Verilog Implementation of MatRaptor: A Sparse-Sparse Matrix Multiplication Accelerator Based on Row-Wise Product

Hazem Taha
Dept. Electrical and Computer Engineering,
McMaster University
Hamilton, ON, Canada
tahah6@mcmaster.ca

Abstract— **This project presents a Verilog implementation and analysis of MatRaptor, a proposed hardware accelerator for efficient sparse-sparse matrix multiplication (SpSpMM) [1]. Utilizing a Row-Wise Product approach, MatRaptor addresses the challenges of computational efficiency and memory bandwidth in SpMM. By processing only non-zero elements in a row-major format, it ensures high data locality and parallelism without the need for inter-process synchronization. The compact and scalable design of MatRaptor demonstrates a marked improvement in performance on FPGA and ASIC platforms, making it an effective solution for high-speed SpSpMM tasks.**

Keywords—Matrix Multiplication, Accelerator, Sparse Matricies

I. Introduction

Sparse-Sparse Matrix multiplication is an essential operation in various scientific and engineering applications, especially in the realm of high-performance computing, artificial intelligence, and graph analytics. Sparse matrix multiplication, particularly the multiplication of two sparse matrices, is particularly challenging due to its irregular computation pattern and memory accesses.

Traditional solutions for sparse matrix multiplication primarily fall into two categories: inner product and outer product methods. Both solutions have their own advantages and disadvantages as follows:

A. Inner Product Method

The inner product method computes the product of two matrices by using dot product of vectors to calculate specific entry in the resultant matrix at a time. For two matrices $A\ (m \times n)\ and\ B\ (n \times p)$ the entry C_{ij} in the product matrix C is given by (1).

$$C[i,j] = \sum_{k=1}^{n} A[i,k] \times B[k,j] \tag{1}$$

Some advantages of the inner product method include that it can be parallelized and does not require synchronization as every entry can be calculated independently, it has low on-chip memory requirement. On the other hand, major disadvantages are inconsistent formatting for matrix storage as matrix A is stored in row-major format and matrix B in column-major format, and inefficient index matching as matching indices is required even for zero output value.

B. Outter Product Method

The outer product method calculates the matrix product by summing the partial sum matrices produced by taking the outer products of the column vectors of the first matrix with row vectors of the second matrix. For two matrices $A\ (m \times n)\ and\ B\ (n \times p)$ the resulting matrix C is given by (2).

$$C = \sum_{i=1}^{n} A[:,i] \times B[i,:] \tag{2}$$

Where $A[:,i]$ is the i-th column in matrix A and $B[i,:]$ is the i-th row in matrix B.

A disadvantage of the outer product method is that, although it can parallelized, synchronization must be applied to avoid race condition when accumulating the resulting partial sum matrices. The method also suffers from inconsistent formatting as well as high on-chip memory requirements because at each step we need to store the entire partial sum matrix in the buffer. One advantage of this method is its ability to efficiently handle sparse matrices by computing only the necessary non-zero elements, thus potentially reducing the number of computations and memory accesses required.

As can be seen, there are multiple challenges when choosing an algorithm to perform sparse-sparse matrix multiplication. These challenges include: inefficiency of the commonly used approaches, memory bandwidth utilization (due to low data reuse), and the unpredictability of the sparse output matrix, which complicates parallel computation because the positions of non-zero elements are unknown prior to computation and must be determined dynamically, often requiring complex and costly synchronization to avoid data races.

II. Solution

A. Row-Wise Product

In this proposed solution [1], both matrices A and B will be stored in memory in row-major format. At every step, a non-zero row of matrix A will be multiplied by all non-zero rows of matrix B as shown in (3).

$$C[i,:] = \sum_{k=1}^{n} A[i,k] \times B[k,:] \tag{3}$$

The core computation in this approach is scalar-vector product. Hence it computes only non-zero entries of the output matrices and does not require inefficient index matching of the inputs as in the inner product approach. Another advantage of the Row-Wise Product method is that it naturally lends itself to parallel computation as multiplication of different rows can be distributed across multiple processors or cores without the need for complex synchronization mechanisms. This absence of synchronization avoids the overhead and potential bottlenecks associated with coordinating parallel tasks, which can be a significant issue when dealing with large-scale parallel computations.

