CNN Digit Recognizer for Configurable Image Sizes

Name: Amey Kulkarni

PSU ID: 933959421

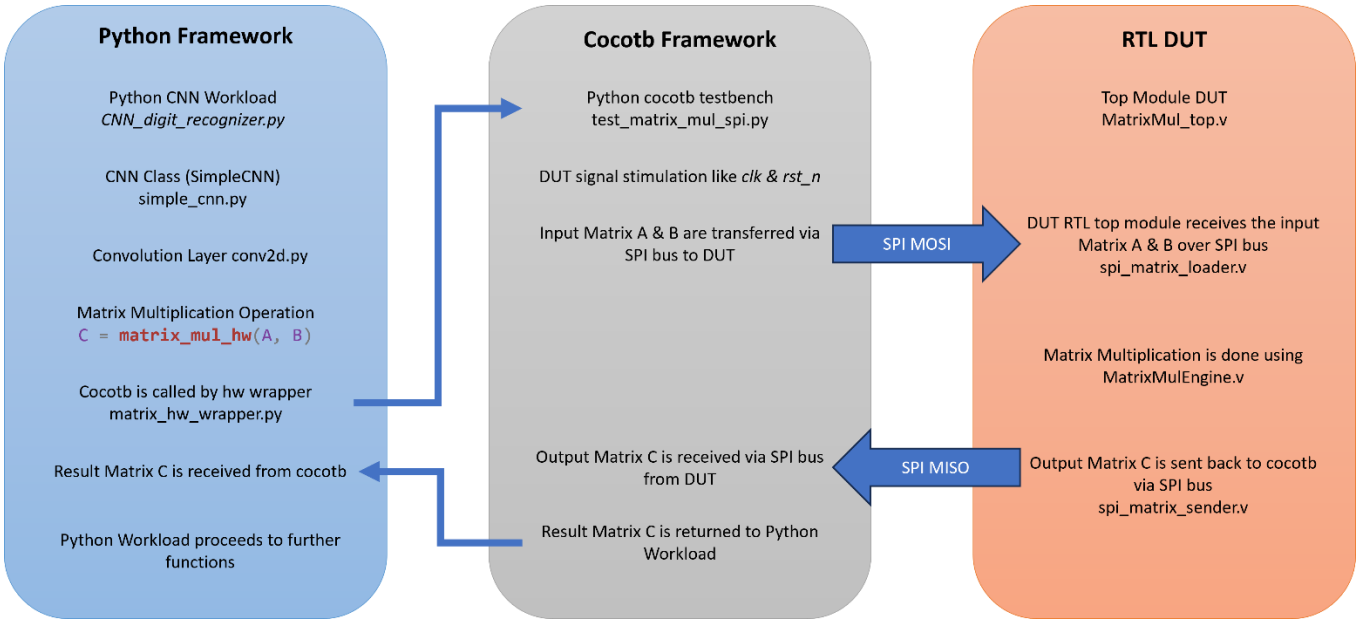
Project Overview:

This project demonstrates a hybrid software/hardware approach to CNN-based handwritten digit recognition.

The CNN is implemented in Python (*NumPy*), but all matrix multiplications (the computational bottleneck in CNNs) are offloaded to a custom-designed hardware accelerator written in Verilog.

The hardware is simulated using *cocotb*, and Python communicates with the hardware via a simulated "SPI protocol".

Project Architecture:



Project Success?:

Task	Success?	Comments
SW algorithm benchmark	✅ Yes	Benchmarked using '*cProfile*' and 'snakeviz' module.
Designed, built, and tested a custom HW accelerator	✅ Yes	Designed MatrixMul_top.v DUT in Verilog.
Simulated HW and SW together as a System and it is working?	✅ Yes	Simulated the system together using 'cocotb' and getting functionally correct results similar to SW only flow.
Synthesized the RTL using OpenLane-2 Flow	❌ No	OpenLane-2 has limitations with System Verilog. Please read documentation for further details.
Synthesized the RTL using Synopsys Design Compiler	✅ Yes	Used Synopsys Design Compiler available on PSU servers to Synthesize design and generate results.
Synthesized the RTL using Vivado for FPGA	✅ Yes	Used Vivado to synthesize the design for Nexys4 board with Artix-7 FPGA. Will test the design on FPGA in Summer Term.
Evaluated and benchmarked the entire System with SPI communication	✅ Yes	Results are shown in this Project Report.

Python Workload:

Neural Network Layers:

The CNN is implemented from scratch using NumPy, with the following layers:

Conv2D: Convolutional layer, offloads matrix multiplication to hardware.

ReLU: Activation function.

Flatten: Flattens 4D tensors to 2D for dense layers.

Dense: Fully connected layer, also offloads matrix multiplication to hardware.

Softmax: Output activation for classification.

Hardware Integration:

matrix_hw_wrapper.py: Provides the *matrix_mul_hw* function, which:

- Serializes matrices A and B to input_buffer.txt.
- Invokes the cocotb/Verilog simulation via make.
- Waits for output_buffer.txt with result matrix C.
- Reads and returns C as a NumPy array.

conv2d.py: Uses *matrix_mul_hw* for core matrix multiplication, thus transparently offloading heavy computation to hardware.

Training and Inference:

CNN_digit_recognizer.py: Main script for training and inference.

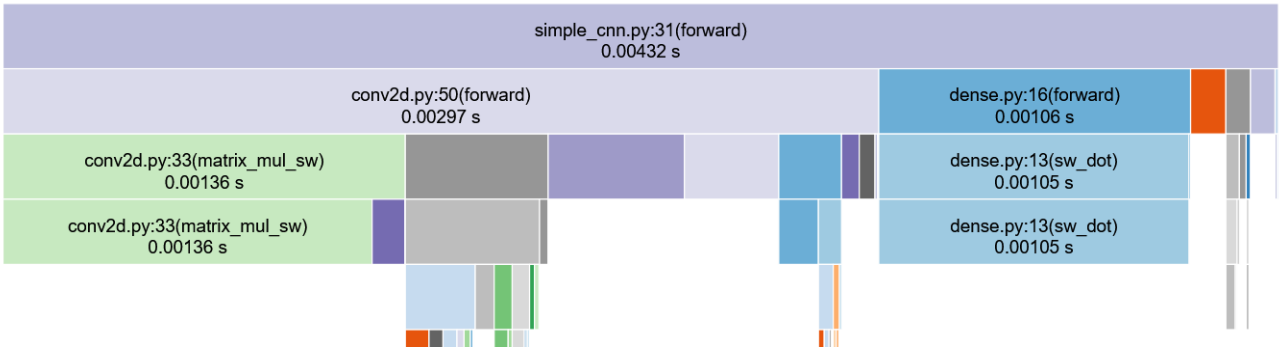
- Loads images, preprocesses, trains the CNN, or runs inference.
- During forward/backward passes, all matrix multiplications are performed by the hardware accelerator.
- Uses Software Matrix Multiplication while training "*matrix_mul_sw(A, B)*".
- Only Uses Hardware Matrix Multiplication while inference "*matrix_mul_hw(A, B)*".

