Optimization of Sparse Matrix Arithmetic Operations and Performance Improvement using FPGA

Dinesh Kumar Murthy
Ingram School of Engineering
Texas State University
San Marcos, TX, USA
d_m410 @txstate.edu

Semih Aslan
Ingram School of Engineering
Texas State University
San Marcos, TX, USA
aslan@txstate.edu

Abstract — **The increasing importance of sparse connectivity representing real-world data has been exemplified by recent work in the areas of graph analytics, machine language, and high-performance computing. Sparse matrices are the critical component in many scientific computing applications, where increasing sparse operation efficiency can contribute significantly to improving overall system efficiency. The main challenge lies in efficiently handling the nonzero values by storing them using a specific storage format and then performing matrix operations, taking advantage of the sparsity. This paper proposes an optimized algorithm for performing sparse matrix operations in storage and hardware implementation on Field-Programmable Gate Arrays (FPGAs). The results are obtained from implementing the sparse algorithm on hardware for matrices of different sizes. Sparsity percentages and sparsity patterns achieved low latency and high throughput compared with the standard algorithm. Further, the number of resources utilized was primarily reduced, enabling the FPGAs to focus on larger, more interesting problems.**

Keywords - Sparse matrix; latency; throughput; memory; FPGA; hardware architecture.

I. INTRODUCTION

We live in a "big data" era where graph processing has become increasingly important, because the amount of data generated and collected from many real-world applications such as sensors, social networks, portable devices. Graphs are used to model many systems of interest to engineers and scientists; today, useful information is being extracted. Once entered into a computer, the data no longer looks like a graph. Often, it is in the form of a sparsely populated matrix with mostly zeros compared to nonzeros [1] [2]. When the number of zeros is relatively large, efficient data structures are required. Numerous studies have addressed finding new algorithms for sparsely distributed matrices.

When obtaining information in a graph algorithm with a small number of nonzero entries but millions of rows and columns, memory would be wasted by storing redundant zeros [3][4]. There are two ways one would take advantage of the sparsity of a matrix: one would be to store the nonzero elements of a matrix, and the second is to process only the nonzero elements of a matrix [5]. However, large graphs are hard to deal with as inputs, and outputs limit the state-of-the-art graph processing systems. For the most part, Central Processing Units (CPUs) and Graphics Processing Units (GPUs) compute well on a performance scale. However, there is a small niche where an FPGA has been an attractive platform that can handle the same computation task for acceleration and achieve high performance with low power computation for many applications. Specifically, due to the memory access pattern of graph problems, it is still challenging to develop high throughput and energy-efficient FPGA design [6].

This paper's primary goal is to develop an efficient algorithm for various sparse matrix arithmetic operations like addition, subtraction, multiplication, element by element multiplication, and square root. By utilizing the sparse matrix storage method, storage requirements should be reduced when compared to a standard matrix operation algorithm. The main goal is to improve efficiency in terms of latency and throughput [7][8]. The performance analysis is calculated based on the design that minimizes gate count, area, and reducing the number of multipliers and adders. The architectural design is scalable, simple to implement, and capable of handling matrices of various sizes. This paper is organized as follows. In Section II, the basics of matrix operations are discussed. In Section III, the proposed algorithm and system design are explained. FPGA simulation and mapping are discussed in Sections IV and V, respectively. Sections VI and VII show the detailed performance analysis and the results. This paper concludes in Section VIII.

II. MATRIX OPERATION

The design performs sparse matrix addition operations of two sparse matrices where only the nonzero values are stored, and the required operation is performed. It is performed by using two algorithms:

- A symbolic algorithm, which determines the structure of the resulting matrix.
- A numerical algorithm, which determines the values of nonzero elements using the knowledge of their positions.

$$c_{i,j} = (a_{i,j}) + (b_{i,j}) \tag{1}$$

Each nonzero (nz) element of matrices **A** and **B** needs one floating-point operation, so the total number of floating-point operations to be performed is the number of nz elements. When the computation is completed, the number of nz output operations is written on the external memory.