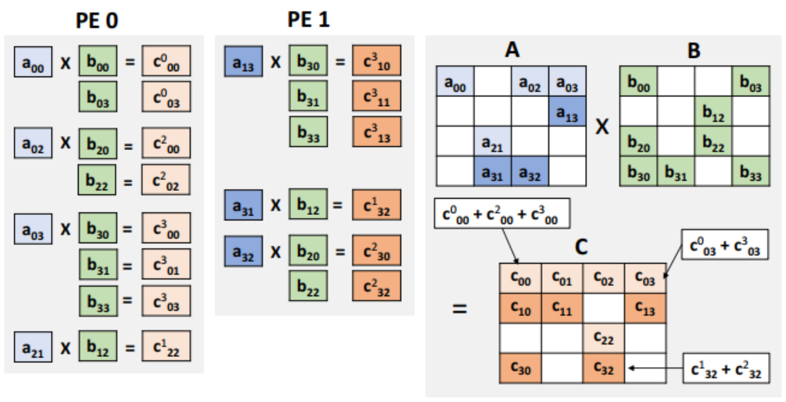


Fig. 1: **Parallelization of row-wise product on two Pes** — PE0 is assigned to rows 0 and 2 of input matrix A and computes rows 0 and 2 of output matrix C; PE1 is assigned rows 1 and 3 of the matrix A and computes rows 1 and 3 of the matrix C; The matrix B is shared between the two Pes [1]

The method also addresses the issue of memory bandwidth limitations by enhancing data reuse; since each row is loaded into the cache once and used for computation with all relevant non-zero rows of the other matrix, making the most out of the loaded data before it is evicted from the cache. Furthermore, since a single output row is computed at a time, the required output buffer size is much smaller is contrast with the outer product approach, which requires the entire output matrix to be stored on chip with a buffer size.

Fig. 1 gives an example which illustrates the computation of SpGEMM with two PEs using the row-wise product approach. Here, each of the two PEs reads entire rows of A and B matrices and writes entire rows of the output matrix C.

B. *C^2SR for Sparse-Sparse MM*

The widely used Compressed Sparse Row (CSR) format for representing sparse matrices presents several drawbacks in parallel processing scenarios [1]. CSR's structure, comprising arrays for non-zero values and column indices alongside an array for row pointers, leads to fragmented vectorized memory access, as memory requests may span multiple channels. This issue, alongside memory channel conflicts when multiple Processing Elements (PEs) access the same channel, is illustrated in Figure 2e. Furthermore, CSR can result in reading extraneous data not required by the PEs and necessitates synchronization for writing outputs, causing delays as each PE must wait for others processing preceding rows. These limitations, as shown in Figure 2e and 2f.

In addressing these limitations, the Channel Cyclic Sparse Row (C2SR) format emerges as a robust alternative [1]. The C2SR format, illustrated in Figure 2c and 2d, assigns matrix rows to memory channels in a round-robin fashion, ensuring that each row is uniquely mapped to a channel, thereby eliminating channel conflicts. This results in contiguous storage of non-zero elements within channels, enhancing spatial locality and enabling vectorized, efficient memory reads. Importantly, C2SR allows for independent and parallel writes to the output matrix by different PEs, significantly streamlining the computation process. The adoption of C2SR is thus key to optimizing parallel processing of sparse matrices.

C. *Original MatRaptor System Architecture*

The original proposed micro-architecture of MatRaptor consists of Sparse Matrix A Loaders (SpAL), Sparse Matrix B Loaders (SpBL), and compute Processing Elements (PEs) as shown in Fig.4. Also figure 3 shows an illustration of how MatRaptor works. You may refer to the original text to find more details, but I will explain the important parts within the system architecture. The PEs form a one-dimensional systolic array that operates in parallel across N rows, with matrix A rows distributed in a round-robin fashion. operations, and the queues on the right, adder tree and minimum column index logic performing phase II of the merge operations. [1]

SpALs are responsible for reading non-zero elements from matrix A, along with their indices, and forwarding this data to the SpBLs. SpBLs, upon receiving data from SpALs, retrieve the corresponding non-zero elements from matrix B using the column indices provided. They then send both sets of values and indices to the PEs for computation. The Pes perform the actual multiplication and then merge these products into the resultant matrix C, which is written back to the main memory. Similarly, SpBLs operate on matrix B and maintain a queue for outstanding requests and responses, ensuring continuous data flow.