Python File-by-File Breakdown:

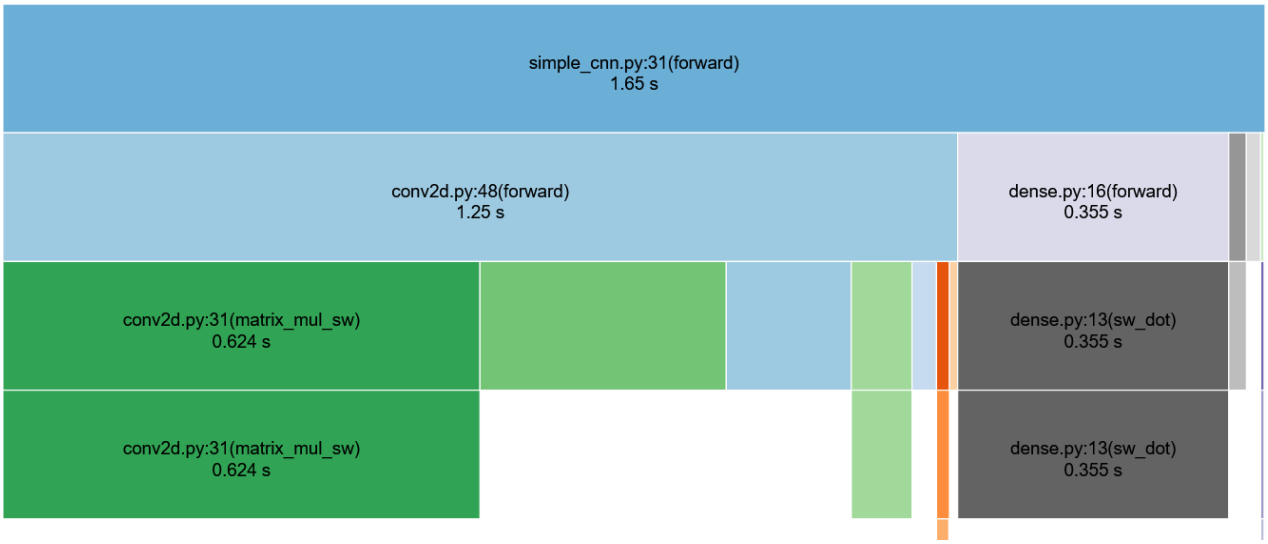
1. CNN_digit_recognizer.py
    a. Main entry point for training and inference.
    b. Loads images, handles data preprocessing, batching, and evaluation.
    c. Calls into SimpleCNN for model operations.
2. simple_cnn.py
    a. Implements the SimpleCNN class, which wires together all layers.
    b. Handles forward and backward propagation, as well as model save/load.
3. conv2d.py
    a. Implements the convolutional layer.
    b. Converts convolution into matrix multiplication (im2col), then calls matrix_mul_hw.
    c. Handles bias addition and output reshaping.
4. dense.py
    a. Implements the fully connected layer.
    b. Calls matrix_mul_hw for matrix multiplication.
5. flatten.py
    a. Implements the flattening operation between convolutional and dense layers.
6. relu_softmax.py
    a. Implements ReLU and Softmax activation functions.
7. neuron.py
    a. Implements a single neuron (not used in main CNN, but useful for extension).
8. matrix_hw_wrapper.py
    a. Handles all communication with the hardware accelerator.
    b. Serializes matrices to input_buffer.txt, invokes cocotb/Verilog simulation, and reads results from output_buffer.txt.
9. do_matrix_mul.py
    a. Standalone script to test hardware matrix multiplication.
    b. Generates random matrices, calls matrix_mul_hw, and compares results to NumPy.
10. run_profiler.py
    a. Profiles the inference function for performance analysis.
11. test_matrix_mul_spi.py
    a. cocotb testbench for end-to-end SPI-based matrix multiplication.
    b. Drives the Verilog hardware with matrices from input_buffer.txt and writes results to output_buffer.txt.
12. input_buffer.txt / output_buffer.txt
    a. Temporary files for passing matrix data between Python and the hardware simulation.

Python Benchmark:

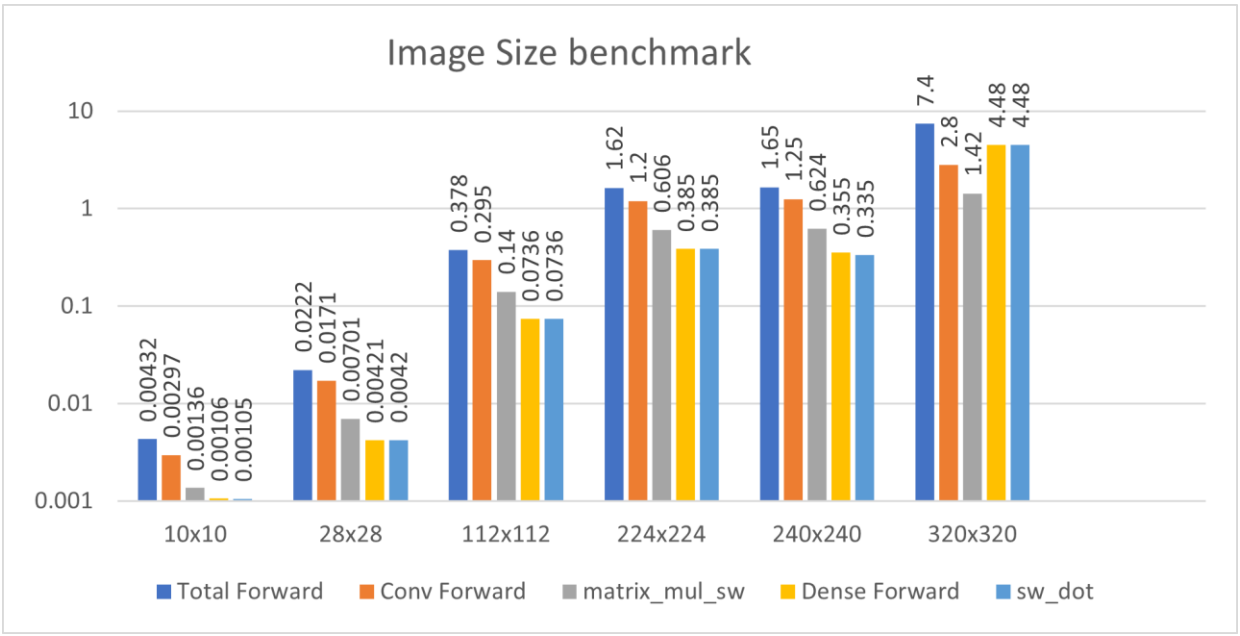
1. cProfile and Snakeviz for Images of 10x10 size:



