HLS Self-paced Learning Project
[pp4fpga] Matrix Multiplication

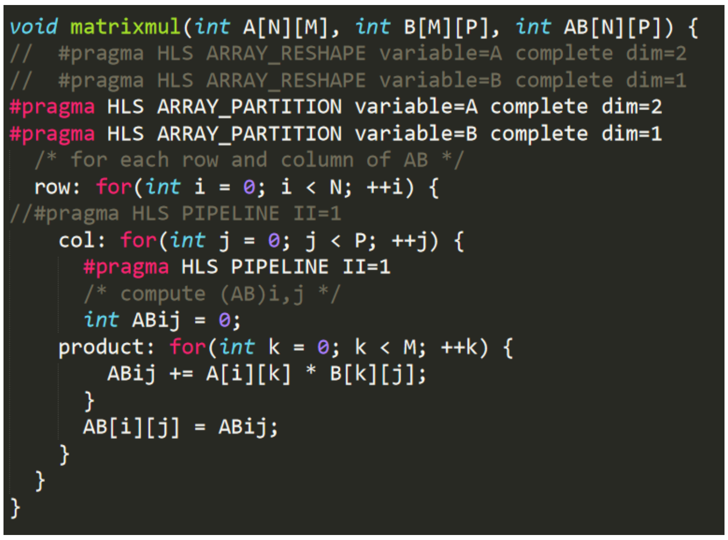
r08943154 謝承翰

1. Matrix Multiplication

   Matrix multiplication is a widely used calculation which combines two matrices into another. Given two matrices A (size: $\mathrm{n} \times \mathrm{m}$) and B (size: $\mathrm{m} \times \mathrm{p}$), the multiplication AB (size: $\mathrm{n} \times \mathrm{p}$) is defined as

$$AB_{ij} = \sum_{k=1}^{m} A_{ik} B_{kj}$$

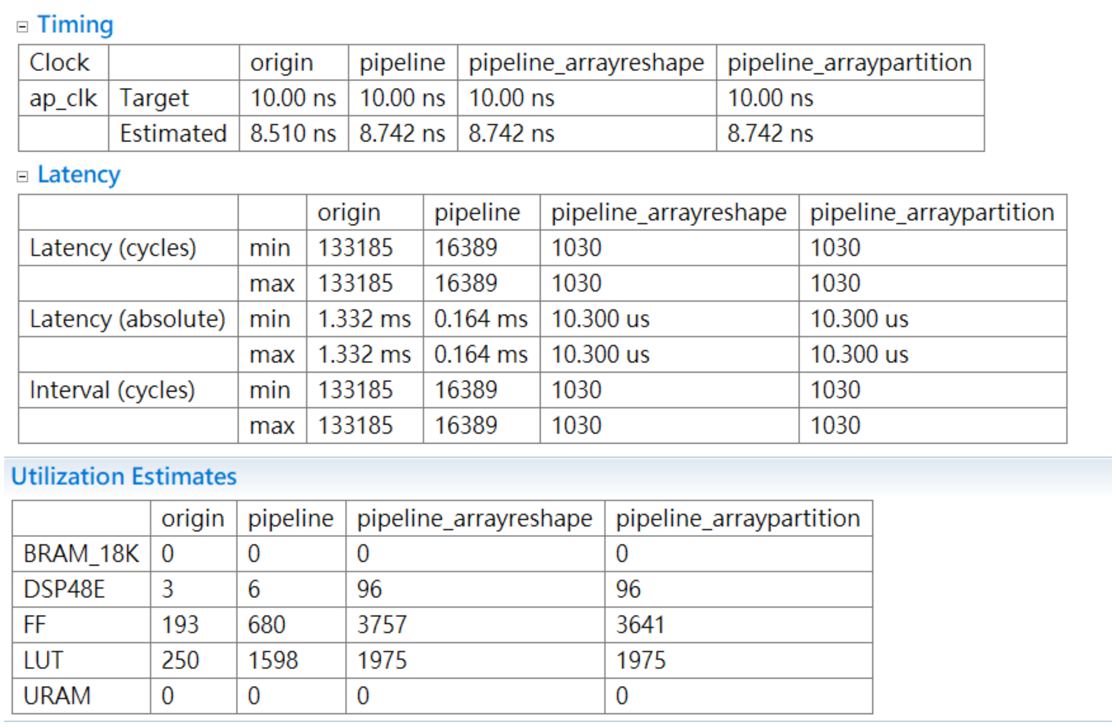
   The overall software code is a 3-level for loop – row enumeration, column enumeration and the summation of the product terms.



```
void matrixmul(int A[N][M], int B[M][P], int AB[N][P]) {
//  #pragma HLS ARRAY_RESHAPE variable=A complete dim=2
//  #pragma HLS ARRAY_RESHAPE variable=B complete dim=1
#pragma HLS ARRAY_PARTITION variable=A complete dim=2
#pragma HLS ARRAY_PARTITION variable=B complete dim=1
  /* for each row and column of AB */
  row: for(int i = 0; i < N; ++i) {
//#pragma HLS PIPELINE II=1
    col: for(int j = 0; j < P; ++j) {
      #pragma HLS PIPELINE II=1
      /* compute (AB)i,j */
      int ABij = 0;
    product: for(int k = 0; k < M; ++k) {
        ABij += A[i][k] * B[k][j];
      }
      AB[i][j] = ABij;
    }
  }
}
```