The PEs receive pairs of values from the SpBLs and carry out multiplication and merging operations. Each PE has a multiplier and two sets of queues to manage the data. These queues are part of a double buffering system that allows for simultaneous execution of multiplication (Phase I) and merging (Phase II) operations, leading to optimal utilization of the multipliers and no stalling in the pipeline. Furthermore, the queues are connected to an adder tree and a logic for selecting the minimum column index, which aids in the efficient merging of partial sums. This setup allows for continuous computation and merging phases, with inactive components shown in gray and dotted lines in Fig. 4b, indicating the two phases of operation.

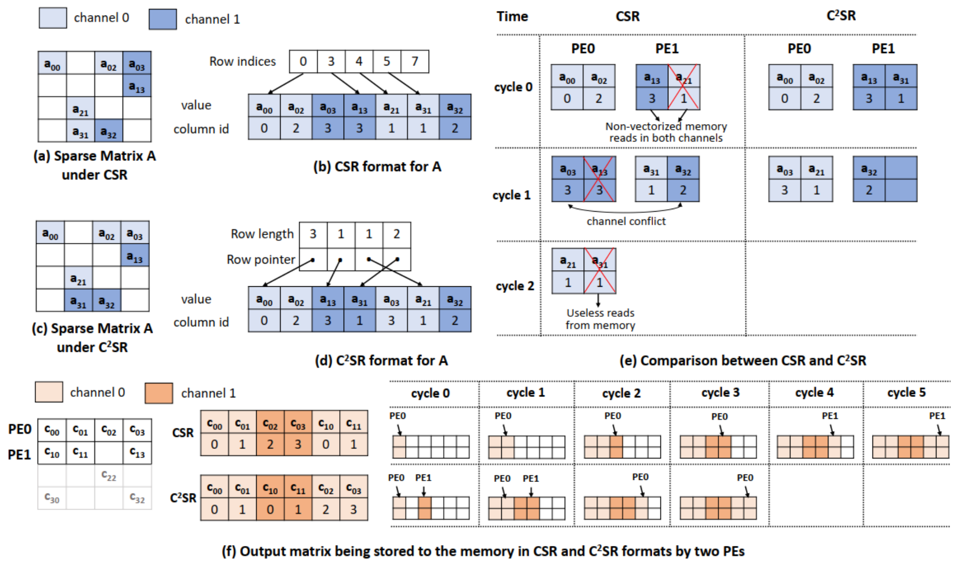

channel 0
channel 1
(a) Sparse Matrix A under CSR
Row indices
value
column id
(b) CSR format for A
(c) Sparse Matrix A under C²SR
Row length
Row pointer
(d) C²SR format for A
Time
CSR
C²SR
PE0
PE1
cycle 0
cycle 1
cycle 2
Non-vectorized memory reads in both channels
channel conflict
Useless reads from memory
(e) Comparison between CSR and C²SR
cycle 3
cycle 4
cycle 5
(f) Output matrix being stored to the memory in CSR and C²SR formats by two PEs

Fig. 2: **Comparison of sparse storage formats** — (a) shows the sparse matrix A in its dense representation; (b) shows matrix A stored in the CSR storage format and assignment of non-zero elements to two channels using 4-element channel interleaving; (c) shows the same sparse matrix A in dense representation where each row is mapped to a unique channel; (d) shows the sparse matrix A in C2SR format; (e) shows the cycle by cycle execution of two PEs where PE0 and PE1 read rows 0 and 2, and rows 1 and 3 of matrix A, respectively; and (f) shows the output matrix C being written by the two PEs in CSR and C^{2}SR formats.

