Design and Implementation of a 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 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 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.

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.(TODO: Add some measurement data later)

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

1.1 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.

1.2 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.

1.3 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. **Performance Optimization:** Achieve a high throughput of operations per second (GOPS) and efficient resource utilization on an FPGA.
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:

1. **Input Layer:** The input to the network is a 28x28x1 gray scale image.
2. **Convolutional Layer 1:** This layer consists of six 5x5 filters (kernels), producing six feature maps of size 24x24.
3. **Subsampling (Pooling) Layer 1:** A 2x2 max pooling operation reduces the size of each feature map to 12x12.
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 4x4.
6. **Fully Connected Layer 1:** The flattened output from the second pooling layer (16 x 4 x 4 = 256 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

3.2 Convolution Operation

3.3 Data Storage and Reuse

3.4 Quantization Operations

3.5 Non Critical Components

3.6 Performance

Chapter 4

Hardware Testing and Results

4.1 Introduction

4.1.1 Processor Code

4.1.2 Accelerator Interface

4.1.3 Results

Chapter 5

Summary

References

List of Publications

Articles Published in International/National Journals

1. Article item 1
2. Article item 2

Presentations and Proceedings in International/National Conferences

1. Conference item 1
2. Conference item 2