Acceleration of Matrix Multiplication using Systolic Arrays

ECE 8893 Parallel Programming for FPGAs - Final report

Ram Jaswanth
Department of ECE
Georgia Institute of Technology
nchunduru3@gatech.edu

Krithika Kandasamy
Department of ECE
Georgia Institute of Technology
kkandasamy7@gatech.edu

Disha Gulur
Department of ECE
Georgia Institute of Technology
dgulur3@gatech.edu

I. Problem Description

Matrix Multiplication is one of the most important operations in many numerical algorithms and machine learning applications. The sequential implementation of matrix multiplication is a very time consuming task especially for large matrices. Given that many applications where matrix multiplication is used are time sensitive, the latency of this operation becomes an important parameter to tune and optimize. Systolic Array implementation of matrix multiplication is one such approach to gain computational speed-up.

The main idea behind systolic arrays is the decomposition of mathematical operation into a sequence of simple calculations such as multiply and accumulate operations. This specialized hardware performs a large number of these simple calculations in parallel to achieve high performance. Systolic arrays are essentially a network of Processing Elements (PEs) that rhythmically compute and pass data through the system.

It is also essential to keep in mind that most ML applications use a large amount of data stored in large matrices. This large data cannot be processed all at once on an FPGA due to the limited resources available. Our project aims at accelerating matrix multiplication operation of large input matrices using a 16x16 systolic array. This would involve tiling the matrices efficiently, computing and storing intermediate data and then eventually storing the final results in the memory. We aim to observe and analyze the number of cycles it takes to setup the systolic array, utilize all the processing elements and finally, drain the values.

II. Literature Review

We referred papers [1] and [2] to understand the systolic array architecture. [1] presents Systolic CNN, an OpenCL-defined scalable, run-time flexible accelerator architecture optimized for accelerating inference of various convolutional neural networks (CNNs). The paper adopts a highly pipelined and paralleled 1-D systolic processing element (PE) array architecture for convolutions. The proposed architecture is parametrized for scalability. [2] proposes an analytical model for identifying high performance and resource utilization with automated design space exploration framework for systolic array architectures. We also looked into the matrix multiplication implementation proposed in [3]. It proposes a model to optimize matrix multiplication for FPGA platforms, targeting maximum performance with minimum off-chip data movement. In this implementation, data is streamed across the processing elements instead of accessing it from off-chip memory.

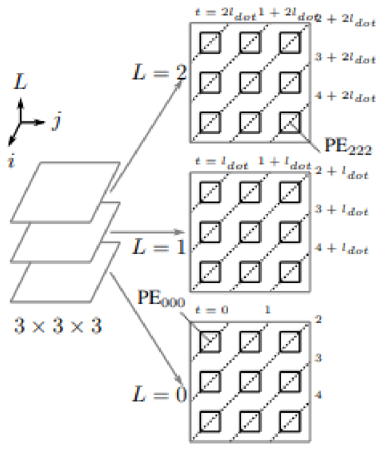


Fig. 1: Three-dimension systolic array architecture made of 9 PEs distributed over three 3 layers. The diagonal dashed lines represent the activation times of the intersected PEs

[4] is about the HLS implementation of a three-dimensional systolic array architecture for matrix multiplication that targets specific characteristics of Intel Stratix 10 FPGAs in order to produce designs that achieve a high floating point throughput using most of the DSPs at high frequencies. Figure 1 shows the three-dimensional systolic array architecture investigated in this paper.

The systolic array architecture investigated in [4] differs from our 2-D design. First, these PEs are made of a dot product unit performing more floating-point operations than a single multiply-accumulation. Second, the grid structure is three-dimensional. The time dimension of the classical systolic array is partially projected into the third dimension. In this regard, the investigated architecture can be considered as a stack of bi-dimensional layers, as shown in Figure 1. The value computed by the dot product unit is no more stationary within a PE but it is sent through the third dimension.

[5] incorporates 2 linear systolic arrays to perform matrix multiplication.

III. Midterm progress

The progress of our project till April 18, 2022 is listed below:

- Completed literature review to understand how systolic arrays are implemented for various applications.
- Implemented a 3x3 systolic array using C structures in Vitis HLS and verified the functionality for smaller matrices (without tiling). The data output was as expected. The matrix multiplication took about 65 cycles and the latency was 1.410 us. Ideally, a 3x3 multiplication should take only 7 cycles if the data movement and computation happens in parallel. However, that was not the case with this implementation. Also, since we had used C structures for the PEs, it was implemented as 2 port memory in hardware. As a result, we faced a lot of II violations while trying to synthesize the code.
- We started exploring different ways to incorporate data streaming with our existing code, in order to parallelize the data movement and computation functions.

IV. Systolic Array Design

We implemented a 16x16 systolic array consisting of 256 processing elements in total. For the design, let us consider a 4x4 systolic array as shown in Fig 2. The two matrices A and B are shifted into the boundary PEs in row 1 and column 1, respectively. The leading and trailing 0s in rows and columns are employed so that right elements A and B arrive at the PE for the multiply and accumulate operation to be performed.

Whenever a PE receives two inputs B and A from the left and the top, respectively, it performs the following set of operations, in this order:

- Computes AxB (Partial product)
- Accumulates the partial product with the previous value.
- Passes A to the processing element in the bottom.
- Passes B to the processing element in the right.

