Final Project:

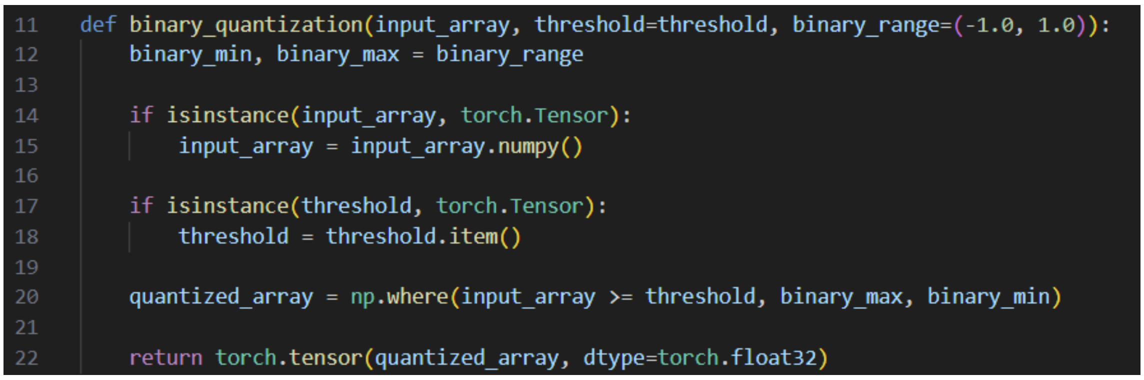
CNN Accelerator for MNIST Classification

20201082 H. Kim, 20211184 H. Yang

I. Used Quantization methods

1. Binary Neural Network (BNN)

In this project, binary neural network (BNN) was used for better hardware utilization. To do so, we binarized weights and activation to -1 and 1 with `binary_quantization` function as shown below, based on the value of threshold.

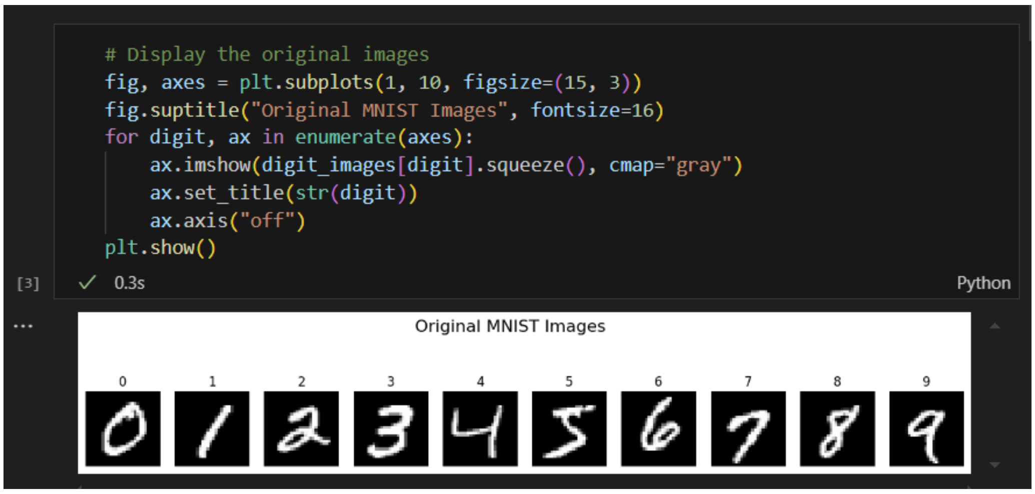


```python
def binary_quantization(input_array, threshold=threshold, binary_range=(-1.0, 1.0)):
    binary_min, binary_max = binary_range

    if isinstance(input_array, torch.Tensor):
        input_array = input_array.numpy()

    if isinstance(threshold, torch.Tensor):
        threshold = threshold.item()

    quantized_array = np.where(input_array >= threshold, binary_max, binary_min)

    return torch.tensor(quantized_array, dtype=torch.float32)
```

Figure 1 Binary Quantization

2. Input Binarization

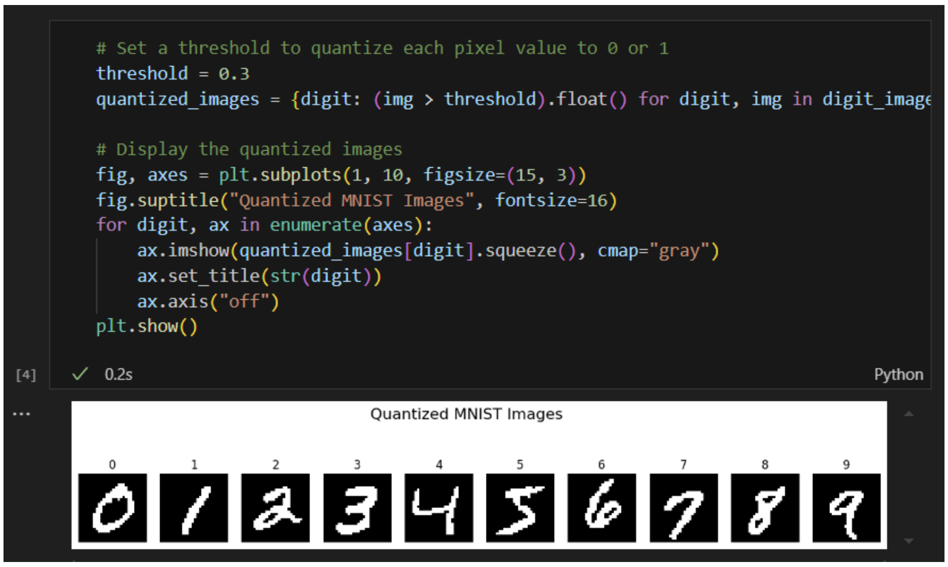
Here, the value of threshold is set by visualizing each label.



```python
# Display the original images
fig, axes = plt.subplots(1, 10, figsize=(15, 3))
fig.suptitle("Original MNIST Images", fontsize=16)
for digit, ax in enumerate(axes):
    ax.imshow(digit_images[digit].squeeze(), cmap="gray")
    ax.set_title(str(digit))
    ax.axis("off")
plt.show()
```

Figure 2 Original MNIST Data

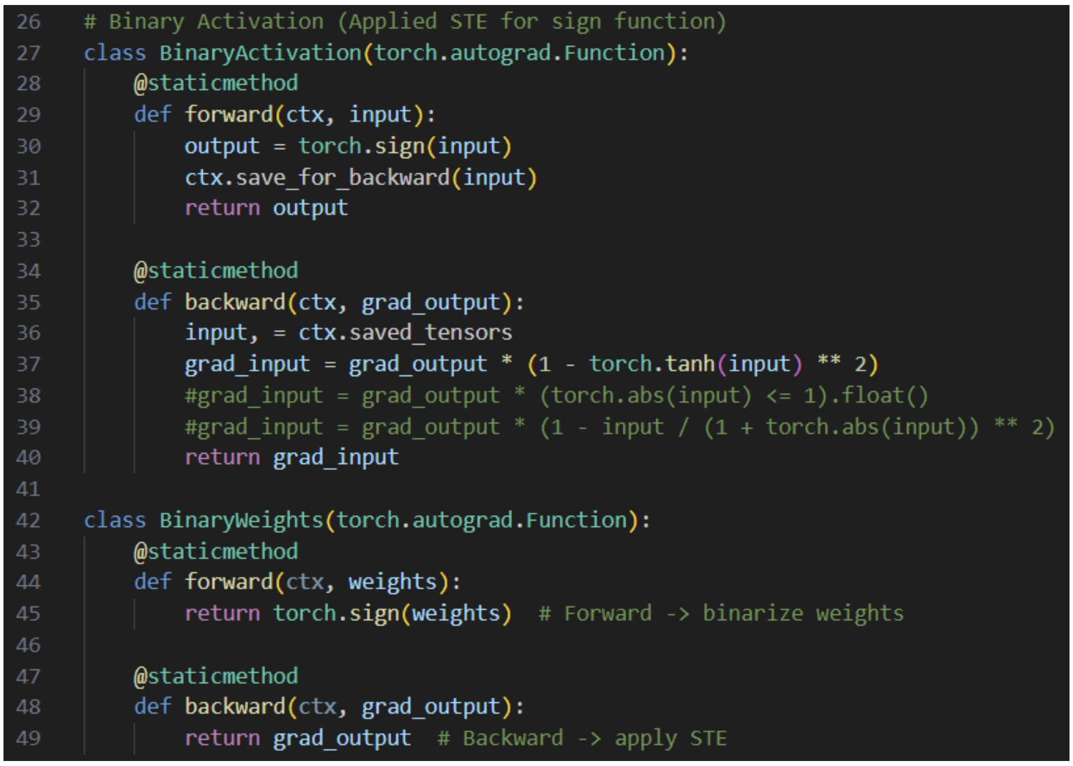
With simulation, the best threshold for our model was 0.2.



```
# Set a threshold to quantize each pixel value to 0 or 1
threshold = 0.3
quantized_images = {digit: (img > threshold).float() for digit, img in digit_image

# Display the quantized images
fig, axes = plt.subplots(1, 10, figsize=(15, 3))
fig.suptitle("Quantized MNIST Images", fontsize=16)
for digit, ax in enumerate(axes):
    ax.imshow(quantized_images[digit].squeeze(), cmap="gray")
    ax.set_title(str(digit))
    ax.axis("off")
plt.show()
```

Figure 3 Tested Quantized MNIST Images from threshold 0.1 to 0.9, and concluded that 0.2 shows the clearest image.

3. Weight Binarization / Activation Binarization



```
# Binary Activation (Applied STE for sign function)
class BinaryActivation(torch.autograd.Function):
    @staticmethod
    def forward(ctx, input):
        output = torch.sign(input)
        ctx.save_for_backward(input)
        return output

    @staticmethod
    def backward(ctx, grad_output):
        input, = ctx.saved_tensors
        grad_input = grad_output * (1 - torch.tanh(input) ** 2)
        #grad_input = grad_output * (torch.abs(input) <= 1).float()
        #grad_input = grad_output * (1 - input / (1 + torch.abs(input)) ** 2)
        return grad_input

class BinaryWeights(torch.autograd.Function):
    @staticmethod
    def forward(ctx, weights):
        return torch.sign(weights)  # Forward -> binarize weights

    @staticmethod
    def backward(ctx, grad_output):
        return grad_output  # Backward -> apply STE
```

Figure 4 Activation Binarization

In case of sign function, the back propagation is not applied thus we used a STE method which is:

$$\text{grad\_input} \approx 1 - \tanh^2 x$$

It is derived from the derivative properties of the hyperbolic tangent (tanh) function. The tanh function is defined as:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Its range is $(-1, 1)$, with values approaching 1 as $x \to \infty$ and -1 as $x \to -\infty$. The square of $\tanh(x)$ spans the range $[0, 1)$ and describes how closely the value of $\tanh(x)$ approaches its upper bound. The derivative of $\tanh(x)$ is:

$$\frac{d}{dx}\tanh(x) = 1 - \tanh^2(x)$$

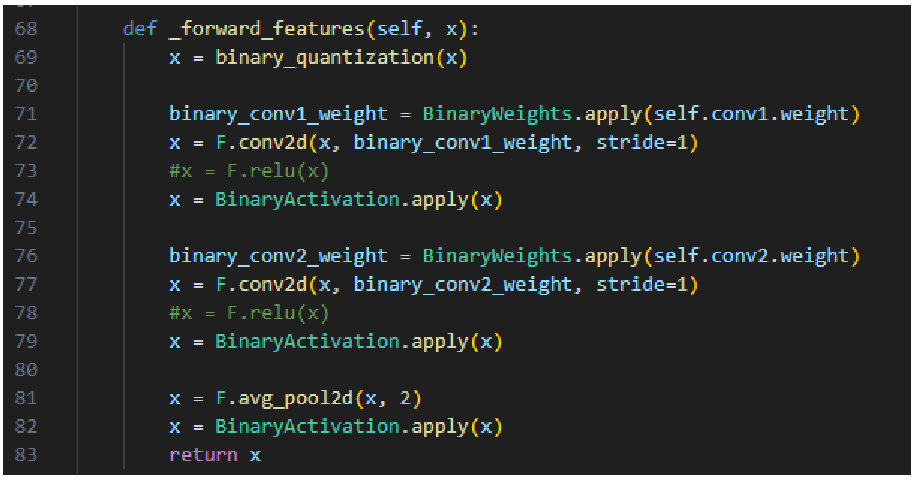
Here, the equation $\text{grad\_input} \approx 1 - \tanh^2(x)$ works because it is derivative of approximated sign function.

