**Project 3 – Using Vitis HLS to Implement Matrix Multiplication**

**Report for ELEC 522**

**College of Engineering and Computer Science**

**Rice University**

**Houston, TX**

**28th October, ‘2022**

**Karan Venaik**

**I. INTRODUCTION/PROJECT DESCRIPTION**

The objective of this project is analogous to that of Project #2 i.e., design a systolic matrix multiplier capable of multiplying two 4x4 matrices. The multiplier must utilize a maximum of 16 DSP elements/multiply-accumulate blocks. The multiplier should also load and unload data systolically i.e., take the input and pass it through the compute block.

**III. THEORETICAL BACKGROUND**

The HLS matrix multiplication was implemented by modifying the given matrix multiplication code. The given matrix multiplication program has three for loops. The three for loops are as follows:

1. Row iterator: Iterates through the rows of input matrix A,
2. Column iterator: Iterates through the columns of input matrix B,
3. Resultant iterator: Completes the inner product of row from input matrix A, column from input matrix B.

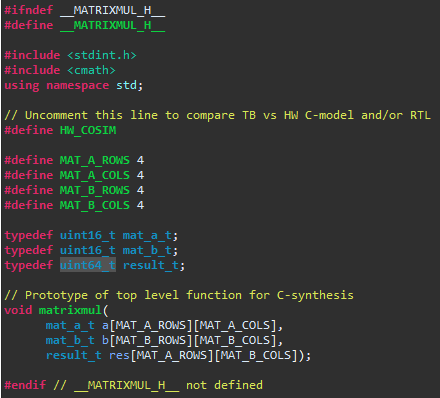
Vitis HLS allows for the use of “pragmas” which act as compiler comments/directives to modify/implement the user-written C/C++ code to optimize for latency, resources, throughput etc.

The pragmas used in the lab are as follows:

1. HLS INTERFACE AP\_FIFO: Defines RTL port with a standard FIFO Interface
2. HLS ARRAY\_RESHAPE: Converts a vector into an array allowing for parallel access to data.
3. HLS PIPELINE: Allows for concurrent execution of operations by processing new inputs every N clock cycle.
4. HLS UNROLL: Performs loop unrolling by creating multiple copies of the loop body with pre-incremented variables during compile time.

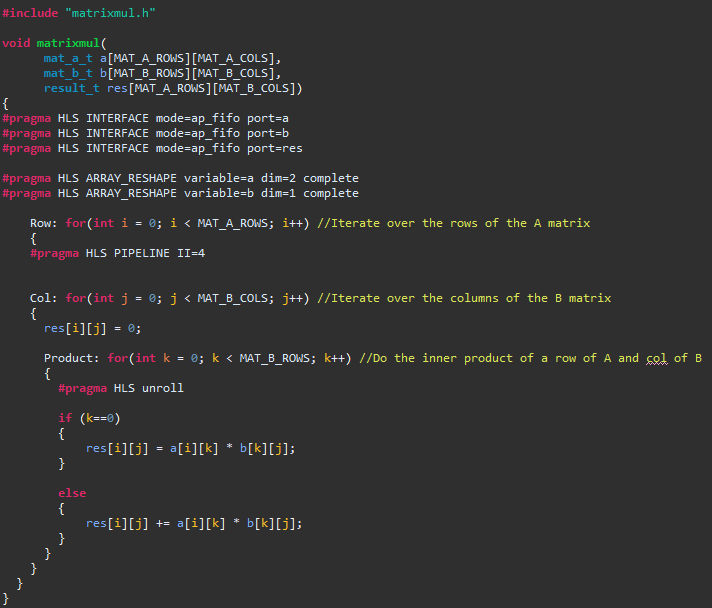
**III. DESIGN**

The matrix multiplication was achieved by modifying the original stub provided as part of the assignment.



