Design and Implementation of a Quantized CNN Inference Engine on FPGA

Dissertation submitted in partial fulfilment of the requirements
for the award of

Dual Degree(B.Tech and M.Tech)

by

Vennapusa Indrahas Reddy

(Roll No. 19D070067)

Under the Supervision of

Prof. Madhav P. Desai

Department of Electrical Engineering

INDIAN INSTITUTE OF TECHNOLOGY BOMBAY

Mumbai - 400076, India

June, 2024

Dedicated to my beloved parents.

Thesis Approval

This dissertation entitled **Design and Implementation of a Quantized CNN Inference Engine on FPGA** by **Vennapusa Indrahas Reddy**, Roll No. 19D070067, is approved for the degree of **B.Tech and M.Tech** from the Indian Institute of Technology Bombay.

.....	.....
Sachin Patkar	Virendra Singh
(Examiner 1)	(Examiner 2)
.....	.....
Prof. Madhav P. Desai	Virendra Singh
(Supervisor)	(Chairman)

Date: 28 - 06 - 2024

Place: IIT Bombay

Declaration

I declare that this written submission represents my ideas in my own words. Where others' ideas and words have been included, I have adequately cited and referenced the original source. I declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated, or falsified any idea/data/-fact/source in my submission. I understand that any violation of the above will cause disciplinary action by the Institute and can also evoke penal action from the source which has thus not been properly cited or from whom proper permission has not been taken when needed.

. . . . . .

Vennapusa Indrahas Reddy
Roll No.: 19D070067
Date: 28 - 06 - 2024
Place: IIT Bombay

Acknowledgements

I take this opportunity to acknowledge and express my gratitude to all those who supported and guided me during the dissertation work. I am grateful to the Almighty for the abundant grace and blessings that enabled me to complete this dissertation successfully.

I would also like to thank Siddhant Tomar, Rutuja and Jitendra for their collaboration in the development and validation of the inference engine.

Vennapusa Indrahas Reddy

Abstract

The escalating computational cost and memory requirements for training and deploying state-of-the-art neural networks have necessitated explorations into efficient optimization techniques that minimize resource utilization while maintaining acceptable accuracy. Leveraging lower precision data types has been shown to incur negligible accuracy losses, making it a viable strategy. Furthermore, while powerful hardware setups are often required for training, there is a growing need for deployable neural networks that can operate on resource-constrained and low-power architectures, highlighting the potential benefits of reconfigurable architectures over conventional GPUs for specific applications. To address these challenges, we have developed a Quantized CNN inference engine, to store the data in 8 bit integer format, using the PyAHIR and AHIR-V2 high-level synthesis framework, targeting FPGA implementation.

Our design leverages 8-bit data storage for efficient hardware resource utilization and employs Ethernet for data transmission to the hardware. The engine is integrated and validated on an FPGA, with minimal accuracy loss. We demonstrate the efficacy of our design with comprehensive performance metrics, showcasing its capability to handle complex CNN workloads effectively.

The results of our implementation indicate a substantial improvement in computational efficiency and data handling compared to traditional approaches, making it a viable solution for deploying CNNs in resource-constrained environments. Our work sets the stage for further exploration into optimizing FPGA-based accelerators for machine learning tasks.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The widespread adoption of Convolutional Neural Networks (CNNs) in various cognitive applications, including image recognition, natural language processing, object detection, voice recognition, and autonomous driving[5,6], can be attributed to their exceptional accuracy and performance. However, the computational intensity and memory requirements of CNNs pose significant challenges, particularly in resource-constrained environments. Moreover, the recent advancements in CNN accuracy have been achieved through the development of over-parameterized models, which introduce additional complexity and highlight the need for efficient optimization techniques to facilitate their deployment in resource-limited settings.[5,7]

In recent years, the hardware implementation of Convolutional Neural Networks (CNNs) has garnered significant attention and optimization efforts, driven by the need to achieve high accuracy and real-time performance while minimizing power consumption and energy expenditure. The growing trend of implementing CNNs on digital devices, embedded systems, and edge devices, which are characterized by limited resources, has highlighted the challenges associated with deploying CNNs on resource-constrained devices (RCDs)[8,9]. Notably, as the complexity of CNNs increases, the hardware implementation faces significant obstacles, including quadratic

increases in area, energy, and delay, as well as substantial memory traffic, thereby rendering the realization of large-scale CNNs in hardware a formidable challenge that warrants innovative solutions.[10]

Resource-Constrained Devices (RCDs), commonly found in embedded systems and edge devices, are characterized by stringent limitations, including restricted memory capacity (both RAM and ROM) for executing applications and storing data/parameters, humble data processing capabilities, finite battery life, and compact physical dimensions. These constraints render the design and deployment of Convolutional Neural Networks (CNNs) on RCDs exceedingly challenging, necessitating a meticulous trade-off between resource allocation and model accuracy[11]. Moreover, the implementation of larger CNN models on RCDs is often rendered infeasible due to the severe resource constraints, thereby underscoring the need for innovative design methodologies and optimization techniques to enable efficient CNN deployment on resource-constrained platforms.[8,12,13]

The optimization of Convolutional Neural Network (CNN) designs is a complex task, as various layers exhibit distinct challenges that hinder the identification of a single optimal approach for reducing complexity. Specifically, convolutional layers are computationally intensive, characterized by a high volume of mathematical operations, while fully connected (FC) layers are memory-intensive, requiring substantial memory bandwidth. Therefore, a proficient optimization technique should aim to concurrently simplify computational requirements and memory access patterns, addressing the diverse challenges posed by different CNN layers to achieve efficient optimization.[9]

To alleviate the design complexity and facilitate the implementation of Convolutional Neural Networks (CNNs), various optimization techniques have been explored. One

such technique is quantization, which entails the substitution of real-valued numbers with low-precision, fixed-point integers. By reducing the bit width of the parameters, including weights and biases, and the outputs from each layer, namely activations, quantization enables significant improvements in computational speed, reduces power consumption, and minimizes memory requirements, thereby yielding a more efficient and hardware-friendly CNN design.

