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.

.....	.....
Prof. Name	Prof. Name
(Examiner 1)	(Examiner 2)
.....	.....
Prof. Madhav P. Desai	Prof. Name
(Supervisor)	(Chairman)

Date: .....

Place: .....

Certificate

This is to certify that the dissertation entitled **"Design and Implementation of a Quantized CNN Inference Engine on FPGA"**, submitted by **Vennapusa Indrahas Reddy** to the Indian Institute of Technology Bombay, for the award of the degree of **B.Tech + M.Tech** in Electrical Engineering, is a record of the original, bona fide research work carried out by him under our supervision and guidance. The dissertation has reached the standards fulfilling the requirements of the regulations related to the award of the degree.

The results contained in this dissertation have not been submitted in part or in full to any other University or Institute for the award of any degree or diploma to the best of our knowledge.

.....

Prof. Madhav P. Desai
Department of Electrical Engineering,
Indian Institute of Technology 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:
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

Convolutional Neural Networks (CNNs) play a critical role in contemporary image processing tasks. The increasing depth and complexity of these networks necessitate significant computational resources, particularly for operations such as convolution, pooling, and activation. To address these challenges, we have developed a high-performance CNN inference engine using the 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. This setup facilitates the deployment of our inference engine in various embedded and real-time applications requiring robust image processing capabilities. The engine is integrated and validated on an FPGA, achieving significant computational throughput while maintaining a low power footprint. 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

Machine Learning (ML) has emerged as a pivotal area within the realm of Artificial Intelligence (AI), primarily due to its ability to design and deploy algorithms capable of learning and making predictions based on data. Convolutional Neural Networks (CNNs), a subset of ML algorithms, have shown remarkable efficacy in a range of applications, notably in image processing. CNNs excel at capturing spatial hierarchies in images through their layered architecture, which involves convolutions, pooling, and various forms of activation functions.

As CNNs evolve, their depth and complexity increase, leading to larger intermediate feature maps and a substantial rise in computational requirements. Each image processed by a CNN necessitates billions of Multiply-Accumulate (MAC) operations, which places significant demands on computational resources. The limited memory bandwidth and small on-chip buffer sizes further exacerbate these challenges, highlighting the need for specialized hardware accelerators designed for efficient data reuse and high throughput.

Field-Programmable Gate Arrays (FPGAs) offer a compelling solution for accelerating CNN inference due to their reconfigurability, parallel processing capabilities, and lower power consumption compared to traditional processors such as CPUs and

GPUs. FPGAs can be tailored to specific workloads, enabling optimized performance for ML tasks. However, designing efficient FPGA-based accelerators requires a robust toolchain and a detailed understanding of both hardware and algorithmic intricacies.

In this project, we present the design and implementation of a CNN inference engine on FPGA using the AHIR-V2 toolchain. AHIR-V2 facilitates high-level synthesis from algorithmic descriptions, allowing for a streamlined design process and efficient hardware generation. Our engine utilizes 8-bit data storage to optimize hardware resource usage and supports data transmission via Ethernet, making it suitable for embedded and real-time applications.

We validate our design on a commercially available FPGA, VCU128, demonstrating significant computational throughput and efficiency. The engine achieves high performance metrics, showcasing its capability to process complex CNN workloads while maintaining low power consumption. This project not only highlights the potential of FPGA-based accelerators in ML applications but also sets a foundation for further enhancements in hardware-software co-design for AI inference engines.

1.1 Problem Statement

In the field of machine learning, specifically in image processing, the computational requirements of Convolutional Neural Networks (CNNs) have grown significantly due to increased network depth and larger feature maps. These requirements, often amounting to billions of Multiply and Accumulate (MAC) operations per image, pose a challenge for traditional CPUs, which are limited in their ability to handle such parallel data operations efficiently. This issue is compounded by constraints

on on-chip memory bandwidth, making it impractical to store all intermediate data. Therefore, there is a pressing need for a high-performance, data-reuse-efficient CNN inference engine that can manage these operations effectively while minimizing memory bandwidth usage. This thesis aims to design and implement a CNN inference engine using python, and AHIR-V2, a high-level synthesis framework, to achieve substantial computational throughput with minimal memory bandwidth, targeting an end-to-end image segmentation pipeline on an FPGA. The goal is to demonstrate competitive performance metrics, including a high compute-to-memory ratio and significant data reuse, achieving operational efficiency suitable for both research and commercial applications.