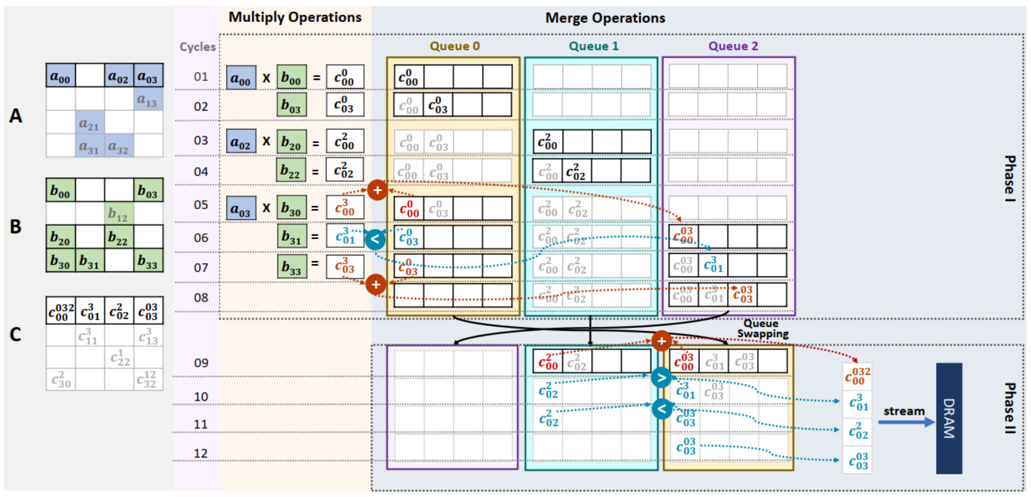


Fig. 3: **Illustration of multiply and merge operations involved in computing the results for a single row of the output matrix** — Phase I corresponds to the cycles when the multiplications are performed and the result of the multiplication is merged with the (*data; col id*) values in one of the queues; and Phase II corresponds to the cycles when the (*data; col id*) values in different queues are merged together and streamed out to the DRAM.

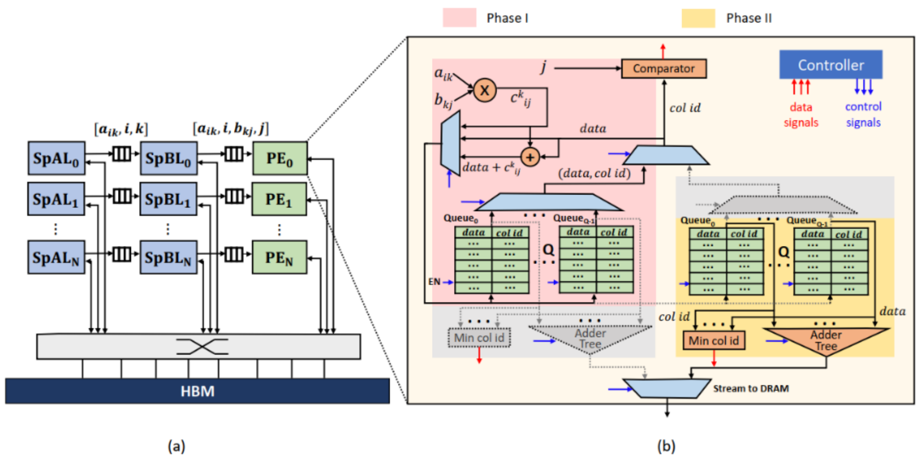


Fig. 4: **MatRaptor architecture** — (a) shows the MatRaptor microarchitecture consisting of Sparse A Loaders (SpALs), Sparse B Loaders (SpBLs), Processing Elements (PEs), system crossbar and high-bandwidth memory (HBM); (b) shows the microarchitecture of a single PE consisting of multipliers, adders, comparator and queues on the left performing phase I of the multiply and merge operations, and the queues on the right, adder tree and minimum column index logic performing phase II of the merge operations.

D. Modified MatRaptor System Architecture

For this project, a modified version of the MatRaptor architecture has been developed to make it simpler and more efficient (this claim will need more tests and research to proof it's validity). The main changes done was within the PE logic and how it handles the multiplication and merging operations.

Figure 5 shows an illustration of how the new PEs works. As seen in figure 5(a), the Multiply phase, the first value from the designated row of matrix A is being multiplied with all values from the target row of matrix B (matching A's element column index). All the results are stored in the first queue indexed 0 which is declared **active queue**. It's important to notice that each of the results of multiplication doesn't get pushed to the queue, but rather directly written in a specific index (matching B's element column index). This is critical to preserve the information we have about the column index of each multiplication without having to write that separately. Next, the next queue with index 1 becomes the active queue. The next element of the designated row in matrix A is multiplied with the target row in matrix B and the results are stored in the active row. This keeps happening until all values of row A get multiplied with rows of B. Hence, we will need to have a number of queues equal to the number of columns of A.

