

Part 1:

The high level overview of our testbench file is that the testbench is designed to verify the correctness of the matrix multiplication function ``matrix_mult`` implemented in the ``matrix_mult.h`` header file.

1. Setup:

- The ``main`` function initializes matrices for two tests: one with matrices filled with all ones and the other with matrices filled with random integer values.

2. Matrix Initialization:

- Matrices ``a`` and ``b`` are filled with ones for the first test.
- Matrices ``rand_a`` and ``rand_b`` are filled with random integer values between 0 and 9 for the second test.

3. Matrix Multiplication:

- The ``matrix_mult`` function is called with the matrices initialized in the previous step to perform matrix multiplication.
- For the random matrix test, the Boost library is used to perform matrix multiplication (``boost::numeric::ublas::prod(rand_a, rand_b)``).

4. Conversion and Verification:

- The resulting matrices from both the custom implementation and the Boost library multiplication are converted to the same format (``mat_prod``).
- The converted matrices are then compared element-wise with the expected result matrices (``prod_expected`` and ``rand_prod_expected``).
- If any element of the resulting matrix differs from the expected result, the corresponding test is marked as failed.

5. Print and Report:

- The test bench prints whether each test passed or failed based on the comparison results.
- It outputs "BASIC TEST PASSED" or "BASIC TEST FAILED" for the test with matrices filled with ones and "RANDOM TEST PASSED" or "RANDOM TEST FAILED" for the test with random matrices.

Output:

```
[stiruchi@c024:~/Desktop/Lab 2]$ ./matrix_mult.out
BASIC TEST PASSED
RANDOM TEST PASSED
```

Timing Estimate

Target Estimated Uncertainty

10.00 ns	6.343 ns	2.70 ns
----------	----------	---------

Performance & Resource Estimates

Modules & Loops

Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP
matrix_mult	II Violation			-	2057	2.057E4		2058	-	no	0	8
Row_Col	II Violation	Resource Limitation		-	2055	2.055E4	16	8	256	yes	-	-

HW Interfaces

AP_MEMORY

HW Interfaces

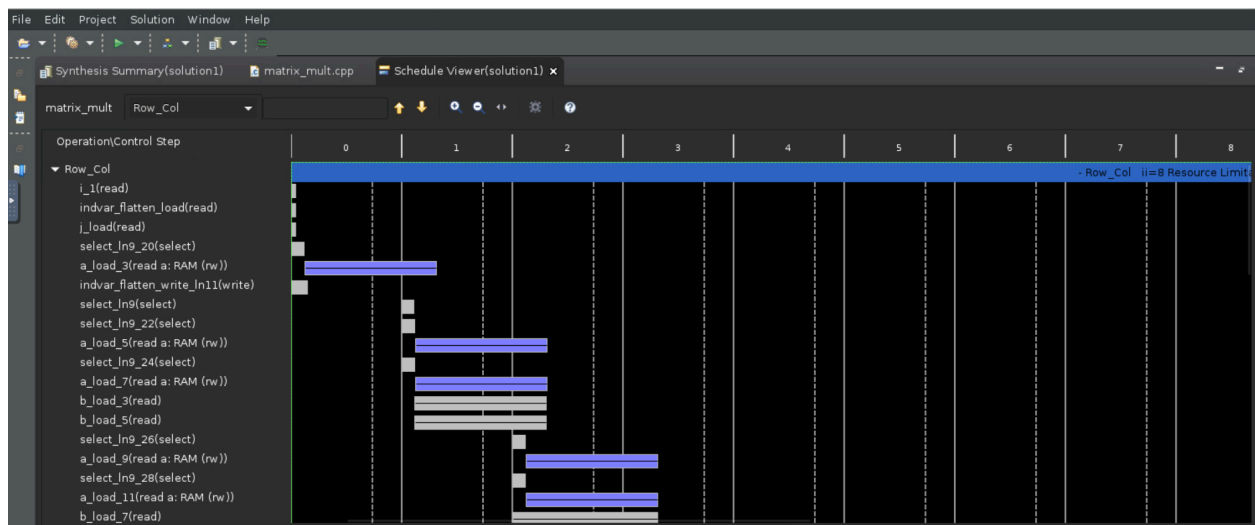
AP_MEMORY

Interface	Bitwidth
a_address0	8
a_address1	8
a_q0	8
a_q1	8
b_address0	8
b_address1	8
b_q0	8
b_q1	8
prod_address0	8
prod_d0	32

TOP LEVEL CONTROL

Interface	Type	Ports
ap_clk	clock	ap_clk
ap_rst	reset	ap_rst
ap_ctrl	ap_ctrl_hs	ap_done ap_idle ap_ready ap_start

Snapshot of Schedule Viewer:



Verilog Files Generated:

Matrix_mult_flow_control_loop_pipe.v: This Verilog module implements the control logic necessary to coordinate the matrix multiplication operation, including initialization, start, completion, and continuation signals.

Matrix_mult_mac_muladd_8ns_8ns_16ns_17_4_1.v: This Verilog code defines a matrix multiplication module optimized for efficient hardware implementation using DSP48 slices available in Xilinx FPGAs.

