that lacks a well-defined notion of neighborhood, developing an effective Mixer layer represents another promising direction for future research. Lastly, in PolyLUT-Add, the generality of the current adder tree is limited for polynomial degrees greater than 1. We believe that addressing this limitation through optimization in future work could result in further performance improvements.

For the qubit detection system explored in Chapter 4, retraining the DNN model using real-world images—accounting for imperfections such as variations in temperature and optical aberrations—could further increase its efficacy. Achieving this requires a meticulously designed dataset collection process that reflects the operational conditions of the system. Additionally, improving the simulation dataset generator to more closely mimic real-world characteristics would facilitate quicker validation with more reliable simulation datasets. Together, these improvements could significantly increase the system's practicality for the trapped-ion research community.

Lastly, for the composable FPGA framework discussed in Chapter 5, a key future direction is to enhance its programmability by developing module generators capable of automatically producing optimized HLS directives and detector parameters, such as ensemble size. Expanding the framework's library to include a broader range of anomaly detection methods would further enhance its capabilities. Additionally, leveraging the composable framework for qubit detection to facilitate the reuse of diverse ML solutions, including LUTEnsemble and ViT, on a single FPGA die represents another promising avenue for research.

Bibliography

[1] Binglei Lou et al. 'PolyLUT-Add: FPGA-based LUT Inference with Wide Inputs'. In: *2024 34th International Conference on Field-Programmable Logic and Applications (FPL)* (2024), pp. 149–155.

[2] Binglei Lou, David Boland and Philip H.W. Leong. 'fSEAD: A Composable FPGA-Based Streaming Ensemble Anomaly Detection Library'. In: *ACM Transactions on Reconfigurable Technology and Systems* 16.3 (2023), pp. 1–27.

[3] Boming Huang et al. 'Automated trading systems statistical and machine learning methods and hardware implementation: a survey'. In: *Enterprise Information Systems* 13.1 (2019), pp. 132–144.

[4] Chance Tarver, Alexios Balatsoukas-Stimming and Joseph R Cavallaro. 'Design and implementation of a neural network based predistorter for enhanced mobile broadband'. In: *2019 IEEE international workshop on signal processing systems (SiPS)*. IEEE. 2019, pp. 296–301.

[5] Mohamed Elsayed et al. 'Low complexity neural network structures for self-interference cancellation in full-duplex radio'. In: *IEEE Communications Letters* 25.1 (2020), pp. 181–185. ISSN: 1089-7798.

[6] Shih-Chieh Lin et al. 'The architectural implications of autonomous driving: Constraints and acceleration'. In: *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. 2018, pp. 751–766.

[7] Euan Jones et al. 'Autonomous driving developed with an FPGA design'. In: *2019 International Conference on Field-Programmable Technology (ICFPT)*. IEEE. 2019, pp. 431–434.

[8] Javier Duarte et al. 'Fast inference of deep neural networks in FPGAs for particle physics'. In: *Journal of Instrumentation* 13.07 (2018), P07027.

[9] Tadej Murovic and Andrej Trost. ‘Massively parallel combinational binary neural networks for edge processing’. In: *Elektrotehniski Vestnik* 86.1/2 (2019), pp. 47–53.

[10] Duncan JM Moss et al. ‘Real-time FPGA-based anomaly detection for radio frequency signals’. In: *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*. Florence, Italy: IEEE, 2018, pp. 1–5. ISBN: 1538648814.

[11] ‘Edge AI Hardware Market’. In: URL: https://market.us/report/edge-ai-hardware-market/.

[12] Yoav Freund and Robert E Schapire. ‘A decision-theoretic generalization of on-line learning and an application to boosting’. In: *Journal of computer and system sciences* 55.1 (1997), pp. 119–139. ISSN: 0022-0000.

[13] Yoav Freund and Robert E Schapire. ‘Experiments with a new boosting algorithm’. In: *icml*. Vol. 96. Bari, Italy: Citeseer, 1996, pp. 148–156.

[14] Marta Andronic and George A Constantinides. ‘PolyLUT: learning piecewise polynomials for ultra-low latency FPGA LUT-based inference’. In: *2023 International Conference on Field Programmable Technology (ICFPT)*. IEEE. 2023, pp. 60–68.

[15] Xilinx Inc. *Python productivity for zynq*. 2022. URL: http://www.pynq.io/home.html.

[16] Kurt Hornik, Maxwell Stinchcombe et al. ‘Multilayer feedforward networks are universal approximators’. In: *Neural networks* 2.5 (1989), pp. 359–366.

[17] Ashish Vaswani et al. ‘Attention is all you need’. In: *Advances in neural information processing systems* 30 (2017).