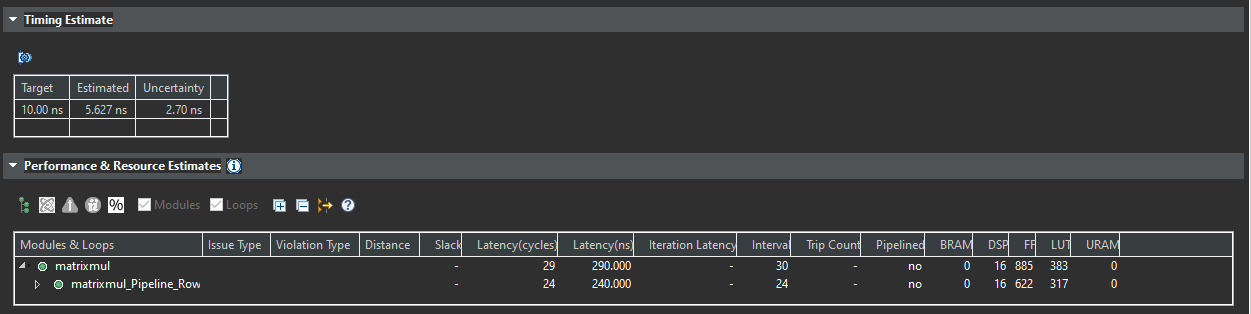
**Figure #1:** Screen capture of modified matrixmul.h (header) file.

As can be seen in fig. #1, the header file was modified to allow for matrices of size 4 (I.e., 4x4 = 16 elements). Additionally, the input matrices were set to allow unsigned, 16b wide elements, but resultant matrix was set to 32b wide to allow for larger product elements (since the product of two 16b unsigned integers may be greater than 16b wide).



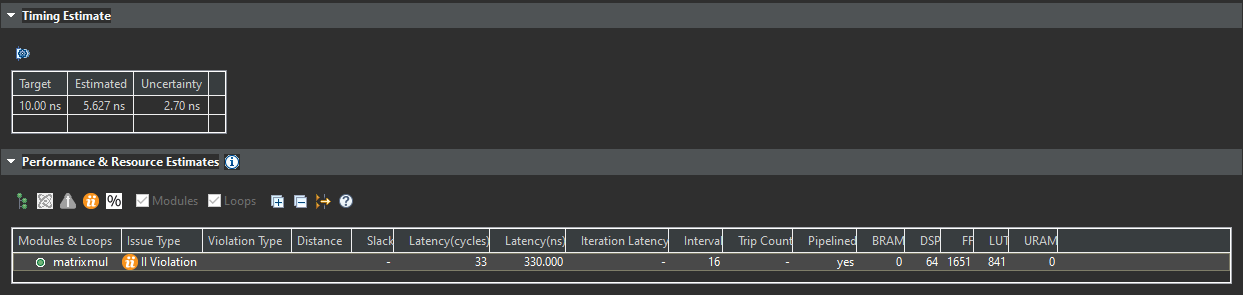
**Figure #2:** Screen capture of modified matrixmul.cpp (actual c++) file.

In the actual matrixmult function shown in fig. #2, we can notice the changes made from the original 3x3 multiplication function provided. The ports are set using the HLS INTERFACE pragma, which allows for FIFO usage. Additionally, the input arrays are reshaped. Pipelining and loop unrolling can be done at various stages for experimentation. Fig. #3 shows the latency and resource utilization when the body of the row iterator is pipelined.



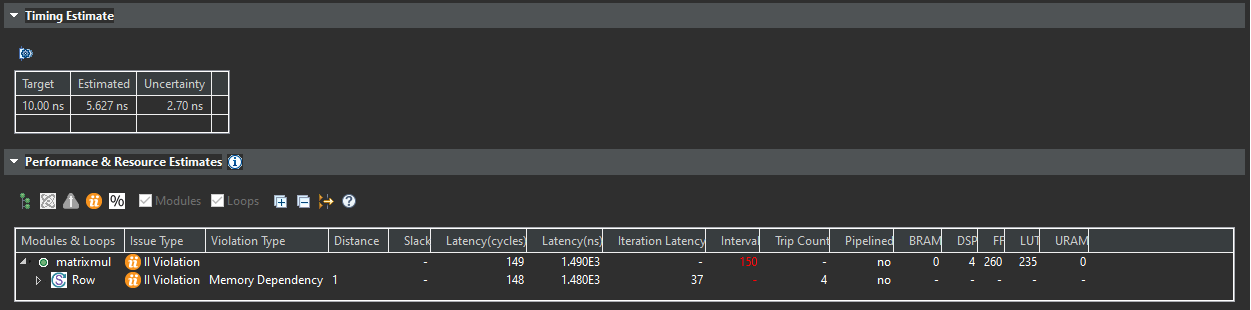
**Figure #3:** Screen capture of synthesis report when column iterator is pipelined.

We can alternatively test by pipelining the entire program (all three for loops – introducing the pragma before the row iterator). The latency and resource utilization of this approach is shown in fig. #4.