Matrix_mult_mul_8ns_8ns_16_1_1.v: This Verilog module implements a simple multiplier for fixed-point multiplication, where the inputs are sign-extended and multiplied to produce the output. It is parameterized to support different input and output widths.

Matrix_mult.v: Main matrix multiplier file

RTL Design being verified by co-simulation:

Co-simulation provides a robust method for verifying RTL designs by comparing them against equivalent high-level models, ensuring functional correctness and validating the hardware

implementation against the original algorithmic behavior. The steps taken to ensure that are: RTL generation, testbench setup, simulation environment, and data exchange. First the RTL gets generated by the high level c++ code. Second, the test bench is created to see if the RTL is working properly. Both codes are simulated on the test bench. Output is verified in the data exchange to make sure RTL matches the high level output.

Snapshots of output and waveform:

```
## run all
/////////////////////////////////////////////////////////////////
// Inter-Transaction Progress: Completed Transaction / Total Transaction
// Intra-Transaction Progress: Measured Latency / Latency Estimation * 100%
//
// RTL Simulation : "Inter-Transaction Progress" ["Intra-Transaction Progress"] @ "Simulation Time"
/////////////////////////////////////////////////////////////////
// RTL Simulation : 0 / 2 [0.00%] @ "125000"
// RTL Simulation : 1 / 2 [100.00%] @ "20695000"
// RTL Simulation : 2 / 2 [100.00%] @ "41255000"
/////////////////////////////////////////////////////////////////
$finish called at time : 41315 ns : File "/home/stiruchi/Desktop/Vitis/Lab2/hls_gemm/solution1/sim/verilog/matrix_mult.autotb.v" Line 337
```

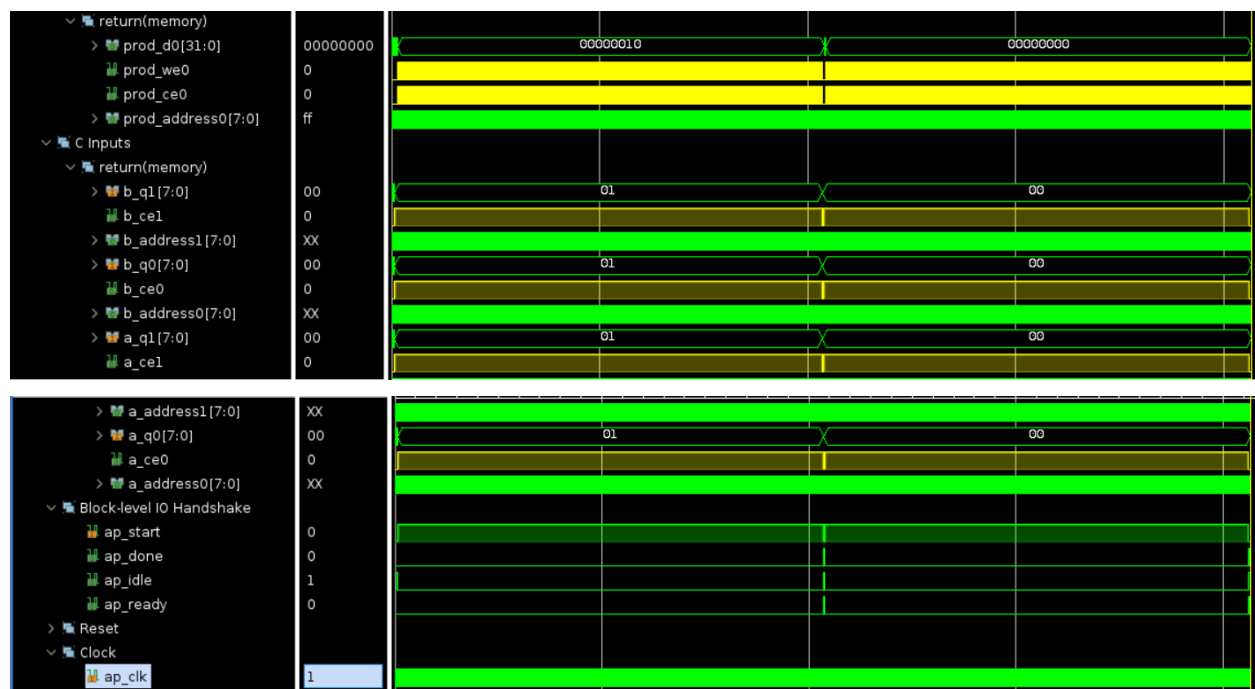


Table Comparing HLS and RTL:

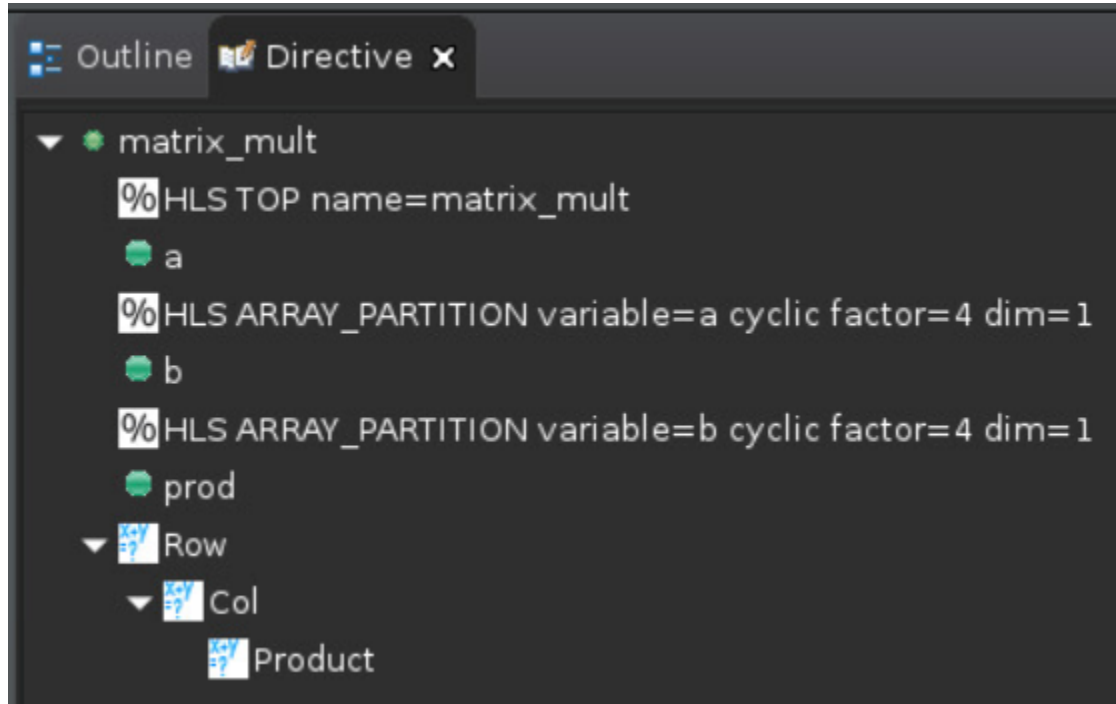
	HLS	RTL
Timing Estimate	6.343 ns	6.594 ns
DSP	8	16
FF	376	214
LUT	1264	280

Part 3:

Architectural Alternatives Tested:

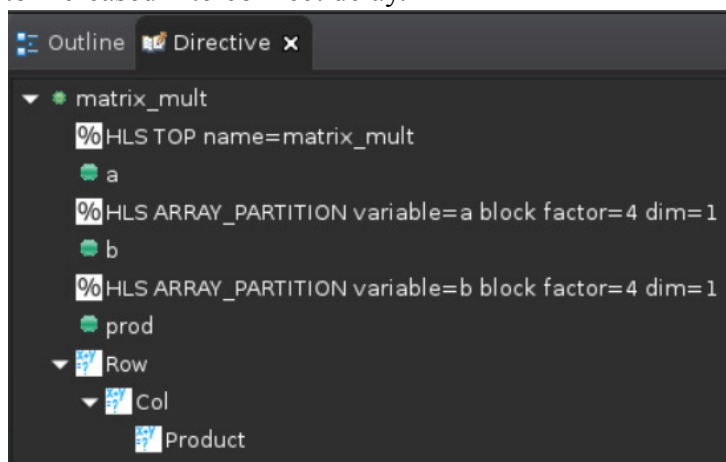
AP: Cyclic:

We started out by testing each array partition. Cyclic was first. We tested 4 as our main factor due to the arrays being in 16x16. We wanted to test the square of the dimensions. We also only did the a and b arrays since we were only writing for the prod array. The result gave us slightly better timing, but the overhead in LUTs and FFs did increase. Latency and throughput were the same. This is in line with the benefits of cyclic partitioning, better timing but more overhead.



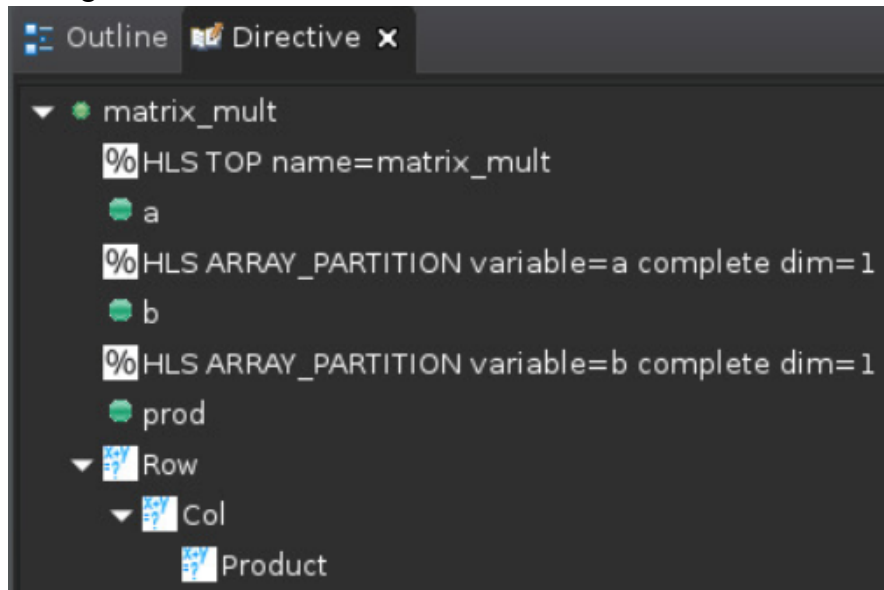
AP: Block:

Block array partitioning was next. Same factor value and arrays tests. The result gave us the same results as cyclic but with worse clock frequency than even the base. This was probably due to increased interconnect delay.