However, there are a few drawbacks to this method.

1. Precision Near Zero
   Around $x = 0$, the approximation may deviate significantly from the true sign function due to the nature of $\tanh(x)$, which does not change abruptly at $x = 0$.

2. Approximation Error
   For extreme values of x, $1 - \tanh^2(x)$ does not converge exactly to ± 1, leading to slight inaccuracies compared to the ideal sign function.

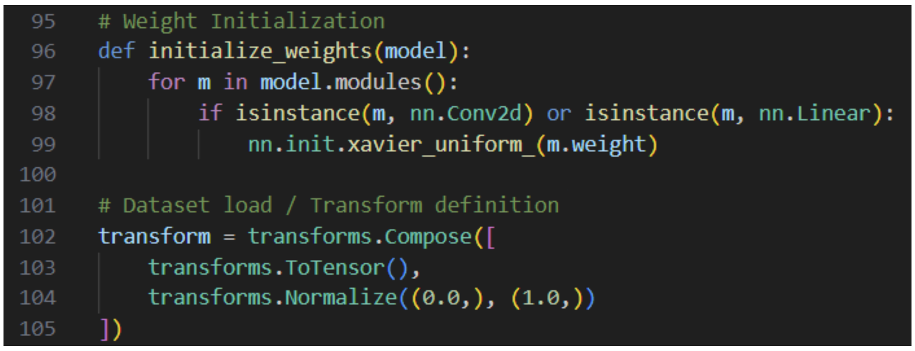
Thus, the proposed method for sign function results in a reduction of final accuracy. This could be resolved by avoiding the values near zero to be determined incorrectly, with other methods.



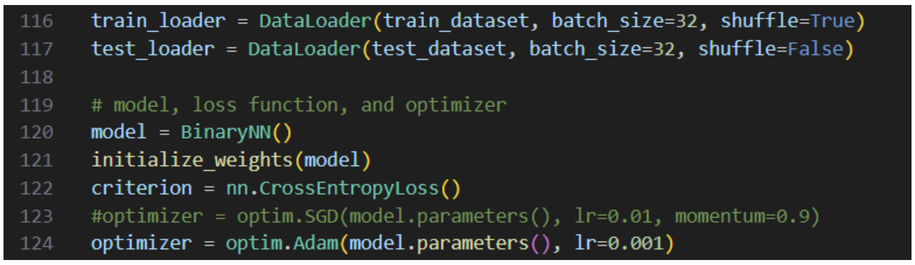
```python
    def _forward_features(self, x):
        x = binary_quantization(x)

        binary_conv1_weight = BinaryWeights.apply(self.conv1.weight)
        x = F.conv2d(x, binary_conv1_weight, stride=1)
        #x = F.relu(x)
        x = BinaryActivation.apply(x)

        binary_conv2_weight = BinaryWeights.apply(self.conv2.weight)
        x = F.conv2d(x, binary_conv2_weight, stride=1)
        #x = F.relu(x)
        x = BinaryActivation.apply(x)

        x = F.avg_pool2d(x, 2)
        x = BinaryActivation.apply(x)
        return x
```

Figure 5 In `BinaryNN` *function,* `BinaryActivation` *functions were applied between each layer.*

The training step was done with binarization functions between each layer.


```python
# Weight Initialization
def initialize_weights(model):
    for m in model.modules():
        if isinstance(m, nn.Conv2d) or isinstance(m, nn.Linear):
            nn.init.xavier_uniform_(m.weight)

# Dataset load / Transform definition
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.0,), (1.0,))
])
```

For better accuracy, image normalization and appropriate weight initialization function were used.


```python
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False)

# model, loss function, and optimizer
model = BinaryNN()
initialize_weights(model)
criterion = nn.CrossEntropyLoss()
#optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

For BNN, smaller batch size leads to better training thus we've set the batch size to 32. For gradient descent method, we used Adam.

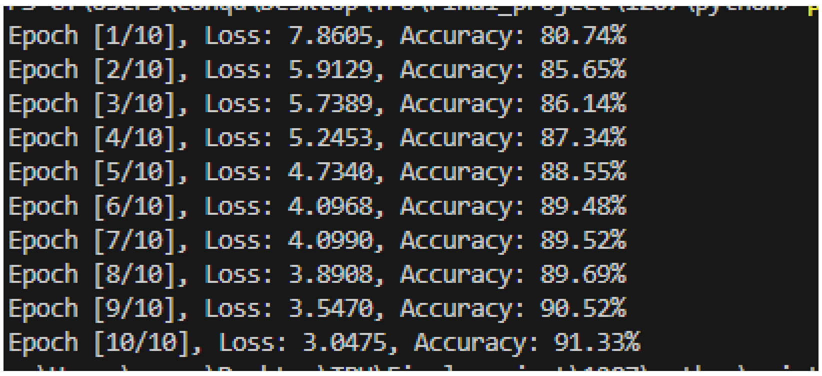
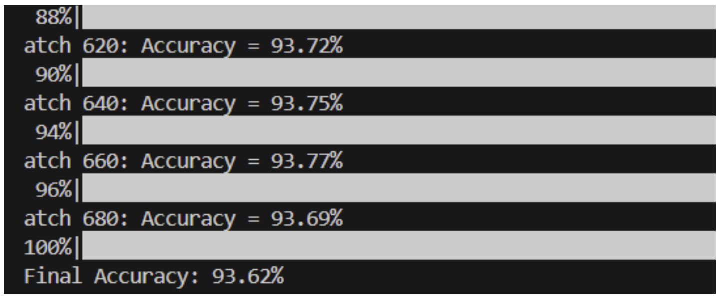

```
Epoch [1/10], Loss: 7.8605, Accuracy: 80.74%
Epoch [2/10], Loss: 5.9129, Accuracy: 85.65%
Epoch [3/10], Loss: 5.7389, Accuracy: 86.14%
Epoch [4/10], Loss: 5.2453, Accuracy: 87.34%
Epoch [5/10], Loss: 4.7340, Accuracy: 88.55%
Epoch [6/10], Loss: 4.0968, Accuracy: 89.48%
Epoch [7/10], Loss: 4.0990, Accuracy: 89.52%
Epoch [8/10], Loss: 3.8908, Accuracy: 89.69%
Epoch [9/10], Loss: 3.5470, Accuracy: 90.52%
Epoch [10/10], Loss: 3.0475, Accuracy: 91.33%
```

Figure 6 Training result of BNN

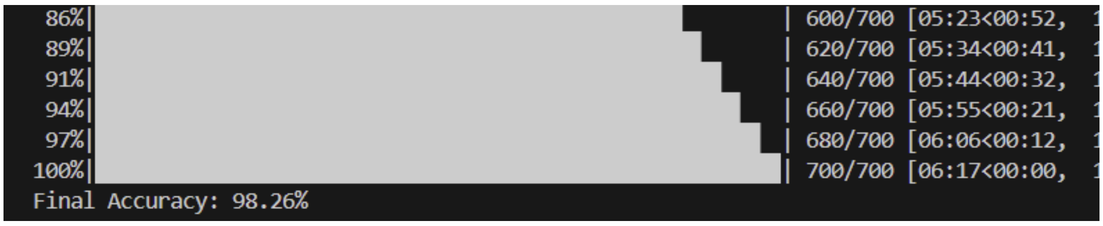
The training results are shown above.

4. Quantization Performance (Inference Accuracy)

The inference accuracy of quantized BNN was 93.62%


```
 88%|
atch 620: Accuracy = 93.72%
 90%|
atch 640: Accuracy = 93.75%
 94%|
atch 660: Accuracy = 93.77%
 96%|
atch 680: Accuracy = 93.69%
100%|
Final Accuracy: 93.62%
```

while FP32 showed 98.26% accuracy.

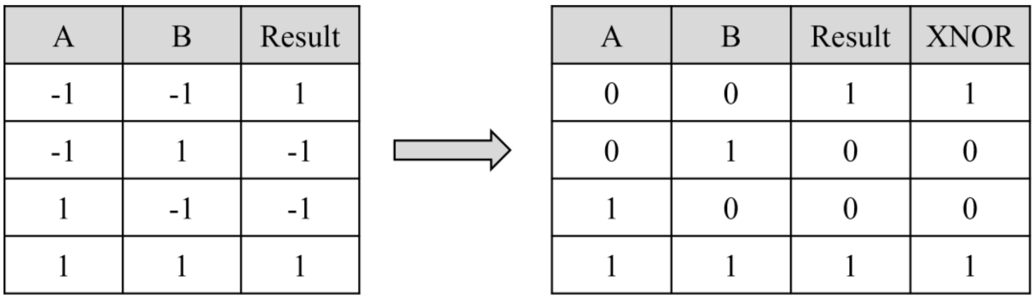