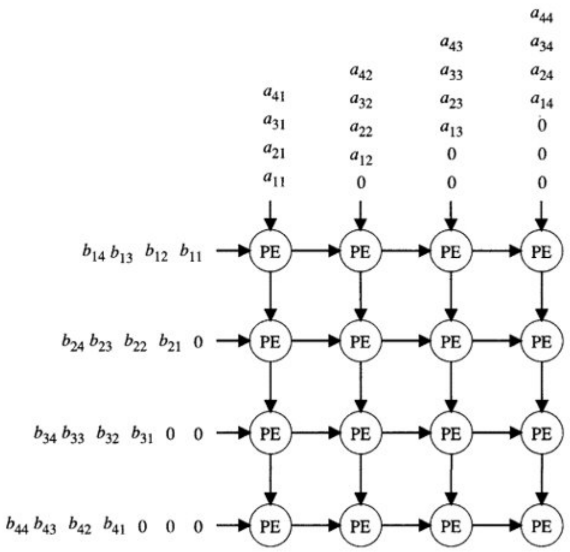
This algorithm takes time O(n), for n x n matrices.

V. Implementation

A. Systolic Array

With Fig 2. as the reference for our design, we implemented a 16x16 systolic array in HLS. The three different aspects of the design are listed below:

- Implementation of the processing elements



a_{41} a_{31} a_{21} a_{11}
a_{42} a_{32} a_{22} a_{12} 0
a_{43} a_{33} a_{23} a_{13} 0 0
a_{44} a_{34} a_{24} a_{14} 0 0 0
b_{14} b_{13} b_{12} b_{11}
b_{24} b_{23} b_{22} b_{21} 0
b_{34} b_{33} b_{32} b_{31} 0 0
b_{44} b_{43} b_{42} b_{41} 0 0 0
PE

Fig. 2: 4x4 systolic array

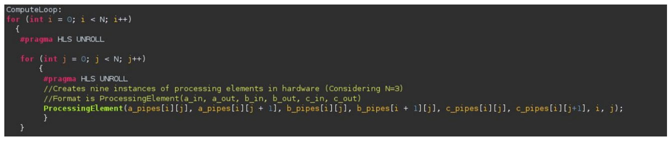
- Communication between the processing elements
- Parallelising the compute, read/write operations inside the processing element

As we know, C functions translate to modules in RTL. So, each PE was designed to be a function in C. Unrolling in HLS is equivalent to scaling. It has more meaning in FPGA than instruction based architectures since each unrolled instruction translates to a separate hardware. Since there are 256 PEs, we unrolled the PE function 16 times in depth and width.

All the PEs have to communicate with their neighbouring PEs for passing the input and output values. Communication between the PEs can be enabled through two interfaces - Synchronous or Asynchronous interface. In synchronous interfaces, all the PEs operate as a single unit and read/write/compute operations happen at the same clock cycle. Since synchronous interfaces are difficult to implement, we chose to have an asynchronous interface between the PEs. This is acheived with the help of streams between the PEs. A stream is a FIFO buffer whose size can automatically grow/shrink based on the data input to/output from the stream.

Once a PE element reads the data from the stream to a local buffer, it starts the computation. During computation, the read data can be written to the output stream. This helps in parallelizing the compute and write operations of the PE. This pipelining at the function level is achieved with the help of dataflow pragma.

The HLS implementation and the hardware equivalent of the same for N=2 (4x4 systolic array) are shown in Fig 3. and Fig 4. From Fig 4., we can see that each PE has six streams in total (3 streams for the input and 3 streams for the output)



```
ComputeLoop:
for (int i = 0; i < N; i++)
  {
   #pragma HLS UNROLL

   for (int j = 0; j < N; j++)
       {
        #pragma HLS UNROLL
        //Creates nine instances of processing elements in hardware (Considering N=3)
        //Format is ProcessingElement(a_in, a_out, b_in, b_out, c_in, c_out)
        ProcessingElement(a_pipes[i][j], a_pipes[i][j + 1], b_pipes[i][j], b_pipes[i + 1][j], c_pipes[i][j], c_pipes[i][j+1], i, j);
        }
   }
```

Fig. 3: HLS Implementation

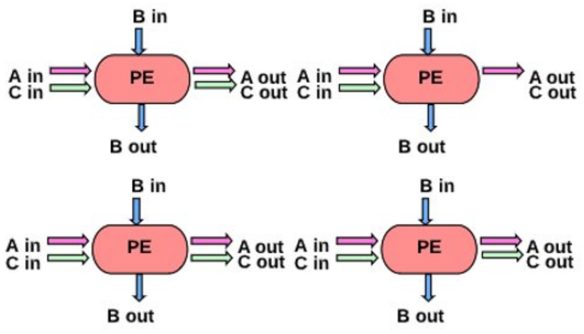


Fig. 4: Hardware Implementation

B. *Tiling Matrix Multiplication*

Owing to the limited number of resources, we implemented tiling in order to accommodate multiplication of large matrices. Fig 5. demonstrates how tiling is performed in matrix multiplication.

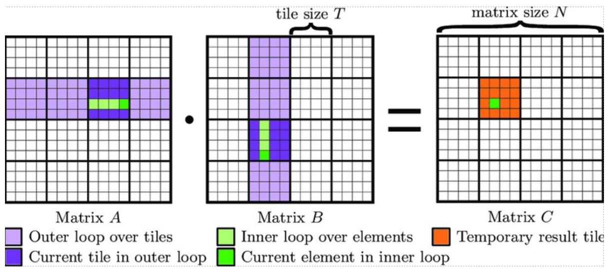


Fig. 5: Tiled Matrix Multiplication

As shown in the figure above, in a loop, tiles from both input matrices A and B are loaded first. The tiles are then multiplied and the result is accumulated partially. Once the loop ends, the partial output is written into the output tile of Matrix C. We use output stationary, which means a tile from Matrix C is loaded only once in a loop unlike the input matrices A and B. Once an output tile is ready with it's output values, it is written back to DRAM.