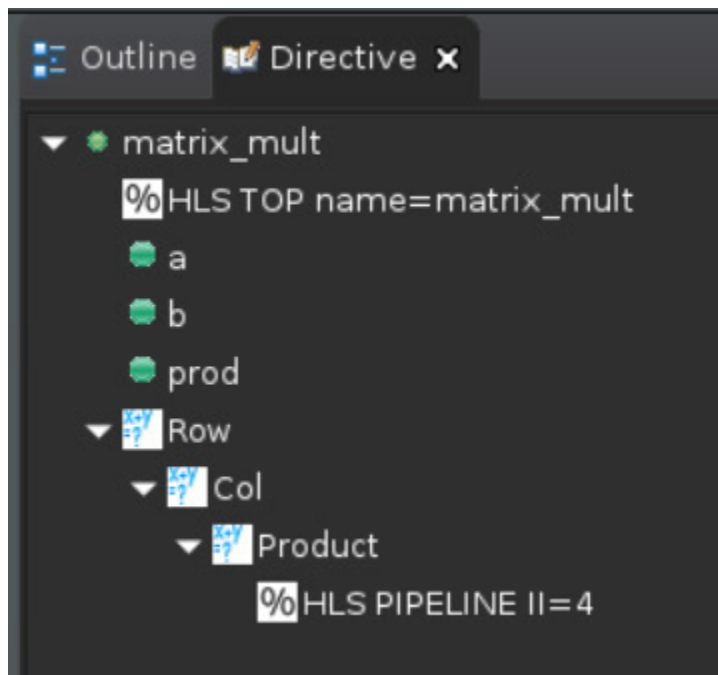
AP: Complete:

Complete was the last one tested. Using the same two arrays as the previous array partitioning. We saw a much better result. Though we had a major increase in overhead in LUTs and FFs due to how many components were split, we decreased latency and throughput by almost 90%. Timing was the same as baseline. This was a tradeoff to consider for further solutions.



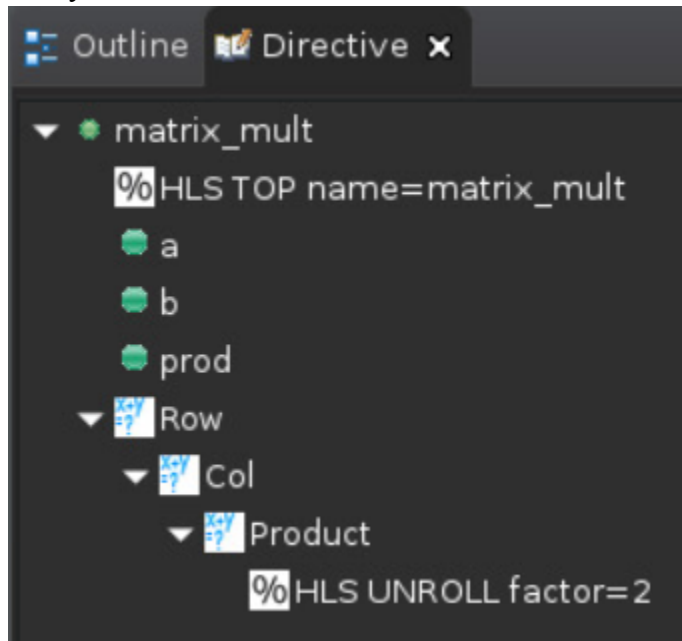
Pipelining:

Next was trying out pipelining. Used the innermost loop which was product and gave it a II value of 4. This gave us lower overhead, higher latency and throughput but a lower clock frequency. We also only saw the use of one DSP instead of 8 which was good. The results came out as such since pipelining has increased parallelism. We use less resources and faster cycles, but a longer final result time.



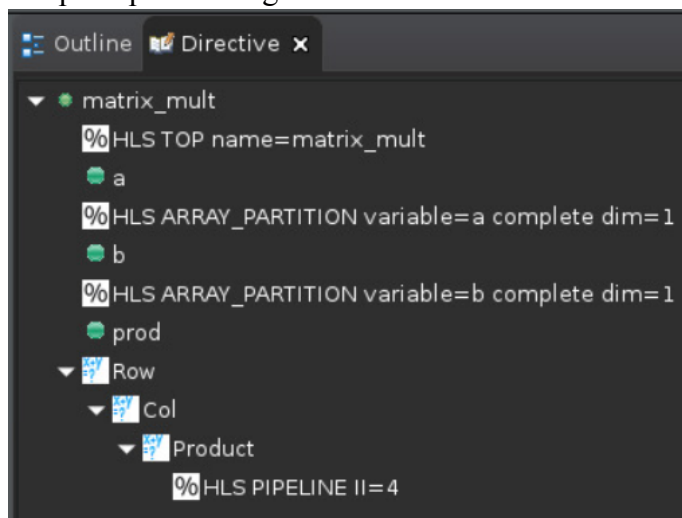
Unrolling:

For unrolling we tried it out with a factor of two for the prod loop trying to get lower latency from the baseline if possible. Instead, we did get lower overhead, lower DSP usage, and a lower clock frequency. Unrolling had a slightly higher latency probably from the lower overhead, but not by much from the baseline.



