The University of Sydney

Honours Thesis

Effective Hardware Block Design Modifications: Impact on Neural Network Performance

Author:
Prosper Su

Supervisor:
Dr. David Boland

A thesis submitted in fulfilment of the requirements for the degree of Bachelor of Engineering in the

SCHOOL OF ELECTRICAL

AND COMPUTER ENGINEERING

October 2024

Declaration of Authorship

I, Prosper Su, declared that this thesis and the work presented in it are my own. I confirm that:

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

Signed: _____

Date: _____

Disclaimer:

The views expressed in this thesis are those of the student and do not necessarily express the view of his supervisor of The University of Sydney.

Abstract

As the machine learning models develop rapidly, the demand for efficient hardware accelerators increases significantly. Field Programmable Gate Arrays (FPGAs) have been highlighted as their high reconfigurability, performance and power efficiency compared to traditional CPU and GPU platforms which have disadvantages such as low-latency and low-throughputs in embedded systems and edge computing environments.

This thesis explores the impact of hardware-level modifications in FPGA block designs on neural network outputs, focusing on efficient and cost-effective solutions to improve the overall performance. The research examines in the transition from Binarized Neural Networks (BNNs) to Ternary Neural Networks (TNNs) by applying weight randomization techniques to matrix-vector activation unit (MVAU).

The exploration can be broken down into two parts, the first method is to utilize software simulations to evaluate the impacts of weight adjustments on accuracy in Brevitas. The second method is to estimate changes in the resource allocations by applying direct hardware-level modification in the High-level Synthesis environment using FINN frameworks. The combined results from these two methods effectively bridge the gap between software simulations and real hardware implementation, suggesting that feasible approaches can enhance the performance of FPGA-based neural networks.

This research contributes valuable insights into neural network optimizations on FPGAs, leading to further exploration in more complex but efficient hardware designs.

Bachelor of Engineering

Thesis

by Prosper Su

Acknowledgements

Thank you to my supervisor Dr. David Boland for all the important advice and hours of help.

Contents

List of Figures

List of Tables

List of Acronyms

FPGA	Programmable Gate Arrays
CPU	Central Processing Unit
GPU	Graphic Processing Unit
BNN	Binary Neural Network
TNN	Ternary Neural Network
HLS	High-Level Synthesis
ASIC	Application Specific Integrated Circuit
CLB	Configurable Logic Block
QNN	Quantized Neural Network
DNN	Deep Neural Network
DMA	Direct Memory Access
FIFO	First In, First Out
LUT	LookUp Table
FF	Flip Flop
MVAU	Matrix Vector Activation Unit
DSP	Digital Signal Processing
BRAM	Block Random Access Memory
MLP	Multilayer Perceptron

Chapter 1

Introduction

1.1 Motivation

Recently, the shift toward hardware-accelerated neural network implementations has generated a pressing need for applications requiring low latency and high throughput. Field Programmable Gate Arrays (FPGAs) becomes one of the most popular platform selections by providing a customizable and energy-efficient alternative to traditional CPUs and GPUs. It also offers a benefit for developers to design custom architectures that best fit to the application requirements, allowing for optimised neural network deployments that overcomes the constraints of real-time applications.

Despite these advantages, deploying neural networks on FPGAs faces challenges in balancing accuracy with resource allocations. Binary Neural Networks (BNNs) which is one of forks of Quantized Neural Networks have been developed to address these challenges due to their low-precision representation, leading to memory usage and computational requirements reductions. However, BNNs' binary nature causes to accuracy degradation during the testing and training process. This trade-off between efficiency and accuracy has gained interest in exploring alternative methods that can generate impact on model's accuracy and resources allocations within hardware domain.

This thesis explores direct modifications at the hardware level within FPGA block designs to find their impacts on neural network outputs. Modifying the weight allocation within certain block designs supports to effectively control over resource allocation and computation accuracy. This research investigates whether intentional alterations within FPGA-based neural networks can produce improved hardware-based neural network architectures in a more efficient and accurate manner.

1.2 Research Question

How can hardware-level modifications be utilized as a quick and cost-effective approach to improve the performance of FPGA-based neural networks?

1.3 Aims & Objectives

The aim of this research is to explore the impact of hardware-level modifications within FPGA block design by altering the weight allocation and quantization processes on the outputs of sample neural network algorithms. This study determines whether such modifications can enhance neural network performance via balancing accuracy and with slightly increase in FPGA resources usage. The set of objectives for the aim are shown in the following:

1. Understand the quantized neural networks by testing their accuracies, resource utilizations, and limitations within software simulations.

2. Design and implement direct modification techniques to FPGA-based neural networks and record its performance metrics such as accuracy and resource allocation in hardware-software platforms like Brevitas and FINN.

3. Evaluate the effectiveness of FPGA tuning as a cost-effect and fast approach by conducting the performance comparisons between FPGA-based neural networks with and without modification applied.

1.4 Contributions

This thesis makes the following critical contributions to FPGA-based neural network optimization:

1. Establish a basic understanding of BNNs by experimenting on implementations with benchmarks of accuracy and resource usage to set the stage for further enhancements.

2. Apply ternary quantization through weight randomization which demonstrates its potential to alter accuracy.

3. Introduce synthesis block modifications in the HLS environment to simulate hardware adjustments which showcases the various resources usage on FPGA.

4. Analyze the impact of hardware modifications above on both accuracy and resources. This provides insights into the effects of these software hardware level changes on real hardware implementation outputs.

1.5 Thesis Outline

Chapter 1 shows the motivation, the research question, aims, objectives, and key contributions behind the research. It outlines the importance of exploring hardware-level modifications in FPGA block design can have potential to impact the neural network outputs.

Chapter 2 demonstrates a review of relevant literature on neural network implementations across various hardware platforms, including CPUs, GPUs, and FPGAs, justifying the choice of FPGAs. Then, the impact of hardware modifications versus software optimizations in neural network implementations is detailed. Additionally, it introduces Binarized Neural Networks (BNNs) and Ternary Neural Networks (TNNs), followed by an overview of the FINN and Brevitas frameworks. Finally, it explores concepts from a review of literature on weight randomization.

Chapter 3 is a summary of the requirements formulation process, providing the functional and non-functional requirements, constraints needed to explore hardware modifications on neural networks in FPGA block designs.

Chapter 4 outlines the methodology for implementing and testing the neural network outputs, including software-based simulations, transitioning from BNNs to TNNs through weight randomization within software environment and the similar modifications applied within the High-Level Synthesis (HLS) environment.

Chapter 5 presents the results obtained from simulations and FPGA-based testing. The analysis of key performance metrics such as accuracy and resource

utilization is also conducted.

Chapter 6 summarizes the main findings, including results from experiments and implications of hardware-level modifications in FPGA neural network implementations based on experimental results. Finally, a summary of future works is presented.

Chapter 2

Literature Review

This chapter details neural network implementations on FPGA, as well as further developments in algorithms and hardware system designs.

2.1 Neural network implementations on various hardware platforms

The implementation of neural networks on various hardware platforms, including CPUs, GPUs, ASICs and FPGAs, has become important as neural network models starts to grow in complexity and demands for multiple applications. Each hardware platform offers unique advantages and trade-offs in terms of performance, energy efficiency, flexibility, and scalability. Different options can create a significant impact on the effectiveness of neural network deployment.

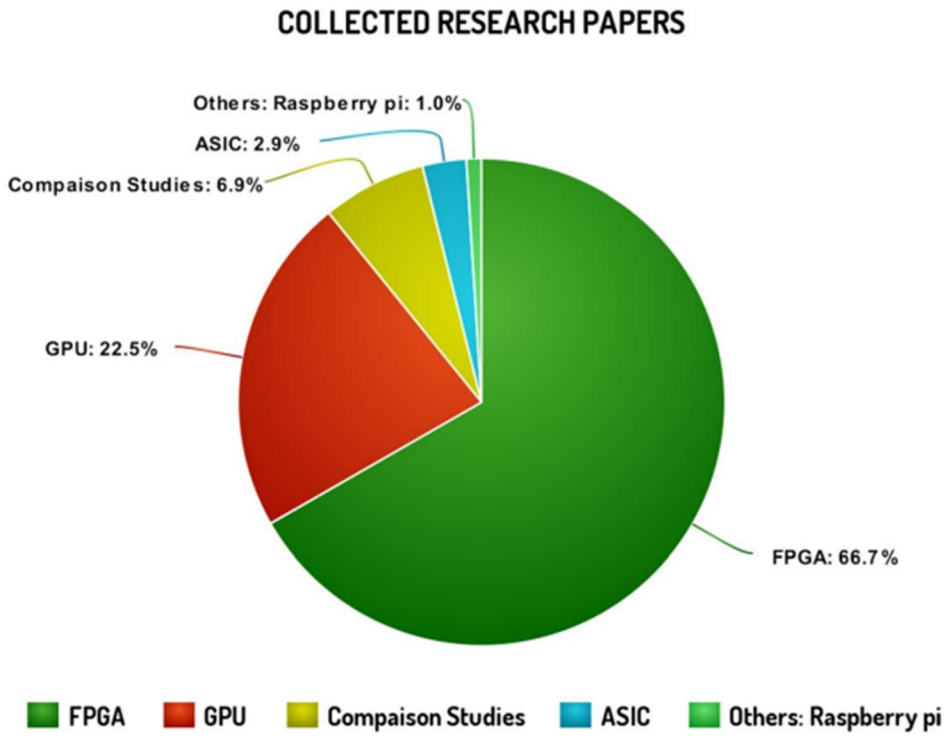


Figure 1. Hardware implementation platforms distribution chart [1]

CPUs as common general-purpose processors are widely used due to their good flexibility and programmability. Hardware platforms distribution chart (figure 1) demonstrates only the small percentage of developer utilise the CPUs for

algorithm implementation. Although CPUs can provide sequential processing capabilities to handle diverse operations within neural networks, the limitations in parallelism compared to other platforms reduce their feasibility in high-performance scenarios. This explains that CPUs are often limited by computational bottlenecks and higher energy consumption [1].

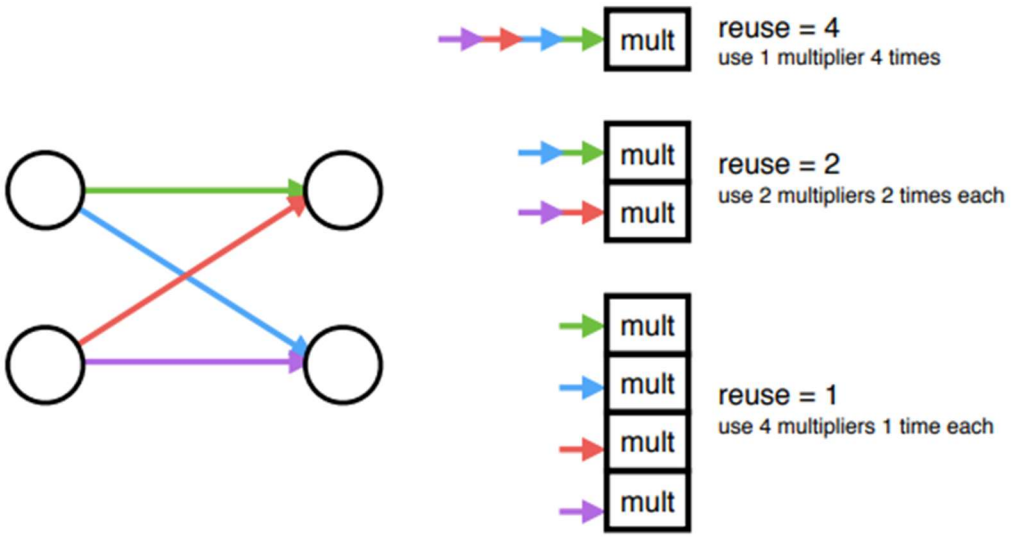


Figure 2. Illustration of parallelism for neural nodes operations. [2]

GPUs are designed for high-performance parallel processing operations which make them become one of recent popular platforms for training and testing neural network models. GPUs offers advantages of extensive floating-point computations by using high memory bandwidth that enables rapid data transfers between memory and compute cores. Figure 2 illustrates how GPUs can outperform CPUs in terms of parallel processing capabilities. With design of high-performance parallelism in GPUs, they can perform four nodes at same time, whereas CPUs can perform 2 nodes parallelly. However, this leads to GPUs consume much more power which hits with power constraints easily, making them not as flexible as other platforms. This limitation restricts their application in those energy-sensitive and real-time processing models, where low power and configurability are demands [2].

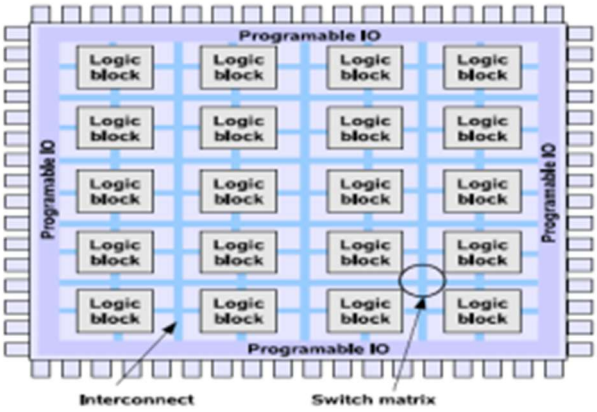


Figure 3. FPGAs structure [3]

FPGAs offers the parallelism of GPUs with the flexibility of CPUs to custom hardware required. FPGAs can be customized for specific neural network architectures by its reconfigurable logic property which supports to optimise data and memory paths to achieve high throughput and low latency at lower power consumption levels. These characteristics make FPGAs ideal to execute real-time applications such as autonomous vehicles and robotics. According to the diagram of FPGA architecture (figure 3), it presents those configurable logic blocks and interconnections that allow FPGAs to provide enhanced parallelism to specific operation more efficiently. However, FPGA reconfiguration can be time-consuming which require extra development cycles to employ the implementations [4].