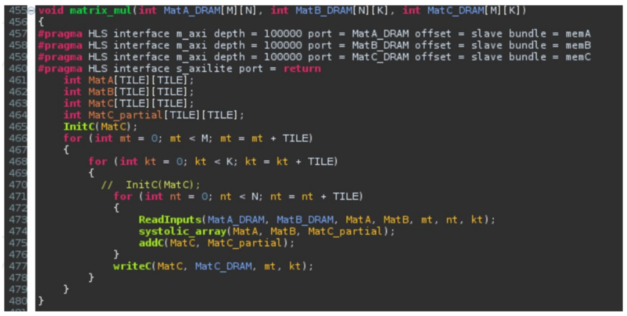
VI. Optimization

A. *Baseline Design:*

Before starting any optimization in HLS the latency of the baseline fully functional tiled matrix multiplication using systolic arrays was recorded. This baseline code is what is iteratively improved to get speedup in results.

Our baseline matrix multiplication code involves tiling the input matrices and then reading the inputs from the DRAM, passing them through the systolic array where the computation is done in multiple PE's, accumulation of partial results and finally writing the output value to DRAM. This is indicated in the Fig 6.

The Latency Achieved at this stage is 1.8 seconds.



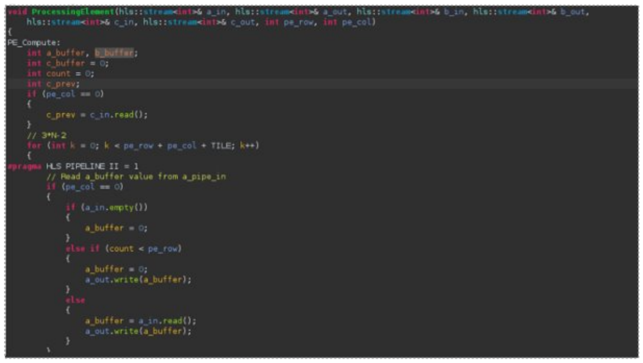
```
void matrix_mul(int MatA_DRAM[M][N], int MatB_DRAM[N][K], int MatC_DRAM[M][K])
{
#pragma HLS interface m_axi depth = 100000 port = MatA_DRAM offset = slave bundle = memA
#pragma HLS interface m_axi depth = 100000 port = MatB_DRAM offset = slave bundle = memB
#pragma HLS interface m_axi depth = 100000 port = MatC_DRAM offset = slave bundle = memC
#pragma HLS interface s_axilite port = return
    int MatA[TILE][TILE];
    int MatB[TILE][TILE];
    int MatC[TILE][TILE];
    int MatC_partial[TILE][TILE];
    InitC(MatC);
    for (int mt = 0; mt < M; mt = mt + TILE)
    {
        for (int kt = 0; kt < K; kt = kt + TILE)
        {
          //  InitC(MatC);
            for (int nt = 0; nt < N; nt = nt + TILE)
            {
                ReadInputs(MatA_DRAM, MatB_DRAM, MatA, MatB, mt, nt, kt);
                systolic_array(MatA, MatB, MatC_partial);
                addC(MatC, MatC_partial);
            }
            writeC(MatC, MatC_DRAM, mt, kt);
        }
    }
}
```

Fig. 6: Baseline Code

B. *Pipelining:*

Pipeline is a very powerful pragma available in HLS to improve latency. In our implementation we use pipeline in two major places.

1) *Pipelining the PEs: :* The processing elements perform the element-wise multiplication and addition operations in a given cycle. Within a given PE this operation can be pipelined across different cycles as shown in Fig 7. Note that this PE function is called multiple times in the systolic array.

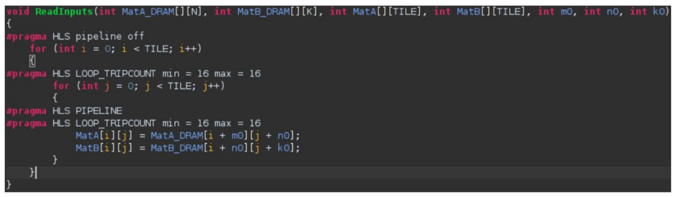


```
void ProcessingElement(hls::stream<int>& a_in, hls::stream<int>& a_out, hls::stream<int>& b_in, hls::stream<int>& b_out,
    hls::stream<int>& c_in, hls::stream<int>& c_out, int pe_row, int pe_col)
{
PE_Compute:
    int a_buffer, b_buffer;
    int c_buffer = 0;
    int count = 0;
    int c_prev;
    if (pe_col == 0)
    {
        c_prev = c_in.read();
    }
    // 3*N-2
    for (int k = 0; k < pe_row + pe_col + TILE; k++)
    {
#pragma HLS PIPELINE II = 1
        // Read a_buffer value from a_pipe_in
        if (pe_col == 0)
        {
            if (a_in.empty())
            {
                a_buffer = 0;
            }
            else if (count < pe_row)
            {
                a_buffer = 0;
                a_out.write(a_buffer);
            }
            else
            {
                a_buffer = a_in.read();
                a_out.write(a_buffer);
            }
        }
```

Fig. 7: Pipelining within a Processing Element

2) *Pipelining while reading Data: :* Since we are processing extrememly large matrices, sufficient tiling will be required. This will in-turn lead to multiple memory accesses from/to the DRAM. The speed at which the memory read operation happens can be improved by using pipeline pragma as shown in Fig 8. Since our matrices are of equal size and symmetric, the memory read operation for both the input matrices can be done within the same 'for loop'.

The Latency Achieved at this stage is 0.348 seconds.