**Figure #4:** Screen capture of synthesis report when all iteration loops are pipelined.

As can be seen from the results in fig. #4, the number is DSP units is quadrupled while also causing a pipelining violation. Additionally, the latency is increased from the previous approach. Lastly, we can consider pipelining only the product loop/Column loop body. The latency and resource utilization of this approach is shown in fig. #5.

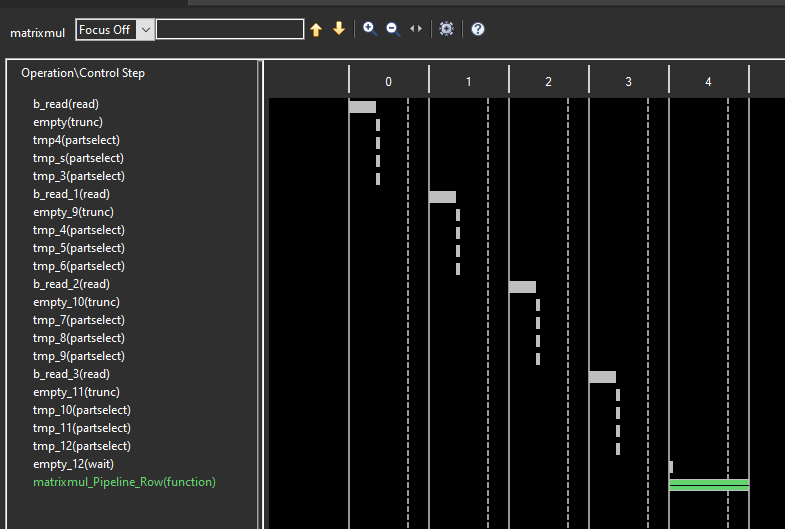


**Figure #5:** Screen capture of synthesis report when only the innermost/multiply-accumulate loop is pipelined.

Pipelining only the innermost loop results in the same outcome as pipelining all loops – pipeline violation with increased latency. However, the number of DSPs is greatly reduced.

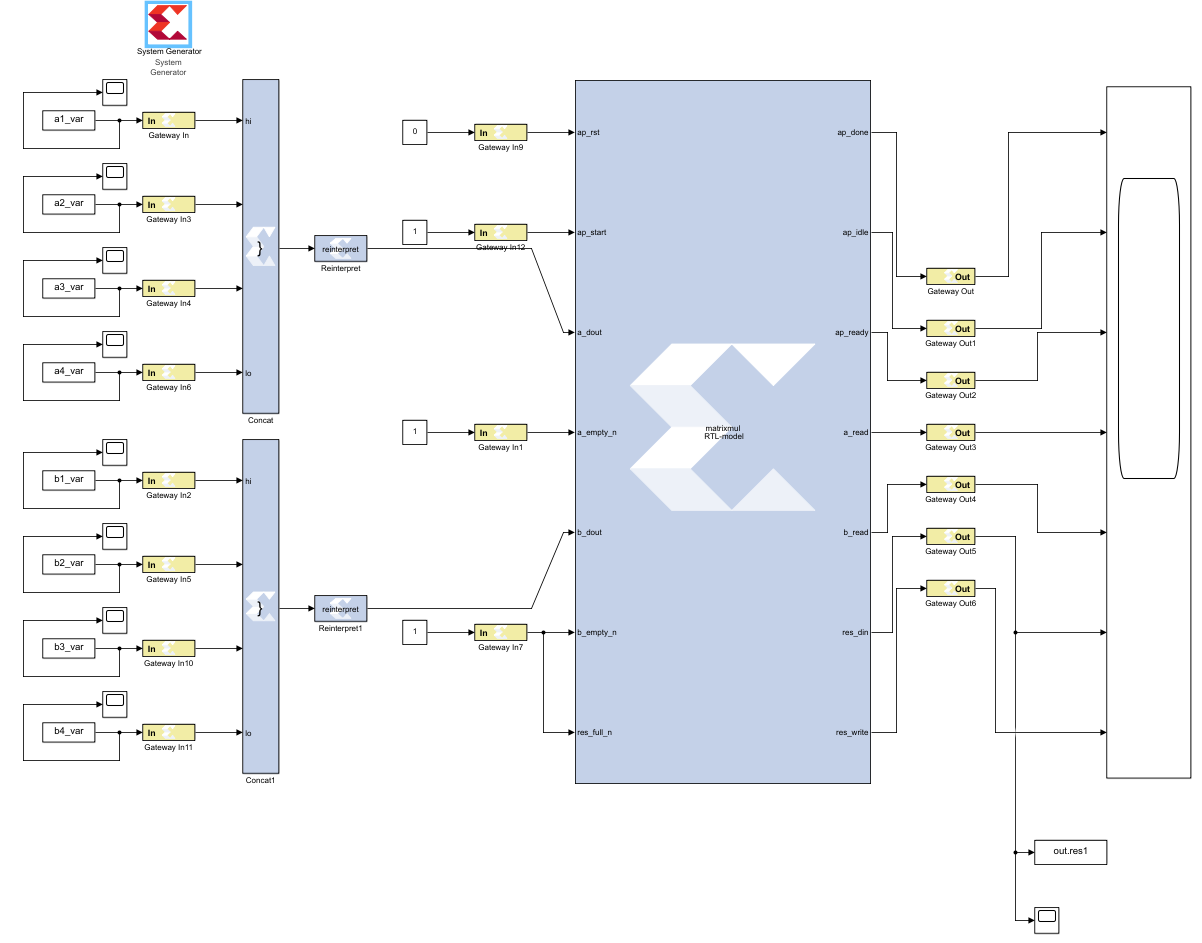
Alternatively, I also experimented with removing the HLS UNROLL pragma for the innermost loop while pipelining the column iterator. This resulted in no difference in performance, latency, and resource utilization presumably due to the pipeline pragma already unrolling the loops. Thus, the UNROLL pragma was removed for the final design.