III. System Design

A. Storage Format

The proposed architectural algorithm performs sparse matrix addition in which the number of rows and number of columns of two matrices is equal. A parallel implementation of the addition with enough fast memory algorithm, is proposed. Consider a matrix addition of **A**+**B**, where **A** has a density s_1 percentage with size n×n (a square matrix is considered), and matrix **B** has a density s_2 percentage with size n×n. Density s_x percentage is defined as the number of nonzero elements to the total number of elements in the matrix n^2. The matrix addition performs the operation row-wise and column-wise throughout the matrix only for the nonzero elements present, leaving behind the zeros. When an addition operation must be performed on both input matrices, the number of rows and columns are first compared to determine if they are equal, i.e., both the matrices should be of the same size. An additional operation cannot be performed if the matrices are of different sizes. Then, the matrix elements are checked row-wise and column-wise from top-to-bottom order for nonzero elements, as shown in Fig. 1. Two separate counters, *A_count* and *B_count*, are used to increment the row and column for both the **A** and **B** input matrices. This keeps incrementing from n to n+1 for the size of the matrix. The algorithm for the sparse matrix addition **A**+**B** is presented in Fig. 2.

The most important part of this algorithm is the index comparison, which is represented as *A_index* for matrix **A** and *B_index* for matrix **B**. After first storing the nonzero elements, the row value of matrix **A** is compared with the row value of matrix **B** for each operation. If the index of *A_sr* is equal to the index of *B_sr*, then the next step of comparing the column value of both matrices is performed. If the index of *A_sc* is equal to the index of *B_sc*, then a matrix addition operation is performed. The *VAL* array of the respective row and column, i.e., *A_sv* and *B_sv*, are added to each other as a sum. The assumption is made that the nonzero element is located anywhere in the matrix and is highly sparse. Finally, the nonzero element of input matrix **A** that does not match the row and column of matrix **B** is given directly as the sum in the output matrix.

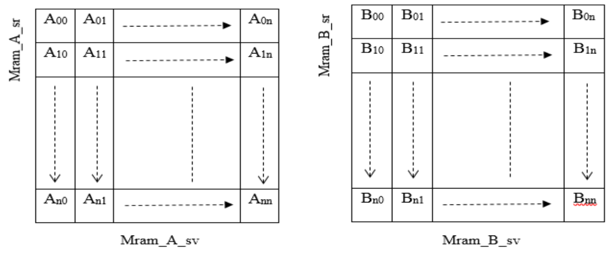


Figure 1. Representing row and column access of matrices

B. Design Algorithm

```
A → n×n sparse matrix
B → n×n sparse matrix
for i → 0 to MAT_SIZE do
        if (A[i]≠ 0) then
                Indexing row and column = i + 1
                A_sv [i] =A [i]
                A_index = A_count + 1
        end
        if (B[i] ≠ 0) then
                Index2rc = i + 1
                B_index = B_count + 1
                B_sv [ i] = B [ i]
        end
        if((A_sr[A_index]==B_sr[B_index])&&
        (A_sc[A_index] ==B_sc[B_index])) do
                Row <= A_sr [A_index]
                Col <= A_sc [A_index]
                Sum <= A_sv [A_index] + B_sv [B_index]
        end
        if (A_sv [A_index] ≠0) then
                Row <= A_sr [A_index]
                Col <= A_sc [A_index]
                Sum <= A_sv [A_index]
        end
        if (B_sv [B_index] ≠ 0) then
                Row <= B_sr[B_index]
                Col <= B_sc[B_index]
                Sum <= B_sv[B_index]
        end
end
```

Figure 2. Algorithm for Sparse Matrix Addition Operation

C. Memory Control

Memory control plays a crucial part in architectural design. The memory control block oversees real enable sign and assigning a memory access address, so accurate data is acquired by the algorithm logic through all stages. The operation is performed at the row level, so throughput is not affected by the latency of data reading while performing the arithmetic operation.

As shown in Fig. 3, the memory control module is designed as a finite state machine. At the beginning of the finite state machine, reset is set to Idle, which resets all the registers to predefined values. After this state, the matrix values are inferred for writing data to the Block RAMs (BRAMs), which triggers the memory control transition from the Idle State to the Read and Write state.