Extensive research has explored the application of quantization to Convolutional Neural Networks (CNNs), with various studies demonstrating the feasibility of using 32-bit, 16-bit, and 8-bit quantization[9,10,14]. Furthermore, recent investigations have proposed aggressive quantization techniques, which restrict the precision of model parameters to as few as two bits (ternary networks) or even a single bit (binarized networks). However, such aggressive quantization methods can potentially compromise CNN accuracy, necessitating the introduction of compensatory measures, such as increasing the number of neurons, which may inadvertently offset the benefits of quantization, including reduced computational complexity and memory requirements.

1.1 Objectives

This thesis endeavors to develop a quantization method tailored for Field-Programmable Gate Arrays (FPGAs) to optimize the hardware design of Convolutional Neural Networks (CNNs) and to validate the CNN on FPGA. By achieving this objective, this thesis aims to enable the deployment of CNNs on Resource-Constrained Devices (RCDs), such as FPGA devices, and to provide a framework for rapid evaluation of Deep Neural Network (DNN) design choices. The key contributions of this thesis successfully address the aforementioned objective, and are summarized as follows:

1. **Training Quantized CNN:** A novel algorithm has been designed and validated, which facilitates the quantization of the entire Convolutional Neural Network (CNN) model. The algorithm's innovative approach lies in its post-training quantization techniques (PQT), enabling comprehensive model quantization, including both weights and activations, without necessitating an increase in the number of neurons. This approach achieves full model quantization, thereby reducing computational complexity and memory requirements while maintaining accuracy.

2. **Design Efficiency:** Develop a CNN inference engine that maximizes data reuse and minimizes memory bandwidth usage.

3. **Maximizing Accuracy:** Ensure the highest possible accuracy of the CNN model is maintained despite storing intermediate results in 8-bit integer format by employing an effective quantization scheme.

4. **System Integration:** Integrate the engine into a System-on-Chip (SoC) architecture, including an AJIT processor and Network Interface Controller (NIC), to validate real-world performance and application readiness.

5. **Benchmarking:** Compare the designed engine's performance against existing state-of-the-art FPGA implementations to demonstrate competitive advantages.

This dissertation consists of 5 chapters, including the present chapter (Chapter 1) of introduction to the research topic. This chapter describes the need for the study, and objectives of the study, and a brief thesis outline. Chapter 2 presents a comprehensive overview of the LeNet architecture, and delves into the specifics of the quantization scheme employed. Chapter 3 elaborates on the design and implementation of the custom AI/ML accelerator, encompassing the hardware architecture, data

processing pipeline, and memory management. Chapter 4 reports on the experimental evaluation of the hardware accelerator. Chapter 5 summarizes the conclusions and presents further optimizations for future work.

1.2 Acknowledgement of Previous Work

This thesis is an extension of the research conducted by Aman Dhammani [1], who pioneered the development of an AI/ML Accelerator, demonstrating exceptional compute utilization rates of nearly 90%. Nevertheless, the earlier work faced obstacles in quantizing the convolution output to 8-bit fixed-point representation, which hindered the model's accuracy. This investigation aims to overcome this limitation by exploring innovative quantization strategies and output scaling techniques, thereby enhancing the model's performance and precision.

Chapter 2

LeNet Architecture and Quantization

2.1 Introduction

In the domain of computer vision and image processing, Convolutional Neural Networks (CNNs) have emerged as a powerful tool for image classification tasks. One of the pioneering architectures in this field is LeNet, introduced by Yann LeCun and his colleagues in the late 1980s and early 1990s. LeNet was specifically designed to recognize handwritten digits in the MNIST dataset, a benchmark dataset consisting of 60,000 training images and 10,000 testing images of handwritten digits from 0 to 9. Each image in the dataset is of size 28x28 pixels with a single color channel (grayscale).

The LeNet architecture is structured to efficiently handle the spatial hierarchy of features within an image. It consists of two convolutional layers, each followed by a subsampling (pooling) layer, and two fully connected layers leading to an output layer with 10 neurons, each corresponding to one of the digit classes. The design of LeNet allows it to automatically learn hierarchical features from raw pixel values,

significantly improving the accuracy of digit recognition.

Architecture Overview:

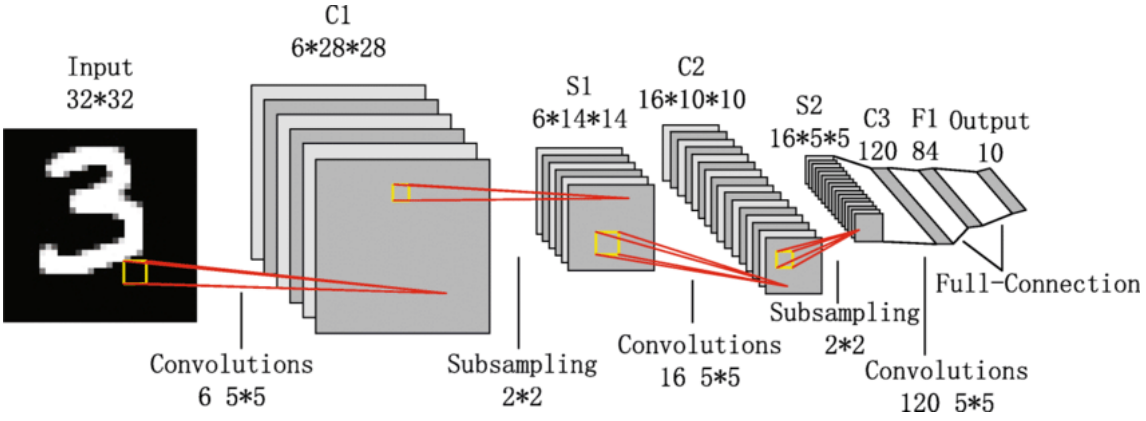

FIGURE 2.1: LeNet Architecture

1. **Input Layer:** The input to the network is a 32x32x1 gray scale image.

2. **Convolutional Layer 1:** This layer consists of six 5x5 filters (kernels), producing six feature maps of size 28x28.

3. **Subsampling (Pooling) Layer 1:** A 2x2 max pooling operation reduces the size of each feature map to 14x14.

4. **Convolutional Layer 2:** This layer applies sixteen 5x5 filters to the pooled output, resulting in sixteen 8x8 feature maps.

