Application Acceleration with High-Level Synthesis

Freud L. Lewis Piercius 110061422

LabA – Design Optimization

Introduction

In this lab, we want to explore design optimization with Vitis. In hardware design, we want to focus mostly on latency and area, as area affects greatly power consumption. For that, when designing, implementing a system, we want to optimize as much as possible, in the process there are indeed some tradeoffs to make. For this lab, we will be exploring designing a 3x3 matrix multiplication with Vitis HLS, which we can use to optimize our hardware module with either **#pragma** or other built-in directives. Our system contains the multiplication hardware module which will be implemented on the PL side on Alveo U50 acceleration card, and one testbench on the host. Our main objective is to analyze the hardware performance, FPGA utilization by comparing different methods.

System Implementation

In lab1, the original code is written as shown in Fig. 1, it consists of a triple nested loop. As we are doing matrix multiplication C = A * B where the number of rows of the product matrix C, is equal the number of rows of matrix C and number of columns of C equal number of number of columns of B, we need to through every cell of the product matrix C_{ij}, which will be the accumulation of A_{ik} * B_{kj}. In the code below, there is an initialization of the cell, res[i][j] = 0.

```
// Iterate over the rows of the A matrix
Row: for(int i = 0; i < MAT_A_ROWS; i++) {
    // Iterate over the columns of the B matrix
    Col: for(int j = 0; j < MAT_B_COLS; j++) {
        res[i][j] = 0;
        // Do the inner product of a row of A and col of B
        Product: for(int k = 0; k < MAT_B_ROWS; k++) {
            res[i][j] += a[i][k] * b[k][j];
        }
    }
}
```

Fig. 1.

Design Optimization with Vitis High-Level Synthesis

As Vitis v2022.1 automatically try to optimize the design, in order to have a better understanding about the three techniques (unroll, pipeline and array repartition) we can explore to optimize our design, I use pragma to deactivate automatic loop unrolling and pipelining as shown in Fig. 2. To turn a pragma directive

on or off for loop, we need to place it above the loop. The clock chosen for the system to operate is 75MHz.

```
  // Iterate over the rows of the A matrix
   Row: for(int i = 0; i < MAT_A_ROWS; i++) {
      // Iterate over the columns of the B matrix
#pragma HLS PIPELINE off
      Col: for(int j = 0; j < MAT_B_COLS; j++) {
         res[i][j] = 0;
         // Do the inner product of a row of A and col of B
#pragma HLS UNROLL off
         Product: for(int k = 0; k < MAT_B_ROWS; k++) {
            res[i][j] += a[i][k] * b[k][j];
         }
```

Fig. 2.

In new Vitis versions, optimization is automatically applied to inner loop if there is any. As depicted in Fig. 4, Pipelining is automatically applied to Col loop, and unrolling to product loop consequently by Vitis tool. Therefore, there will be some parts(solutions) that are different from the tutorial

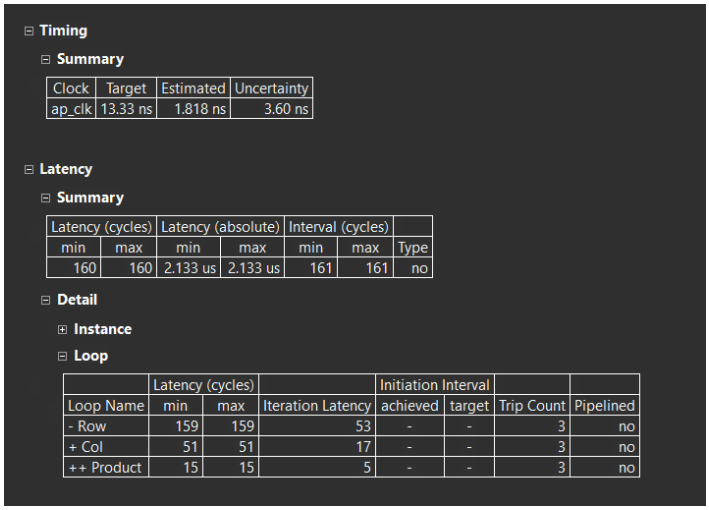
```
INFO: [HLS 200-10] Starting code transformations ...
INFO: [HLS 200-111] Finished Standard Transforms: CPU user time: 0 seconds. CPU system time: 0 seconds. Elapsed time: 0.01 seconds; current allocated memory: 462.957 MB.
INFO: [HLS 200-10] Checking synthesizability ...
INFO: [HLS 200-111] Finished Checking Synthesizability: CPU user time: 0.01 seconds. CPU system time: 0 seconds. Elapsed time: 0.01 seconds; current allocated memory: 463.051 MB.
INFO: [XFORM 203-510] Pipelining loop 'Col' (matrixmul.cpp:56) in function 'matrixmul' automatically.
INFO: [XFORM 203-502] Unrolling all sub-loops inside loop 'Col' (matrixmul.cpp:56) in function 'matrixmul' for pipelining.
INFO: [HLS 200-489] Unrolling loop 'Product' (matrixmul.cpp:60) in function 'matrixmul' completely with a factor of 3.
INFO: [HLS 200-111] Finished Loop, function and other optimizations: CPU user time: 0.03 seconds. CPU system time: 0.01 seconds. Elapsed time: 0.08 seconds; current allocated memory: 484.461 MB.
INFO: [XFORM 203-541] Flattening a loop nest 'Row' (matrixmul.cpp:54:17) in function 'matrixmul'.
INFO: [HLS 200-111] Finished Architecture Synthesis: CPU user time: 0.03 seconds. CPU system time: 0 seconds. Elapsed time: 0.02 seconds; current allocated memory: 484.461 MB.
INFO: [HLS 200-10] Starting hardware synthesis
```