Complete + Pipelining:

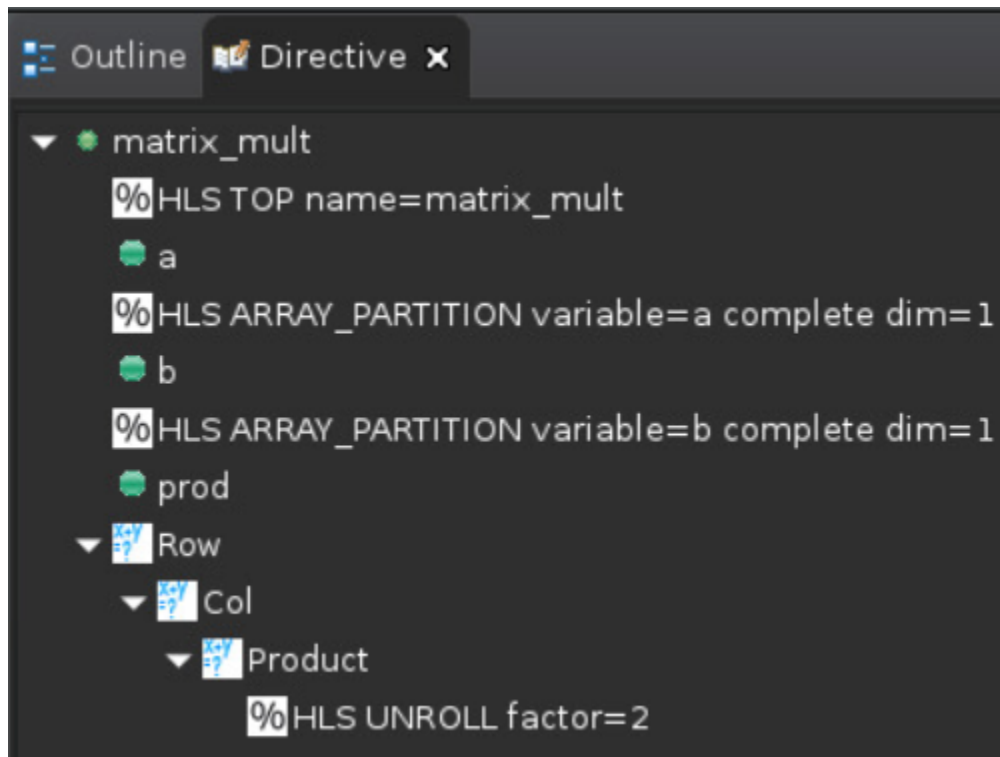
The next solution we tried was taking the best partitioning and testing it against pipelining and unrolling. We started with pipelining with a II value of 4 and complete array partitioning for `a` and `b`. The results were the same latency, throughput, DSP usage, and clock frequency as pipelining, but slightly higher overhead. The reason for the higher overhead was due to the complete partitioning.



Complete + Unrolling:

Then we tried unrolling with a factor value of 2 and complete array partitioning for `a` and `b`. The results were the same latency, throughput, DSP usage, and clock frequency as unrolling, but

slightly higher overhead. The reason for the higher overhead was due to the complete partitioning.



Optimized:

Lastly came the optimized solution. This one did a complete partition for all arrays, and unrolling for the row loop while partitioning the prod loop. The unrolling was with a factor of 2, while pipeline was II value of 4. This gave us the lowest clock frequency, and lower hardware usage than the base, yet with high latency. We took the tradeoff of the pipelining to get lower clock frequency.