Once the elements are written, it calculates the nonzero values by checking row-wise and column-wise throughout the array by increasing the pointer locations by one. With the nonzero elements located successfully, separate arrays are created for matrix storage format in the order of *ROW*, *COL*, and *VAL*. As the name indicates, the row and column values are stored starting from 0 to the maximum, and the respective integer values are written accordingly. Once the sparse matrix storage format is generated, the arithmetic design algorithm checks the *ROW* and *COL* arrays and performs addition if both are equal. Otherwise, the design sends the values directly to the output, since addition is not required there. When the system performs all arithmetic operations, the finite state goes back to Idle State. By operating this way, only the nonzero elements undergo additional processes, and in the final state, the output is sent back.

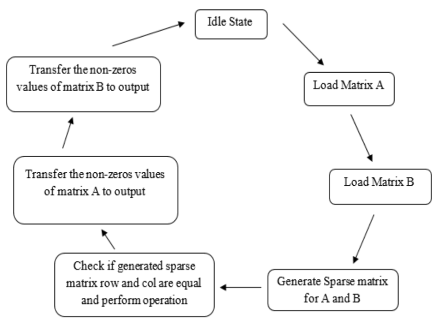
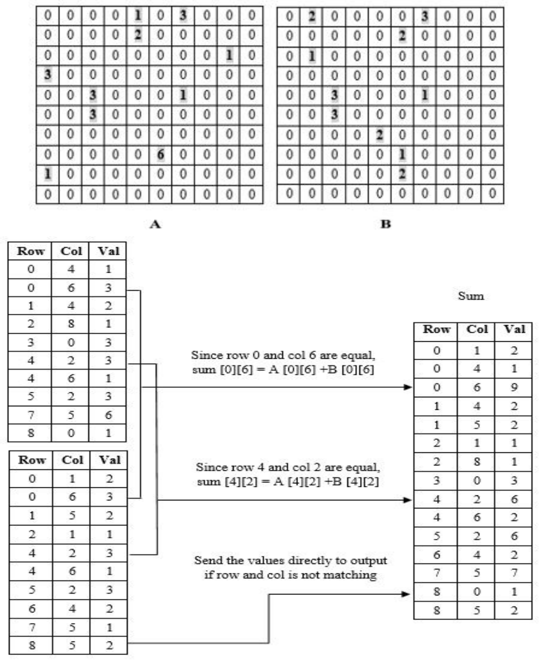

Idle State
Load Matrix A
Load Matrix B
Generate Sparse matrix for A and B
Check if generated sparse matrix row and col are equal and perform operation
Transfer the non-zeros values of matrix A to output
Transfer the non-zeros values of matrix B to output

Figure 3. State transition diagram of the memory control

For example, if there are two matrices **A** and **B** with ten nonzeros each, as shown in Fig. 4. The state machine will read the values and write the nonzero values in the storage format illustrated above. The necessary arithmetic operation is then performed from the Idle state, staying in hold for the state until it receives an end signal from the controller.



$$\begin{array}{cc}
\begin{bmatrix}
0&0&0&0&1&0&3&0&0&0\\
0&0&0&0&2&0&0&0&0&0\\
0&0&0&0&0&0&0&0&1&0\\
3&0&0&0&0&0&0&0&0&0\\
0&0&3&0&0&0&1&0&0&0\\
0&0&3&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&6&0&0&0&0\\
1&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0
\end{bmatrix} &
\begin{bmatrix}
0&2&0&0&0&0&3&0&0&0\\
0&0&0&0&0&2&0&0&0&0\\
0&1&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0\\
0&0&3&0&0&0&1&0&0&0\\
0&0&3&0&0&0&0&0&0&0\\
0&0&0&0&2&0&0&0&0&0\\
0&0&0&0&0&1&0&0&0&0\\
0&0&0&0&0&2&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0
\end{bmatrix} \\
\textbf{A} & \textbf{B}
\end{array}$$

Row	Col	Val
0	4	1
0	6	3
1	4	2
2	8	1
3	0	3
4	2	3
4	6	1
5	2	3
7	5	6
8	0	1

Row	Col	Val
0	1	2
0	6	3
1	5	2
2	1	1
4	2	3
4	6	1
5	2	3
6	4	2
7	5	1
8	5	2

Since row 0 and col 6 are equal, sum [0][6] = A [0][6] +B [0][6]