[18] Jean-Baptiste Cordonnier, Andreas Loukas and Martin Jaggi. ‘On the Relationship between Self-Attention and Convolutional Layers’. In: *ArXiv* abs/1911.03584 (2019).

[19] Alexey Dosovitskiy et al. ‘An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale’. In: *ArXiv* abs/2010.11929 (2020).

[20] Léon Bottou. ‘Stochastic gradient descent tricks’. In: *Neural networks: Tricks of the trade*. Springer, 2012, pp. 421–436.

[21] Shiliang Sun et al. ‘A survey of optimization methods from a machine learning perspective’. In: *IEEE transactions on cybernetics* 50.8 (2019), pp. 3668–3681. ISSN: 2168-2267.

[22] Michael A Nielsen. *Neural networks and deep learning*. Vol. 25. Determination press San Francisco, CA, USA, 2015.

[23] Adam Paszke et al. 'PyTorch: An Imperative Style, High-Performance Deep Learning Library'. In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[24] Boris T Polyak. 'Some methods of speeding up the convergence of iteration methods'. In: *Ussr computational mathematics and mathematical physics* 4.5 (1964), pp. 1–17. ISSN: 0041-5553.

[25] Adam Paszke et al. 'PyTorch: An Imperative Style, High-Performance Deep Learning Library'. In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[26] Zewen Li et al. 'A survey of convolutional neural networks: analysis, applications, and prospects'. In: *IEEE transactions on neural networks and learning systems* 33.12 (2021), pp. 6999–7019.

[27] Dan Hendrycks and Kevin Gimpel. 'Gaussian error linear units (gelus)'. In: *arXiv preprint arXiv:1606.08415* (2016).

[28] Varun Chandola, Arindam Banerjee and Vipin Kumar. 'Anomaly detection: A survey'. In: *ACM computing surveys (CSUR)* 41.3 (2009), pp. 1–58. ISSN: 0360-0300.

[29] Junshui Ma and Simon Perkins. 'Online novelty detection on temporal sequences'. In: *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*. New York, NY, USA: Association for Computing Machinery, 2007, pp. 613–618.

[30] Sridhar Ramaswamy, Rajeev Rastogi and Kyuseok Shim. 'Efficient algorithms for mining outliers from large data sets'. In: *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*. New York, NY, USA: Association for Computing Machinery, 2000, pp. 427–438.

[31] Markus M Breunig et al. ‘LOF: identifying density-based local outliers’. In: *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*. Vol. 29. New York, NY, USA: ACM, 2000, pp. 93–104.

[32] Mei-Ling Shyu et al. *A novel anomaly detection scheme based on principal component classifier*. Report. Miami Univ Coral Gables Fl Dept of Electrical and Computer Engineering, 2003.

[33] Yue Zhao et al. ‘SUOD: Accelerating Large-Scale Unsupervised Heterogeneous Outlier Detection’. In: *Proceedings of Machine Learning and Systems* 3 (2021), pp. 463–478.

[34] Selim F. Yilmaz and Suleyman Serdar Kozat. ‘PySAD: A Streaming Anomaly Detection Framework in Python’. In: *ArXiv* abs/2009.02572 (2020).

[35] Tomáš Pevný. ‘Loda: Lightweight on-line detector of anomalies’. In: *Machine Learning* 102.2 (2016), pp. 275–304. ISSN: 0885-6125.

[36] Saket Sathe and Charu C Aggarwal. ‘Subspace outlier detection in linear time with randomized hashing’. In: *2016 IEEE 16th International Conference on Data Mining (ICDM)*. Barcelona: IEEE, 2016, pp. 459–468. ISBN: 1509054731.

[37] Emaad Manzoor, Hemank Lamba and Leman Akoglu. ‘xstream: Outlier detection in feature-evolving data streams’. In: *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. New York, NY, USA: Association for Computing Machinery, 2018, pp. 1963–1972.

[38] Fei Tony Liu, Kai Ming Ting and Zhi-Hua Zhou. ‘Isolation forest’. In: *2008 eighth ieee international conference on data mining*. USA: IEEE Computer Society, 2008, pp. 413–422. ISBN: 076953502X.

[39] Ke Wu et al. ‘RS-Forest: A Rapid Density Estimator for Streaming Anomaly Detection’. In: *2014 IEEE International Conference on Data Mining*. Shenzhen: IEEE Computer Society, 2014, pp. 600–609. DOI: 10.1109/ICDM.2014.45.

[40] Swee Chuan Tan, Kai Ming Ting and Tony Fei Liu. ‘Fast anomaly detection for streaming data’. In: *Twenty-Second International Joint Conference on Artificial Intelligence*. Barcelona, Catalonia, Spain: AAAI Press, 2011, pp. 1511–1516.