FPGAs also support for integer and low-precision arithmetic where GPUs don't support these. This improves FPGAs efficiency by reducing resource usage and computational difficulty in low-bit quantized models such as Binarized Neural Networks (BNNs) and Ternary Neural Networks (TNNs). FPGAs can maintain accuracy while minimizing power consumption which is less feasible on CPUs and GPUs.

Those FPGAs advantages mentioned above makes them an ideal platform for different neural network applications where high configurability, low power, and high performance are required for growing neural network development [1]. As a result, this thesis focuses on FPGA to employ the potential modifications at the hardware level to optimize neural network performance for the real-time data sets.

2.2 Hardware & Software modifications comparison

Improving the performance and efficiency of neural networks on hardware platforms can be affected through both hardware modifications and software-based optimizations. Hardware modifications on FPGAs mean to changes the structural block design, potentially increasing energy efficiency, latency, and throughput. In contrast, software modifications, pruning, quantization, and model compression can utilise resource more efficiently without changing the underlying hardware. This section compares the trade-offs and potential benefits of each approach and determines the desired approach used in the thesis.

Software Optimizations are techniques used at the model level to improve neural network algorithms by reducing memory or computational costs. Common methods include quantization, lowering the bit width of weights and activations, and pruning, redundant parameters removal in the network.

These optimizations are effective by achieving faster inference times with minimal loss in accuracy in multiple platforms [5]. Quantization, conversion of weights and activations from floating-point to lower bit-widths can result in reduced memory footprint and increased computation. Pruning techniques allow models to execute fewer operations with less memory access by eliminate less significant neural network parameters. However, these software optimizations are usually limited by the fixed hardware configurations of CPUs and GPUs, which lack the flexibility to fully utilise the reconfigurable logic available in FPGAs [6].

Hardware Modifications on FPGAs allow developers to tailor custom configurations to neural network architectures by modifying reconfigurable logic blocks. This flexibility brings many benefits such as custom memory hierarchies and optimized data paths which significantly improve performance for tasks. By adjusting configurations at the logic level, FPGAs allows data to flow continuously through various processing units by implementing hardware-level pipelining. This increases efficiency in operations such as convolutional layers that involve repetitive computations. Custom hardware modifications in interconnects and memory blocks optimisation improve data processing efficiency and minimizing the need for frequent memory access which is a typical bottleneck in CPU- and GPU-based implementations [7].

FPGA hardware modifications provide a greater potential for increasing energy efficiency while maintaining high throughput. For example, custom memory configurations tailor to the data flow needs of neural networks can improve energy efficiency compared to traditional architectures, aligning with neural network requirements with power saving by avoiding redundant data movements. Similar custom configurations can utilise the unique properties of FPGAs, whereas software optimizations alone cannot achieve in fixed architectures like CPUs and GPUs [8].

In summary, although software-based optimizations are valuable for enhancing the efficiency of neural networks across various platforms, hardware modifications on FPGAs can effectively improve energy usage, latency, and parallel processing capabilities. The ability to make direct changes at the hardware level can maximise the flexibility of FPGAs which makes them an attractive and sensible choice for high-performance and low-power consumption neural network applications. This indicates that hardware modification can be tuned to balance energy efficiency with processing speed with better capabilities than conventional software optimization techniques.

2.3 Binarized neural networks and Ternary neural networks

Quantized neural networks, especially Binarized Neural Networks and Ternary Neural Networks, are optimized models for deployment on hardware platforms like FPGAs. These networks represent weights and activations using two-level (binary) or three-level (ternary) quantization, providing resource efficiency and computational advantages. This section introduces BNNs and TNNs structures, hardware advantages, and suitability for FPGA implementation.

Binarized Neural Networks (BNNs)

BNNs are neural networks with weights and activations to binary values represented as -1 and +1 [9]. This simplification enables computations to only use bitwise operations such as XNOR and popcount to reduce both memory usage and computational demands than traditional multiply-accumulate operations. This characteristic makes BNNs are well-suited for deployment on FPGAs that require low-power consumption with constrained resources [10].

The primary advantage of BNNs for hardware is their reduced computational complexity and memory usage. By applying binary operations, BNNs can perform efficient parallel processing on FPGA architectures by replacing traditional arithmetic operations with logical operations, enhancing processing speed and reducing energy consumption. Furthermore, BNNs are designed to have an acceptable accuracy loss compared to other full-precision networks which has potential to improve through optimized training strategies [9][11].

Ternary Neural Networks (TNNs)

TNNs adds quantization to three levels from two levels represented as -1, 0, and +1. This additional level manipulated zero values which can skip certain computations and thus reduce the overall computational load. TNNs aim to provide a compromise between binary and full-precision neural networks by achieving higher accuracy than BNNs with only a slight increase in resource usage [12]. For example, implementing zero-skip operations can help TNNs to balance accuracy with computational efficiency. This benefits to real-time processing applications that require accurate precision, but hardware resources are limited [13].

Additionally, TNNs offer greater flexibility than BNNs with their three-level quantization. It offers more expressive modelling without significantly increasing memory and processing demands. One of the FPGA implementations on TNNs has shown improved accuracy while maintaining efficiency through the methods that utilise optimized ternary inner products and configurable logic blocks (CLBs) within FPGA [12].

Motivation for Exploring TNNs

The development of TNNs shows a logical choice compared to BNNs which targets to enhance accuracy while maintaining efficiency. With the additional quantization level (0) inside TNNs, it introduces sparsity to further reduces computational complexity and can increase processing speed on FPGAs [10]. Furthermore, one of studies shows that TNN model can achieve nearly equivalent accuracy to higher-bit networks while reducing the resources usage for those applications that demands a balance of accuracy and resource efficiency [13].

2.4 Introduction to FINN and Brevitas Frameworks

The integration of FINN and Brevitas facilitates a streamlined approach to quantization and neural network model deployment on FPGAs, providing necessary functionalities for this research.

FINN Framework

FINN is an open-source framework designed to support quantified neural networks (QNNs) on FPGAs by Xilinx. FINN emphasizes customizable dataflow architectures and parameterized hardware for layer-wise optimizations while implementing QNNs on FPGA platforms [14][15] compared to traditional frameworks with fixed hardware configurations.

This framework offers full steps from network optimization to deployment with a workflow that includes graph transformations, high-level synthesis (HLS) optimizations, and finally hardware compilation. In this research, FINN's capability to transforms complex neural networks into FPGA-optimized designs enables developer to effectively evaluate impact of different quantization levels on hardware resource usage and neural network performance on Xilinx devices [16].

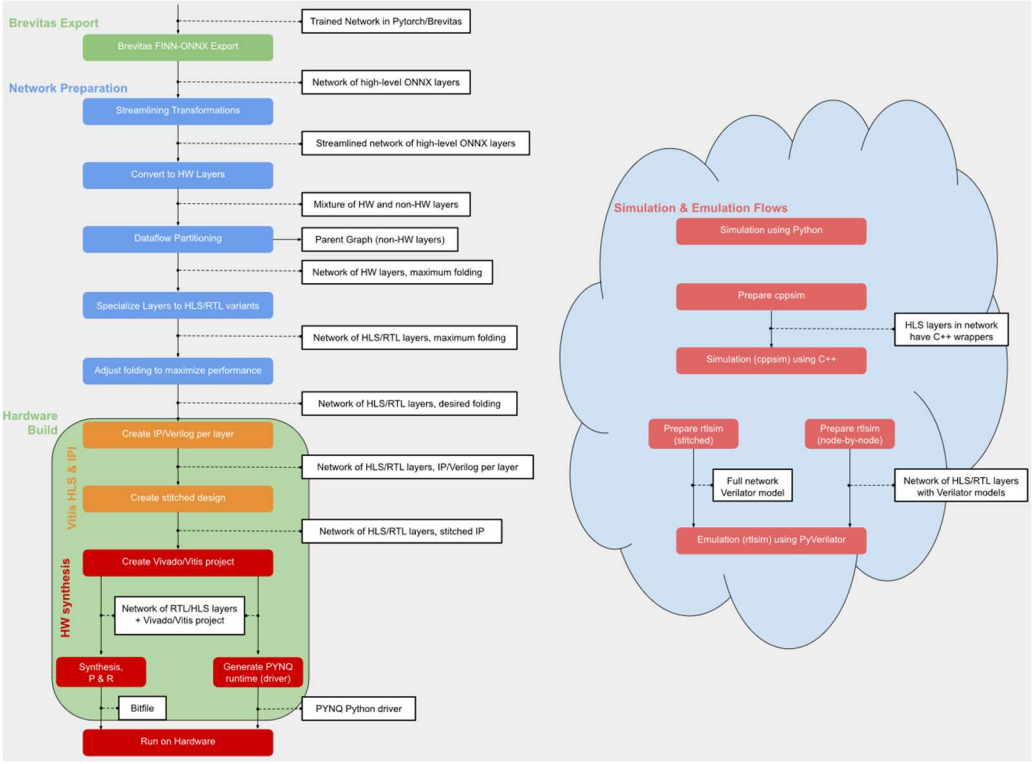


Figure 4. FINN and Brevitas frameworks end-to-end flow chart [23][24]

Furthermore, FINN can specialize modules for folding and parallelism configurations that are important to optimize hardware resources while

maintaining high throughput. This design flow (figure 4) can perform optimized resource allocation by further breaking down neural network partitions into hardware-compatible blocks connected through FIFO streams and Direct Memory Access (DMA) modules. By applying different bit-width settings, FINN can balance accuracy and performance. This approach of bit-width flexibility supports the exploration of BNN and TNN conversions in the project [14][15].

Brevitas Framework

Brevitas is a quantization-aware training library based on PyTorch and is a core application to train neural networks with precision parameters. It provides flexible bit-width configurations for weights and activations, facilitating efficient quantization schemes during the training phase to prevent the limitation in standard neural network libraries during post-training [17]. This feature is useful in this research as Brevitas permits the experimentation with varying quantization strategies in terms of randomized weights for TNNs. After the configurations are confirmed, Brevitas exports it to ONNX format which is compatible with FINN for further development [15][16].

```
import torch
import torch.nn as nn
import brevitas.nn as bnn
# Classic sequence in PyTorch
classic_sequence = nn.Sequential(
  nn.Linear(120, 84),
  nn.Tanh()
)
# Quantized sequence in Brevitas
quantized_sequence = nn.Sequential(
  bnn.QuantLinear(120, 10, bit_width = 3),
  bnn.QuantTanh(bit_width = 5)
)
```

Figure 5. Layer structure in Brevitas and PyTorch [14]

Figure 5 demonstrates the comparison between classic sequence of layers in PyTorch and quantized sequence in Brevitas. It firstly means that Brevitas has a similar bit-width-specific configurations on layers to PyTorch for neural network and implies that quantized operations such as QuantLinear can be implemented with lower bit-widths to suit FPGA constraints.

Through compatible integration with FINN, Brevitas-trained networks can be directly exported to FINN compiler which supports a design and deployment cycle. This leads to lowering memory consumption and increase processing rates without significant accuracy loss. FINN and Brevitas are used to optimize the neural networks for FPGA implementation for model training to deployment while offering the flexibility to experiment with various hardware

configurations.

In summary, FINN and Brevitas validate an approach to vary flexibility, speed, and efficient resource utilization for neural network applications on constrained hardware platforms.

2.5 Idea of Weights and activations Quantization methodology

The efficient deployment of QNNs on FPGAs often requires suitable compression strategies to satisfy the resource constraints and optimize the performance capabilities. Layer-targeted quantization of weights and activations is a highly effective approach for achieving this goal by using the distinct characteristics of each layer to balance accuracy and resource usage. This section explains the key elements of layer-specific quantization via emphasizing how weight randomization can contribute to change FPGA performance.

Layer-targeted quantization focuses on selectively implementing different bit-widths for various layers according to their sensitivity and resource demand. By reducing the bit-width of weights and activations in less critical layers, FPGAs can still achieve high computational throughput without affecting accuracy significantly. In recent research, layer-targeted methods have shown that varying quantization per layer generates reductions in memory and DSP requirements with FPGA streaming architectures [17]. This targeted approach not only optimizes FPGA resource allocations but also accomplishes with the low-power consumption and high-speed goals for DNN acceleration on FPGAs.

Randomizing weights within a quantized layer, for example, transition from binary to ternary representations, has shown potential developments DNN models further. This method adds weights across defined values (e.g., -1, 0, +1) and optimizes memory use by reducing the need for frequent visit it. Studies have illustrated that weight randomization can produce FPGA designs with fewer DSP and RAMs blocks, resulting in significant power savings [18]. Moreover, when combined with quantized activations technique, resulting in models like VGG16 to operate with reduced computational loads without significant accuracy degradation highlights its effectiveness on large-scale

networks [19].

