MP3 Report

By: Nomaan Dossaji (ndossa2)
and
Jay Patel (jdpatel4)

Part A:

Assumptions

The pragma options given are optional and we are allowed to use the options to our advantage to complete each part. Because of this assumption, we found that HLS DATAFLOW was not needed since we were able to sufficiently optimize in part B without it. We also decided not to use it because there would be a much greater power utilization and resource constraint.

What we did

1. For part 1 we were not required to use any pragmas.
2. For part 2, we used the pragma HLS PIPELINE with II=1. We put the pragma before the most inner for loop because it pipelines that loop. This loop contains the operation and contains the most latency.
3. For part 3 we used pragma HLS UNROLL in order to parallelize the operations that were occurring. We put this before the line that has the operation to indicate this is the line we wanted to unroll.
4. Part 4 required a FIFO. To accomplish this, we transposed B before calling the function and put the pragma HLS interface ap_fifo after we instantiated the variable that indicated the way we wanted the FIFO to access the memory. This line creates FIFOs for all 3 matrices.
5. Part 5 is similar to part 2 since we put the pragma HLS PIPELINE before the inner most loop to pipeline that loop. This loop contains the operation and the most latency. We also needed the ap_fifo pragma lines to maintain the fifos.
6. Part 6 is similar to part 3 since we used the pragma HLS UNROLL to parallelize the operations that were occurring. We put these lines before the operation to indicate that was to be unrolled and we also had the ap_fifo pragma to maintain the fifos.

Changes

The only change we needed to make to the original code was transitioning from part 3 to part 4. Part 4 requires a FIFO, so we needed to transpose B, before sending it to the function to complete the matrix multiplication. The multiplication operation also changed in the base algorithm since we transposed B. The only other changes between each part dealt with the pragmas.

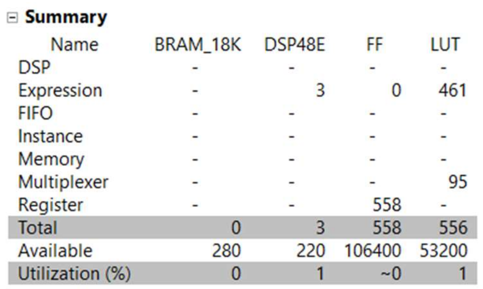
Limitations/Modifications

There were no limitations and we needed to modify the input to parts 4, 5, and 6 in order to use the FIFO. To use the FIFO we transposed B and within the algorithm we changed the multiplication operation to be compatible with the change we made.

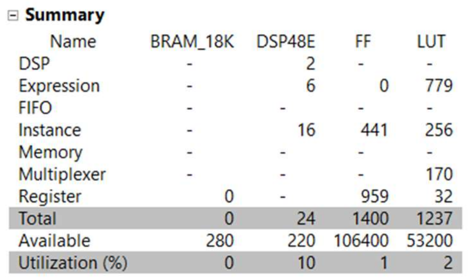
You Will Report Section

Resource Utilization:

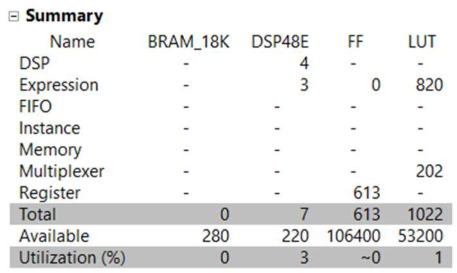
For this section we mainly focus on the FF (flip flops) and LUT expression and multiplexers.

1.

Summary

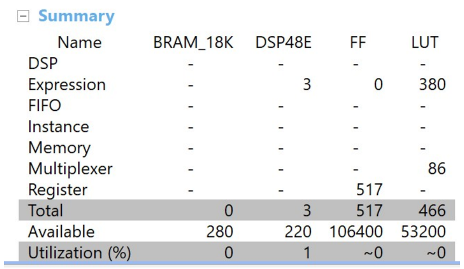
Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	3	0	461
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	95
Register	-	-	558	-
Total	0	3	558	556
Available	280	220	106400	53200
Utilization (%)	0	1	~0	1

2.

Summary

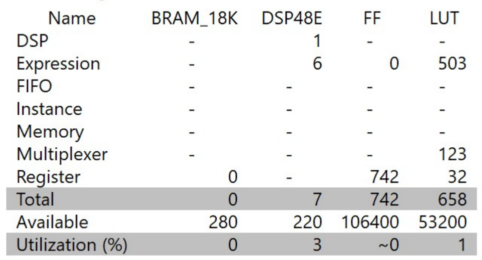
Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	2	-	-
Expression	-	6	0	779
FIFO	-	-	-	-
Instance	-	16	441	256
Memory	-	-	-	-
Multiplexer	-	-	-	170
Register	0	-	959	32
Total	0	24	1400	1237
Available	280	220	106400	53200
Utilization (%)	0	10	1	2

3.