5. **Subsampling (Pooling) Layer 2:** Another 2x2 max pooling operation further reduces the feature maps to 5x5.

6. **Fully Connected Layer 1:** The flattened output from the second pooling layer (16 x 5 x 5 = 400 units) is fed into a fully connected layer with 120 units.

7. **Fully Connected Layer 2:** This layer consists of 84 units, further refining the learned features.

8. **Output Layer:** Finally, the network outputs a probability distribution over the 10 digit classes using a soft max activation function.

LeNet's simple yet effective architecture has laid the foundation for more complex CNNs and has been instrumental in advancing the field of deep learning. Its ability to achieve high accuracy on the MNIST dataset with relatively few parameters makes it an ideal starting point for exploring CNNs in the context of handwritten digit recognition.

2.2 Convolution Operation

The convolution operation is a mathematical process used to extract features from an input image. It involves sliding a filter (also known as a kernel) across the input image and computing the dot product between the filter and a region of the image. The result of this operation is a feature map that highlights various aspects of the input image, such as edges, textures, and patterns.
Mathematically, the convolution operation for a single output pixel can be expressed as:

$$(I * K)(i, j) = \sum_{m} \sum_{n} I(i + m, j + n) \cdot K(m, n) \tag{2.1}$$

where:

- I is the input image,
- K is the kernel,
- i, j are the coordinates of the output feature map,

- m, n are the coordinates within the kernel.

The pseudocode for the convolution operation:

```
function Convolution2D(input, kernel):
    input_height, input_width = dimensions of input
    kernel_height, kernel_width = dimensions of kernel
    output_height = input_height - kernel_height + 1
    output_width = input_width - kernel_width + 1

    # initialize output as zeros with dimensions (output_height,
    output_width)

    for i = 0 to output_height - 1:
        for j = 0 to output_width - 1:
            sum = 0
            for m = 0 to kernel_height - 1:
                for n = 0 to kernel_width - 1:
                    sum += input[i+m][j+n] * kernel[m][n]
            output[i][j] = sum

    return output
```

Listing 2.1: Pseudocode for 2D Convolution Operation

2.3 Pooling Operation

Pooling is a down-sampling operation that reduces the spatial dimensions of the feature map, thereby reducing the number of parameters and computation in the

network. It also helps in making the detection of features invariant to small translations.

The max pooling operation can be expressed as:

$$Y(i,j) = \max_{0 \le m < p} \max_{0 \le n < q} X(i \cdot s + m, j \cdot s + n) \tag{2.2}$$

where:

- X is the input feature map,
- Y is the output feature map after max pooling,
- i, j are the coordinates of the output feature map,
- p, q are the height and width of the pooling window,
- s is the stride of the pooling operation,
- m, n are the coordinates within the pooling window.

2.4 Non Linear Activation

Non-linear activation functions introduce non-linearity into the neural network, allowing it to learn complex patterns in the data. Without non-linear activation functions, the network would behave like a linear model, regardless of the number of layers, limiting its ability to solve complex tasks. Here are some commonly used non-linear activation functions

1. **Sigmoid Function:** The sigmoid activation function maps the input values to a range between 0 and 1. It is often used in the output layer for binary

classification problems.

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.3}$$

2. **Rectified Linear Unit (ReLU) Function:** The ReLU activation function is widely used in deep learning models due to its simplicity and effectiveness. It maps the input to zero if it is negative and leaves it unchanged if it is positive.

$$\text{ReLU}(x) = \max(0, x) \tag{2.4}$$

3. **Leaky ReLU Function:** Leaky ReLU is a variant of ReLU that allows a small, non-zero gradient when the input is negative. This helps to avoid the "dying ReLU" problem where neurons get stuck during training.

$$\text{Leaky ReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{if } x \leq 0 \end{cases} \tag{2.5}$$

2.5 Model Training using Posit Data Type

2.5.1 Posit number system

The Posit number system, a Type-III universal number representation (Unum), is characterized by two primary parameters: the word size (n) and the exponent field size (es). Unlike conventional floating-point numbers (float), Posit features a distinct regime field encoding scheme based on the run-length method, which enables a more optimal trade-off between dynamic range and numerical precision. This unique encoding approach allows Posit to exhibit improved performance in terms of both range and precision, making it a promising alternative to traditional floating-point

representations.

As illustrated in Figure 2.2, a posit number, denoted as (n, es), comprises four

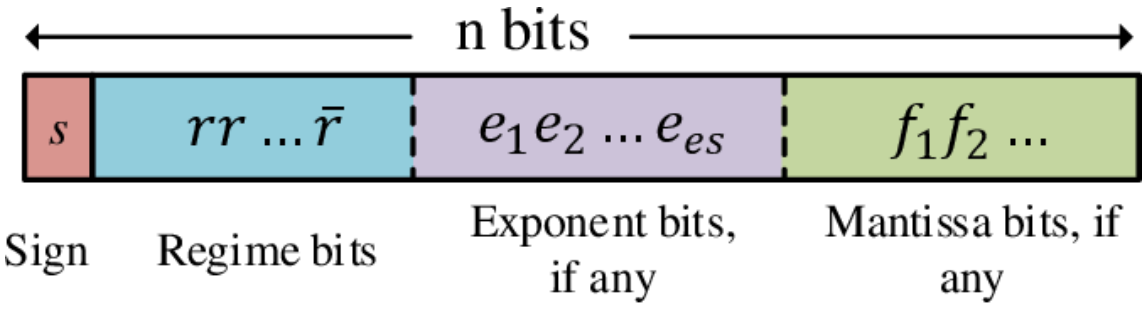

FIGURE 2.2: General posit format

distinct components: a **sign** bit, **regime** field, **exponent** field, and **mantissa** field. The regime field value is decoded by interpreting consecutive sequences of '1's followed by a '0' as k-1 and consecutive sequences of '0's followed by a '1' as -k, where k represents the number of consecutive bits. The value of posit number p(binary code) is given by:

$$\begin{cases} 0, & p = 0, \\ \pm\infty, & p = -2^{n-1}, \\ \text{sign}(p) \times useed^k \times 2^e \times f, & \text{all other } p. \end{cases} \tag{2.6}$$

2.5.2 Models Trained using Posit Data Type

To investigate the impact of employing the Posit data type for storing the output of convolutional layers, a modification was introduced in the training phase of the LeNet architecture, wherein the output of each convolutional stage was stored in the Posit format. This allowed for an exploration of the effects of Posit-based storage on the accuracy of the convolutional neural network (CNN). The resulting accuracy metrics, presented in the following table, provide insight into the efficacy of utilizing Posit data type in this context.