The application of layer-specific quantization on FPGAs has potential to offer improvements in both performance and efficiency. For example, VGG-SSD, one of large-scale DNNs. Its on-chip memory requirements are considerable, but this layer-targeted quantization method reduces memory overhead and DSP usage before hitting limits. As demonstrated in recent findings, this approach can reduce model size up to 32 times for weights and at least five times for activations with only a minor accuracy loss (e.g., around 1-2%) [20]. These advantages ensure that layer-targeted quantization is a feasible strategy for deploying DNNs on resource-constrained hardware.

In summary, this layer-targeted quantization and weight randomization method is a flexible and efficient method for tailoring neural networks to FPGA limitations by achieving high-speed processing and energy efficiency.

2.6 Summary

Literatures about Key frameworks and methodologies for deploying neural networks on FPGAs are reviewed. Firstly, an overview of a comparison of hardware platforms justified the selection of FPGAs for their configurability and efficiency. Then, the advantages of hardware modifications over software-based optimizations were discussed, followed by an introduction to BNNs and TNNs as efficient quantized models to provide an understanding of targeted neural networks. Next, the FINN and Brevitas frameworks were reviewed for their role in the research. Finally, layer-targeted quantization and weight randomization methods are analyzed.

Chapter 3

Requirements Formulation

3.1 Requirements Overview

The technical requirements for this project are from the main objective of examining the effects of direct modifications to hardware block designs on neural network accuracy and its resource utilization on hardware platform. The following requirements for neural network models and FPGA implementations must be satisfied to achieve the objectives:

- Models shall be quantized to generate the impact of quantization on hardware efficiency.
- The local environment shall support the FINN and Brevitas frameworks to deploy neural networks on FPGA platform.
- Weight randomization techniques shall be compatible with various networks configurations.
- The frameworks shall provide sample accuracy and resource rating for further comparisons after modifications are applied.

The above requirements ensure that the implementation enables developers to explore hardware modifications while maintaining the stability and compatibility of neural network deployment with the working platform. To be more specific, the requirements are broken down into functional, non-functional and constraints. The functional requirements are desired or expected behaviors of the system, whereas the non-functional requirements specify a set of criteria that can be used to estimate the performance of the system. As for the constraints, a set of limitations is shown to limit the ability of a system to achieve goals.

3.2 Functional Requirements

1. Neural network model configuration

    - Testing neural networks shall support various quantization techniques such 2 bits or 1 bit to conduct comparisons in accuracy

and resources usage between different quantization level.
- Neural networks models shall be compatible with Brevitas and FINN frameworks to ensure to accuracy and resource allocation testings can be processed.

2. Randomization and testing progress

- Brevitas shall support weight randomization for all layers such as MVAU to observe its effect on model's accuracy.
- FINN frameworks shall allow the modifications to the MVAU to observe any changes in resource utilization such as Look-Up tables (LUTs) or flip-flop (FF) numbers on FPGA.
- Brevitas and FINN frameworks shall provide the initial results of accuracy and resource as a baseline for further comparisons.

3.3 Non-functional Requirements

Comparison requirements

- The test process for MVAU shall prioritize resources usage in FPGA by maintaining low latency and high throughput for real-time applications.
- The modifications on weight or activations should balance accuracy and resource utilization without reaching FPGA's constraints.
- The exploration on direct modifications on hardware blocks shall support more adjustments to model quantization, layer configuration and testing variables to ensure its adaptability to more experiments.

3.4 Constraints

The testing and modification process can be affected by several limitations:

- Resource constraints include limited numbers of LUTs, DSPs and BRAM may affect the feasibility of neural networks implementation on the certain FPGA model.
- Compatibility constraints of Brevitas and FINN frameworks may restrict the model modification and configurations or even the quantization

methods.

- Time constraints such as time-consuming synthesis cycles and testing may affect the time taken to fulfil the scope of modification and goals for this research.

3.5 Summary

This chapter outlined the process-oriented requirements for testing and modifying FPGA-based neural networks to achieve the goals. These requirements validate that from baseline testing to iterative modification stage, the process can meet the accuracy and resource allocation goals without compromising constraints and project timelines.

Chapter 4

Implementation

4.1 Introduction

This chapter outlines the methods of implementing and testing quantized neural networks on FPGA with specific hardware modifications to explore accuracy and resource utilization changes. The focus is on observing the effects of changing weight configurations within the Matrix-Vector Activation Unit (MVAU) where input weights transform from binary (-1,1) to ternary (-1, 0, 1) representations.

The testing and modification workflow is structured into the following steps, each addressing a different domain of the neural network implementation and examining its role for real-world FPGA deployment:

- Acquire baseline accuracy and resource metrics using a default Brevitas and FINN framework model

- After establishing the sample results, weight randomization is applied within Brevitas to evaluate its effectiveness on accuracy by recording results.

- Finally, modifications are applied directly to the MVAU within FINN where similar randomized weight mechanisms are introduced.

This structured approach ensures a comprehensive workflow that highlights the impact of hardware-level modifications on FPGA-based neural networks. The process allows developers to gain key results about accuracy and resource efficiency to support the research objectives. The complete modification codes are provided in Appendix sections.

4.2 Initial Testing with Default Neural Networks setting

The first step was to establish sample accuracy and FPGA resource utilization using a default quantized neural network model. This sample can be used as a

reference point for addressing the effects of hardware-level modifications on it. The first test was conducted in Brevitas to determine model accuracy, followed by resource allocation analysis in FINN.

4.2.1 Accuracy Testing in Brevitas

The default model, a quantized Multi-Layer Perceptron (MLP), was configured with binary weights and activations (BNN configuration). It was trained on the UNSW-NB15 dataset for the accuracy of network intrusion detection.

Define the Quantized MLP Model

We'll now define an MLP model that will be trained to perform inference with quantized weights and activations. For this, we'll use the quantization-aware training (QAT) capabilities offered by Brevitas.

Our MLP will have four fully-connected (FC) layers in total: three hidden layers with 64 neurons, and a final output layer with a single output, all using 2-bit weights. We'll use 2-bit quantized ReLU activation functions, and apply batch normalization between each FC layer and its activation.

In case you'd like to experiment with different quantization settings or topology parameters, we'll define all these topology settings as variables.

[7]:
```
input_size = 593
hidden1 = 64
hidden2 = 64
hidden3 = 64
weight_bit_width = 2
act_bit_width = 2
num_classes = 1
```

Now we can define our MLP using the layer primitives provided by Brevitas:

[8]:
```
from brevitas.nn import QuantLinear, QuantReLU
import torch.nn as nn

# Setting seeds for reproducibility
torch.manual_seed(0)

model = nn.Sequential(
      QuantLinear(input_size, hidden1, bias=True, weight_bit_width=weight_bit_width),
      nn.BatchNorm1d(hidden1),
      nn.Dropout(0.5),
      QuantReLU(bit_width=act_bit_width),
      QuantLinear(hidden1, hidden2, bias=True, weight_bit_width=weight_bit_width),
      nn.BatchNorm1d(hidden2),
      nn.Dropout(0.5),
      QuantReLU(bit_width=act_bit_width),
      QuantLinear(hidden2, hidden3, bias=True, weight_bit_width=weight_bit_width),
      nn.BatchNorm1d(hidden3),
      nn.Dropout(0.5),
      QuantReLU(bit_width=act_bit_width),
      QuantLinear(hidden3, num_classes, bias=True, weight_bit_width=weight_bit_width)
)

model.to(device)
```

Figure 6. Default Quantized MLP model settings

Specifications for this MLP model (figure 6):

- **Architecture**: Three fully connected hidden layers, each with 64 neurons and 2-bit quantized weight.
- **Activations**: Each hidden layer was integrated with binarized ReLU activations to maintain consistency with the BNN configuration.
- **Output**: The final layer generated a binary classification output

after passing through a sigmoid activation function which enhances model's stability.

The model was trained from scratch by utilizing Brevitas's quantization-aware training capabilities to obtain the accuracy. This default accuracy was a benchmark for comparison after the further modifications to the weight configuration.

4.2.2 Resource Allocation Analysis in FINN

After the accuracy was tested, the model was exported to FINN framework in Quantized ONNX (QONNX) format to find FPGA resource utilization. The primary goal for this section was to set a reference for a comparison of FPGA source utilization before and after modifications in the MVAU.

```
[ ]: import finn.builder.build_dataflow as build
     import finn.builder.build_dataflow_config as build_cfg
     import os
     import shutil

     model_file = model_dir + "/cybsec-mlp-ready.onnx"

     rtlsim_output_dir = "output_ipstitch_ooc_rtlsim"

     #Delete previous run results if exist
     if os.path.exists(rtlsim_output_dir):
         shutil.rmtree(rtlsim_output_dir)
         print("Previous run results deleted!")

     cfg_stitched_ip = build.DataflowBuildConfig(
         output_dir          = rtlsim_output_dir,
         mvau_wwidth_max     = 80,
         target_fps          = 1000000,
         synth_clk_period_ns = 10.0,
         fpga_part           = "xc7z020clg400-1",
         generate_outputs=[
             build_cfg.DataflowOutputType.STITCHED_IP,
             build_cfg.DataflowOutputType.RTLSIM_PERFORMANCE,
             build_cfg.DataflowOutputType.OOC_SYNTH,
         ]
     )

[ ]: %%time
     build.build_dataflow_cfg(model_file, cfg_stitched_ip)
```

Figure 7. build_dataflow function for resources allocation data

Resource allocation analysis steps are shown as follows:

- Model export: The quantized model was exported from Brevitas to FINN with 2-bit binary quantization in this case.
- Dataflow compilation: The **[build_dataflow]** (figure 7) function in FINN was used to divide the entire network into layers compatible with FPGA. Those layers were then mapped to FPGA hardware with optimization settings.
- Synthesis and reference data collection: Key FPGA resources such Look-Up Tables (LUTs), Flip-Flops (FFs), Digital Signal Processing

(DSP) blocks, and Block RAM (BRAM). were saved and would be used to evaluate changes when modifications to the MVAU are implemented.

These initial testing in Brevitas and FINN set up reference accuracy and hardware for future experiments. This foundation is essential for exploring the effects of hardware-level modifications on neural networks outputs.

4.3 Weight randomization in Brevitas

Following the initial testing, a weight randomization mechanism was introduced within Brevitas to simulate potential hardware-level interruptions or faults which allow the observations to randomizations effects on model accuracy.

4.3.1 Randomization function

The weight randomization process was implemented through custom functions which created a masking of weights in each layer. This approach created a randomized mask that changed the active weights during inference through zeroing out a specified percentage of weights across the network.

```
[16]: import torch
from sklearn.metrics import accuracy_score

def apply_weight_mask(layer, mask_ratio):
    """
    Apply a random mask to the weights of the input layer.

        layer: Different types neural network layer (e.g., QuantLinear).
        mask_ratio: The precentage of weights to zero out (between 0 and 1).

    """
    with torch.no_grad():
        # Find the weight tensor
        weight = layer.weight

        # Create a mask with the same shape as the previous weight tensor
        mask = torch.ones_like(weight)

        # Get the number of elements to zero out
        num_elements = weight.numel()
        num_zero = int(mask_ratio * num_elements)

        # Genearte random indices to zero out
        indices = torch.randperm(num_elements)[:num_zero]

        # zero out selected indices
        mask.view(-1)[indices] = 0

        # apply the mask to the weights
        weight.data.mul_(mask)
```

Figure 8. apply_weight_mask function for accuracy data

- Random Mask mechanism: A mask was created for each weight

tensor based on a specified mask ratio which set the percentage of weights to zero. This method could preserve a proportion of weight when simulating sparse connectivity.

- Weight Masking Application: The mask was applied on each QuantLinear layer during inference. By zeroing out the controlled weights, the model effectively operated with partially randomized weights and could be tested its accuracy under various conditions.
- Function Design: The randomization **[apply_weight_mask]** function could apply the randomization ratio flexibly for testing across the networks.

4.3.2 Accuracy testing with randomized weights

To explore the effects of randomization on accuracy, multiple testing iterations were executed with various mask ratios. Each test began by resetting the model's original weights to ensure that previous randomization effects did not accumulate.

```
def test_with_weight_mask(model, test_loader, mask_ratio):
    """
    Test the model with a random weight mask applied during inference.

    """
    # Ensure model is in evaluation mode
    model.eval()
    y_true = []
    y_pred = []

    # Apply the weight mask to each QuantLinear layer
    for layer in model:
        if isinstance(layer, QuantLinear):
            apply_weight_mask(layer, mask_ratio)

    with torch.no_grad():
        for data in test_loader:
            inputs, target = data
            inputs, target = inputs.to(device), target.to(device)
            output_orig = model(inputs.float())
            # Apply sigmoid activation
            output = torch.sigmoid(output_orig)
            # Threshold outputs at 0.5 to obtain binary predictions
            pred = (output.detach().cpu().numpy() > 0.5).astype(int)
            target = target.cpu().float()
            y_true.extend(target.tolist())
            y_pred.extend(pred.reshape(-1).tolist())

    # Return accuracy
    accuracy = accuracy_score(y_true, y_pred)
    return accuracy
```

```
import numpy as np
from sklearn.metrics import accuracy_score
from tqdm import tqdm, trange



mask_ratios = [0] # Zero out 0%, 10%, 20%, 50% of the weights

running_loss = []
running_test_acc = []
t = trange(num_epochs, desc="Training loss", leave=True)

for epoch in t:
        loss_epoch = train(model, train_quantized_loader, optimizer,criterion)
        test_acc = test_with_weight_mask(model, test_quantized_loader, mask_ratio)
        t.set_description("Training loss = %f test accuracy = %f" % (np.mean(loss_epoch), test_acc))
        t.refresh() # to show immediately the update
        running_loss.append(loss_epoch)
        running_test_acc.append(test_acc)
```

Figure 9. Accuracy testing function demonstration, the accuracy changes by setting mask_ratio to 0.1 or higher

- Mask ratios and testing procedure: Mask ratios were tested with different percentages (e.g., 0%, 10%, 20%, 50% as figure 9 shown) to examine accuracy at various levels of weight randomization and accuracy shifts were recorded for further comparisons.
- Accuracy evaluation: Accuracy results were recorded and could perform a direct comparison of performance with and without weight randomization. The outputs still went through a sigmoid activation to enhance the stability of the model after the randomizations were applied.