[41] Mayu Sakurada and Takehisa Yairi. 'Anomaly detection using autoencoders with non-linear dimensionality reduction'. In: *Proceedings of the MLSDA 2014 2nd workshop on machine learning for sensory data analysis*. New York, NY, USA: Association for Computing Machinery, 2014, pp. 4–11.

[42] Thomas Schlegl et al. 'f-AnoGAN: Fast unsupervised anomaly detection with generative adversarial networks'. In: *Medical image analysis* 54 (2019), pp. 30–44. ISSN: 1361-8415.

[43] Jia Deng et al. 'Imagenet: a large-scale hierarchical image database'. In: *IEEE Conference on Computer Vision and Pattern Recognition*. IEEE. 2009, pp. 248–255.

[44] Xulei Yang et al. 'Deep learning for practical image recognition: case study on kaggle competitions'. In: *The 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2018, pp. 923–931.

[45] Balaji Lakshminarayanan, Alexander Pritzel and Charles Blundell. 'Simple and scalable predictive uncertainty estimation using deep ensembles'. In: *Advances in Neural Information Processing Systems* 30 (2017).

[46] Marton Havasi et al. 'Training independent subnetworks for robust prediction'. In: *International Conference on Learning Representations*. 2021. URL: https://openreview.net/forum?id=OGg9XnKxFAH.

[47] Alexandre Ramé, Rémy Sun and Matthieu Cord. 'Mixmo: Mixing multiple inputs for multiple outputs via deep subnetworks'. In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2021, pp. 823–833.

[48] Martin Ferianc and Miguel Rodrigues. 'MIMMO: Multi-Input Massive Multi-Output Neural Network'. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2023, pp. 4564–4569.

[49] Elke Achtert et al. 'Visual evaluation of outlier detection models'. In: *International Conference on Database Systems for Advanced Applications*. Berlin: Springer, 2010, pp. 396–399.

[50] Markus Hofmann and Ralf Klinkenberg. *RapidMiner: Data mining use cases and business analytics applications*. New York: CRC Press, 2016. ISBN: 1498759866.

[51] Lukasz Komsta and Maintainer Lukasz Komsta. ‘Package ‘outliers’’. In: *Medical University of Lublin, Lublin* (2011).

[52] Yue Zhao, Zain Nasrullah and Zheng Li. ‘PyOD: A Python Toolbox for Scalable Outlier Detection’. In: *Journal of Machine Learning Research* 20.96 (2019), pp. 1–7. URL: http://jmlr.org/papers/v20/19-011.html.

[53] Yue Zhao et al. ‘Combining machine learning models using Combo library’. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 34. 09. 2020, pp. 13648–13649.

[54] Arthur Zimek, Ricardo JGB Campello and Jörg Sander. ‘Ensembles for unsupervised outlier detection: challenges and research questions a position paper’. In: *Acm Sigkdd Explorations Newsletter* 15.1 (2014), pp. 11–22. ISSN: 1931-0145.

[55] Aleksandar Lazarevic and Vipin Kumar. ‘Feature bagging for outlier detection’. In: *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*. New York, NY, USA: Association for Computing Machinery, 2005, pp. 157–166.

[56] Charu C Aggarwal and Saket Sathe. ‘Theoretical foundations and algorithms for outlier ensembles’. In: *Acm sigkdd explorations newsletter* 17.1 (2015), pp. 24–47. ISSN: 1931-0145.

[57] Tin Kam Ho, Jonathan J. Hull and Sargur N. Srihari. ‘Decision combination in multiple classifier systems’. In: *IEEE transactions on pattern analysis and machine intelligence* 16.1 (1994), pp. 66–75. ISSN: 0162-8828.

[58] Yue Zhao et al. ‘LSCP: Locally selective combination in parallel outlier ensembles’. In: *Proceedings of the 2019 SIAM International Conference on Data Mining*. SIAM. 2019, pp. 585–593.

[59] Jeremy Fowers et al. ‘A performance and energy comparison of convolution on GPUs, FPGAs, and multicore processors’. In: *ACM Transactions on Architecture and Code Optimization (TACO)* 9.4 (2013), pp. 1–21.

[60] Yunxiang Hu, Yuhao Liu and Zhuovuan Liu. ‘A survey on convolutional neural network accelerators: GPU, FPGA and ASIC’. In: *2022 14th International Conference on Computer Research and Development (ICCRD)*. IEEE. 2022, pp. 100–107.

[61] Mario Vestias and Horácio Neto. 'Trends of CPU, GPU and FPGA for high-performance computing'. In: *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE. 2014, pp. 1–6.

[62] Xilinx Inc. *Xilinx. Vivado Design Suite User Guide: High-Level Synthesis (UG902)*. 2020. URL: https://docs.xilinx.com/v/u/en-US/ug902-vivado-high-level-synthesis.