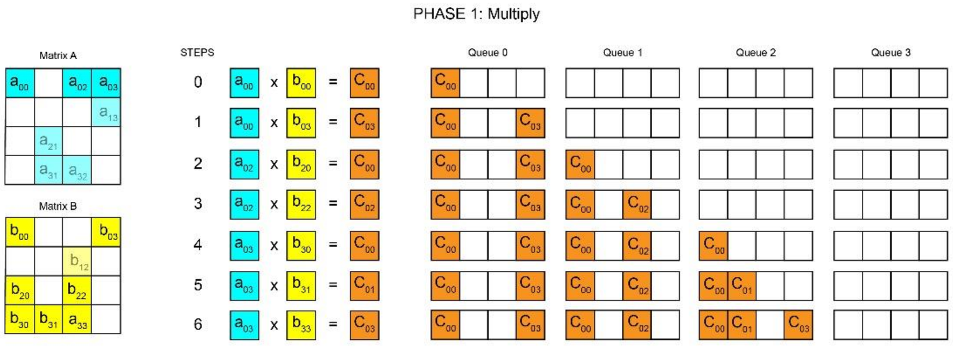
In the second stage Merge as shown in figure 5(b), if at least one element at index 0 in all queues is not zero, all index 0 elements are added together and written back to the DRAM along with column index 0. Then, index 0 elements get popped out of the queue while other elements are being swapped preparing for the next addition. The process repeats while the returned column index is incremented until all elements are popped out of the queues.
Figure 6 shows the new architecture of the modified MatRaptor system applying the discussed techniques. Notice that some elements appear in both phases in figure 6 such as the Queues and PE Control Unit. These are the exact elements but at different timing. Other elements that only appear exclusively in a certain phase such as adders, multipliers, or connections, are only used in a specific phase and are hidden in the other phase for simplicity.

E. MatRaptor RTL Testings and Validation

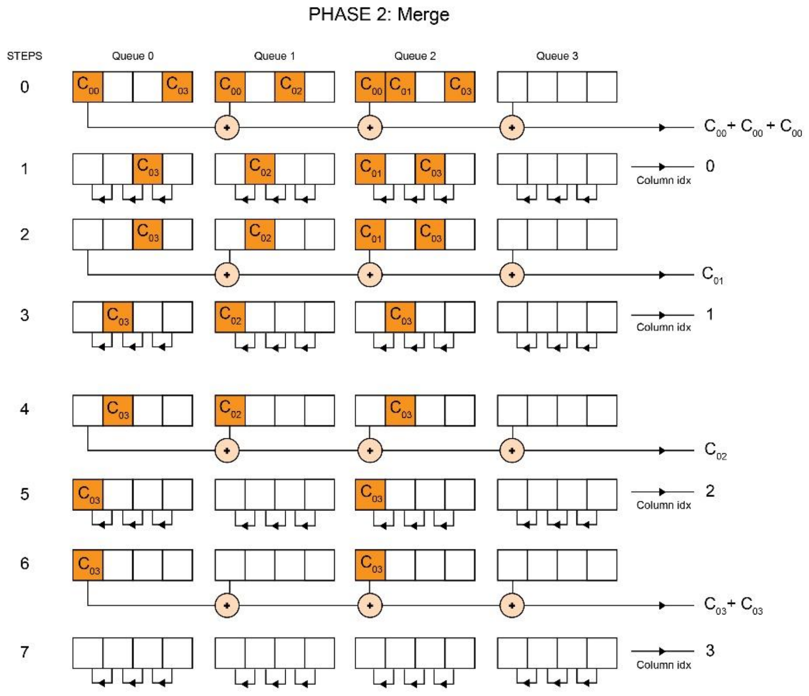
At the end of the RTL development process, a testbench has been designed to validate the functionality and timing of the proposed architecture. The testbench was designed as follows:

1. MatRaptor (2 PEs) has been instantiated.
2. The external memory connected to MatRaptor was composed of three 2-D arrays of registers (representing BRAMs). The reason we need 3 separate blocks each PE along with the control unit needs a dedicated memory channel so they can operate in parallel.
3. Two matrices A and B were initialized in the memories in C^2SR format.
4. The pointers to A, B and a pointer to the row headers of output matrix are given to MatRaptor RTL along with 2 pointers for each PE to write the output values and column indices.
5. MatRaptor is signaled to start.

After plotting the waveform, it was seen that the output row heads (length, pointers) and the elements were exactly as expected and stored in the given locations. The waveform figure will not be put in the report as it would be difficult to trace any values. However, all code implementations will be given with the report.



(a)



(b)

Fig. 5: **Illustration of the modifed MatRaprot** — (a) shows first phase (multiplication); (b) shows the second phase (merging).

