Project 2 Report

ESE 5690 Special Topics in Machine Learning Accelerator
Spring 2020

Yuqi Liu
liu.yuqi@wustl.edu

Ben Guan
j.guan@wustl.edu

Introduction

To implement a Binary Neural Network (BNN), three layers of convolution network and one layer of fully connected network were implemented in C++. The idea is that the C++ code can be synthesized to verilog code which can be run on an actual Pynq Board. C-simulation was performed to check the result at each layer. A Matlab Script was developed to cross-validate the result as well.

Design Strategy

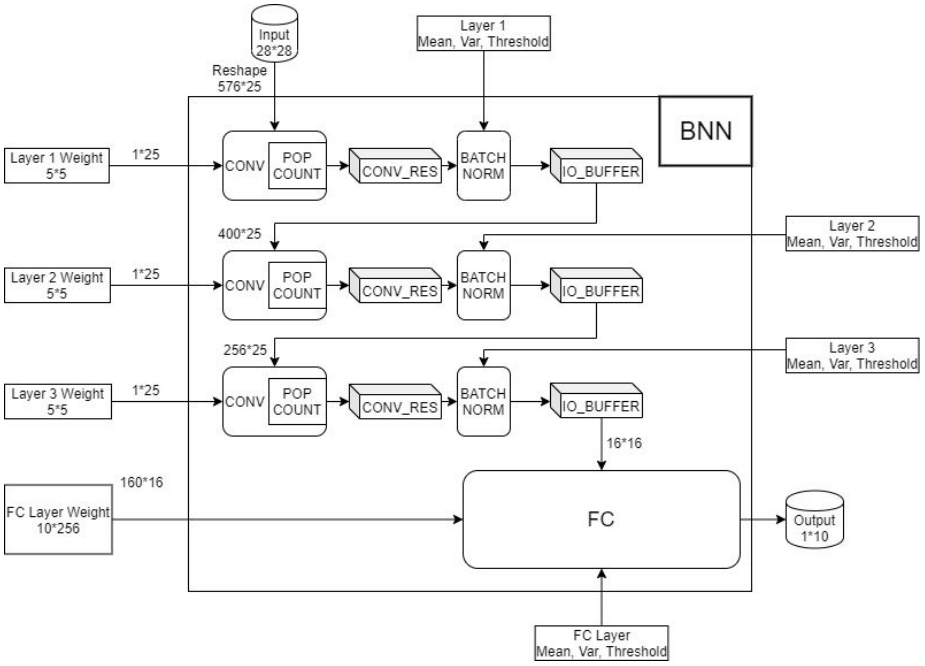

Figure 1. Design Block Diagram

Figure 1 shows the top level diagram of the BNN network. It takes the input activation, weight for three layers and the fully-connected (FC) layer, as well as the mean, variance and threshold used in batch normalization for all four layers as input and produces the output result of the BNN. Because every CONV layer performs essentially the same function despite the difference in size, it is natural to have the same subroutine and call the subroutine three times for each layer. Therefore, we divided the structure of our C code into the subroutines: CONV, POCOUNT and BATCH_NORM for the convolution layer as well as an FC function for the fully-connected layer. They are implemented in *utility.cpp*. The top level function which calls subroutines and executes the operation sequentially shown in the diagram is implemented in *bnn.cpp*.

All the inputs are stored in a text file in a comma separated style and their sizes (row by column) are specified in the diagram above. The size of the input is a 28 by 28 binary

matrix which is a representation of a single image. Then the filters of the convolution layer are 5 by 5 matrices. “Sliding window” techniques are used to produce the output at each layer and the result gets stored in CONV_RES. Then batch normalization takes the result and produces and stores the normalized result in the IO_BUFFER. Since the input and output are sequentially, CONV_RES and IO_BUFFER can be reused for all layers, which is shown in the block diagram. These two buffers are declared as arrays in the top level function.