1.2 Objectives

The objectives that are achieved in this thesis are:

1. **Design Efficiency:** Develop a CNN inference engine that maximizes data reuse and minimizes memory bandwidth usage.
2. **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.
3. **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.
4. **Benchmarking:** Compare the designed engine's performance against existing state-of-the-art FPGA implementations to demonstrate competitive advantages.

Chapter 2

LeNet Architecture and Post Training Static 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:

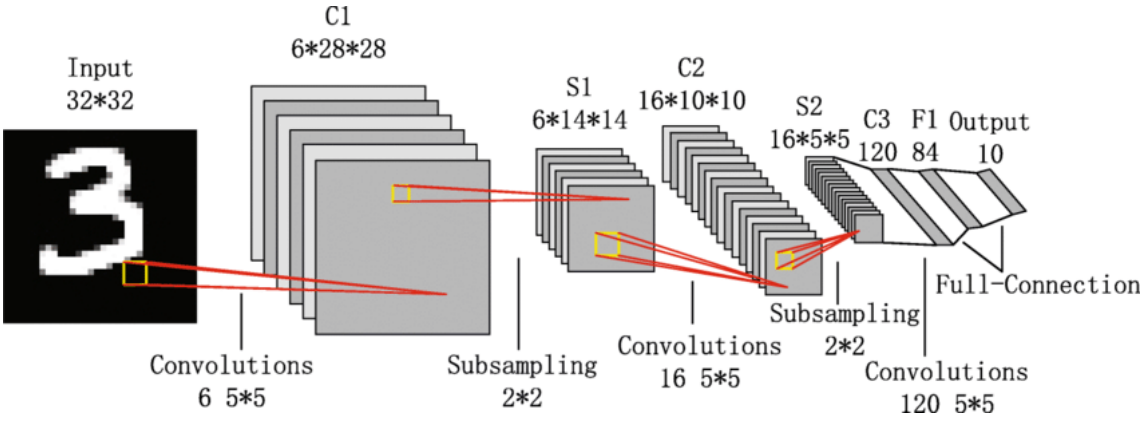

FIGURE 2.1: LeNet Architecture

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 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

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 should run at full rate.