Summary

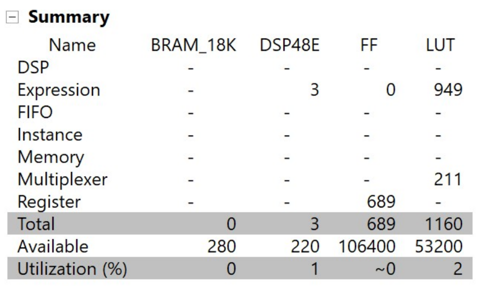
Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	4	-	-
Expression	-	3	0	820
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	202
Register	-	-	613	-
Total	0	7	613	1022
Available	280	220	106400	53200
Utilization (%)	0	3	~0	1

4.

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	3	0	380
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	86
Register	-	-	517	-
Total	0	3	517	466
Available	280	220	106400	53200
Utilization (%)	0	1	~0	~0

5.


Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	1	-	-
Expression	-	6	0	503
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	123
Register	0	-	742	32
Total	0	7	742	658
Available	280	220	106400	53200
Utilization (%)	0	3	~0	1

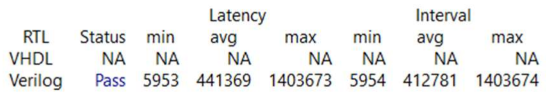
6. 

Summary

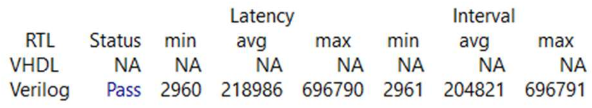
Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	3	0	949
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	211
Register	-	-	689	-
Total	0	3	689	1160
Available	280	220	106400	53200
Utilization (%)	0	1	~0	2

Latency:

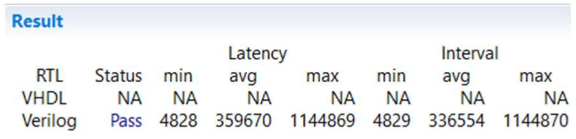
For this part we just look at the average latency below. Parts 2 and 3 were more optimized than part 1. Parts 5 and 6 were more optimized than part 4.

1. 

		Latency			Interval		
RTL	Status	min	avg	max	min	avg	max
VHDL	NA	NA	NA	NA	NA	NA	NA
Verilog	Pass	5953	441369	1403673	5954	412781	1403674

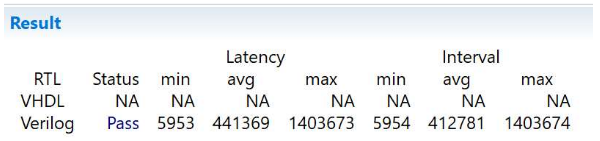
2. 

		Latency			Interval		
RTL	Status	min	avg	max	min	avg	max
VHDL	NA	NA	NA	NA	NA	NA	NA
Verilog	Pass	2960	218986	696790	2961	204821	696791

3. 

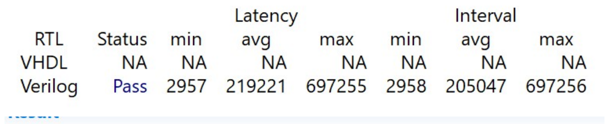
Result

		Latency			Interval		
RTL	Status	min	avg	max	min	avg	max
VHDL	NA	NA	NA	NA	NA	NA	NA
Verilog	Pass	4828	359670	1144869	4829	336554	1144870

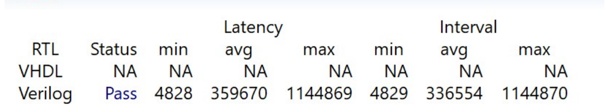
4. 

Result

		Latency			Interval		
RTL	Status	min	avg	max	min	avg	max
VHDL	NA	NA	NA	NA	NA	NA	NA
Verilog	Pass	5953	441369	1403673	5954	412781	1403674

5. 

		Latency			Interval		
RTL	Status	min	avg	max	min	avg	max
VHDL	NA	NA	NA	NA	NA	NA	NA
Verilog	Pass	2957	219221	697255	2958	205047	697256

6. 