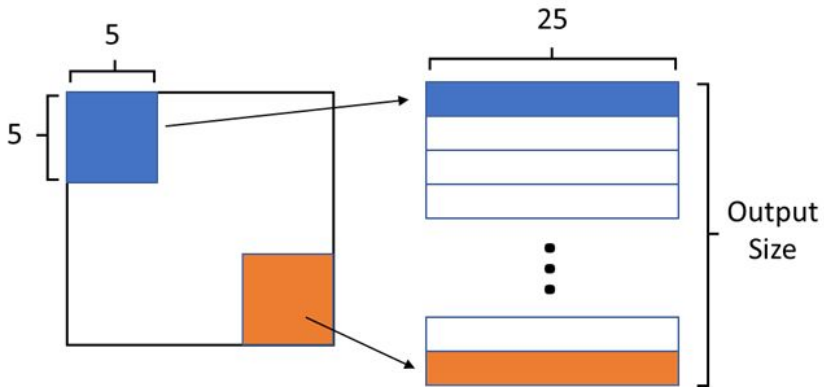

Figure 2. Input Flattening

To facilitate computation, the inputs are “flattened” to match one window of convolution, as shown in the figure above. Flattening will create duplicate bits as every input is in multiple windows, but we consider the benefit from ease of implementation and parallelism greater than the increase in storage. For the CONV layers, one window consists of 25 bits. Therefore, the input is reshaped into an 25-bit element array with its length equal to the output length. For example, the input 28*28 array is reshaped into a (24*24)*25 = 576*25 array and all 576 outputs from the first layer can be calculated by convolving each entry of the array with the flattened 25-bit weight vector. For the input data, we assume this flattening is done before passing them to the BNN network and this data packing is performed in the testbench. The BNN network will flatten the output from intermediate layers.

This flattening is the same for the FC layer but with a slightly different “window” size. The FC layer can be thought of a combination of 10 16*16 filters. Thus the window size is 256, but this is somehow large. Therefore, we used an actual window size of 16 and accumulated every 16 convolution results to get one output from the FC layer.

The convolution result stored in CONV_RES is not the actual numerical value of the convolution, instead it is the number of 1s in the result. This number is at maximum 25 and thus each entry of the CONV_RES array is set to 5-bit. Similarly, the IO buffer has a maximum size of 400, corresponding to the number of outputs of the second layer and

each entry is 25-bit. To finely control the bit-width which leads to more efficient hardware, we used arbitrary precision integer types for the data types by including *ap_int.h*. The type definitions of all kinds of data involved in the BNN is shown in the figure below.

```
// Type Definitions - width of each data
typedef ap_int<6> dataSize_t; //Size variables 6-bit
typedef ap_uint<weight_col*weight_row> weight_t; // Size of a weight 25-bit
typedef ap_uint<weight_col*weight_row> activation_t; // Size of a window 25-bit

typedef ap_uint<input_col> input1_t;    //Layer 1 Activation 28-bit
typedef ap_uint<out1_col> data_layer2;   //Layer 1 Output Layer 2 Input 24-bit
typedef ap_uint<out2_col> data_layer3; //Layer 2 Output Layer 3 Input 20-bit

typedef ap_uint<out3_col> data_fc;  //Data width for fully-connected layer (Row Stationary) 16-bit
typedef ap_uint<out3_col> wt_fc;    //Weight width for fully-connected layer (Row Stationary) 16-bit
typedef ap_uint<fc_weight_col> out_t; //10-bit Output

typedef ap_uint<5> conv_result; //Intermediate Conv Result (before batch norm) # of 1s in binary representation
typedef ap_int<6> value_t; //Actual numeric value of Conv Result
typedef ap_int<11> fc_result; //Numeric FC Multiplication Result
```

Figure 3. Width of different data

POPCOUNT

```
void PopCount(ap_uint<(weight_row*weight_col)> in, conv_result & out)
 {
#pragma HLS PIPELINE
   out = 0;
   POP: for ( int i = 0; i < in.length(); i++ ) {
     out += in[i];
   }
 }
```

Figure 4. Popcount Function

In the top level diagram, the popcount function is used in the convolution to determine how many ones are in the XNOR result. The implementation of the popcount is shown in the figure below. It takes a 25-bit number input and stores the output in the 5-bit data out, which is CONV_RES buffer in this case.