```
 86%|                                                   | 600/700 [05:23<00:52,  1
 89%|                                                   | 620/700 [05:34<00:41,  1
 91%|                                                   | 640/700 [05:44<00:32,  1
 94%|                                                   | 660/700 [05:55<00:21,  1
 97%|                                                   | 680/700 [06:06<00:12,  1
100%|                                                   | 700/700 [06:17<00:00,  1
Final Accuracy: 98.26%
```

II. Hardware Architecture

1. XNOR and Popcount

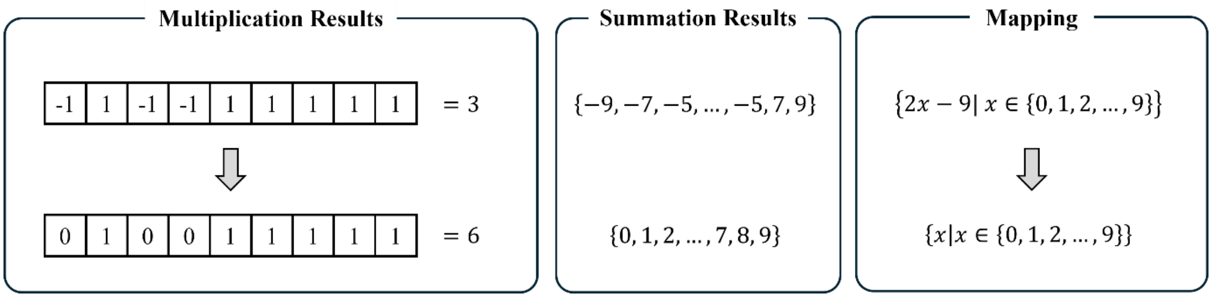
The weights and activation, which was set as -1 and 1 in software, is assigned to 0 and 1 in hardware. In hardware, we can implement multiplication of weights and activations in software with XNOR operation, and addition with popcount.



A	B	Result
-1	-1	1
-1	1	-1
1	-1	-1
1	1	1

A	B	Result	XNOR
0	0	1	1
0	1	0	0
1	0	0	0
1	1	1	1

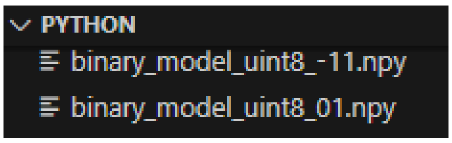
Figure 7 Results of XNOR is the equivalent with Multiplication of BNN.



Multiplication Results
-1 1 -1 -1 1 1 1 1 1 = 3
0 1 0 0 1 1 1 1 1 = 6
Summation Results
$\{-9, -7, -5, \ldots, -5, 7, 9\}$
$\{0, 1, 2, \ldots, 7, 8, 9\}$
Mapping
$\{2x - 9 \mid x \in \{0, 1, 2, \ldots, 9\}\}$
$\{x \mid x \in \{0, 1, 2, \ldots, 9\}\}$

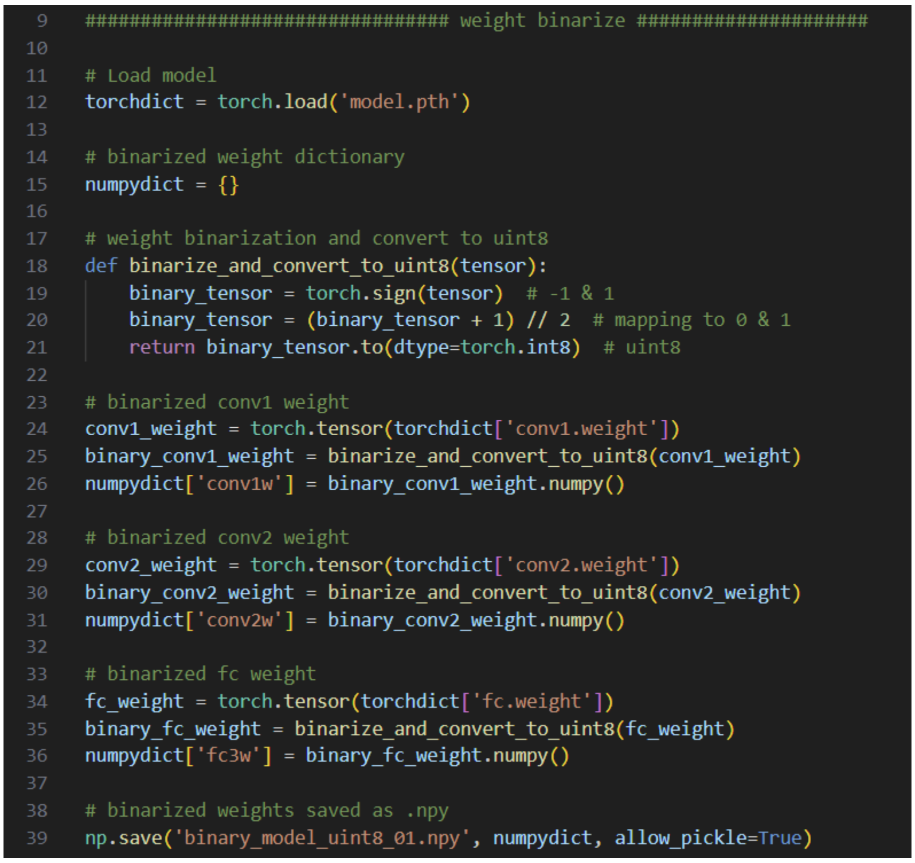
Figure 8 Popcount can replace Summation Results in convolution when BNN.

In weight binarization step, we save trained weights in `.npy` file in two different ways: -1/1 for software computation and 0/1 for hardware computation.



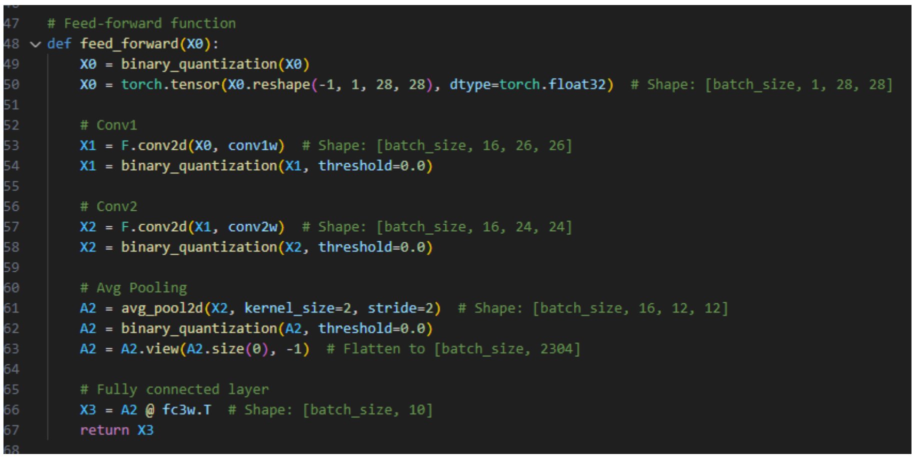
```
PYTHON
  binary_model_uint8_-11.npy
  binary_model_uint8_01.npy
```

The one with weight values of -1 and 1 was used on inference step in CPU (for verifying pretrained model), while the one with weight values of 0 and 1 was used to hardware design.



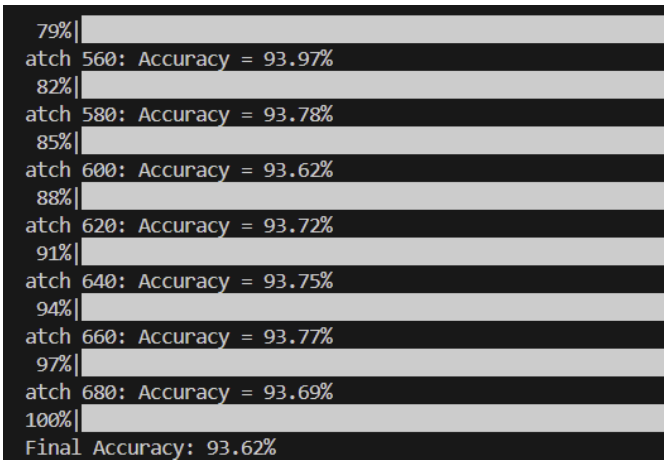
```python
################################## weight binarize #####################

# Load model
torchdict = torch.load('model.pth')

# binarized weight dictionary
numpydict = {}

# weight binarization and convert to uint8
def binarize_and_convert_to_uint8(tensor):
    binary_tensor = torch.sign(tensor)  # -1 & 1
    binary_tensor = (binary_tensor + 1) // 2  # mapping to 0 & 1
    return binary_tensor.to(dtype=torch.int8)  # uint8

# binarized conv1 weight
conv1_weight = torch.tensor(torchdict['conv1.weight'])
binary_conv1_weight = binarize_and_convert_to_uint8(conv1_weight)
numpydict['conv1w'] = binary_conv1_weight.numpy()

# binarized conv2 weight
conv2_weight = torch.tensor(torchdict['conv2.weight'])
binary_conv2_weight = binarize_and_convert_to_uint8(conv2_weight)
numpydict['conv2w'] = binary_conv2_weight.numpy()

# binarized fc weight
fc_weight = torch.tensor(torchdict['fc.weight'])
binary_fc_weight = binarize_and_convert_to_uint8(fc_weight)
numpydict['fc3w'] = binary_fc_weight.numpy()

# binarized weights saved as .npy
np.save('binary_model_uint8_01.npy', numpydict, allow_pickle=True)
```

Figure 9 Creation of numpy file based on model file.



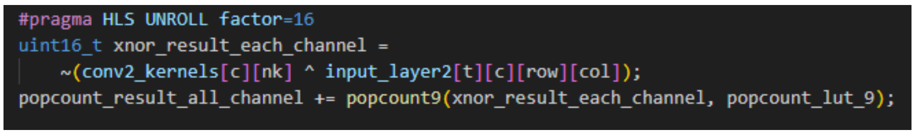
```python
# Feed-forward function
def feed_forward(X0):
    X0 = binary_quantization(X0)
    X0 = torch.tensor(X0.reshape(-1, 1, 28, 28), dtype=torch.float32)  # Shape: [batch_size, 1, 28, 28]

    # Conv1
    X1 = F.conv2d(X0, conv1w)  # Shape: [batch_size, 16, 26, 26]
    X1 = binary_quantization(X1, threshold=0.0)

    # Conv2
    X2 = F.conv2d(X1, conv2w)  # Shape: [batch_size, 16, 24, 24]
    X2 = binary_quantization(X2, threshold=0.0)

    # Avg Pooling
    A2 = avg_pool2d(X2, kernel_size=2, stride=2)  # Shape: [batch_size, 16, 12, 12]
    A2 = binary_quantization(A2, threshold=0.0)
    A2 = A2.view(A2.size(0), -1)  # Flatten to [batch_size, 2304]

    # Fully connected layer
    X3 = A2 @ fc3w.T  # Shape: [batch_size, 10]
    return X3
```

With the weights saved as `numpy` files, the verification of model is done by evaluating accuracy when going through the same algorithm with hardware as forward propagation.

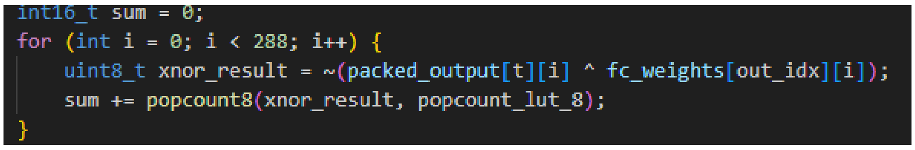


```
 79%|
atch 560: Accuracy = 93.97%
 82%|
atch 580: Accuracy = 93.78%
 85%|
atch 600: Accuracy = 93.62%
 88%|
atch 620: Accuracy = 93.72%
 91%|
atch 640: Accuracy = 93.75%
 94%|
atch 660: Accuracy = 93.77%
 97%|
atch 680: Accuracy = 93.69%
100%|
Final Accuracy: 93.62%
```

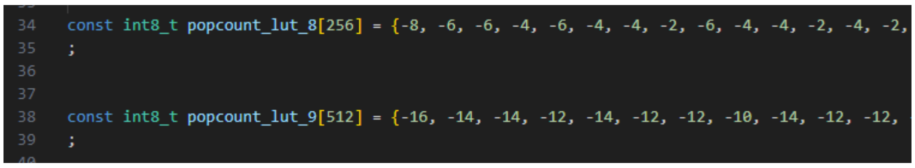
Inferencing gives us 93.62% accuracy as well as training step.



```
#pragma HLS UNROLL factor=16
uint16_t xnor_result_each_channel =
    ~(conv2_kernels[c][nk] ^ input_layer2[t][c][row][col]);
popcount_result_all_channel += popcount9(xnor_result_each_channel, popcount_lut_9);
```



```
int16_t sum = 0;
for (int i = 0; i < 288; i++) {
    uint8_t xnor_result = ~(packed_output[t][i] ^ fc_weights[out_idx][i]);
    sum += popcount8(xnor_result, popcount_lut_8);
}
```



```
const int8_t popcount_lut_8[256] = {-8, -6, -6, -4, -6, -4, -4, -2, -6, -4, -4, -2, -4, -2,
;


