1. AI(SW) design
   1. Quantization (BNN)

We decided to implement BNN. To do so, we binarize the weights into -1 and 1. For the values less or equal to 0, it goes to -1. Originally, the 0 should be set to 1. However, the float point error causes error when implementing this on to the C. So, we use the special binarization method to set 0 as -1.

|  |
| --- |
|  |
| [Binarize function] |

Using this, generates the model with the same structure but binarized to train it.

|  |  |
| --- | --- |
|  |  |
| [Binarize function] | |
|  | |
| [Binarized model] | |

When validate the output of the model, input also binarized, if the value is greater than 0.5, it goes to the 1, otherwise it goes to the 0.

|  |
| --- |
| 텍스트, 스크린샷, 폰트, 라인이(가) 표시된 사진  AI 생성 콘텐츠는 정확하지 않을 수 있습니다. |
| [Binarize input] |

Comparing with the FP32 result, the accuracy of quantized model is better than FP32 model.

|  |  |
| --- | --- |
|  |  |
| [left(FP32) right(BNN)] | |

Also, when checking the model removing the bias increases the accuracy, so we decided to remove bias.

|  |
| --- |
|  |
| [Accuracy of different kinds] |

1. Hardware design
   1. Utilize bit operations

The weight and processing data is binarized to -1 and 1 because we use BNN, However, in the case of first layer, the weight is binarized into -1 and 1 while input is binarized into 0 and 1. To handle this, we utilized bitwise and operation and not operation.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Input** | **Weight** | **Result** | **Input&Weight** | **Input&~Weight** |
| 0 | 0 | 0\*(-1) = 0 | 0&0=0 | 0&1=0 |
| 0 | 1 | 0\*1 = 0 | 0&1=0 | 0&0=0 |
| 1 | 0 | 1\*(-1) = -1 | 1&0=0 | 1&1=1 |
| 1 | 1 | 1\*1 = 1 | 1&1=1 | 1&0=0 |

As we can see the number of 1 in the result of bitwise operation Input&weight means the number of 1s in the convolution summation. Likewise, the number of 1 in the result of bitwise operation Input&~weight means the number of -1s in the convolution summation. Therefore, if we compare the number of 1s between two results, the sign of result is generated.

Other layer is binarized into -1 and 1, we can use the xor and not operation.

|  |  |  |  |
| --- | --- | --- | --- |
| **Input** | **Weight** | **Result** | **~(Input^Weight)** |
| 0 | 0 | (-1)\*(-1) = 1 | ~(0^0)=1 |
| 0 | 1 | (-1)\*1 = -1 | ~(0^1)=0 |
| 1 | 0 | 1\*(-1) = -1 | ~(1^0)=0 |
| 1 | 1 | 1\*1 = 1 | ~(1^1)=1 |

The number of 1s in the result means the 1 in the convolution summation, while number of 0s in the result means the 0 in the convolution summation. In this manner, we can interpret the result with popcount: the number of 1s in the bit-representation.

For example, suppose the bit representation is 10001001. Then the result is 1 \* 3 + (-1) \* 5 = -2. We hardcoded the result of this for 0~255 and use it to calculate the result.

For the average pooling, if the number of 1s in bit-representation for 4 inputs is greater than 3, the average is greater than 0, result should be 1. To handle this, we used bit operation again. Suppose each input is a, b, c, d. Then, (a & b & c) | (a & b & d) | (a & c & d) | (b & c & d) is 1 when 3 or 4 of inputs are 1 otherwise 0.

* 1. Input packing

For input, we also binarize it again, and packed each image into 25 32-bit numbers. Originally, the value ranges from 0 to 255. So, if the value is greater than 128, it converted into 1, otherwise 0. This can be implemented by shifting the value to right by 7. For each 32 bit in the input, we packed into 32 bit integer where lowest bit indicates the first bit. This is because we want to avoid a minus operation. If we pack the first bit into highest bit, the bit access should be like w[31-j] for j-th bit of input. Also, since 784 = 32 \* 24 + 16, we added a padding of 16bit to ensure all input size are equal.

|  |
| --- |
|  |
| [Input packing] |

* 1. Output packing

In order to use bit operation efficiently, we used a packing to each output of conv layers. We packed the output with (channel \* width \* height) into channel-bit width \* height array. For example, the output of first convolution layer is packed into 26 \* 26 array with data type ap\_uint<16>.

|  |
| --- |
|  |
| [Output packing] |

* 1. Weight packing

We packed the weight by input channels. For example, if the shape of weight is (32, 16, 3, 3), we packed it into (32, 3, 3) with data type ap\_uint<16>. By combining this with bit operation, we remove the loop that runs for the input channel since the calculation for all input channels is done by just bit operations. And the pseudo code for input (C, H, W) and weight(O, kH, kW) is reduced as follows.

|  |
| --- |
| For row = 0; row < H; row++      For col = 0; col < W; col++          If row >= kH - 1 and col >= kW - 1              For oc = 0; oc < O; oc++                  res = 0                  For k = 0; k < kH; k++                      For l = 0; l < kW; l++                          bitwiseHandle(input[row + k][col + l], weight[oc][k][l])                      end For                  end For                 output[row][col].range(oc, oc) = res              end For          end If      end For  end For |
| [pseudo code for reduced CNN convolution] |

output[row][col].range(oc, oc) indicates oc-th bit in output[row][col].