This weight randomization testing in Brevitas created a basis for examining the impacts of MVAU modifications in hardware by transiting from binary weights to ternary weights.

4.4 Weight modification in FINN

The next step was to directly modify the Matrix-Vector Activation Unit (MVAU) within the FINN framework after the weight randomization in Brevitas. The MVAU, a core component in quantized neural network to perform the matrix-vector computations between fully connected and convolutional layers, was decided to be modified. The goal of this phase was to observe the changes within the MVAU to impact FPGA resource utilization by applying weight randomization.

4.4.1 Verification testing with default model

The first step was to verify MVAU function and its impact on resource allocation by utilizing the default neural network model. To achieve this, an "extreme test" was designed to insert static weight configurations that could simplify the MVAU computational load and reduce its resource requirements.

```
#define WEIGHTS_ZERO  // Uncomment to set all weights to zero
#define WEIGHTS_ONE  // Uncomment to set all weights to one

'''
WEIGHTS_ONE Defined: If WEIGHTS_ONE is defined, weight is set to 1.
WEIGHTS_ZERO Defined: If WEIGHTS_ZERO is defined, weight is set to 0.
'''

auto const wgt =
  #ifdef WEIGHTS_ONE
    static_cast<decltype(TWeightI()(w[pe]))>(1);
  #elif defined(WEIGHTS_ZERO)
    static_cast<decltype(TWeightI()(w[pe]))>(0);
  #else
    TWeightI()(w[pe]);
  #endif

'''
When weights are zero, the MAC operation can be skipped entirely since any multiplication by zero will give you zero.

'''
#ifdef WEIGHTS_ZERO
  // Skip MAC operation since weights are zero
  accu[0][pe] = activation.init(nf, pe);
#else
  // Perform MAC operation as usual
  auto const wgt = /* as defined above */;
  accu[0][pe] = mac<SIMD>(accu[0][pe], wgt, act, r, 0);
#endif
```

Figure 10. Extreme test mechanism demonstration.

- Static weight modifications: Static configurations were applied by modifying the mvau.hpp file to set all weights within the MVAU to either zero or one. At the top of mvap.hpp file, **[WEIGHTS_ZERO]** and **[WEIGHTS_ONE]** (figure 10) were defined to switch between weight values and a set of modified operations demonstrated the simplified logic path.
- Goals: This modification could generate FPGA resource shifts such as changes in Look-Up Tables (LUTs) numbers and critical path delays by analyzing results after applying the codes.

This testing helped to verify whether MVAU was the place for modification as its operations significantly could influence overall resource allocation and data path timing in the network.

4.4.2 Verification testing with default model

After confirming that MVAU would have effect on resource allocation, modifications were implemented to simulate weight randomization at the hardware level by zeroing out weights in a pseudo-randomized manner within the MVAU.

```
# define counter to start with 0
unsigned int rand_counter = 0;

# Increment the counter
rand_counter++;

# Generate a pseudo-random bit using XOR operations
bool rand_bit = (rand_counter ^ (rand_counter >> 1)) & 0x1;

# If random bit is zero, set weight to zero
for(unsigned pe = 0; pe < PE; pe++) {
    #pragma HLS UNROLL
    if (rand_bit == 0) {
        w.m_weights[pe] = 0;
    }
}
```

Figure 11. Randomization mechanism demonstration.

- Randomization Method: To implement randomization, a counter-based pseudo-random generator was used in mvau.hpp. This generator applied an XOR operation with a shifted version of itself to produce a **[rand_bit]** (figure 11) value which could be utilized to control zeroing of weights within the processing elements (PEs).
- Selective Weight Zeroing: The **[rand_bit]** functioned as partial weight randomization simulation by setting individual weights to zero during computation. This method allowed controlled yet varied modifications to the MVAU weight values.
- Resource evaluation: The network was synthesized in FINN to maintain a consistency across builds for each randomization test. This approach could show FPGA resource responses to different levels of randomization in the MVAU for further analysis under varied conditions.

These MVAU modifications in FINN effectively evaluated the hardware-level influence on resource. This phase was essential for establishing hardware level modifications in FPGA could affect the resource allocation on the neural networks.

4.5 Summary

This chapter outlined the methods and process implemented to explore the impact of hardware-level modifications on quantized neural networks deployed on FPGA. Initial testing with default neural network models established referencing metrics for accuracy and resource allocation in Brevitas and FINN for later comparisons. Weight randomization was then applied in Brevitas, generating the various results to simulate the hardware level influence on accuracy. Following this, the Matrix-Vector Activation Unit (MVAU) in FINN was tested and modified to simulate weight randomization effect on FPGA resource utilization for the physical FPGA deployments.

This sequential approach provided a structured steps from baseline analysis to hardware modifications, enhancing its potential to access the accuracy and efficiency of physical FPGA-based neural networks deployments under varying configurations. This section effectively provided insights into the effects of hardware alterations on FPGA-implemented neural networks.

Chapter 5

Experimental Results and Analysis

5.1 Introduction

This chapter demonstrates the experimental results and analysis of testing FPGA-based quantized neural networks with focus of accuracy and resource utilization by implementing weight randomization and MVAU modifications on accuracy and resource allocation.

The results are broken down into phases to provide a well-structured analysis:

- Testing with Default Neural Networks: This phase publishes reference metrics for accuracy and resource allocation by using default neural network configurations in Brevitas and FINN. These results can provide a standard for comparing the effects of hardware-level modifications.
- Weight Randomization in Brevitas: This section analyses the impact of weight randomization to simulate the hardware-level modifications on model's accuracy.
- MVAU Modifications in FINN: The final phase indicates the impact of MVAU modifications on resource utilization changes with various neural networks to simulate the hardware-level adjustments.

This structured results and analysis present understanding of how accuracy and resource efficiency are affected by simulated hardware-level modifications. The findings in this chapter highlight the potential trade-offs and benefits of direct modifications to the actual FPGA hardware while implementing neural networks.

5.2 Results with default neural networks

The results involved training the model in Brevitas for accuracy and exporting it

to FINN for FPGA resource allocation analysis, resulting in establishment of a benchmark against those generated by simulated hardware-level modifications.

5.2.1 Accuracy in Brevitas

The default neural network model was trained and evaluated on the UNSW-NB15 dataset. The training process aimed to optimize the model's accuracy without any modifications providing a clear reference point for later comparisons.

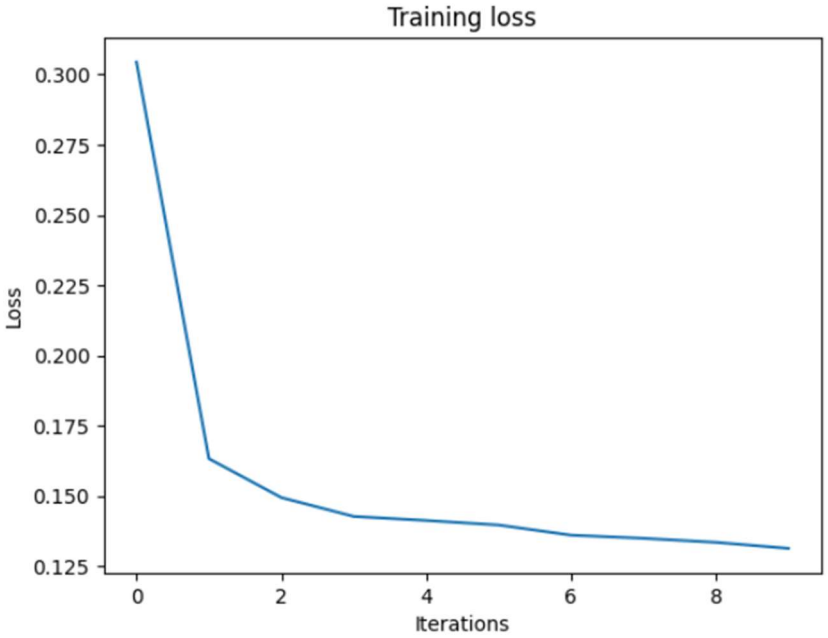
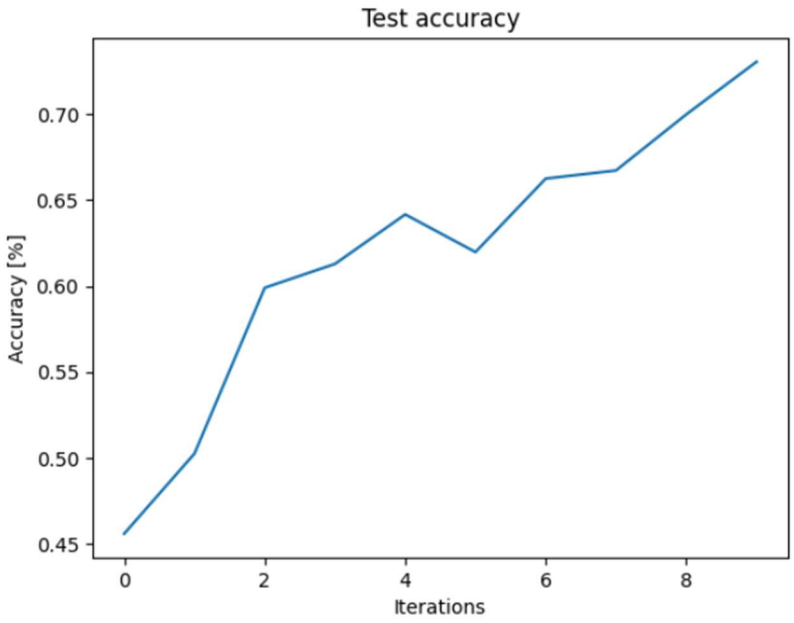


Figure 12. Training loss for default model



```
test(model, test_quantized_loader)
```

```
0.730505757178254
```

Figure 13. Accuracy for default model

- The first graph (figure 12) indicates the model's convergence as it learns from the dataset by showing the sharp decline trend over the iterations. The loss started high and progressively decreased to a low value which reflects the effectiveness of the training process.
- The second graph (figure 13) indicates the test accuracy improvement over successive iterations to reach a final accuracy of approximately 73%. This accuracy rate is set as the benchmark for analyzing the impact of weight randomization and hardware modifications in later phases.

5.2.2 Resource usage in FINN

The FINN framework was able to outline detailed hardware resources consumed by the default model when implemented on FPGA. The resources report is shown (figure 14).


```
{
  "vivado_proj_folder": "/tmp/finn_dev_su/synth_out_of_context_xjeubqhy/results_finn_design_wrapper",
  "LUT": 6586.0,
  "LUTRAM": 42.0,
  "FF": 7711.0,
  "DSP": 0.0,
  "BRAM": 22.0,
  "BRAM_18K": 0.0,
  "BRAM_36K": 22.0,
  "URAM": 0.0,
  "Carry": 302.0,
  "WNS": -0.297,
  "Delay": -0.297,
  "vivado_version": 2022.2,
  "vivado_build_no": 3671981.0,
  "": 0,
  "fmax_mhz": 97.11566475672525,
  "estimated_throughput_fps": 1517432.261823832
}
```


Figure 14. Resource report for default model

The estimated throughput reached approximately 1,514,732.26 frames per second (fps) with a maximum clock frequency of 97.12 MHz indicating that emphasizing the advantage which is high throughput and efficiency of using FPGAs for neural network deployment.

These default metrics provide a standard for quantifying and comparing the impact of modifications introduced in later phases.

5.3 Weight Randomization results in Brevitas

This section examines the impact of 10% weight randomization on accuracy fluctuation in Breviats. This analysis aims to prove the potential outcome such as increase in accuracy that directly modify hardware can bring through software simulation.