const int8_t popcount_lut_9[512] = {-16, -14, -14, -12, -14, -12, -12, -10, -14, -12, -12,
;
```

The popcount is done with LUT which reduces latency.

In our hardware implementation, we utilized `uint8_t` and `uint16_t` data types, which adhere to the C++ standard. Consequently, a custom packing method was developed to support XNOR and popcount operations. Although the `ap_uint<>` format offered a more efficient alternative, time constraints prevented us from revising the architecture to incorporate it.

2. Bit Packing

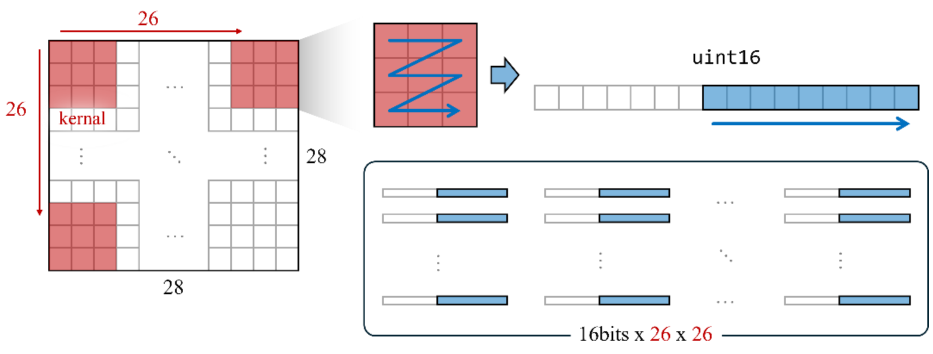


Figure 1 0 Data Dimension transformation after Bit Packing

The 3x3 kernel-based convolution operation was implemented using the IM2COL approach, as illustrated in the figure. This method transforms the input data into a column-major matrix, enabling efficient computation. In this case, for a 28x28 MNIST input, the sliding window approach is replaced with IM2COL to optimize for parallel computation, significantly improving hardware performance for operations involving small kernels.

By flattening the overlapping regions into a structured matrix, the convolution process becomes better suited for matrix multiplication, facilitating parallelism and hardware acceleration.

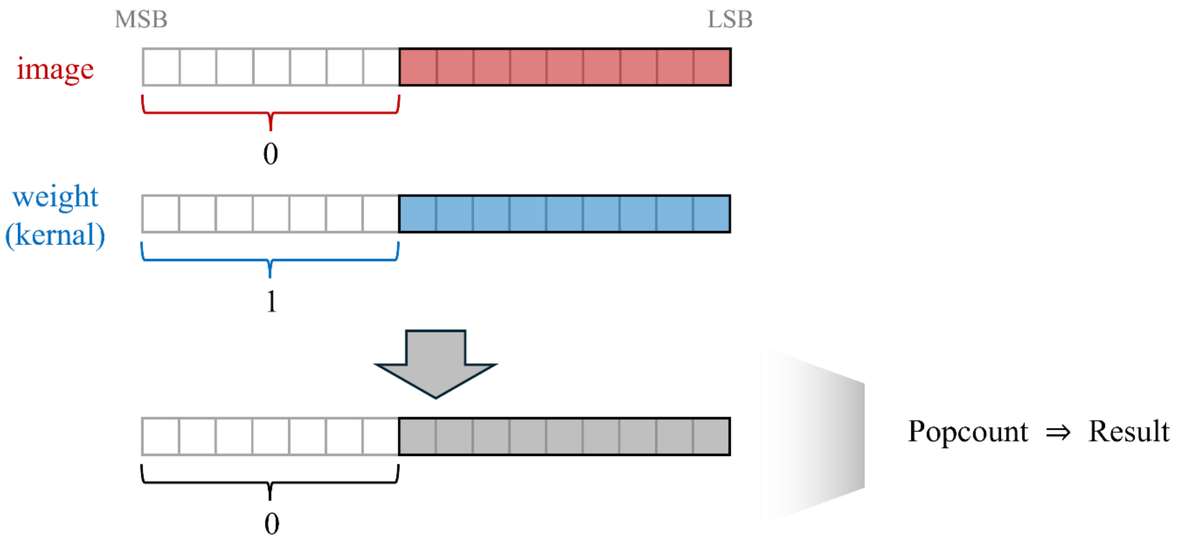
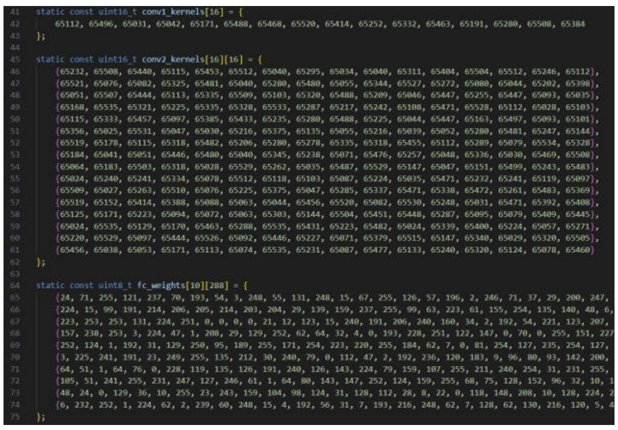


Figure 1 1 Filling Remaining Slots of uint16 without effecting popcount result

As a result, each 3x3 kernel region contains a total of 9 bits of data. For the image data, the upper 7 bits are padded with zeros, while for the weights, the upper 7 bits are padded with ones. This ensures that the XNOR operation produces a base value of -7. Consequently, the binary activation function in hardware is implemented with -7 as the threshold. This approach is applied specifically to `conv1w` and `conv2w`, whereas the `fc_weight` fully utilizes all 8 bits for packing.



```
static const uint16_t conv1_kernels[16] = {
    65112, 65496, 65031, 65042, 65171, 65488, 65468, 65520, 65414, 65252, 65332, 65463, 65191, 65280, 65508, 65384
};

static const uint16_t conv2_kernels[16][16] = {
    {65232, 65508, 65440, 65115, 65453, 65512, 65040, 65295, 65034, 65040, 65311, 65404, 65504, 65512, 65246, 65112},
    {65521, 65076, 65082, 65325, 65481, 65040, 65280, 65480, 65055, 65344, 65527, 65272, 65080, 65044, 65202, 65398},
    {65051, 65507, 65444, 65113, 65335, 65509, 65103, 65320, 65488, 65209, 65046, 65447, 65255, 65447, 65093, 65035},
    {65168, 65535, 65321, 65225, 65335, 65328, 65533, 65287, 65217, 65242, 65108, 65471, 65528, 65112, 65028, 65103},
    {65115, 65333, 65457, 65097, 65385, 65433, 65235, 65280, 65488, 65225, 65044, 65447, 65163, 65497, 65093, 65101},
    {65356, 65025, 65531, 65047, 65030, 65216, 65375, 65135, 65055, 65216, 65039, 65052, 65280, 65481, 65247, 65144},
    {65519, 65178, 65115, 65318, 65482, 65206, 65280, 65278, 65335, 65318, 65455, 65112, 65289, 65079, 65534, 65328},
    {65184, 65041, 65051, 65446, 65480, 65040, 65345, 65238, 65071, 65476, 65257, 65048, 65336, 65030, 65469, 65508},
    {65064, 65183, 65503, 65318, 65028, 65529, 65262, 65035, 65487, 65529, 65147, 65047, 65151, 65499, 65243, 65483},
    {65024, 65240, 65241, 65334, 65078, 65512, 65118, 65103, 65087, 65224, 65035, 65471, 65232, 65241, 65119, 65097},
    {65509, 65027, 65263, 65510, 65076, 65225, 65375, 65047, 65285, 65337, 65471, 65338, 65472, 65261, 65483, 65369},
    {65519, 65152, 65414, 65388, 65088, 65063, 65044, 65456, 65520, 65082, 65530, 65248, 65031, 65471, 65392, 65408},
    {65125, 65171, 65223, 65094, 65072, 65063, 65303, 65144, 65504, 65451, 65448, 65287, 65095, 65079, 65409, 65445},
    {65024, 65535, 65129, 65170, 65463, 65288, 65535, 65431, 65223, 65482, 65024, 65339, 65400, 65224, 65057, 65271},
    {65220, 65529, 65097, 65444, 65526, 65092, 65446, 65227, 65071, 65379, 65515, 65147, 65340, 65029, 65320, 65505},
    {65456, 65038, 65053, 65171, 65113, 65074, 65535, 65231, 65087, 65477, 65133, 65240, 65320, 65124, 65078, 65460}
};

static const uint8_t fc_weights[10][288] = {
    {24, 71, 255, 121, 237, 70, 193, 54, 3, 248, 55, 131, 248, 15, 67, 255, 126, 57, 196, 2, 246, 71, 37, 29, 200, 247,
    {224, 15, 99, 191, 214, 206, 205, 214, 203, 204, 29, 139, 159, 237, 255, 99, 63, 223, 61, 155, 254, 135, 140, 48, 6,
    {223, 253, 253, 131, 224, 251, 0, 0, 0, 0, 21, 12, 123, 15, 240, 191, 206, 240, 160, 34, 2, 192, 54, 221, 123, 207,
    {157, 238, 253, 3, 224, 47, 1, 208, 29, 129, 252, 62, 64, 32, 4, 0, 193, 228, 251, 122, 147, 0, 70, 0, 255, 151, 227
    {252, 124, 1, 192, 31, 129, 250, 95, 189, 255, 171, 254, 223, 220, 255, 184, 62, 7, 0, 81, 254, 127, 235, 254, 127,
    {3, 225, 241, 191, 23, 249, 255, 135, 212, 30, 240, 79, 0, 112, 47, 2, 192, 236, 120, 183, 9, 96, 80, 93, 142, 200,
    {64, 51, 1, 64, 76, 0, 228, 119, 135, 126, 191, 240, 126, 143, 224, 79, 159, 107, 255, 211, 240, 254, 31, 231, 255,
    {105, 51, 241, 255, 231, 247, 127, 246, 61, 1, 64, 80, 143, 147, 252, 124, 159, 255, 68, 75, 128, 152, 96, 32, 10, 1
    {48, 24, 0, 129, 36, 10, 255, 23, 243, 159, 104, 98, 124, 31, 128, 112, 28, 8, 22, 0, 118, 148, 208, 10, 128, 224, 2
    {6, 232, 252, 1, 224, 62, 2, 239, 60, 248, 15, 4, 192, 56, 31, 7, 193, 216, 248, 62, 7, 128, 62, 130, 216, 120, 5, 4
};
```

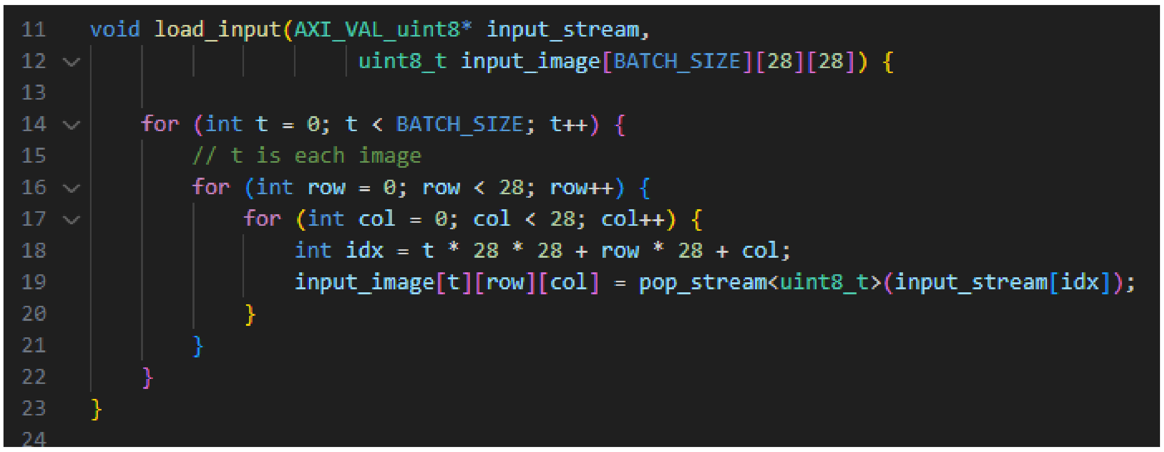
The weights, packed according to the defined rules, are hardcoded into a header file to maintain weight stationarity during computation. This ensures efficient access and reuse of weights throughout the operation.

III. Parallelization

1. Input Stream

The MNIST images are received directly using an input stream without any preprocessing. An input stream refers to a sequential flow of data, where each element (e.g., pixel values) is transmitted in a pipeline-like manner. This method is particularly efficient for hardware accelerators as it minimizes memory latency and allows real-time data ingestion. The input stream delivers grayscale MNIST images of size 28x28 as 8-bit unsigned integers (`uint8_t`).

The `load_input` function loads the data from the input stream into a 3D array `input_image` for further processing. The data from the stream is accessed using the `pop_stream` function, which extracts pixel values sequentially. The indices are computed based on the batch index (t), row, and column to correctly map the streamed data into the 3D array.