Sr.No	Model	Dataset	Datatype for computation	Datatype for storage	Training Accuracy	Testing Accuracy
1	LeNet-5	MNIST Handwritten Digit	32 bit float	32 bit float	98.6%	98.27%
2			16 bit float	16 bit float	97.93%	97.23%
3			32 bit float	8 bit posit	88%	89%
4			16 bit float	8 bit posit	88.7%	89.37%

TABLE 2.1: Impact of Posit Storage on the Accuracy of LeNet Model

The incorporation of Posit-based storage resulted in an accuracy degradation of 8-9% relative to the baseline architecture, which was found to be more pronounced than expected . Consequently, a comprehensive examination of alternative approaches led to the selection of the Post-Training Quantization (PTQ) method, as implemented in the PyTorch framework, to optimize the quantization process and minimize the accuracy loss.

2.6 Post Training Static Quantization

Post Training Static Quantization (PTQ) is a technique used in PyTorch to optimize the inference performance of neural networks by converting the model's weights and activations from floating-point precision (usually 32-bit) to integer precision (usually 8-bit). This conversion results in faster computation and reduced memory footprint, making it particularly useful for deployment on edge devices and resource-constrained environments.

PyTorch's PTQ workflow involves the following key steps:

1. **Preparation:** Modify the model to support quantization by inserting quantization and dequantization nodes.

2. **Calibration:** Collect statistics on the distributions of weights and activations by running inference with representative calibration data. The prepared model is run on a set of representative data to collect the necessary statistics (e.g., min and max values) for each layer's activations and weights. This step is crucial for determining the scaling factors and zero points used in quantization.

3. **Conversion:** Convert the floating-point model to a quantized model using the collected statistics.After calibration, the model is converted to its quantized form. This step replaces the floating-point weights and activations with their quantized counterparts based on the collected statistics.

Here is a complete example of applying PTQ to a PyTorch model:

```
import torch
import torch.quantization

# Define or load your model
model = MyModel()

# Set the model to evaluation mode
model.eval()

# Specify the quantization configuration
model.qconfig = torch.ao.quantization.default_qconfig

# Prepare the model for static quantization
torch.quantization.prepare(model, inplace=True)

# Calibrate the model with representative data
calibration_data = [...]  # Your calibration dataset
with torch.no_grad():
```

```
    for input in calibration_data:
        model(input)

# Convert the model to a quantized version
torch.quantization.convert(model, inplace=True)

# The model is now quantized and ready for inference
```

Listing 2.2: PyTorch Post Training Static Quantization

The resulting accuracy metrics, yielded by the employment of PTQ techniques for the quantization and precision scaling of the trained model, are presented in the following table

Sr.No	**Model**	**Dataset**	**Datatype for computation**	**Datatype for storage**	**Training Accuracy**	**Testing Accuracy**
1	LeNet-5	MNIST Handwritten Digit	32 bit float	32 bit float	98.6%	98.27%
2			32 bit float	8 bit UINT	-	98.01%

Table 2.2: Impact of PTQ technique on the Accuracy of LeNet Model

An examination of the table reveals that the testing accuracy remains relatively invariant, exhibiting no significant degradation, thereby indicating the efficacy of the employed quantization method in maintaining the model's performance. Consequently, this quantization technique was selected for implementation on the hardware platform, as it demonstrates a satisfactory tradeoff between precision and computational efficiency.

Chapter 3

Design and Implementation of Inference Engine

3.1 Introduction

A typical inference engine should perform operations like convolution, max pooling, non linear activation and padding. Apart from this, there are other modules which are required for accessing data from memory. They are in charge of retrieving the inputs and kernels from memory. They are assisted by the readModules, which are linked via a deadlock prevention system. The retrieved data is sent through pipes to the compute convolution module, where the main computations take place. The partial sums are then gathered in the accumulator. Subsequently, the outputs are produced by quantizing and applying activation functions before being written back to memory.

The engine's primary requirement is to perform all the aforementioned operations

with high throughput. To achieve that, the critical modules of the accelerator should run at full rate.