For program testing, the supplied testbench was used. However, the values were modified to ensure that both software test and hardware cosimulation results were in line with each other and as expected.

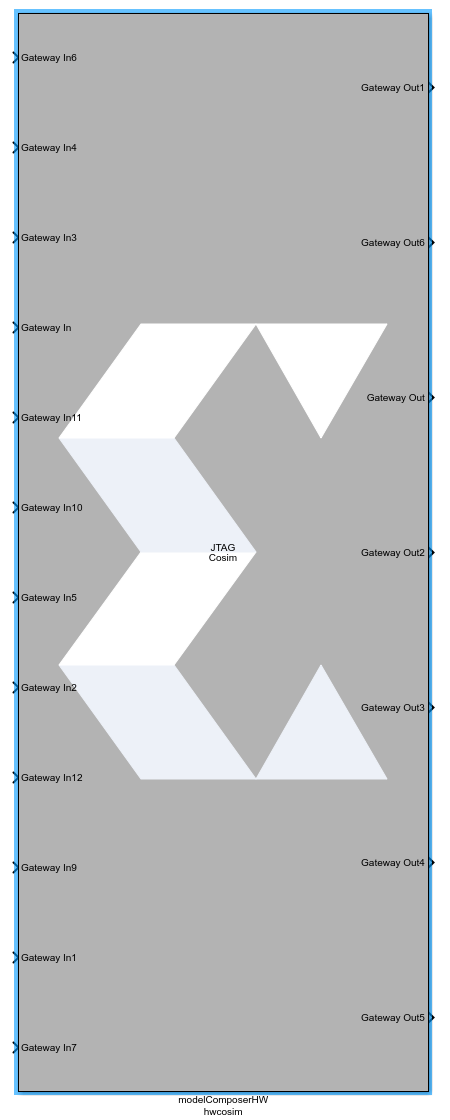


**Figure #6:** Screen capture of schedule viewer when column iterator is pipelined.

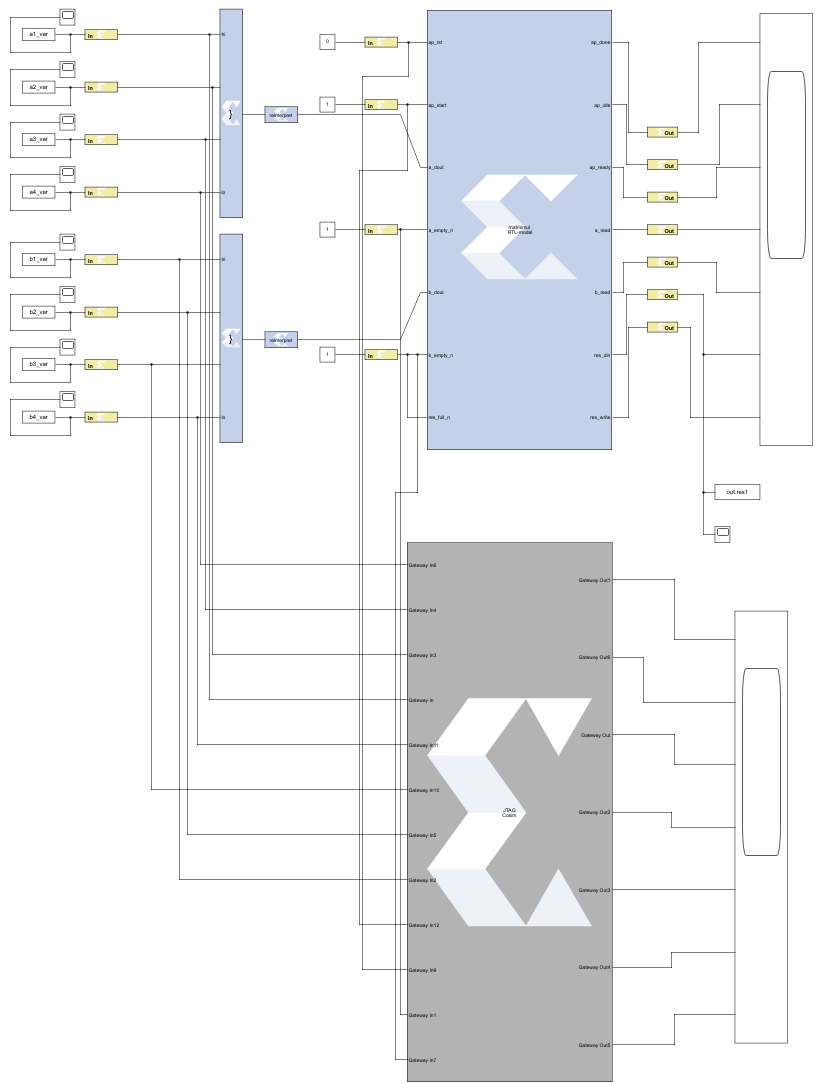
The design was exported from Vitis HLS and imported into Model Composer. Fig. #7 shows the system as wired in Model Composer. The data is loaded from the MATLAB workspace using Xilinx Gateway In blocks. Each Gateway In block samples the vector elements as a fixed-point, 2’s complement, 16-bit wide element. Each A matrix is split into its row vectors. Similarly, each B matrix is transposed and divided into row vectors. After calculation, finalized multiply accumulated data is unloaded to the left in vector forms using the Xilinx Gateway Out block.