[63] Intel. *Intel High Level Synthesis Compiler*. 2024. URL: https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html.

[64] Jennifer Ngadiuba et al. 'Compressing deep neural networks on FPGAs to binary and ternary precision with hls4ml'. In: *Machine Learning: Science and Technology* 2.1 (2020), p. 015001.

[65] Mathworks. *HDL Coder*. 2024. URL: https://au.mathworks.com/help/hdlcoder/.

[66] Aaftab Munshi. 'The opencl specification'. In: *2009 IEEE Hot Chips 21 Symposium (HCS)*. IEEE. 2009, pp. 1–314.

[67] *qkeras*. 2024. URL: https://github.com/google/qkeras.

[68] Tianyi Zhang et al. 'Qpytorch: A low-precision arithmetic simulation framework'. In: *2019 Fifth Workshop on Energy Efficient Machine Learning and Cognitive Computing-NeurIPS Edition (EMC2-NIPS)*. IEEE. 2019, pp. 10–13.

[69] Alessandro Pappalardo. *Xilinx/brevitas*. 2023. DOI: 10.5281/zenodo.3333552. URL: https://doi.org/10.5281/zenodo.3333552.

[70] Yaman Umuroglu et al. 'FINN: A framework for fast, scalable binarized neural network inference'. In: *Proceedings of the 2017 ACM/SIGDA international symposium on field-programmable gate arrays*. 2017, pp. 65–74.

[71] Xiaofan Zhang et al. 'DNNBuilder: An automated tool for building high-performance DNN hardware accelerators for FPGAs'. In: *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE. 2018, pp. 1–8.

[72] Xilinx Inc. *Xilinx. Vivado Design Suite User Guide: Dynamic Function eXchange (UG909)*. 2020. URL: https://docs.xilinx.com/r/en-US/ug909-vivado-partial-reconfiguration.

[73] Xilinx Inc. *Xilinx. Vivado Design Suite Tutorial: Partial Reconfiguration (UG947)*. 2020. URL: https://docs.xilinx.com/r/en-US/ug947-vivado-partial-reconfiguration-tutorial.

[74] Andrew Elbert Wilson. 'Dynamic Reconfigurable Real-Time Video Processing Pipelines on SRAM-Based FPGAs'. Thesis. Brigham Young University, 2020.

[75] Dirk Koch et al. 'Partial reconfiguration on FPGAs in practice—Tools and applications'. In: *ARCS 2012*. IEEE, pp. 1–12. ISBN: 3000379223.

[76] Biruk Seyoum et al. 'Spatio-temporal optimization of deep neural networks for reconfigurable fpga socs'. In: *IEEE Transactions on Computers* 70.11 (2020), pp. 1988–2000. ISSN: 0018-9340.

[77] Florian Kästner et al. 'Hardware/software codesign for convolutional neural networks exploiting dynamic partial reconfiguration on pynq'. In: *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, pp. 154–161. ISBN: 1538655551.

[78] Eman Youssef et al. 'Energy adaptive convolution neural network using dynamic partial reconfiguration'. In: *2020 IEEE 63rd International Midwest Symposium on Circuits and Systems (MWSCAS)*. IEEE, pp. 325–328. ISBN: 1728180589.

[79] Zhuwei Qin et al. 'CaptorX: A Class-Adaptive Convolutional Neural Network Reconfiguration Framework'. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41.3 (2021), pp. 530–543. ISSN: 0278-0070.

[80] Paolo Meloni et al. 'A high-efficiency runtime reconfigurable IP for CNN acceleration on a mid-range all-programmable SoC'. In: *2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE, pp. 1–8. ISBN: 1509037071.

[81] Sandeep Mavadia et al. 'Prediction and real-time compensation of qubit decoherence via machine learning'. In: *Nature communications* 8.1 (2017), p. 14106.

[82] Riddhi Swaroop Gupta and Michael J Biercuk. 'Machine learning for predictive estimation of qubit dynamics subject to dephasing'. In: *Physical Review Applied* 9.6 (2018), p. 064042.

[83] Yang Liu et al. 'Minimization of the micromotion of trapped ions with artificial neural networks'. In: *Applied Physics Letters* 119.13 (2021), p. 134002.

[84] Alireza Seif et al. 'Machine learning assisted readout of trapped-ion qubits'. In: *Journal of Physics B: Atomic, Molecular and Optical Physics* 51.17 (2018), p. 174006.

[85] Junho Jeong, Changhyun Jung, Taehyun Kim et al. 'Using machine learning to improve multi-qubit state discrimination of trapped ions from uncertain EMCCD measurements'. In: *Optics Express* 31.21 (2023), pp. 35113–35130.

[86] Inc PULNiX America. *Specifications of the Camera Link Interface Standard for Digital Cameras and Frame Grabbers*. Oct. 2000.