2. HLS on normal matrix multiplication

   For the HLS experiments, directly use the software code first. Then, a pipeline pragma is added to the second loop. The pipeline can only achieve II=16 because of the memory port. Therefore, we can add the array_reshape or array_partition pragma to increase the bandwidth of the memories. By this pragma, the pipeline can achieve II=1. The difference between arrar_reshape and array_partition is that array_partition just separate the blocks while array_reshape combines the partitioned array again to have a larger data on the memory port. Array_partition can be more flexible while array_reshape sometimes has a good mapping to FPGA resources.



Timing

Clock		origin	pipeline	pipeline_arrayreshape	pipeline_arraypartition
ap_clk	Target	10.00 ns	10.00 ns	10.00 ns	10.00 ns
	Estimated	8.510 ns	8.742 ns	8.742 ns	8.742 ns

Latency

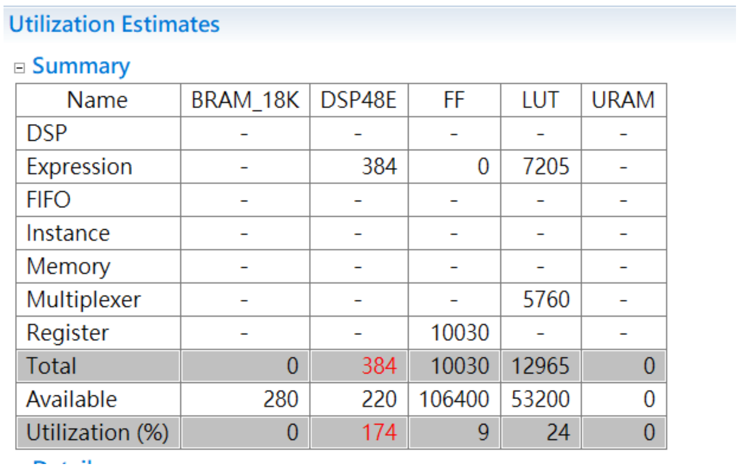
		origin	pipeline	pipeline_arrayreshape	pipeline_arraypartition
Latency (cycles)	min	133185	16389	1030	1030
	max	133185	16389	1030	1030
Latency (absolute)	min	1.332 ms	0.164 ms	10.300 us	10.300 us
	max	1.332 ms	0.164 ms	10.300 us	10.300 us
Interval (cycles)	min	133185	16389	1030	1030
	max	133185	16389	1030	1030