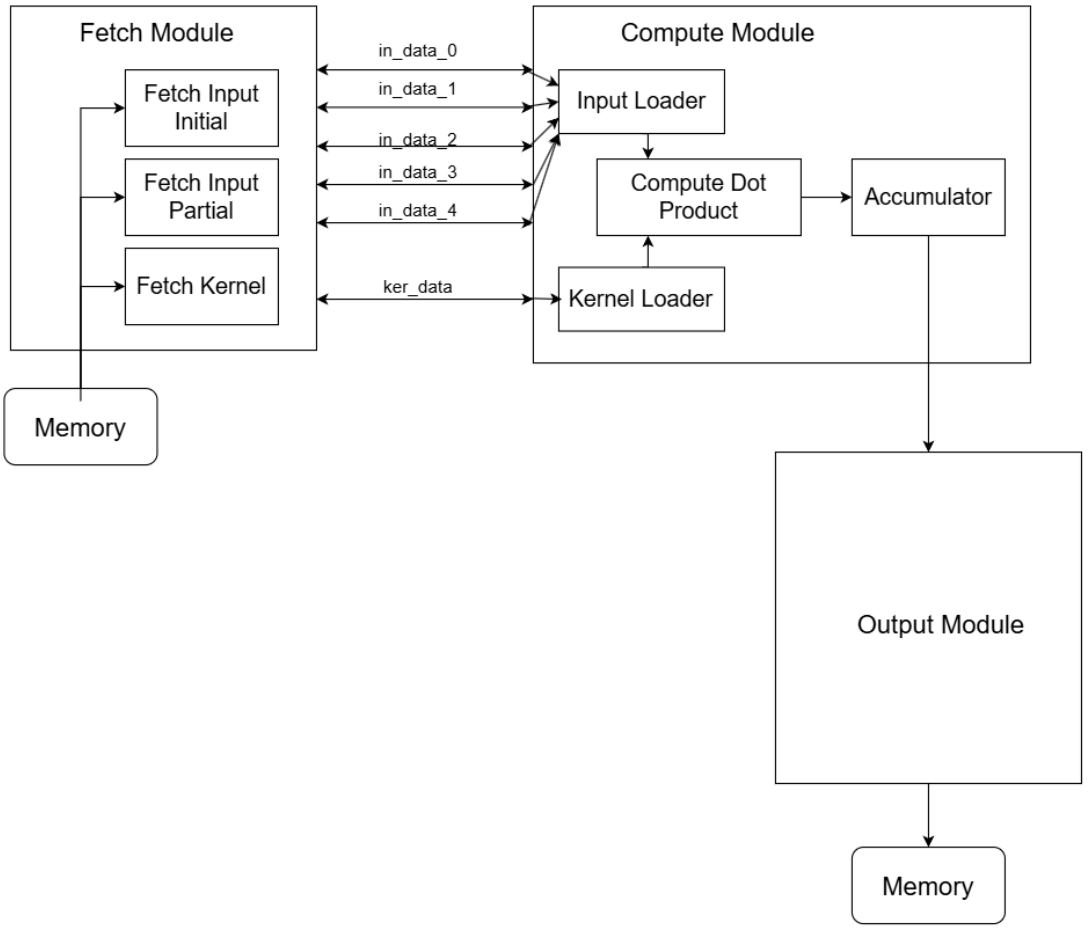

FIGURE 3.1: Block diagram for the engine implementation

3.2 Fetch Module

This module is responsible for providing steady stream of data to the compute module. A series of modules were developed to optimize the delivery rate of inputs according to the specific operation being performed.

3.2.1 Fetch Input Initial

Before starting the compute module, initially (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 are fetched from

the memory and stored in the internal pipes in_data_<i>, where i is the column number of the image. These pipes are read by Input Data Loader to provide steady stream of data to the multiplier used in the compute dot product module. Since we need to reuse the input data to reduce multiple memory access, we also re write the required data into the in_data_<i> pipes.

3.2.2 Fetch Kernel

Similar to fetch input initial, fetch kernel also stores the (k_r x k_c x chn) elements in ker_data pipe before starting the compute module. Once both the sub modules are executed, the compute module is initialized along with the fetch input partial sub module, to stream the remaining input data to the engine.

3.2.3 Fetch Input Partial

Since the compute module already have (k_r x k_c x chn) elements for processing the output, we need to reuse ((k_r) x (k_c - 1) x chn) elements as the kernel strides forward. So before the computation of the next output starts, we just need to fetch (k_r x chn) elements and store it on the pipes. By doing this, the compute module will run at full rate and need not wait for last column at each computation stage.

3.2.4 De-Quantization

Given that the data retrieved from memory is in an 8-bit unsigned integer format, it must be converted to a single-precision floating-point format to facilitate the computation stage. This is done using De-Quantization formula as given by the

PyTorch Framework. The De-Quantization formula typically involves scaling the quantized integer values by a scale factor and adding a zero-point offset.

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

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

3.3 Compute Module

Since this module forms the core of the engine, the throughput of the engine depends on the rate at which this module will run. The submodules present in this ensure that the compute module will run at full rate.

3.3.1 Compute Dot Product

The dot product submodule contains a single 32 bit floating point multiplier. This submodule reads the data from input loader and kernel loader, and then pushes the dot product of both into accumulator. Since number of dot products will be fixed, using the formula (k_r * k_c * chn) * (o_r * o_c), (o_r and o_c are number of output rows and columns), this submodule expects these number of elements from input and kernel loader to perform dot product.