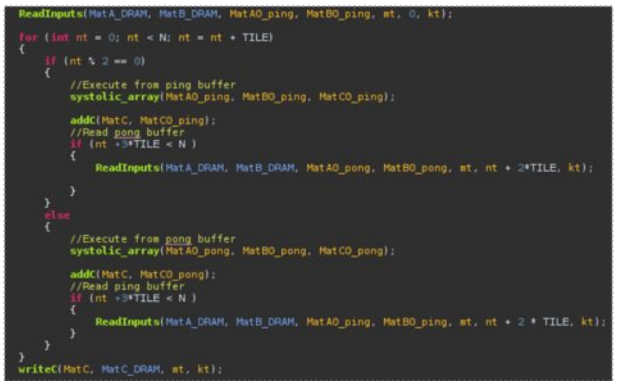


```
void ReadInputs(int MatA_DRAM[][N], int MatB_DRAM[][K], int MatA[][TILE], int MatB[][TILE], int m0, int n0, int k0)
{
#pragma HLS pipeline off
    for (int i = 0; i < TILE; i++)
    {
#pragma HLS LOOP_TRIPCOUNT min = 16 max = 16
        for (int j = 0; j < TILE; j++)
        {
#pragma HLS PIPELINE
#pragma HLS LOOP_TRIPCOUNT min = 16 max = 16
            MatA[i][j] = MatA_DRAM[i + m0][j + n0];
            MatB[i][j] = MatB_DRAM[i + n0][j + k0];
        }
    }
}
```

Fig. 8: Pipelining while accessing DRAM.

C. Ping Pong Buffers:

In ping pong buffers, two buffers are operated simultaneously. This way, two operations involving different data inputs can be done parallely as there is no memory dependency. In our implementation of ping pong buffers as shown in Fig 9., one buffer is used to read data from memory for the next tile while the systolic array operation is done for the current tile. These two operations can be done at the same time as they are operating on different sets of data.



```
ReadInputs(MatA_DRAM, MatB_DRAM, MatA0_ping, MatB0_ping, mt, 0, kt);

for (int nt = 0; nt < N; nt = nt + TILE)
{
    if (nt % 2 == 0)
    {
        //Execute from ping buffer
        systolic_array(MatA0_ping, MatB0_ping, MatC0_ping);

        addC(MatC, MatC0_ping);
        //Read pong buffer
        if (nt +3*TILE < N )
        {
            ReadInputs(MatA_DRAM, MatB_DRAM, MatA0_pong, MatB0_pong, mt, nt + 2*TILE, kt);

        }
    }
    else
    {
        //Execute from pong buffer
        systolic_array(MatA0_pong, MatB0_pong, MatC0_pong);

        addC(MatC, MatC0_pong);
        //Read ping buffer
        if (nt +3*TILE < N )
        {
            ReadInputs(MatA_DRAM, MatB_DRAM, MatA0_ping, MatB0_ping, mt, nt + 2 * TILE, kt);
        }
    }
}
writeC(MatC, MatC_DRAM, mt, kt);
```

Fig. 9: Ping Pong Buffer Implementation.

The Latency Achieved at this stage is 0.269 seconds.

D. 2 Systolic Arrays at the same time:

A single systolic array is 16x16 in size. For a given board if just one systlic array is running, the resource utilization is generally quite low. We can make adequate use of the resources while also reducing the latency by starting 2 systolic arrays at the same time. This operation is coupled with the ping pong buffer operation at this stage as shown in Fig 10.

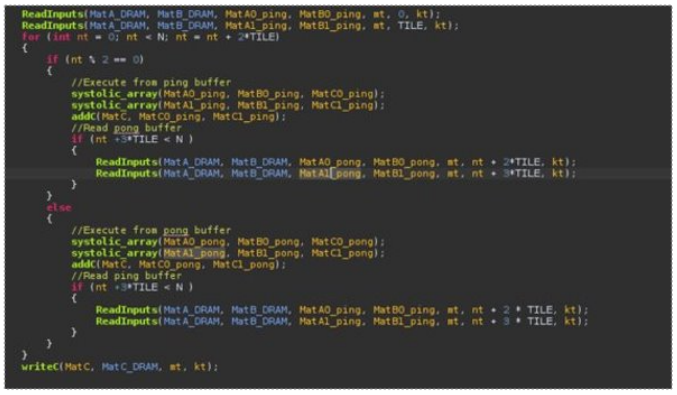
The Latency Achieved at this stage is 0.222 seconds.

E. 4 Systolic Arrays at the same time:

Since parallel systolic arrays reduce the latency, we implemented four systolic arrays that could process different tiles at the same time, as shown in Fig 11. This increased the resource utilisation, but gave a significant speedup compared the base code of one systolic array.

The Latency Achieved at this stage is 0.189 seconds.