```
void load_input(AXI_VAL_uint8* input_stream,
                uint8_t input_image[BATCH_SIZE][28][28]) {

    for (int t = 0; t < BATCH_SIZE; t++) {
        // t is each image
        for (int row = 0; row < 28; row++) {
            for (int col = 0; col < 28; col++) {
                int idx = t * 28 * 28 + row * 28 + col;
                input_image[t][row][col] = pop_stream<uint8_t>(input_stream[idx]);
            }
        }
    }
}
```

Figure 1 2 Load input with Input Stream

- `AXI_VAL_uint8`: Represents the input stream data type for AXI interfaces.
- `pop_stream`: Extracts values from the stream one by one.

The indices ensure that the pixels are placed in the correct position in the `input_image` array for each batch, row, and column.

2. IM2COL

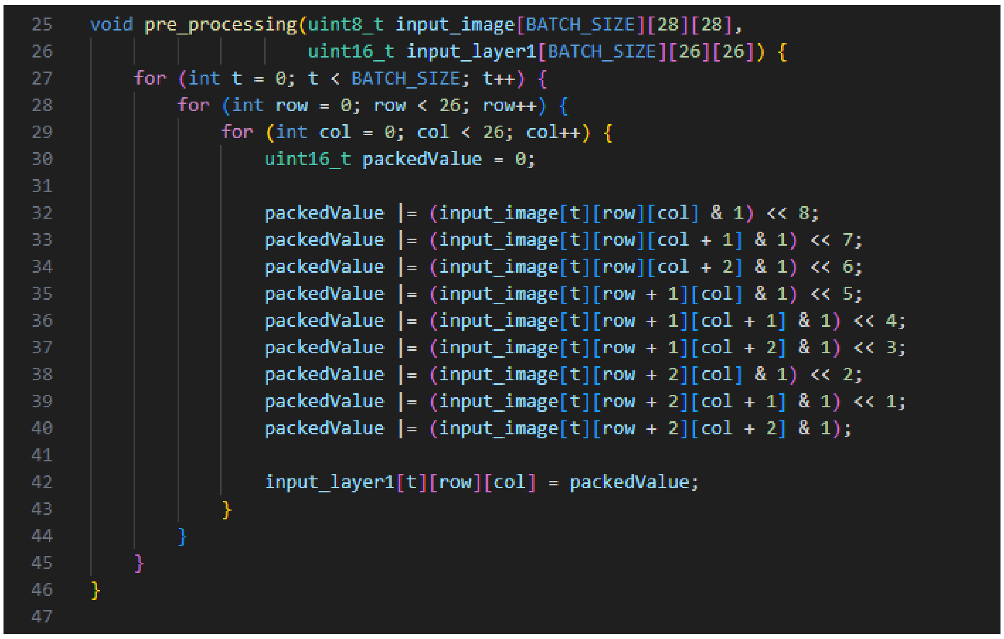
Once the MNIST images are loaded, they are preprocessed using the IM2COL technique to prepare for convolution. IM2COL stands for "Image-to-Column," which is a method that rearranges image data into a structured matrix to align with matrix multiplication. It converts overlapping regions of an image, based on the kernel size (3x3 in this case), into contiguous columns. This transformation enables parallel computation of convolution operations, particularly when used with matrix multiplication hardware accelerators.

In the `pre_processing` function, the 3x3 regions of each image are packed into a 9-bit representation and stored as a 16-bit value (`uint16_t`). Each bit in the packed value corresponds to a single pixel in the 3x3 window, shifted into the correct position:

- Bit 8 to Bit 0:
  Represent the 9 pixels of the 3x3 region, extracted using bitwise AND (&) and shifted using <<.

- This packed format reduces memory usage and accelerates subsequent XNOR operations in binary convolution layers.

The packed values are stored in a 3D array `input_layer1`, where each 3x3 window is represented as a single 16-bit value.



```c
void pre_processing(uint8_t input_image[BATCH_SIZE][28][28],
                    uint16_t input_layer1[BATCH_SIZE][26][26]) {
    for (int t = 0; t < BATCH_SIZE; t++) {
        for (int row = 0; row < 26; row++) {
            for (int col = 0; col < 26; col++) {
                uint16_t packedValue = 0;

                packedValue |= (input_image[t][row][col] & 1) << 8;
                packedValue |= (input_image[t][row][col + 1] & 1) << 7;
                packedValue |= (input_image[t][row][col + 2] & 1) << 6;
                packedValue |= (input_image[t][row + 1][col] & 1) << 5;
                packedValue |= (input_image[t][row + 1][col + 1] & 1) << 4;
                packedValue |= (input_image[t][row + 1][col + 2] & 1) << 3;
                packedValue |= (input_image[t][row + 2][col] & 1) << 2;
                packedValue |= (input_image[t][row + 2][col + 1] & 1) << 1;
                packedValue |= (input_image[t][row + 2][col + 2] & 1);

                input_layer1[t][row][col] = packedValue;
            }
        }
    }
}
```

Figure 1 3 Preprocessing of Input Image (IM2COL)

i. `input_image`:
   3D array holding the original MNIST images of size 28x28.

ii. `input_layer1`:
   3D array storing the packed 3x3 regions of size 26x26 (as the 3x3 kernel sliding window reduces the effective size).

iii. Bit Packing:
The least significant bit (LSB) of each pixel is extracted and shifted to its corresponding position in the 9-bit structure.

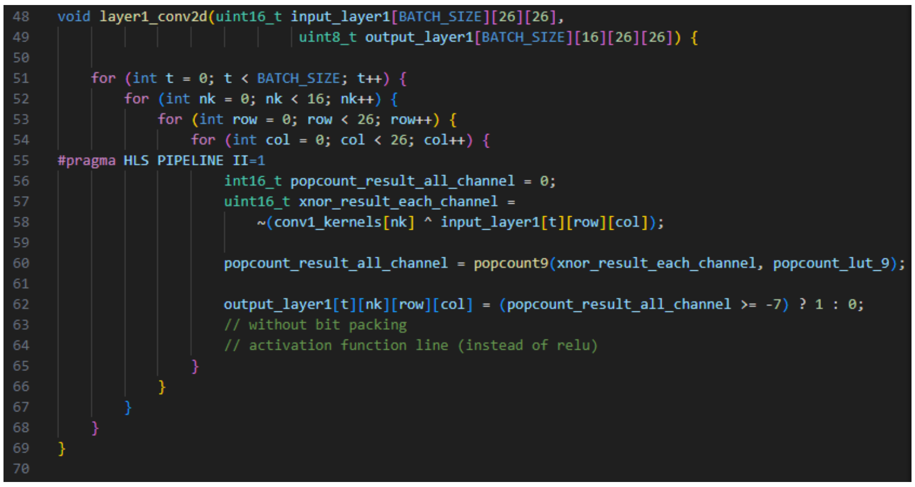
iv. Parallelization:
The code is "unrolled" to explicitly handle each pixel in the 3x3 window, enhancing readability and ensuring efficient parallel implementation.

3. Convolution Layer 1

In the first convolutional layer, computations are performed using XNOR and popcount operations, which are highly efficient for binary data processing. These operations accelerate the convolution process compared to conventional arithmetic operations. The `#pragma HLS PIPELINE` directive is applied to the innermost loop with an Initiation Interval (II) of 1, ensuring that operations are pipelined effectively.

Pipelining allows multiple operations to overlap, improving hardware throughput by processing new iterations before previous ones are fully completed. This is particularly important in the convolution layer, where the same computation (XNOR and popcount) is repeated for a large number of inputs and kernels. The loop is "flattened" and explicitly written to enhance hardware synthesis and maximize parallel execution.

In the following function, `popcount9` computes the number of matching bits from the XNOR result, and the activation function is applied to binarize the output values.

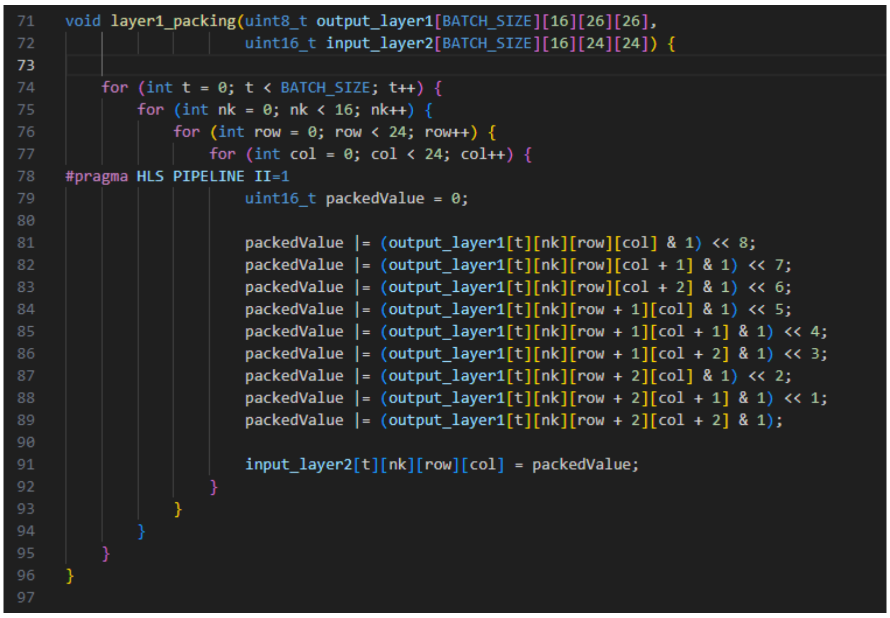


```c
void layer1_conv2d(uint16_t input_layer1[BATCH_SIZE][26][26],
                   uint8_t output_layer1[BATCH_SIZE][16][26][26]) {

    for (int t = 0; t < BATCH_SIZE; t++) {
        for (int nk = 0; nk < 16; nk++) {
            for (int row = 0; row < 26; row++) {
                for (int col = 0; col < 26; col++) {
#pragma HLS PIPELINE II=1
                    int16_t popcount_result_all_channel = 0;
                    uint16_t xnor_result_each_channel =
                        ~(conv1_kernels[nk] ^ input_layer1[t][row][col]);

                    popcount_result_all_channel = popcount9(xnor_result_each_channel, popcount_lut_9);

                    output_layer1[t][nk][row][col] = (popcount_result_all_channel >= -7) ? 1 : 0;
                    // without bit packing
                    // activation function line (instead of relu)
                }
            }
        }
    }
}

```

Figure 1 4 conv1 Layer computation

After passing through the first convolutional layer, the output values are packed into a `uint16_t` format using the IM2COL approach. This transformation reorganizes the binary output data into a compact structure, aligning it for the next convolutional layer.

IM2COL simplifies convolution into matrix multiplications, enabling efficient reuse of the same data and improving hardware throughput. The loop is unrolled and explicitly written to allow parallel computation for each 3x3 sliding window. Similar to the previous function, `#pragma HLS PIPELINE` is applied to ensure that the bit-packing operations are pipelined for maximum efficiency.



```
void layer1_packing(uint8_t output_layer1[BATCH_SIZE][16][26][26],
                    uint16_t input_layer2[BATCH_SIZE][16][24][24]) {

    for (int t = 0; t < BATCH_SIZE; t++) {
        for (int nk = 0; nk < 16; nk++) {
            for (int row = 0; row < 24; row++) {
                for (int col = 0; col < 24; col++) {
#pragma HLS PIPELINE II=1
                    uint16_t packedValue = 0;

                    packedValue |= (output_layer1[t][nk][row][col] & 1) << 8;
                    packedValue |= (output_layer1[t][nk][row][col + 1] & 1) << 7;
                    packedValue |= (output_layer1[t][nk][row][col + 2] & 1) << 6;
                    packedValue |= (output_layer1[t][nk][row + 1][col] & 1) << 5;
                    packedValue |= (output_layer1[t][nk][row + 1][col + 1] & 1) << 4;
                    packedValue |= (output_layer1[t][nk][row + 1][col + 2] & 1) << 3;
                    packedValue |= (output_layer1[t][nk][row + 2][col] & 1) << 2;
                    packedValue |= (output_layer1[t][nk][row + 2][col + 1] & 1) << 1;
                    packedValue |= (output_layer1[t][nk][row + 2][col + 2] & 1);