CONV

As shown in the picture below, the conv function is implemented using the xnor function for every output by repeating the same operation in the for loop. Then the popcount function is used to count the number of ‘1’ in the array. The fully-connected layer uses a similar mechanism where the input and weight were xnor-ed, then count the number of ‘1’, which will be illustrated in the later section.

```
void conv(out_t result_size, activation_t* a, weight_t & w, conv_result* o)
{
    activation_t temp;
    //Out_t used to accommodate output size (576 at maximum)
    CONV: for(out_t i = 0; i < result_size; i++){
#pragma HLS DATAFLOW
            temp = ~(a[i]^w);
            PopCount(temp, o[i]);
            //std::cout << "a[" << i <<"]: " << a[i] << std::endl;
            //std::cout << "XNOR Result[" << i <<"]: " << temp << std::endl;
            //std::cout << "o[" << i <<"]: " << o[i] << std::endl;
    }

}
```

Figure 5. Conv Function

BATCH_NORM

```
void batch_norm(dataSize_t in_rows, dataSize_t in_cols, dataSize_t weight_rows, dataSize_t weight_cols, conv_result* data_in, activation_t* data_out, value_t mean, value_t var, value_t threshold)
{
    value_t buffer;
    //Compute Output dimension w/ sliding window size of next layer
    dataSize_t out_rows = in_rows-weight_rows+1;
    dataSize_t out_cols = in_cols-weight_cols+1;
    ROW: for (dataSize_t i = 0; i < in_rows;i++){
#pragma HLS DATAFLOW
        COL: for (dataSize_t j = 0; j < in_cols; j++){
            //std::cout << "Input before norm[" << i*in_cols+j << "]: " << data_in[i*in_cols+j] << std::endl;
            //std::cout << "Multiplied by 2[" << i*in_cols+j << "]: " << (ap_uint<6>(data_in[i*in_cols+j]) << 1) << std::endl;

            //Convert from # of 1s to actual numeric result
            //buffer = (ap_uint<6>(data_in[i*in_cols+j]) << 1 )-weight_window_size; //Shifting yields same synth results as multiplication
            buffer = data_in[i*in_cols+j]*2 - weight_window_size;
            buffer = (buffer-mean)/var;
            if(buffer >= threshold){
                buffer = 1;
            }
            else{
                buffer = 0;
            }
            //Place Binary Result into the output
            ROW_WINDOW_POS: for(dataSize_t ii = 0; ii < out_rows; ii++){
#pragma HLS PIPELINE
    //ii - Row index of window topleft
                COL_WINDOW_POS: for(dataSize_t jj = 0; jj < out_cols; jj++){ //jj - Column index of window topleft
                    dataSize_t p = (i-ii); //Row index of element within the window
                    dataSize_t q = (j-jj); //Column index of element within the window
                    if((p >= 0) && (q >= 0) && (p < weight_rows) && (q < weight_cols)){ //Check no out-of-bounds
                        data_out[ii*out_cols+jj][p*weight_cols+q] = buffer;
                    }
                }
            }
        }
    }
}
```

Figure 6. Batch Normalization Function

The implementation of batch normalization function is shown above. For single output, the actual numerical output is calculated from the # of 1s stored in the input data (CONV_RES in this case). The output at each layer is then normalized using

Result = (output(i,j) - mean)/variance

The result stored in a local buffer is then compared with the threshold to determine whether the output should be 1 or 0. If the result is greater than or equal to the threshold, then the output is 1, else 0. The values of mean, variance, and threshold should be determined when training the weight.

