WES237C: Project 5 Binary Neural Network

Yi Song
yis057@ucsd.edu
PID# A53266440

Sebastian Nevarez
senevarez@ucsd.edu
PID# A59021322

BNN Design

The binary neural network is 3 layers long with an array of 25 unsigned integers (32-bit each) as input (800-bits total). The output is a 10-integer long array with 10 probabilities of the 10 digits (0-9). The highest probability index is the prediction of the input image. To implement each layer of the algorithm, we start by traversing the input array and the weights. A XNOR_Popcount function is performed to do a bitwise XNOR on each element in the input array and weights and the number of ones in the result is accumulated. Then we do 2*(accumulated ones -16) - 784 to get a reduced middle layer of integers with size 128. Finally, a sign and quantize function is performed to prepare the array of 128 integers for the next layer. The sign function checks the sign of the integer and assigns +1 to positive numbers and -1 to negative numbers. The quantize function checks the signed numbers and assigns 0s to 1s and 1s to -1s. That changes the image background to all black with white digits written on the background.
We repeat the first layer two more times with different array lengths. And finally at layer 3, we take the output before doing the sign and quantize function. That will give us an array of 10 probabilities and the index corresponding to the highest probability is the predicted number.
As a result we can see the vitis version is really efficient in terms of hardware, it uses 0 DSP modules which means there are no multiplications inside the code, all bitwise manipulations.
Our interval is 6098 cycles making the throughput 16.4kHz.

Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	U
bnn				-	6097	6.097E4	-	6098	-	dataflow	12	0	4580	9993	
entry_proc				-	0	0.0	-	0	-	no	0	0	3	29	
Loop_1_proc6				-	6097	6.097E4	-	6097	-	no	12	0	1722	4579	
Loop_1_proc6_Pipeline_1				-	6	60.000	-	6	-	no	0	0	5	47	
Loop_1_proc6_Pipeline_2				-	4	40.000	-	4	-	no	0	0	4	45	
Loop_1_proc6_Pipeline_sign				-	131	1.310E3	-	131	-	no	0	0	30	107	
Loop_1_proc6_Pipeline_quantize				-	133	1.330E3	-	133	-	no	0	0	124	128	
Loop_1_proc6_Pipeline_Pack_l1				-	34	340.000	-	34	-	no	0	0	142	638	
Loop_1_proc6_Pipeline_layer_2_VITIS_LOOP_174_3				-	262	2.620E3	-	262	-	no	1	0	421	1163	
Loop_1_proc6_Pipeline_sign1				-	67	670.000	-	67	-	no	0	0	27	106	
Loop_1_proc6_Pipeline_quantize2				-	69	690.000	-	69	-	no	0	0	122	127	
Loop_1_proc6_Pipeline_Pack_l2_VITIS_LOOP_207_4				-	130	1.300E3	-	130	-	no	0	0	60	322	
layer_1				-	5248	5.248E4	41	-	128	no	-	-	-	-	
Loop_1_proc6_Pipeline_VITIS_LOOP_127_1				-	31	310.000	-	31	-	no	8	0	378	1047	
VITIS_LOOP_127_1				-	29	290.000	6	1	25	yes	-	-	-	-	
Loop_layer_3_proc				-	20	200.000	-	20	-	no	0	0	823	1956	
Loop_layer_3_proc_Pipeline_layer_3				-	18	180.000	-	18	-	no	0	0	754	1916	
layer_3				-	16	160.000	8	1	10	yes	-	-	-	-	
Loop_8_proc				-	21	210.000	-	21	-	no	0	0	92	330	
Loop_8_proc_Pipeline_1				-	13	130.000	-	13	-	no	0	0	19	84	
Loop 1				-	11	110.000	3	1	10	yes	-	-	-	-	

Figure 1: Performance and Resource Utilization

Latency

Summary

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
6097	6097	60.970 us	60.970 us	6098	6098	dataflow

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	12	-
FIFO	-	-	198	135	-
Instance	12	-	4252	9826	0
Memory	0	-	128	2	0
Multiplexer	-	-	-	18	-
Register	-	-	2	-	-
Total	12	0	4580	9993	0
Available	280	220	106400	53200	0
Utilization (%)	4	0	4	18	0

Figure 2: Latency and Resource Utilization Percentages

Python Demo

The main function or our demo starts by defining the run_option which was defined as 1 (XNOR), 2 (MAC), or 3 (HLS). For options 1 and 2, the function run_test is called to run the BNN and the timing is computed using a simple python timer. Option 3 calls the function hlscode to run our implemented BNN and also computes the runtime. See Table 1 below for a comparison of each implementation.

Implementation	Runtime (s)	Accuracy
XNOR	-	89.39
MAC	81.59540247917175	89.39
HLS	-	-

Table 1: Runtime & Accuracy Comparison

The function to run our HLS BNN implementation first loads the bitstream and the provided dataset. From there, our next step is to sanitize our input to match the HLS requirements. To sanitize, we first need to flip all the 0s to 1s and all the non-0s to 0s. The next step is to add 16 0s to the end to reach 800-bits which corresponds to 25 32-bit unsigned integers, the format used in the HLS project and how the weights are trained. Once the input has been sanitized, we are able to run our implementation for BNN. See Table 2 for a snippet of our hlscode function. After running the three layers in the BNN algorithm, we need to output the index with the highest probability as our prediction. A comparison will be made between the output and the label.

```
def hlscode(self):
    """This is a reference implementation for HLS.
    Intentionally, left empty so that students implement the HLS ref design.

    :return:
    """

    # Load the bitstream and dataset
    ol=Overlay('./design_1_wrapper.bit')
    mnist = np.load("dataset/mnist-original.npy", allow_pickle=True)
    X = mnist.item().get("data")
    y = mnist.item().get("label")

    # Sanitize the input
    # Flip 0s to 1s and non-0s to 0s
    # Add 16 0s to the end to reach 800-bits

    # Call BNN
    # Output the index with highest probability
    # Compare output with the label

    print("Done")
```

Table 2: hlscode()