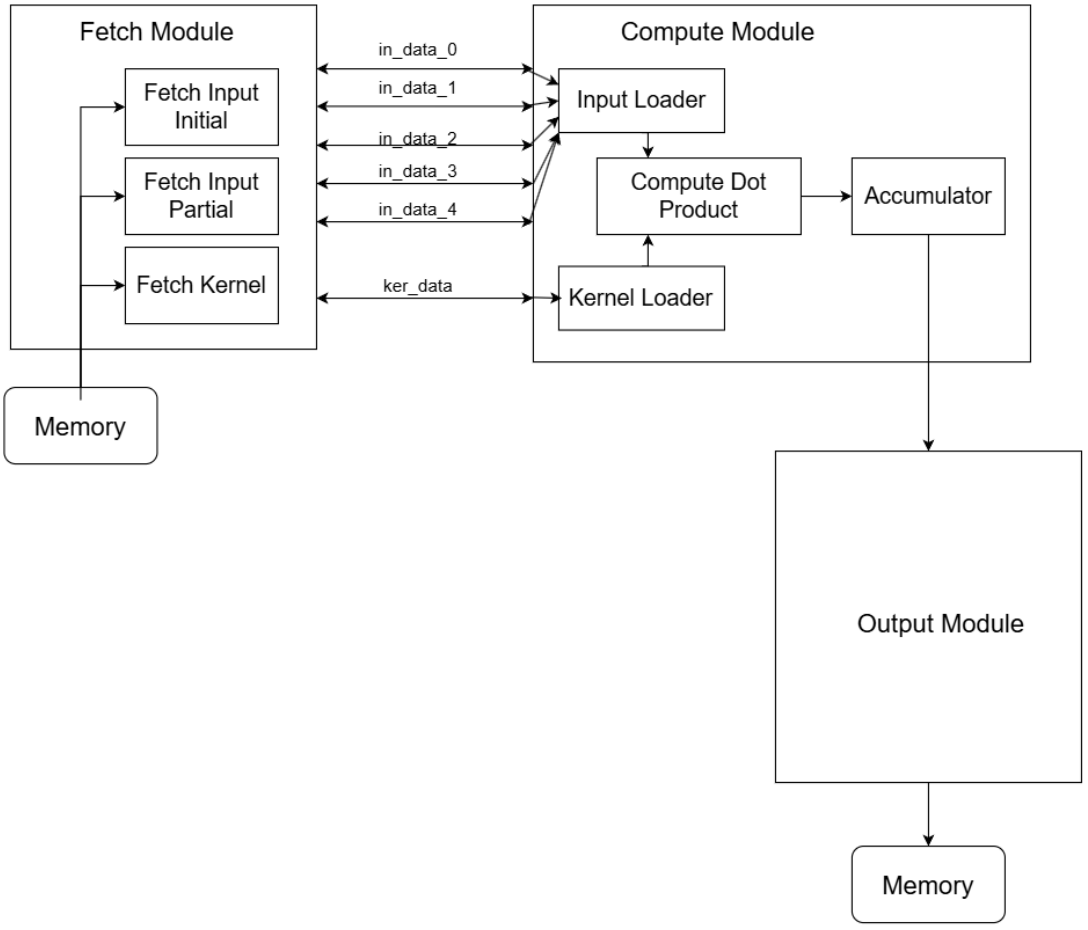

FIGURE 3.1: Block diagram for the engine implementation

3.2 Fetch Module

This module ensures a continuous data stream to the compute module. A series of submodules were designed to optimize input delivery rates, enabling the compute module to operate at maximum throughput without data stalls during each computation stage.

3.2.1 Fetch Input Initial

Prior to initiating the Compute Module, an initial batch of (k_r x k_c x chn)(k_r and k_c are kernel row & columns and chn is the number of input channels) elements is pre-fetched from memory and stored in the internal pipes, denoted as in_data_<i>, where i represents the column index of the image. These pipes are subsequently read by the Input Data Loader module to provide a sustained and high-throughput supply of data to the multiplier unit within the Compute Dot Product Module. To minimize memory access and maximize data reuse, a data rewriting mechanism is employed, wherein the required data elements are re-written into the in_data_<i> pipes, ensuring optimal data locality and reducing memory access latency.

3.2.2 Fetch Kernel

Analogous to the fetch input initial phase, the fetch kernel module preloads the (k_r x k_c x chn) elements in ker_data pipe, prior to initializing the Compute Module. Upon completion of both the fetch input initial and fetch kernel sub-modules, the Compute Module is activated, in conjunction with the fetch input partial sub-module, to stream the remaining input data to the processing engine, ensuring a continuous and high-throughput data flow.

3.2.3 Fetch Input Partial

Due to the Compute Module already possessing the requisite (k_r x k_c x chn) elements for processing the current output, a data reuse strategy is employed to maximize computational efficiency. Specifically, (k_r) x (k_c - 1) x chn) elements are reused as the kernel strides forward, thereby reducing the data fetch requirement.

Consequently, prior to initiating the computation of the subsequent output, only (k_r x chn) elements need to be fetched and stored in the pipes, ensuring that the Compute Module operates at its peak throughput without experiencing any data starvation or wait states during each computation stage.

3.2.4 De-Quantization

Since the data retrieved from memory is represented in an 8-bit unsigned integer format (uint8), a format conversion is necessitated to transform the data into single-precision floating-point numbers (FP32) to facilitate the computation stage. This format conversion is achieved through the application of the De-Quantization formula, as specified by the PyTorch Framework, which involves a scaling operation that multiplies the quantized integer values by a scalar factor, denoted as the scale, and a subsequent addition of a zero-point offset, thereby effectively converting the quantized data to its original floating-point representation.

$$\text{Inp}_{\text{de-quantized}} = \text{scale\_factor} \times (\text{Inp}_{\text{quantized}} - \text{zero\_point})$$

where zero_point is the zero-point offset and scale_factor is the scaling factor.

3.3 Compute Module

Given that the Compute Module serves as the engine's core processing unit, its performance has a direct impact on the engine's overall throughput. The submodules embedded within this module are carefully designed to ensure that the Compute Module maintains an optimal execution rate, thereby enabling the engine to operate at its maximum processing capacity and achieve peak performance.

3.3.1 Compute Dot Product

The Dot Product Submodule, a crucial component of the Compute Module, incorporates a single 32-bit floating-point multiplier, which facilitates the computation of the dot product between the input data and kernel coefficients. This submodule retrieves the requisite data from the Input Loader and Kernel Loader modules, and subsequently performs the dot product operation, yielding a result that is will be accumulated. The number of dot product operations is predetermined, and is calculated using the formula (k_r * k_c * chn) * (o_r * o_c), (o_r and o_c are number of output rows and columns).Accordingly, the Dot Product Submodule expects to receive a specific number of elements from the Input Loader and Kernel Loader modules, which is necessitated for the efficient computation of the dot product.

3.3.2 Compute Accumulator

The Accumulator Module, a critical component of the Compute Engine, expects to receive a predetermined number of elements from the Dot Product Submodule, specifically (k_r * k_c * chn) * (o_r * o_c) elements. Additionally, the Accumulator Module is designed to intelligently manage the accumulation process, recognizing when to forward the accumulated value to the Output Module, which occurs after the summation of (k_r * k_c * chn) dot products between input and kernel coefficients. This triggers the accumulator reset, initiating the accumulation process anew for the subsequent output pixel, thereby ensuring efficient and accurate processing of output data.

3.3.3 Input and Kernel Loader

The Input Data Loader and Kernel Data Loader submodules, integral components of the Compute Engine, are responsible for reading data from the internal pipelines, specifically the in_data and ker_data pipes, which are populated by the Fetch Module. These submodules ensure that the data is supplied to the Dot Product Submodule in a precise and sequential manner, thereby facilitating efficient computation. Moreover, they implement a data reutilization strategy, wherein the accessed data is rewritten back to the internal pipelines in a specific order, minimizing frequent memory access and reducing the associated latency, thus optimizing the overall performance of the Compute Engine.