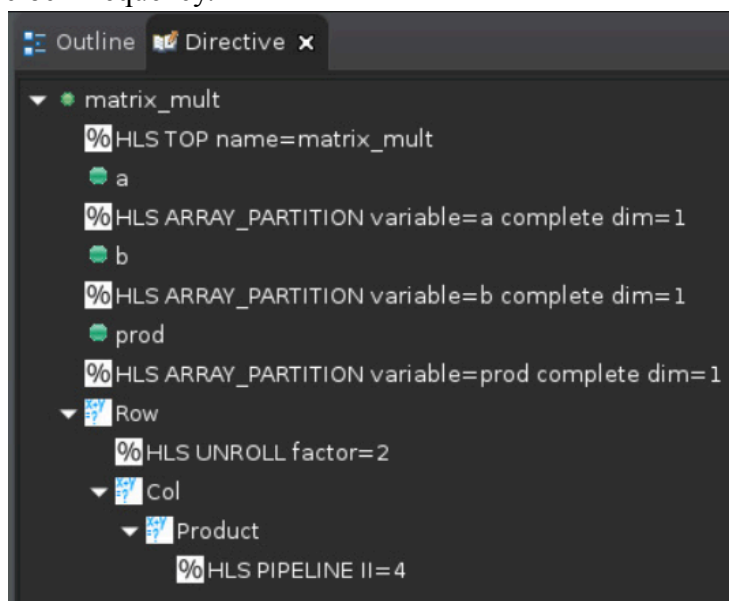
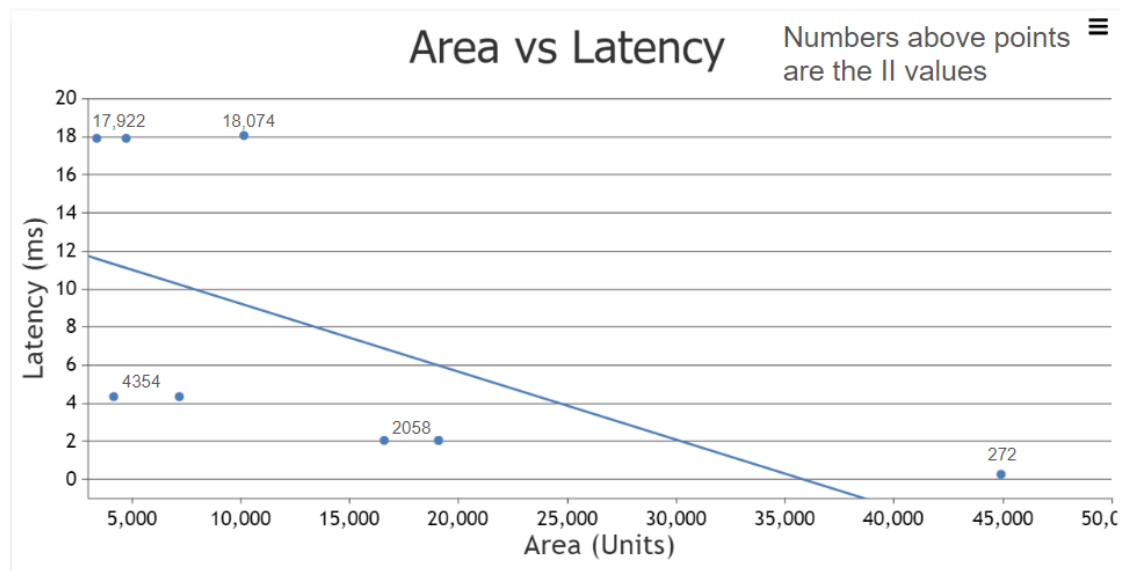


Table:

Architectural Alternative	LUTs	FFs	DSPs	BRAMs	Latency (Cycles)	Throughput (Cycles)	Clock (MHz)	Clock (ns)
Baseline	1264	376	8	0	2057	2058	157.74	6.343
AP: Cyclic	1502	443	8	0	2057	2058	160.39	6.232
AP: Block	1502	442	8	0	2057	2058	148.53	6.726
AP: Complete	3590	2917	8	0	271	272	157.74	6.343
Pipelining	287	78	1	0	17921	17922	168.28	5.947
Unrolling	338	170	1	0	4353	4354	159.24	6.27
Complete + Pipelining	387	212	1	0	17921	17922	168.38	5.947
Complete + Unrolling	600	363	1	0	4353	4354	159.24	6.27
Optimized	883	269	2	0	18073	18074	178.14	5.615