Utilization Estimates

	origin	pipeline	pipeline_arrayreshape	pipeline_arraypartition
BRAM_18K	0	0	0	0
DSP48E	3	6	96	96
FF	193	680	3757	3641
LUT	250	1598	1975	1975
URAM	0	0	0	0

The original design needs 130k cycles for the calculation. Pipeline can improve the latency by about 10x with more resouces about 5x. Then, the array_reshape and array_partition technique can further improve the latency by 16x again with the cost of many more resource usage. In this case, array_partition needs less resource than array_partition.

Moreover, I'm curious about if we can add the pipeline pragma in the first loop. The result shows that this design cannot fit in the board because of the resource limit.



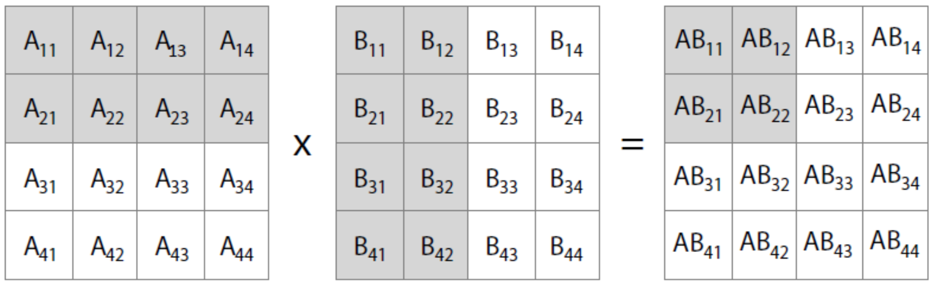
Utilization Estimates

Summary

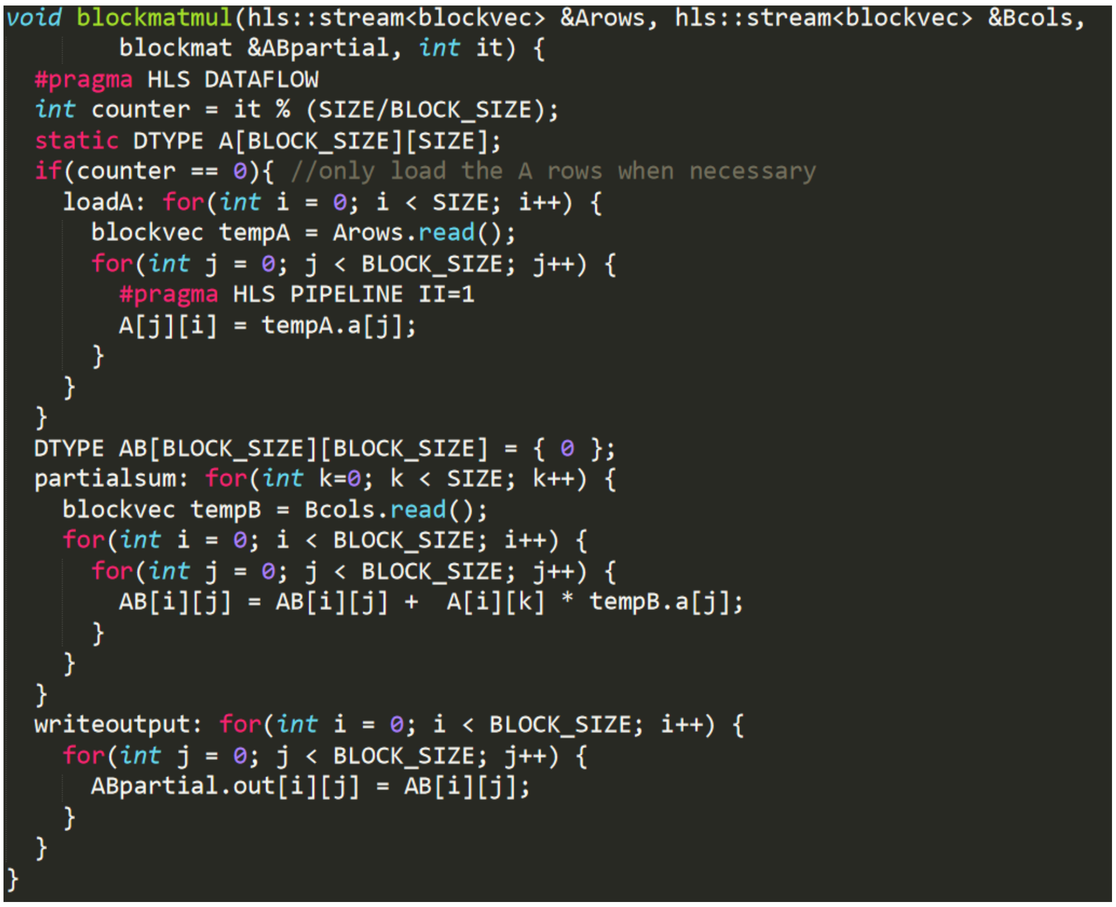
Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	384	0	7205	-
FIFO	-	-	-	-	-
Instance	-	-	-	-	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	5760	-
Register	-	-	10030	-	-
Total	0	384	10030	12965	0
Available	280	220	106400	53200	0
Utilization (%)	0	174	9	24	0