[87] Wojciech Hubert Zurek. 'Decoherence, einselection, and the quantum origins of the classical'. In: *Reviews of modern physics* 75.3 (2003), p. 715.

[88] Peter W Shor. 'Scheme for reducing decoherence in quantum computer memory'. In: *Physical review A* 52.4 (1995), R2493.

[89] Andrew Steane. 'Multiple-particle interference and quantum error correction'. In: *Proceedings of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences* 452.1954 (1996), pp. 2551–2577.

[90] David S. Wang, Austin G. Fowler and Lloyd C. L. Hollenberg. 'Surface code quantum computing with error rates over 1%'. In: *Physical Review A* 83.2 (Feb. 2011). DOI: 10.1103/physreva.83.020302. URL: https://doi.org/10.1103/physreva.83.020302.

[91] AH Myerson et al. 'High-fidelity readout of trapped-ion qubits'. In: *Physical Review Letters* 100.20 (2008), p. 200502.

[92] Sebastian Halama et al. 'Real-time capable CCD-based individual trapped-ion qubit measurement'. In: *arXiv preprint arXiv:2204.09112* (2022).

[93] AH Burrell et al. 'Scalable simultaneous multiqubit readout with 99. 99% single-shot fidelity'. In: *Physical Review A—Atomic, Molecular, and Optical Physics* 81.4 (2010), p. 040302.

[94] Juan M Pino et al. ‘Demonstration of the trapped-ion quantum CCD computer architecture’. In: *Nature* 592.7853 (2021), pp. 209–213.

[95] S Olmschenk et al. ‘Manipulation and detection of a trapped Yb1 ion hyperfine qubit’. In: *Preprint at Ęhttp://arxiv. org/abs/0708.0657 ę* (2007).

[96] Vlad Negnevitsky et al. ‘Repeated multi-qubit readout and feedback with a mixed-species trapped-ion register’. In: *Nature* 563.7732 (2018), pp. 527–531.

[97] The iXon Ultra 897 EMCCD is described at https://andor.oxinst.com/products/ixon-emccd-camera-series/ixon-ultra-897.

[98] David Dussault and Paul Hoess. ‘Noise performance comparison of ICCD with CCD and EMCCD cameras’. In: *Infrared Systems and Photoelectronic Technology*. Ed. by C. Bruce Johnson, Eustace L. Dereniak and Robert E. Sampson. Vol. 5563. International Society for Optics and Photonics. Bellingham WA: SPIE, 2004, pp. 195–204. DOI: 10.1117/12.561839. URL: https://doi.org/10.1117/12.561839.

[99] Nick Schwegler. ‘Towards Low-Latency Parallel Readout of Multiple Trapped Ions’. MA thesis. ETH Zurich, 2018.

[100] S. Halama et al. *Real-time capable CCD-based individual trapped-ion qubit measurement*. 2022. DOI: 10.48550/ARXIV.2204.09112. URL: https://arxiv.org/abs/2204.09112.

[101] AH Burrell et al. ‘Scalable simultaneous multiqubit readout with 99. 99% single-shot fidelity’. In: *Physical Review A* 81.4 (2010), p. 040302.

[102] Md Maruf Hossain Shuvo et al. ‘Efficient acceleration of deep learning inference on resource-constrained edge devices: A review’. In: *Proceedings of the IEEE* 111.1 (2022), pp. 42–91.

[103] Erwei Wang et al. ‘LUTNet: Rethinking inference in FPGA soft logic’. In: *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE. 2019, pp. 26–34.

[104] Yaman Umuroglu et al. ‘LogicNets: Co-designed neural networks and circuits for extreme-throughput applications’. In: *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE. 2020, pp. 291–297.

[105] Mahdi Nazemi, Ghasem Pasandi and Massoud Pedram. 'Energy-efficient, low-latency realization of neural networks through boolean logic minimization'. In: *Proceedings of the 24th Asia and South Pacific design automation conference*. 2019, pp. 274–279.

[106] Marta Andronic and George A Constantinides. 'NeuraLUT: Hiding Neural Network Density in Boolean Synthesizable Functions'. In: *arXiv preprint arXiv:2403.00849* (2024).

[107] Robert M Bell and Yehuda Koren. 'Lessons from the netflix prize challenge'. In: *Acm Sigkdd Explorations Newsletter* 9.2 (2007), pp. 75–79. ISSN: 1931-0145.

[108] Xilinx Inc. *Vivado Design Suite User Guide (UG949)*. 2020. URL: https://docs.xilinx.com/v/u/2020.1-English/ug949-vivado-design-methodology.

[109] Li Deng. 'The MNIST database of handwritten digit images for machine learning research [best of the web]'. In: *IEEE signal processing magazine* 29.6 (2012), pp. 141–142.