Since row 4 and col 2 are equal, sum [4][2] = A [4][2] +B [4][2]

Send the values directly to output if row and col is not matching

Sum

Row	Col	Val
0	1	2
0	4	1
0	6	9
1	4	2
1	5	2
2	1	1
2	8	1
3	0	3
4	2	6
4	6	2
5	2	6
6	4	2
7	5	7
8	0	1
8	5	2

Figure 4. Operational example for the addition of sparse matrices

IV. Simulation

Random matrices of various sizes are generated using MATLAB with variation in sparsity pattern and sparsity percentage. Additionally, two parameters, *MAT_SIZE* (size of the matrix n×n) and *ELEMENT_SIZE* (number of bits of the integer) are included with the design, which is passed to the input as known information.

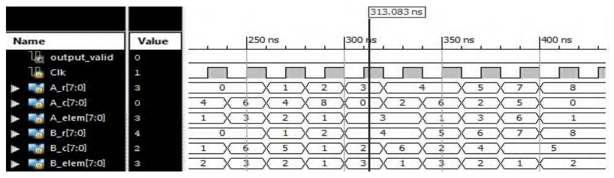


Figure 5. Waveform showing storage of sparse matrices

As shown in Fig. 5, the nonzero elements of the input matrices are stored to BRAMs in the format specified as two-dimensional arrays. The memory controller then reads the BRAMs to perform the required arithmetic operation.

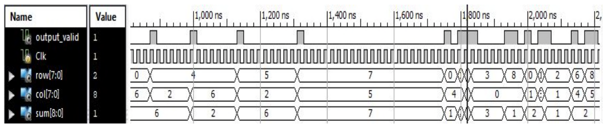


Figure 6. Waveform showing results of arithmetic operation (sum)

Fig. 6 shows the results of the addition operation in a simulation waveform. The algorithm is tested with multiple test values by varying the sparsity percentage and the golden result vectors generated using MATLAB.

V. FPGA Mapping

Using Xilinx ISE Design Suite, the designed algorithm is implemented on the target device Xilinx Artix7 XC7A100T-1CG324C board, comprising of 15,850 logic slices and a maximum of 4,860 Kbits fast BRAM [9] [10]. The hardware implementation is split into two major top modules. The first module is designed to implement the sparse matrix arithmetic operations, and the second module is to implement a Universal Asynchronous Receiver Transmitter (UART) communication and data exchange between the PC and FPGA. Each of the top modules is subdivided into smaller modules to carry out specific operations with the other modules through internal signals as shown in Fig. 7.

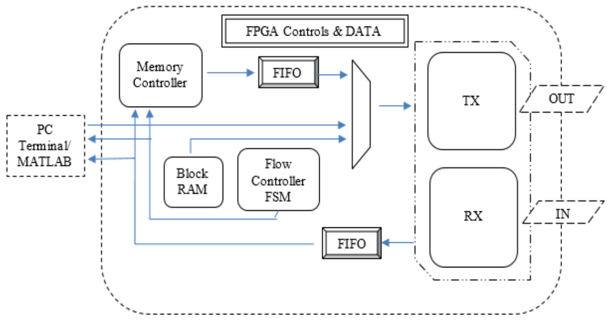


Figure 7. Block Diagram of the TX and RX Module

The transmitter module is used to transfer data over the UART device. It serializes a byte of data and transmits over a Transmit Data (TxD) line. The serialized data has 9600 Baud Rate, 8 data bits (least significant bit first), 1 Stop bit, and no parity. The receiver module double-registers the incoming data. This module makes sure all the bits are sent out. These modules expect the clock generated to be 100 MHz. The Phase-Locked Loop (PLL) is a control system that produces an output signal whose phase is related to an input signal. Keeping the input and output phases in lock steps, the input and output frequencies can be kept the same. These are widely used for synchronization purposes. For our hardware design, which operates at 20 MHz, the phase-locked loop is used to compensate for the required 100 MHz clock frequency. This IP core is generated using the design tool.