3. Block matrix multiplication

The original matrix multiplication assumes that the whole matrices are stored in the memories. For large matrices, this may not feasible and not efficient for the acceleration. We can view a matrix multiplication into several small blocks of small matrix multiplications. Then we can get data with a streaming manner and calculate some parts of the answer.


A_{11} A_{12} A_{13} A_{14}
A_{21} A_{22} A_{23} A_{24}
A_{31} A_{32} A_{33} A_{34}
A_{41} A_{42} A_{43} A_{44}
X
B_{11} B_{12} B_{13} B_{14}
B_{21} B_{22} B_{23} B_{24}
B_{31} B_{32} B_{33} B_{34}
B_{41} B_{42} B_{43} B_{44}
=
AB_{11} AB_{12} AB_{13} AB_{14}
AB_{21} AB_{22} AB_{23} AB_{24}
AB_{31} AB_{32} AB_{33} AB_{34}
AB_{41} AB_{42} AB_{43} AB_{44}

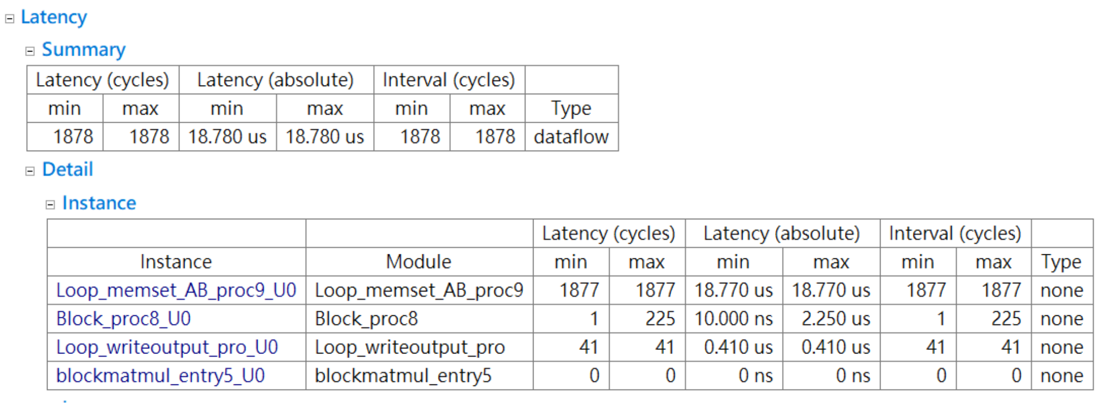
A block matrix multiplication can be illustrated as the figure above. Given a part of the values in A and B, the calculation of some values in AB can be done. The code can be rewrited as



```
void blockmatmul(hls::stream<blockvec> &Arows, hls::stream<blockvec> &Bcols,
        blockmat &ABpartial, int it) {
  #pragma HLS DATAFLOW
  int counter = it % (SIZE/BLOCK_SIZE);
  static DTYPE A[BLOCK_SIZE][SIZE];
  if(counter == 0){ //only load the A rows when necessary
    loadA: for(int i = 0; i < SIZE; i++) {
      blockvec tempA = Arows.read();
      for(int j = 0; j < BLOCK_SIZE; j++) {
        #pragma HLS PIPELINE II=1
        A[j][i] = tempA.a[j];
      }
    }
  }
  DTYPE AB[BLOCK_SIZE][BLOCK_SIZE] = { 0 };
  partialsum: for(int k=0; k < SIZE; k++) {
    blockvec tempB = Bcols.read();
