Demo Video: https://youtu.be/6yMHNaI6sAk

Github Link:

https://github.com/kate-romero/CSE237C_FA25_Projects/tree/main/bnn_project5

1) **Testbench Integration:**

The BNN testbench (bnn_test.cpp) verifies the correctness of our BNN implementation by comparing its outputs against the golden reference values at every layer of the BNN. These golden reference values are found in golden.h, the file that is generated using the bnn_mnist.py file. Within the bnn_mnist.py file, the user can choose which file from the dataset they want to use, the starting sample index, and the number of samples to process. Then the bnn_mnist.py file loads the MNIST test data, processes each sample, executes the Python reference BNN model, and saves the output of each layer for every sample as the golden reference values. Then we pack each sample and reference values into the C++ header file golden.h. We had to be very careful with how we package these values since hardware only operates on {0,1} and the BNN is trained using values in {-1,+1}, so the python file applies quantization, bit packing logic, and padding to accommodate how the hardware operates. After the golden.h file is generated, the bnn_test.cpp file dynamically adjusts to what the user has requested (starting sample and how many samples it wants to process), gets the input values (samples), and verifies the outputs layer by layer with the reference values produced in golden.h file. For each of the samples requested, we call the function:

```
bnn_with_layers(input, layer1_out, layer2_out, final_out)
```

We first verify that the layer 1 output matches the golden layer 1 output (reference values produced by bnn_mnist.py). Then, we verify that the layer 2 output matches the golden layer 2 output. Lastly, we verify that the layer 3 output matches the golden layer 3 output. This testbench integration enables users to have more flexibility on what file they want to use. The testbench lets each layer to be implemented and validated independently, so debugging at each stage becomes easier.

2) **Optimization Techniques:**

bnn() - In the bnn() function we used `#pragma HLS DATAFLOW, #pragma HLS ARRAY_PARTITION variable=variable complete, #pragma HLS UNROLL.`

This optimization `#pragma HLS DATAFLOW,` helped the layers to be overlapped in execution, which greatly improved Interval cycles and overall throughput however the trade-off was resource utilization, especially LUT. Using the `#pragma HLS DATAFLOW` it double the amount of LUT utilization. Without it, we gets less throughput and lower resource utilization. (See table below)

	Throughput	Latency (cycles)	Interval (cycles)	Estimated (ns)	BRAM	DSP	FF	LUT
No #pragma HLS DataFlow	720 KHz	189	190	7.300	33	0	11000	28388
Yes #pragma HLS DataFlow	1.82 MHz	188	64	8.568	33	0	38815	48560

This optimization `#pragma HLS ARRAY_PARTITION variable=variable complete` it fully partitions all the arrays the trade-off of doing this is that uses a lot more resources. Together with `#pragma HLS UNROLL` it allows for many parallel reads/writes in a single cycle.

dense_layer() - This function uses `#pragma HLS INLINE off, #pragma HLS PIPELINE II=1, #pragma HLS UNROLL factor=2.`

This optimization `#pragma HLS INLINE off` is to have a separate hardware module for this function, meaning this function it's its own stage such that `#pragma HLS DATAFLOW` can run different stages of overall function at the same time instead of running it sequencely. This would overall improve throughput.

This optimization `#pragma HLS PIPELINE II=1 and #pragma HLS UNROLL factor=2,` is to be busy each cycle, overall increasing throughput.

sign_and_quantize() - This function uses `#pragma HLS INLINE off, #pragma HLS PIPELINE II=1, #pragma HLS UNROLL.` All of these optimization improve throughput. It

makes this function be its own “stage” that way multiple stage can run at the same time, as wee as we make sure to be busy at every cycle.

popcount() - This helper function was optmized by removing the for-loop we had before, instead of doing 32 iterations, it counts bits in parallel which is optimized for hardware. Also, we only used bitwise operations and adders, which again computes fast on hardware.

xnor ( ) - This function uses `#pragma HLS INLINE` which helps to run things in parallel with other functions what are running concurrently with xnor().

3) HLS IP Core Interface

We interfaced the bnn IP core using AXI Master (m_axi) for data transfer and allocate() for on-board memory management. Using m_axi for the input and output arrays bundled as gmem enabled burst transfers from DDR memory to the FPGA. This allows multiple words to be transferred in a single burst transaction (rather than requiring separate transactions for each word), which reduces data transfer overhead. We configured the burst optimization parameters (max_read_burst_length=32, num_read_outstanding=16) to optimize bandwidth utilization.

Using the s_axilite interface for control allows the processor to pass memory addresses to the IP core and control execution with start/stop/done signals. The processor writes the physical addresses of input/output buffers to specific register offsets and sets the start bit to trigger execution.

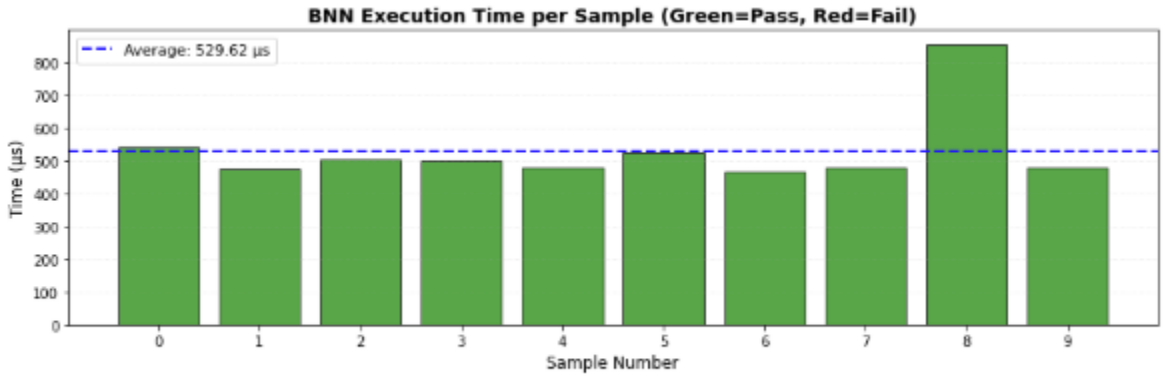
Using PYNQ's allocate() to create contiguous memory buffers that can be accessed by both the processor and FPGA eliminated the need for explicit data copying between the two. This reduction in overhead also improves throughput.

4) **Notebook Demo:**

Our Jupyter Notebook Demo runs our binary neural network implementation on all 10 samples present in the "mnist_test_data_original.npy" data set. First, it displays visualizations of the input data by reconstructing the images from the packaged binary data in the golden output. This, of course results in black and white images (as opposed to grayscale which could be reconstructed from the raw data). This demonstrates the effect of the data packing and visualizes what data our binary neural network will be sorting into numerical values.

Second, our notebook runs the samples and compares the outputs to the golden.h reference at each layer as described in the testbench section. The purpose of testing each layer is so that 1) if something goes wrong, we can determine where our implementation is failing and 2) ensuring that our desired accuracy level is being met throughout every step of computation strengthens our design's ability to perform well on other datasets we may wish to categorize after the completion of this project.

Finally, our notebook displays our implementation's accuracy and time, including a chart of the runtime for each sample. We also included average runtime and the standard deviation.




```
Average: 529.62 µs | Std Dev: 110.77 µs
```