2. cProfile and Snakeviz for Images of 28x28 size:



3. Various Image Sizes



Benchmark Conclusion:

- From above benchmark data, we can clearly see the main bottleneck and the highest contributor in the "Total Forward" inference time is "matrix_mul_sw()" Function.
- With increasing image sizes, the time taken by this function grows exponentially. "Y-axis" in above graph is logarithmic.
- This function does the multiplication of 2 input matrices, A & B. The A Matrix is flattened input image and B matrix is the flattened Filter for convolution operation.
- The next contributor is "sw_dot()" function.

Workload chosen for Accelerating on HW:

- From the benchmarks, we can clearly conclude I need to accelerate the "matrix_mul_sw()" function on HW.
- So I need to build a Floating point Matrix Multiplication unit in Verilog which can perform this multiplication of 2 input matrices (A & B) and return the result matrix (C).

Vibe Coding prompts used:

- **CNN in TensorFlow**
  - Can you help me build a CNN in TensorFlow python for digit recognition? The input is 28x28 grayscale images.
  - Can you show me how to train it on MNIST and test the accuracy?
  - I want to have separate modes called "infer" and "train" which I will specify at runtime using sys.argv. Also I want to dump the trained model after training is done so while running inference, it can just load that trained model.
  - Can you try to write a separate script which can benchmark the inference of the CNN model using "cProfile" and "snakeviz".
  - Somehow the snakeviz is not showing the functions which I want to see and it is showing lot of TensorFlow library function calls rather than the actual function calls.
  - ChatGPT suggested me some modifications but none worked.
  - Can you help me to benchmark this using TensorBoard?
  - I was not able to make TensorBoard show any meaningful data till the end.

- **Pure Python CNN for Local Dataset**
  - Can you write a CNN from scratch in Python using NumPy only? No TensorFlow or PyTorch. Build it using OOPs methodology, so there should be proper classes for each layers of CNN, start from a Neuron class and go all the way up.
  - Instead of MNIST dataset, can you modify it to use dataset which I have locally with images in JPG format. The directory structure if proper, like "Dataset_28x28/0/lot of jpgs", "Dataset_28x28/1/lot of jpgs" and so on, so you can identify the label while training based on the folder name itself.
  - I want to have separate modes called "infer" and "train" which I will specify at runtime using sys.argv. Also I want to dump the trained model after training is done so while running inference, it can just load that trained model.
  - Can you try to write a separate script which can benchmark the inference of the CNN model using "cProfile" and "snakeviz".

RTL DUT:

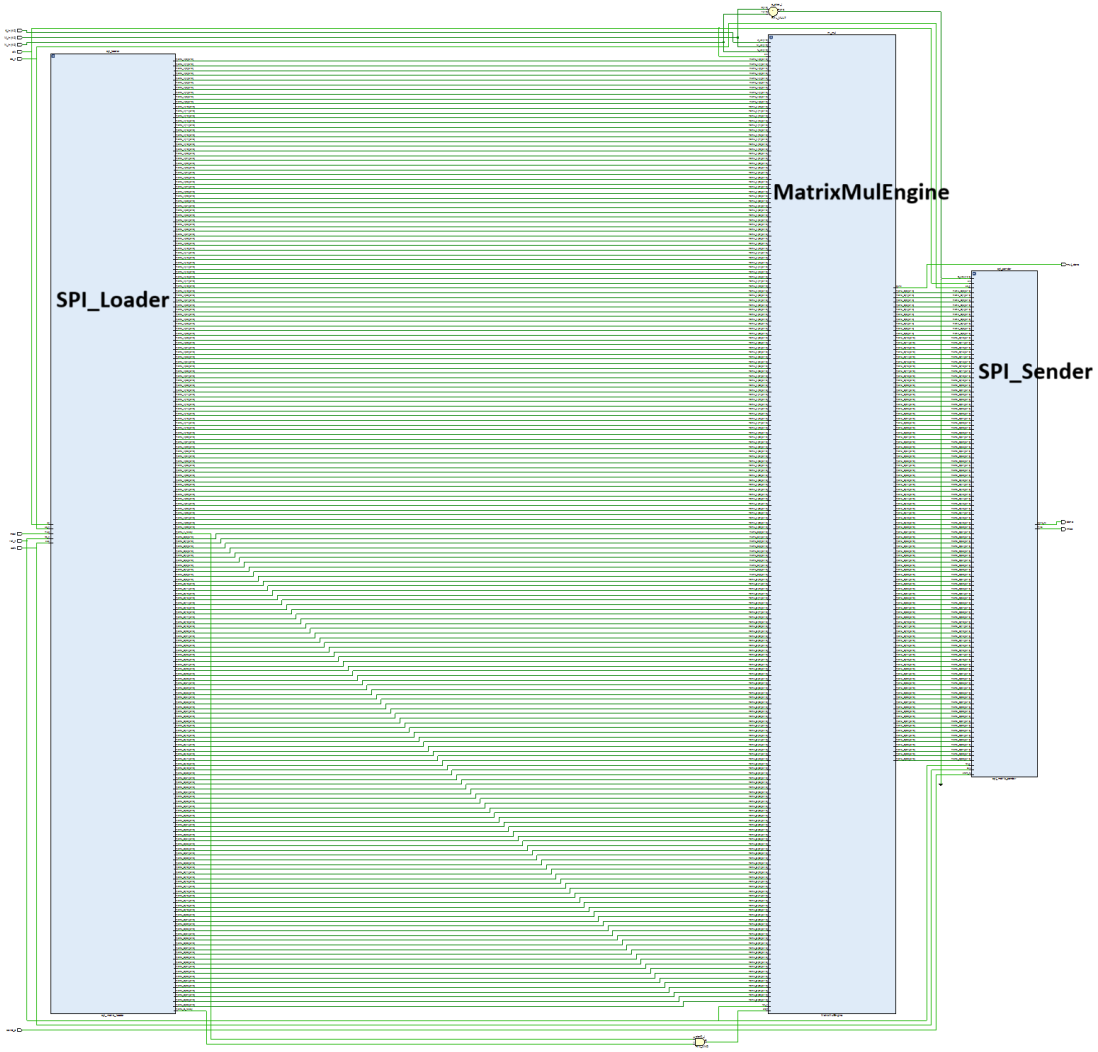
Below is the “Black-Box” representation of my Matrix Multiplication Engine DUT.