                    input_layer2[t][nk][row][col] = packedValue;
                }
            }
        }
    }
}
```

Figure 1 5 Output of conv1 Layer is packed into `uint16_t`.

4. Convolution Layer 2

The second convolutional layer is the most computationally expensive part of the network, as it processes all channels with a high number of operations. To address this bottleneck, significant optimization is applied using loop unrolling and pipelining techniques to maximize parallel execution and minimize latency.

- Loop Unrolling:
  The inner loop that iterates over the channels (`c = 0 to 15`) is unrolled with a factor of 16 using the `#pragma HLS UNROLL factor=16` directive. This aggressive level of unrolling allows simultaneous computation across all 16 channels, significantly reducing the number of clock cycles required. Compared to the implementation without unrolling, this optimization resulted in a 5x speedup.

- Pipelining:
  The second part of the layer performs average pooling over a 2x2

region. To further optimize latency, `#pragma HLS PIPELINE II=1` is applied to ensure that new iterations of the pooling loop begin every clock cycle, maintaining a steady flow of data.

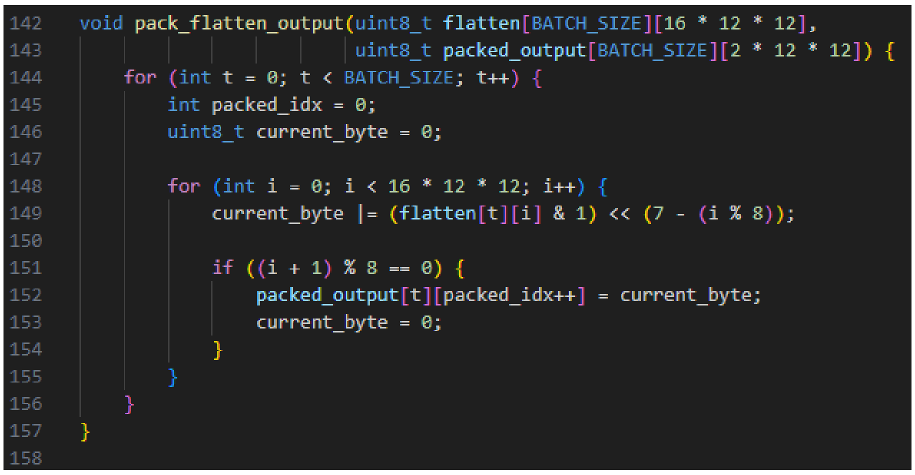
After average pooling, the output values are flattened into a 1D array to prepare for the fully connected layer. This flattening process ensures the data is stored sequentially, ready for further processing.

5. Packing Flatten Output

After the average pooling and flattening stages, the output data is packed into a more compact format using 8-bit containers (`uint8_t`). This process involves filling each byte starting from the most significant bit (MSB) with consecutive binary values from the flattened data. The goal is to reduce the memory footprint and prepare the output for efficient transfer or further computation.

The function `pack_flatten_output` iterates through the flattened data and shifts the binary values into their respective bit positions within an 8-bit `current_byte`. Once all 8 bits of the `current_byte` are filled, the byte is stored in the output array `packed_output`.

Given that the packing process is relatively lightweight in terms of computation time, no further optimizations, such as pipelining or unrolling, were applied.

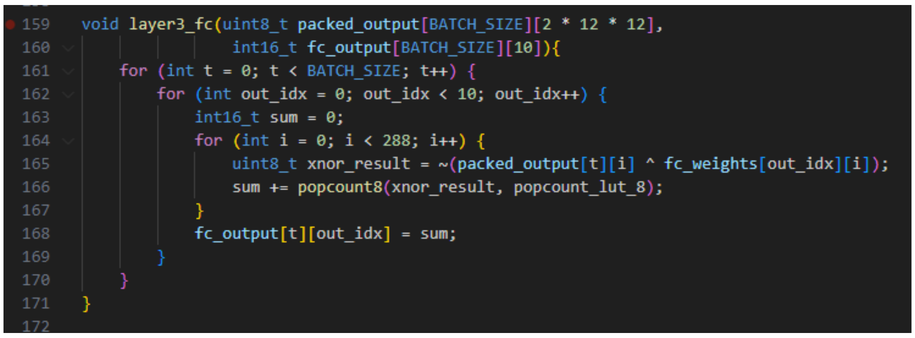


```
void pack_flatten_output(uint8_t flatten[BATCH_SIZE][16 * 12 * 12],
                         uint8_t packed_output[BATCH_SIZE][2 * 12 * 12]) {
    for (int t = 0; t < BATCH_SIZE; t++) {
        int packed_idx = 0;
        uint8_t current_byte = 0;

        for (int i = 0; i < 16 * 12 * 12; i++) {
            current_byte |= (flatten[t][i] & 1) << (7 - (i % 8));

            if ((i + 1) % 8 == 0) {
                packed_output[t][packed_idx++] = current_byte;
                current_byte = 0;
            }
        }
    }
}
```

6. Fully Connected Layer

The fully connected (FC) layer is implemented to perform the final classification stage of the network. This layer computes the weighted sum of the binary-packed input data using XNOR and popcount operations, which are efficient for binary neural networks. Although this design considers optimization, additional resource constraints, such as memory usage, were prioritized, leading to a straightforward implementation.

Compared to earlier stages (convolution and pooling), the fully connected layer requires relatively less computation time and is not a critical performance bottleneck. Therefore, further optimizations like pipelining or unrolling were not applied.



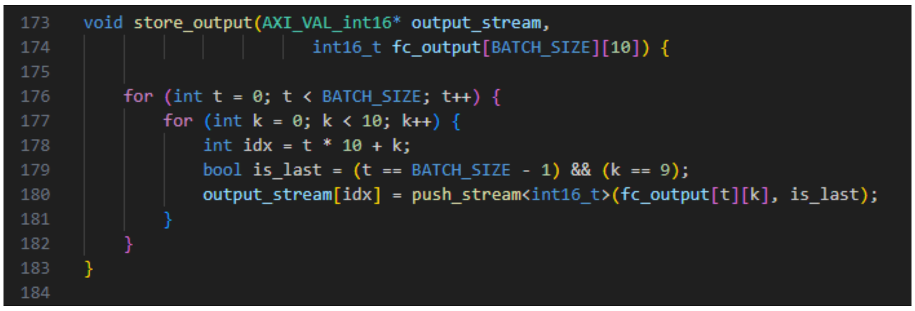
```
void layer3_fc(uint8_t packed_output[BATCH_SIZE][2 * 12 * 12],
               int16_t fc_output[BATCH_SIZE][10]){
    for (int t = 0; t < BATCH_SIZE; t++) {
        for (int out_idx = 0; out_idx < 10; out_idx++) {
            int16_t sum = 0;
            for (int i = 0; i < 288; i++) {
                uint8_t xnor_result = ~(packed_output[t][i] ^ fc_weights[out_idx][i]);
                sum += popcount8(xnor_result, popcount_lut_8);
            }
            fc_output[t][out_idx] = sum;
        }
    }
}
```

7. Output Stream

The final output of the fully connected layer is sent out using an output stream interface, which efficiently transmits data in a sequential manner. This approach aligns with AXI-stream protocols often used in hardware systems, facilitating real-time and resource-efficient data communication.

The `store_output` function iterates through the output data (`fc_output`), which contains the classification results for each batch and outputs the values into an AXI-compatible stream (`output_stream`).

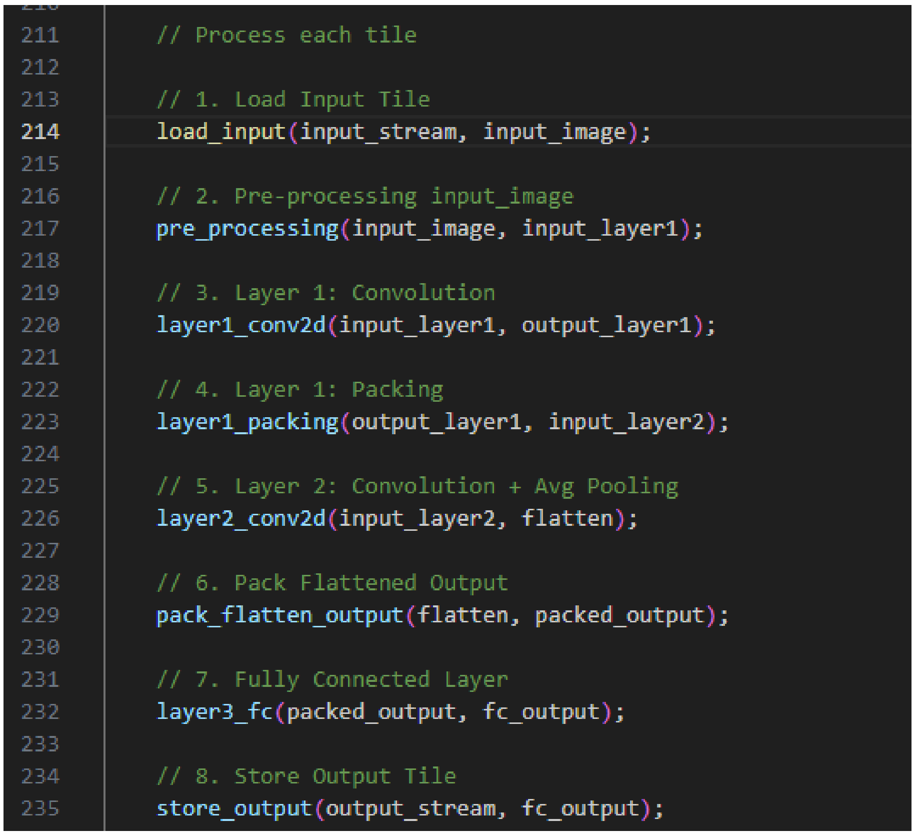
The `push_stream` function wraps the data and appends an end-of-stream (EOS) flag for the last value in the stream, signaling the completion of data transfer.



```
void store_output(AXI_VAL_int16* output_stream,
                  int16_t fc_output[BATCH_SIZE][10]) {

    for (int t = 0; t < BATCH_SIZE; t++) {
        for (int k = 0; k < 10; k++) {
            int idx = t * 10 + k;
            bool is_last = (t == BATCH_SIZE - 1) && (k == 9);
            output_stream[idx] = push_stream<int16_t>(fc_output[t][k], is_last);
        }
    }
}
```

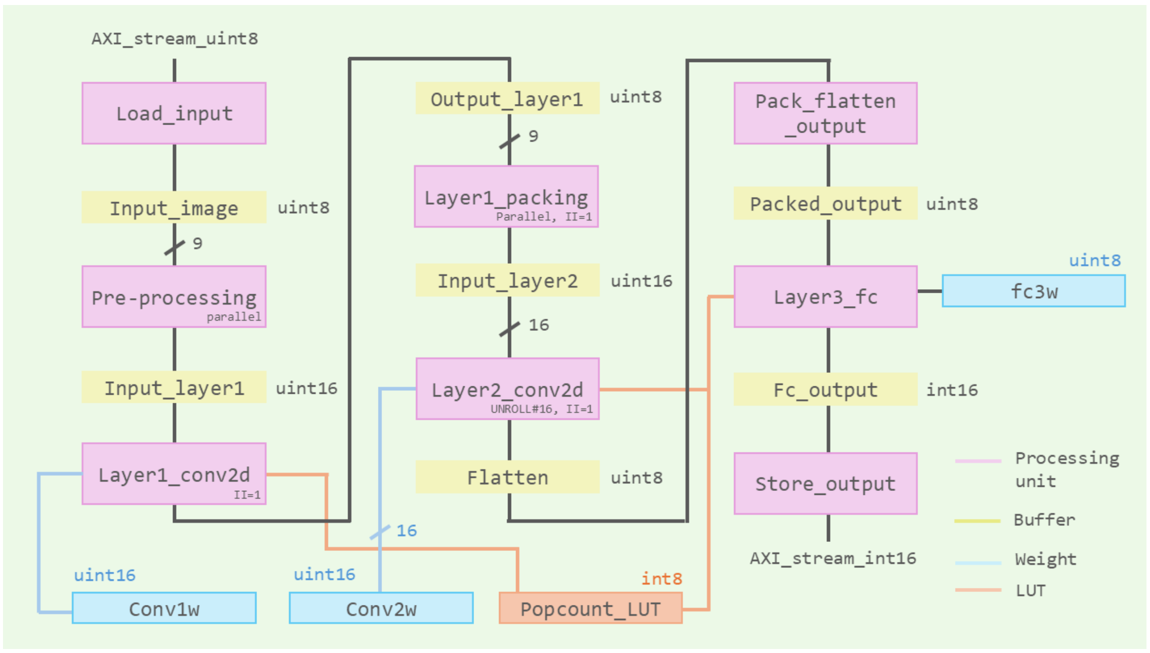
8. Overall

Overall, the structure of the top module is shown below, where array partitioning of buffers is applied to facilitate parallel access during computation.