3.3.2 Compute Accumulator

The accumulator also expects (k_r * k_c * chn) * (o_r * o_c) elements from the dot product submodule. In addition to this, the accumulator module should also be

aware of when to send the accumulated value to the output module. Since a output pixel is generated from summation of (k_r * k_c * chn) dot products between input and kernel, hence the accumulator sends the accumulated value after accumulating (k_r * k_c * chn) elements and then restarts the accumulator process for the next output pixel.

3.3.3 Input and Kernel Loader

These submodules read the data stored in the internal pipes by the Fetch module and supply it to the dot product submodule in the correct sequence. They are also responsible for data reutilization; once the data is accessed from the in_data and ker_data pipes, it is rewritten back to these pipes in the appropriate order to minimize frequent memory accesses.

3.4 Non Critical Components

3.4.1 Output Module

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

3.4.1.1 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.

3.4.1.2 Quantization

After completion of the non linear activation, the 32 bit float is then converted to the 8 bit uint format using the quantization formula provided by PyTorch Framework. The quantization formula involves scaling the floating-point values by a scale factor and adding a zero-point offset, followed by rounding to the nearest integer.

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

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

3.4.1.3 Pooling

The output modules additionally execute maxpooling following convolution. Since we are performing 2 x 2 max pooling, we store two rows of output data and then perform max pooling operation on the output data before writing them back to the memory.

3.4.2 Read and Write Modules

These modules serve as interfaces between functional pipelines and the memoryModule. They facilitate read and write operations to memory by providing addresses and/or data along with appropriate bitmasks. Upon completion of a memory load operation, they return the read data to the calling module.

While there are no cyclic dependencies among the core modules, they share a single resource—memory—accessible through the memoryModule. This shared resource poses a potential risk of deadlock, where one module may block another due to

pipeline dependencies. For instance, if the input module is blocked by pipeline constraints, it can hinder the readModule's ability to retrieve data from memory, consequently causing memory blockage that prevents writeback operations from completing. This domino effect can ultimately lead to deadlock, where all modules become blocked.

To mitigate deadlocks arising from shared resource contention, the read modules employ a mechanism illustrated in Figure 3.5. This mechanism ensures that the number of active iterations in readModule1 is limited by the pipeline depth of 7. Consequently, the pipeline can accommodate a maximum of 7 entries in the PIPE at any given time, preventing blocking scenarios. Thus, each memory access initiated by readModule1 is guaranteed to return, effectively averting resource contention and potential deadlocks.

It's important to note that deadlock prevention mechanisms are not necessary for writeModules. This is because output from the writeModule does not pass through a pipeline and therefore never encounters blocking situations.

3.4.3 Memory Module

The module serves as an intermediary between the engine and external memory. It handles read/write requests from other modules via their respective interfaces, transmitting a 110-bit request to the ahir core bus (ACB). The ACB forwards this request to the memory controller for access, which returns a 65-bit response containing a 1-bit error flag and 64-bit data. This response is then relayed back to the originating module. The memory module abstracts the system design from the memory interface, utilizing the ACB interface. This allows the engine to manage read/write operations—specifying flags, data, address, and bytemask—without needing to manage the specifics of the underlying memory technology (such as BRAM/DRAM).

3.4.4 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.

3.4.5 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.

3.5 Performance

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 8-engine accelerator cluster, 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 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	**Model**	**Dataset**	**Platform**	**Datatype for computation**	**Datatype for storage**	**Accuracy**
1	LeNet	MNIST Handwritten Digits	CPU	32 bit float	32 bit float	98.27%
2	LeNet	MNIST Handwritten Digits	CPU	32 bit float	8 bit posit	89%
3	ResNet-18	MNIST Handwritten Digits	CPU	32 bit float	8 bit posit	94%
4	LeNet	MNIST Handwritten Digits	FPGA	32 bit float	8 bit UINT	98.01%

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 python to llvm conversion 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.

- Supporting half precision arithmetic allows us to double the number of operators, increasing throughput compared to the current use of single precision operators in the hardware.

-

References