Convolutional Neural Network using RVV

Project for Computer Architecture and Assembly Language course under the supervision of Professor Salman Zafar

Muhammad Abdullah Faraz
ERP: 29271
Section: 98496
m.faraz.29271@khi.iba.edu.pk

Muhammad Sharique Baig
ERP: 28369
Section: 98496
m.baig.28369@khi.iba.edu.pk

Syed Hamza Ahsan
ERP: 28903
Section: 98504
s.ahsan.28903@khi.iba.edu.pk

Muhammad Faraz Ansari
ERP: 29201
Section: 98503
m.ansari.29201@khi.iba.edu.pk

***Abstract*—This report presents the implementation of a Convolutional Neural Network in Risc-V assembly code. We implemented two versions, with and without RVV to observe the difference in processing speed. We made use of the RISC-V GNU Toolchain and VeeR ISS for simulation.**

***Index Terms*—RISC-V, CNN, RVV, vector extension, MNIST, neural network, assembly code**

I. Introduction

The Convolution Neural Network (CNN) is a type of neural network used for machine learning. These neural networks are comprised of layers of neurons. The input goes from the input layer through the hidden layers to finally, the output layer, with each layer modifying its inputs. This is the feedforward process that will be our focus of discussion. CNN is a special type of neural network that works with three-dimensional data for image classification. We will focus on building a CNN that works with the MNIST dataset for digit recognition.

II. Convolution Neural Network

The function of a CNN, like a normal neural network, is divided into layers with each layer taking the input from the previous layer and passing it to the next layer after applying some function on it.

The CNN we are constructing is structures as follows:

A. Input Layer

Each of our image is a 28×28 pixel black and white image, with each pixel being a real value between 0 (whiteness) and 1 (blackness). The following equation represents an image:

$$X \in \mathbb{R}^{28 \times 28 \times 1} \tag{1}$$

B. Convolution Layer

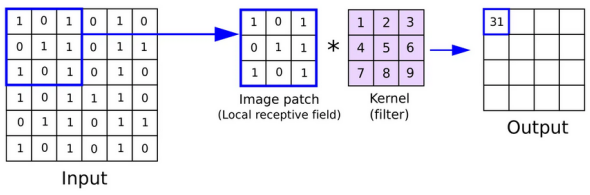

Fig. 1. Applying convolution with a 3×3 filter

In this layer, we apply a filter to the input. The filter is applied to an equal dimension sub-matrix from the input by taking the dot product of the two. This filter window is then moved by one element and applied again. After completing the row, we move to the next column.

The filter operation is more formally defined as following:

- **Filter Size:** $F_i \in \mathbb{R}^{5 \times 5}$ (*weight matrix for ith filter*)
- **Bias:** $b_i \in \mathbb{R}$ (*bias for the ith filter*)
- **Number of Filters:** $k = 8$
- **Stride:** $s = 1$
- **Padding:** $p = 0$ *(valid padding)*

The output Z_i is computed as follows. For each k:

$$Z_k(i,j) = \sum_{m=0}^{4} \sum_{n=0}^{4} F_k(i,j) \cdot X(i+m, j+n) + b_k \tag{2}$$

For $k = 8$ filters, we have:

$$Z \in \mathbb{R}^{24 \times 24 \times 8} \tag{3}$$

C. ReLU Activation

The output from the convolution layer is passed through the ReLU activation. This is not a layer but rather an activation function that removes negative values. This is efficient to compute and helps resolve the vanishing-gradient problem.

The ReLU function is defined as:

$$\text{ReLU}(x) = \max(x, 0) = \frac{x + |x|}{2} \tag{4}$$

This is applied to each element:

$$A(i,j,k) = \text{ReLU}(Z(i,j,k)) \tag{5}$$

Resulting in:

$$A \in \mathbb{R}^{+24 \times 24 \times 8} \tag{6}$$

D. Max Pool Layer

The output from ReLU is max pooled. This reduces the spatial dimensions of the feature maps. It involves sliding a window over the input and selecting the maximum value within that window. This process effectively downsamples the feature map, making it smaller and more manageable for subsequent layers.

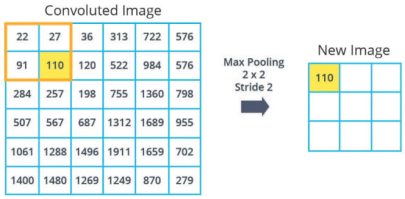

Fig. 2. Max pooling a matrix to reduce its dimensions

The max pooling is defined as:

- **Window Dimension:** 2×2
- **Stride:** 2

$$M(i,j,k) = \max_{(m,n)\in(0,1)\times(0,1)} (2i+m, 2j+n) \tag{7}$$

$$M \in \mathbb{R}^{+12\times12\times8} \tag{8}$$

E. Flatten into Vector

The output from the max pool layer is flattened to a one dimensional vector to be passed to the dense layer.