    for(int i = 0; i < BLOCK_SIZE; i++) {
      for(int j = 0; j < BLOCK_SIZE; j++) {
        AB[i][j] = AB[i][j] +  A[i][k] * tempB.a[j];
      }
    }
  }
  writeoutput: for(int i = 0; i < BLOCK_SIZE; i++) {
    for(int j = 0; j < BLOCK_SIZE; j++) {
      ABpartial.out[i][j] = AB[i][j];
    }
  }
}
```

The SIZE is the height and width of a matrix, and the BLOCK_SIZE is the height and width of a small block. First, load A array if the counter meets the index of the row. Then, multiply the small matrix A and B into an array AB. Finally, write out the results. The testbench is also needed to fit this kind of operation. The testbench will be attached in the files.

4. HLS on block matrix multiplication with stream input
From the report of blocksize=4, it shows that dataflow is useful to pipeline the block_proc8, which loads the data into array. Others module are pipelined with this pragma. Therefore, the latency in the summary is the AB multiplication part.



⊟ Latency

⊟ Summary

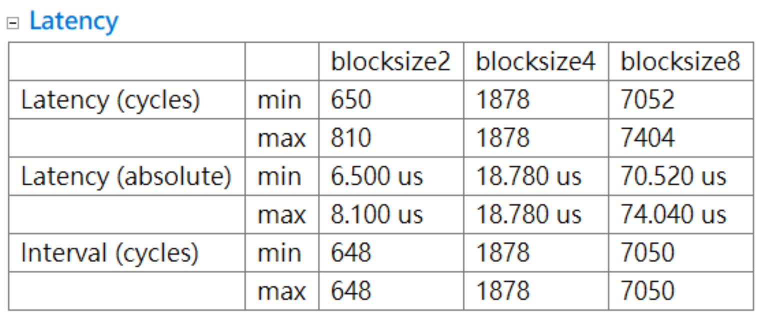
Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
1878	1878	18.780 us	18.780 us	1878	1878	dataflow

⊟ Detail

⊟ Instance

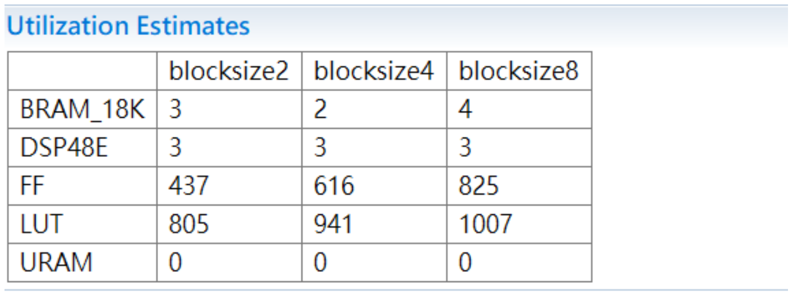
		Latency (cycles)		Latency (absolute)		Interval (cycles)		
Instance	Module	min	max	min	max	min	max	Type
Loop_memset_AB_proc9_U0	Loop_memset_AB_proc9	1877	1877	18.770 us	18.770 us	1877	1877	none
Block_proc8_U0	Block_proc8	1	225	10.000 ns	2.250 us	1	225	none
Loop_writeoutput_pro_U0	Loop_writeoutput_pro	41	41	0.410 us	0.410 us	41	41	none
blockmatmul_entry5_U0	blockmatmul_entry5	0	0	0 ns	0 ns	0	0	none