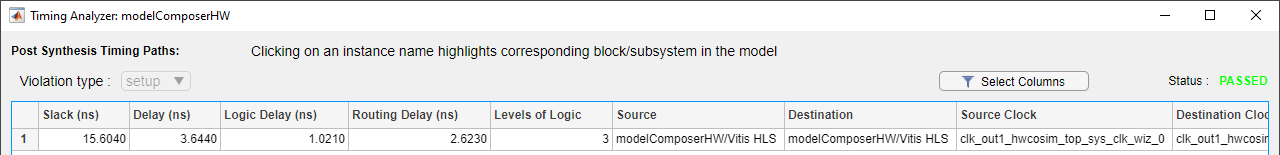


**Figure #7:** Model Composer Diagram of HLS block with code from Vitis HLS

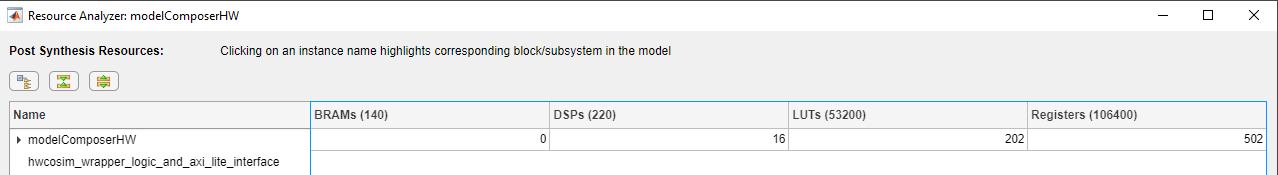


**Figure #8:** HW Cosim block generated by Model Composer



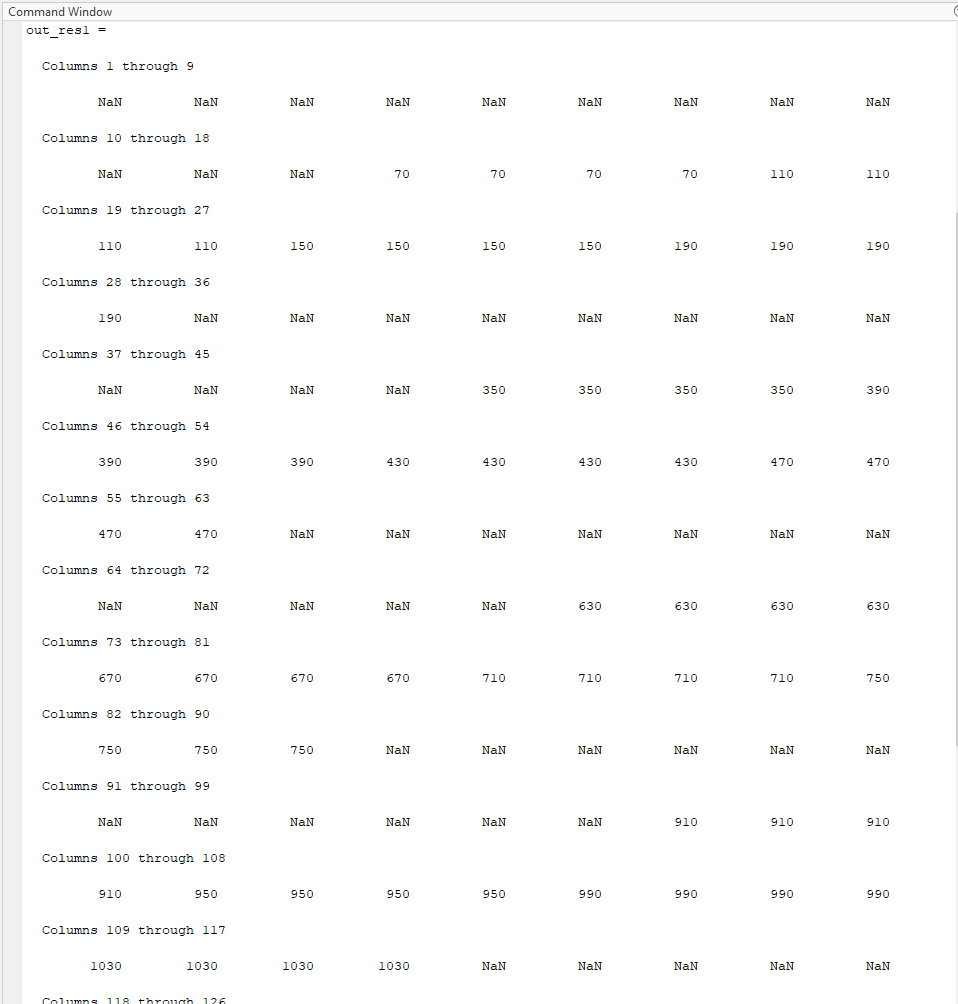


**Figure #9:** Screen Capture of HW Cosim Timing Analyzer generated by Model Composer



**Figure #10:** Screen Capture of HW Cosim Resource Analyzer generated by Model Composer

On running the simulation in model composer, the output generated seems to be incorrect for the data provided. Fig. #11 shows the output as generated by Model Composer. This could be due to errors in the Vitis HLS matrix multiplication code, or in the extraction of from the system using MATLAB. When supplying the same input data to the Vitis HLS model, the correct matrix multiplication results are obtained. Same is true of the hardware cosimulation when performed on Vitis HLS. However, when performing simulation and hardware cosimulation in Model Composer, the resultant output appears incorrect. Thus, this can be attributed to incorrectly reading the output from the Gateway outs. However, the products are systolic and generated correctly as evidenced by Vitis HLS runtime.



**Figure #11:** Screen Capture of HW Cosim Results from Model Composer.

There is substantial difference in the design flows between the projects when implemented using Model composer vs. Vitis HLS. At the completion of this project, I can safely say that I prefer Model Composer. This is due to the visual nature which allows the user to easily identify the data flow as it is happening in the systolic array. For example, as shown in the Lab #2 report, the systolic array when modelled in Model Composer occupies a footprint that is very similar to the topological design of a systolic array where data flows in from the top and the left, and is unloaded to one of the sides (left, right, diagonal). This approach is much more suited to a novice/beginner such as myself. This allowed me to reach a design that was more optimized (Better resource utilization, higher clock speed, more parallel). However, in Vitis HLS, the dataflow between arrays must be considered more thoroughly to avoid conflicts/dependencies/violations which is unintuitive. On the other hand, a graphical interface like Model Composer is especially cumbersome when dealing with larger projects and does not scale well.