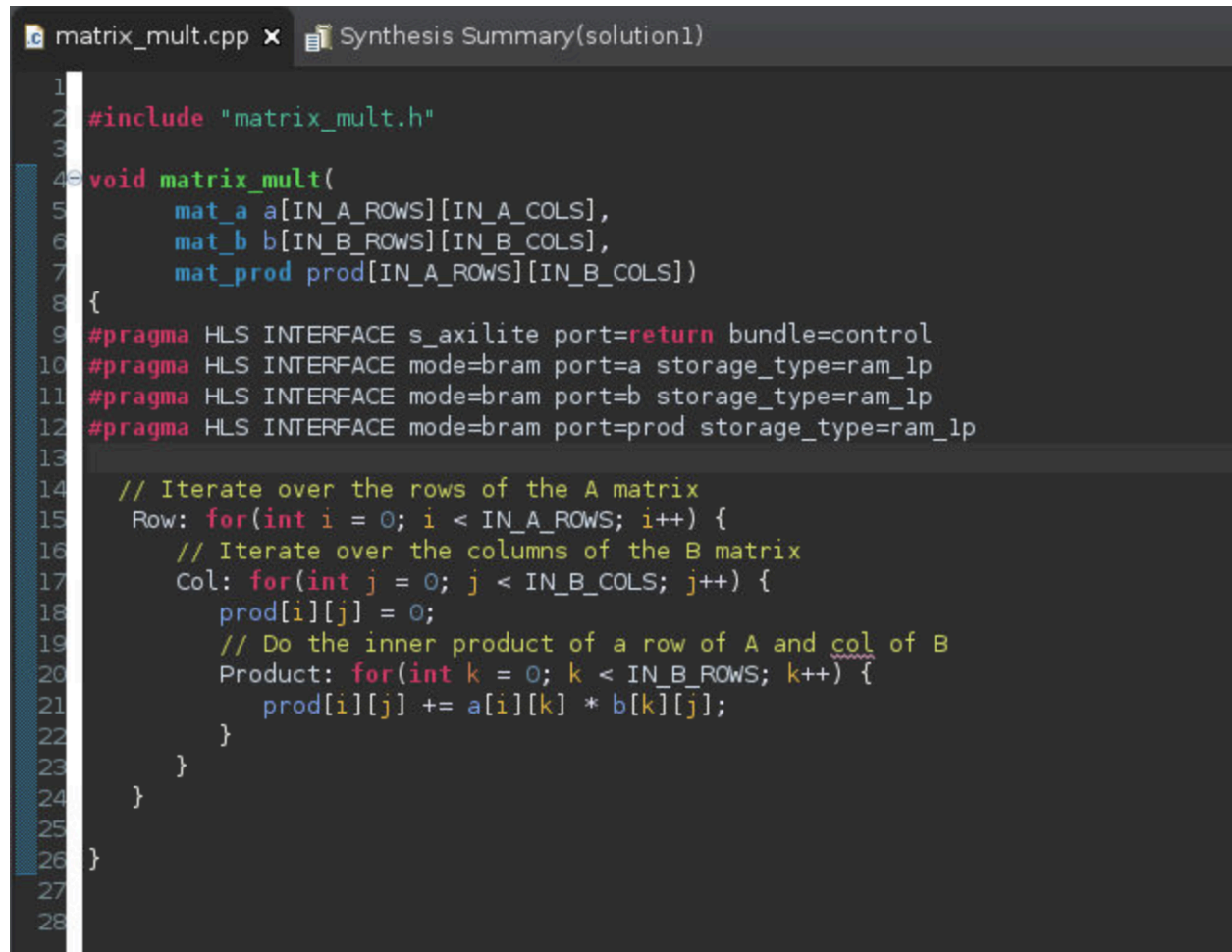
Chart:



Pareto Front Comment:

Based on the solutions we tested, there is not a concrete pareto front that forms in the graph, but there is a start of one that could curve from the 18,073 to the 271 ms latency points. There would need to be more solutions tested to get the pareto front to show more clearly.

Part 4:



```
1
2 #include "matrix_mult.h"
3
4 void matrix_mult(
5     mat_a a[IN_A_ROWS][IN_A_COLS],
6     mat_b b[IN_B_ROWS][IN_B_COLS],
7     mat_prod prod[IN_A_ROWS][IN_B_COLS])
8 {
9     #pragma HLS INTERFACE s_axilite port=return bundle=control
10    #pragma HLS INTERFACE mode=bram port=a storage_type=ram_lp
11    #pragma HLS INTERFACE mode=bram port=b storage_type=ram_lp
12    #pragma HLS INTERFACE mode=bram port=prod storage_type=ram_lp
13
14    // Iterate over the rows of the A matrix
15    Row: for(int i = 0; i < IN_A_ROWS; i++) {
16        // Iterate over the columns of the B matrix
17        Col: for(int j = 0; j < IN_B_COLS; j++) {
18            prod[i][j] = 0;
19            // Do the inner product of a row of A and col of B
20            Product: for(int k = 0; k < IN_B_ROWS; k++) {
21                prod[i][j] += a[i][k] * b[k][j];
22            }
23        }
24    }
25
26 }
27
28
```

Software Only

Starting Calculation 1
duration(ms): 59.31258201599121
Starting Calculation 2
duration(ms): 55.46283721923828
Starting Calculation 3
duration(ms): 54.71539497375488
Starting Calculation 4
duration(ms): 57.854652404785156
Starting Calculation 5
duration(ms): 56.093454360961914
Starting Calculation 6
duration(ms): 55.63497543334961
Starting Calculation 7
duration(ms): 54.93354797363281
Starting Calculation 8
duration(ms): 58.07018280029297
Starting Calculation 9
duration(ms): 57.07859992980957
Starting Calculation 10
duration(ms): 55.45473098754883

Calculation Times Report

Calculation 1: 59.31258201599121 ms
Calculation 2: 55.46283721923828 ms
Calculation 3: 54.71539497375488 ms
Calculation 4: 57.854652404785156 ms
Calculation 5: 56.093454360961914 ms
Calculation 6: 55.63497543334961 ms
Calculation 7: 54.93354797363281 ms
Calculation 8: 58.07018280029297 ms
Calculation 9: 57.07859992980957 ms
Calculation 10: 55.45473098754883 ms
Average Calculation Time: 56.46109580993652 ms