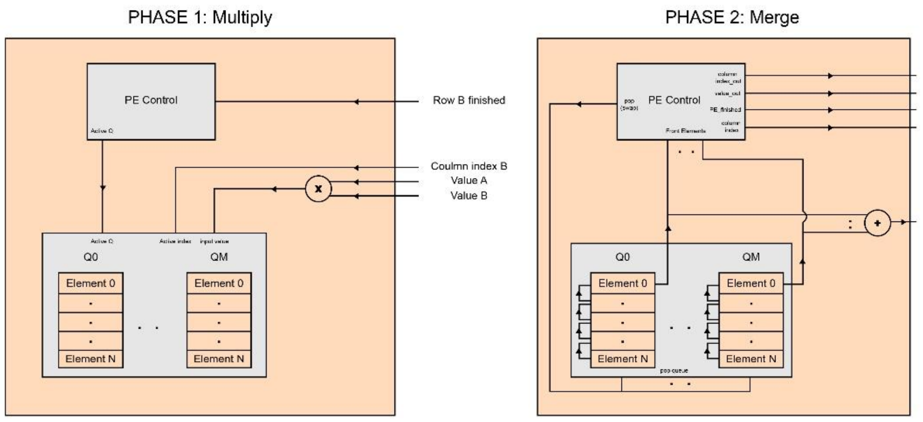


Fig. 6: **Architecture of the modified MatRaprot**

III. Integration with MicroBlaze

The first several attempts to program MicroBlaze on the PYNQ-Z2 SoC weren't successful for many board-specific complexities that came up. There were also very few tutorials and guidance online on how to solve these issues. However, when switched to GENESYS 2 FPGA board, the development was smoother and I managed to easily program the PL with MicroBlaze which is the very basic start in the integration.

After uploading and testing MicroBlaze on the FPGA, I faced another issue related to how can MatRaptor (a non-AXI) component connect to the DDR3. Furthermore, how can I instantiate 3 versions of the DDR3 as it was discussed in earlier sections that the architecture of MatRaptor requires 3 memory channels to be connected with it at the same time. I spent some time researching this and I knew that it would be nearly impossible to do this in the given time and tools.

The solution I thought about is to use BRAM blocks instead of the DDR3. As can be seen in the Block Design in figure 7, three dual-port BRAM blocks were instantiated. The first port of each BRAM is connected to MicroBlaze through AXI-GPIO that connects to AXI-Interconnect and then MicroBlaze. The second port of each BRAM is connected directly to MatRaptor. This allows both MicroBlaze and MatRaptor to read and write from the BRAM cell without any issues.

Furthermore, certain ports of MatRaptor (such as the input start signal and the input matrix pointers) were connected to MicroBlaze in the same way, through an AXI-GPIO. This method proved to be working as I did separate testing to assure the connectivity between the different parts in the design.

Finally, a C-code has been developed to run the entire process as follows:

1- Declaring the A and B matrices.

2- Converting the matrices into C^2SR format.

3- Loading matrices into designated locations in the BRAMs.

4- Validate that population of the BRAMs.

5- Initialize MatRaptor with the required pointers.

6- Pool over a finished signal from MatRaptor.