Fig. 3.

Different techniques were utilized to optimized the design, such as loop Unrolling, Pipeline, AP_FIFO and Array reshaping. The next paragraphs contain the timing, performance and utilization analysis of these techniques separately.

Baseline

As stated in **system Implementation**, we use #pragma to turn the automatic unroll, pipeline off. In Fig. 4 depicted the baseline performance estimates. The latency of one iteration in Product loop is 5 clock cycles, since the trip count is 3, so the total latency for the Product loop MAC is 3 * 5 = 15. For the Col loop, it takes 1 clock cycle to enter and 1 to exit the loop, and because it includes the product loop, so for one iteration the latency is 1 + 15 + 1 = 17 clock cycles, for 3 iterations, we have 51 clock cycles. The latency for Row loop can be explained just like Col's, 1 + 51 + 1= 53; 53 * 3 = 159. In the summary table, it shows the latency in both cycles and microseconds. The total latency is 160 clock cycles, 160 * 1/75MHz = 2.133us



Timing

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	13.33 ns	1.818 ns	3.60 ns

Latency

Summary

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
160	160	2.133 us	2.133 us	161	161	no

Detail

Instance

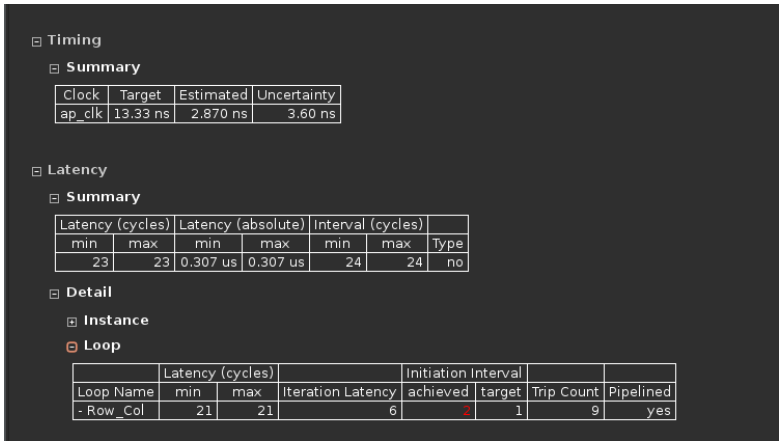
Loop

	Latency (cycles)			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Row	159	159	53	-	-	3	no
+ Col	51	51	17	-	-	3	no
++ Product	15	15	5	-	-	3	no

Fi. 4.

Pipeline Col loop + inner loop unrolling

When Vitis automatically applies pipeline to Col loop and unroll its inner loop (Product), we get these performance data shown in Fig. 5. As the inner loop is unrolled, there is automatic renaming although not shown in this report. As we can see, there is a violation in the figure below, there reason is because there is limited memory ports, so the load operation cannot be schedules as shown in Fig. 6. With the pipelining we can see that total latency 23, the system is 6.95 times faster than the baseline.



Timing

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	13.33 ns	2.870 ns	3.60 ns

Latency

Summary

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
23	23	0.307 us	0.307 us	24	24	no

Detail

Instance

Loop

	Latency (cycles)			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Row_Col	21	21	6	2	1	9	yes

Fig. 5.