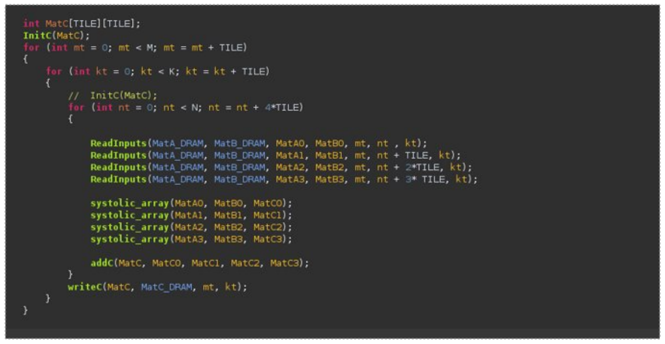
A summary of the different optimization techniques, the latency achieved and the speedup is consolidated in a table.



```
ReadInputs(MatA_DRAM, MatB_DRAM, MatA0_ping, MatB0_ping, mt, 0, kt);
ReadInputs(MatA_DRAM, MatB_DRAM, MatA1_ping, MatB1_ping, mt, TILE, kt);
for (int nt = 0; nt < N; nt = nt + 2*TILE)
{
    if (nt % 2 == 0)
    {
        //Execute from ping buffer
        systolic_array(MatA0_ping, MatB0_ping, MatC0_ping);
        systolic_array(MatA1_ping, MatB1_ping, MatC1_ping);
        addC(MatC, MatC0_ping, MatC1_ping);
        //Read pong buffer
        if (nt +3*TILE < N )
        {
            ReadInputs(MatA_DRAM, MatB_DRAM, MatA0_pong, MatB0_pong, mt, nt + 2*TILE, kt);
            ReadInputs(MatA_DRAM, MatB_DRAM, MatA1_pong, MatB1_pong, mt, nt + 3*TILE, kt);
        }
    }
    else
    {
        //Execute from pong buffer
        systolic_array(MatA0_pong, MatB0_pong, MatC0_pong);
        systolic_array(MatA1_pong, MatB1_pong, MatC1_pong);
        addC(MatC, MatC0_pong, MatC1_pong);
        //Read ping buffer
        if (nt +3*TILE < N )
        {
            ReadInputs(MatA_DRAM, MatB_DRAM, MatA0_ping, MatB0_ping, mt, nt + 2 * TILE, kt);
            ReadInputs(MatA_DRAM, MatB_DRAM, MatA1_ping, MatB1_ping, mt, nt + 3 * TILE, kt);
        }
    }
}
writeC(MatC, MatC_DRAM, mt, kt);
```

Fig. 10: 2 simultaneous systolic arrays implementation

Sl No.	Optimization	Latency (sec)	Speed-Up
1	Baseline Systolic Array	1.8	-
2	Pipeline Pragma	0.348	5.17x
3	Ping Pong Buffers	0.269	6.69x
4	2 Systolic Arrays + Ping Pong Buffers	69 (best case)	22x
5	4 Systolic Arrays	0.189	9.52x

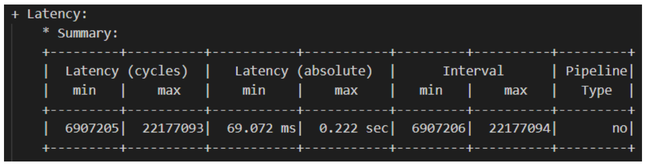


```
int MatC[TILE][TILE];
InitC(MatC);
for (int mt = 0; mt < M; mt = mt + TILE)
{
    for (int kt = 0; kt < K; kt = kt + TILE)
    {
        //  InitC(MatC);
        for (int nt = 0; nt < N; nt = nt + 4*TILE)
        {

            ReadInputs(MatA_DRAM, MatB_DRAM, MatA0, MatB0, mt, nt , kt);
            ReadInputs(MatA_DRAM, MatB_DRAM, MatA1, MatB1, mt, nt + TILE, kt);
            ReadInputs(MatA_DRAM, MatB_DRAM, MatA2, MatB2, mt, nt + 2*TILE, kt);
            ReadInputs(MatA_DRAM, MatB_DRAM, MatA3, MatB3, mt, nt + 3* TILE, kt);

            systolic_array(MatA0, MatB0, MatC0);
            systolic_array(MatA1, MatB1, MatC1);
            systolic_array(MatA2, MatB2, MatC2);
            systolic_array(MatA3, MatB3, MatC3);

            addC(MatC, MatC0, MatC1, MatC2, MatC3);
        }
        writeC(MatC, MatC_DRAM, mt, kt);
    }
}
```

Fig. 11: 4 simultaneous systolic arrays implementation

VII. HLS Synthesis Results

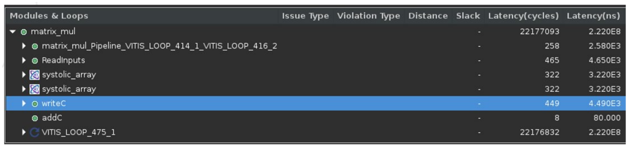
After various optimizations, our current design has two 16x16 systolic arrays in parallel along with ping pong buffers for memory access in order to overcome the memory overhead. Figure 12. shows the latency of our current design. We achieved a lowest latency of 69ms and a worst case latency of 222ms for a 512x512 matrix multiplication.



```
+ Latency:
    * Summary:
    +---------+---------+----------+----------+---------+---------+---------+
    |  Latency (cycles) |  Latency (absolute) |      Interval     | Pipeline|
    |   min   |   max   |    min   |    max   |   min   |   max   |   Type  |
    +---------+---------+----------+----------+---------+---------+---------+
    |  6907205| 22177093| 69.072 ms| 0.222 sec|  6907206| 22177094|       no|
    +---------+---------+----------+----------+---------+---------+---------+
```

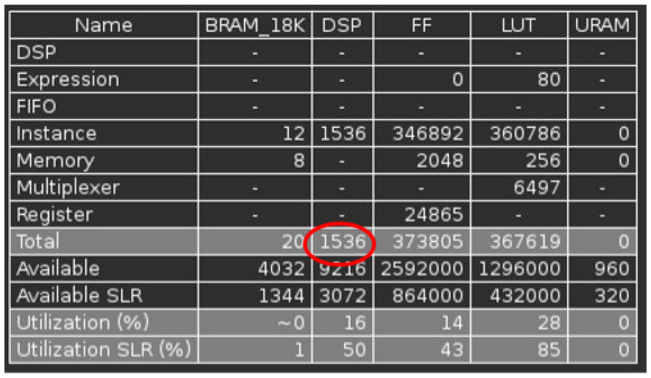
Fig. 12: Latency of the complete design

Figure 13. shows the latency of each subroutine in our implementation. A systolic array function is active for 322 cycles whereas the read_inputs and write_outputs functions take around 460 cycles or even more. Clearly, the bottle neck is memory/DRAM access.



Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)
matrix_mul				-	22177093	2.220E8
matrix_mul_Pipeline_VITIS_LOOP_414_1_VITIS_LOOP_416_2				-	258	2.580E3
ReadInputs				-	465	4.650E3
systolic_array				-	322	3.220E3
systolic_array				-	322	3.220E3
writeC				-	449	4.490E3
addC				-	8	80.000
VITIS_LOOP_475_1				-	22176832	2.220E8

Fig. 13: Latency of each function in the design



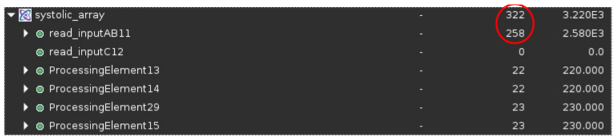
Name	BRAM_18K	DSP	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	80	-
FIFO	-	-	-	-	-
Instance	12	1536	346892	360786	0
Memory	8	-	2048	256	0
Multiplexer	-	-	-	6497	-
Register	-	-	24865	-	-
Total	20	1536	373805	367619	0
Available	4032	9216	2592000	1296000	960
Available SLR	1344	3072	864000	432000	320
Utilization (%)	~0	16	14	28	0
Utilization SLR (%)	1	50	43	85	0

Fig. 14: Resource utilization of the design

Fig 14. presents the resource utilization of our current design. The BRAM usage is not even 1%, which is the least among all. However, 43% of the FFs and 85% of the LUTs are consumed by this design. This is because of the *hls_stream* data type which uses more FFs instead of BRAM units. HLS streams are nothing but FIFOs and FIFOs are fundamentally made of flipflops. This technique of using HLS streams and avoiding BRAM usage also helps in decreasing memory access time.

Let's look at DSPs' usage, which is quite interesting. The number of DSPs used by the design is 1536, which is around 16% of the resources available on the FPGA board. In this design, the tile size or the systolic array size is 16x16. This implies that ideally 256 DSPs should be used by a systolic array. So, for two systolic arrays running in parallel in our design, 512 DSPs should be consumed ideally. But the number we see here is 3x the expected number. This is because our design uses integer matrices (floating point) which is 32-bits wide (precision). HLS uses 3 DSPs to perform integers' multiplication. Therefore, it's 512x3 which is equal to 1536 DSPs. When fixed point numbers are used in the design, this number comes down to the expected value.

Fig 15. shows the time taken by *systolic_array* function. This function takes 322 cycles in total, read_input and *write_result* functions take 258 cycles. It seems that reading all the data into the streams and writing back the data completely into the output streams is the bottleneck again. But this overhead is not due to DRAM access, but instead something which cannot be avoided without any advanced optimization techniques, which are not covered in this paper. A PE takes 22 cycles to compute the multiplication at the minimum and 51 cycles at most. These numbers are different because of the warmup and cooldown times. The PEs in the initial rows and columns are active in computation for lesser number of cycles than that of the PEs on the other end of the systolic array. The last PEs are active since the beginning but need to compute for more cycles because of the time taken by the PEs in the systolic array to pass the values from one end to the other end of the array. However, the overall latency is still 322 cycles but not 258+258+51 cycles. It's because reading inputs, computing in the PEs and writing back results are pipelined/overlapped. It means that while a result is being computed in a PE, the input values can be passed over to the next PEs in the systolic arrays using streams and also the next input values can be loaded from the input stream. This overlapping of tasks using streaming is called dataflow architecture. The *#pragma HLS dataflow* allows us to do so.



systolic_array	-	322	3.220E3
read_inputAB11	-	258	2.580E3
read_inputC12	-	0	0.0
ProcessingElement13	-	22	220.000
ProcessingElement14	-	22	220.000
ProcessingElement29	-	23	230.000
ProcessingElement15	-	23	230.000

Fig. 15: Latency of systolic array subroutine

VIII. HLS Cosim Results

We implemented a 3x3 systolic array on xc7z020clg400-1 board and the results obtained from the board match the theoretical values. In this section, we analyze the results obtained after cosimulation.

Fig 16. shows the dataflow viewer after cosimulation. The blue boxes indicate the PEs while the green arrows connecting them represent the channels. The average, minimum and the maximum latency of the PEs are shown in the graph. This viewer was used to verify the connection between the PEs.

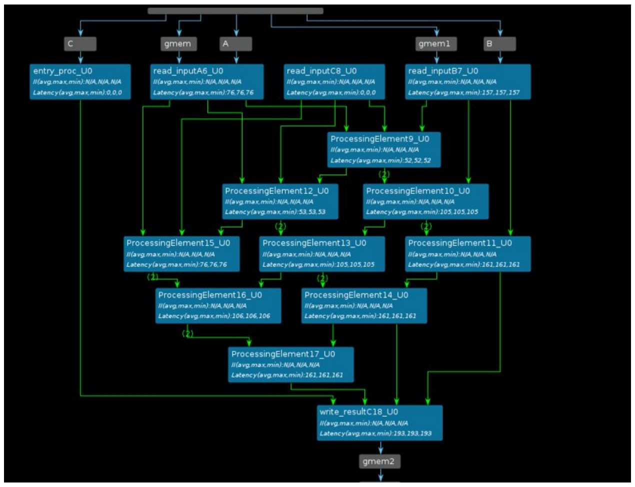
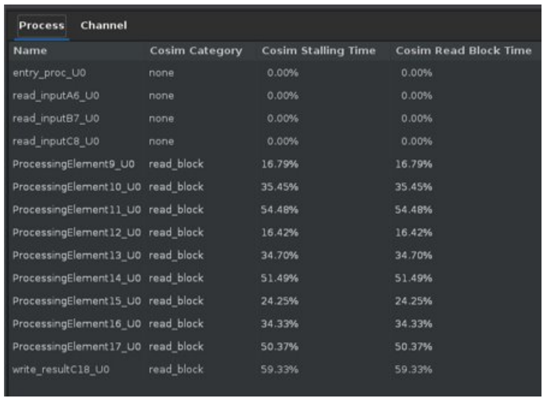