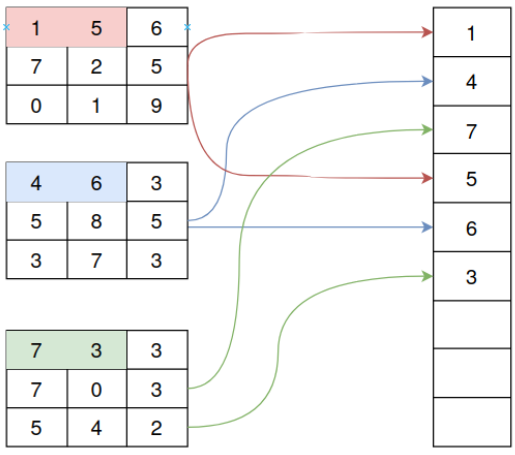

Fig. 3. Flattening three matrices into a vector

The flattened vector is computed as:

$$V_i = M(\left\lfloor \frac{i}{12\cdot 8} \right\rfloor, \left\lfloor \frac{i \mod (12\cdot 8)}{8} \right\rfloor, i \mod 8) \tag{9}$$

Note that we pick each *i*th element from the matrices rather than a complete matrix at a time. The final output we get in

$$V \in \mathbb{R}^{+1152} \tag{10}$$

F. Fully Connected (Dense) Layer

The dense layer has its weights and biases which work on the output from the previous layer. The corresponding weights are multiplied with each of the inputs and a bias value is added. This is more formally defined as:

- **Weights:** $W \in \mathbb{R}^{10\times1152}$
- **Biases:** $b \in \mathbb{R}^{10}$

The *i*th entry of output D is computed as:

$$D_i = W_i \cdot V + b_i \tag{11}$$

$$D \in \mathbb{R}^{10} \tag{12}$$

G. Softmax Activation

We pass D through the softmax function to obtain probabilities whose sum is one.

The *i*th is obtained by:

$$p_i = \frac{e^{D_i}}{\sum_{i=1}^{10} e^{D_i}} \tag{13}$$

H. Finding Label

The value predicted by the CNN is simply the index of the largest element in p:

$$i = \arg\max_{0\le j\le 9} p_j \tag{14}$$

The final result of our CNN is i, a integer digit that the model *thinks* was the input.

III. Obtaining the Weights and Biases

In a a complete CNN, we train our model through a fairly large training data set to obtain the model parameters. Since we are only concerned with the feedforward process in our implementations, the necessary weights and biases are obtained from a Python program making use of the TensorFlow library:

```
import tensorflow as tf
import numpy as np

# Assume our cnn is in the variable model

# Extract Convolutional Layer Parameters
conv_layer = model.get_layer('conv_layer')
W_conv, b_conv = conv_layer.get_weights()

# Extract Fully Connected Layer Parameters
fc_layer = model.get_layer('dense_layer')
W_fc, b_fc = fc_layer.get_weights()

# Save the parameters
np.savez(filename,
             W_conv=W_conv,
             b_conv=b_conv,
             W_fc=W_fc,
             b_fc=b_fc)
```

This gives us the following needed parameters:

- The 8 filters with their 5×5 values
- The 8 biases for each filter
- The 10×1152 weights for dense layer
- The 10 biases for the dense layer

IV. Implementing in RISC-V - Non Vectorized

The first thing we needed to do was properly format our data and allocate space for computation results and assign labels to access them. We structured each label as we used them in sequence:

- **Input:** The 28×28 image is stored here
- **Filter_1_Weights - Filter_8_Weights:** The eight 5×5 filters
- **Bias_1 - Bias_8:** The eight biases of each filter
- **Output_1 - Output_8:** The outputs from the convolution layer
- **Maxpool_Output_1 - Maxpool_Output_8:** The outputs from max pooling
- **Flatten_Data:** Holds the flattened data
- **Dense_Weights:** The 10×1152 weights for dense layer
- **Dense_Bias:** The 10 biases for dense layer
- **Dense_Result:** Holds output from dense layer
- **Probabilities:** Holds the probabilities from softmax
- **Prediction:** Holds the value predicted by the model

With our data properly handled, the next step is to start implementing each of the layers.

We start off with the convolution layer. Since this is a layer that will be used eight times, it has been implemented as a callable function.

```
li s0, 28     # image size
li s1, 784    # image length
li s2, 5     # filter size
li s3, 25     # filter length
li s4, 24     # output size
li s5, 576    # output length

la a0, Input # Load base address of INPUT

la a1, Filter_1_Weights # base address filter
la a2, Bias_1 # base address of BIAS
la a3, Output_1 # base address of OUTPUT
flw f0, 0(a2) # bias into f0
```

The other seven filters will be called in a similar way.