Design:

Top-level Ports:

- INPUTs –
  - clk: Main Design clock
  - rst_n: Active low reset signal
  - sclk: clock signal for SPI Bus
  - mosi: Master Out Slave In port of SPI Bus
  - cs_n: Active low chip select signal for SPI Bus
  - send_c: Input to the engine to signal when to send the result
  - M_in[0:7]: M-Dimension for Input Matrix
  - N_in[0:7]: N-Dimension for Input Matrix
  - K_in[0:7]: K-Dimension for Input Matrix
- OUPUTs –
  - miso: Master In Slave Out port of SPI Bus
  - mul_done: Output from DUT to signal matrix multiplication is complete
  - done: Output from DUT to signal output matrix C is successfully sent over SPI

Inner Workings of the design:



- SPI_Loader –
    - This module receives the Input Matrix A & B from ‘Python Workload’ via SPI Bus.
    - It then stores them into the respective buffers for ‘MatrixMulEngine’ to process.
- MatrixMulEngine –
    - This module is the heart of the design which multiplies two Matrices A & B and generates the output Matrix C.
    - Once the results are generated, it hands over the result matrix C to SPI Sender.
    - It raises the “mul_done” signal High for one clock cycle once multiplication is done.
- SPI_Sender –
    - This module sends the result Matrix C to ‘Python’ via SPI Bus.
    - It raises the “done” signal High for one clock cycle once matrix is successfully sent over the SPI Bus.

Independent Verification of the design:

- I have developed this design from scratch and verified it at every step. Below are GitHub repositories I created at each step –
    - Link - Floating_Point_MAC_HW_Accelerator:
        - This was the first step, I developed a Floating Point MAC unit which can take 3 32 bit Floating Point IEE-754 numbers as inputs (a, b & c) and generate a MAC 'result' as output.
        - Exact operation being performed here –

            `result = a + b * c`
        - All the details are mentioned in the README.md file.
        - I have tested this module heavily with 'cocotb' framework with script 'main_MAC_test.py' and made sure it produced the correct results.
    - Link - Dot_Prod_HW_Accelerator:
        - This was second step, it is an FSM which can take a 1D vector of Floating Point numbers and perform the MAC operation on it.
        - This Dot Product HW Accelerator can take 2 1D vectors of Floating Point numbers (called Patch and Filter) and perform the dot product operation on it.
        - The exact operation it performs is -

            result = ∑ patch[i] * filter[i]
        - This module is independently tested using traditional Verilog Testbench (tb_DotProductEngine.v) to make sure it produces expected results.
    - Link - Matrix_Dot_Product_HW_Accelerator
        - This was the third step, it is an FSM which can take a 2D matrix of Floating Point numbers and perform the dot product operation on it.
        - This Matrix Dot Product HW Accelerator can take 2 2D matrices of Floating Point numbers (called Patch and Filter) and perform the dot product operation on it.
        - This module is independently tested using traditional Verilog Testbench (tb_MatrixMulEngine.v) to make sure it produces expected results.
    - Link - MatrixMul_cocotb
        - This was the fourth step, I have created a full cocotb framework to verify the functionality of the MatrixMulEngine Verilog DUT.
    - Link - W08_C25_SPI_for_CNN
        - This was the fifth step, where I designed, tested and benchmarked an independent SPI communication channel to be integrated into the main design.
        - I tested this with both traditional testbench using Verilog (tb_spi_top.v) and cocotb testbench with (test_tx_rx_spi.py)
        - Benchmarking results are available in the README.md file.
    - Link - CNN_hand_written_digit
        - This is the final step, where everything comes together.
        - I tested the full design with cocotb (test_matrix_mul_spi.py) to make sure it is producing expected results.

Vibe Coding prompts used:

- **Floating Point MAC HW Accelerator**
  - Can you write Verilog code for a floating point MAC unit? It should take 3 inputs a, b, c and compute result = a * b + c. Use IEE-754 32-bit floating-point format for this.
  - I want to test this MAC unit using cocotb. Can you write a testbench script and full cocotb framework inclusing Makefile by using random floating point values? Also compute SW version of these values and cross check the results.
  - The RTL code is not working, something is wrong with the shifting logic for alignment. Can you try a different approach to align and add the floating point values?
  - Now cocotb seems fine but the RTL code is still not working and it seems to be logically incorrect, can you try to draw some inspiration from following code from GitHub repo: https://github.com/hankshyu/RISC-V_MAC.git
  - I tried these prompts in multiple LLMs, but none of them were able to do design this unit which can produce correct output. So I wrote most part of the code myself.
  - How do I convert float to 32-bit hex for input and back to float for checking result in cocotb?

- **Dot Product HW Accelerator**
  - Can you make a Verilog FSM that takes two 1D vectors (say patch and filter) and computes dot product using the MAC unit we designed earlier?
  - Write a Verilog testbench (not cocotb) for this dot product engine. Give fixed inputs and I'll verify the result manually.
  - Can you now generate some fixed random ~20 inputs and store them into register, then pass them one by one to the dut and write the generated output to a file called "dot_product_vectors.txt".
  - Now write a python script which reads the results from "dot_product_vectors.txt" and cross checks them with a sw version of results.

- **Matrix Dot Product HW Accelerator**
  - Now I want to extend the DotProduct unit to take two 2D matrices and compute their dot product using the same MAC unit. Can you write Verilog FSM for this?
  - Write a Verilog testbench for this MatrixMul module. Use 2 small 2x2 matrices and I'll check the result manually.
  - The FSM is not working, can you carefully check the state transitions for matrix row/column stepping. Also you need to give proper signals to the FSM which is present inside the DotProduct Engine. So make sure you are transitioning the proper signals in this FSM which the DotProduct Engine's FSM is expecting.
  - Add a "mul_done" signal in the FSM which gives a High pulse when multiplication is complete.
  - Can you extend the Verilog TB you wrote earlier to generate random 32 bit floating point values. First write a function which generates these values and then call it at the time of giving the stimulus. Give at least 10 random stimuli. Also dump the results in a file called "matrix_result_dump.txt".
  - Now write a python script which reads the results from "matrix_result_dump.txt" and cross checks them with a sw version of results.