VI. Performance Analysis

The following metrics were calculated to show the algorithm's efficiency, such as latency, throughput, and resources utilized. Latency is the amount of time it takes to complete an operation, the time between reading the first element of the input matrix and writing the first element of the output matrix. Throughput is the number of such operations executed per unit of time.

The latency for matrix addition operation was significantly reduced, and high throughput was achieved using the proposed algorithm compared with the standard matrix algorithm. Table II illustrates the comparison of different test values with matrix sizes ranging from 10x10 to 100x100 with sparsity ranging from 1% to 10% for both proposed sparse and standard matrix algorithms for different operations.

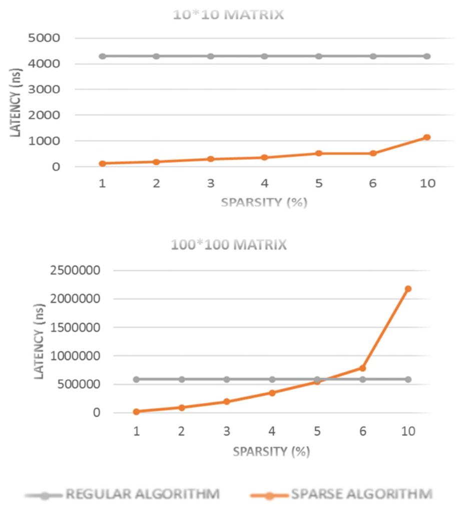


Figure 8. Latency for Sparse Matrix Addition

The comparison of latency calculated is plotted as a graph, which is shown in Fig. 8. The difference between the standard algorithm and the sparse algorithm is shown. Fig. 9. shows the difference in throughput between the two methods and shows that the proposed algorithm achieved high throughput.

After experimentation with different test values, there are improvements in latency and throughput for smaller matrices with high sparsity percentage and larger matrices with low sparsity percentage. Once the mapping of matrices is implemented on the FPGA platform, the resources utilized are shown in Table I.

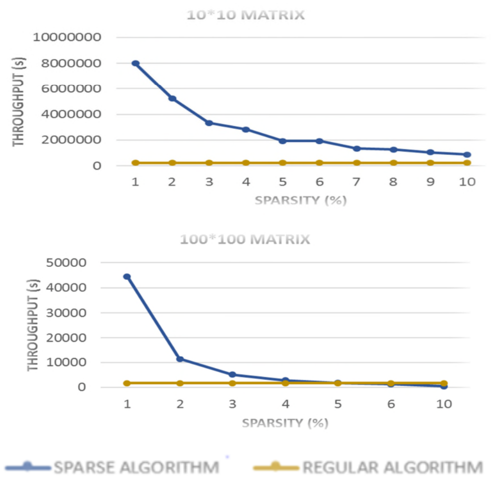


Figure 9. Latency for Sparse Matrix Addition

VII. Results and Discussion

In most cases, it is evident that latency and throughput are directly dependent on the number of nonzero elements present in the matrix. The efficiency of the design can be further improved by increasing the frequency of the overall design clock. The maximum speedup of the design for any matrix depends on the number of rows and columns being processed. One primary purpose of this paper is to reduce the storage space used in an FPGA when implemented. This is also accomplished when the design is implemented in an Artix 7 FPGA board. The amount of resources utilized for the proposed sparse algorithm is less than the standard algorithm. The comparison is tabulated in Table I. The design uses only 3 percent of the total FPGA resources. Further, pipelining can be implemented to increase the computational speed of the system. For arithmetic operations performed on large matrices or memory-based algorithms and for small matrices, a pipelined algorithm will be quite efficient.

VIII. Conclusion

Today's applications require higher computational throughput and a distributed memory approach for real-time applications. This research is primarily focused on designing an optimized architecture for sparse matrix operations, allowing for more efficiency than standard operations. The functionality of the design is verified by different sets of test cases under a specific size. The system contains a memory control which fetches the data from memory and passes it on for various arithmetic operations. Research improvement in this area is needed to increase logic resources by a comparable increase in I/O bandwidth and on-chip memory capacity, especially when the matrix sparsity is unstructured and randomly distributed.