This design can be explored by the size of the small matrices. I change the blocksize as 2, 4 and 8 in a 32x32 matrix multiplication.



⊟ Latency

		blocksize2	blocksize4	blocksize8
Latency (cycles)	min	650	1878	7052
	max	810	1878	7404
Latency (absolute)	min	6.500 us	18.780 us	70.520 us
	max	8.100 us	18.780 us	74.040 us
Interval (cycles)	min	648	1878	7050
	max	648	1878	7050



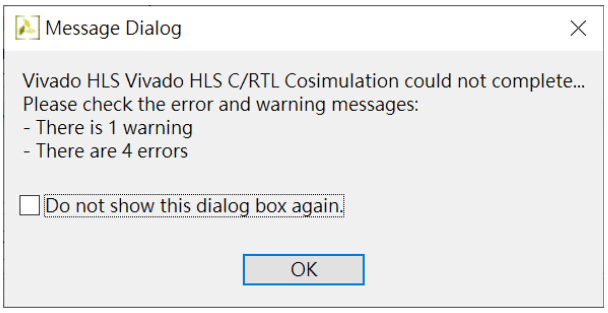
Utilization Estimates

	blocksize2	blocksize4	blocksize8
BRAM_18K	3	2	4
DSP48E	3	3	3
FF	437	616	825
LUT	805	941	1007
URAM	0	0	0

From the latency, we can estimate the total runtime. For blocksize=2, there are total (32/2)*(32/2) iterations for the calculation. The latency of the whole calculation is between 166k cycles to 207k cycles. For blocksize=4, the latency is 120k cycles. And the latency is 112k cycles for blocksize=8. In conclusion, the increasing of blocksize decrease the latency but might be converged because it is more similar to the original matrix multiplication.

5. Cosim Problem

In the co-simulation, the simulation will fail and say that a deadlock is detected when simulating the RTL code.

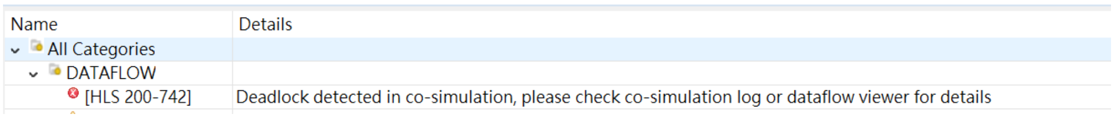


Message Dialog

Vivado HLS Vivado HLS C/RTL Cosimulation could not complete...
Please check the error and warning messages:
- There is 1 warning
- There are 4 errors

Do not show this dialog box again.