- **Cocotb for MatrixMul**
    - Can you write a cocotb testbench for the MatrixMulEngine Verilog module? I want to compare the DUT result with NumPy matrix multiplication.
    - Show me how to convert float32 to hex and back in Python for cocotb testing.
    - Can you write a function which can generate random matrices with random 32 bit floating point values for given M, N and K sizes, A is MxK, B is KxN and results C is MxN.
    - Now try to automate the checking process and print a scoreboard at the end which shows summary of how many testcases passed.

- **SPI Interface**
    - Can you write a Verilog SPI module that sends and receives 32-bit float data? Design both Master and Slave modules.
    - Now can you write a top module which instantiates both Master and Slave in it so I can do a loopback test.
    - Give me a Verilog testbench for this SPI loopback. Just test a roundtrip of a single 32-bit float value Also modify the DUT to have it give a pulse on signal called “done”.
    - Can you write a cocotb testbench where the Master now is in python and Slave remains in the DUT.
    - The generated cocotb is not working, can you try to do one-on-one translation of the verlog TB you generated above to python? I see you have missed quite a few signals which you need to toggle at the right time.
    - How can I benchmark this SPI module? I want to benchmark it both ways for sending and receiving 32 bit Hex data. Can you modify the TB to display this at the end?

- **Final CNN Integration**
    - Now I want to integrate MatrixMulEngine and SPI. The design should take patches and filters from SPI and output the result back over SPI. Create a top module called “MatrixMul_top” in which you need to instantiate following modules – SPI_Loader, MatrixMulEngine, and SPI_Sender.
    - Can you write a Python cocotb to send two randomly generated matrices and get back the result and cross check them? Lets start small with both A & B being 2x2 matrices.
    - The generated results by HW are not matching the expected results form SW, can you help me identify where things might have gone wrong? Since we have independently verified the MatrixMulEngine, I don’t suspect any issues in that. So I believe something is wrong while sending / receiving the data.
    - ChatGPT was not able to figure out what is going wrong, so I opened the waveforms and noticed that SPI_Loader has miss-aligned the MSB while sending 32 bit hex values.
    - I opened the waveform and checked that there is miss-alignment at MSB for the data and it is not latching the correct value because of that. We have independently tested both SPI and MatrixMulEngine so none of them have any bugs, something is wrong while connecting them together. Can you re-check everything and try to figure out the issue with the MSB.
    - It was not able to figure out, so I manually figured out the issue and fixed the bit alignment issue. The issue was coz of sclk sampling, but for some god reason ChatGPT was hallucinating and not able to find this issue.

COCOTB Framework and Simulation Results:

Cocotb is a free, open-source, Python-based framework used for verifying hardware designs described in VHDL or Verilog. It stands for "COroutine-based COsimulation TestBench," and it allows users to create testbenches in Python to verify these designs.

I have made full use of cocotb framework here to test my HW<>SW flow together.

Python <-> RTL Integration:

- Data Flow:
    - Python writes matrices A and B to `input_buffer.txt`.
    - Python invokes the cocotb testbench via `make`.
    - The cocotb testbench (`test_matrix_mul_spi.py`) reads `input_buffer.txt`, drives the SPI signals to the Verilog hardware, and loads matrices A and B.
    - The hardware computes matrix C.
    - The testbench triggers the hardware to send matrix C over SPI.
    - The testbench writes matrix C to `output_buffer.txt`.
    - Python reads `output_buffer.txt` and returns C as a NumPy array.

- SPI Protocol:
    - Each matrix is preceded by a header word indicating which matrix (A or B), and its dimensions.
    - Data is sent/received 32 bits at a time, MSB first.
    - Matrix C is sent back in the same order.

Simulation Flow:

- User runs the infer mode with command like –

```
python3 CNN_digit_recognizer.py infer
../Generate_Modified_Images/Dataset_10x10/0/0_10.jpg
```

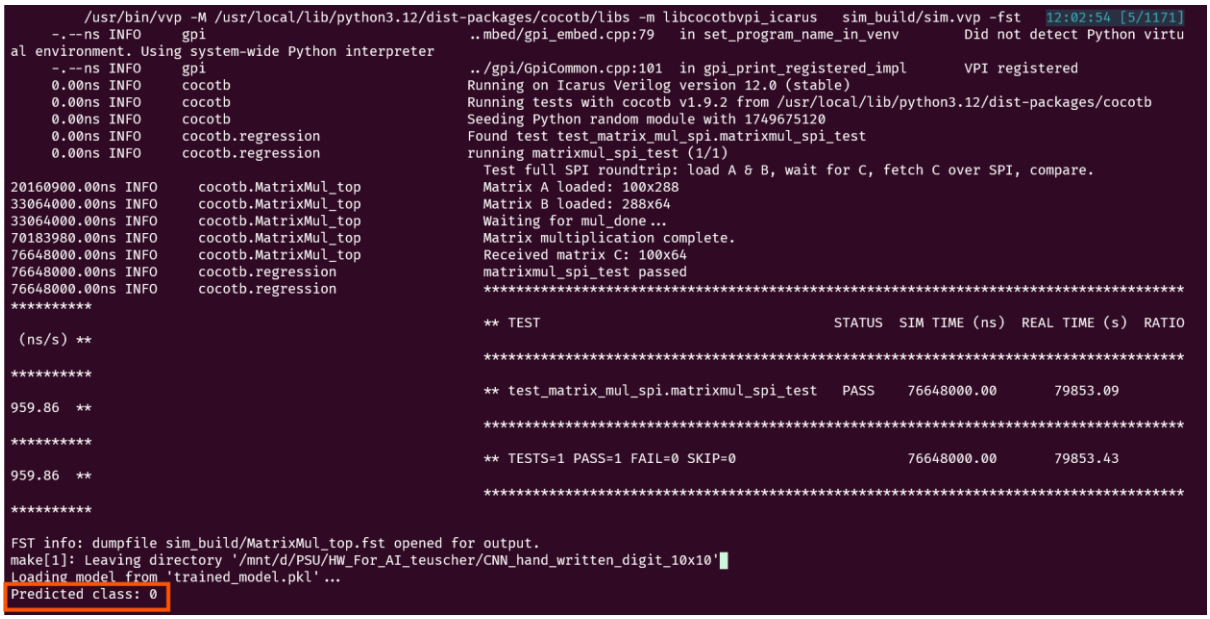
- When the CNN comes to the point to perform Matrix Multiplication it calls ‘matrix_mul_hw(A, B)’ which is provided by the HW Wrapper.
- matrix_hw_wrapper.py writes ‘input_buffer.txt’, runs ‘make’ (which launches cocotb and Verilog simulation).
- test_matrix_mul_spi.py (cocotb) reads ‘input_buffer.txt’, drives SPI to load A and B, waits for computation, triggers C transmission, and writes ‘output_buffer.txt’.
- matrix_mul_hw reads ‘output_buffer.txt’ and sends back the Matrix C result to the main CNN code.
- CNN Code receives the Matrix C result value and continues.
- Since there are 3 CNN layers, this operation is performed 3 times.

