Internship Report

Zaqi Momin

August 15, 2024

—

Ecole Polytechnique

—

Professor Jean Pierre David

INTRODUCTION

Modern convolutional neural networks (CNNs) use significant power for matrix-vector multiplications. At inference, network parameters are fixed and arranged in matrices. By pre-determining these constant matrices, we can enhance throughput, hardware utilization, and energy efficiency. This report focuses on optimizing and reducing computational costs of CNN on FPGA by exploiting redundancies in common sub-expressions across multiple products and outputs.

Constant Matrix Multiplication

Algorithm

The core operation in digital signal processors (DSPs) is the multiplication of binary numbers. Our approach focuses on this operation by examining the principle of summing partial products. For an input variable X and a constant C, both X and C are represented in terms of their binary bits as:

$$X = \sum_{i=0}^{N-1} x_i 2^i, C = \sum_{j=0}^{M-1} c_j 2^j$$

Where x_i and c_i denote the i-th and j-th bits of X and C respectively, with N and M representing the bit widths of X and C.

The product of X and C, represented by P, can be obtained through the accumulation of their partial products. This involves multiplying the respective bits of the multiplicand and the multiplier and subsequently left-shifting by an appropriate number of positions:

$$P = \sum_{i=0}^{M-1} X c_i 2^i$$

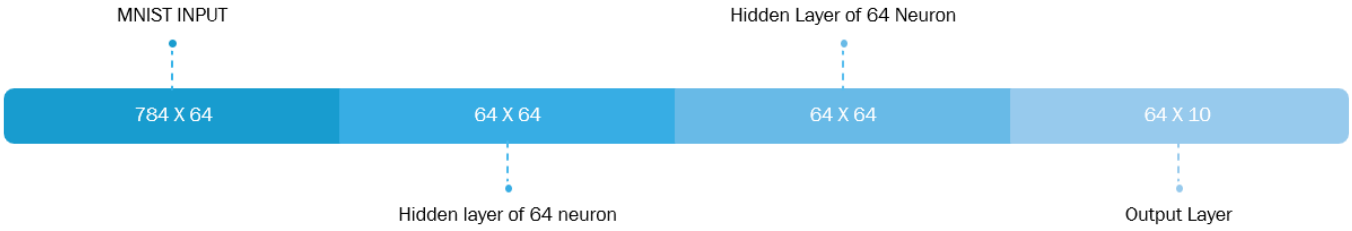
This fundamental principle forms the basis of the proposed algorithm. When the constant C is known in advance, the zero bits $c_i = 0$ can be ignored, allowing the constant multiplication to be computed with fewer shift and add operations. For example, multiplying a variable X by 193 can be expressed as a sum of terms, each being a power of 2 multiplied by X:

$$X * 193 = X(2^7 + 2^6 + 2^0)$$

This reduces the multiplication to a series of shifts and additions. Using this Algorithm I have implemented a custom neural network which was trained on MNIST Dataset.

Neural Network using Brevitas

Neural Network Architecture.



I have implemented the architecture shown in the image is a simple feedforward neural network designed for the MNIST dataset using PyTorch.

Input Layer (784 x 64): The input layer has 784 neurons (representing the 28x28 pixels of the MNIST images) connected to the first hidden layer, which has 64 neurons.

Hidden Layers (64 x 64): There are two hidden layers, each with 64 neurons. These layers process the features extracted from the input data.

Output Layer (64 x 10): The output layer has 10 neurons, corresponding to the 10 possible digit classes (0-9) in the MNIST dataset.

This network is small because the MNIST dataset is relatively simple and low-dimensional.

Brevitas Library

I have implemented the above neural network in pyTorch using Brevitas Library. Brevitas is a PyTorch library for neural network quantization, with support for both *post-training quantization (PTQ)* and *quantization-aware training (QAT)*. In my research work I used quantization-aware training because it was giving me better accuracy. Below image shows the implementation of our network:

```
def __init__(self):
    super(SimpleNN, self).__init__()
    self.quant_inp1 = qnn.QuantIdentity(bit_width=8, signed=False, return_quant_tensor=True)
    self.fc1 = qnn.QuantLinear(784, 64, bias=False,
                               weight_bit_width=8,
                               bias_bit_width=8)
    self.relu1 = qnn.QuantReLU(bit_width=26, return_quant_tensor=True)
    self.quant_inp2 = qnn.QuantIdentity(bit_width=8, signed=False, return_quant_tensor=True)
    self.fc2 = qnn.QuantLinear(64, 64, bias=False,
                               weight_bit_width=8,
                               bias_bit_width=8)
    self.relu2 = qnn.QuantReLU(bit_width=22, return_quant_tensor=True)
    self.quant_inp3 = qnn.QuantIdentity(bit_width=8, signed=False, return_quant_tensor=True)
    self.fc3 = qnn.QuantLinear(64, 64, bias=False,
                               weight_bit_width=8,
                               bias_bit_width=8)
    self.relu3 = qnn.QuantReLU(bit_width=22, return_quant_tensor=True)
    self.quant_inp4 = qnn.QuantIdentity(bit_width=8, signed=False, return_quant_tensor=True)
    self.fc4 = qnn.QuantLinear(64, 10, bias=False,
                               weight_bit_width=8,
                               bias_bit_width=8)
    self.relu4 = qnn.QuantReLU(bit_width=22, return_quant_tensor=True)
    self.quant_inp5 = qnn.QuantIdentity(bit_width=8, signed=False, return_quant_tensor=True)
```