C
gmem
A
gmem1
B
entry_proc_U0
II(avg,max,min):N/A,N/A,N/A
Latency(avg,max,min):0,0,0
read_inputA6_U0
II(avg,max,min):N/A,N/A,N/A
Latency(avg,max,min):76,76,76
read_inputC8_U0
II(avg,max,min):N/A,N/A,N/A
Latency(avg,max,min):0,0,0
read_inputB7_U0
II(avg,max,min):N/A,N/A,N/A
Latency(avg,max,min):157,157,157
ProcessingElement9_U0
II(avg,max,min):N/A,N/A,N/A
Latency(avg,max,min):52,52,52
(2)
ProcessingElement12_U0
II(avg,max,min):N/A,N/A,N/A
Latency(avg,max,min):53,53,53
ProcessingElement10_U0
II(avg,max,min):N/A,N/A,N/A
Latency(avg,max,min):105,105,105
(2)
(2)
ProcessingElement15_U0
II(avg,max,min):N/A,N/A,N/A
Latency(avg,max,min):76,76,76
ProcessingElement13_U0
II(avg,max,min):N/A,N/A,N/A
Latency(avg,max,min):105,105,105
ProcessingElement11_U0
II(avg,max,min):N/A,N/A,N/A
Latency(avg,max,min):161,161,161
(2)
(2)
ProcessingElement16_U0
II(avg,max,min):N/A,N/A,N/A
Latency(avg,max,min):106,106,106
ProcessingElement14_U0
II(avg,max,min):N/A,N/A,N/A
Latency(avg,max,min):161,161,161
(2)
ProcessingElement17_U0
II(avg,max,min):N/A,N/A,N/A
Latency(avg,max,min):161,161,161
write_resultC18_U0
II(avg,max,min):N/A,N/A,N/A
Latency(avg,max,min):193,193,193
gmem2

Fig. 16: Data flow viewer

Fig 17. shows the data flow property of the processes (PEs). The Cosim stall time column indicates what percentage of the simulation time was spent stalling for this particular process. Additionally, Cosim Read Block Time or Cosim Write Block Time shows the percentage of time blocked from reading or writing to the process's channels



Name	Cosim Category	Cosim Stalling Time	Cosim Read Block Time
entry_proc_U0	none	0.00%	0.00%
read_inputA6_U0	none	0.00%	0.00%
read_inputB7_U0	none	0.00%	0.00%
read_inputC8_U0	none	0.00%	0.00%
ProcessingElement9_U0	read_block	16.79%	16.79%
ProcessingElement10_U0	read_block	35.45%	35.45%
ProcessingElement11_U0	read_block	54.48%	54.48%
ProcessingElement12_U0	read_block	16.42%	16.42%
ProcessingElement13_U0	read_block	34.70%	34.70%
ProcessingElement14_U0	read_block	51.49%	51.49%
ProcessingElement15_U0	read_block	24.25%	24.25%
ProcessingElement16_U0	read_block	34.33%	34.33%
ProcessingElement17_U0	read_block	50.37%	50.37%
write_resultC18_U0	read_block	59.33%	59.33%

Fig. 17: Data flow property of the processes

Fig 18. shows the data flow property of one of the output c channels.It shows the clock cycles for which the channel is empty, full or number of elements present in it. This is a useful measure of the utilization of the channel and also helpful in determining if the channel is over-utilized (leads to deadlock) or under-utilized.

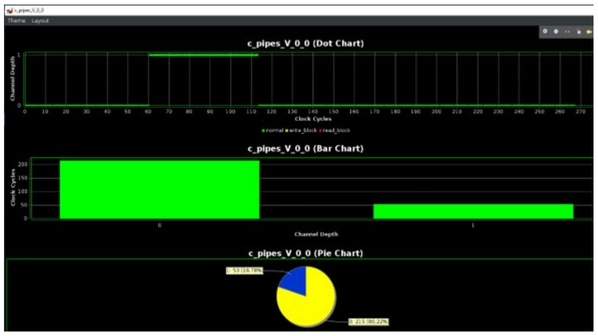


Fig. 18: Data flow property of one of the channels

Fig 19. shows the waveform viewer for one of the PEs. Since data is streamed from the input to the output streams, Ain and Bin values are instantaneously available at Aout and Bout channels. However, Cout is produced after a few clock cycles from the time Cin is available.

Fig 20. shows the activity time of all the PEs in the systolic array. As seen from the image, the PEs at the top left part operate for less time compared to the other PEs. This is expected since the PEs at the bottom right part have to wait for the data to flow sequentially through the systolic array.