Simulation Results:

- I am able to test the full HW <> SW design together in the cocotb, below are the results –
- Below results are for 10x10 Image –
- Command used –

```
python3 CNN_digit_recognizer.py infer
../Generate_Modified_Images/Dataset_10x10/0/0_10.jpg
```

- Result (Full log is available here on GitHub: Link) –



- We passed the input image of Class 0 and the predicted result is 0, so that means our model is producing correct results.
- We can test similarly for other input images and model should produce correct results.
- We can also re-train this model for larger image sizes like 28x28 or 240x240.
- However, when I try to do so, the simulation takes lot of time and the Python Process gets killed. I even tried the PSU server machines, but I see the same issue –

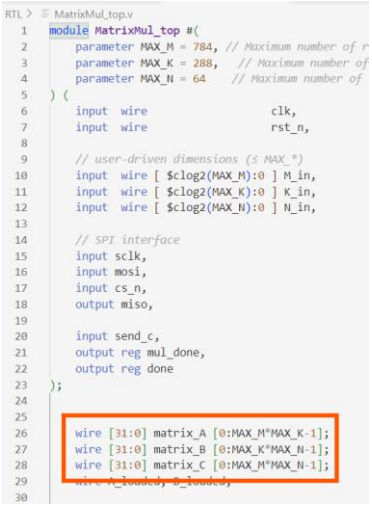
Synthesis Results:

- For Synthesis, I tried to use OpenLane-2 flow, however, OpenLane uses Yosys tool for Synthesis and unfortunately, Yosys doesn't support SystemVerilog language.
- Hence, I get following error (Full log is available here on GitHub: Link) –

```
                    10. Executing Verilog-2005 frontend:
                    /mnt/d/PSU/HW_For_AI_teuscher/OpenLane_Project_
                    v1/rtl/MatrixMulEngine.v
                    /mnt/d/PSU/HW_For_AI_teuscher/OpenLane_Project_
                    v1/rtl/MatrixMulEngine.v:24: ERROR: syntax
                    error, unexpected '[', expecting ',' or '=' or
                    ')'

[21:09:58] ERROR    Full log file:                                step.py:1372
                    'runs/RUN_2025-06-12_21-09-48/05-yosys-jsonhead
                    er/yosys-jsonheader.log'
Classic - Stage 5 - Generate JSON Header ━━━━━━━━━━━━━━━━━━━━━━━━  4/78 0:00:10
[21:09:58] WARNING  The following warnings were generated by the     flow.py:673
                    flow:
[21:09:58] WARNING  [Checker.LintWarnings] 512 Lint warnings found.  flow.py:675
[21:09:58] ERROR    The following error was encountered while    __main__.py:187
                    running the flow:
                    Generate JSON Header: subprocess (1,
                    ['yosys', '-y',
                    '/nix/store/ss2cw3sxbrwwx9jl0rrppbw4kgcmgi2n
                    -python3-3.11.9-env/lib/python3.11/site-pack
                    ages/openlane/scripts/pyosys/json_header.py'
                    , '--', '--config-in',
                    '/mnt/d/PSU/HW_For_AI_teuscher/OpenLane_Proj
                    ect_v1/runs/RUN_2025-06-12_21-09-48/05-yosys
                    -jsonheader/config.json', '--extra-in',
                    '/mnt/d/PSU/HW_For_AI_teuscher/OpenLane_Proj
                    ect_v1/runs/RUN_2025-06-12_21-09-48/05-yosys
                    -jsonheader/extra.json', '--output',
                    '/mnt/d/PSU/HW_For_AI_teuscher/OpenLane_Proj
                    ect_v1/runs/RUN_2025-06-12_21-09-48/05-yosys
                    -jsonheader/MatrixMul_top.h.json']) failed
[21:09:58] ERROR    OpenLane will now quit.                   __main__.py:188
```

- This error is because of 2D port declaration in my DUT RTL –



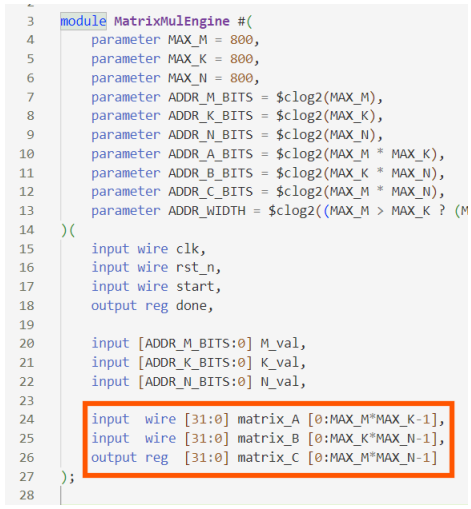
```
RTL > MatrixMul_top.v
module MatrixMul_top #(
    parameter MAX_M = 784, // Maximum number of r
    parameter MAX_K = 288,   // Maximum number of
    parameter MAX_N = 64    // Maximum number of
) (
    input  wire                 clk,
    input  wire                 rst_n,

    // user-driven dimensions (≤ MAX_*)
    input  wire [ $clog2(MAX_M):0 ] M_in,
    input  wire [ $clog2(MAX_K):0 ] K_in,
    input  wire [ $clog2(MAX_N):0 ] N_in,

    // SPI interface
    input sclk,
    input mosi,
    input cs_n,
    output miso,

    input send_c,
    output reg mul_done,
    output reg done
);

    wire [31:0] matrix_A [0:MAX_M*MAX_K-1];
    wire [31:0] matrix_B [0:MAX_K*MAX_N-1];
    wire [31:0] matrix_C [0:MAX_M*MAX_N-1];
```



```
module MatrixMulEngine #(
    parameter MAX_M = 800,
    parameter MAX_K = 800,
    parameter MAX_N = 800,
    parameter ADDR_M_BITS = $clog2(MAX_M),
    parameter ADDR_K_BITS = $clog2(MAX_K),
    parameter ADDR_N_BITS = $clog2(MAX_N),
    parameter ADDR_A_BITS = $clog2(MAX_M * MAX_K),
    parameter ADDR_B_BITS = $clog2(MAX_K * MAX_N),
    parameter ADDR_C_BITS = $clog2(MAX_M * MAX_N),
    parameter ADDR_WIDTH = $clog2((MAX_M > MAX_K ? (M
)(
    input wire clk,
    input wire rst_n,
    input wire start,
    output reg done,

    input [ADDR_M_BITS:0] M_val,
    input [ADDR_K_BITS:0] K_val,
    input [ADDR_N_BITS:0] N_val,

    input  wire [31:0] matrix_A [0:MAX_M*MAX_K-1],
    input  wire [31:0] matrix_B [0:MAX_K*MAX_N-1],
    output reg  [31:0] matrix_C [0:MAX_M*MAX_N-1]
);
```