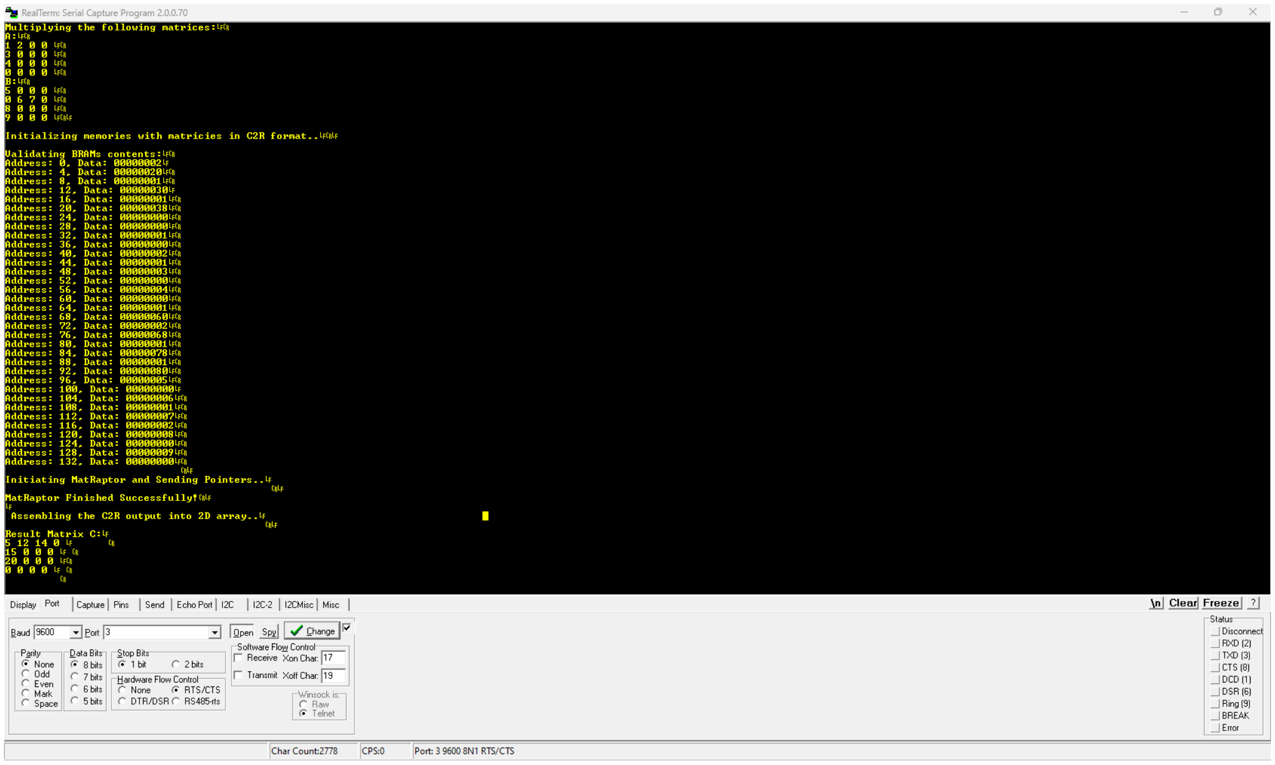
7- Fech the C^2SR output data from the memory and convert it into 2D Matrix C.

8- Print the final Matrix.

After many updates to the code and the hardware desing, The process ran sucssufly and the oupput was as expected. Shown in figure 8 is the UART output of runinig the code on MicrBlaze visualized in RealTerm.

IV. References

[1] N. Srivastava, H. Jin, J. Liu, D. Albonesi, and Z. Zhang, "Matraptor: A sparse-sparse matrix multiplication accelerator based on row-wise product," 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2020. doi:10.1109/micro50266.2020.00068.



```
Multiplying the following matrices:
A:
1 2 0 0
3 0 0 0
4 0 0 0
0 0 0 0
B:
5 0 0 0
0 6 7 0
8 0 0 0
9 0 0 0

Initializing memories with matricies in C2R format..

Validating BRAMs contents:
Address: 0, Data: 00000002
Address: 4, Data: 00000020
Address: 8, Data: 00000001
Address: 12, Data: 00000030
Address: 16, Data: 00000001
Address: 20, Data: 00000038
Address: 24, Data: 00000000
Address: 28, Data: 00000000
Address: 32, Data: 00000001
Address: 36, Data: 00000000
Address: 40, Data: 00000002
Address: 44, Data: 00000001
Address: 48, Data: 00000003
Address: 52, Data: 00000000
Address: 56, Data: 00000004
Address: 60, Data: 00000000
Address: 64, Data: 00000001
Address: 68, Data: 00000060
Address: 72, Data: 00000002
Address: 76, Data: 00000068
Address: 80, Data: 00000001
Address: 84, Data: 00000078
Address: 88, Data: 00000001
Address: 92, Data: 00000080
Address: 96, Data: 00000005
Address: 100, Data: 00000000
Address: 104, Data: 00000006
Address: 108, Data: 00000001
Address: 112, Data: 00000007
Address: 116, Data: 00000002
Address: 120, Data: 00000008
Address: 124, Data: 00000000
Address: 128, Data: 00000009
Address: 132, Data: 00000000

Initiating MatRaptor and Sending Pointers..

MatRaptor Finished Successfully!

 Assembling the C2R output into 2D array..

Result Matrix C:
5 12 14 0
15 0 0 0
20 0 0 0
0 0 0 0
```