The above code snippet is implementing a quantized neural network using pytorch_quantization. Here's explanation:

- **QuantIdentity Layers:** These layers apply quantization to the inputs of the following layers. They are used to convert the floating-point inputs to a quantized format with a specified bit width.

- **QuantLinear Layers:** These are fully connected (dense) layers with quantization applied to both weights and biases. The bit widths for both weights and biases are set to 8 bits signed.

- **QuantReLU Layers:** These are ReLU activation functions with quantization, which restrict the output to the specified bit width.

The network structure corresponds to the architecture shown earlier, with quantization applied to every layer to compare pyTorch and Verilog implementation and obtain accuracy on the MNIST dataset.

Verilog Code Generation

Verilog using Python

I used a constant matrix multiplication algorithm to generate Verilog modules. In this process, we first determine the number of operations needed to convert multiplications into shift and addition operations. Then, we create a memory array to store intermediate results. The operations are assigned according to the algorithm to generate the final output.

The bit-width of the output and memory are decided using the following formula:

$$outputBitwidth = weightBitwidth + inputBitwidth + math.ceil(math.\log(cols - 1,2))$$

Why We Use This Formula?

- **Weight Bit-width**: Represents the bit-width of the weights used in the multiplication.
- **Input Bit-width**: Represents the bit-width of the input data.
- **Logarithmic Term**: The term $\lceil log2(cols - 1) \rceil$ accounts for the maximum possible additions (or shifts) needed when combining the results of multiple multiplication operations. The number of columns (cols) corresponds to the number of elements being summed, and the logarithm ensures that enough bits are allocated to handle the accumulation of these sums without overflow.

This formula ensures that the output bit-width is sufficient to represent the final result without losing precision or encountering overflow during the matrix multiplication process.

Rectified Linear Unit

In my implementation I have used to most commonly used activation function which is ReLU. The module checks if the output is negative or positive by examining the **most significant bit (MSB)** of the input data.

- **MSB (Most Significant Bit)**: In a signed number, the MSB indicates the sign of the number.
  - If MSB = 0, the number is positive or zero.
  - If MSB = 1, the number is negative.
- **ReLU Functionality**: The module implements the ReLU function by zeroing out negative inputs and passing through non-negative inputs.
- **Bit Manipulation**: After applying ReLU, it slices the output data to a specific bit-width (8 bits), adjusting the data according to most suitable approximation obtain using python.

The design ensures that the output data is properly scaled and formatted, following the ReLU activation function and necessary bit-width adjustments.

Approximations

In my implementation, the output of the previous layer was in the 22-26 bit range, while the next layer required 8-bit inputs. Initially, I attempted to downscale the values by dividing them by $2^{18} - 2^{14}$, but this approach resulted in feeding zero values into the next layer, severely affecting the model's performance.

To address this issue, I calculated the maximum possible value of the image across the entire dataset and determined the number of bits required to represent it accurately. This allowed me to rescale the output appropriately, ensuring that the values fed into the next layer were within the required 8-bit range while preserving significant information.

This approach yielded significantly better results, aligning closely with the expected outcomes.

Additionally, I noted that Brevitas, a tool often used for quantization-aware training, also trains the activation scale, and the results obtained from my method were consistent with those from Brevitas. This further validated the effectiveness of my approach.

Results

I trained the model for 10 epochs on the MNIST dataset and achieved 95% accuracy. To validate the Verilog implementation, I compared it with the results obtained from PyTorch using Brevitas on 60,000 images. The comparison revealed only 32 mismatches, indicating that the Verilog implementation closely mirrors the performance of the PyTorch model.

Additionally, I developed a testbench in Verilog that processes input image files and outputs the predicted class in an output file. This further ensured the reliability and accuracy of the Verilog model in practical applications.

Thank You.