3.4 Output Module

The output modules receive data from the accumulator, perform necessary operations on them, and subsequently store the results back into memory.

3.4.1 Non Linear Activation

Upon receiving of the data, the module applies the Rectified Linear Unit (ReLU) activation function, contingent upon the activation of the corresponding non-linearity flag. The operation is executed on the accumulated outputs prior to quantization and then written back to the memory.

3.4.2 Pooling

The Output Modules additionally perform max pooling operations subsequent to convolution, implementing a 2 x 2 max pooling kernel. In order to facilitate this operation, two rows of output data are buffered and temporarily stored, followed by the execution of the max pooling algorithm, which selects the maximum values from the pooled regions. The resultant output data is then written back to memory, thereby reducing spatial dimensions and retaining essential features.

3.4.3 Quantization

Following the completion of the non-linear activation, the 32-bit floating-point numbers are subsequently quantized to 8-bit unsigned integers (uint8) using the quantization formula prescribed by the PyTorch Framework, which entails a scaling operation that multiplies the floating-point values by a scalar factor, followed by the addition of a zero-point offset, and culminates in a rounding operation to the nearest integer, thereby effecting a precise conversion from floating-point to integer representation.

$$\mathrm{Out}_{\mathrm{quantized}} = \mathrm{round}\left(\frac{\mathrm{Out}_{\mathrm{float}}}{\mathrm{scale\_factor}} + \mathrm{zero\_point}\right)$$

where zero_point is the zero-point offset and scale_factor is the scaling factor.

3.5 Interface to Access The Accelerator

Two continuously operating daemons manage the engine. The first is the accelerator_control_daemon, responsible for initializing registers, launching the worker

daemon, and receiving configuration details from the processor or calling module via the AFB interface. Using this information, it configures the registers accordingly.

The second daemon is the accelerator_worker_daemon, which monitors the control register R0. It continuously checks until bits 0 and 2 are set by the calling module. Upon meeting this condition, it reads the remaining registers and triggers the convolution engine through a command call.

The convolution engine function call is represented as:

```
$call convengine (
    in_start_addr out_start_addr ker_start_addr out_grp_no in_rows
    in_cols in_channels out_channels groups ker_size pool_cols inp_scale
    inp_zero_point ker_scale ker_zero_point conv_scale conv_zero_point
    padReq poolReq isLinear isActivation isFlatten flattenOffset out_chn_ind
    ) ()
```

where:

- **in_start_addr**: Starting address of the input data.
- **out_start_addr**: Starting address where the output data will be stored.
- **ker_start_addr**: Starting address of the kernel data.
- **out_grp_no**: Output group number.
- **in_rows**: Number of rows in the input data.
- **in_cols**: Number of columns in the input data.
- **in_channels**: Number of input channels.
- **out_channels**: Number of output channels.

- **groups**: Number of groups in the convolution.
- **ker_size**: Size of the kernel.
- **pool_cols**: Pooling columns.
- **inp_scale**: Input scaling factor.
- **inp_zero_point**: Input zero point.
- **ker_scale**: Kernel scaling factor.
- **ker_zero_point**: Kernel zero point.
- **conv_scale**: Convolution scaling factor.
- **conv_zero_point**: Convolution zero point.
- **padReq**: Padding requirement.
- **poolReq**: Pooling requirement.
- **isLinear**: Flag indicating if the operation is linear.
- **isActivation**: Flag indicating if activation is applied.
- **isFlatten**: Flag indicating if flattening is performed.
- **flattenOffset**: Offset used for flattening.
- **out_chn_ind**: Output channel index.

3.5.1 Register File Format

The accelerator engine utilizes 16 registers of 32-bit each for communication with external interfaces, facilitating the exchange of specific information as follows:

- Register 0: Functions as the controller for the accelerator and monitors status flags.
- Register 1: Stores the base address for the input tensor
- Register 2: Stores the base address for the output tensor
- Register 3: Stores the base address for the kernel tensor
- Register 4: The index of the current output group
- Register 5: Stores the value of the number of input rows (31 downto 16) and number of input columns (15 downto 0)
- Register 6: Stores the value of the number of input channels (31 downto 16) and number of input groups (15 downto 0)
- Register 7: Stores the kernel size
- Register 8: Stores the number of columns present in the pooling layer
- Register 9: Stores the base address for scale and zero point values required for the quantization functions
- Register 10: Stores the control for padding (bit 4), pooling (bit 3), linear convolution (bit 2), activation (bit 1) and flatten (bit 0)
- Register 11: Stores the value of the number of output channels (31 downto 16) and current index of the output (15 downto 0)
- Register 12: Stores the offset address required during flattening of the output tensor

Registers 13-15 are currently unused but can be repurposed for additional parameters or for debugging purposes.

Register 0 has the following bits used:

- Bit 0: "accl enable" controls the activation of the engine and can be toggled externally.
- Bit 1: "interrupt enable" sets or clears the interrupt flag, which can be controlled by invoking the module.
- Bit 2: The "start cmd" is triggered by invoking the module, directing the accelerator to commence execution.
- Bit 3: "cmd complete" is set by the accelerator to signify the completion of execution, and it is reset by the processor when preparing to signal the next command.
- Bit 4: "accl done" is activated by the accelerator to indicate the completion of execution and prompt the generation of an interrupt.

3.6 Resource Utilitzation

Reference	**Quantization Method**	**Clock**	**LUTs**	**DSP**	**FF**	**Accuracy**
[2]Mohd, Bassam et.al	Quantization Aware Training (QAT)	200 MHz	50k	-	-	96.67%
[3] Blott, Michaela et.al	FINN	250 MHz	67k	0	-	98.8%
[4] Ji, Mengfei et.al	FPQNet	250 MHz	53k	2614	94k	98.8%
This Work	FINN	125 MHz	67k	183	74k	98%

Table 3.1: Resource utilization and comparison with other work

Chapter 4

Hardware Testing and Results

4.1 Introduction

We assess and define the performance of the accelerator by incorporating it into a System-on-Chip (SoC) that utilizes the AJIT Processor. This processor issues commands to the accelerator and retrieves the data. Additionally, the entire system is linked to a Network Interface Controller (NIC), which ensures high-speed input/output for rapid image loading into memory. The system's architecture is depicted in Figure 4.1.