* 1. Sliding window & stream

To reduce memory usage and enable efficient pipelining in stream-based convolution, we adopted a sliding window and line buffer approach. Let kH and kW denote the height and width of the convolution kernel. The line buffer stores kH rows of the input, each row having the same width as the input image.

During each iteration over the input columns, new input values are streamed in and stored in the line buffer. The window, which is of size kH × kW, shifts left by one column, and the empty rightmost column is filled using the corresponding values from the line buffer based on the current input position.

This approach ensures that only the necessary data for computing the current convolution output is stored and reused, minimizing memory usage while allowing for efficient and continuous data processing. Once the window is fully updated, the convolution can be computed using only the values inside the window.

|  |
| --- |
|  |
| [Using Sliding window and stream for demanding storing of the input] |

* 1. Combine into HLS (Implement Parallelism)
     1. Conv1

|  |
| --- |
| 텍스트, 스크린샷, 소프트웨어, 폰트이(가) 표시된 사진  AI 생성 콘텐츠는 정확하지 않을 수 있습니다. |
| [Upper part of conv1] |

For the upper part, reads the 32bit integer from the input stream for each 32 iterations, Applied the pipeline for iterations to increase the throughput. Set the target row for line buffer. Originally, it was ii mod 3. However, the modular operation consumes many cycles. So, we hard-coded of mod 3 for possible ii values. In the similar reason, we substitute the cnt mod 32 to bit operations. Then shift the window and set the values of the last column of the window with the line buffer value and read value. Applied the unroll to shifting window for increasing parallelism. Applied the array partition to the linebuf, win, and trow for multiple accessing.

|  |
| --- |
| 텍스트, 스크린샷이(가) 표시된 사진  AI 생성 콘텐츠는 정확하지 않을 수 있습니다. |
| [Lower part of conv1] |

The window is fully charged only if ii >=2 and jj >=2. So, calculate only ii >= 2 and jj >= 2 Also, we applied the unroll manually because the outer loop has pipelined. Vivado synthesis ignores the unroll due to it. conv1w\_bin and conv1w\_bin\_neg is hardcoded value of the weight of the conv1 layer. For efficiently, we packed one more time. So, its shape is not the array shaped (16, 3, 3) with bit1 data type. It is array shaped (3, 3) with ap\_uint<16> data type. Therefore, conv1w\_bin[r][c][f] indicates weight[f][r][c] for the pseudo code. Apply bit operations to the inputs and weights and pack the results into ap\_uint<16> data type. Send the packed result to the stream. Note that unroll is applied to increase parallelism. Although the partition isn’t applied to the conv1w, it automatically parallelized according to the schedule viewer. So, we don’t apply the array partition to reduce the amount of the LUT.

|  |
| --- |
| 텍스트, 스크린샷, 평행이(가) 표시된 사진  AI 생성 콘텐츠는 정확하지 않을 수 있습니다. |
| parallelized without the paritioning] |
|  |
| [Packed weight of conv1] |

* + 1. Conv2

|  |
| --- |
|  |
| [Upper part of conv2] |
|  |

The upper part of the conv2 is similar to the conv1. The change is that it reads from the stream for every iteration because the one output of the previous layer is one pixel for this time. Note that II size is 3 due to LUT and BRAM usage control. It also runs 4 channels at a single time like the previous layer. Hence, the array partition is applied to the weights. Note that array partition is only applied to dimension 1 because the operations are already parallelized well without partitioning dimension 2 and 3.

|  |
| --- |
|  |
| [parallelized without the paritioning]    [Lower part of conv2] |
|  |
| [popcount\_table] |

The Lower part is also similar to the conv1. It unrolled manually and used xnor and popcount to calculate the output. And applied unroll to the inner loops to increase parallelism. Popcount\_table is hard-coded in the header file from 0 to 255.

* + 1. Conv3

|  |
| --- |
|  |
| [Upper part of conv3] |

The upper part of the conv3 is similar to the upper part of conv2. Note that II is set to the 4 because of the LUT and BRAM usage control. Also, the array partition is only applied to dimension 1 because the operations are already parallelized well without partitioning dimension 2 and 3 like previous layers.

|  |
| --- |
| 텍스트, 스크린샷, 평행, 디자인이(가) 표시된 사진  AI 생성 콘텐츠는 정확하지 않을 수 있습니다. |
| [parallelized without the paritioning]    [Lower part of conv3] |

The Lower part is same as with the conv2 except that the bit range is increased to 32.