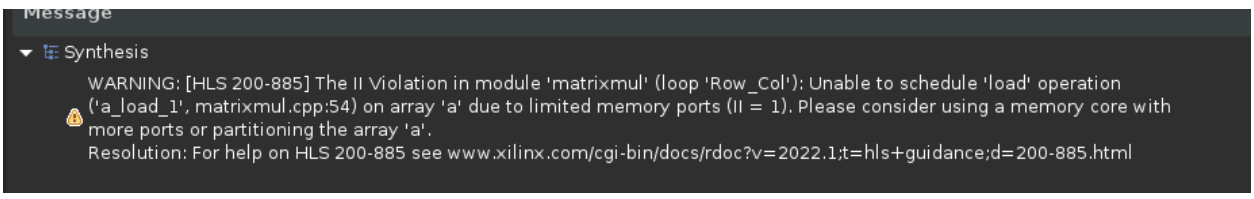
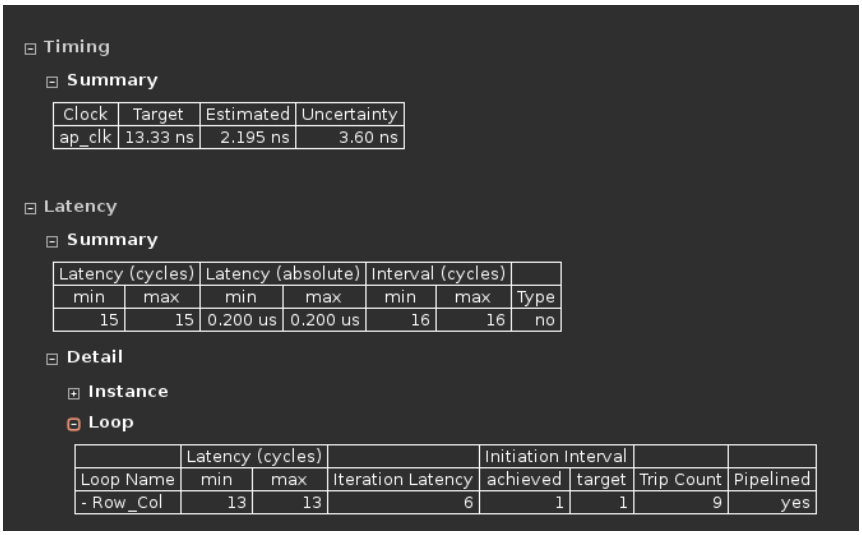


Fig. 6.

Array Reshaping

In Fig. 6 Vitis suggests array partitioning, so we do as it suggests by giving array “a” the dimension 2 and “b”, 1. The performance is as shown in Fig. 7. The total latency is 15, which is 1.5 faster than when only pipeline is applied to Col loop.



Timing

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	13.33 ns	2.195 ns	3.60 ns

Latency

Summary

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
15	15	0.200 us	0.200 us	16	16	no

Detail

Instance

Loop

	Latency (cycles)			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Row_Col	13	13	6	1	1	9	yes

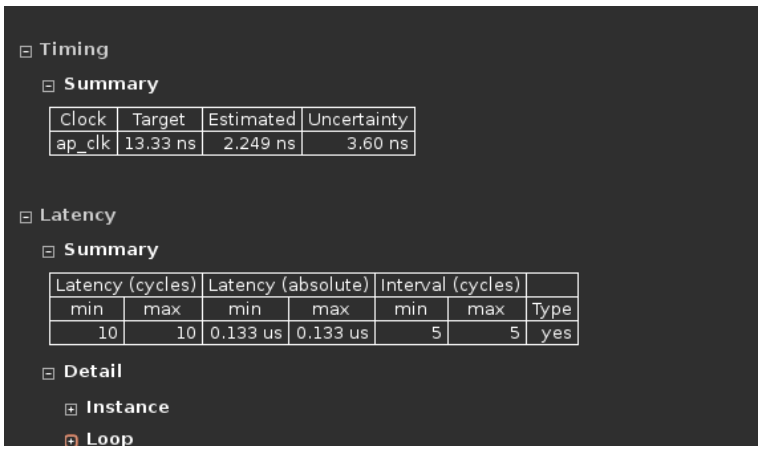
Fig. 7.

FIFO interface

FIFO interface cannot be used with the code in Fig. 1. Because the built-in directives are not adequate enough. To use FIFO, we will need to rewrite the code.

Pipeline of the function

If we pipeline the whole function, we see that the total latency decreases by 5 cycles. But to what cost?



Timing

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	13.33 ns	2.249 ns	3.60 ns

Latency

Summary

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
10	10	0.133 us	0.133 us	5	5	yes

Detail

Instance

Loop

Fig. 8.