After calling the function, our function itself would iterate over all elements in the image and compute the dot product with the filter:

```
loop_D:
    beq t3, s2, end_loop_D

    add t4, t0, t2   # im_row = i + m
    add t5, t1, t3   # im_col = j + n

    # im_index = im_row * IM_SIZE + im_col
    mul t4, t4, s0
    add t4, t4, t5

    mul t5, t2, s2   # filter index
    add t5, t5, t3

    slli t4, t4, 2   # get image address
    add t4, t4, a0

    slli t5, t5, 2   # get filter address
    add t5, t5, a1

    flw f2, 0(t4) # image value
    flw f3, 0(t5) # filter value

    fmul.s f4, f2, f3
    fadd.s f1, f1, f4

    addi t3, t3, 1
    j loop_D
end_loop_D:
```

This data would be passed forward for ReLU activation:

```
relu_loop:
    beq t1, t0, end_relu_loop

    # t2 = i * 4 (offset of 4 bytes)
    slli t2, t1, 2
    # t3 holds address of current element
    add t3, a0, t2

    flw f1, 0(t3)         # t3 = INPUT[i]

    fmax.s f2, f1, f0 # f2 = max(f1, 0)

    fsw f2, 0(t3)
    addi t1, t1, 1
    j relu_loop
end_relu_loop:
```

This result will be passed to the max pool layer. That layer functions similar to ReLU but differs as it computes the max over four elements instead of two here.

With that, the data is flattened:

```
flatten_loop:
    beq t0, t2, end_flatten_loop        # col
    li t3, 0
inner_flatten_loop:
    beq t3, t1, end_inner_flatten_loop  # row

    mul s0, t3, t2
    add s0, s0, t0

    slli s0, s0, 2
    add s0, s0, a0

    flw f0, 0(s0)

    fsw f0, 0(a1)

    addi a1, a1, 4


    addi t3, t3, 1
    j inner_flatten_loop

end_inner_flatten_loop:
    addi t0, t0, 1
    j flatten_loop
end_flatten_loop:
```

This is passed to the dense layer, where the result is just a linear combination of the weights, inputs, and biases.

Next, we apply softmax. Since there is no way to compute exponential in assembly directly, we made use of the Taylor approximation:

$$e^x \approx \sum_{n=0}^{N} \frac{x^n}{n!} \tag{15}$$

The exponential grows very fast. So, for numerical stability, we subtract the maximum value from each value

$$y_i' = y_i - \max y \tag{16}$$

Another problem we encountered was that for larger negative values , this series was not approximating correctly, So we used the identity:

$$e^{-x} = \frac{1}{e^x} \tag{17}$$

After that we simply needed to find the maximum element and its corresponding index.

This was done by using `sw t0, 0(a0)`. After simulation, we look over the `log.txt` for what value was stored:

```
... 00000005   sw        t0, 0x0(a0)
```

The above line shows that the number guessed was 5

V. Implementing in RISC-V - Vectorized

The vectorized version follows the same format as the non-vectorized one.

Starting of with the convolution layer, the outer loops iterate over the output image dimensions, with the inner loops applying the filter to each windows. If four or more columns remain, the code processes four outputs at once using vector operations: it loads a 1D slice of the input, multiplies it with a scalar filter weight, and accumulates the result in vector registers. The result is ReLU-activated and stored. For remaining columns (less than 4), scalar computation is used instead. This scalar path performs standard nested loop convolution, applying the filter element-wise and accumulating into a scalar register before ReLU and storing.

```
#  accumulate
# v0 += input * filter_value
vfmacc.vf v0, ft0, v4

# some code in between

# Apply ReLU and store 4 results
fcvt.s.w ft5, zero
vfmax.vf v0, v0, ft5  # ReLU activation
# output address
mul t2, t0, s4
add t2, t2, t1
slli t2, t2, 2
add t2, a3, t2

# Store
vse32.v v0, (t2)
```

Notice that ReLU was added into convolution part to speed up the process.

For max pooling, we did not vectorize it as we could not find a way to do it efficiently. Using RVV only made it more complicated and increased the number of instructions, so we stuck with the same logic here.

The same reasoning applied to the flattening layer.