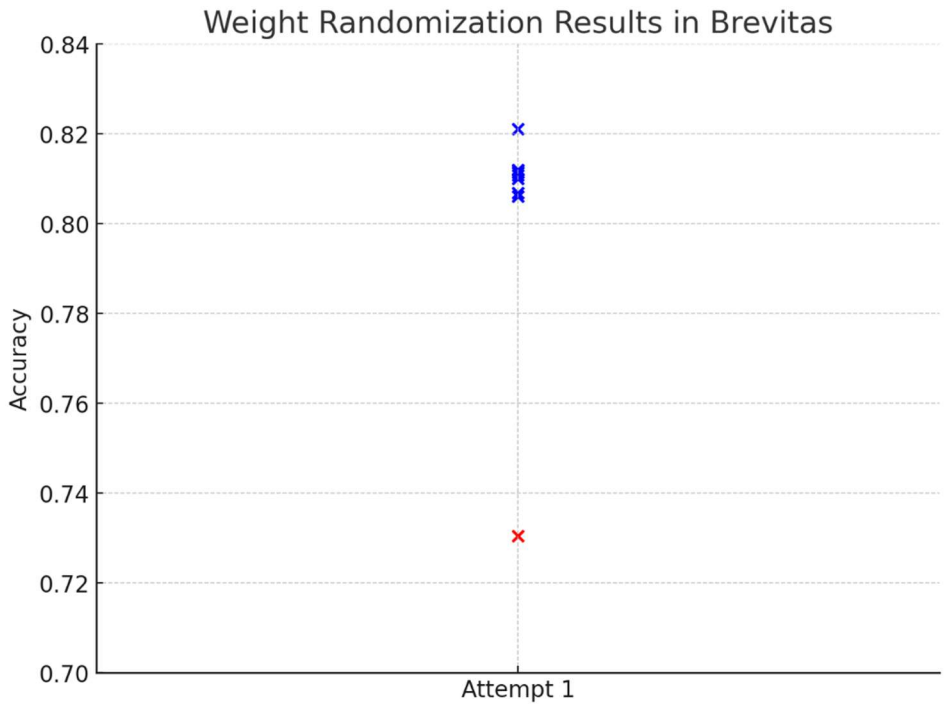


Figure 15. Weight Randomization results in Brevitas

Default Model	10% Randomization
0.730505	0.806794
	0.811786
	0.810851
	0.806151
	0.811446
	0.821029
	0.811981
	0.811422
	0.810098

Table 1. Accuracy results in Brevitas

5.3.1 Accuracy and randomization level relationship

In this experiment, 10% of the weights were randomly set to zero in a layer. This process was repeated 10 times to collect a set of accuracy measurements with proper reset to the weight to avoid accumulate changes. The blue crosses present accuracy results with randomization, whereas the red cross marks the

default accuracy without randomization.

The trend shows that the highest recorded accuracy with randomization could reach approximately 82.1% (0.821) compared to the original 73% (0.73) accuracy. (table 1) This means a significant improvement in accuracy at approximately 12.5% when applying randomized configuration. This is because the randomization process transits binary weight (-1,1) to ternary (-1,0,1) by adding zeros to weight. With the 3 states weight representation, the model gains more flexibility to fine-tune its ability to generalize and capture complex patterns form the data. Furthermore, 3 states weight also means that it increases the sparsity of the network to prevent it from becoming overly reliant on all weights. Hence, it leads to potential to increase the accuracy of the model like this case.

5.3.2 Optimal Configuration Selection

The observed accuracy improvement suggests that randomization could be introducing a form of regularization which increases the sparsity of the model to prevent it from becoming overly reliant on all weights. By setting 10% of weights to zero, the network may be forced to rely on more important weights, leading to a more robust and generalized model with higher accuracy.

This result indicates that optimal configuration such as the one has the highest accuracy can be remembered and selected every time. This means that tuning neural networks on the actual hardware to acquire the best performance is feasible.

5.4 MVAU Modifications results in FINN

This section introduces the effects of applying weight randomization modifications to the Matrix-Vector Activation Unit (MVAU) in FINN. The focus is to analyze resource utilization changes (LUTs and FFs) among various neural network configurations to understand the impact of MVAU designs in FPGA implementations.

5.4.1 Extreme test results

The initial step was testing MVAU resources usage by setting all weights to a constant value to confirm the feasibility of modifications. The resource

allocation results (figure 16) are shown,


```
{
  "vivado_proj_folder": "/tmp/finn_dev_su/synth_out_of_context_tq4cjji1/results_finn_design_wrapper",
  "LUT": 432.0,
  "LUTRAM": 0.0,
  "FF": 303.0,
  "DSP": 0.0,
  "BRAM": 0.0,
  "BRAM_18K": 0.0,
  "BRAM_36K": 0.0,
  "URAM": 0.0,
  "Carry": 40.0,
  "WNS": 1.806,
  "Delay": 1.806,
  "vivado_version": 2022.2,
  "vivado_build_no": 3671981.0,
  "": 0,
  "fmax_mhz": 122.040517451794,
  "estimated_throughput_fps": 1906883.0851842812
}
```


Figure 16. Resource report for extreme test

The number of LUT dropped dramatically from 6586 (figure 14) to 432 (figure 16) and FF was from 7711 (figure 14) to 303 (figure 16) respectively. This confirms that the MVAU is a modification target by demonstrating the related changes as expected. The changes should be large due to the modification which skipped most of processing units inside the design.

5.4.2 Results analysis with randomized weights

The next phase implemented weight randomization within the MVAU to observe its effects on resource utilization. Four different network models were utilized to measure the impacts on resources such as Look-Up Tables (LUTs) and Flip-Flops (FFs).

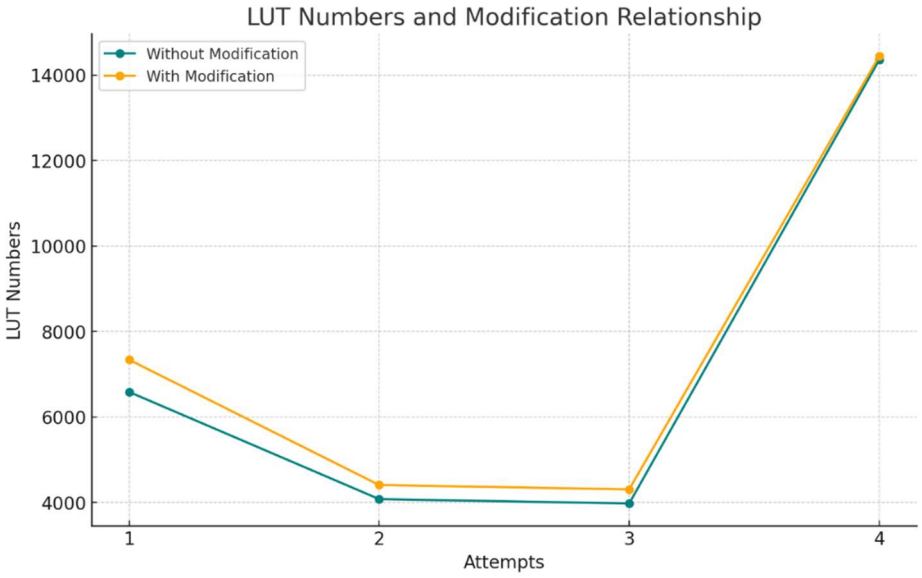


Figure 17. LUT numbers for four models with and without modifications

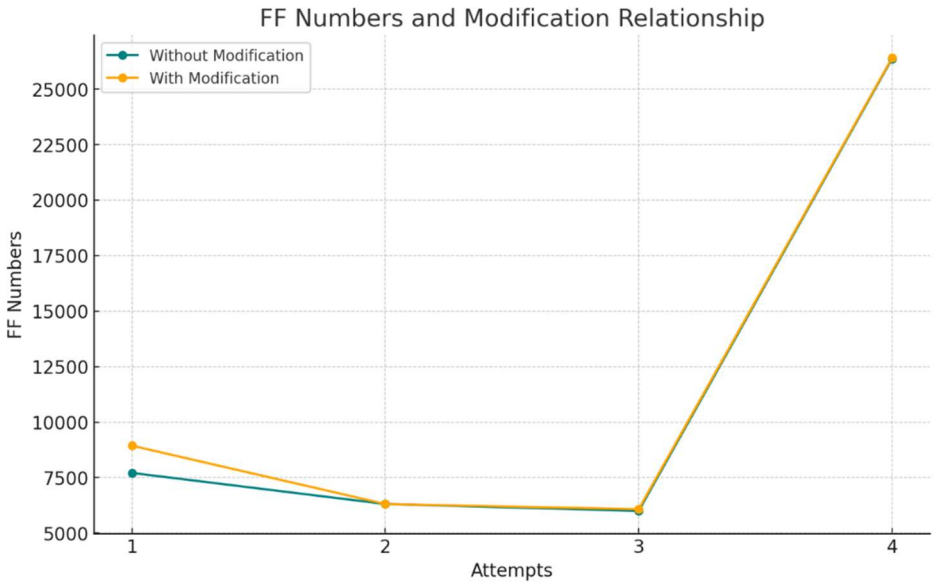


Figure 18. FF numbers for four models with and without modifications

Model Number	LUT (Without Modification)	LUT (With Modification)	FF (Without Modification)	FF (With Modification)
1	6586	7339	7711	8939
2	4079	4409	6312	6311
3	3977	4307	6000	6079
4	14354	14443	26371	26410

Table 2. LUT and FF numbers for 4 models with and without modifications

1. LUT Utilization
    - LUT usage varied significantly across models due to their network structures. (figure 17)
    - An increase in LUT usage occurred in all models which indicated that additional resources were required to perform the modification of randomness in weight values. For example, LUT usage raised from 6,586 to 7,339 approximately 11.4% increase for model 1, 8.1% for model 2, 8.3% for model 3 and 0.6% for model 4. The average was 7.1% increase overall. (Table 2.)
2. FF Utilization
    - FF usage also saw a slight increase with randomization, although the impact was less pronounced than for LUTs. Model 1's FF usage increased from 7,711 to 8,393 (15.9%), 0% increase for model 2, followed by 1.3% for model 3 and 0.1% for model 4. The average was 4.325% increase overall. (Table 2.)

Both LUT and FF utilization numbers increased slightly with weight randomization for all models. However, models with higher initial utilization

such as Model 4 could experience relatively small increase due to the already substantial resource allocation.

5.4.3 Resource efficiency Comparisons

The comparison between modified and unmodified configurations leads following conclusion:

- Resource Trade-offs: Weight randomization introduces a small but consistent increase in resource usage with a 7.1% increase in LUT and 4.325% FF counts in average. This increase highlights the trade-off between accuracy improvements (as mentioned in Weight Randomisation in Brevitas section) and additional hardware requirements.
- Hardware Flexibility: The results demonstrate that MVAU modifications can be applied with minimal impact on resource constraints while increasing slightly in accuracy.

In summary, the MVAU as a functional target for hardware modifications can allow direct changes to vary its resource allocation. This flexibility reinforces the potential for FPGA-based neural networks to incorporate hardware-level optimizations on the physical model that enhances performance without substantially increasing the cost.

5.5 Summary

This chapter demonstrates the experimental results and analysis of FPGA-based quantized neural network implementations by targeting on the effects of hardware-level modifications on model accuracy and resource allocation. The findings from each section can be integrated into how changes in neural network configuration and hardware design in software or software-based hardware impact the overall efficiency and performance of physical FPGA implementations. The summary of findings is listed below:

1. The initial results are 73% of accuracy and 6586 LUT and 7711 FF numbers recorded by testing default configurations in Brevitas and FINN.
2. Applying weight randomization demonstrates an accuracy improvement of around 12.5% through transition from binary to

ternary weight representation.

3. Modifying the MVAU with weight randomization can lead to an increase in resource utilization across multiple neural network models which has average of 7.1% increase in LUT and 4.325% increase in FF.

The experimental outcomes of hardware-level modifications such as weight randomization and MVAU adjustments can improve model accuracy and robustness. However, these improvement in accuracy can possibly come with a slight increase in resource utilization in this research.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

This research aims to explore the potential of hardware-level modifications to improve the accuracy and efficiency of quantized neural networks on FPGA platforms, focusing on techniques such as weight randomization within the Matrix-Vector Activation Unit (MVAU). The findings highlight that directly tuning FPGA hardware is an effective and low-cost approach for performance enhancement in environments requiring adaptability such as real-world autonomous navigation. Those key findings are also listed below,

1. **Efficiency and Effectiveness of Weight Randomization**
   By applying weight randomization technique, transitioning from binary (-1, 1) to ternary (-1, 0, 1) representations, this study can derive at most 12.5% of accuracy improvement. This method not only enhances performance but also only costs little in resource usage. This demonstrates that hardware tuning can produce significant benefits without adding substantial computational costs. The low-cost nature of this technique makes it ideal for environments where frequent updates or adaptive responses are expected by using a straightforward method for enhancing accuracy without reconfiguring the entire system.

2. **Targeted modifications for Optimizations**
   The MVAU was modified and tested with randomized weights, resulting in slight increases in resource usage (LUTs and FFs) but confirmed that FPGA-based neural networks can achieve improved performance (at most 12.5 %) and flexibility. The ability to make such direct changes on MVAU showing that this targeted modification has potential to be applied to other unit to further impact the accuracy.

3. **FPGA-Specific Optimizations**

FPGAs can offer a rapid and adaptable approach to performance tuning by applying direct modifications at the hardware level compared to fixed hardware platforms such as CPUs or GPUs. The results highlight the suitability of FPGAs for neural network deployments in environments that require constant reconfiguration, such as edge computing.

Overall Impact:
The outcomes of this project address that hardware tuning on FPGAs is a cheap, quick, and effective strategy for neural network optimization with minimal impact on resource usage. This approach is highly appliable in environments where constant updates or adaptive responses are required due to its a simple but effective techniques to improve accuracy without compromising hardware efficiency. These findings contribute to a broader understanding of software-hardware modifications and highlight to the benefits of FPGA-based tuning for real-time processing applications.

In summary, this thesis identifies the potential of FPGA-based neural network optimization can be achieved through efficient and simple hardware tuning.