The system features a 32-bit wide AJIT processor at its core. This processor uses an ACB interface to interact with memory and other modules. The AFB interface, connected to the Network Interface Controller (NIC) and the accelerator engine, is utilized to transfer configuration data from the processor and relay status flags back to it. The modules also have an ACB interface for memory access. The memory subsystem includes a DRAM controller linked to an ACB to UI conversion protocol, which translates memory requests from the ACB bus to DRAM requests and vice versa for responses. Additionally, the processor is equipped with a couple of

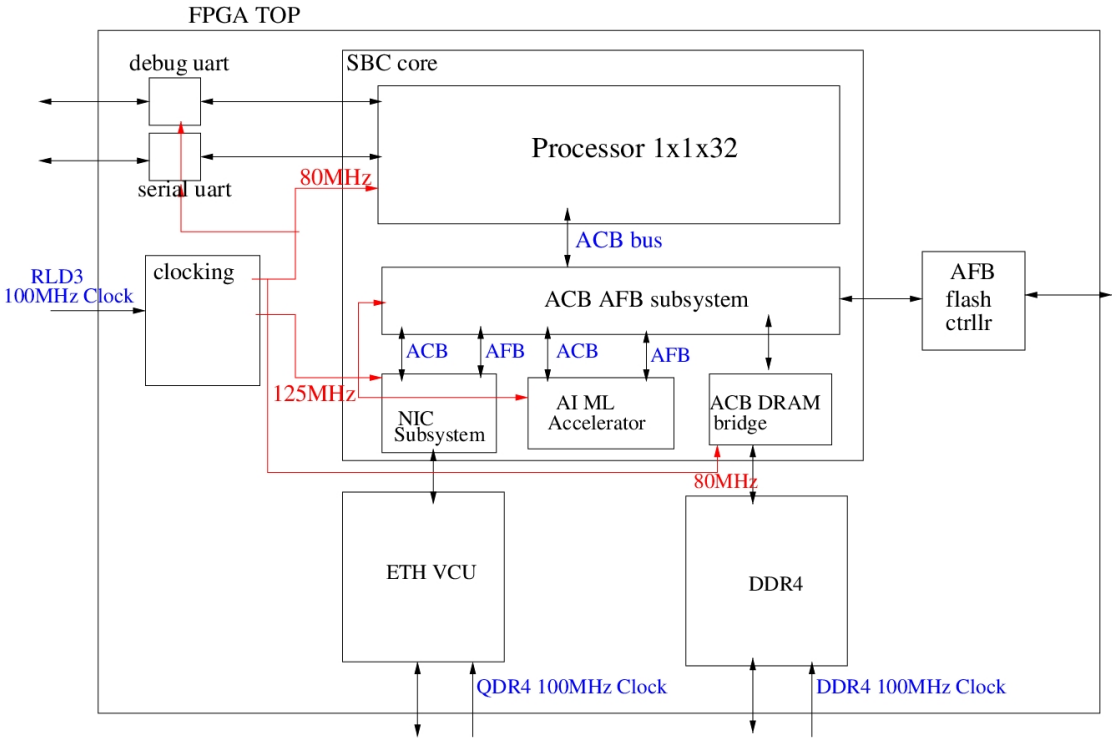

FIGURE 4.1: Block Diagram for SoC

UART interfaces for external communication. The NIC also has an interface to connect with the Xilinx MAC IP, enabling communication with the host machine via Ethernet, currently configured to operate at 100Mbps.

4.1.1 Processor Code

Below algorithm details the functioning of the processor code. It begins by initializing the memory spaces required by all modules, including the NIC queues, the kernel addresses for the accelerators, and the memory spaces for receiving and storing tensors during engine computation. Once the memory spaces are initialized, all NIC queues are configured and initialized, and the Network Interface Controller starts operating. The kernels are then fetched via Ethernet, followed by the input tensors . With the setup complete, the NIC is turned off, and the engine process the data

stage by stage. After processing all the stages, the output data is sent to the host PC via Ethernet, where the results are verified for accuracy.

Algorithm 1 Pseudocode for processor

```
Initialize memory spaces for all modules:
    • NIC queues
    • Kernel addresses for accelerator
    • Memory spaces for receiving and storing tensors
Configure and initialize all NIC queues
Start Network Interface Controller
Fetch kernels via Ethernet
while 1 do
    Fetch input tensor via Ethernet
    Process the input
    Send output data to host PC via Ethernet
end while
```

4.2 NIC Interface

The processor initializes the NIC queues and activates the NIC. Upon receiving data from the MAC, the NIC retrieves an address from the free queue, which contains addresses of empty packet buffers, and writes the packet to that buffer. After successfully writing the packet, the NIC places the buffer's address into the RX queue. An address in the RX queue signifies that a packet has been received and requires processor action. During this process, the processor continuously polls the RX queue. Upon detecting data, it reads the packet and makes a decision. The processor then stores the packet in a shared memory location accessible to the accelerator through its registers. Once the packet storage is complete, the processor places the address into the TX queue, serving as an acknowledgment for the host. Upon receiving this acknowledgment, the host sends a new packet. This describes the overall process for storing files in memory.

To transmit a file via the Ethernet interface, the processor first breaks the file into multiple packets. It then retrieves an empty buffer address from the free queue and stores each packet at the corresponding address. Once the packet is successfully stored, the processor pushes the buffer address into the TX queue. The NIC's transmit engine, which continuously polls the TX queue, reads the buffer and sends the packet out. This process is repeated until all packets are transmitted.

4.3 Accelerator Interface

The processor begins by loading data into registers 1 to 12. It then sets bits 0 and 2 of register 0, signaling the accelerator to commence operation. The processor can either proceed with other tasks or poll register 0, awaiting the completion signal indicated by bit 3 of register 0 being set high. Once the accelerator completes its task, the processor updates the registers with the parameters for the next stage or image.