```
// Process each tile

// 1. Load Input Tile
load_input(input_stream, input_image);

// 2. Pre-processing input_image
pre_processing(input_image, input_layer1);

// 3. Layer 1: Convolution
layer1_conv2d(input_layer1, output_layer1);

// 4. Layer 1: Packing
layer1_packing(output_layer1, input_layer2);

// 5. Layer 2: Convolution + Avg Pooling
layer2_conv2d(input_layer2, flatten);

// 6. Pack Flattened Output
pack_flatten_output(flatten, packed_output);

// 7. Fully Connected Layer
layer3_fc(packed_output, fc_output);

// 8. Store Output Tile
store_output(output_stream, fc_output);
```

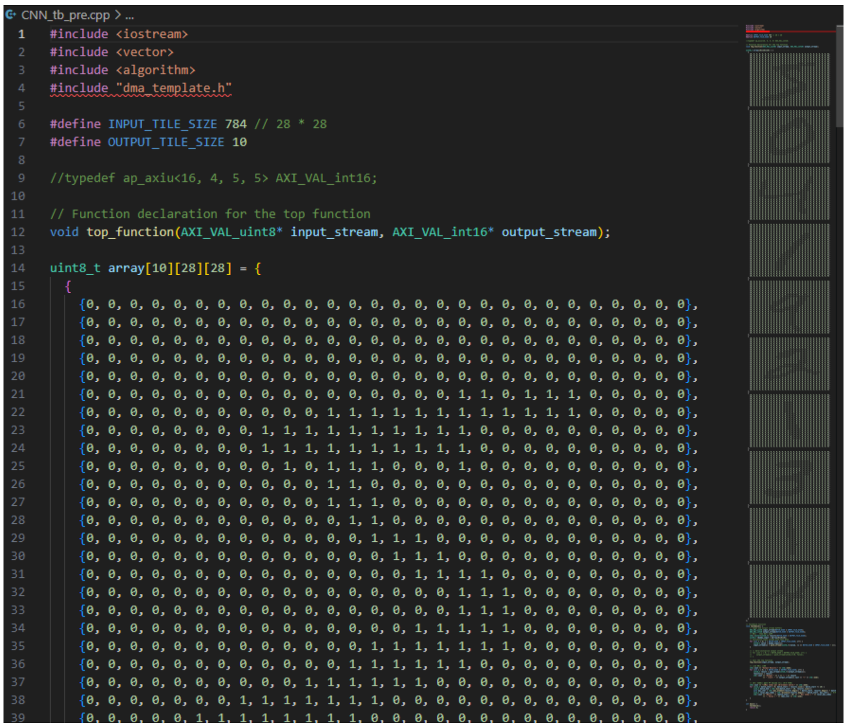
9. Hardware Dataflow and Structure



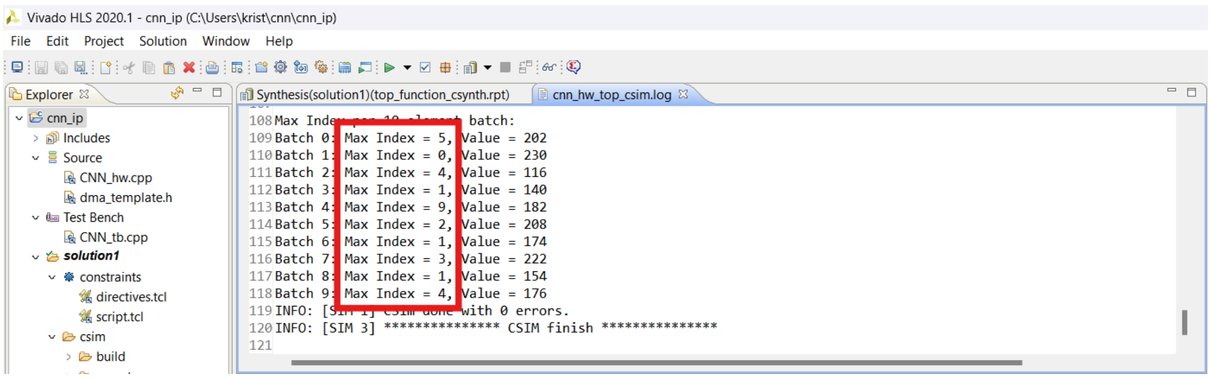
IV. Hardware Synthesis

1. C-simulation

The testbench is set as below. (See the number patterns on the right.)



```
#include <iostream>
#include <vector>
#include <algorithm>
#include "dma_template.h"

#define INPUT_TILE_SIZE 784 // 28 * 28
#define OUTPUT_TILE_SIZE 10

//typedef ap_axiu<16, 4, 5, 5> AXI_VAL_int16;

// Function declaration for the top function
void top_function(AXI_VAL_uint8* input_stream, AXI_VAL_int16* output_stream);

uint8_t array[10][28][28] = {
  {
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
```

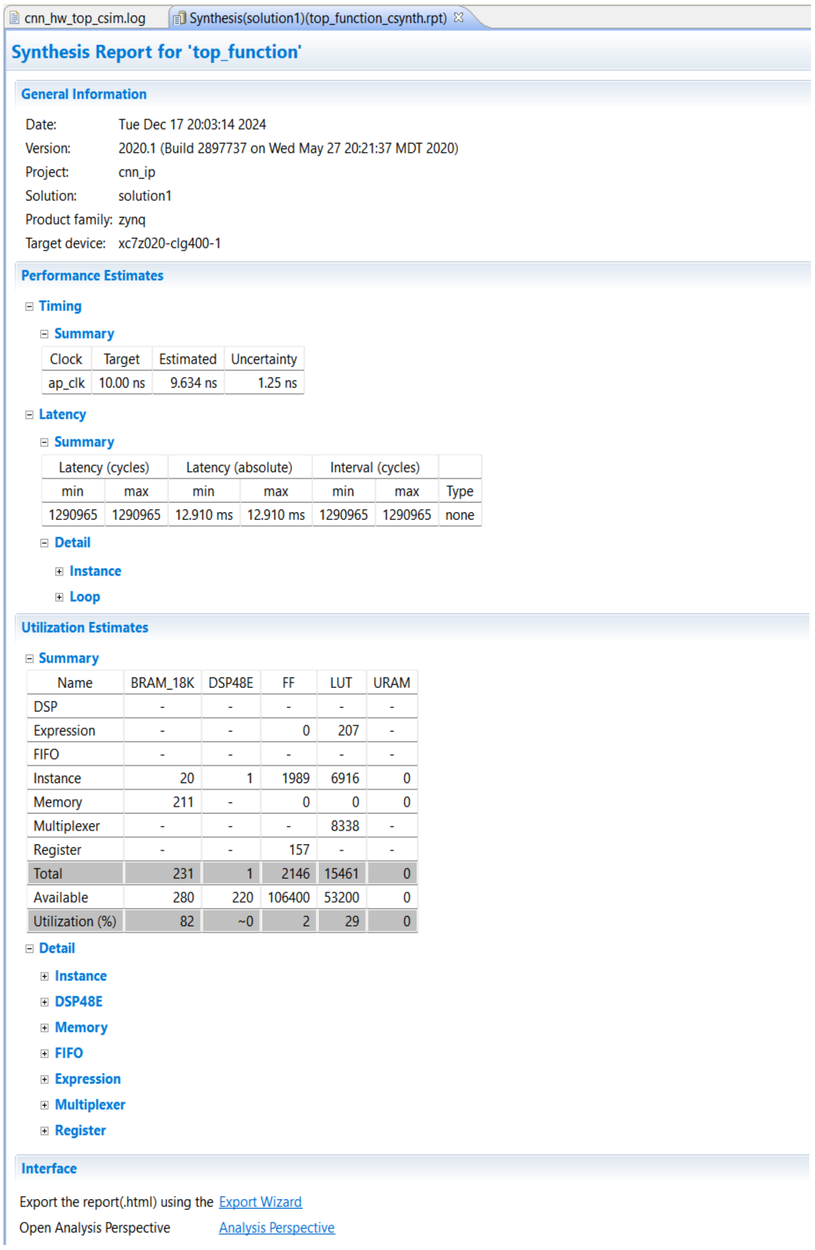