Lab2

As we couldn't use FIFO interface, we needed to rewrite the code, this lab provides us a sample of rewritten code in order to use streaming interface. Fig. 9 shows the code. In order to use stream interface, we need to use the correct pragmas. Fig. 10 shows the addresses generated to read a, b and the array res. We must have the data saved internally to avoid data being re-read. Hence in Fig. 9's code, we can see temp variable (to initialize the cell) and 2 arrays a_row (to store data in row of a) and b_copy (to store b, cell by cell).

```
{
#pragma HLS ARRAY_RESHAPE variable=b complete dim=1
#pragma HLS ARRAY_RESHAPE variable=a complete dim=2
#pragma HLS INTERFACE ap_fifo port=a
#pragma HLS INTERFACE ap_fifo port=b
#pragma HLS INTERFACE ap_fifo port=res
  mat_a_t a_row[MAT_A_ROWS];
  mat_b_t b_copy[MAT_B_ROWS][MAT_B_COLS];
  int tmp = 0;

  // Iterate over the rowa of the A matrix
  Row: for(int i = 0; i < MAT_A_ROWS; i++) {
    // Iterate over the columns of the B matrix
    Col: for(int j = 0; j < MAT_B_COLS; j++) {
#pragma HLS PIPELINE
      // Do the inner product of a row of A and col of B
      tmp=0;
      // Cache each row (so it's only read once per function)
      if (j == 0)
        Cache_Row: for(int k = 0; k < MAT_A_ROWS; k++)
          a_row[k] = a[i][k];

       // Cache all cols (so they are only read once per function)
     if (i == 0)
            Cache_Col: for(int k = 0; k < MAT_B_ROWS; k++)
              b_copy[k][j] = b[k][j];

      Product: for(int k = 0; k < MAT_B_ROWS; k++) {
        tmp += a_row[k] * b_copy[k][j];
      }
      res[i][j] = tmp;
    }
  }
```

Fig. 9.

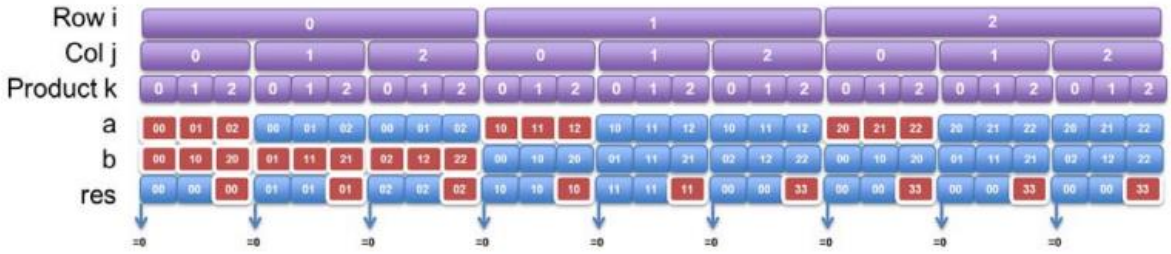


Figure 7-20: **Matrix Multiplier Address Accesses**

Fig. 10.

The total latency is 17 which is slightly slower than array reshaping alone.

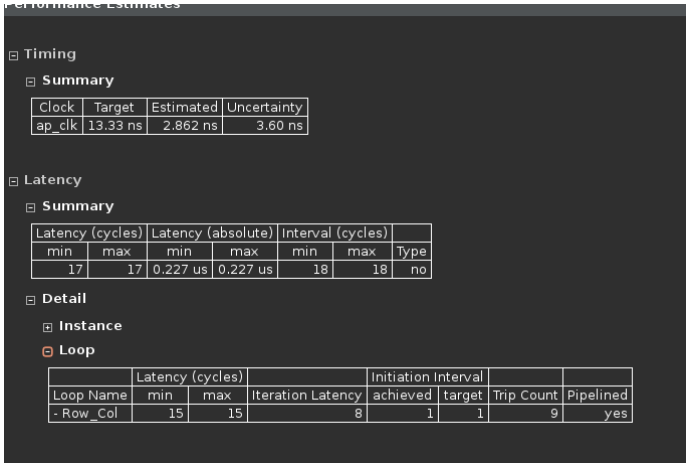


Fig. 11.

Utilization Estimates

We see that the baseline although very slow with 160 clock cycles uses same amount of FPGA resources as basic automatic pipeline and inner loop unrolling by Vitis HLS. As stated in Introduction, when designing hardware there are tradeoffs to consider, when pipelining the whole function, the design takes fewer cycles to complete, but the number of resources used is enormous as compared to array reshaping. The reason is all the loops in the design were unrolled in other to pipeline the function.

	Baseline	Col Pipeline + prod. Unroll	Array Reshaping	Function Pipeline	Lab2
BRAM_18K	0	0	0	0	0
DSP	2	2	2	18	2
FF	109	109	204	635	234
LUT	356	356	269	553	317
URAM	0	0	0	0	0

Table 1.

Conclusion

I think the function pipeline result in a better performance when considering the throughput as compared to Array reshaping and stream access. However, it will use lots of FPGAs components.

What I have learned

I've learned that each technique as their advantages and disadvantages, I also learned that just because one optimization technique works doesn't mean that it can be added to another. As in the case of FIFO streaming.

Github link: https://github.com/freud96/Design_Optimization.git