Additionally, the accelerator is managed by a control daemon that resets the registers, reads data from the AFB ACCELERATOR REQUEST, writes this data to the accelerator registers, and sends back a response to the AFB ACCELERATOR RESPONSE. This control daemon runs continuously, waiting for requests from the processor on the AFB ACCELERATOR REQUEST.

The accelerator also includes a worker daemon that monitors the r0 register for specific bits to be set. When the processor issues an execution request through the registers, the worker daemon invokes the core function with the parameters stored by the processor in other registers. Upon completing the execution, it sets bit 3 of r0 back to 0, signaling to the processor that the computation is finished.

4.4 Results

Sr.No	**Platform**	**Datatype for storage**	**Accuracy**
1	CPU	32 bit float	98.27%
2	CPU	UINT8	98%
3	FPGA	UINT8	98%

Table 4.1: Comparison of LeNet Architecture on Different Platforms

Chapter 5

Summary

In this work, we present a CNN inference engine on FPGA which can produce results with higher accuracy despite storing those results in 8 bit integer format. This engine is designed using AHIR-V2 and PyAHIR tools. We demonstrate an application of the engine by developing image classification algorithm using LeNet architecture, a well known ML model for image classification.

We also integrated the engine with an AJIT processor and Network Interface Controller (NIC) to generate a system-on-chip (SoC) capable of performing at-edge AI/ML inference tasks. Using the SoC, we established the correctness of the acceleration engine using a test bench to validate the output sent to the host machine after every stage.

Our analysis of the engine's characteristics reveals opportunities for enhancing performance and robustness. These potential optimizations are outlined as future work here:

- To increase the parallel computations in the hardware, multiple copies of the engines can be deployed, where each engine will work with separate filter, but

on same input. This will help in obtaining the output of multiple channels simultaneously, thus decreasing the overall inference time for the output while maintaining same hardware utilization.

- By supporting half precision arithmetic, a significant throughput enhancement can be achieved, stemming from the ability to perform twice the number of operations, compared to the current hardware implementation that exclusively employs single precision operators, thereby leveraging the reduced numerical precision requirements to double operational bandwidth and accelerate computational throughput.

- Through hardware design optimization, the engine's hardware utilization can be maximized, facilitating the extension of its capabilities to accommodate the execution of sophisticated CNN operations, such as Batch Normalization and depthwise convolution, in addition to standard convolution operations, thereby enhancing the engine's versatility and computational range.

References

[1] Aman Dhammani, "An AI/ML Inference Engine Developed Using AHIR-V2 Tools", [Unpublished Dual Degree Thesis, IIT Bombay].

[2]Mohd, Bassam & Ahmad Yousef, Khalil & Almajali, Anas & Hayajneh, Thaier. (2024). Quantization-Based Optimization Algorithm for Hardware Implementation of Convolution Neural Networks. Electronics. 13. 1-25. 10.3390/electronics13091727.

[3] Blott, Michaela & Preußer, Thomas & Fraser, Nicholas & Gambardella, Giulio & O'Brien, Kenneth & Umuroglu, Yaman & Leeser, Miriam & Vissers, Kees. (2018). FINN- R: An End-to-End Deep-Learning Framework for Fast Exploration of Quantized Neural Networks. ACM Transactions on Reconfigurable Technology and Systems. 11. 1-23. 10.1145/3242897.

[4] Ji, Mengfei & Al-Ars, Zaid & Hofstee, H.P. & Chang, Yuchun & Zhang, Baolin. (2023). FPQNet: Fully Pipelined and Quantized CNN for Ultra-Low Latency Image Classification on FPGAs Using OpenCAPI. Electronics. 12. 4085. 10.3390/electronics12194085.

[5] Zhang, W. A Survey of Field Programmable Gate Array-Based Convolutional Neural Network Accelerators. Int. J. Electron.Commun. Eng. 2020, 14, 419–427.

[6] Mittal, S. A survey of FPGA-based accelerators for convolutional neural networks. Neural Comput. Appl. 2020, 32, 1109–1139.

[7] Gholami, A.; Kim, S.; Dong, Z.; Yao, Z.; Mahoney, M.W.; Keutzer, K. A survey of quantization methods for efficient neuralnetwork inference. arXiv 2021, arXiv:2103.13630

[8] Kim, Y.D.; Park, E.; Yoo, S.; Choi, T.; Yang, L.; Shin, D. Compression of deep convolutional neural networks for fast and lowpower mobile applications. arXiv 2015, arXiv:1511.06530.

[9] Qiu, J.; Wang, J.; Yao, S.; Guo, K.; Li, B.; Zhou, E.; Yu, J.; Tang, T.; Xu, N.; Song, S.; et al. Going deeper with embedded fpga platformfor convolutional neural network. In Proceedings of the 2016 ACM/SIGDA International Symposium on Field-ProgrammableGate Arrays, Monterey, CA, USA, 21–23 February 2016; pp. 26–35.

[10] Chen, T.; Du, Z.; Sun, N.; Wang, J.; Wu, C.; Chen, Y.; Temam, O. Diannao: A small-footprint high-throughput accelerator forubiquitous machine-learning. ACM SIGARCH Comput. Archit. News 2014, 42, 269–284.

[11] Mohd, B.J.; Hayajneh, T.; Vasilakos, A.V. A survey on lightweight block ciphers for low-resource devices: Comparative study andopen issues. J. Netw. Comput. Appl. 2015, 58, 73–93.

[12] Zhuang, B.; Shen, C.; Tan, M.; Liu, L.; Reid, I. Towards effective low-bitwidth convolutional neural networks. In Proceedings ofthe IEEE Conference on Computer Vision and Pattern Recognition, Salt Lake City, UT, USA, 18–22 June 2018; pp. 7920–7928.

[13] Zhu, C.; Han, S.; Mao, H.; Dally, W.J. Trained ternary quantization. arXiv 2016, arXiv:1612.01064

[14] Miyashita, D.; Lee, E.H.; Murmann, B. Convolutional neural networks using logarithmic data representation. arXiv 2016,arXiv:1603.01025.