```
Max Index per 10-element batch:
Batch 0: Max Index = 5, Value = 202
Batch 1: Max Index = 0, Value = 230
Batch 2: Max Index = 4, Value = 116
Batch 3: Max Index = 1, Value = 140
Batch 4: Max Index = 9, Value = 182
Batch 5: Max Index = 2, Value = 208
Batch 6: Max Index = 1, Value = 174
Batch 7: Max Index = 3, Value = 222
Batch 8: Max Index = 1, Value = 154
Batch 9: Max Index = 4, Value = 176
INFO: [SIM 1] CSim done with 0 errors.
INFO: [SIM 3] *************** CSIM finish ***************
```

The results show that Max Index matches the number shown in each input.

2. C synthesis

After synthesizing the ip, the Synthesis Report shows:



cnn_hw_top_csim.log | Synthesis(solution1)(top_function_csynth.rpt)

Synthesis Report for 'top_function'

General Information

Date:	Tue Dec 17 20:03:14 2024
Version:	2020.1 (Build 2897737 on Wed May 27 20:21:37 MDT 2020)
Project:	cnn_ip
Solution:	solution1
Product family:	zynq
Target device:	xc7z020-clg400-1

Performance Estimates

⊟ **Timing**

⊟ **Summary**

Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	9.634 ns	1.25 ns

⊟ **Latency**

⊟ **Summary**

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
1290965	1290965	12.910 ms	12.910 ms	1290965	1290965	none

⊟ **Detail**

⊞ **Instance**

⊞ **Loop**

Utilization Estimates

⊟ **Summary**

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	207	-
FIFO	-	-	-	-	-
Instance	20	1	1989	6916	0
Memory	211	-	0	0	0
Multiplexer	-	-	-	8338	-
Register	-	-	157	-	-
Total	231	1	2146	15461	0
Available	280	220	106400	53200	0
Utilization (%)	82	~0	2	29	0

⊟ **Detail**

⊞ **Instance**

⊞ **DSP48E**

⊞ **Memory**

⊞ **FIFO**

⊞ **Expression**

⊞ **Multiplexer**

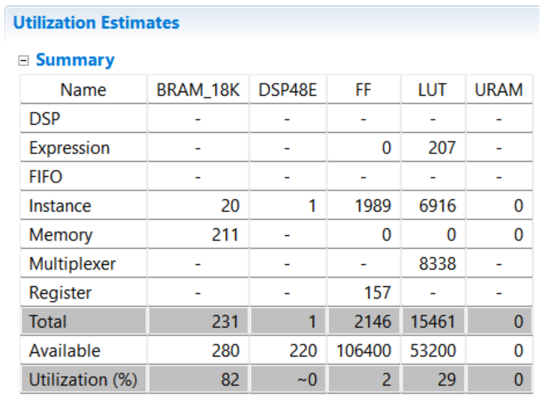
⊞ **Register**

Interface

Export the report(.html) using the Export Wizard

Open Analysis Perspective Analysis Perspective

- Clock Frequency: 10.00ns (estimated: 9.634ns)
- Resource Utilization:



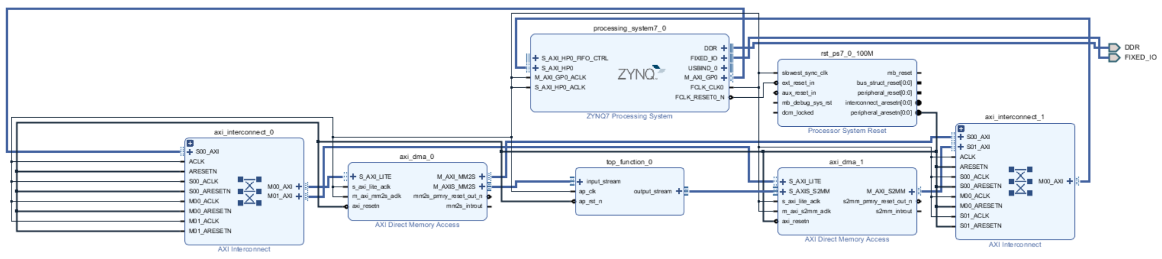
Utilization Estimates

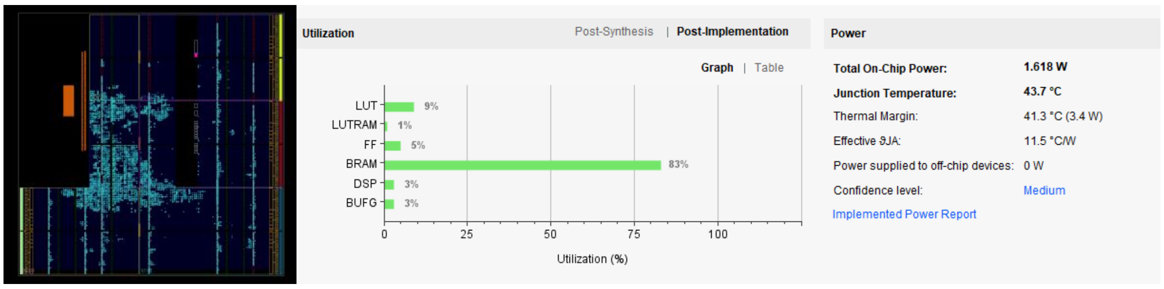
⊟ **Summary**

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	207	-
FIFO	-	-	-	-	-
Instance	20	1	1989	6916	0
Memory	211	-	0	0	0
Multiplexer	-	-	-	8338	-
Register	-	-	157	-	-
Total	231	1	2146	15461	0
Available	280	220	106400	53200	0
Utilization (%)	82	~0	2	29	0

- # of Cycles: 1,292,965 (12.910 ms)

3. Vivado Top diagram



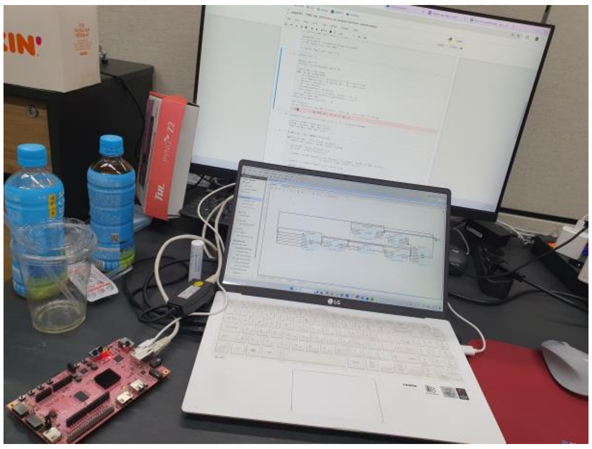


Power

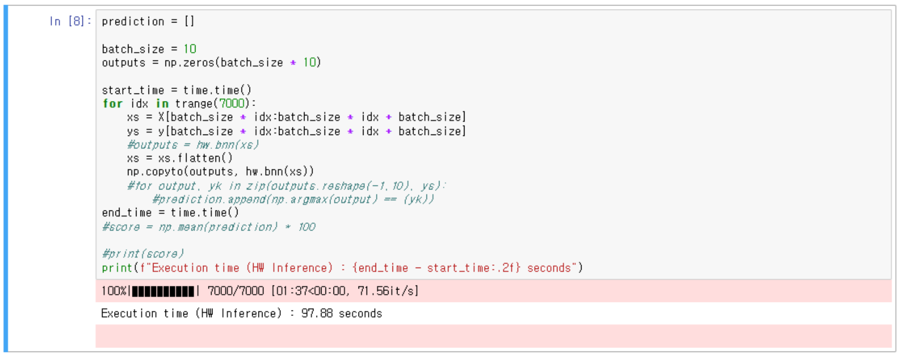
Total On-Chip Power:	**1.618 W**
Junction Temperature:	**43.7 °C**
Thermal Margin:	41.3 °C (3.4 W)
Effective ϑJA:	11.5 °C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium

Implemented Power Report

4. Execution on PYNQ

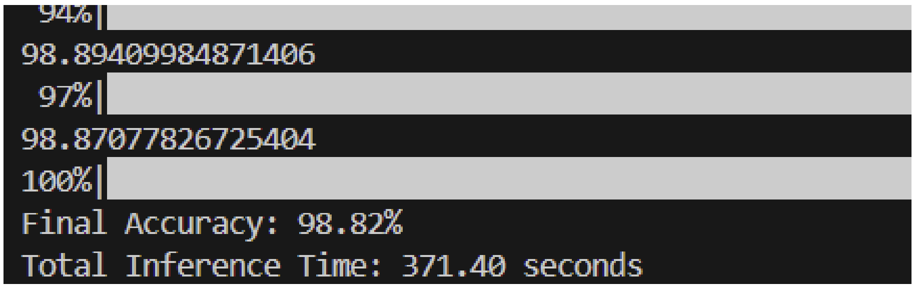


Running on PYNQ, the time took for execution of all 70,000 images was 97.88 seconds



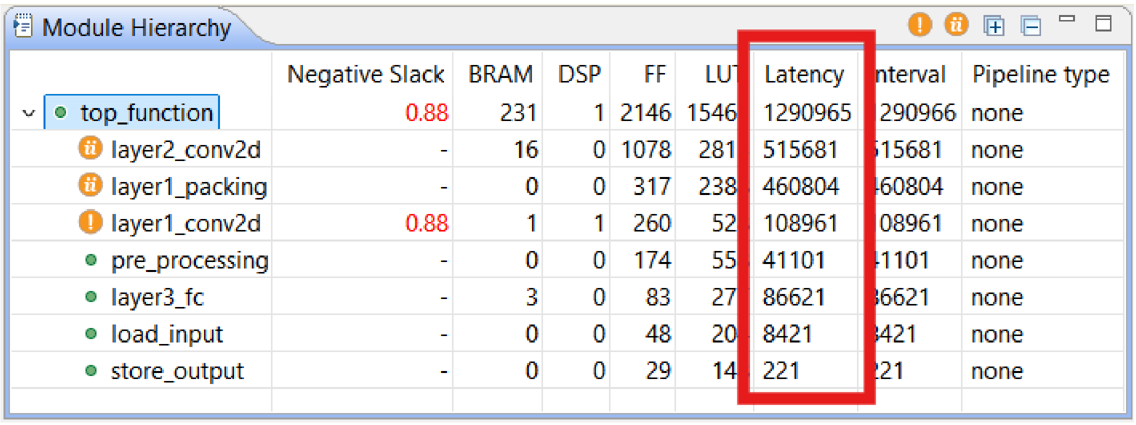
```
In [8]: prediction = []

        batch_size = 10
        outputs = np.zeros(batch_size * 10)

        start_time = time.time()
        for idx in trange(7000):
            xs = X[batch_size * idx:batch_size * idx + batch_size]
            ys = y[batch_size * idx:batch_size * idx + batch_size]
            #outputs = hw.bnn(xs)
            xs = xs.flatten()
            np.copyto(outputs, hw.bnn(xs))
            #for output, yk in zip(outputs.reshape(-1, 10), ys):
                #prediction.append(np.argmax(output) == (yk))
        end_time = time.time()
        #score = np.mean(prediction) * 100

        #print(score)
        print(f"Execution time (HW Inference) : {end_time - start_time:.2f} seconds")

        100%|██████████| 7000/7000 [01:37<00:00, 71.56it/s]
        Execution time (HW Inference) : 97.88 seconds
```

Figure 1 6 Execution time of TPU on PYNQ while FP32 CPU took 371.40 seconds.



```
 94%|
98.89409984871406
 97%|
98.87077826725404
100%|
Final Accuracy: 98.82%
Total Inference Time: 371.40 seconds
```

Figure 1 7 Accuracy and Runtime of FP32 CPU



Module Hierarchy

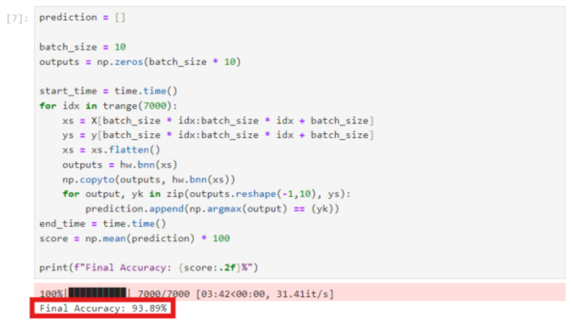
	Negative Slack	BRAM	DSP	FF	LU	Latency	nterval	Pipeline type
top_function	0.88	231	1	2146	1546	1290965	290966	none
layer2_conv2d	-	16	0	1078	281	515681	15681	none
layer1_packing	-	0	0	317	238	460804	60804	none
layer1_conv2d	0.88	1	1	260	52	108961	08961	none
pre_processing	-	0	0	174	55	41101	1101	none
layer3_fc	-	3	0	83	27	86621	6621	none
load_input	-	0	0	48	20	8421	421	none
store_output	-	0	0	29	14	221	21	none

Runtime breakdown:

- Load Input (8,421): 0.65%
- Pre-processing (41,101): 3.18%
- Conv1 (108,961): 8.44%
- Conv1 Packing (460,804): 35.69%
- Conv2 (515,681): 39.95%
- Fully Connected (86,621): 6.71%
- Store Output (221): 0.02%

V. Discussion

In our project, the hardware implementation achieved an accuracy of 93.89%, which is slightly lower than the 98.82% accuracy obtained on the FP32 CPU implementation. However, the execution time for our hardware was 97.88 seconds, significantly reducing the FP32 inference time of 371.40 seconds by approximately 73.65%. On the PYNQ board, the FP32 inference took even longer, requiring about 3.5 hours, which underscores the efficiency of our hardware implementation.



```
[7]: prediction = []

     batch_size = 10
     outputs = np.zeros(batch_size * 10)

     start_time = time.time()
     for idx in trange(7000):
         xs = X[batch_size * idx:batch_size * idx + batch_size]
         ys = y[batch_size * idx:batch_size * idx + batch_size]
         xs = xs.flatten()
         outputs = hw.bnn(xs)
         np.copyto(outputs, hw.bnn(xs))
         for output, yk in zip(outputs.reshape(-1,10), ys):
             prediction.append(np.argmax(output) == (yk))
     end_time = time.time()
     score = np.mean(prediction) * 100

     print(f"Final Accuracy: {score:.2f}%")
```

```
100%|██████████| 7000/7000 [03:42<00:00, 31.41it/s]
Final Accuracy: 93.89%
```

Figure 1 8 Final Accuracy of BNN on PYNQ board

Despite these improvements, there are still areas for further optimization. As mentioned earlier, the sign function was approximated using the STE-based tanh function, which can introduce errors for values close to zero due to the approximation's inherent nonlinearity. This limitation may slightly affect the final accuracy of the network.

Additionally, we used the C++ standard data types `uint8` and `uint16` for activation and weight storage. This approach was suboptimal for the 9-bit binary activations and weights, as the design used 16 bits for storage despite requiring only 9 bits. This resulted in inefficient resource utilization. By replacing the standard types with `ap_uint<>`, we expect to achieve better utilization of hardware resources, reducing unnecessary overhead and improving efficiency.

Overall, while our implementation demonstrates substantial improvements in inference time compared to FP32, future work addressing data type efficiency and approximation accuracy can further enhance the performance and resource utilization of the design.

VI. Role & contributions of each team member

20201082 H. Kim

- Design and implementation of hardware architecture, including HLS coding, functional debugging, and performance optimization
- Development of the initial BNN software pipeline, including training and inference framework setup
- Project management through schedule planning, role distribution, and overall task coordination

20211184 H. Yang

- BNN model optimization through parameter tuning (threshold adjustments) to achieve high-accuracy weight extraction
- Development of auxiliary scripts for HW implementation, including LUT value formatting and bit-packing automation
- Refining final report into the completed version, creating project presentation (PPT) and drafting the presentation script