After calculating the output, the flattening procedure is performed by the inner two for loops that will place the binary value into corresponding locations in the IO_BUFFER. It

checks whether an element fits inside every window of convolution and will place the element at the location corresponding to its location in the window it fits in that window.

FC

Eventually, the output of the 3rd convolution layer is then multiplied with the weight of the fully connected layer which is 256 by 10. The output is a 10 by 1 matrix. The implementation of the FC function is shown in the figure below. Inside the MAC for loop, the same XNOR operation in the CONV function is done to calculate the output from each 16-element window. Because the actual window is of size 256, the element being accumulated is the total number of 1s in the result, which is calculated from the POPCOUNT loop.

```
//Fully-connected Layer
void fc(activation_t* a, wt_fc* w, out_t & o, fc_result mean, fc_result var, fc_result threshold){
    fc_result mac_buffer;
    data_fc xnor_buffer;

#ifndef __SYNTHESIS__
        //File storing actual output
        std::ofstream outfc("fcout_raw.txt");
#endif

    FC: for(dataSize_t i = 0; i < fc_weight_col;i++){
        mac_buffer = 0;
        //Calculation of one output - Assume Output-Stationary
        MAC: for(dataSize_t j = 0; j < out3_row;j++){
#pragma HLS PIPELINE
            xnor_buffer= ~(a[j]^w[i*out3_row+j]); //Multiplication
            //PopCount
            POPCOUNT: for(dataSize_t k = 0; k < out3_col; k++){
                mac_buffer += xnor_buffer[k];
            }
        }
        //Batch-Normalization
        //mac_buffer = (mac_buffer << 1) - fc_weight_row; //11-bit needed for *2 operation
        mac_buffer = mac_buffer*2 - fc_weight_row; //Multiply version
#ifndef __SYNTHESIS__
        //Write Actual Output to file
        if(i) outfc << ",";
        outfc << mac_buffer;
#endif

        mac_buffer = (mac_buffer-mean)/var;

        if(mac_buffer >= threshold){
            o[i] = 1;
        }
        else{
            o[i] = 0;
        }
    }
```

Figure 7. FC Function

After this accumulation where the convolution/matrix-vector multiplication is done, the actual numeric result of the output is converted and stored in the local buffer. Note the total number of bits subtracted during the conversion process is 256 since this is the size of the actual window. Then batch normalization is performed which produces the final output value. This process is repeated in the top FC for loop to calculate all 10 outputs.

Testing

Different inputs and weights are used to test the C++ code. In C simulation, the bnn function will print the intermediate results and write them to file. A matlab script was implemented to cross-validate the result of BNN in each layer. Outputs of the BNN are consistent from the MatLab result.

Optimization

To minimize the run time and maximize efficiency, the weights at each layer and the input are flattened to a vector which decreases the number of for-loops needed for the computation. We also added several pipeline techniques which increase parallelism, but as a result, the utilization of resources such as Flip-flops and LUTs also increase. PIPELINE and DATAFLOW directives are placed in the subroutines at the locations shown in the figures above. The strategy is to place it at the second level for loop for every double for loop pair to enable parallel processing of one row. By doing so, we see a four-fold minimum latency improvement than the synthesis result without any directives, which is shown below.

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	7.541	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
81405	1807917	81405	1807917	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	118	-
FIFO	-	-	-	-	-
Instance	-	0	664	1213	-
Memory	2	-	0	0	0
Multiplexer	-	-	-	415	-
Register	-	-	160	-	-
Total	2	0	824	1746	0
Available	280	220	106400	53200	0

Figure 8. Synthesis Result without optimization directive

The total resource used doubles and we think the 4x performance increase is worthwhile of this 2x cost increase. The synthesis result of the final design will be discussed in the next section. Regarding when to use PIPELINE and when to use

DATAFLOW, these two directives are similar to each other and from the documentation, it seems PIPELINE targets more detailed and low-level instruction pipelining and DATAFLOW is for high-level task pipelining. For locations where both two directives can be applied (DATAFLOW applied to some location caused synthesis errors), they are both tried and the one with better performance is picked.