For the dense layer, it begins by loading the biases into the output buffer. For each output neuron, it loads the corresponding weight row and adds the dot product with the input vector. The input is processed in tiles of up to 64 elements using RISC-V vector instructions. Each tile is loaded along with its corresponding weight slice, multiplied element-wise, and reduced to a scalar sum. The partial sums are accumulated into the output using floating-point addition. Vector registers are cleared between iterations to avoid data hazards.

```
vle32.v v4, (s3)         # Load input tile
vle32.v v8, (s4)         # Load weight tile

# Multiply and reduce
vfmul.vv v12, v4, v8 # Element-wise multiply
vfredusum.vs v16, v12, v16 # Reduce to scalar

# some code in between

# Extract accumulated sum and add to result
vfmv.f.s f1, v16       # Extract sum
fadd.s f0, f0, f1      # Add to current result
fsw f0, 0(s2)          # Store updated result

# Clear accumulator for next output
vmv.v.i v16, 0
```

For softmax, after subtracting the maximum element, we created a mask for negative elements and turned them positive. After that the Taylor approximation was applied to the ten elements in one go and negative values were handled in the final result.

```
# Finding max and subtracting it
vle32.v v0, (a0)
vfredmax.vs v1, v0, v1
vfmv.f.s f1, v1
vfsub.vf v2, v0, f1
```

```
# Masking negative numbers
fcvt.s.w f0, zero # f0 = 0.0
# v0 = mask for negative values
vmflt.vf v0, v2, f0
vfabs.v v6, v2      # v6 = |x|
```

```
# Computing till first 5 terms
# Term 1: x/1! = x (already in v5)
vfadd.vv v4, v4, v5      # result += x

# Term 2: x^2/2!
vfmul.vv v7, v7, v6
li t3, 2
fcvt.s.w f3, t3
vfdiv.vf v8, v7, f3
vfadd.vv v4, v4, v8

# Continue for remaining
```

```
# Handle negative values
li t3, 1
fcvt.s.w f3, t3
vfrdiv.vf v9, v4, f3
vmerge.vvm v4, v4, v9, v0
```

```
# Compute sum of all exp and store
vfredusum.vs v10, v4, v10  # Sum all exp
vfmv.f.s f0, v10           # Extract sum to f0
vfdiv.vf v11, v4, f0 # v11 = exp_values / sum

vse32.v v11, (a1)
```

After that, find the max index and store it:

```
la a0, Probabilities
li t2, 10
vsetvli t1, t2, e32, m1, ta, ma
vle32.v v1, (a0)         # Load probabilities

# Find global maximum
vmv.s.x v5, x0
vredmax.vs v5, v1, v5 # v5[0] = global max

# Create mask for positions equal to maximum
vfmv.f.s f1, v5
vmfeq.vf v0, v1, f1 # Mask all elements = max

# Find first index with maximum value
vfirst.m t0, v0 # First set bit index

# Store result
la a1, Prediction
sw t0, 0(a1)
```

VI. Comparison between RVV and non-RVV

Performance Metric	non-RVV	RVV
Instructions	2,357,637	567,536
PET	4.09s	1.16s
Instr/sec	575,984	489,148

As observed, the use of RVV led to huge performance gains. The total instructions needed and the PET reduced drastically. And while the intruction per second decreased slightly due to the overhead of vector instructions, the overall performance gains make this negligible.

VII. The Vector Instructions

Instruction	Count	Description
vle32.v	6	Load data from an address into a vector
vsetvli	5	Set vector length
vfmv.v.f	5	Move fp into v[0] register
vfmul.vv	5	Multiply two vectors
vfdiv.vf	5	Divide vector by a scalar
vfadd.vv	5	Add two vectors
vfmv.f.s	4	Move v[0] into fp register
vse32.v	3	Save vector to an address
vmv.v.v	3	Move between vector registers
vfredusum.vs	2	Vector reduction unordered sum
vredmax.vs	1	Vector reduction for max element
vmv.v.i	1	Move immediate to vector
vmv.s.x	1	Move integer to vector
vmflt.vf	1	v0[i] = (v1[i] $<$ f2) ? 1 : 0
vmfeq.vf	1	v0[i] = (v1[i] == f2) ? 1 : 0
vmerge.vvm	1	Merge two vector based on a mask
vfsub.vf	1	Subtract scalar value from vector
vfredmax.vs	1	Vector reduction for max element with floating point value
vfrdiv.vf	1	Vector reciprocal division with scalar
vfmax.vf	1	vd[i]=max(vs2[i],fs1)
vfmacc.vf	1	vd[i]=vd[i]+(fs1×vs2[i])
vfirst.m	1	Locating the first element in a vector satisfying the condition
vfabs.v	1	Takes absolute of each element
Total	56	

Acknowledgment

We would like to thank our professor and mentor, Sir Salman Zafar for teaching us this course and guiding us through this project. We would also like to thank the TAs for this course, Rafay Ahmed, Azqa Aqeel, Abdul Wasay Imran and Saad Imam for helping us throughout this project.