IX. Key learnings

A few of the challenges that we faced with dataflow architecture and the learnings from them are listed below:

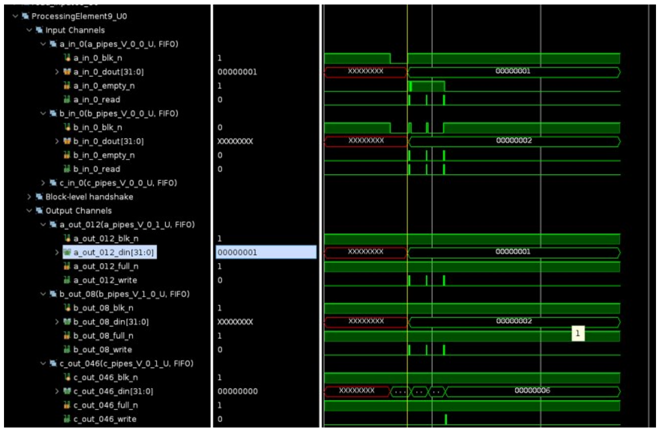

ProcessingElement9_U0
Input Channels
a_in_0(a_pipes_V_0_0_U, FIFO)
b_in_0(b_pipes_V_0_0_U, FIFO)
c_in_0(c_pipes_V_0_0_U, FIFO)
Block-level handshake
Output Channels
a_out_012(a_pipes_V_0_1_U, FIFO)
b_out_08(b_pipes_V_1_0_U, FIFO)
c_out_046(c_pipes_V_0_1_U, FIFO)

Fig. 19: Waveform viewer for one of the PEs

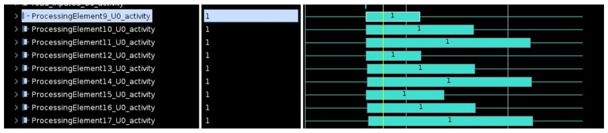

ProcessingElement9_U0_activity
ProcessingElement10_U0_activity
ProcessingElement11_U0_activity
ProcessingElement12_U0_activity
ProcessingElement13_U0_activity
ProcessingElement14_U0_activity
ProcessingElement15_U0_activity
ProcessingElement16_U0_activity
ProcessingElement17_U0_activity

Fig. 20: Activity time of all PEs

- Each stream should only have one producer and one consumer. If it has more than one, HLS throws an error. So, it is important to ensure that there are no multiple read/write functions to the streams.
- Due to the dynamic nature of the dataflow optimization, and the propensity of different parallel tasks to execute at different rates, it is possible that poorly sized dataflow channels can cause loss of performance and/or deadlock.
- Deadlocks due to insufficient FIFO depths always exhibit at least one blocked writer. If not, it is most likely a design issue - typically due to non-blocking reads or writes, or reads and writes conditioned by empty() and full().
- The co-simulation data helps with the FIFO sizing problem by tracking the maximum size of the FIFO during the course of the simulation and thereby giving a good reference point on how to size the FIFOs.

X. Conclusion

We presented a HLS design for matrix multiplication of large matrices in an efficient way. The size of the matrix is parameterized. The main component of our design is a 2-D systolic array which can be reused by tiling the matrices. The user can specify any matrix size and depending on the size of the systolic array defined, the tiling is done on the input and output matrices to reuse the same resources. The *hls_stream* variables were used to accommodate movement of data smoothly across the PEs. The speciality of this systolic array is the streaming or dataflow architecture employed by the system. This dataflow architecture enables overlapping of different tasks and efficiently promotes parallelism and pipelining of the tasks involved in matrix multiplication. The best latency achieved by our design to multiply a 512x512 size matrix is 69ms. However, GPUs can easily beat this performance due to their architecture. We have tried several optimizations such as using 4 systolic arrays running simultaneously, array partitioning and loop unrolling, loop fusion, etc.

XI. Future Work

Our current design calls for a huge improvement in terms of design, performance and resource utilization. The following are some of the follow-on ideas for our current design:

- The memory access is the bottleneck in the present design of our systolic array. This has to be addressed using advanced techniques like data packing, more efficient implementation of ping pong buffers, etc.
- We currently use 2 systolic arrays in parallel with ping pong buffers. We have also experimented running 4 systolic arrays in parallel which is exhausting all the resources on the board. Therefore, there is a limit to number of systolic arrays we could run simultaneously. So, we believe it's better to increase the size of each systolic array instead of increasing the number of arrays.
- Compare/analyse our design with existing works from which our design was inspired.
- Our design certainly works for multiplication of square matrices. We are yet to test the design for other matrices by generalizing our design parameters (size of matrices).
- There are many applications of matrix multiplication in the world today. One closest example is convolution. We would like to extend our systolic array architecture from matrix multiplication to implementing convolution in the future. We believe this is a great area for design exploration.

References

[1] Akshay Dua, Yixing Li, Fengbo Ren: Systolic-CNN: An OpenCL-defined Scalable Run-time-flexible FPGA Accelerator Architecture for Accelerating Convolutional Neural Network Inference in Cloud/Edge Computing, arXiv:2012.03177v1, Dec 2020

[2] Xuechao Wei, Cody Hao Yu, Peng Zhang, Youxiang Chen, Yuxin Wang, Han Hu, Yun Liang, Jason Cong: Automated Systolic Array Architecture Synthesis for High Throughput CNN Inference on FPGAs, 2017

[3] Johannes de Fine Licht ,Grzegorz Kwasniewski , Torsten Hoefler :Flexible Communication Avoiding Matrix Multiplication on FPGA with High-Level Synthesis , 2019

[4] Paolo Gorlani, Christian Plessl: High Level Synthesis Implementation of a Three-dimensional Systolic Array Architecture for Matrix Multiplications on Intel Stratix 10 FPGAs, arxiv 2021.

[5] Emina I. Milovanovic, Mile K. Stojcev, tatjana R. Nikolic: Design of Linear Systolic Arrays for Matrix Multiplication, 2014.