		Latency			Interval		
RTL	Status	min	avg	max	min	avg	max
VHDL	NA	NA	NA	NA	NA	NA	NA
Verilog	Pass	4828	359670	1144869	4829	336554	1144870

Modify Base Algorithm:

We created our own base algorithm, we did not use the example project given by Vivado. We did not do anything in particular to leverage a type of interface or optimization directive. For the first three parts, we were able to keep the base algorithm. Parts 4, 5, and 6 needed to transpose B, which we did in the test bench and then also made changes to the base algorithm in the way we accessed the matrix. The changes we made allowed us to utilize the FIFO.

Possible:

We were able to complete all parts, with and without the FIFO. All of the parts are possible since we were able to make modifications to the base algorithm.

Scenarios:

1. For part 1 it is useful for when there are not enough resources in the system. Part 1 is also useful for power efficiency.
2. Part 2 is useful if there are enough flip flops and to reduce latency drastically. However, one would use part 2’s design if there are not enough resources for operations to be done in parallel.
3. Part 3 is useful when there are enough resources and there is no dependence between each iteration of the loop. For example, iteration 3 is not dependent on iteration 2. We do not care about power for this part, simply performance.
4. Part 4 is useful if the memory accesses are serial and there are not enough resources to parallelize and for power efficiency.
5. Part 5 is useful for serial access and not enough resources to execute in parallel, but there are enough flip flops to pipeline in order to reduce latency.
6. Part 6 is useful if the system has enough resources to parallelize the operations and we are using serial memory accesses. We do not care about power for this part, simply performance.

Latency Variation:

Unoptimized

1. This is the base latency without any optimizations. The latency is the most here since we are not using any optimizations, however we have the least resources being used so the area is the smallest. This was with or without the FIFO.
2. The FIFO here seemed to have the advantage. The latency stayed the same and there were less resources used so the area was smaller.

Loop Pipelining

1. Using the pipeline optimization allowed the latency to be less than without the optimization and less than the unrolling optimization. However, the area was more than both unrolling or not having the optimization. This was with or without the FIFO.
2. The FIFO created more latency, but less area.

Loop Unrolling

1. The unrolling optimization was in the middle between not having an optimization or using the pipeline optimization. The latency was more than the pipeline, but less than not having an optimization. The area was more than not having an optimization, but less than having the pipeline optimization. The same results showed with the FIFO as well.
2. Adding the FIFO did not change the latency, but did increase the area.

DATAFLOW Pragma:

The DATAFLOW pragma allows task-level pipelining. Instead of running functions sequentially, this pragma allows the functions to overlap.

Latency Calculations

1. The latency numbers are based on the datapath and the code that was written for the first part. This is the basic layout, so it gives a standard on what to improve from. To estimate the latency, I would need to calculate the number of memory access and add that to the number of multiplications and additions that occur.
2. This latency is calculated with pipelining in mind. That is why the latency is less. Since we are only adding pipelining, we can infer that the reduction in latency is due to the pipelining optimization. To estimate the latency, we would determine the pipeline factor with the multiplication and additions that is occurring.
3. This latency is calculated with the unrolling in mind. The reduction in latency is from having less operations since we are using more area to compute the instructions. To estimate the latency, you would need to determine the amount is unrolled and factor that into the latency for each operation that is occurring. This would parallelize the operations.
4. The latency number shows what happens when adding a FIFO. We see that there is no additional latency which means the FIFO did not burden or take away any latency. To estimate this, we can do the same calculations as done in part 1. I would need to calculate the number of memory access and add that to the number of multiplications and additions that occur.
5. This latency is showing using the FIFO and pipelining. There is a reduced latency, but not as much as the reduction without the FIFO. This shows the FIFO is adding latency when pipelining. To calculate the latency, you would need to determine the latency from multiplying and adding when pipelining. On top of this you would need to determine the latency of the FIFO.
6. This latency is showing using the FIFO and unrolling. Since the latency did not increase from part 3 with the unrolling optimization and no FIFO, we can assume that there is no latency from the FIFO. To calculate the latency, we can reuse the steps from part 3.

Difficulties/Bugs

One difficulty we encountered was when we first started to use the pragma for the FIFO. We did not realize that we needed to have linear accesses to the matrix. In order to satisfy this, we transposed the second matrix and accessed the matrices linearly instead.

What We Learned

With this part of the MP we learned how to properly use pragmas within the Vivado HLS. We also learned how to access matrices with a FIFO.

Workspace

We used the 2018.2 Vivado HLS on a Windows 10 machine.