* + 1. Pool + Flatten

|  |
| --- |
| 텍스트, 스크린샷, 폰트이(가) 표시된 사진  AI 생성 콘텐츠는 정확하지 않을 수 있습니다. |
| [Pooling layer] |

Since the input is packed into 32bit, our pooling method processes the 32 channels at once. Pipeline is introduced to increase the throughput. In binarized method, three or more of four bits indicates the result of average pooling is 1. So, bit operation for checking whether three or more than four bits is 1 is applied. Also, the packing order of the output is different from the actual flattening of the result. So, the packing of the next layer is different.

|  |
| --- |
| 라인, 도표, 스크린샷, 그래프이(가) 표시된 사진  AI 생성 콘텐츠는 정확하지 않을 수 있습니다. |
| [Pooling packing] |

* + 1. FC

|  |
| --- |
|  |
| [fc layer] |

Fc layer also uses the popcount and XNOR operation. But the packing of the fcwT weight is silghtly different. Since the first 32 bit of s4 is actually (0, 0, 0), (1, 0, 0), … (31, 0, 0) in actual model code in the python before flattening, The first bit of fcwT[k][0] should be originally fc\_w.T[0][k], second bit is fc\_w.T[121][k], …. Likewise, the first bit of fcwT[k][1] should be originally fc\_w.T[1][k], second bit is fc\_w.T[122][k] ….. By packing like this, we can use the XNOR operation to calculate fast. Although

* + 1. data\_out

|  |
| --- |
|  |
| [data\_out] |

Just write the result to the DMA. Make sure that mark last at the index batch\_size \* 10 – 1

* + 1. network(top function)

|  |
| --- |
|  |
| [network] |

Defines the streams connects the layers. Make sure use a pragma DATAFLOW to overlap each layer and construct the dataflow.

* 1. Overall Dataflow

|  |
| --- |
|  |
| [dataflow] |

1. Hardware Testing
   1. C-Simulation

In the C-simulation, the packed input values for first batch and prints the result matrix with batch\_size \* 10. By comparing the result of the python, the result is verified.

|  |
| --- |
| 텍스트, 스크린샷, 번호이(가) 표시된 사진  AI 생성 콘텐츠는 정확하지 않을 수 있습니다. |
| [Test bench code] |

|  |
| --- |
| 텍스트, 스크린샷, 폰트이(가) 표시된 사진  AI 생성 콘텐츠는 정확하지 않을 수 있습니다. |
| [C-simulation result] |
| 텍스트, 스크린샷이(가) 표시된 사진  AI 생성 콘텐츠는 정확하지 않을 수 있습니다. |
| [Python result] |

* 1. C-Synthesis

Synthesis the ip with vivado HLS, the report is below.

|  |
| --- |
|  |
| [Synthesis result-1] |
|  |
| [Synthesis result-2] |

The estimated clock cycle is 8.705ns. And the Latency is 12.534ms for batch\_size = 512. The utilization of BRAM and LUT is 83% and 85% each.

* 1. Implementation and Bitstream

After exporting the ip, generate the block design and bitstream.

|  |
| --- |
|  |
| [Top-level diagram] |

|  |
| --- |
| 스크린샷이(가) 표시된 사진  AI 생성 콘텐츠는 정확하지 않을 수 있습니다. |
| [Implemented device and worst negative slack] |
| 텍스트, 폰트, 라인, 번호이(가) 표시된 사진  AI 생성 콘텐츠는 정확하지 않을 수 있습니다. |
| [Design time summary] |
|  |
| [Design utilization and Power information] |

* 1. Run with the python code

Upload the bitstream to the board’s jupyter notebook and run the inference of 70000 images

|  |
| --- |
| 텍스트, 컴퓨터, 컴퓨터 하드웨어, 실내이(가) 표시된 사진  AI 생성 콘텐츠는 정확하지 않을 수 있습니다. |
| [Actual running image] |

The accuracy is 96.56% and run time is around 25.94 seconds.

|  |
| --- |
|  |
| [Running on the FP32 CPU] |

Compared with the runtime of CPU it really decreased while keeping the accuracy.

* 1. Run time break down

|  |
| --- |
|  |
| [Latency for each layers] |

Conv1: 5.018ms per batch size, 15.54%

Conv2: 11.448ms per batch size, 35.45%

Conv3: 12.534ms per batch size, 38.82%

Pool: 2.478ms per batch size, 7.67%

fc: 0.758ms per batch size, 2.35%

data\_out: 51.230us per batch size, 0.16%.

1. Discussion

Our project result generates a much better result than FP32 CPU. The runtime of TPU is 0.03% of the runtime of the FP32 CPU. Also, the accuracy is the same between them.

Although there is huge improvement, there are so many optimization ways such as implementing the im2col. It allows converting the calculation of the convolution to the just single matrix multiplication. Also, we don’t apply the tiling technique because when initially implemented, the BRAM usage is over 100%. So, we discarded tiling.

In conclusion, our TPU design improves a lot in terms of the running time. However, there are many other methods and further works to get more optimization.

1. Role of each team member

-Yeseung Lee(20211238): implemented sliding window and convolution with bit operations, optimize the hardware with pragmas, writing the report.

-Sanghun Pyo(20231405): implemented BNN training and inference, optimize the hardware - unrolling channel loop for convolution.

-Dohyeon Ha(20211352) : Try to apply tiling technic and window parallelization at conv1 and conv2. Change the conv1 “xnor-pop count” to “and+not-pop-count” using Lee’s idea. Manipulate pragma. Make a ppt.