- These are the most important ports in my design and since I am dealing with 2D matrices to begin with, I had created 2D ports in my design.
- I tried asking ChatGPT to flatten them, but the generated RTL is not correct and it messed-up with the bit alignment. Considering the timeline of the project, I decided to skip OpenLane-2 and used Synopsys Design Compiler instead.

Synopsys Design Compiler Synthesis Results:

- I am able to completely Synthesize the RTL using DC and here are the results.
- All the detailed logs and results are available here in this GitHub Repo – Link
- I am using ***osu05_stdcells*** library which is very old and has generic ***0.5µm*** technology.

Metric	Value
Timing – WNS	Slack met with 30 ns clk i.e. 33.33 MHz
Total Power	51.4263 W
Combinational area:	853694451.000000 μm^2
Buf/Inv area:	146932992.000000 μm^2
Non-combinational area:	830151216.000000 μm^2
Total cell area:	1683845667.000000 μm^2

Vivado Synthesis Results:

- I synthesized the design to Bitstream for FPGA as well, since I plan to run this design later on an FPGA.
- I used Nexsys4 board with Artix-7 FPGA as a target and here are the results.
- All the detailed logs and results are available here in this GitHub Repo – Link

Metric	Value
Timing – WNS	0.124 (Timing met with clk period *50 ns* i.e. *20 MHz*)
Total Power	0.13 W
Area – LUT	5760 (9%)
Area – DFF	9995 (8%)
Area – DSP	1 (1%)

Results and analysis (for 10x10 image size):

- The SPI Benchmark shows I can send and receive 32 bits data in – 2.74 ms
- For details on SPI benchmarking, checkout this repo - W08_C25_SPI_for_CNN
- So one-way latency per word = 2.74 ms / 2 = 1.37 ms
- I am able to synthesize the design with DC at clock period – 30 ns
- My DUT can finish 1 Matrix Multiplication in 1 clock cycle.
- Since I have 3 CNN layers, it takes 3 clock cycles for my DUT to finish multiplication of 3 matrices.
- The current Software runtime for multiplication of 3 matrices is – 0.00136 sec
- So speedup –

  Speedup = Total SW time / Total HW time

  Speedup = 0.00136 sec / (3 x 30 ns)

  $= 1.36 \times 10^{-3} / 9 \times 10^{-8}$

  = 15,111.11
- So we have **acceleration of ~15,111x** over the software without considering SPI timings.
- The Dimensions of 3 Matrix Multiplications are –
  - Matrix Mul SW: A shape: (100, 9), B shape: (9, 8), received C shape: (100, 8)
  - Matrix Mul SW: A shape: (100, 72), B shape: (72, 32), received C shape: (100, 32)
  - Matrix Mul SW: A shape: (100, 288), B shape: (288, 64), received C shape: (100, 64)
- Which is –
  - (100 x 9) + (9 x 8) + (100 x 8) = 1,772 words
  - (100 x 72) + (72 x 32) + (100 x 32) = 12,704 words
  - (100 x 288) + (288 x 64) + (100 x 64) = 53,632 words
- So, total data being transferred over SPI –
  - 1772 + 12704 + 53632 = 68,108 words
  - **So we are transferring *266 KBytes* of data over SPI**
- Hence, total time required to transfer/receive this data over SPI –
  - 68108 x 1.37 ms = 93307.96 ms = 93.30796 sec
  - We need to add this time in the Total HW time.
- Hence total HW time –
  - 9×10^{-8} + 93.30796 sec = 93.308 sec = 1.55 mins
- So speedup –

  Speedup = Total SW time / Total HW time

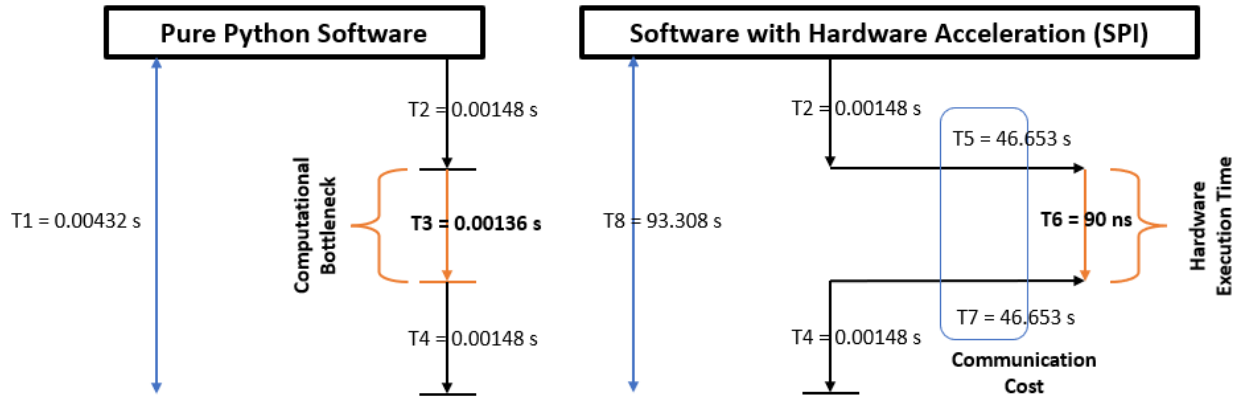
  Speedup = 0.00136 sec / 93.308 sec

  $= 1.4575 \times 10^{-5} = 0.000014575$
- That means HW (with SPI) is **0.000014575x times slower** than SW.
- That is because SPI is a really slow interface and it is not really ideal to transfer this much data (266 KB) over.
- If we consider a much more practical solution like PCI-E, which has –
  - PCIe Gen3 x4 throughput:
  - Per lane: 8 Gbps (raw), effective ~7.877 Gbps
  - x4 lanes: 4 × 7.877 Gbps = ~31.5 Gbps (SPI Throughput is 11.5 Mbps)
  - In bytes/sec = 3.94 GBytes/sec