6.2 Achievements

The key achievements and contributions of this study are

1. Weight Randomization technique developed to increase the accuracy of the FPGA-based neural networks
2. Validate the Matrix-Vector Activation Unit (MVAU) as an effective target for resource-efficient hardware modifications while achieving good accuracy.
3. Address that FPGA tuning can provide quick and low-cost accuracy enhancements without changing hardware or software substantially.

6.3 Learning outcomes

The key learning outcomes are

1. Identification of FPGA hardware tuning as a quick and cost-effective

method for neural network optimization,
2. Developments of hardware-software co-design using Brevitas and FINN
3. Understanding the balance between accuracy and resource efficiency in hardware.

6.4 Future work

The success of demonstration of hardware-level modifications on FPGA-based neural networks provides a further development in neural network optimization. Future work aims to extend the adaptability and efficiency of these approaches to apply them to more complex neural network architectures and environments that require resource sensitive solutions.

The MVAU and weight randomization techniques present several opportunities for continued exploration. For example, if these modifications have potential to be applied to large convolutional neural networks by providing scalability to the broader applications. Additionally, implementing adaptive quantization can lead to finer control over resource allocation and energy efficiency by altering bit widths based on performance needs.

Physical FPGA deployment also presents further opportunities where testing these configurations on physical hardware would effectively validate the findings in practical scenarios by providing deeper insights into latency, energy consumption, and fault tolerance under various conditions. Studies focusing on energy-efficient designs such as low-power applications could design solutions ideal for remote or battery-operated devices, for instance, technologies about medical wearables and IoT sensors. In conclusion, all these applications provide further insight into the future direction of FPGA-based neural can be in multiple dimensions.

Appendix

Appendix 1: Codes to train the default neural network model

Train a Quantized MLP on UNSW-NB15 with Brevitas

In [2]:
```
import os
import onnx
import torch

model_dir = os.environ['FINN_ROOT'] + "/notebooks/end2end_example/cybersecurity"

! wget -O unsw_nb15_binarized.npz https://zenodo.org/record/4519767/files/unsw_nb15_binarized.npz?download=1
```

In [4]:
```
import numpy as np
from torch.utils.data import TensorDataset

def get_preqnt_dataset(data_dir: str, train: bool):
    unsw_nb15_data = np.load(data_dir + "/unsw_nb15_binarized.npz")
    if train:
        partition = "train"
    else:
        partition = "test"
    part_data = unsw_nb15_data[partition].astype(np.float32)
    part_data = torch.from_numpy(part_data)
    part_data_in = part_data[:, :-1]
    part_data_out = part_data[:, -1]
    return TensorDataset(part_data_in, part_data_out)

train_quantized_dataset = get_preqnt_dataset(".", True)
test_quantized_dataset = get_preqnt_dataset(".", False)

print("Samples in each set: train = %d, test = %s" % (len(train_quantized_dataset), len(test_quantized_dataset)))
print("Shape of one input sample: " +  str(train_quantized_dataset[0][0].shape))
```

```
Samples in each set: train = 175341, test = 82332
Shape of one input sample: torch.Size([593])
```

In [5]:
```
from torch.utils.data import DataLoader, Dataset

batch_size = 1000

# dataset loaders
train_quantized_loader = DataLoader(train_quantized_dataset, batch_size=batch_size, shuffle=True)
test_quantized_loader = DataLoader(test_quantized_dataset, batch_size=batch_size, shuffle=False)
```

In [6]:
```
count = 0
for x,y in train_quantized_loader:
    print("Input shape for 1 batch: " + str(x.shape))
    print("Label shape for 1 batch: " + str(y.shape))
    count += 1
    if count == 1:
        break
```

```
Input shape for 1 batch: torch.Size([1000, 593])
Label shape for 1 batch: torch.Size([1000])
```

In [7]:
```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Target device: " + str(device))
```

```
Target device: cpu
```

In [8]:
```
input_size = 593
hidden1 = 64
hidden2 = 64
hidden3 = 64
weight_bit_width = 2
act_bit_width = 2
num_classes = 1
```

In [9]:
```
from brevitas.nn import QuantLinear, QuantReLU
import torch.nn as nn

# Setting seeds for reproducibility
torch.manual_seed(0)

model = nn.Sequential(
      QuantLinear(input_size, hidden1, bias=True, weight_bit_width=weight_bit_width),
      nn.BatchNorm1d(hidden1),
      nn.Dropout(0.5),
      QuantReLU(bit_width=act_bit_width),
      QuantLinear(hidden1, hidden2, bias=True, weight_bit_width=weight_bit_width),
      nn.BatchNorm1d(hidden2),
      nn.Dropout(0.5),
      QuantReLU(bit_width=act_bit_width),
      QuantLinear(hidden2, hidden3, bias=True, weight_bit_width=weight_bit_width),
      nn.BatchNorm1d(hidden3),
      nn.Dropout(0.5),
      QuantReLU(bit_width=act_bit_width),
      QuantLinear(hidden3, num_classes, bias=True, weight_bit_width=weight_bit_width)
)
```

In [9]:
```
def train(model, train_loader, optimizer, criterion):
    losses = []
    # ensure model is in training mode
    model.train()

    for i, data in enumerate(train_loader, 0):
        inputs, target = data
        inputs, target = inputs.to(device), target.to(device)
        optimizer.zero_grad()

        # forward pass
        output = model(inputs.float())
        loss = criterion(output, target.unsqueeze(1))

        # backward pass + run optimizer to update weights
        loss.backward()
        optimizer.step()

        # keep track of loss value
        losses.append(loss.data.cpu().numpy())

    return losses
```

In [10]:
```
import torch
from sklearn.metrics import accuracy_score

def test(model, test_loader):
    # ensure model is in eval mode
    model.eval()
    y_true = []
    y_pred = []

    with torch.no_grad():
        for data in test_loader:
            inputs, target = data
            inputs, target = inputs.to(device), target.to(device)
            output_orig = model(inputs.float())
            # run the output through sigmoid
            output = torch.sigmoid(output_orig)
            # compare against a threshold of 0.5 to generate 0/1
            pred = (output.detach().cpu().numpy() > 0.5) * 1
            target = target.cpu().float()
            y_true.extend(target.tolist())
            y_pred.extend(pred.reshape(-1).tolist())

    return accuracy_score(y_true, y_pred)
```

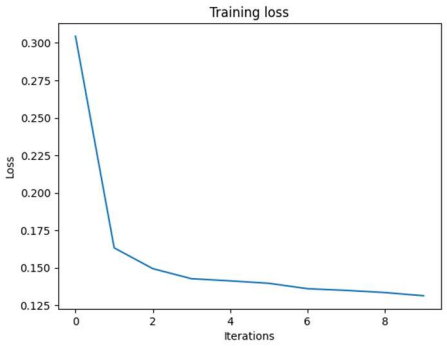
In [11]:
```
num_epochs = 10
lr = 0.001

def display_loss_plot(losses, title="Training loss", xlabel="Iterations", ylabel="Loss"):
    x_axis = [i for i in range(len(losses))]
    plt.plot(x_axis,losses)
    plt.title(title)
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.show()
```

In [12]:
```
# Loss criterion and optimizer
criterion = nn.BCEWithLogitsLoss().to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=lr, betas=(0.9, 0.999))
```

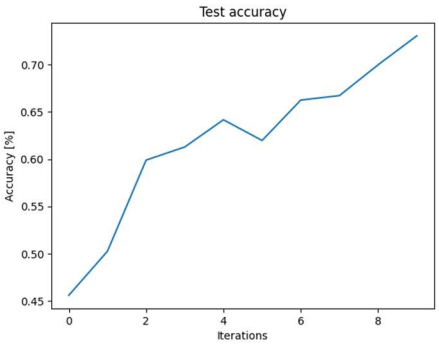
In [13]:
```
import numpy as np
from sklearn.metrics import accuracy_score
from tqdm import tqdm, trange

# Setting seeds for reproducibility
torch.manual_seed(0)
np.random.seed(0)

running_loss = []
running_test_acc = []
t = trange(num_epochs, desc="Training loss", leave=True)

for epoch in t:
        loss_epoch = train(model, train_quantized_loader, optimizer,criterion)
        test_acc = test(model, test_quantized_loader)
        t.set_description("Training loss = %f test accuracy = %f" % (np.mean(loss_epoch), test_acc))
        t.refresh() # to show immediately the update
        running_loss.append(loss_epoch)
        running_test_acc.append(test_acc)
```

```
Training loss:   0%|                                                                                | 0/10 [00:00<?, ?it/s]/usr/local/lib/python3.10/dist-packages/torch/_tensor.py:1255: UserWarning: Named tens
ors and all their associated APIs are an experimental feature and subject to change. Please do not use them for anything important until they are released as stable. (Triggered internally at ../c10/core/Ten
sorImpl.h:1758.)
  return super(Tensor, self).rename(names)
Training loss = 0.131306 test accuracy = 0.730506: 100%|████████████████████████| 10/10 [00:28<00:00,  2.83s/it]
```

In [14]:
```
%matplotlib inline
import matplotlib.pyplot as plt

loss_per_epoch = [np.mean(loss_per_epoch) for loss_per_epoch in running_loss]
display_loss_plot(loss_per_epoch)
```



In [15]:
```
acc_per_epoch = [np.mean(acc_per_epoch) for acc_per_epoch in running_test_acc]
display_loss_plot(acc_per_epoch, title="Test accuracy", ylabel="Accuracy [%]")
```



In [16]:
```
test(model, test_quantized_loader)
```

Out[16]:
```
0.730505757178254
```

Appendix 2: Codes to collect the hardware information from default model

Launch a Build: Stitched IP, out-of-context synth Performance

In [ ]:
```
import finn.builder.build_dataflow as build
import finn.builder.build_dataflow_config as build_cfg
import os
import shutil

model_file = model_dir + "/cybsec-mlp-ready.onnx"

rtlsim_output_dir = "output_ipstitch_ooc_rtlsim"

#Delete previous run results if exist
if os.path.exists(rtlsim_output_dir):
    shutil.rmtree(rtlsim_output_dir)
    print("Previous run results deleted!")

cfg_stitched_ip = build.DataflowBuildConfig(
    output_dir          = rtlsim_output_dir,
    mvau_wwidth_max     = 80,
    target_fps          = 1000000,
    synth_clk_period_ns = 10.0,
    fpga_part           = "xc7z020clg400-1",
    generate_outputs=[
        build_cfg.DataflowOutputType.STITCHED_IP,
        build_cfg.DataflowOutputType.RTLSIM_PERFORMANCE,
        build_cfg.DataflowOutputType.OOC_SYNTH,
    ]
)
```

In [ ]:
```
%%time
build.build_dataflow_cfg(model_file, cfg_stitched_ip)
```

In [ ]:
```
assert os.path.exists(rtlsim_output_dir + "/report/ooc_synth_and_timing.json")
```

Among the output products, we will find the accelerator exported as a stitched IP block design:

In [ ]:
```
! ls {rtlsim_output_dir}/stitched_ip
```

We also have a few reports generated by these output products, different from the ones generated by `ESTIMATE_REPORTS`.

In [ ]:
```
! ls {rtlsim_output_dir}/report
```

In `ooc_synth_and_timing.json` we can find the post-synthesis and maximum clock frequency estimate for the accelerator. Note that the clock frequency estimate here tends to be optimistic, since out-of-context synthesis is less constrained.

In [ ]:
```
! cat {rtlsim_output_dir}/report/ooc_synth_and_timing.json
```

In `rtlsim_performance.json` we can find the steady-state throughput and latency for the accelerator, as obtained by rtlsim. If the DRAM bandwidth numbers reported here are below what the hardware platform is capable of (i.e. the accelerator is not memory-bound), you can expect the same steady-state throughput (excluding any software/driver overheads) in real hardware.