[110] Jennifer Ngadiuba et al. 'Compressing deep neural networks on FPGAs to binary and ternary precision with hls4ml'. In: *Machine Learning: Science and Technology* 2.1 (2020), p. 015001.

[111] Claudionor N Coelho et al. 'Automatic heterogeneous quantization of deep neural networks for low-latency inference on the edge for particle detectors'. In: *Nature Machine Intelligence* 3.8 (2021), pp. 675–686.

[112] Farah Fahim et al. 'hls4ml: An open-source codesign workflow to empower scientific low-power machine learning devices'. In: *arXiv preprint arXiv:2103.05579* (2021).

[113] Tadej Murovič and Andrej Trost. 'Genetically optimized massively parallel binary neural networks for intrusion detection systems'. In: *Computer Communications* 179 (2021), pp. 1–10.

[114] Nour Moustafa and Jill Slay. 'UNSW-NB15: a comprehensive data set for network intrusion detection systems (UNSW-NB15 network data set)'. In: *2015 military communications and information systems conference (MilCIS)*. IEEE. 2015, pp. 1–6.

[115] Ekim Yurtsever et al. 'A survey of autonomous driving: common practices and emerging technologies'. In: *IEEE Access* 8 (2020), pp. 58443–58469.

[116] Igor Kononenko. ‘Machine learning for medical diagnosis: history, state of the art and perspective’. In: *Artificial Intelligence in Medicine* 23.1 (2001), pp. 89–109.

[117] Marzieh Fathi et al. ‘Big data analytics in weather forecasting: a systematic review’. In: *Archives of Computational Methods in Engineering* 29.2 (2022), pp. 1247–1275.

[118] Lov K Grover. ‘Fixed-point quantum search’. In: *Physical Review Letters* 95.15 (2005), p. 150501.

[119] Peter W Shor. ‘Algorithms for quantum computation: discrete logarithms and factoring’. In: *Proceedings 35th annual symposium on foundations of computer science*. Ieee. 1994, pp. 124–134.

[120] Andrew M Childs et al. ‘Exponential algorithmic speedup by a quantum walk’. In: *Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*. 2003, pp. 59–68.

[121] Ashley Montanaro. ‘Quantum algorithms: an overview’. In: *npj Quantum Information* 2.1 (2016), pp. 1–8.

[122] David J Wineland et al. ‘Experimental issues in coherent quantum-state manipulation of trapped atomic ions’. In: *Journal of research of the National Institute of Standards and Technology* 103.3 (1998), p. 259.

[123] QA Turchette et al. ‘Deterministic entanglement of two trapped ions’. In: *Physical Review Letters* 81.17 (1998), p. 3631.

[124] Patricia J Lee. ‘Quantum information processing with two trapped cadmium ions’. PhD thesis. University of Michigan, 2006.

[125] Alistair Robertson Milne. ‘Construction of a linear ion trap and engineering controlled spin-motional interactions’. PhD thesis. The University of Sydney, 2021.

[126] Ryszard Horodecki et al. ‘Quantum entanglement’. In: *Reviews of modern physics* 81.2 (2009), p. 865.

[127] Tom Manovitz. ‘Individual Addressing and Imaging of Ions in a Paul Trap’. MA thesis. Weizmann Institute of Science, 2016.

[128] CL Edmunds et al. ‘Scalable hyperfine qubit state detection via electron shelving in the 2 D 5/2 and 2 F 7/2 manifolds in 171 Yb+’. In: *Physical Review A* 104.1 (2021), p. 012606.

[129] The AXI4-Stream Infrastructure IP Suite (PG085) is described at https://docs.xilinx.com/r/en-US/pg085-axi4stream-infrastructure.

[130] The ARTIQ system is described at http://m-labs.hk/experiment-control/artiq/.

[131] Paweł Kulik et al. 'Latest developments in the Sinara open hardware ecosystem'. In: *2022 IEEE International Conference on Quantum Computing and Engineering (QCE)*. 2022, pp. 799–802. DOI: 10.1109/QCE53715.2022.00123.

[132] Sebastian Halama et al. 'Real-time capable CCD-based individual trapped-ion qubit measurement'. In: *arXiv preprint arXiv:2204.09112* 2022 (2022).

[133] The SOLAR EXPRESS 120 (SE120) ZYNQ ULTRASCALE+ MPSoC PCIe Board is described at https://www.sundancedsp.com/products/fpga-boards-modules/pcie/se120-xilinx-zynq-ultrascale-mpsoc-pcie-card-with-fmc-site/.

[134] Filip Wojcicki et al. 'Accelerating Transformer Neural Networks on FPGAs for High Energy Physics Experiments'. In: *2022 International Conference on Field-Programmable Technology (ICFPT)*. Hong Kong: IEEE, 2022, pp. 1–8. DOI: 10.1109/ICFPT56656.2022.9974463.