Part B

Assumptions

We talked to the TA and the TA said that 10x improvement of performance is what we should aim for.

What We Did

For part B we first cached the data on the on chip RAM. Then we simply combined all the optimations that was done for PartA. Doing this gave enough improvement to show that the latency was reduced far more than 10x.

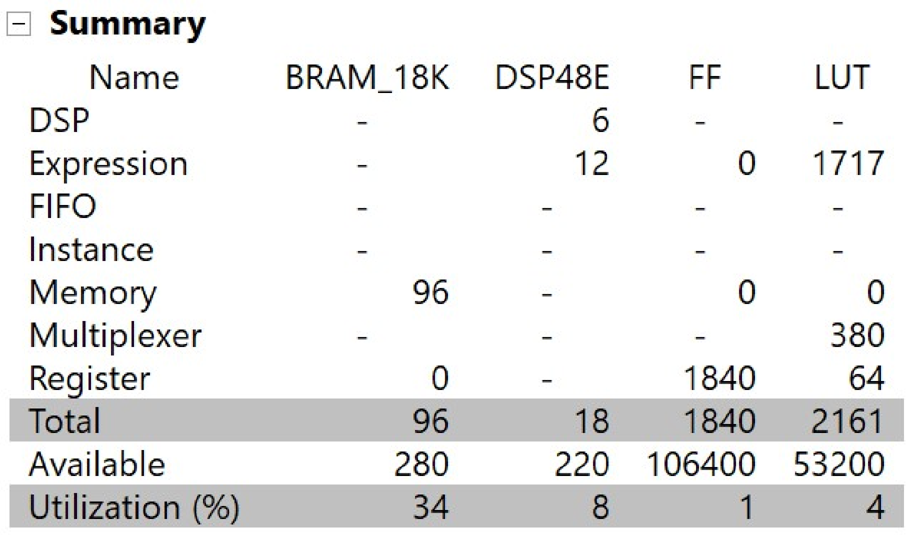
Changes to Code

The base algorithm was not changed only that we combined all the optimazations and cached the data on the chip so the reads and writes to main memory.

Limitations/Modifications

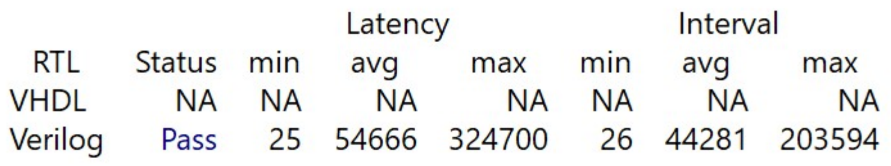
There were no limitations that we encountered for this part. We didn't modify the base algorithm we just added internal buffer to speed up calculations.

You Will Report

1. 

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	6	-	-
Expression	-	12	0	1717
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	96	-	0	0
Multiplexer	-	-	-	380
Register	0	-	1840	64
Total	96	18	1840	2161
Available	280	220	106400	53200
Utilization (%)	34	8	1	4

2. 

		Latency			Interval		
RTL	Status	min	avg	max	min	avg	max
VHDL	NA	NA	NA	NA	NA	NA	NA
Verilog	Pass	25	54666	324700	26	44281	203594

3. This situation would be useful for high throughput matrix multiplication such as Convolutional Neural Networks. Any kind of program that requires high performance and does suffer from very limited on-chip RAM.
4. Some constraints or limitations are on chip memory. The on-chip memory may not be sufficient, so to compensate you can use external memory. Another constraint can be not having enough LUTs or Flip Flops. To deal with this, the developer can make the algorithm less efficient or acquire more resources for the board.
5. The optimizations we used are loop unrolling and loop pipelining together. There is always more we can do to optimize, but using more optimization requires more resources. We could improve performance by utilizing the pragma HLS DATAFLOW. This would definitely improve the latency, but cause much more FLIP FLOPs to be required.

Latency Calculations

The latency shows that there is a lot of latency reduced from both pipelining and unrolling. The calculations can be done by combining unrolling and pipelining optimizations described in part a. These calculations will show the significant reduction in latency.

Difficulties/Bugs

Using Vivado HLS was difficult since there is no way to be certain that what we did was correct or true. Sometimes we got strange error when we were comparing C simulation to RTL simulation. However, since we had learned from the first part, we were more familiar with the software and this part when smoothly.

What We Learned

We were able to use optimization techniques and test the effectiveness of them. We learned how to use different optimizations to do matrix multiplication.

Workspace

We used the 2018.2 Vivado HLS on a Windows 10 machine.