Appendix 3: Complete weight randomization codes in Brevitas

In [ ]:
```
def train(model, train_loader, optimizer, criterion):
    losses = []
    # ensure model is in training mode
    model.train()

    for i, data in enumerate(train_loader, 0):
        inputs, target = data
        inputs, target = inputs.to(device), target.to(device)
        optimizer.zero_grad()

        # forward pass
        output = model(inputs.float())
        loss = criterion(output, target.unsqueeze(1))

        # backward pass + run optimizer to update weights
        loss.backward()
        optimizer.step()

        # keep track of loss value
        losses.append(loss.data.cpu().numpy())

    return losses
```

In [ ]:
```
import torch
from sklearn.metrics import accuracy_score

def apply_weight_mask(layer, mask_ratio):
    """
    Apply a random mask to the weights of the input layer.

        layer: Different types neural network layer (e.g., QuantLinear).
        mask_ratio: The precentage of weights to zero out (between 0 and 1).

    """
    with torch.no_grad():
        # Find the weight tensor
        weight = layer.weight

        # Create a mask with the same shape as the previous weight tensor
        mask = torch.ones_like(weight)

        # Get the number of elements to zero out
        num_elements = weight.numel()
        num_zero = int(mask_ratio * num_elements)

        # Genearte random indices to zero out
        indices = torch.randperm(num_elements)[:num_zero]

        # zero out selected indices
        mask.view(-1)[indices] = 0

        # apply the mask to the weights
        weight.data.mul_(mask)
```

```python
def test_with_weight_mask(model, test_loader, mask_ratio):
    """
    Test the model with a random weight mask applied during inference.

    """
    # Ensure model is in evaluation mode
    model.eval()
    y_true = []
    y_pred = []

    # Apply the weight mask to each QuantLinear layer
    for layer in model:
        if isinstance(layer, QuantLinear):
            apply_weight_mask(layer, mask_ratio)

    with torch.no_grad():
        for data in test_loader:
            inputs, target = data
            inputs, target = inputs.to(device), target.to(device)
            output_orig = model(inputs.float())
            # Apply sigmoid activation
            output = torch.sigmoid(output_orig)
            # Threshold outputs at 0.5 to obtain binary predictions
            pred = (output.detach().cpu().numpy() > 0.5).astype(int)
            target = target.cpu().float()
            y_true.extend(target.tolist())
            y_pred.extend(pred.reshape(-1).tolist())

    # Return accuracy
    accuracy = accuracy_score(y_true, y_pred)
    return accuracy
```

In [ ]:
```python
# loss criterion and optimizer
criterion = nn.BCEWithLogitsLoss().to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=lr, betas=(0.9, 0.999))
mask_ratio = 0.1
```

In [ ]:
```python
import numpy as np
from sklearn.metrics import accuracy_score
from tqdm import tqdm, trange


# Setting seeds for reproducibility
torch.manual_seed(0)
np.random.seed(0)

running_loss = []
running_test_acc = []
t = trange(num_epochs, desc="Training loss", leave=True)

for epoch in t:
        loss_epoch = train(model, train_quantized_loader, optimizer,criterion)
        test_acc = test_with_weight_mask(model, test_quantized_loader, mask_ratio)
        t.set_description("Training loss = %f test accuracy = %f" % (np.mean(loss_epoch), test_acc))
        t.refresh() # to show immediately the update
        running_loss.append(loss_epoch)
        running_test_acc.append(test_acc)
```

Appendix 4: extreme test implementation in mvau.app file. *Note: original mavu.app file can be accessed through: https://github.com/Xilinx/finn-hlslib/blob/master/mvau.hpp

```
52  #define MVAU_HPP
53
54  #include "hls_stream.h"
55
56  #include "mac.hpp"
57  #include "interpret.hpp"
58  #include "weights.hpp"
59  #define WEIGHTS_ZERO  // Uncomment to set all weights to zero
60  // #define WEIGHTS_ONE  // Uncomment to set all weights to one
61
62  /**
63   * \brief Matrix vector activate function
64   *
65   * The function performs the multiplication between a weigth matrix and the input activation
    vector,
66   * accumulating the results and then applying an activation function on the accumulated
    result.
67   *
68   *
69   * \tparam MatrixW    Width of the input matrix
70   * \tparam MatrixH    Heigth of the input matrix
71   * \tparam SIMD       Number of input columns computed in parallel
72   * \tparam PE         Number of output rows computed in parallel
73   * \tparam MMV        Number of output pixels computed in parallel
74   * \tparam TSrcI      DataType of the input activation (as used in the MAC)
75   * \tparam TDstI      DataType of the output activation (as generated by the activation)
76   * \tparam TWeightI   DataType of the weights and how to access them in the array
77   * \tparam TI         DataType of the input stream - safely deducible from the paramaters
78   * \tparam TO         DataType of the output stream - safely deducible from the paramaters
79   * \tparam TW         DataType of the weights matrix - safely deducible from the paramaters
80   * \tparam TA         DataType of the activation class (e.g. thresholds) - safely deducible
    from the paramaters
81   * \tparam R          Datatype for the resource used for FPGA implementation of the MAC  -
    safely deducible from the paramaters
82   *
83   * \param in          Input stream
84   * \param out         Output stream
85   * \param weights     Weights matrix (currently supports BinaryWeights or FixedPointWeights)
86   * \param activation  Activation class
87   * \param reps        Number of time the function has to be repeatedly executed (e.g. number
    of images)
88   * \param r           Resource type for the hardware implementation of the MAC block
89   */
90  template<
91    unsigned MatrixW, unsigned MatrixH, unsigned SIMD, unsigned PE, unsigned MMV,
92    typename TSrcI = Identity, typename TDstI = Identity, typename TWeightI = Identity,
93    typename TI, typename TO, typename TW, typename TA, typename R
94  >
95  void Matrix_Vector_Activate_Batch(hls::stream<TI> &in,
96                    hls::stream<TO> &out,
97                    TW  const &weights,
98                    TA  const &activation,
99                    int const  reps,
100                   R const &r) {
```

```
256       // read input from stream
257       inElem = in.read();
258       // store in appropriate buffer for reuse
259       inputBuf[sf] = inElem;
260     }
261     else {
262       // reuse buffered input
263       inElem = inputBuf[sf];
264     }
265
266     // read from the parameter stream
267     W_packed = weight.read();
268     for (unsigned pe = 0; pe < PE; pe++) {
269 #pragma HLS UNROLL
270
271     #ifdef WEIGHTS_ZERO
272         // Skip MAC operation since weights are zero
273         // Optionally reset accumulator if needed
274         // accu[0][pe] = activation.init(nf, pe);
275     #else
276         auto const act = TSrcI()(inElem, 0);
277         auto const wgt =
278             #ifdef WEIGHTS_ONE
279                 static_cast<decltype(TWeightI()(w[pe]))>(1);
280             #else
281                 TWeightI()(w[pe]);
282             #endif
283         accu[0][pe] = mac<SIMD>(accu[0][pe], wgt, act, r, 0);
284     #endif
285
286
287     // Threshold Initialisation
288     if(sf == 0) {
289       for(unsigned pe = 0; pe < PE; pe++) {
290 #pragma HLS UNROLL
291       accu[0][pe] = activation.init(nf, pe);
292       }
293     }
294
295     // compute matrix-vector product for each processing element
296     for(unsigned  pe = 0; pe < PE; pe++) {
297 #pragma HLS UNROLL
298       auto const  act = TSrcI()(inElem, 0);
299       auto const  wgt = TWeightI()(w[pe]);
300       //auto const  wgt = w[pe];
301       accu[0][pe] = mac<SIMD>(accu[0][pe], wgt, act, r, 0);
302     }
303
304     // keep track of which folded synapse/neuron we are processing
305     ++tile;
306     if(++sf == SF) {
307       // produce output and clear accumulators
308       auto  outElem = TDstI().template operator()<TO>();
309       for (unsigned  pe = 0; pe < PE; pe++) {
```

Appendix 5: Weight randomization implementation in mvau.app file

\\wsl.localhost\Ubuntu-22.04\home\su\Xilinx\finn\deps\finn-hlslib\mvau.hpp

```
/******************************************************************************
 *  Copyright (c) 2019, Xilinx, Inc.
 *  All rights reserved.
 *
 *  Redistribution and use in source and binary forms, with or without
 *  modification, are permitted provided that the following conditions are met:
 *
 *  1.  Redistributions of source code must retain the above copyright notice,
 *     this list of conditions and the following disclaimer.
 *
 *  2.  Redistributions in binary form must reproduce the above copyright
 *      notice, this list of conditions and the following disclaimer in the
 *      documentation and/or other materials provided with the distribution.
 *
 *  3.  Neither the name of the copyright holder nor the names of its
 *      contributors may be used to endorse or promote products derived from
 *      this software without specific prior written permission.
 *
 *  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
 *  AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
 *  THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
 *  PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR
 *  CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
 *  EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
 *  PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS;
 *  OR BUSINESS INTERRUPTION). HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
 *  WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR
 *  OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF
 *  ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 *
 *****************************************************************************/

/******************************************************************************
 *
 *  Authors: Giulio Gambardella <giuliog@xilinx.com>
 *           Thomas B. Preusser <thomas.preusser@utexas.edu>
 *             Marie-Curie Fellow, Xilinx Ireland, Grant Agreement No. 751339
 *           Christoph Doehring <cdoehrin@xilinx.com>
 *
 *  \file mvau.hpp
 *
 *  This file lists a templated funtion used to implement
 *  Matrix-Vector-Activation Unit
 *
 *  This project has received funding from the European Union's Framework
 *  Programme for Research and Innovation Horizon 2020 (2014-2020) under
 *  the Marie Skłodowska-Curie Grant Agreement No. 751339.
 *
 *****************************************************************************/

#ifndef MVAU_HPP
```

```
52  #define MVAU_HPP
53
54  #include "hls_stream.h"
55
56  #include "mac.hpp"
57  #include "interpret.hpp"
58  #include "weights.hpp"
59  #define WEIGHTS_RANDOM
60
61  /**
62   * \brief Matrix vector activate function
63   *
64   * The function performs the multiplication between a weigth matrix and the input activation
    vector,
65   * accumulating the results and then applying an activation function on the accumulated
    result.
66   *
67   *
68   * \tparam MatrixW    Width of the input matrix
69   * \tparam MatrixH    Heigth of the input matrix
70   * \tparam SIMD       Number of input columns computed in parallel
71   * \tparam PE         Number of output rows computed in parallel
72   * \tparam MMV        Number of output pixels computed in parallel
73   * \tparam TSrcI      DataType of the input activation (as used in the MAC)
74   * \tparam TDstI      DataType of the output activation (as generated by the activation)
75   * \tparam TWeightI   DataType of the weights and how to access them in the array
76   * \tparam TI         DataType of the input stream - safely deducible from the paramaters
77   * \tparam TO         DataType of the output stream - safely deducible from the paramaters
78   * \tparam TW         DataType of the weights matrix - safely deducible from the paramaters
79   * \tparam TA         DataType of the activation class (e.g. thresholds) - safely deducible
    from the paramaters
80   * \tparam R          Datatype for the resource used for FPGA implementation of the MAC  -
    safely deducible from the paramaters
81   *
82   * \param in          Input stream
83   * \param out         Output stream
84   * \param weights     Weights matrix (currently supports BinaryWeights or FixedPointWeights)
85   * \param activation  Activation class
86   * \param reps        Number of time the function has to be repeatedly executed (e.g. number
    of images)
87   * \param r           Resource type for the hardware implementation of the MAC block
88   */
89  template<
90    unsigned MatrixW, unsigned MatrixH, unsigned SIMD, unsigned PE, unsigned MMV,
91    typename TSrcI = Identity, typename TDstI = Identity, typename TWeightI = Identity,
92    typename TI, typename TO, typename TW, typename TA, typename R
93  >
94  void Matrix_Vector_Activate_Batch(hls::stream<TI> &in,
95                    hls::stream<TO> &out,
96                    TW  const &weights,
97                    TA  const &activation,
98                    int const  reps,
99                    R const &r) {
100
```

```
  // how many different rows each neuron will compute
  // alternatively: number of vertical matrix chunks
  unsigned const  NF = MatrixH / PE;

  // how many synapse groups each row is split into
  // alternatively: number of horizontal matrix chunks
  unsigned const  SF = MatrixW / SIMD;

  // input vector buffers
  TI  inputBuf[SF];
#pragma HLS ARRAY_PARTITION variable=inputBuf complete dim=0


  decltype(activation.init(0,0))  accu[MMV][PE];
#pragma HLS ARRAY_PARTITION variable=accu complete dim=0

  unsigned  nf   = 0;
  unsigned  sf   = 0;
  unsigned  tile = 0; // invariant: tile = nf*SF + sf

  // everything merged into a common iteration space (one "big" loop instead
  // of smaller nested loops) to get the pipelinening the way we want
  unsigned const TOTAL_FOLD = NF * SF;
  for(unsigned  i = 0; i < reps * TOTAL_FOLD; i++) {
#pragma HLS pipeline style=flp II=1
    TI  inElem;
    if(nf == 0) {
      // read input from stream
      inElem = in.read();
      // store in appropriate buffer for reuse
      inputBuf[sf] = inElem;
    }
    else {
      // reuse buffered input
      inElem = inputBuf[sf];
    }

    // Threshold Initialisation
    if(sf == 0) {
      for(unsigned  pe = 0; pe < PE; pe++) {
#pragma HLS UNROLL
        for(unsigned mmv = 0; mmv < MMV; mmv++) {
#pragma HLS UNROLL
          accu[mmv][pe] = activation.init(nf, pe);
        }
      }
    }

    // compute matrix-vector product for each processing element
    auto const &w = weights.weights(tile);
    for(unsigned  pe = 0; pe < PE; pe++) {
#pragma HLS UNROLL
      auto const  wgt = TWeightI()(w[pe]);
      for (unsigned mmv = 0; mmv < MMV; mmv++){
```

```
155         auto const  act = TSrcI()(inElem, mmv);
156         accu[mmv][pe] = mac<SIMD>(accu[mmv][pe], wgt, act, r, mmv);
157       }
158     }
159
160     // keep track of which folded synapse/neuron we are processing
161     ++tile;
162     if(++sf == SF) {
163       // produce output and clear accumulators
164       auto  outElem = TDstI().template operator()<TO>();
165       for (unsigned  pe = 0; pe < PE; pe++) {
166 #pragma HLS UNROLL
167         for (unsigned mmv = 0; mmv < MMV; mmv++){
168 #pragma HLS UNROLL
169           outElem(pe,mmv,1) = activation.activate(nf, pe, accu[mmv][pe]);
170         }
171       }
172       out.write(outElem);
173       // next folded neuron or image
174       sf = 0;
175       if(++nf == NF) {
176         nf   = 0;
177         tile = 0;
178       }
179     }
180   }
181 }
182
183
184 /**
185  * \brief Matrix vector activate function with streaming weights
186  *
187  * The function performs the multiplication between a weigth matrix, presnted as an input
    stream, and the input activation vector,
188  * accumulating the results and then applying an activation function on the accumulated
    result. Does not support MMV.
189  *
190  *
191  * \tparam MatrixW    Width of the input matrix
192  * \tparam MatrixH    Heigth of the input matrix
193  * \tparam SIMD       Number of input columns computed in parallel
194  * \tparam PE         Number of output rows computed in parallel
195  * \tparam TSrcI      DataType of the input activation (as used in the MAC)
196  * \tparam TDstI      DataType of the output activation (as generated by the activation)
197  * \tparam TWeightI   DataType of the weights and how to access them in the array
198  * \tparam TW         DataType of the weights (as used in the MAC) - not deducible from the
    paramaters
199  * \tparam TI         DataType of the input stream - safely deducible from the paramaters
200  * \tparam TO         DataType of the output stream - safely deducible from the paramaters
201  * \tparam TA         DataType of the activation class (e.g. thresholds) - safely deducible
    from the paramaters
202  * \tparam R          Datatype for the resource used for FPGA implementation of the MAC  -
    safely deducible from the paramaters
203  *
```

```
 * \param in          Input stream
 * \param out         Output stream
 * \param weight      Weight stream (currently supports BinaryWeights or FixedPointWeights)
 * \param activation  Activation class
 * \param reps        Number of time the function has to be repeatedly executed (e.g. number
of images)
 * \param r           Resource type for the hardware implementation of the MAC block
 */
template<
  unsigned MatrixW, unsigned MatrixH, unsigned SIMD, unsigned PE,
  typename TSrcI = Identity, typename TDstI = Identity, typename TWeightI = Identity,
typename TW,
  typename TI, typename TO, typename TA, typename R
>
void Matrix_Vector_Activate_Stream_Batch(hls::stream<TI> &in,
          hls::stream<TO> &out,
          hls::stream<ap_uint<PE*SIMD*TW::width>> &weight,
          TA  const &activation,
          int const  reps,
          R const &r) {

  // how many different rows each neuron will compute
  // alternatively: number of vertical matrix chunks
  unsigned const  NF = MatrixH / PE;

  // how many synapse groups each row is split into
  // alternatively: number of horizontal matrix chunks
  unsigned const  SF = MatrixW / SIMD;

  // input vector buffers
  TI  inputBuf[SF];
#pragma HLS ARRAY_PARTITION variable=inputBuf complete dim=1
  // accumulators
  decltype(activation.init(0,0))  accu[1][PE];
#pragma HLS ARRAY_PARTITION variable=accu complete dim=0
  // unpacked and packed buffers for weight stream
  Weights_Tile<SIMD, TW, PE > w;
#pragma HLS ARRAY_PARTITION variable=w.m_weights complete dim=0
  ap_uint<PE * SIMD * TW::width> W_packed;

  unsigned int rand_counter = 0;
  unsigned  nf   = 0;
  unsigned  sf   = 0;
  unsigned  tile = 0; // invariant: tile = nf*SF + sf

  // everything merged into a common iteration space (one "big" loop instead
  // of smaller nested loops) to get the pipelinening the way we want
  unsigned const TOTAL_FOLD = NF * SF;
  for(unsigned  i = 0; i < reps * TOTAL_FOLD; i++) {
#pragma HLS pipeline style=flp II=1
    TI  inElem;

    if(nf == 0) {
      // read input from stream
```

```
256       inElem = in.read();
257       // store in appropriate buffer for reuse
258       inputBuf[sf] = inElem;
259     }
260     else {
261       // reuse buffered input
262       inElem = inputBuf[sf];
263     }
264
265     // read from the parameter stream
266     W_packed = weight.read();
267     for (unsigned pe = 0; pe < PE; pe++) {
268 #pragma HLS UNROLL
269       w.m_weights[pe] = W_packed((pe+1)*SIMD*TW::width-1,pe*SIMD*TW::width);
270     }
271
272     // Randomly zero out weights
273     #ifdef WEIGHTS_RANDOM
274     // Increment the counter
275     rand_counter++;
276     // Generate a pseudo-random bit using XOR operations
277     bool rand_bit = (rand_counter ^ (rand_counter >> 1)) & 0x1;
278     for(unsigned pe = 0; pe < PE; pe++) {
279         #pragma HLS UNROLL
280         if (rand_bit == 0) {
281             w.m_weights[pe] = 0;
282         }
283     }
284     #endif
285
286
287     // Threshold Initialisation
288     if(sf == 0) {
289       for(unsigned pe = 0; pe < PE; pe++) {
290 #pragma HLS UNROLL
291       accu[0][pe] = activation.init(nf, pe);
292       }
293     }
294
295     // compute matrix-vector product for each processing element
296     for(unsigned  pe = 0; pe < PE; pe++) {
297 #pragma HLS UNROLL
298       auto const  act = TSrcI()(inElem, 0);
299       auto const  wgt = TWeightI()(w[pe]);
300       //auto const  wgt = w[pe];
301       accu[0][pe] = mac<SIMD>(accu[0][pe], wgt, act, r, 0);
302     }
303
304     // keep track of which folded synapse/neuron we are processing
305     ++tile;
306     if(++sf == SF) {
307       // produce output and clear accumulators
308       auto  outElem = TDstI().template operator()<TO>();
309       for (unsigned  pe = 0; pe < PE; pe++) {
```