OK

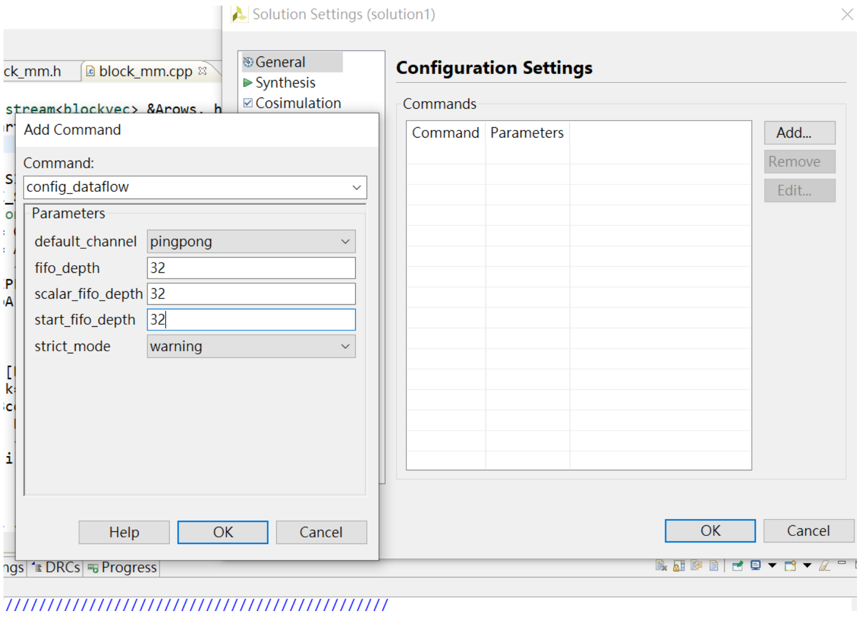


Name	Details
All Categories	
DATAFLOW	
[HLS 200-742]	Deadlock detected in co-simulation, please check co-simulation log or dataflow viewer for details

From the error messages in the console, there are also some people in the community suffer from this. Finally, I found that this is caused by lack of the depth of the ping-pong buffer.

```
//////////////////////////////////////////////////////////////////////////////
// ERROR!!! DEADLOCK DETECTED at 68840000 ns! SIMULATION WILL BE STOPPED! //
//////////////////////////////////////////////////////////////////////////////
/////////////////////////
// Dependence cycle 1:
// (1): Process: blockmatmul.Block_proc8_U0
//      Deadlocked by sync logic between input processes
//      Please increase channel depth
// (2): Process: blockmatmul.Loop_memset_AB_proc9_U0
//      Channel: blockmatmul.A_U, EMPTY
////////////////////////////////////////////////////////////////////////
// Totally 1 cycles detected!
////////////////////////////////////////////////////////////////////////
```

Through the setting, we can specify the depth of the buffer such that the co-sim can pass.


Solution Settings (solution1)

Configuration Settings

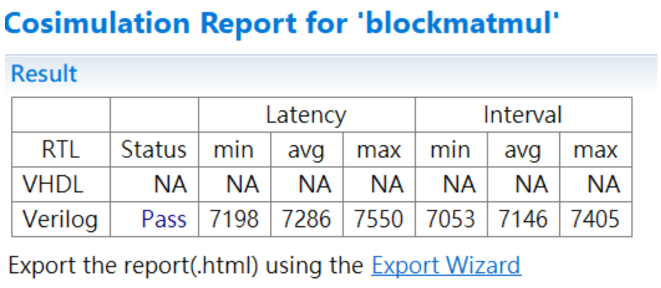
Add Command

Command: config_dataflow

Parameters

default_channel	pingpong
fifo_depth	32
scalar_fifo_depth	32
start_fifo_depth	32
strict_mode	warning

After the setting, the cosim passes.



Cosimulation Report for 'blockmatmul'

Result

		Latency			Interval		
RTL	Status	min	avg	max	min	avg	max
VHDL	NA	NA	NA	NA	NA	NA	NA
Verilog	Pass	7198	7286	7550	7053	7146	7405

Export the report(.html) using the Export Wizard

6. Conclusion
   - (1) Properly use pipeline and array partition technique can significantly improve the latency by 2 orders.
   - (2) Block matrix multiplication can separate the memory of a big matrix and benefit from streaming input with dataflow directive.
   - (3) With a dataflow pragma, we need to modify the setting of ping-pong buffer's depth in the co-simulation to pass.
   - (4) Codes and report are available on my github https://github.com/qmo1222/MSOC_HLS
7. Reference
   - (1) Textbook and codes in https://github.com/KastnerRG/pp4fpgas