Software + Hardware Execution

```
Starting Calculation 1
duration(ms): 0.40030479431152344
Starting Calculation 2
duration(ms): 0.3192424774169922
Starting Calculation 3
duration(ms): 0.35953521728515625
Starting Calculation 4
duration(ms): 0.3437995910644531
Starting Calculation 5
duration(ms): 0.34308433532714844
Starting Calculation 6
duration(ms): 0.2875328063964844
Starting Calculation 7
duration(ms): 0.34117698669433594
Starting Calculation 8
duration(ms): 0.34046173095703125
Starting Calculation 9
duration(ms): 0.33974647521972656
Starting Calculation 10
duration(ms): 0.33593177795410156
```

Calculation Times Report

```
Calculation 1: 0.40030479431152344 ms
Calculation 2: 0.3192424774169922 ms
Calculation 3: 0.35953521728515625 ms
Calculation 4: 0.3437995910644531 ms
Calculation 5: 0.34308433532714844 ms
Calculation 6: 0.2875328063964844 ms
Calculation 7: 0.34117698669433594 ms
Calculation 8: 0.34046173095703125 ms
Calculation 9: 0.33974647521972656 ms
Calculation 10: 0.33593177795410156 ms
Average Calculation Time: 0.3410816192626953 ms
```

For the baseline design, there is an obvious performance increase when performing the calculation using the processor only which had an average calculation time of ~56.4 ms compared to performing the calculations with the FPGA programmable logic ~0.34 ms. This can be explained due to the process in which a microprocessor typically performs calculations. When running the computation on the processor, our code is compiled into assembly code. The processor takes the compiled code and performs each calculation on a per instruction basis. This does not account for the additional time being used for fetching, decoding etc. that is performed for each instruction. Processors also must deal with branches and the appropriate action to take when coming upon a branch. Compared to our programmable logic which operates on a per-clock cycle basis these all eventually add up. Since our accelerator does not need to deal with these constraints it is capable of taking in the data and immediately performing each calculation.

Although our accelerator proves to be much more efficient than running the calculations with the processor. There can be much more improvements that we can make to achieve an even higher efficiency. The above calculations use the baseline design for the SW+HW execution, which is not optimized at all for area-latency. **We can improve this design by reducing the latency while also increasing the area to achieve a higher throughput with inline pragmas in the HLS stage. We do this by taking our optimized solution from part 3 and running it with the python drivers that we have created. This will optimize the RTL code and those performance improvements will be visible in our Vivado reports as well when looking at the utilization (Shown Below).**

Baseline Vitis Utilization:

Target	Estimated	Uncertainty
10.00 ns	6.923 ns	2.70 ns

Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
matrix_mult	II Violation			-	4113	4.113E4	-	4114	-	no	0	39	3403	1903	0
Row_Col	II Violation	Resource Limitation		-	4111	4.111E4	32	16	256	yes	-	-	-	-	-

Baseline Vivado Utilization:

Resource	Utilization	Available	Utilization %
LUT	2351	53200	4.42
LUTRAM	67	17400	0.39
FF	3078	106400	2.89
BRAM	6	140	4.29
DSP	39	220	17.73
BUFG	1	32	3.13

Optimized Vitis Utilization:

Target	Estimated	Uncertainty
10.00 ns	6.923 ns	2.70 ns

Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
matrix_mult				-	1425	1.425E4	-	1426	-	no	0	48	5408	4220	0
Row				-	1424	1.424E4	178	-	8	no	-	-	-	-	-

Optimized Vivado Utilization:

Resource	Utilization	Available	Utilization %
LUT	3968	53200	7.46
LUTRAM	95	17400	0.55
FF	5632	106400	5.29
BRAM	18	140	12.86
DSP	48	220	21.82
BUFG	1	32	3.13

Software + Hardware Execution Optimized

Starting Calculation 1
duration(ms): 0.3275871276855469
Calculation 16 Passed!
Starting Calculation 2
duration(ms): 0.30612945556640625
Calculation 16 Passed!
Starting Calculation 3
duration(ms): 0.3159046173095703
Calculation 16 Passed!
Starting Calculation 4
duration(ms): 0.3330707550048828
Calculation 16 Passed!
Starting Calculation 5
duration(ms): 0.30994415283203125
Calculation 16 Passed!
Starting Calculation 6
duration(ms): 0.3159046173095703
Calculation 16 Passed!
Starting Calculation 7
duration(ms): 0.33545494079589844
Calculation 16 Passed!
Starting Calculation 8
duration(ms): 0.3108978271484375
Calculation 16 Passed!
Starting Calculation 9
duration(ms): 0.3006458282470703
Calculation 16 Passed!
Starting Calculation 10
duration(ms): 0.3325939178466797
Calculation 16 Passed!

Calculation Times Report

Calculation 1: 0.3275871276855469 ms
Calculation 2: 0.30612945556640625 ms
Calculation 3: 0.3159046173095703 ms
Calculation 4: 0.3330707550048828 ms
Calculation 5: 0.30994415283203125 ms
Calculation 6: 0.3159046173095703 ms
Calculation 7: 0.33545494079589844 ms
Calculation 8: 0.3108978271484375 ms
Calculation 9: 0.3006458282470703 ms
Calculation 10: 0.3325939178466797 ms
Average Calculation Time: 0.3188133239746094 ms