Baud 9600 Port 3
Parity: [✓ None] Odd Even Mark Space
Data Bits: [✓ 8 bits] 7 bits 6 bits 5 bits
Stop Bits: [✓ 1 bit] 2 bits
Hardware Flow Control: None [✓ RTS/CTS] DTR/DSR RS485-rts
Software Flow Control: Receive Xon Char: 17 Transmit Xoff Char: 19
Char Count:2778 CPS:0 Port: 3 9600 8N1 RTS/CTS

Fig. 8: **MicroBlaze UART Readings on RealTerm**

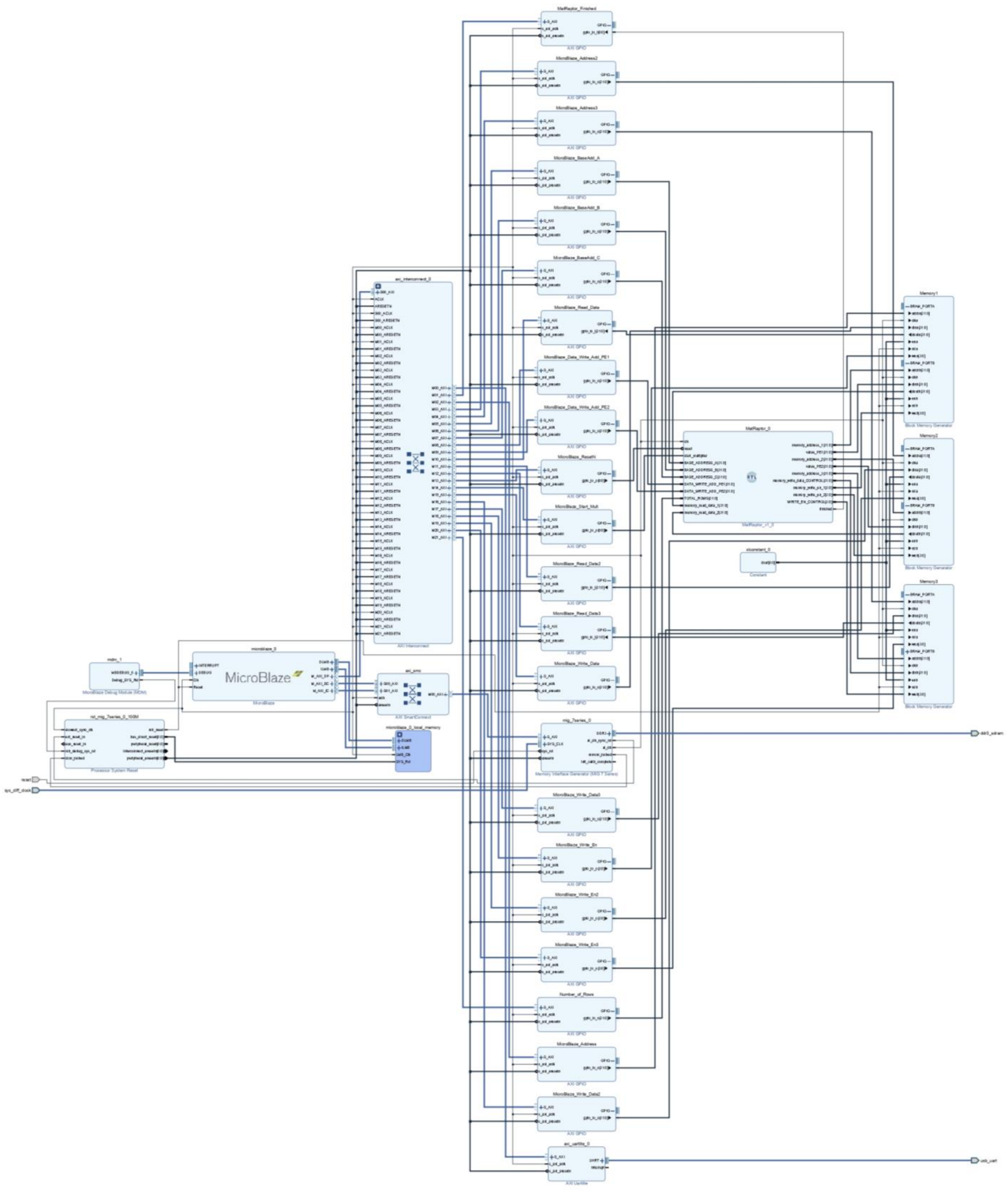


Fig. 7: **Block Design of the Integration of MicroBlaze and MatRaptor**