[135] The Vivado Design Suite User Guide High-Level Synthesis (UG902) is described at https://docs.xilinx.com/v/u/en-US/ug902-vivado-high-level-synthesis.

[136] Zhiqiang Que et al. 'LL-GNN: Low Latency Graph Neural Networks on FPGAs for High Energy Physics'. In: *ACM Transactions on Embedded Computing Systems* (2024).

[137] Steve Olmschenk et al. 'Manipulation and detection of a trapped Yb+ hyperfine qubit'. In: *Physical Review A* 76.5 (Nov. 2007). DOI: 10.1103/physreva.76.052314. URL: https://doi.org/10.1103/physreva.76.052314.

[138] The Teledyne X64 Xcelera-CL LX1 Base Card is described at https://www.teledynedalsa.com/en/products/imaging/frame-grabbers/xcelera-cl-lx1-base/.

[139] Advantech PCIE-1751 GPIO card is described at https://www.advantech.com/en-au/products/1-2mlkco/pcie-1751/mod_8d9d9d1d-c24f-47ca-9d65-4b934092714d.

[140] Suyog Gupta et al. 'Deep learning with limited numerical precision'. In: *International conference on machine learning*. PMLR. 2015, pp. 1737–1746.

[141] Kaiming He et al. 'Deep residual learning for image recognition'. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[142] Alireza Seif et al. 'Machine learning assisted readout of trapped-ion qubits'. In: *Journal of Physics B: Atomic, Molecular and Optical Physics* 51.17 (2018), p. 174006.

[143] The Integrated Logic Analyzer v2.0 Data Sheet (DS875) is described at https://docs.xilinx.com/v/u/en-US/ds875-ila.

[144] Yuan Yuan, Jianwu Fang and Qi Wang. 'Online anomaly detection in crowd scenes via structure analysis'. In: *IEEE transactions on cybernetics* 45.3 (2014), pp. 548–561. ISSN: 2168-2267.

[145] Mohiuddin Ahmed, Abdun Naser Mahmood and Md Rafiqul Islam. 'A survey of anomaly detection techniques in financial domain'. In: *Future Generation Computer Systems* 55 (2016), pp. 278–288. ISSN: 0167-739X.

[146] Pedro Garcia-Teodoro et al. 'Anomaly-based network intrusion detection: Techniques, systems and challenges'. In: *computers & security* 28.1-2 (2009), pp. 18–28. ISSN: 0167-4048.

[147] Luis Basora, Xavier Olive and Thomas Dubot. 'Recent advances in anomaly detection methods applied to aviation'. In: *Aerospace* 6.11 (2019), p. 117.

[148] Osman Salem et al. 'Anomaly detection in medical wireless sensor networks using SVM and linear regression models'. In: *International Journal of E-Health and Medical Communications (IJEHMC)* 5.1 (2014), pp. 20–45.

[149] Indre Zliobaite, Mykola Pechenizkiy and Joao Gama. 'An Overview of Concept Drift Applications'. English. In: *Big Data Analysis*. Studies in Big Data. Switzerland: Springer International Publishing AG, 2016, pp. 91–114. ISBN: 978-3-319-26987-0. DOI: 10.1007/978-3-319-26989-4.

[150] Abhishek Das et al. ‘An FPGA-based network intrusion detection architecture’. In: *IEEE Transactions on Information Forensics and Security* 3.1 (2008), pp. 118–132. ISSN: 1556-6013.

[151] Ami Hayashi and Hiroki Matsutani. ‘An FPGA-based In-NIC Cache approach for lazy learning outlier filtering’. In: *25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*. St. Petersburg, Russian Federation: IEEE, 2017, pp. 15–22. ISBN: 1509060588.

[152] Alex R Bucknall, Shanker Shreejith and Suhaib A Fahmy. ‘Build automation and runtime abstraction for partial reconfiguration on Xilinx Zynq Ultrascale+’. In: *2020 International Conference on Field-Programmable Technology (ICFPT)*. Virtual conference: IEEE, 2020, pp. 215–220. ISBN: 1665423021.

[153] G. Korcyl and P. Korcyl. *Optimized implementation of the conjugate gradient algorithm for FPGA-based platforms using the Dirac-Wilson operator as an example*. 2020. URL: https://arxiv.org/abs/2001.05218.

[154] Graham Cormode and Shan Muthukrishnan. ‘An improved data stream summary: the count-min sketch and its applications’. In: *Journal of Algorithms* 55.1 (2005), pp. 58–75. ISSN: 0196-6774.