TABLE I. DESIGN RESOURCE UTILIZATION SUMMARY

Slice Logic utilization		
Number of Slice Registers	4,799 out of 126,800	3%
Number of Slice Look-up Tables (LUTs)	6,702 out of 63,400	10%
Slice Logic Distribution		
Number of occupied Slices	2,413 out of 15,850	15%
Input/Output (IO) Utilization		
Number of bonded IO Blocks	3 out of 210	1%
Specific Feature Utilization		
Number of Block RAM/FIFO	2 out of 270	1%

REFERENCES

[1] X. Lin and J. Xu, "Special Issue on Graph Processing: Technique and Applications," Data Sci. Eng., vol. 2, no.1, p. 1, 2017.

[2] A.Ching, S. Edunov, M. Kabiljo, D. Logothetis,and ,S. Muthukrishnan, “One Trillion Edges : Graph Processing at Facebook-Scale," , Proceedings of the VLDB Endownment, vol. 8, no. 12, pp. 1804-1815, 2015.

[3] M. Ryan, "FPGA Hardware Accelerators - Case Study on Design Methodologies and Trade-Offs", 2013. Thesis. Rochester Institute of Technology. Accessed from http://scholarworks.rit.edu/theses/959.

[4] T. Mattson et al., "Standards for graph algorithm primitives," IEEE High Performance Extreme Computing Conference (HPEC), Waltham, MA, USA, 2013, pp. 1-2, doi: 10.1109/HPEC.2013.6670338.

[5] S. Jain, N. Kumar, J. Singh, and M. Tiwari, “FPGA Implementation of Latency, Computational time Improvements in Matrix Multiplication,” International Journal of Computer Applications, 2014, vol.86, no.8, doi:10.5120/15007-3261.

[6] S. Aslan and J. Saniie, "Matrix Operations Design Tool for FPGA and VLSI Systems," 2016, Circuits and Systems, vol. 7, no.2, pp. 43–50, doi: 10.4236/cs.2016.72005.

[7] P. Grigoras, P. Burovskiy, E. Hung and W. Luk, "Accelerating SpMV on FPGAs by Compressing Nonzero Values," 2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines, Vancouver, BC, Canada, 2015, pp. 64-67, doi: 10.1109/FCCM.2015.30.

[8] L. Zhuo and V. Prasanna, “Sparse Matrix-Vector multiplication on FPGAs,” In Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays *(FPGA '05).* ACM, New York, NY, USA, 63-74.

[9] B. Hamraz, N. Caldwell, and P. Clarkson "A Matrix-Calculation-Based Algorithm for Numerical Change Analysis”, IEEE Transaction on Engineering Management, Vol.60, No.1 February 2013.

[10] Nexys 4 DDR board – Reference Manual.

TABLE II. LATENCY AND THROUGHPUT CALCULATION

Matrix Size (n*n)	**Number of nonzero (nnz)**	**Sparsity**	**Sparse Algorithm**		**Standard Algorithm**	
			Latency (ns)	**Throughput**	**Latency (ns)**	**Throughput**
Matrix Addition						
1010	10	0.1	1137.169	879376.7681	4298.0515	232663.5688
20x20	32	0.08	8550.567	116951.3086	18688.1835	53509.74855
40x40	96	0.06	57464.964	17401.90771	93787.3685	10662.41666
60x60	144	0.04	123981.857	8065.696257	214929.95	4652.678698
100x100	100	0.01	22427.802	44587.51687	588369.806	1699.61135
Matrix Subtraction						
1010	9	0.09	911.1375	1097529.187	3205.4335	311970.2842
20x20	28	0.07	6156.6615	162425.6913	19199.5965	52084.42792
40x40	80	0.05	43638.191	22915.70702	38884.05	25717.4857
60x60	108	0.03	84721.1265	11803.43134	214411.526	4663.928375
100x100	100	0.01	70001.865	14285.33368	589749.829	1695.634235
Matrix Multiplication (Element-by-Element)						
1010	10	0.1	1107.282	903112.3056	3205.4335	311970.2842
20x20	36	0.09	9780.263	105482.3057	19199.5965	52084.42792
40x40	80	0.05	42519.648	23518.53901	38884.05	25717.4857
60x60	72	0.02	32073.866	31178.03136	214411.526	4663.9283
100x100	100	0.01	5152.62	18364.9265	589749.82	1695.6342