Synthesis

One thing we encountered is that the same codes have different synthesis results on each other's HLS software. Therefore, both are included here. As we can see from the figure below, the estimated clock period is 8.639 ns and 8.393 ns which is smaller than the target clock period of 10 ns.

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.393	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
20495	1661615	20495	1661615	none

Figure 9. Timing and Latency Result

The figure below shows the execution of functions in a sequential order. First, the convolution layer is computed then the result is normalized. This process is repeated two more times. Then the result of the 3rd layer is multiplied with the fully-connected weight matrix. Reading input and writing output to the .txt files are done at the beginning and at the end, respectively.

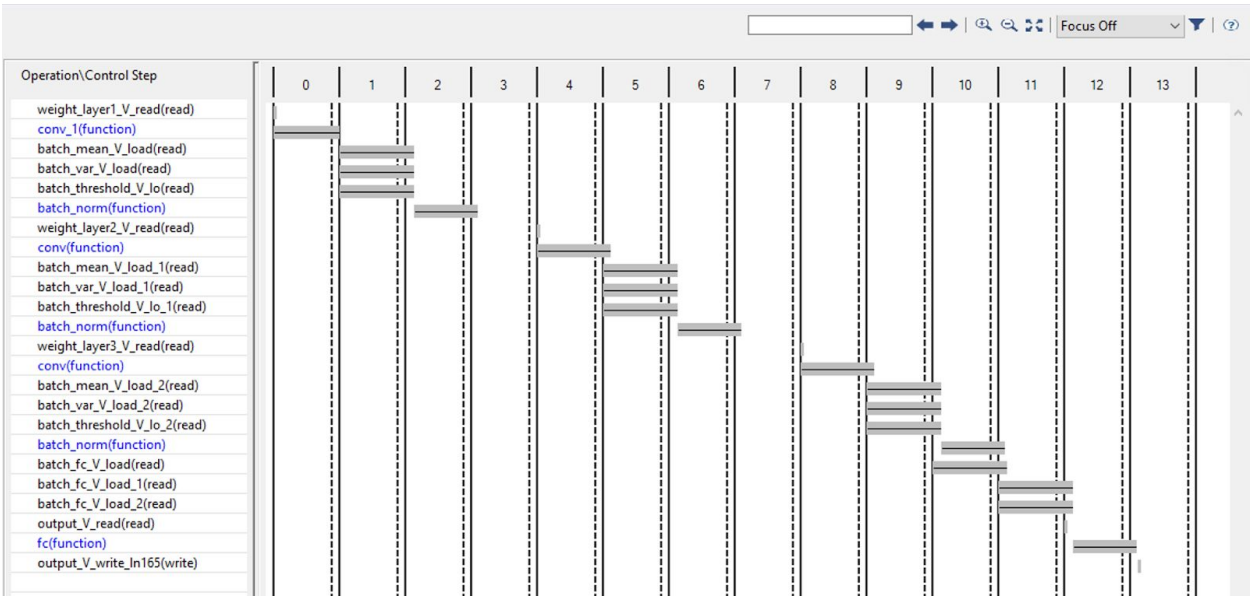

Figure 10. Execution Schedule

The BRAM was used to store the convolution layer result, i.e. COV_RES, as well as the input and output, i.e. IO_BUFFER. Most of the resources used are flip-flops and LUTs.

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	12	-
FIFO	-	-	-	-	-
Instance	0	1	1782	2932	-
Memory	2	-	0	0	0
Multiplexer	-	-	-	374	-
Register	-	-	75	-	-
Total	2	1	1857	3318	0
Available	280	220	106400	53200	0
Utilization (%)	~0	~0	1	6	0

Figure 11. Resource Utilization Result