- If we plug in this number in our equations –
    - Total Transfer time over PCI-E = 266 x 10^3 Bytes / 3.94 x 10^9 Bytes/sec = 0.0000675127 = 67512.7 nano sec
    - Total HW time = 67512.7 ns + 90 ns = 67602.7 ns
- Acceleration with PCI-E –
    - Speedup = Total SW time / Total HW time
    - Speedup = 0.00136 sec / 0.0000675127 sec = 20.144x
    - **So, the HW is approx. ***20 times*** faster than SW**

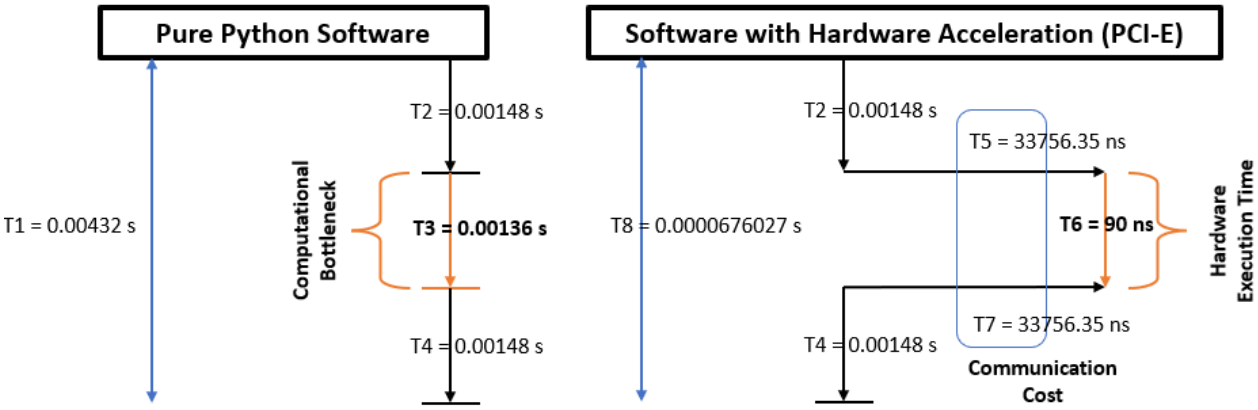
Conclusion on Acceleration:

1. With SPI –



With SPI, HW is **0.000014575x times slower** than SW.
i.e. With SPI, ***SW is 68,594 times faster than HW***.

2. With PCI-E –



With PCI-E, HW is ***~20 times faster*** than SW.

- Quick Summary Table –

Standalone Hardware	***15,111* times Faster** than Software	This doesn't include communication cost, so communication channel overhead is 0% here
Hardware + SPI	***0.000014575* times slower** than SW ***SW is 68,594 times faster than HW***	This is because of the communication overhead added by SPI is way more than Acceleration achieved by pure HW
Hardware + PCI-E	***20 times* faster than SW**	PCI-E gives has less overhead compared to SPI, hence we can see some acceleration happening over here.

Related work

1. DSLR-CNN: Efficient CNN Acceleration using Digit-Serial Left-to-Right Arithmetic
    a. Authors: Malik Zohaib Nisar et al.
    b. Key Idea: Digit-serial left-to-right arithmetic for convolution, enabling digit-level pipelining.
    c. Findings: Synthesized in 45 nm tech, achieves 4.37×–569× higher GOPS and 3.58×–44.8× higher TOPS/W vs. conventional bit-serial designs
    d. Link: arXiv
2. A high-speed reusable quantized hardware accelerator for CNN on constrained edge device
    a. Authors: Rama Muni Reddy Yanamala & Muralidhar Pullakandam
    b. Key Idea: FPGA accelerator (PYNQ-Z2) using quantized arithmetic, pipelining, loop unrolling, array partitioning.
    c. Findings: ~92.7% faster than Intel Core i3 CPU and ~90%+ faster than GPU/K80. Delivers 4.4 GOP/s, ~2× faster than conventional single-PE hardware.
    d. Link: springer
3. FPGA-Based Neural Network Acceleration for Handwritten Digit Recognition
    a. Authors: Eshel19
    b. Key Idea: LeNet-5 on FPGA via Vivado HLS for MNIST acceleration.
    c. Findings: Achieves 61.6 GFLOPS at 100 MHz, outperforming GPUs in throughput. Professional-grade FPGA acceleration of digit recognition.
    d. Link: GitHub

How my work is different

1. I am using full 32-bit IEEE-754 Floating Point standard, so this design has high accuracy, which is very close to actual CPU compared to the implementation by others.
2. My CNN code is fully modular and can be easily retrained to variety of different image sizes. I have extensively benchmarked the bottleneck with different image sizes and plotted a graph.
3. I am using 3 CNN layers, so the matrix multiplication of patch and filter happens 3 times in total.
4. The model can be trained to digit any image (like characters and patterns) and not just limited to numbers.

Future work

1. Experiment with precision –
    a. Because of 32-bit Floating Point precision, I need to send 266 KBytes of data over my communication channel, which is much higher compared to others.
    b. If I reduce the precision to 8-bit / 4-bit, I need to send only –
        i. 68108 x 8 = 544864 bits = 66 KBytes for 8-bit precision
        ii. 68108 x 8 = 272432 bits = 33 KBytes for 4-bit precision
    c. This will significantly reduce my communication overhead, and make –
        i. SPI –
            1. Comm time = 66 KBytes/1437.5 KBytes/sec = 0.0459130434 sec
            2. Comm time = 33 KBytes/1437.5 KBytes/sec = 0.0229565217 sec
        ii. PCIE-E –
            1. Comm time = 66 KBytes/3.94 x 10^6 KBytes/sec = 16751.269 ns
            2. Comm time = 33 KBytes/3.94 x 10^6 KBytes/sec =8375.634 ns
2. Take this design to FPGA
    a. I have already synthesized the design in Vivado for Nexys4 Board which uses Artix-7 FPGA.
    b. Will work on a different communication channel which is supported by FPGA to infer this design on this board.
3. Add pipelining and systolic array like structure for MatrixMulEngine which can improve the HW runtime even less than 90ns.