[155] Yeyong Pang et al. ‘A low latency kernel recursive least squares processor using FPGA technology’. In: *2013 International Conference on Field-Programmable Technology (FPT)*. Kyoto, Japan: IEEE, 2013, pp. 144–151. ISBN: 147992198X.

[156] Quoc Le, Tamás Sarlós and Alex Smola. ‘Fastfood-approximating kernel expansions in loglinear time’. In: *Proceedings of the international conference on machine learning*. Vol. 85. 2013.

[157] Dragoljub Pokrajac, Aleksandar Lazarevic and Longin Jan Latecki. ‘Incremental local outlier detection for data streams’. In: *2007 IEEE symposium on computational intelligence and data mining*. IEEE, 2007, pp. 504–515. ISBN: 1424407052.

[158] Brian Van Essen et al. ‘Accelerating a random forest classifier: Multi-core, GP-GPU, or FPGA?’ In: *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*. USA: IEEE, 2012, pp. 232–239. ISBN: 1467316059.

[159] Akira Jinguji, Shimpei Sato and Hiroki Nakahara. 'An FPGA realization of a random forest with k-means clustering using a high-level synthesis design'. In: *IEICE TRANSACTIONS on Information and Systems* 101.2 (2018), pp. 354–362. ISSN: 1745-1361.

[160] Xiang Lin, RD Shawn Blanton and Donald E Thomas. 'Random forest architectures on FPGA for multiple applications'. In: *Proceedings of the on Great Lakes Symposium on VLSI 2017*. New York, NY, USA: Association for Computing Machinery, 2017, pp. 415–418.

[161] Xilinx Inc. *AXI4-Stream Infrastructure IP Suite v3.0 Product Guide (PG085)*. 2020. URL: https://docs.xilinx.com/v/u/en-US/pg085-axi4stream-infrastructure.

[162] Fabian Pedregosa et al. 'Scikit-learn: Machine learning in Python'. In: *Journal of machine learning research* 12.Oct (2011), pp. 2825–2830.

[163] Tianqi Chen and Carlos Guestrin. 'XGBoost: A Scalable Tree Boosting System'. In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '16. San Francisco, California, USA: ACM, 2016, pp. 785–794. ISBN: 978-1-4503-4232-2. DOI: 10.1145/2939672.2939785. URL: http://doi.acm.org/10.1145/2939672.2939785.

[164] Guolin Ke et al. 'Lightgbm: A highly efficient gradient boosting decision tree'. In: *Advances in neural information processing systems* 30 (2017), pp. 3146–3154.

[165] Xilinx Inc. *AXI4-Stream Interconnect v1.1 Product Guide (PG035)*. 2020. URL: https://docs.xilinx.com/v/u/en-US/pg035_axis_interconnect.

[166] Mahbod Tavallaee et al. 'A detailed analysis of the KDD CUP 99 data set'. In: *2009 IEEE symposium on computational intelligence for security and defense applications*. Ottawa: Ieee, 2009, pp. 1–6. ISBN: 1424437636.

[167] Shebuti Rayana. *ODDS Library*. 2016. URL: http://odds.cs.stonybrook.edu.

[168] Dheeru Dua and Casey Graff. *UCI Machine Learning Repository*. 2017. URL: http://archive.ics.uci.edu/ml.

[169] G.O.Campos and A.Zimek et al. *DAMI Datasets*. 2016. URL: https://www.dbs.ifi.lmu.de/research/outlier-evaluation/DAMI.

[170] David Faraggi and Benjamin Reiser. 'Estimation of the area under the ROC curve'. In: *Statistics in medicine* 21.20 (2002), pp. 3093–3106. ISSN: 0277-6715.

[171] Charles R. Harris et al. 'Array programming with NumPy'. In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[172] Intel Inc. *Intel Advisor*. 2022. URL: https://www.intel.com/content/www/us/en/develop/documentation/get-started-with-advisor/top.html.

[173] Servesh Muralidharan, Kenneth O'Brien and Christian Lalanne. 'A semi-automated tool flow for roofline anaylsis of opencl kernels on accelerators'. In: *First International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC'15)*. 2015.

[174] Xilinx Inc. *ZCU111 Evaluation Board User Guide*. 2018. URL: https://docs.xilinx.com/v/u/en-US/ug1271-zcu111-eval-bd.

[175] Xilinx Inc. *Xilinx. Vivado Design Suite User Guide: Power Analysis and Optimization (UG907)*. 2020. URL: https://docs.xilinx.com/v/u/2020.1-English/ug907-vivado-power-analysis-optimization.

[176] Ecoflow Inc. *River Max Portable Power Station*. 2022. URL: https://au.ecoflow.com/products/river-max-portable-power-station?variant=40043438211270.

[177] Ziming Liu et al. 'Kan: Kolmogorov-arnold networks'. In: *arXiv preprint arXiv:2404.19756* (2024).