Implementation location shown above.

```
310 #pragma HLS UNROLL
311       outElem(pe,0,1) = activation.activate(nf, pe, accu[0][pe]);
312           }
313 
314       out.write(outElem);
315 
316       // next folded neuron or image
317       sf = 0;
318       if(++nf == NF) {
319       nf   = 0;
320       tile = 0;
321       }
322     }
323   }
324 }
325 
326 #endif
```

Bibliography

[1] M. A. Talib, S. Majzoub, Q. Nasir, and D. Jamal, "A systematic literature review on hardware implementation of artificial intelligence algorithms," *The Journal of Supercomputing*, vol. 77, pp. 1897–1938, May 2020, doi: 10.1007/s11227-020-03325-8.

[2] J. Duarte, S. Han, P. Harris, S. Jindariani, E. Kreinar, B. Kreis, J. Ngadiuba, M. Pierini, R. Rivera, N. Tran, and Z. Wu, "Fast inference of deep neural networks in FPGAs for particle physics," *Journal of Instrumentation*, vol. 13, P07027, July 2018, doi: 10.1088/1748-0221/13/07/P07027.

[3] A. Shawahna, S. M. Sait, and A. El-Maleh, "FPGA-based accelerators of deep learning networks for learning and classification: A review," *IEEE Access*, vol. 7, pp. 7823–7859, Jan. 2019, doi: 10.1109/ACCESS.2018.2890150.

[4] G. Lacey, G. Taylor, and S. Areibi, "Deep learning on FPGAs: Past, present, and future," *arXiv preprint arXiv:1602.04283*, Feb. 2016.

[5] M. Capra, B. Bussolino, A. Marchisio, G. Masera, M. Martina, and M. Shafique, "Hardware and software optimizations for accelerating deep neural networks: Survey of current trends, challenges, and the road ahead," *IEEE Access*, vol. 8, pp. 225134–225156, Dec. 2020, doi: 10.1109/ACCESS.2020.3039858.

[6] J. Ahmad, M. Jervis, and R. Venkata, "Altera FPGAs and SoCs with FPGA AI Suite and OpenVINO Toolkit drive embedded/edge AI/machine learning applications," *Intel White Paper*, 2021.

[7] J. Si, S. L. Harris, and E. Yfantis, "Neural networks on an FPGA and hardware-friendly activation functions," *Journal of Computer and Communications*, vol. 8, pp. 251–277, Dec. 2020, doi: 10.4236/jcc.2020.812021.

[8] A. Shawahna, S. M. Sait, and A. El-Maleh, "FPGA-based accelerators of deep learning networks for learning and classification: A review," *IEEE Access*, vol. 7, pp. 7823–7859, Jan. 2019, doi: 10.1109/ACCESS.2018.2890150.

[9] T. Simons and D.-J. Lee, "A review of binarized neural networks," *Electronics*, vol. 8, no. 6, p. 661, June 2019, doi: 10.3390/electronics8060661.

[10] M. Xiang and T. H. Teo, "Implementation of Binarized Neural Networks in All-Programmable System-on-Chip Platforms," *Electronics*, vol. 11, no. 4, p. 663, Feb. 2022, doi: 10.3390/electronics11040663.

[11] P. Chen, B. Zhuang, and C. Shen, "FATNN: Fast and Accurate Ternary Neural Networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 5219–5228, 2021.

[12] S. Ahn, J. Lee, and T. Na, "Tiny Machine Learning Hardware Implementation of Handwritten Digit Inference Using Arduino and Ternary Output Binary Neural Network," in *2023 IEEE International Conference on Consumer Electronics-Asia (ICCE-Asia)*, pp. 103–110, Oct. 2023, doi: 10.1109/ICCE-Asia59966.2023.10326336.

[13] R. Kayanoma and H. Nakahara, "Fast Interface with Ensemble Ternary Neural Network," in *2022 IEEE 52nd International Symposium on Multiple-Valued Logic (ISMVL)*, pp. 182–183, May 2022, doi: 10.1109/ISMVL52857.2022.00035.

[14] Q. Ducasse, P. Cotret, L. Lagadec, and R. Stewart, "Benchmarking Quantized Neural Networks on FPGAs with FINN," in *DATE Friday Workshop on System-level Design Methods for Deep Learning on Heterogeneous Architectures*.

[15] S. A. Alam, D. Gregg, G. Gambardella, M. Blott, and T. Preusser, "On the RTL Implementation of FINN Matrix Vector Compute Unit," *arXiv preprint arXiv:2201.11409*, Apr. 2022

[16] J. C. Njuguna, A. T. Çelebi, and A. Çelebi, "Implementation and Optimization of LeNet-5 Model for Handwritten Digits Recognition on FPGAs using Brevitas and FINN," in *Innovations in Intelligent Systems and Applications Conference (ASYU)*, Oct. 2023, pp. 102–108, doi: 10.1109/ASYU58738.2023.1029630.

[17] D. Chawda and B. Senouci, "Fast Prototyping of Quantized Neural Networks on an FPGA Edge Computing Device with Brevitas and FINN," in *15th International Conference on Ubiquitous and Future Networks (ICUFN)*, Jul. 2024, pp. 238–243, doi: 10.1109/ICUFN61752.2024.10625618.

[18] M. Q. Hoang, P. L. Nguyen, H. V. Tran, H. Q. Nguyen, V. T. Nguyen, and C. Vo-Le, "FPGA Oriented Compression of DNN Using Layer-Targeted Weights and Activations Quantization," in *2020 IEEE 8th International Conference on Communications and Electronics (ICCE)*, pp. 31–37, 2020, doi: 10.1109/ICCE48956.2021.9352106.

[19] D. Dai, Y. Zhang, J. Zhang, Z. Hu, Y. Cai, Q. Sun, and Z. Zhang, "Trainable Fixed-Point Quantization for Deep Learning Acceleration on FPGAs," *ACM Transactions on Embedded Computing Systems*, 2024.

[20] Z. Li, J. Chen, L. Wang, B. Cheng, J. Yu, and S. Jiang, "CNN Weight Parameter Quantization Method for FPGA," in *2020 IEEE 6th International Conference on Computer and Communications (ICCC)*, pp. 1548–1552, 2020, doi: 10.1109/ICCC51575.2020.9345248.

[21] L. Deng, P. Jiao, J. Pei, Z. Wu, and G. Li, "GXNOR-Net: Training Deep Neural Networks with Ternary Weights and Activations Without Full-Precision Memory Under a Unified Discretization Framework," *Neural Networks*, vol. 100, pp. 49–58, 2018, doi: 10.1016/j.neunet.2018.01.001.

[22] Larq Documentation. [Online]. Available: https://docs.larq.dev/larq/. [Accessed: Nov. 1, 2024].

[23] M. Blott, T. B. Preußer, N. J. Fraser, G. Gambardella, K. O'Brien, Y. Umuroglu, M. Leeser, and K. Vissers, "FINN-R: An end-to-end deep-learning framework for fast exploration of quantized neural networks," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 11, no. 3, pp. 1–23, 2018. Available: https://github.com/Xilinx/finn/tree/main.

[24] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "FINN: A Framework for Fast, Scalable Binarized Neural Network Inference," in

Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, 2017, pp. 65–74.

[25] R. Zhao, W. Song, W. Zhang, T. Xing, J.-H. Lin, M. Srivastava, R. Gupta, and Z. Zhang, "Accelerating Binarized Convolutional Neural Networks with Software-Programmable FPGAs," in *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA)*, Monterey, CA, USA, Feb. 2017.

[26] *Xilinx Inc.*, "finn-hlslib," GitHub repository, 2019. Available: https://github.com/Xilinx/finn-hlslib.

The University of Sydney

SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

PROJECT CLEARANCE FORM

Unit of Study Code and Name

ELEC4713 Thesis B

This is to certify that my student

Student Name Prosper Su

SID 490447846

Has:

- **Returned all books and reference material;**
- **Returned all equipment and keys; and**
- **Tidied their work place.**

ECE Academic supervisor:

Signature:  **Date:** 1/11/2024

Name: David Boland

External supervisor (if applicable):

Signature: **Date:**

Name: