OPTIMIZED SPARSE MATRIX OPERATIONS AND HARDWARE IMPLEMENTATION USING FPGA

by

Dinesh Kumar Murthy, B.E.

A thesis submitted to the Graduate Council of
Texas State University in partial fulfillment
of the requirements for the degree of
Master of Science
with a Major in Engineering
August 2021

DEDICATION

I would like to dedicate this thesis to my loving dad and mom for all the guidance, encouragement, and support throughout my life. I also dedicate this thesis to my friends for their constant help and support in every step of my life.

ACKNOWLEDGEMENTS

I would like to express my gratitude to all who helped me during the writing of this thesis at Texas State.

First, I would like to express my deep gratitude to Dr. Semih Aslan, my supervisor, for his continuous support and encouragement, for his patience, motivation, enthusiasm, and immense knowledge. He also provides me with an excellent atmosphere for conducting this research project. Without his consistent and illuminating instruction, this thesis could not have reached its present form.

I would also like to express my heartfelt gratitude to my thesis committee members: Dr. Dan Tamir, Dr. Bill Stapleton and Dr. Jesus Jimenez for their insightful comments and constructive comments in the early and final version of the work.

I owe a special debt of gratitude to Dr. Vishu Viswanathan, Graduate Advisor of Engineering, and Dr. Stan McClellan, Former Director of Ingram School of Engineering, for providing facility support. I would like to thank Ms. Sarah Rivas in Ingram School of Engineering and Department Engineering Technology for their kind help.

Finally, I express my gratitude to my parents, family, and friends for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them.

TABLE OF CONTENTS

Page

ACKNOWLEDGEMENTS....v

LIST OF TABLES....viii

LIST OF FIGURES....ix

LIST OF ABBREVIATIONS....xiv

ABSTRACT....xvi

CHAPTER

1. INTRODUCTION....1

    1.1 Problem Statement....4
    1.2 Research Goals....4
    1.3 Goal Measurement Metrics....5
    1.4 Tools Used....7

2. LITERATURE REVIEW....8

3. BACKGROUND....12

    3.1 Graph Processing....12
    3.2 Sparse Matrices....14
    3.3 Sparse Matrix Storage Formats....16
    3.4 Finite State Machines (FSMs)....20
    3.5 Field Programmable Gate Arrays (FPGAs)....21
        3.5.1 FPGA Architecture....21
        3.5.2 Design and Programming....22
            3.5.2.1 Design Entry....23
            3.5.2.2 Behavioral Simulation....23
            3.5.2.3 RTL Analysis....24
            3.5.2.4 Synthesis....25
            3.5.2.5 Implementation....26
            3.5.2.6 Timing Analysis....27
            3.5.2.7 Power Usage Analysis....28

3.5.2.8 Programming the Board......................................29
3.5.3 NEXYS4 DDR ARTIX-7 FPGA Board.................................29

4. HARDWARE DESIGN ARCHITECTURE..31

4.1 Arithmetic Operations..38
4.1.1 Sparse Matrix Addition 38
4.1.2 Sparse Matrix Multiplication..48
4.2 Matrix Decomposition 58
4.2.1 LU Decomposition..59
4.2.1.1 Sparse LU Decomposition Architecture.............62
4.2.1.2 Implementation and Error Analysis 67
4.2.2 QR Decomposition..71
4.2.2.1 Householder Transformation...............................73
4.2.2.2 Design Flow and Optimization............................76

5. ALGORITHM PERFORMANCE RESULTS..85

5.1 Sparse Matrix Addition vs. Regular Matrix Addition...........................85
5.2 Sparse Matrix Subtraction vs. Regular Matrix Subtraction 98
5.3 Element-By-Element Multiplication 101
5.4 Sparse Matrix Multiplication vs. Regular Matrix Multiplication........104
5.5 Sparse LU Decomposition vs. Regular LU Decomposition................110
5.6 Sparse QR Decomposition vs. Regular QR Decomposition 114
5.7 Execution Time Analysis 117

6. CONCLUSION AND FUTURE WORK 119

APPENDIX SECTION...121

REFERENCES 146

LIST OF TABLES

Table **Page**

1. Adjacency Matrix Representation of the Graph 13
2. Comparison of Various Sparse Matrix Storage Formats 17
3. Computational Complexity 32
4. Error Analysis for LU Decomposition Operation 70
5. Error Analysis for QR Decomposition Operation 84
6. Hardware Implementation: Sparse Matrix Arithmetic Operation 97
7. Latency and Throughput Comparison from Implemented Design 98
8. Resources Utilization for Sparse Matrix Element by Element Multiplication 104
9. Hardware Implementation: Sparse Matrix Multiplication 108
10. Hardware Implementation: LU Decomposition 113
11. Hardware Implementation: QR Decomposition 115
12. Benchmark matrices, performance, and pattern 117

LIST OF FIGURES

Figure **Page**

1. Real World Sparse Matrix Applications 1
2. Graph Representing Several Cities in Southern California 12
3. Sparsity Pattern of Matrices 15
4. Storage of Input Matrix 19
5. State Diagram of a Simplified Finite State Machine 20
6. Sketch of FPGA Architecture and Design of Simple Logic Block 22
7. Vivado Design Suite 22
8. Vivado Project Manager 23
9. Vivado Simulation Environment 24
10. Vivado RTL Analysis Tool 24
11. Vivado Synthesis Analysis Tool 25
12. Vivado Synthesis Report Analysis 25
13. Vivado Implementation Tool 26
14. Vivado Timing Constraints Tool 27
15. Vivado Power Analysis Tool 28
16. Nexys4 DDR Artix-7 FPGA Board 30
17. High Level Flow Chart of Work Proposed 31
18. FPGA Implementation and Verification Flow 32
19. Overview of Hardware and Software Implementation 33

20. Block Diagram of TX and RX Module..34

21. Clock Cycle for TX and RX 35

22. Fundamental Design: Transmission FSM...36

23. Fundamental Design: Receiver FSM 37

24. Representation of Row and Column Access of Input Matrices 40

25. FSM Transition States for Sparse Matrix Arithmetic Operations...............................42

26. Sparse Matrix Addition Operation Methodology 43

27. Design Simulation: Sparse Matrix Addition Operation..44

28. Implemented Design: Arithmetic Operation Engine...46

29. Interconnect Design: Input Valid Schematic 47

30. Interconnect Design: Output Valid Schematic..47

31. Matrix Multiplication Hardware Architecture 49

32. Overall Design Flow 50

33. Sparse Matrix Multiplication Methodology...53

34. Design Simulation: Sparse Matrix Multiplication 54

35. Elaborated Implemented Design: Sparse Matrix Multiplication 55

36. Interconnect Design: Sparse Matrix Multiplication..56

37. Interconnect Design: Temporary Registers for Intermediate Output..........................57

38. Matrix Decomposition 58

39. MATLAB Representation of LU Decomposition of 20x20 Combinatorial
Problem from Suite Sparse Matrix Collection...61

40. Control Logic for LU Decomposition Hardware 62

41. Block Diagram of Proposed LU Decomposition Hardware 65

42. Design Simulation: LU Decomposition 68

43. Implemented Design Engine: LU Decomposition 69

44. MATLAB Representation of QR Decomposition of 100x100 Structural Problem from Suite Sparse Matrix Collection 72

45. QR Decomposition Core 76

46. Usage of Registers for QR Decomposition of 2x2 Matrix 77

47. Block Diagram for QR Decomposition using Householder Transformation 79

48.Simulation Waveform: QR Decomposition 80

49. Implemented Design: QR Decomposition 81

50. Interconnect Schematic: Output FSM 82

51. Interconnect Schematic: Square Root Module 83

52. Latency Comparison: Proposed vs Regular for 10×10 Matrix 86

53. Latency Comparison: Proposed vs Regular for 20×20 Matrix 86

54. Latency Comparison: Proposed vs Regular for 30×30 Matrix 87

55. Latency Comparison: Proposed vs Regular for 40×40 Matrix 87

56. Latency Comparison: Proposed vs Regular for 50×50 Matrix 88

57. Latency Comparison: Proposed vs Regular for 100×100 Matrix 89

58. Throughput Comparison: Proposed vs Regular for 10×10 Matrix 90

59. Throughput Comparison: Proposed vs Regular for 20×20 Matrix 91

60. Throughput Comparison: Proposed vs Regular for 30×30 Matrix 91

61. Throughput Comparison: Proposed vs Regular for 40×40 Matrix 92

62. Throughput Comparison: Proposed vs Regular for 50×50 Matrix 93

63. Throughput Comparison: Proposed vs Regular for 100×100 Matrix 93

64. Increase in Throughput for Different Matrix Dimensions with 1% Sparsity 94

65. Increase in Throughput for Different Matrix Dimensions with 5% Sparsity 95

66. Increase in Throughput for Different Matrix Dimensions with 10% Sparsity 95

67. FPGA Resource Utilization: Regular vs Sparse-Sparse Matrix Addition Operation ..96

68. Power Analysis: Sparse Matrix Arithmetic Operation 97

69. FPGA Resource Utilization: Regular vs Sparse-Sparse Matrix Subtraction
Operation...100

70. Latency Comparison: Proposed vs Regular for Sparse Matrix
Element-by-Element Multiplication Operation 101

71. Increase in Throughput for Different Matrix Dimensions with 1% Sparsity 102

72. Increase in Throughput for Different Matrix Dimensions with 5% Sparsity 102

73. Increase in Throughput for Different Matrix Dimensions with 10% Sparsity 103

74. Throughput Comparison: Sparse Matrix Multiplication Operation...........................105

75. Increase in Throughput for Different Matrix Dimensions with 1% Sparsity 106

76. Increase in Throughput for Different Matrix Dimensions with 5% Sparsity 106

77. Increase in Throughput for Different Matrix Dimensions with 10% Sparsity 107

78. FPGA Resource Utilization: Regular vs Sparse-Sparse Matrix Multiplication
Operation...108

79. Power Analysis: Sparse Matrix Multiplication Operation..109

80. Latency Comparison: Run01 vs Run02 110

81. Throughput Comparison: Run01 vs Run02 111

82. FPGA Resource Utilization: Regular vs Sparse-Sparse Matrix LU
Decomposition..112

83. Power Analysis: Sparse Matrix LU Decomposition...113

84. Latency Comparison: Run01 vs Run02 114

85. Throughput Comparison: Run01 vs Run02 115

86. FPGA Resource Utilization: Regular vs Sparse-Sparse Matrix QR
Decomposition..116

87. Comparison of execution time between MATLAB and FPGA................................117

LIST OF ABBREVIATIONS

ABBREVIATION	DESCRIPTION
FPGA	Field Programmable Gate Arrays
N_{lz}	Length of the largest non-zero diagonal
N_{dia}	Number of non-zero diagonal offsets
N_{mnzr}	Maximum number of non-zero values
NZV	Number of non-zero values in the matrix
r	Represents block size
$N_{mnzr}g(i)$	Maximum number of non-zero elements per row
G_{size}	Number of rows clumped together to form a group
N_g	Number of groups
N_q	Number of quadrants
$CSR_{storage}$	Storage space required for CSR
N	Order of the matrix
N_s	Number of streams
NZS	Number of non-zero values per stream
S_r	Number of rows per stream
N_{nzb}	Number of non-zero blocks in the matrix
b_{size}	Block size
N_S	Number of slices

N_{nzv}	Maximum number of nonzero elements
FSM	Finite State Machines
MAT_SIZE	Size of the Matrix (n×n)
ELEMENT_SIZE	Number of Bit (8-bit, 16-bit, 32-bit)
NZE	Maximum Number of non-zeros
A_sr	Row of sparse matrix A
B_sr	Row of sparse matrix B
A_sc	Column of sparse matrix A
B_sc	Column of sparse matrix B
A_sv	Value of sparse matrix A
B_sv	Value of sparse matrix B
A_count	Counter to increase index values of matrix A
B_count	Counter to increase index values of matrix B
A_index	Index of matrix A
B_index	Index of matrix B

ABSTRACT

The increasing importance of sparse connectivity representing real-world data has been exemplified by the recent work in areas of graph analytics, machine language, and high-performance. Sparse matrices are the critical component in many scientific computing applications, where increasing the sparse matrix operation efficiency can contribute significantly to improve overall system efficiency. The primary challenge is handling the nonzero values efficiently by storing them using specific storage format and performing matrix operations, taking advantage of the sparsity. This thesis proposes an optimized algorithm for performing sparse matrix operations concerning storage and hardware implementation on FPGAs. The proposed thesis work includes simple arithmetic operations to complex decomposition algorithms using Verilog design. Operations of the sparse matrix are tested with testbench matrices of different size, sparsity percentage, and sparsity pattern. The design was able to achieve low latency, high throughput, and minimal resources utilization when compared with the conventional matrix algorithm. Our approach enables solving more significant problems than previously possible, allowing FPGAs to more interesting issues.

1. INTRODUCTION

We live in a "big data" era where Graph Processing has become increasingly important with the amount of data volume generated and collected from many real-world applications such as sensors, social networks, portable devices. As you expect, graphs can sometimes be very complicated. With the demand for data-analysis continuing to grow, large-scale graph processing has become challenging.

A graph consists of a finite set of vertices and a set of edges composed of distinct, unordered pairs of vertices. A dot represents the vertex, and an edge represents a line segment connecting the dots associated with the edge. If one vertex is directed to another vertex by the edges of a graph, then the graph is called directed graph. If it is undirected, then the graph is called undirected graph.

Graph-based applications are used to represent physical structures from social network analyses to anomaly detections. Computing these graphs are entirely determined by specifying either its adjacency structure or its incidence structure. As computers are more adept at manipulating numbers than at recognizing pictures, it is a standard practice to communicate the graph specifications to a computer in matrix form.

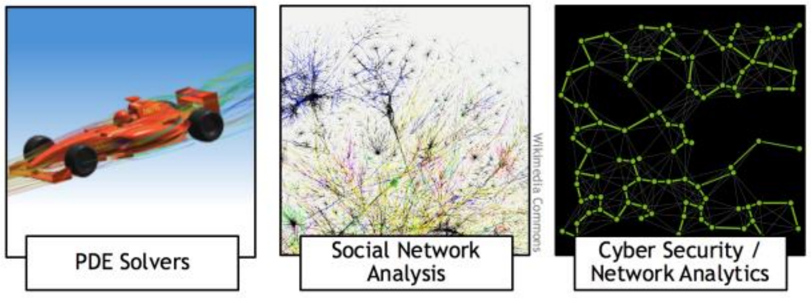


Figure 1. Real World Sparse Matrix Applications

Graphs are used to model many systems which are of interest to engineers and scientists today, through which useful information is extracted. Once entered a computer, the data from real-world applications no longer looks like a graph. Often it is in the form of a sparsely populated matrix with most non-zeros compared to zeros [1]. When the number of zeroes is relatively large, a requirement for more efficient data structures arises. We are drifting away from serial computing towards parallel distributed computing over a large variety of architectural designs. The generic implementation of data structures allows one to reuse the most appealing one, which may not be the fastest.

In a graph algorithm, to obtain information where there is a small number of nonzero entries, but millions of rows and columns of memory could be wasted by storing redundant zeros. There are two ways one would exploit the sparsity of a matrix: One, to save the non-zero elements of a matrix and second is to process only the non-zero elements of a matrix[2]. However, large graphs are hard to deal with as IO limits the state-of-art graph processing systems.

Numerous studies have been addressed to specialize in finding new algorithms for the sparsely distributed matrices. Running parallelized programs on GPU gives large speedup, however high-performance GPUs consume considerable amount of power. Many computations are difficult to parallelize, incurring extra overhead for transferring data between CPU and GPU. For the most-part, CPUs and GPUs compute well in performance scale. FPGA based designs may avert those problems due to low power nature, with efficient customized pipelines. In a comparison of performance of FPGA and GPU, it is reported both have similar performance, while FPGAs can be 15 times faster and 61 times energy efficient than GPU for uniform random generation [3]. Another work shows GPU implementation can be 11 times faster than FPGA, but on the contrary FPGA implementation can be up to 15 times better than GPU in terms of performance per watt [4]. It is feasible to get a latency of around or below 1 microsecond using FPGA, whereas with CPU a latency of 50 microsecond is already good. Moreover, one of the main reasons for low latency is they do not depend on generic operating system and the communication do not have to go via generic buses such as USB or PCIe. GPUs multi-dimensional threading structure or multi core platform is not strongly suitable for highly data-dependent transformations for matrix decomposition. The performance improvement of GPU and multi-core platform-based decomposition algorithms is limited due to the iterative thread sync and irregular memory access [5]. While GPUs shows satisfactory compute efficiency on sparse matrix-matrix operations, they have showed that compute units are significantly underutilized when the sparsity drops below 0.1% achieving low throughput [6]. However, there is a small niche, where FPGA has been an attractive platform which can handle the same computation task for acceleration and achieve high performance with low power computation for many applications. Previous implementation of FPGA based performance improvement for many applications like linear algebra, graphic computation was demonstrated. Compared to other parallel platforms, FPGAs are a better solution for performance improvement by parallelizing decomposition algorithms with flexibility, reconfigurability and low energy consumption.

The primary focus of this work is divided into four subdivisions: Matrix Operations, Storage Format, software implementation and finally hardware platform. After carefully reviewing all the previous methods of approaching the sparse matrices, the next reliable step for improving the performance no longer involves proposing new expensive optimization but applying the optimizations whenever they are useful.

The primary goal of this project is to develop an efficient algorithm for various sparse matrix operations and compare with the regular matrix operations. By utilizing the sparse matrix storage method, the storage requirements for storing were significantly reduced to be processed in a single FPGA. Finally, the matrix values are sent as input to the FPGA board, performing the necessary matrix operations, and the output values are sent back for verification. The performance calculations are carried out and are represented as individual graphs for comparison.

1.1 Problem Statement

1. Indirect addressing**:** Indirect addresses must address the non-zero entries of a sparse matrix in its index array leading to random accesses that require more memory transactions and lower cache hit rate.
2. Memory Allocation: The distribution of zero and non-zero entries are not known in advance. Pre-allocating memory blocks of a specific size may waste memory when the intersection of nodes is large.
3. Low Arithmetic Intensity: This is caused by the lack of temporal locality in the access to sparse matrices. If the matrix is not structured or blocked, most of the entries in cache line fetched to get an element remain unused causing high memory overhead per sparse matrix operation.

1.2 Research Goals

1. To determine an algorithm for various sparse matrix operations by minimizing gate count, area, computational time, latency, number of multiplication & addition hardware and to improve throughput.

2. The developed algorithm must be capable of handling matrices of various sizes and should be simple to implement and highly scalable.

   The implementation of the algorithm on an FPGA board involves the following steps.

- Design of an arithmetic logic unit in Verilog. This unit should implement the Sparse matrix algorithm for arithmetic operations like addition, subtraction, multiplication, as well as decomposition methods including LU and QR decomposition.
- Implement the design of sparse matrix algorithms and optimize for the problem size concerning area, speed, and latency.
- Design of a Universal Asynchronous Receiver/Transmitter (UART) communication module in Verilog for transferring the data from PC/UART port for sparse matrix algorithm computation. Results are verified with MATLAB results for error analysis.
- Comparison of the results and investigate the possible solutions and approaches for scaling up the design for larger matrix more efficiently.

1.3 Goal Measurement Metrics

The two basic hardware design methodologies include language-based design using synthesis tool and schematic-based design. Synthesis tools continue to improve more optimized methods in terms of both area and speed when compared with schematic implementation. The schematic-based design is no longer feasible for supporting architectural complexity for modern FPGAs. Also, this research focuses on reducing the number of multipliers and adders to provide improvements in performance.

For the computation of two sparse matrix operation, there are a certain number of arithmetic operations regardless of the storage of the matrix which include multiplications and addition of nonzero values. The primary goal of the thesis is to improve efficiency and reduce the resources used for the operation. The performance analysis is calculated in terms of improvements in latency, computation time, throughput for performing matrix operations and which reduces the number of multiplication and additions hardware utilize.

- **Latency**

It is the amount of time for completing an operation. This is defined as the time between reading the first element of the input matrices, A and B, and writing the first element to the result matrix C. The Latency of an operation is calculated based on the number of clocks consumed by the Hardware accelerator to produce an output after the application of the input (i.e.) the time from reading the first element of input matrix and writing the first element to output matrix.

$$Total\ Time = T_{out} - T_{in}$$

where T_{out} *– time taken for the last output to be calculated;* T_{in} *– time taken for the first input to become available*

$$\text{Total Number of Clock Cycles } (T_c) = \frac{Total\ Time}{Clock\_Period}$$

Clock_Period – timing constraint

$$Time\ for\ one\ matrix\ output\ (T_m) = T_c \times T_{min}$$

where T_{min} *– Minimum period of the clock for the design during synthesis;*

$$Time\ for\ n\ matrix\ output\ (T_n) = T_m \times \text{n} \times \text{n}$$

where *n×n – matrix size, with n=10,20,30, …100*

• **Throughput**

Throughput represents the rate at which the design can process inputs. It is the number of operations executed or produced per unit of time. Through this thesis, 1sec (1000ms) is considered as the unit of time, thus representing the throughput as elements/sec. As latency is defined by the time consumed by the design to produce one element, the throughput over a time interval of one sec can be derived as follow:

$$Throughput(elements/sec) = \frac{1\,sec}{T_n}$$

where *n×n – matrix size, with n = 10,20,30, … 100.*

• **Resource Utilization**

The amount of resource available on the FPGA board such as Lookup Tables, Memory, logic registers, BRAM, flipflops, nets and logic interconnects are valuable. A comparison of resources utilized for the proposed method and the regular method is presented for analysis.

• **Power Analysis**

The power utilization report provides the static and dynamic power consumption of the implemented design, for which the comparison is provided for data analytics.

1.4 Tools Used

The following tools will be used to carry out synthesis, implementation, and verification of results:

1. Digilent Nexys4 DDR FPGA.
2. Xilinx ISE Design Tool.
3. Vivado Design Suite.
4. MATLAB Software

2. LITERATURE REVIEW

The Basic Linear Algebra Subprograms (BLAS) has been used in a wide range of software, which provides basic building block routines for vector and matrix operations [7]. Some of the optimizations for BLAS library on general-purpose processors includes loop unrolling and register blocking. Because many of the optimizations are specific to a platform, ATLAS was implemented which automatically optimizes the numerical software for processors with pipelined designs. Linear Algebra Library (LAPACK) solves system of linear equations, least-square solutions, eigenvalue problems and singular value problems.

The main reason why FPGAs are considered over traditional computer is they can be configured as required by the application. The FPGAs can be reprogrammed to given hardware acceleration which offers the best of both hardware and software. And most importantly they are becoming extremely inexpensive when compared with super-computers like such as CRAY with millions on logic gates and LUT.

There have been several works done for the acceleration of sparse matrix operations that uses Multicore processor, GPU and FPGA based approaches. The implementation using CPU keeps all the data associated with the operation in cache, while the GPUs largely focus on the efficient memory bandwidth usage, whereas FPGA focus on compressed storage of matrix data to reduce the memory bandwidth requirements. Recently, FPGA implementation have been greatly used in data centers like in, researchers from Microsoft uses an FPGA-based design for accelerating the “Bing” search engine [8].

Most of the studies target Sparse Matrix by Vector Multiplication (SpMV), yet Sparse Matrix by Matrix multiplication (SpMM) has been rarely addressed in prior research. A detailed literature by explains the optimization techniques in sparse matrix multiplication. In [9] Zhuo had proposed an FPGA based design, which demonstrated significant performance improvement over general-purpose processor for matrices with irregular sparsity structures. There was another implementation for FPGA based SpMM using a single FPGA node showing how sparsity of a single matrix is affecting the performance of the operations [10]. In [11] the authors have proposed separate architecture for matrix multiplication, where operation speed is a main issue. The pipelining and parallel processing of elements were used to decrease the computation time in [12]. The former method has considered area and latency, while the second had taken area and maximum running frequency considering the energy dissipation.

Some of the works on efficient sparse LU Decomposition architecture for sparse matrices are either Target Domain-Specific pattern targeting a specific application domain or require pre-ordered symmetric matrix. Only a few FPGA-based architectural designs for Sparse LU Decomposition have been proposed due to:

a. These sparse matrices have irregular sparsity structure, and it is difficult to devise an efficient and common hardware design for all application domains.
b. A detailed study on the nonzero structure of the sparse matrix is to be performed for designing suitable input parameters for the hardware design.

Consider the work by [13], where author proposes an efficient LU Decomposition hardware Architecture targeting the Power Flow Analysis Application Domain implementing right-looking algorithm along with mechanisms for pivoting operations.

But the performance of the work is primarily I/O bandwidth limited. Whereas in [14], the work is primarily dependent for Circuit Simulation Application domain. The author proposed in a matrix factorization graph which is generated to capture the static sparsity pattern of the matrices and is exploited for distributing the explicit data flow representation of computation across PE’s. In the work on [15], a more general hardware design for sparse LU Decomposition was proposed for a wider range of application domains. The hardware architecture parallelizes Left-looking Algorithm to efficiently decompose position symmetric positive definite or diagonally dominant matrices. This design is indeed efficient except for the fact, when the performance of the design arises from dynamic data dependency during column-by-column factorization leading their processing elements stalling for synchronizing to resolve data dependency. Also, the matrices used as benchmarks are either semi-dense or symmetric in topology but none in terms of nonzero elements. The hardware utilization of some of the previous implementations on reconfigurable architectures including Multicores, GPU and FPGA never exceeded 20% mark. The main reason for this poor performance is the irregularity of computation and memory access. The hardware resource utilization of sparse algorithm is very high, because of large hardware dynamic scheduling which is limited by scalability.

The previous FPGA implementations adopts dynamic dataflow, incurring in large overhead and poor hardware resource utilization. The proposed algorithm in this thesis introduces a synchronous dataflow FPGA implementation addressing the main problems of Sparse Operations. A customized data storage format is employed to organize memory access to eliminate time-consuming data address calculations. One of the limiting factors is the time required for pivot search. Reducing the pivot search during LU decomposition of eliminating will lead to higher performance gain. In our work, we had improvement by overlapping the next pivot search with the current update unit, which although depends on reuse of rows from an elimination step.

The first QR Decomposition was used in weight computation for adaptive beamforming application [16]. In [17] a Squared Givens Rotation algorithm was used to avoid the square root operation. The was followed up by the work of [18] which used SGR algorithm for implementing a linear array on Xilinx Virtex-E FPGA allowing a maximum of 9 processors and achieving 150MHz clock rate and throughput of 20GFLOPS with floating point operation. The first implementation of linear array architecture using CORDIC algorithm for rotation computing was developed. There are many commercial QR-D IP cores using CORDIC algorithm. An algorithm for Inverse QR-based decomposition was proposed by S. Thomas Alexander and Avinash L. Ghirnikar [19] which was later applied to adaptive beam forming. A fixed-point QR decomposition was developed with modified Gram-Schmidt (MGS) algorithm using LUT based approach. Later for polynomial matrices, Polynomial Givens rotation [20] was developed.

3. BACKGROUND

3.1. Graph Processing

Graphs are a collection of nodes and edges. The edge of the graph provides a connection between one node to another. By default, an edge is bidirectional. Typically, graphs are used to model collection of things along with their relationships. For example, Figure 2 shows a graph with cities as nodes and roads connecting them as edges. The graph represents several cities in Southern California. Vertices represent the cities in the graph while the fact that an edge connecting two vertices show two of the cities connected.

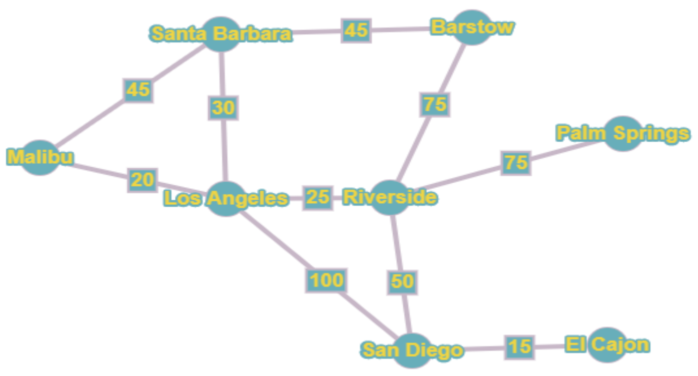


Figure 2. Graph Representing Several Cities in Southern California

Since the structure of real-world graphs can vary tremendously, there is a need for an efficient algorithm for obtaining high performance [21]. When these graphs are processed in a computer, they get stored in the form of an adjacency matrix. For a graph with n nodes, an adjacency matrix is represented as an n×n two-dimensional array. For a weighted graph, the array elements would give the cost of the edge between them, and for an unweighted graph, the array would be Booleans. The following Table 1 is an example of an adjacency matrix representation of the graph in the table. The able in Figure 1 shows the graph is represented as a sparsely populated matrix. The number of rows and columns is equal to the number of vertices in the graph. The edge is represented by intersecting rows and columns of two vertices it connects.

Table 1. Adjacency Matrix Representation of the Graph

	MALIBU	SANTA BARBARA	LOS ANGELES	RIVERSIDE	BARSTOW	PALM SPRINGS	SAN DIEGO	EL CAJON
MALIBU	0	45	20	0	0	0	0	0
SANTA BARBARA	45	0	30	0	45	0	0	0
LOS ANGELES	20	30	0	25	0	0	100	0
RIVERSIDE	0	0	25	0	75	0	0	0
BARSTOW	0	45	0	75	0	0	0	0
PALM SPRINGS	0	0	0	75	0	0	0	0
SAN DIEGO	0	0	100	50	0	0	0	15
EL CAJON	0	0	0	0	0	0	15	0

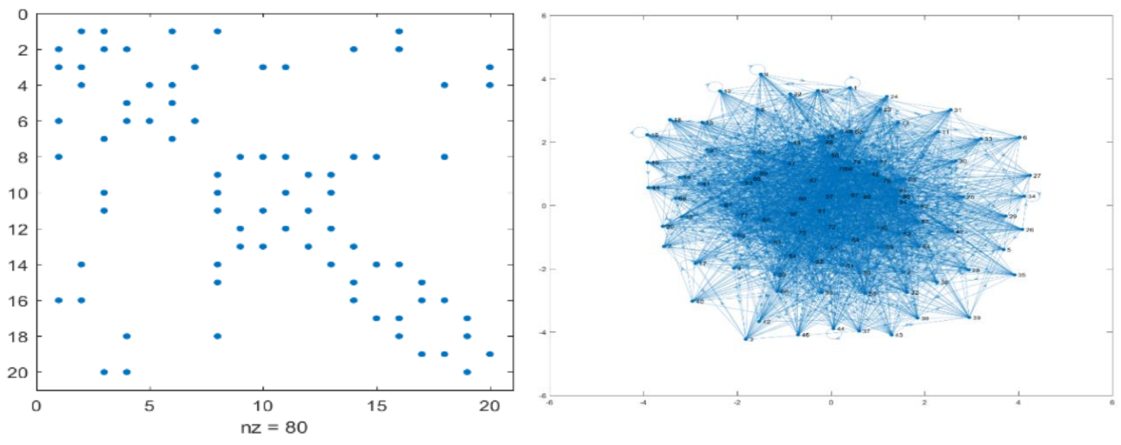
The computers are responsible for locating the essential vertices, and once these graphs continue to grow large, the algorithms come into play. The matrix representation of these type of graphs is commonly large and sparsely populated. From the adjacency matrix in table 1, it is evident that there are 64 cells in which only 18 entries contain the nonzero value. For a graph with N vertices, the adjacency matrix comprises N^2 cells. The betweenness centrality(BC) algorithms are used to find the shortest path between vertices, which is complicated and outside the scope of this thesis, but still, the performance is dominated by sparse matrix multiply performance[22]. When dealing with tens or even hundreds of thousands of vertices extracted from graphs, adjacency matrix becomes too large to be processed. Since, the number of zeros in the sparse matrix is high, multiplying or adding two nonzero values together is low and consumes hardware[23]. During sparse matrix performance on a processor, the frequency of non-zero calculations with computer's clock cycle is little between the ranges of 0.5% to 0.1%

which also directly depends on the size of the matrix. For efficient handling of the sparse matrices, various storage formats can be used to store only the nonzero value, thereby reducing the size of the matrix in memory on an embedded system [24]. As embedded digital systems are limited in both their memory size and their computational power, the key is to make the algorithms faster to reduce the requirements.

“I observed that most of the coefficients in our matrices were zero, i.e., the nonzero were ‘sparse’ in the matrix, and that typically the triangular matrices associated with the forward and back solution provided by Gaussian elimination would remain sparse if pivot elements were chosen with care” - Harry Markowitz.

3.2 Sparse Matrices

Sparse matrices are generally considered to be populated with zeros than nonzero. There is no rule defining when a matrix is sparse. According to Gilbert, any matrix, which allows special techniques to take advantage of many zeros, is a sparse matrix. When storing and performing operations on a sparse matrix, it is desirable to modify the standard algorithm to take advantage of the sparsity[25]. By nature, sparse data yields savings in memory usage. Sparse matrix arises from data communication networks, connections in electronic circuits, with constraints in a linear or non-linear programming formulation, in the discretization of ordinary or partial differential equations in simulation models[26]. Many of the sparse matrices are used in science and engineering today with larger dimensional; there is a lot of research carried out only to store and operate on the non-zero elements of a matrix. This is true when working on large volumes of data with less spatial locality which would do not fit into a CPU’s chip memory cache especially for sparse matrix computations and convolution [27]. There are different and specific forms of sparsity patterns, where indices are used so that one can know where the nonzero is located within the matrix. To maximize the performance of sparse matrix operations, it is especially important to optimize the operations and not just within individual operations. In Verilog and VHDL, there will always be more than one way to code the same problem. It also provides several alternatives to the designer as to how to accomplish the same task. Therefore, a choice of a coding style is needed to achieve specific performance goals and to minimize resource utilization on a chip. The computational complexity of sparse operations is proportional to the number of nonzero elements in the matrix. The storage of a given sparse matrix will be O(nonzeros). The time required for particular operation on the sparse matrix is close to O(flops). Figure 3(a) is a representation of 20% sparsely populated 20×20 matrix with 80 nonzero values, and Figure 3(b) is a representation of 30% sparsely populated 100×100 matrix with 3000 nonzero values.



(a) (b)

Figure 3. Sparsity Pattern of Matrices

Sparse matrices are useful for computing large scale applications that dense matrices cannot handle. The finite element method is one way of solving partial differential equations where the coefficients are usually sparse. The size of the coefficient is large for getting an accurate approximation to solve PDEs and rely on sparse matrix operations.

3.3 Sparse Matrix Storage Formats

Numerous efforts have been devoted to data storage formats with the aim of maximizing performance. To fully optimize the sparse matrix operations, we will have to design a compression algorithm which will take the sparse matrices structures into account. The section of the thesis briefly describes the most common compression/storage formats to date. The primary goal of these different format variations relies on either improve the architectures ability to access the nonzero data and to perform computations by reducing the total space required to store the matrix[28]. Out of all the formats, Compressed Sparse Row (CSR) is the most common format regardless of the processor which stores the elements row-wise. Another form is the Compressed Sparse Column (CSC) that stores the elements column-wise.

There are many methods for storing only the nonzero elements of a sparse matrix out of which the following have gained a lot of attention due to their computational capability and the efficiency in storing the elements.

1. Compressed Row Storage (CRS)
2. Compressed Column Storage (CCS)
3. Block Compressed Row Storage (BCRS)
4. Compressed Diagonal Storage (CDS)
5. Coordinate Format (COO)

Table 2 gives a summary of the various storage formats used for implementation, their storage space computation depending on the number of nonzero values available in the sparse matrix, their advantage, and disadvantage.

Table 2. Comparison of Various Sparse Matrix Storage Formats

Storage Format	**Storage Space**	**Advantage**	**Disadvantage**
Basic Storage Formats			
Coordinate Format (COO)	$3 \times NZV$	It is suitable for any random sparse matrix.	It occupies a lot of space.
Compressed Sparse Column (CSC)	$2 \times NZV + n + 1$	This reduces storage allowing row pointers to facilitate fast multiplication.	Not suitable for GPU due to load imbalance, reduce parallelism and irregular memory access patterns.
Diagonal Format	$N_{dia} + N_{lz} \times N_{dia}$	It is very effective for matrix with non-zero elements only in the diagonal.	It is applicable only for matrices whose diagonal elements are non-zeros.
ELLPACK	$2(N_{mnzr} \times m)$	It is well suited for semi-structured and unstructured meshes.	It requires to know the maximum number of non-zero elements present in the matrix.
Compressed Sparse Row (CSR)	$2 \times NZV + m + 1$	It is effective for structured and unstructured sparse matrices	It uses one-dimensional arrays.
Block Based Storage Formats			
Blocked CSR Format	$(N_{nzb} \times 2r) + N_{nzb} + m / r + 1$	It reduces the number of load operations.	It requires an extra loop for matrix operation and suffers from additional overhead.

Row-Grouped CSR Format	2X + m + N_g X=$\sum_{i=1}^{N_g}\left(N_{mnzr} g(i)\right) \times G_{size}$	Number of allocated elements per row vary from one group to another group.	It is time consuming process and requires 4 arrays
Quad Tree CSR Format	$N_q \times CSR_{storage}$	Sparse matrix vector multiplication is faster.	It requires space overhead
Minimal Quad Tree Format (MQT)	$Min(MQT)_{storage} = 4 \times \left(N/3 + log_4(n^2/N)\right)$ $Max(MQT)_{storage} = 4 \times N(1/3 + log_4(n^2/N))$	It is efficiently used in I/O operations	It requires space overhead in storing the pointers
Vectorizable Format			
Compressed Multi-Row Storage Format (CMRS)	$(3 \times NZV) + N_s + 1$	It does not require any zero padding and row and column reordering.	It is suitable only for GPU architecture
Adaptive CSR Format	$2 \times NZV + m + 1$	It is effective for GPU specific formats	The transformation overhead poses storage and runtime overhead
Streamed Storage Format			
Streamed CSR Format	$2(N_s \times \max(NZS)) + S_r + 1$	It improves the computation speed	It is suitable for coprocessor SIMD architecture only
Streamed BCSR Format	$N_S \times (N_{nzb} \times b_{size}) + 2(N_{nzb})$	It provides better speedup than BCSR	It is suitable for coprocessor SIMD architecture only
Sliced ELLPACK-C-Sigma Format	$N_S + 1 + 4 \times (\sum_{i=1}^{N_S} N_{nzv}(i))$	It reduced the number of zero padding.	Sorting globally will reduce the spatial and temporal locality

The storage format used to store the nonzero values of the given sparse matrix **A** with size **N**×**N** row-size using three one-dimensional arrays. Let nnz denote the number of nonzero elements of **A**. The first array is called ROW and is of length **M**+1, i.e., one entry per row, plus one which contains the row index of **A** where the nonzero element is located. ROW array of the matrix **A** extends from the start of one row to the last row of size **N×N**. The last entry of the **ROW** array will be the last row of the matrix depending on the nonzero elements of the matrix. The second array is called **COL**, which contains the column index of matrix **A** where the nonzero element is located. The **COL** array entries start from the first column until the last column, based on the number of columns available. The third array is called **VALUE** and is of the length of the number of nonzero. This array holds the values of all the nonzero elements of matrix **A** investigating left-to-right and then top-to-bottom order. A depiction of the sparse matrix **A** is shown with the storage format used in this thesis in the below figure.

$$\boldsymbol{A} = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 6 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Figure 4(a). Sparse matrix **A**

$$\begin{aligned} ROW &= [\,0 \quad 0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 4 \quad 5 \quad 7 \quad 8\,] \\ COL &= [\,4 \quad 6 \quad 4 \quad 8 \quad 1 \quad 2 \quad 6 \quad 2 \quad 5 \quad 0\,] \\ VAL &= [\,1 \quad 3 \quad 2 \quad 1 \quad 3 \quad 3 \quad 1 \quad 3 \quad 6 \quad 0\,] \end{aligned}$$

Figure 4(b). Sparse matrix **A** in storage format

Figure 4. Storage of Input Matrix

An analysis of five FPGA-based architectures indicates COO format achieves higher efficiency at the cost of locally storing a copy of vector in each processing element (PE) by eliminating references to the vector.

3.4 Finite State Machines (FSMs)

Finite State Machines (FSMs) are a useful abstraction for sequential circuits with "states" of operation. It has a final internal memory, and the operation of FSM begins from one state goes through a transition to different states. Due to their simplicity, they are quick to implement easy for implementation and fast in execution [29]. At each clock edge, combinational logic block computes outputs and next state as a function of inputs and present state. One of the critical factors for optimizing an FSM design is the choice of state coding, which influences the complexity of the logic functions, the hardware costs of the circuits, timing issues, power usage, etc. One of the disadvantages is that it is not suitable for all domain problems, but only when all the state transitions and conditions need to be known upfront and defined. Some of the FSM encoding styles are one-hot, gray code, Johnson code, Compact, Sequential, and Speed1. Each encoding technique has their performance improvements. Speed1 encoding style was able to achieve higher timing performance for matrix operations on an FPGA board. The state transition of speed1 encoding style is shown in Figure 5.

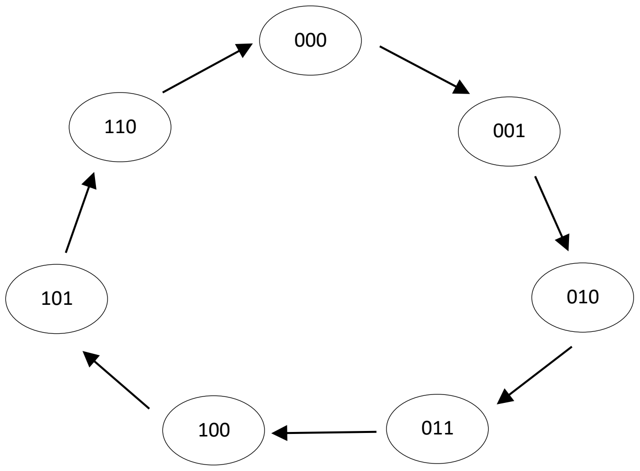


Figure 5. State Diagram of a Simplified Finite State Machine

3.5 Field Programmable Gate Arrays (FPGAs)

3.5.1 FPGA Architecture. FPGAs are digital integrated circuits (ICs) belonging to a family of Programmable Logic Devices (**PLD**). The FPGA chip includes I/O blocks and core programming fabric. These I/O blocks are located around the periphery of the chip, which provides programmable I/O connections for various I/O standards. It also consists of Configurable Logic Blocks (**CLB**) and programmable routing architectures. The appropriate configuration is used in an FPGA for implementing any digital circuit considering the available resources on the board. The figure shows a general FPGA fabric, which represents a popular architecture in the FPGAs, are based. Many different architectures with programming technologies have evolved to provide better results making them the economically viable alternative to Application Specific Intergerat4ed Circuits (**ASIC)**. FPGA's offers excellent flexibility than ASIC's and offers low-level optimization opportunities to improve run-time performance. It has been proved they are considerably more power efficient than multi-core CPUs and GPUs. These logic chips can be reconfigured to implement custom applications. This results in lower time-to-market than traditional ASIC making them significantly faster than general-purpose hardware. It has the necessary resources such as the Look-Up Tables (**LUT**), Flip-Flops (**FF**), Digital Signal Processors (**DSP**) and Block Ram (**BRAM**) available in-built for implementing logical functions and arithmetic operations [30].

Modern FPGAs provide superior logic density, low chip cost, and better performance improvements. It can be used to implement systems that need to be operated up to 550 MHz in most of the design the entire operation can be performed on a single FPGA, and do not require custom hardware. The typical frequency of FPGA design is in low hundreds of MHz, but they have a much finer granularity. Figure 6 shows a rough sketch of the FPGA architecture and the design of a logic block.

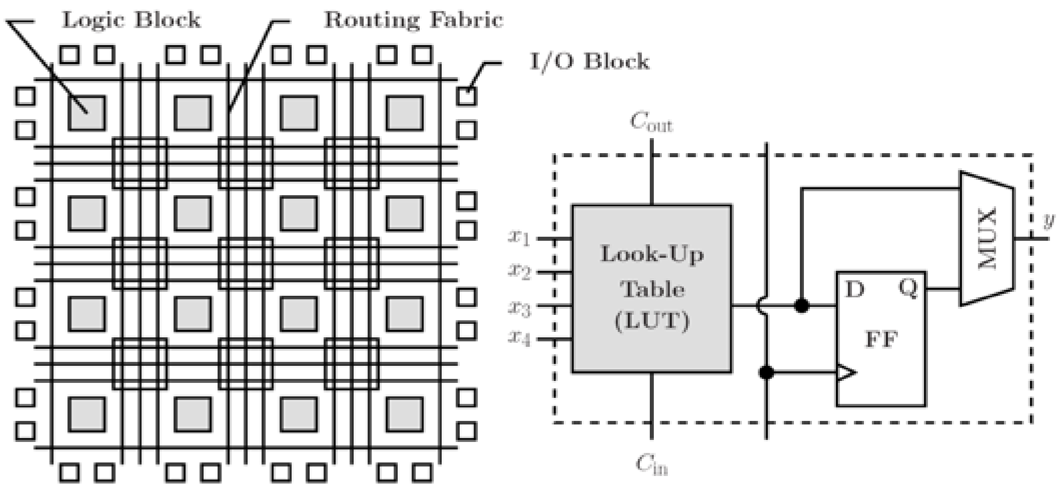
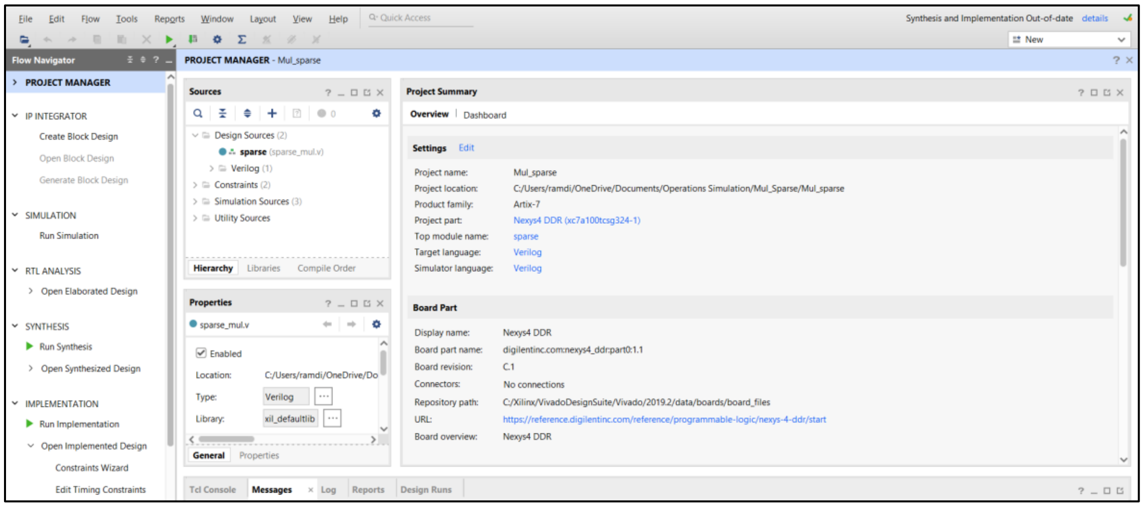


Figure 6. Sketch of FPGA Architecture and Design of Simple Logic Block

3.5.2 Design and Programming.



Enabled [✓]
Location: C:/Users/ramdi/OneDrive/Do
Type: Verilog
Library: xil_defaultlib

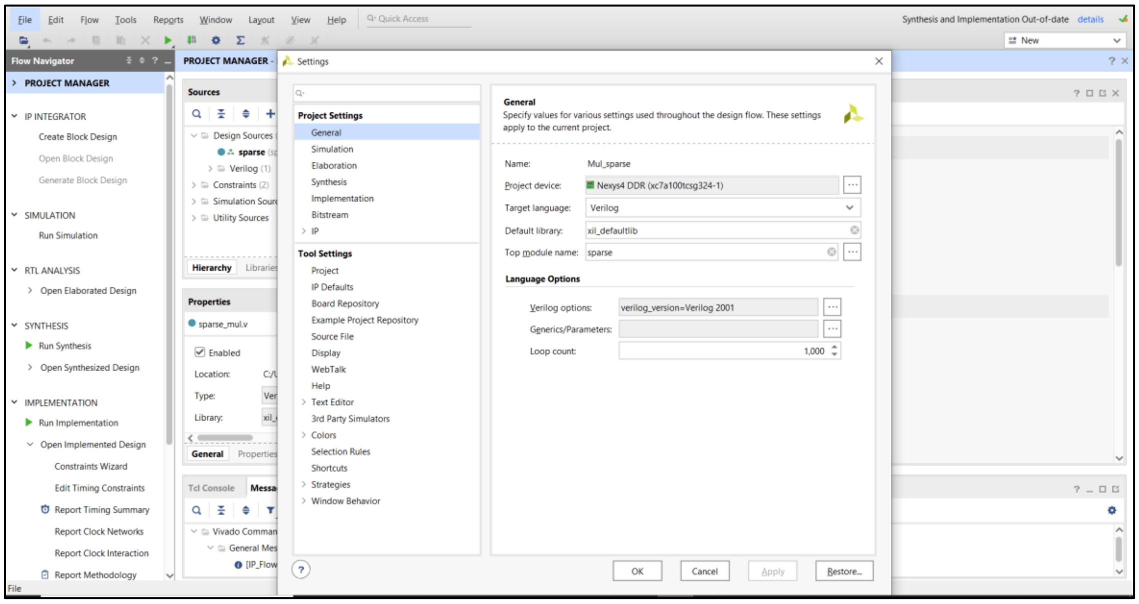
Project name:	Mul_sparse
Project location:	C:/Users/ramdi/OneDrive/Documents/Operations Simulation/Mul_Sparse/Mul_sparse
Product family:	Artix-7
Project part:	Nexys4 DDR (xc7a100tcsg324-1)
Top module name:	sparse
Target language:	Verilog
Simulator language:	Verilog

Display name:	Nexys4 DDR
Board part name:	digilentinc.com:nexys4_ddr:part0:1.1
Board revision:	C.1
Connectors:	No connections
Repository path:	C:/Xilinx/VivadoDesignSuite/Vivado/2019.2/data/boards/board_files
URL:	https://reference.digilentinc.com/reference/programmable-logic/nexys-4-ddr/start
Board overview:	Nexys4 DDR

Figure 7. Vivado Design Suite

The hardware design is primarily implemented using the EDA Tool Vivado Design Suite/Xilinx ISE Design Tool through programming, simulation, synthesis, implementation through debugging and the results are analyzed. Figure 7 shows an overview of the EDA Design Suite.

3.5.2.1 Design Entry.



General
Specify values for various settings used throughout the design flow. These settings apply to the current project.

Name: Mul_sparse
Project device: Nexys4 DDR (xc7a100tcsg324-1)
Target language: Verilog
Default library: xil_defaultlib
Top module name: sparse

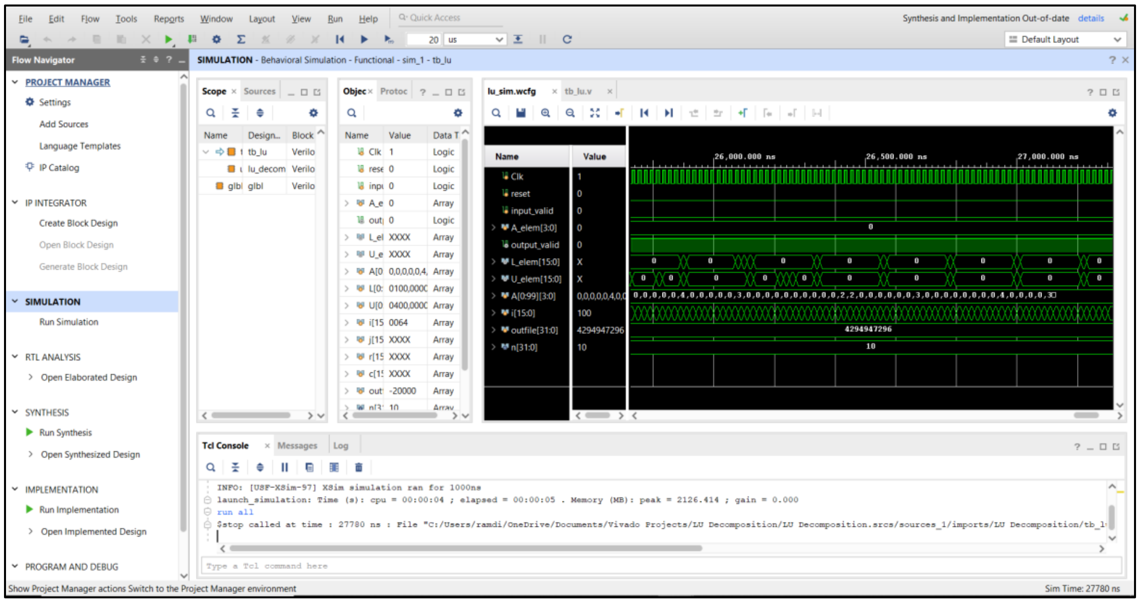
Language Options

Verilog options: verilog_version=Verilog 2001
Generics/Parameters:
Loop count: 1,000

Figure 8. Vivado Project Manager

Figure 8 shows the window of the Project Manager used to manage the implementation from start to end. This block describes the functionality of the design. The design entry can be done by schematic capture or a state transition diagram or by constructing an HDL based model using Verilog/VHDL. The model is built by writing HDL code using a text editor. Recent synthesis tools like Vivado and Xilinx ISE provide facilities for insertion of language templates for easier coding [31]. This step also allows analyzing the internal form for syntax and semantics for the HDL source.

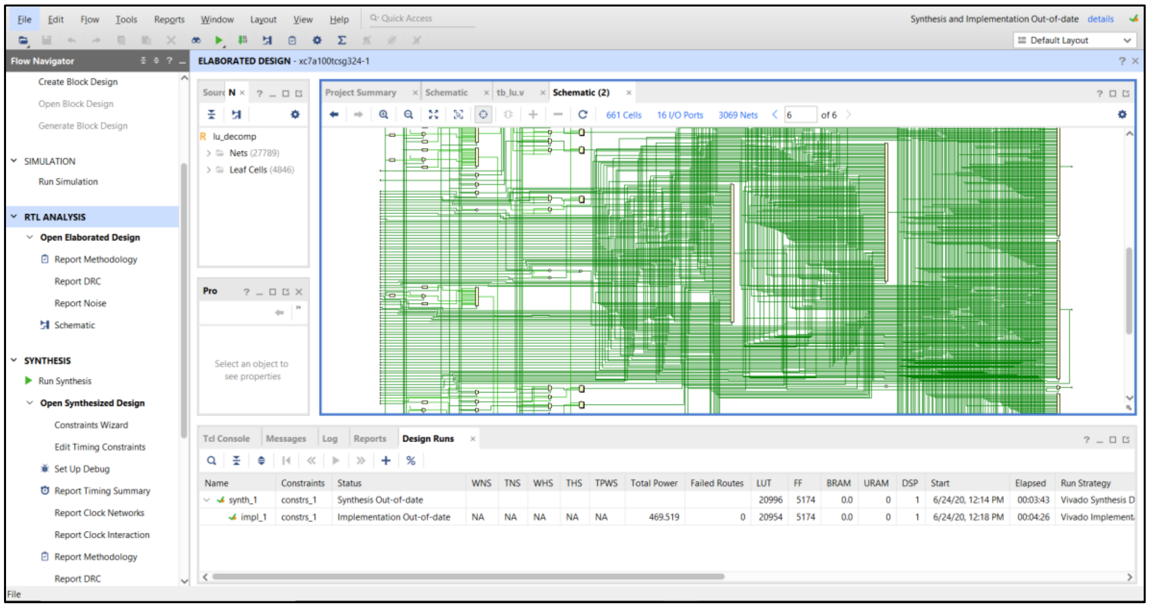
3.5.2.2 Behavioral Simulation. The HDL module designed during design entry if then simulated at the Register Transfer Level (**RTL**) to establish functional correctness. This is the primary step involving simulation of the code to determine that it is working as per the design and that it will produce the required results. Simulation is essential to get as many bugs out from the HDL module[8]. If an error arises, the design entry step is investigated, and necessary changes are made for a successful simulation.



```
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
launch_simulation: Time (s): cpu = 00:00:04 ; elapsed = 00:00:05 . Memory (MB): peak = 2126.414 ; gain = 0.000
run all
$stop called at time : 27780 ns : File "C:/Users/ramdi/OneDrive/Documents/Vivado Projects/LU Decomposition/LU Decomposition.srcs/sources_1/imports/LU Decomposition/tb_l
```

Figure 9. Vivado Simulation Environment

3.5.2.3 RTL Analysis.

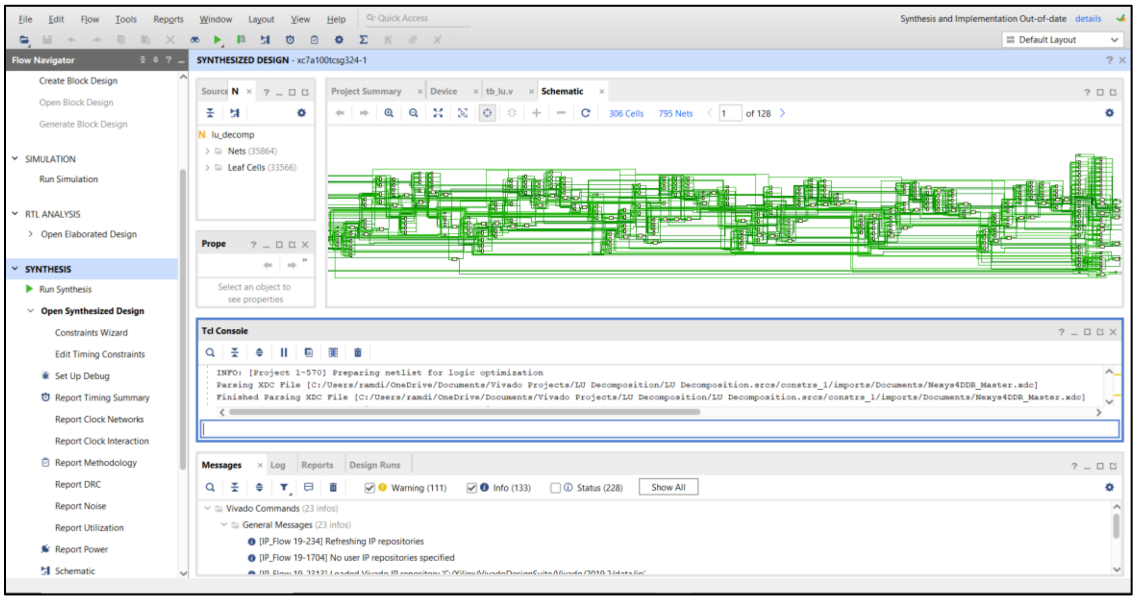


Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Total Power	Failed Routes	LUT	FF	BRAM	URAM	DSP	Start	Elapsed	Run Strategy
synth_1	constrs_1	Synthesis Out-of-date								20996	5174	0.0	0	1	6/24/20, 12:14 PM	00:03:43	Vivado Synthesis D
impl_1	constrs_1	Implementation Out-of-date	NA	NA	NA	NA	NA	469.519	0	20954	5174	0.0	0	1	6/24/20, 12:18 PM	00:04:26	Vivado Implement

Figure 10. Vivado RTL Analysis Tool

Figure 10 provides how an RTL analysis tool on Vivado Design Suite would be helpful in overseeing the schematic for potential issues. The RTL Analysis is used for analyzing the syntactic and semantic issues, identifying potential implementation issues with latches and nets. This tool helps in realizing the mapping of LUT onto the FPGA resources.

3.5.2.4 Synthesis.

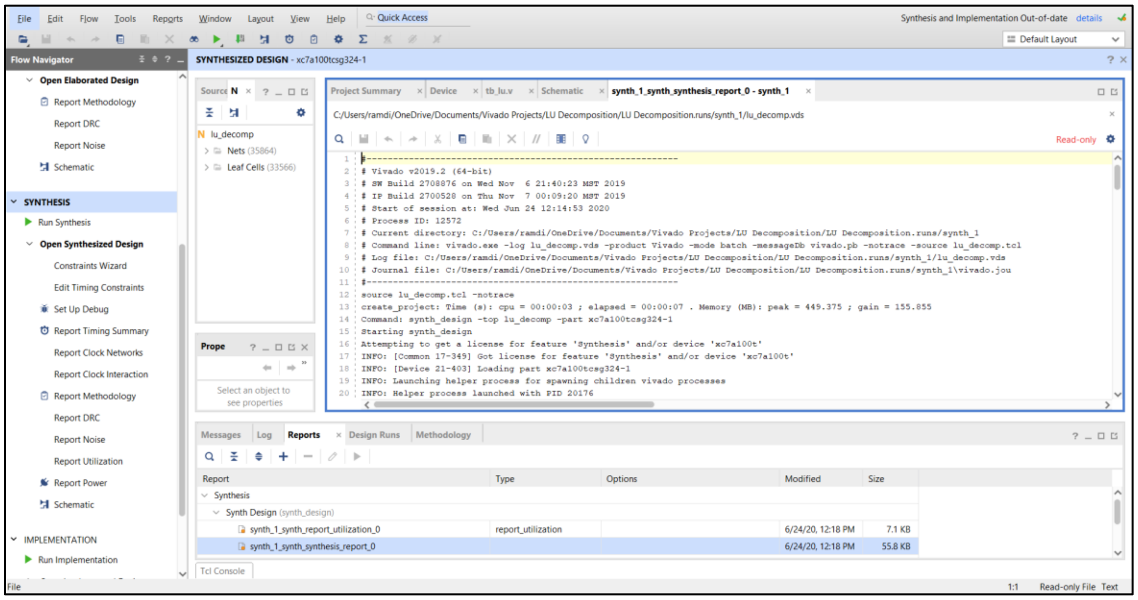


```
INFO: [Project 1-570] Preparing netlist for logic optimization
Parsing XDC File [C:/Users/ramdi/OneDrive/Documents/Vivado Projects/LU Decomposition/LU Decomposition.srcs/constrs_1/imports/Documents/Nexys4DDR_Master.xdc]
Finished Parsing XDC File [C:/Users/ramdi/OneDrive/Documents/Vivado Projects/LU Decomposition/LU Decomposition.srcs/constrs_1/imports/Documents/Nexys4DDR_Master.xdc]
```

Warning (111) Info (133) Status (228)

- Vivado Commands (23 infos)
  - General Messages (23 infos)
    - [IP_Flow 19-234] Refreshing IP repositories
    - [IP_Flow 19-1704] No user IP repositories specified

Figure 11. Vivado Synthesis Analysis Tool



SYNTHESIZED DESIGN - xc7a100tcsg324-1
synth_1_synth_synthesis_report_0 - synth_1
Read-only

C:/Users/ramdi/OneDrive/Documents/Vivado Projects/LU Decomposition/LU Decomposition.runs/synth_1/lu_decomp.vds

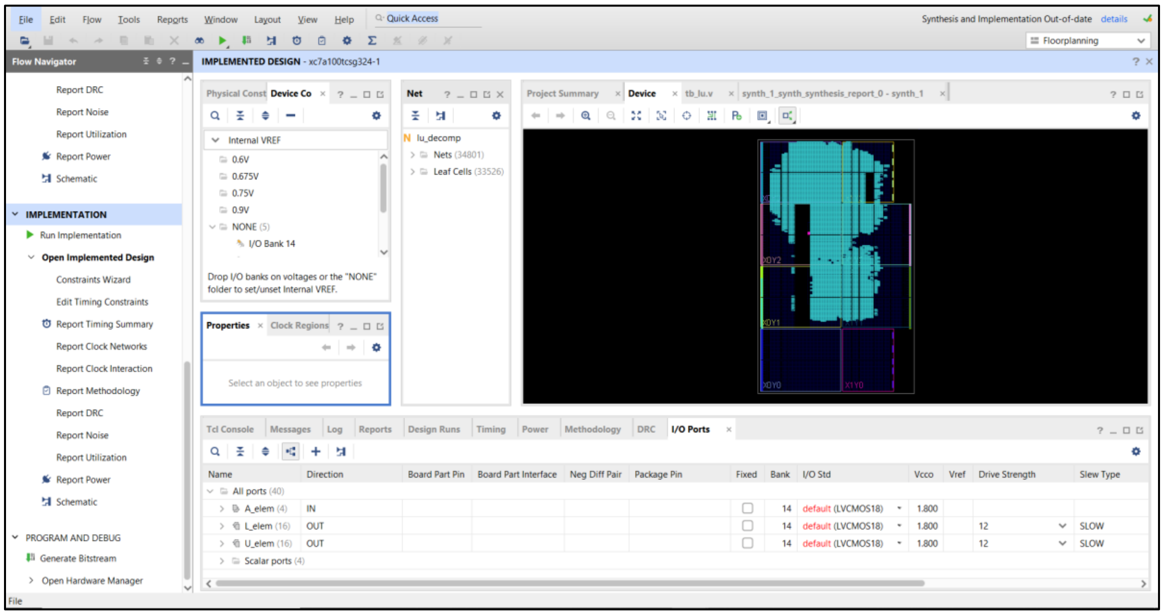
```
#-----------------------------------------------------------
# Vivado v2019.2 (64-bit)
# SW Build 2708876 on Wed Nov  6 21:40:23 MST 2019
# IP Build 2700528 on Thu Nov  7 00:09:20 MST 2019
# Start of session at: Wed Jun 24 12:14:53 2020
# Process ID: 12572
# Current directory: C:/Users/ramdi/OneDrive/Documents/Vivado Projects/LU Decomposition/LU Decomposition.runs/synth_1
# Command line: vivado.exe -log lu_decomp.vds -product Vivado -mode batch -messageDb vivado.pb -notrace -source lu_decomp.tcl
# Log file: C:/Users/ramdi/OneDrive/Documents/Vivado Projects/LU Decomposition/LU Decomposition.runs/synth_1/lu_decomp.vds
# Journal file: C:/Users/ramdi/OneDrive/Documents/Vivado Projects/LU Decomposition/LU Decomposition.runs/synth_1\vivado.jou
#-----------------------------------------------------------
source lu_decomp.tcl -notrace
create_project: Time (s): cpu = 00:00:03 ; elapsed = 00:00:07 . Memory (MB): peak = 449.375 ; gain = 155.855
Command: synth_design -top lu_decomp -part xc7a100tcsg324-1
Starting synth_design
Attempting to get a license for feature 'Synthesis' and/or device 'xc7a100t'
INFO: [Common 17-349] Got license for feature 'Synthesis' and/or device 'xc7a100t'
INFO: [Device 21-403] Loading part xc7a100tcsg324-1
INFO: Launching helper process for spawning children vivado processes
INFO: Helper process launched with PID 20176
```

Report	Type	Options	Modified	Size
Synthesis				
Synth Design (synth_design)				
synth_1_synth_report_utilization_0	report_utilization		6/24/20, 12:18 PM	7.1 KB
synth_1_synth_synthesis_report_0			6/24/20, 12:18 PM	55.8 KB

Figure 12. Vivado Synthesis Report Analysis

Figure 11 depicts the Synthesis Analysis Tool in managing the nets and logics of the design. The process where the RTL design is translated to gate-level design is called synthesis. Later the design can be mapped to the logic blocks in the FPGA which checks whether the design will meet the timing and area constraints. A device netlist format is created during this step. Figure 12 shows the Report Analysis where the usage of the adders and multipliers are mentioned, for a detailed analysis.

3.5.2.5 Implementation.



Name	Direction	Board Part Pin	Board Part Interface	Neg Diff Pair	Package Pin	Fixed	Bank	I/O Std	Vcco	Vref	Drive Strength	Slew Type
All ports (40)												
A_elem (4)	IN					☐	14	default (LVCMOS18)	1.800			
L_elem (16)	OUT					☐	14	default (LVCMOS18)	1.800		12	SLOW
U_elem (16)	OUT					☐	14	default (LVCMOS18)	1.800		12	SLOW
Scalar ports (4)												

Figure 13. Vivado Implementation Tool

The implementation step consists of the steps of mapping, placing, routing, and generating a BIT file for the HDL design.

• Mapping

After creating the gate-level netlist, the design is mapped onto the FPGA. The primitives such as function generators, latches or flip-flops used in the target chip are accumulated during this process.

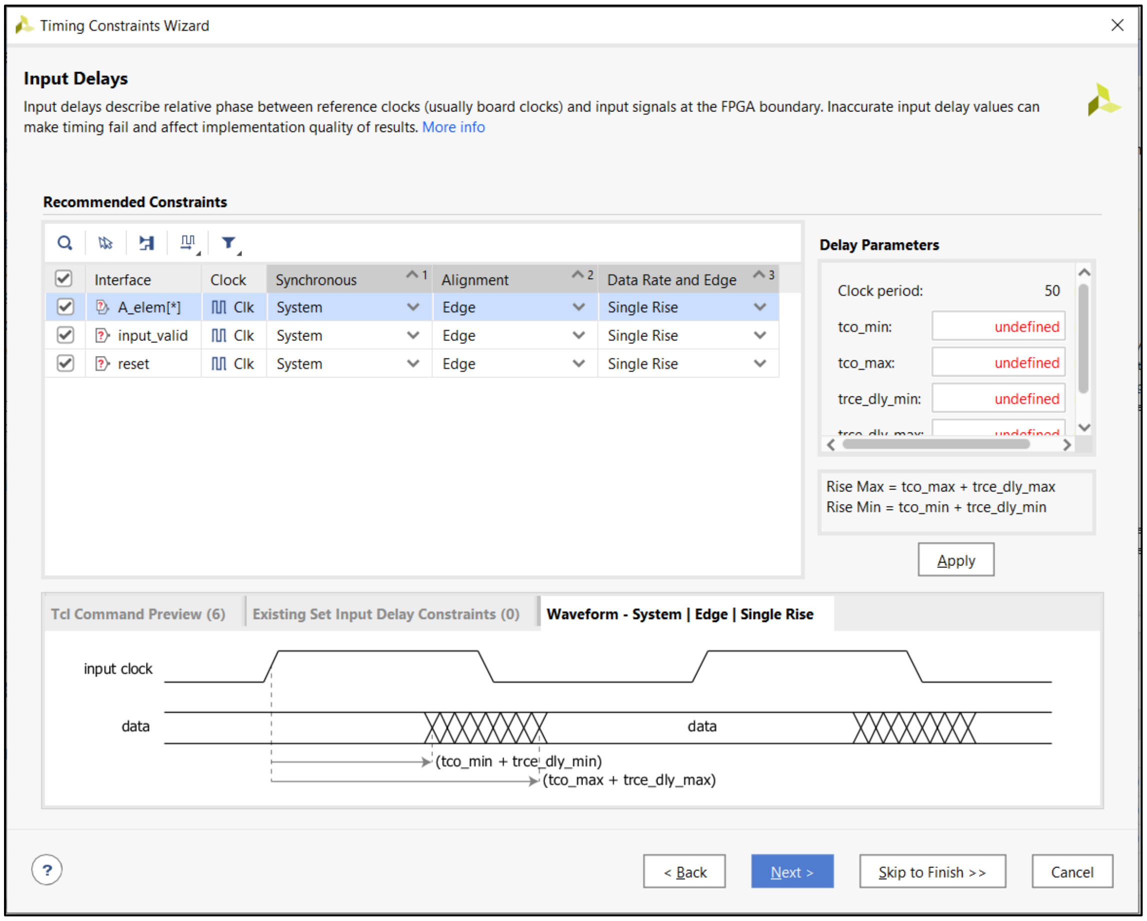
- Place and Route

After mapping the design, the primitives are assigned to the Configurable Logic Blocks (**CLB**) during Place and Route step. The primitives are then connected by routing the connections through the switch matrix. The process provides accurate information about the timing delays between parts of the circuit. The design verification process is simulated which is more accurate than the functional simulation.

- Bitstream Generation

A bitstream file is created from the physical place and route information.

3.5.2.6 Timing Analysis.


Timing Constraints Wizard

Input Delays

Input delays describe relative phase between reference clocks (usually board clocks) and input signals at the FPGA boundary. Inaccurate input delay values can make timing fail and affect implementation quality of results. More info

Recommended Constraints

	Interface	Clock	Synchronous	Alignment	Data Rate and Edge
✓	A_elem[*]	Clk	System	Edge	Single Rise
✓	input_valid	Clk	System	Edge	Single Rise
✓	reset	Clk	System	Edge	Single Rise

Delay Parameters

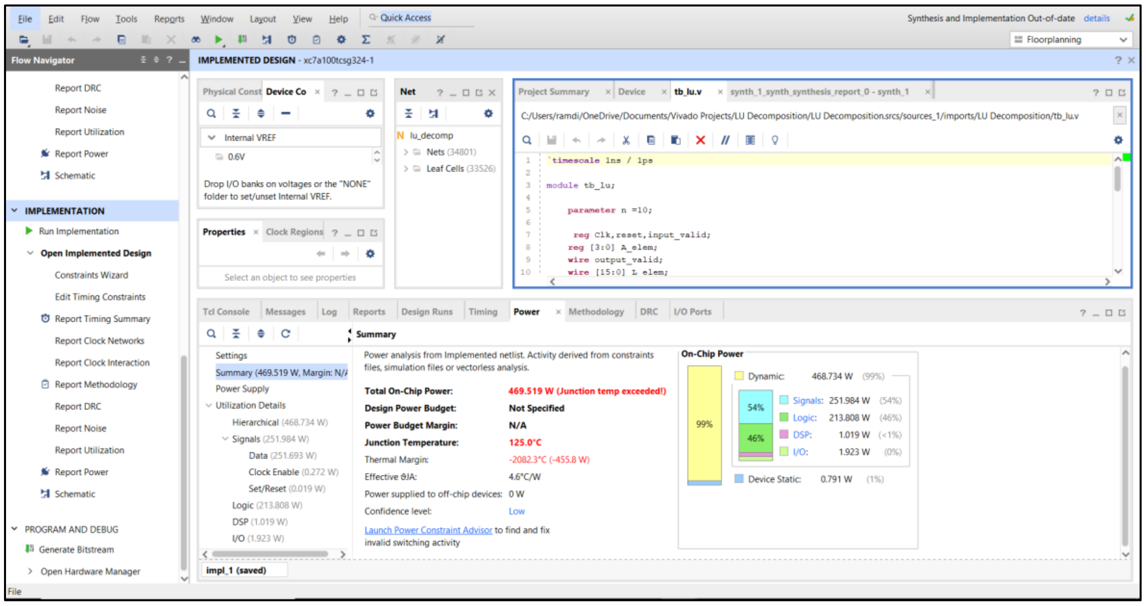
Clock period:	50
tco_min:	undefined
tco_max:	undefined
trce_dly_min:	undefined
trce_dly_max:	undefined

Rise Max = tco_max + trce_dly_max
Rise Min = tco_min + trce_dly_min

Figure 14. Vivado Timing Constraints Tool

Figure 14 is the Timing Constraints Wizard which is used for employing our used desired timing, and analysis. Once the design is mapped, placed, and routed, the delays of the signals and the components of the design are used to produce a new, more detailed netlist leading to a timing accurate simulation.

3.5.2.7 Power Usage Analysis.



```
`timescale 1ns / 1ps

module tb_lu;

    parameter n =10;

     reg Clk,reset,input_valid;
    reg [3:0] A_elem;
    wire output_valid;
    wire [15:0] L elem;
```

Summary

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**469.519 W (Junction temp exceeded!)**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**125.0°C**
Thermal Margin:	-2082.3°C (-455.8 W)
Effective ϑJA:	4.6°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Launch Power Constraint Advisor to find and fix invalid switching activity

On-Chip Power

Dynamic:	468.734 W	(99%)
Signals:	251.984 W	(54%)
Logic:	213.808 W	(46%)
DSP:	1.019 W	(<1%)
I/O:	1.923 W	(0%)
Device Static:	0.791 W	(1%)

Settings
Summary (469.519 W, Margin: N/A
Power Supply
Utilization Details
Hierarchical (468.734 W)
Signals (251.984 W)
Data (251.693 W)
Clock Enable (0.272 W)
Set/Reset (0.019 W)
Logic (213.808 W)
DSP (1.019 W)
I/O (1.923 W)

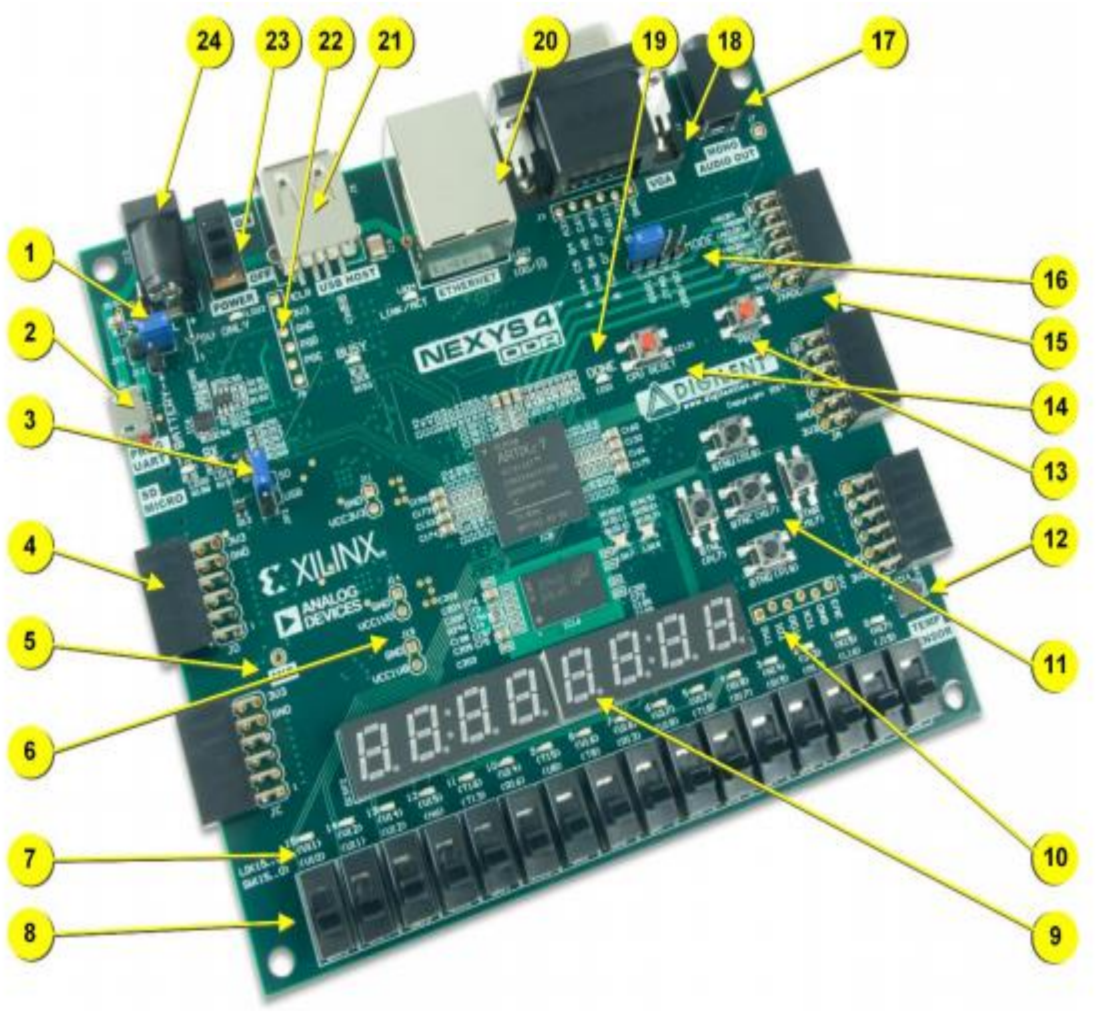
Figure 15. Vivado Power Analysis Tool

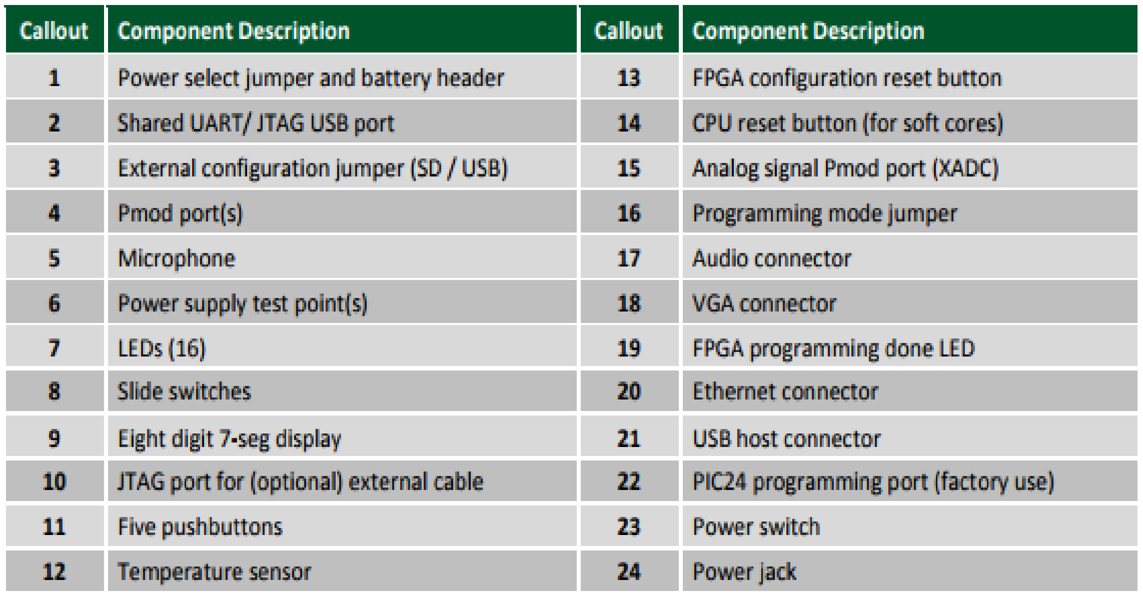
Once the timing and area constraints are met with the design implementation, the Power Analysis tool gives the cost of the design in terms of dynamic and static usage of the design.

This tool also provides improvements on the implementation flow to meet the constraints. It automatically identifies the target FPGA board presented and analysis if the design meets the power constraints.

3.5.2.8 Programming the Board. The bitstream file generated is loaded onto the target FPGA. Once the programming of the board is finished, the chip will now be configured to implement the design. The EDA tool used for this thesis work is Vivado Design Suite and Xilinx ISE (Integrated Synthesis Environment) 14.2 using Verilog.

3.5.3 NEXYS4 DDR ARTIX-7 FPGA Board. The FPGA board used for implementing the final design of this project is Nexys4 DDR Artix-7 development board. The Nexys4 DDR board features Artix-7 family processor from Xilinx with the high-performance logic block, more capacity, higher performance, and more resources. This high-capacity FPGA comes with USB, Ethernet, and other ports so hosting designs from combinational circuits to powerful embedded processors is possible. For this thesis, we used UART terminal to send and receive matrix values to and from computer and FPGA board[8]. The board is programmed through JTAG cable, with an **E3** pin as Clock port, **D4** as UART Transmitter (**UART TX**) and **C4** as UART Receiver (**UART RX**). The FPGA board is shown in Figure 16, with all the necessary details of the board. The UART connector is marked as number 2 in Figure 15, which also has TX and RX led lights.





Callout	Component Description	Callout	Component Description
1	Power select jumper and battery header	13	FPGA configuration reset button
2	Shared UART/ JTAG USB port	14	CPU reset button (for soft cores)
3	External configuration jumper (SD / USB)	15	Analog signal Pmod port (XADC)
4	Pmod port(s)	16	Programming mode jumper
5	Microphone	17	Audio connector
6	Power supply test point(s)	18	VGA connector
7	LEDs (16)	19	FPGA programming done LED
8	Slide switches	20	Ethernet connector
9	Eight digit 7-seg display	21	USB host connector
10	JTAG port for (optional) external cable	22	PIC24 programming port (factory use)
11	Five pushbuttons	23	Power switch
12	Temperature sensor	24	Power jack

Figure 16. Nexys4 DDR Artix-7 FPGA Board

4. HARDWARE DESIGN ARCHITECTURE

Algorithm for sparse matric arithmetic and decomposition operations are designed, and the operations are implemented in hardware with Nexys4 DDR FPGA Board and the results are compared with conventional matrix operation algorithm. The design approach worked with this thesis is shown in Figure 17. Vivado Design Suite is used for the simulation, synthesis, and implementation of the Verilog design and MATLAB is used for result comparison and error analysis.

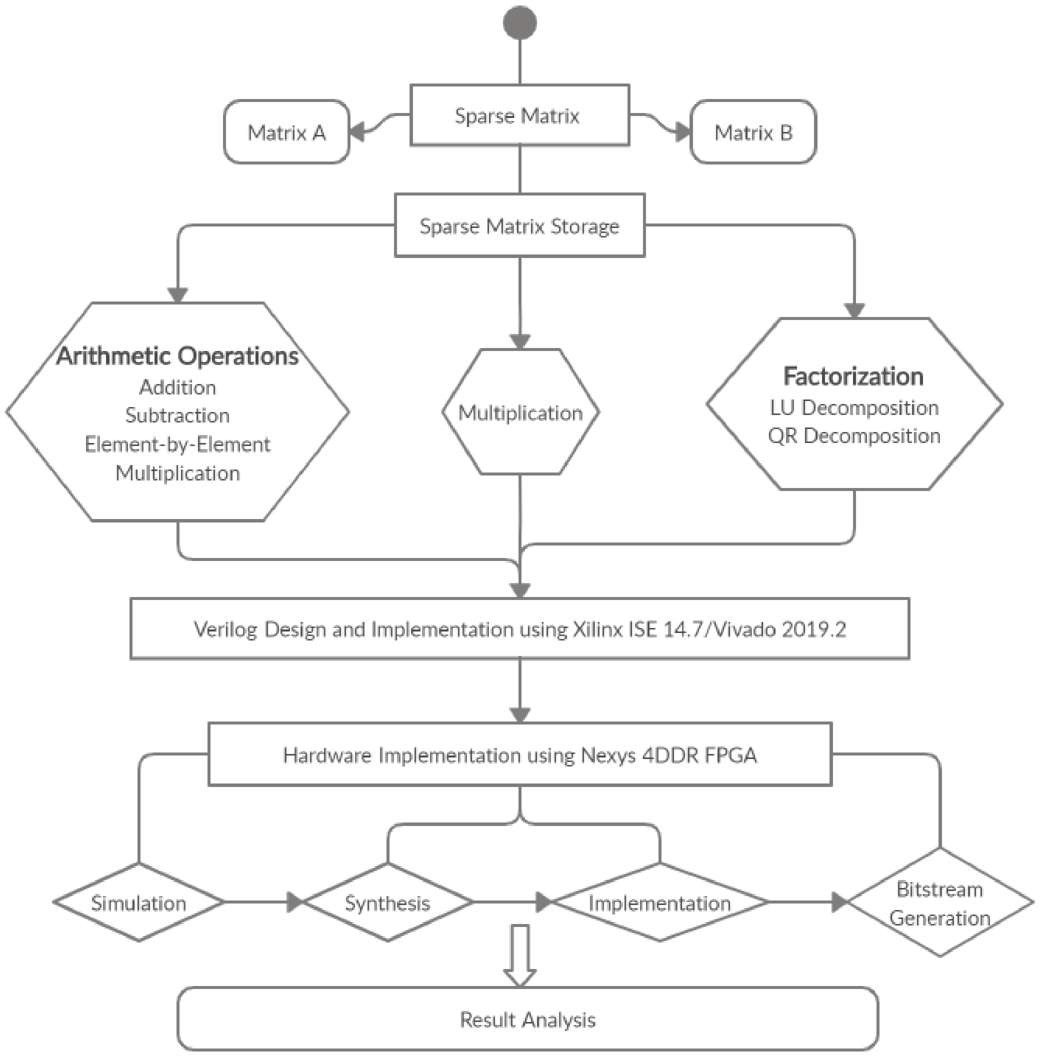


Figure 17. High Level Flow Chart of Work Proposed

• Computational Complexity

Table 3 gives the comparison between the computational complexity of different matrix operations.

Table 3. Computational Complexity

Matrix Operation	Input	Output	Complexity
Addition/Subtraction	2 n×n- matrix	n×n matrix	n^2
Multiplication	2 n×n- matrix	n×n matrix	n^3
Square root	1 n×n- matrix	n×n matrix	n^2-1
Decomposition	1 n×n- matrix	n×n matrix	1/3×n^3

Design and Validation have become a significant step involving various steps from RTL design, logic synthesis, physical design, and verification at an early stage. This makes the testing and verification of a new and complex hardware architecture system a time-consuming process as shown in Figure 18.

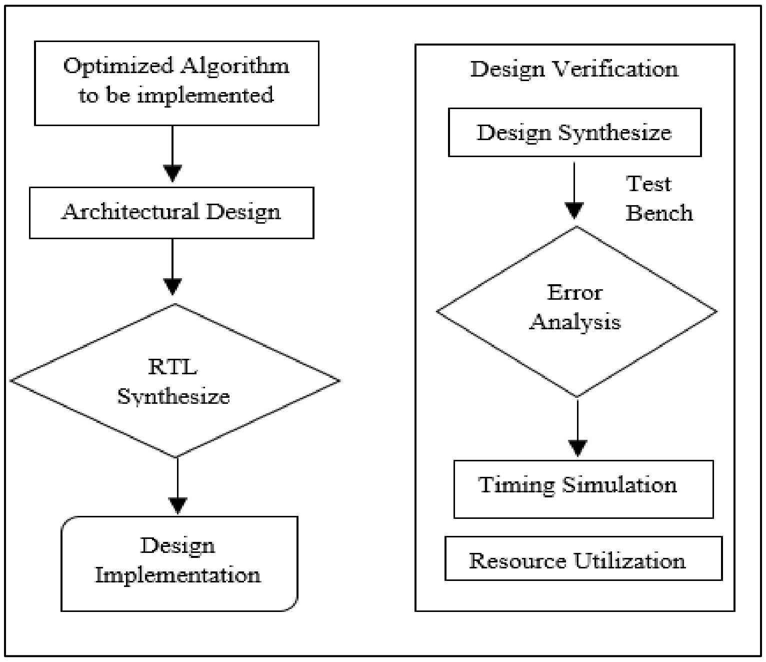


Figure 18. FPGA Implementation and Verification Flow

The hardware design for implementation is based on two factors: precision and area.

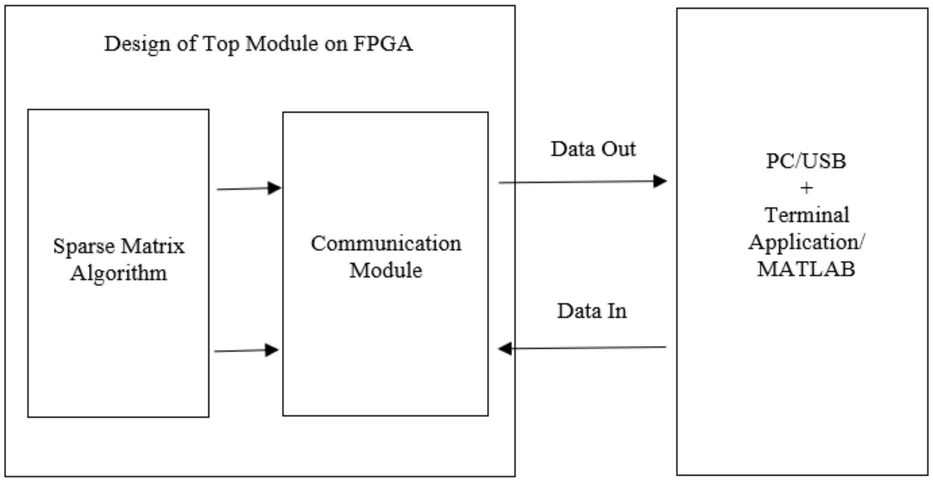

Design of Top Module on FPGA
Sparse Matrix Algorithm
Communication Module
Data Out
Data In
PC/USB + Terminal Application/ MATLAB

Figure 19. Overview of Hardware and Software Implementation

Figure 19 gives an overview of the module designed and how the communication is established between processor and FPGA for implementation and testing. As illustrated above the objective of this thesis, one of the purposes is to reduce the resources utilized in FPGA [32]. Hence, significant attention was given to the design process implemented, as well as to obtain low latency and high throughput compared to the normal matrix operation [33]. This describes the methodologies than influenced the design and design considerations carried out. The preference of FPGA over traditional CPUs and GPUs is because of the advantages offered by FPGAs and CPLDs. After the matrix algorithm for sparse matrix was studied carefully, the next most significant step was the design itself. The design was done using Verilog because of the ease with which large projects can be managed.

The hardware implementation is split into two major top modules for simplifying the design. The first module is designed to implement the necessary sparse matrix operation like addition, subtraction, multiplication, LU decomposition, QR decomposition algorithms and the necessary computations. And the second module is designed to implement the UART communication and data exchange between the PC to the FPGA hardware board with which it will be communicating. Each of the top modules is subdivided into smaller modules to carry out specific matrix operations with the other modules through internal signals. The Figure 20 gives the flow of how the architecture is designed, along with the flow of memory controllers and transition states.

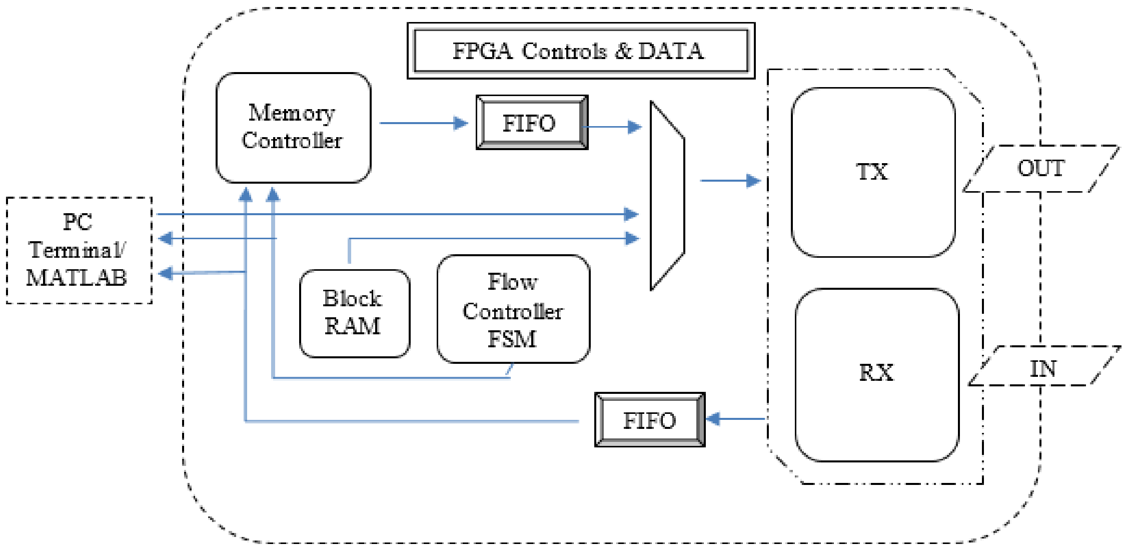


Figure 20. Block Diagram of TX and RX Module

In any asynchronous interface, the first thing we need to know when in time the data should be sampled. If the data is not sampled at the right time, we might get the wrong data. To receive data correctly, the transmitter and receiver must agree on the baud rate. The baud rate if the rate at which the data is transmitted. For example, 9600 baud mean 9600 bits per second.

The Verilog code uses a generic or a parameter to determine how many clock cycles are there in each bit. This is how the baud rate gets determined.

The FPGA is continuously sampling the line. Once it sees the line transition from high to low, it knows the UART data is coming. The first transition indicated the start bit. Once the beginning of the start bit is found, the FPGA waits for one half of a bit period. This ensures that the middle of the data bits gets sampled. From then on, the FPGA just needs to wait once bit period (as specified by the baud rate) and sample the rest of the data. The following Figure 21 is an example of how the output becomes valid when the input clock is high, making the data to be valid for the required operation.

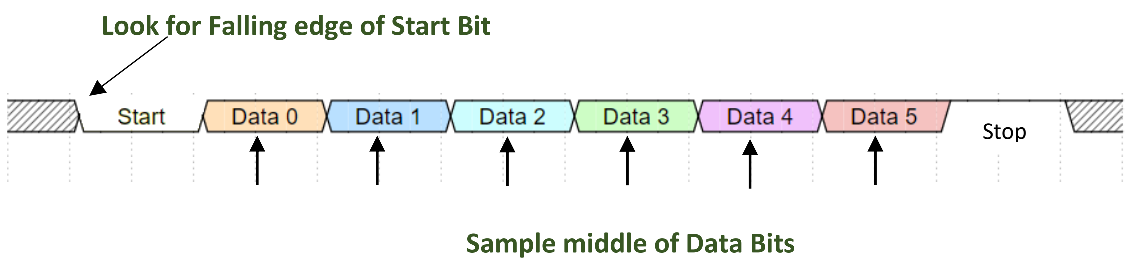

Look for Falling edge of Start Bit
Start
Data 0
Data 1
Data 2
Data 3
Data 4
Data 5
Stop
Sample middle of Data Bits

Figure 21. Clock Cycle for TX and RX

This component is used to transfer data over a UART device. It will serialize a byte of data and transmit it over a TXD line. The serialized data has the following characteristics:

- 9600 Baud Rate
- 8 bits, LSB first
- 1 stop bit
- No parity

TX

- S0_RDY: This signal goes low once a send operation is begun and remains low until it has completed, and the module is ready to send another bit. The counter that keeps track of the number of clocks cycles the current bit has been held stable over the UART. The combinatorial logic that foes high when the counter has counted to the proper value to the correct baud rate.
- S1_LOAD_BIT: The parallel data to be sent. Must be valid the clock cycle when SEND has gone high. Contains the index of the next bit that needs to be transferred. A register that holds the current data being sent over.
- S2_SEND_BIT: Used to trigger a send operation. The upper layer logic should set this signal high for a single clock cycle to trigger a send. When this signal is set high DATA must be valid. Should not be asserted unless READY is high. A register that contains the whole data packet to be sent, including start and stop bits.

Figure 22 captures the FSM diagram of the Transmission bandwidth of the data bit with detailed state transitions and their respective conditions.

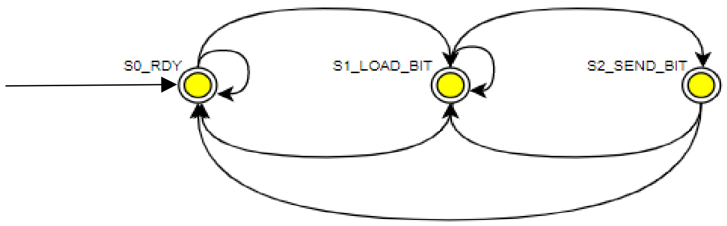


Figure 22. Fundamental Design: Transmission FSM

RX

The purpose is to double-register the incoming data. This allows it to be used in the UART RX Clock Domain. It removes problems created by metastability.

- S1_RX_START_BIT: Checks the middle of start bit to make sure it is still low. The reset counter resets the middle when the middle value is found.
- S2_RX_DATA_BIT: Waits for CLK_PER_BIT-1 clock cycles to sample serial data. Checks if we have sent out all the bits.
- S3_RX_STOP_BIT: Waits for CLK_PER_BIT-1 clock cycles for Stop bit to finish.
- S4_CLEANUP: Stays for I clock cycle.

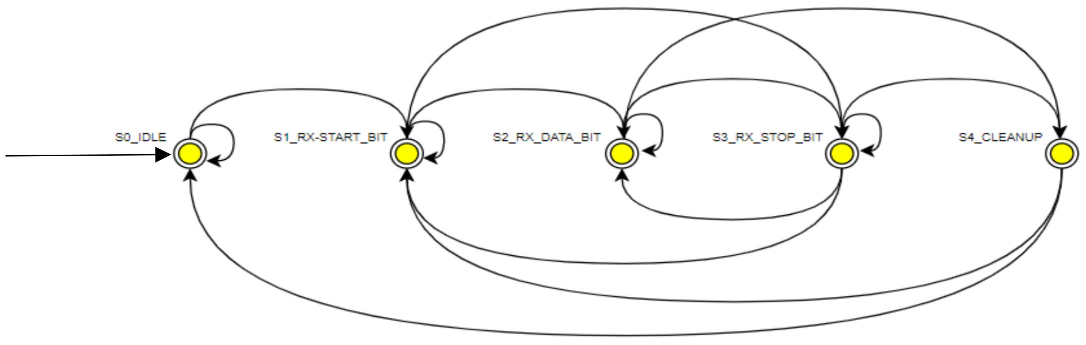


Figure 23. Fundamental Design: Receiver FSM

Figure 23 captures the FSM diagram of the Receiver bandwidth of the data bit with detailed state transitions and their respective conditions.

4.1 Arithmetic Operations

4.1.1 Sparse Matrix Addition. The proposed design performs addition operation of two sparse matrices where only the nonzero values are stored, and required operation is performed. The standard matrix addition stores and performs the operation for all the elements inside the matrix regardless of whether the values are zero or not. The design follows the steps below.

a) A symbolic algorithm, which determined the structure of the resulting matrix.
b) A numerical algorithm which determines the values of the nonzero knowing the knowledge of their positions.

$$\boldsymbol{c_{i,j}} = \left(\boldsymbol{a_{i,j}}\right) + \left(\boldsymbol{b_{i,j}}\right)$$

The proposed architectural algorithm performs sparse matrix addition in which the number of rows and number of columns of two matrices should be equal. A parallel implementation of the addition, with enough fast memory algorithm, is proposed. Consider the matrix addition of **A**+**B**, where **A** has a density **s** percentage with size n×n (square matrix is considered, however, the same methodology can be used for rectangular size), and matrix **B** has a density s percentage with size **n**×**n**. Density s percentage is defined as the number of nonzero elements to the total number of elements in the matrix $\mathbf{n^2}$. The matrix addition performs the operation row-wise and column-wise throughout the matrix only for the nonzero elements present leaving behind the zeros. An algorithm for the sparse matrix addition **A**+**B** is presented in Listing 1. When addition operation must be performed on both the input matrices, first the number of rows and columns are checked if its equal, i.e., both the matrix should be of the same size. Addition operation cannot be performed if the matrices are of different size. Then the elements of the matrix are checked row-wise and column-wise from top-to-bottom order for non-zero elements as shown in the figure. Two separate counters ***A_count*** and ***B_count*** is used to increment the row and column for both **A** and **B** input matrix. This keeps incrementing from **n** to **n+1** for the size of the matrix. The below Listing 1 shows the pseudo code algorithm for the respective arithmetic operation carries out.

```
Input: A, B
Output: C
Input parameter: MAT_SIZE, ELEMENT_SIZE, NZE

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
```

Listing 1. Sparse Matrix Addition Algorithm

The nonzero elements are located from matrix **A**, and the values are stored in the memory $\boldsymbol{Mram\_A\_sv}$ for the corresponding row and column index $\boldsymbol{Mram\_A\_sr}$ and $\boldsymbol{Mram\_A\_sc}$ respectively. Similarly, for the second matrix **B**, the nonzero value gets stored in memory $\boldsymbol{Mram\_B\_sv}$ for the corresponding row and column index $\boldsymbol{Mram\_B\_sr}$ and $\boldsymbol{Mram\_B\_sc}$ respectively. This operation is carried out for the given size of the matrix and is shown in Figure 24 in detail. Once the nonzero is located, the values are stored in terms of row, col, and the corresponding value for which addition operation is to be performed.

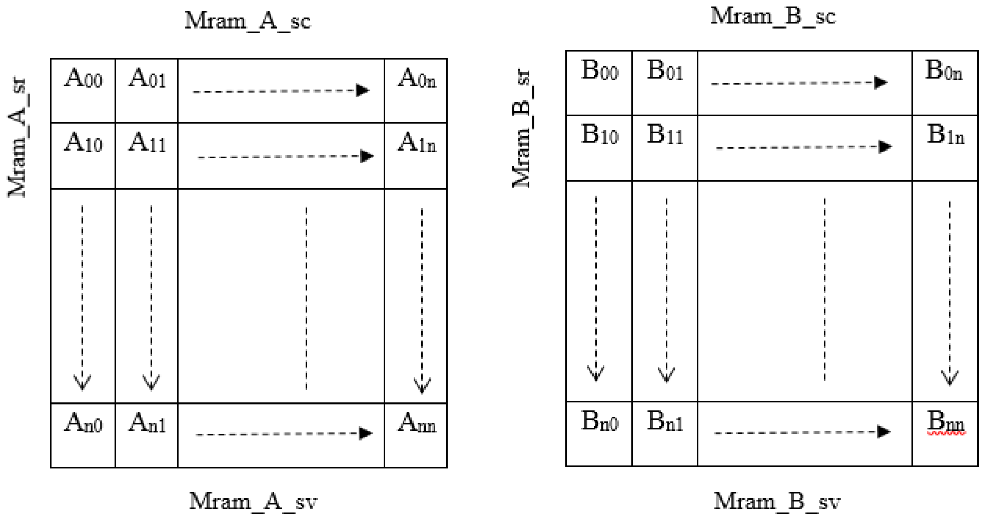


Figure 24. Representation of Row and Column Access of Input Matrices

The most important part of this algorithm is the index comparison which is represented as $\boldsymbol{A\_index}$ for matrix **A** and $\boldsymbol{B\_index}$ for matrix **B**. Initially, once the values are stored the row value of matrix **A** are compared with the row value of matrix **B**. If the index of $\boldsymbol{A\_sr}$ is equal to the index of $\boldsymbol{B\_sr}$ then the next step of comparing the column value of both the matrices. And, if the index of $\boldsymbol{A\_sc}$ is equal to the index of $\boldsymbol{B\_sc}$, then matrix addition is performed. The **VAL** array of the respective row and column, i.e., $\boldsymbol{A\_sv}$ and $\boldsymbol{B\_sv}$ are added with each other as the sum. The assumption is made that the nonzero is located anywhere in the matrix and is highly sparse. Finally, if the nonzero of the same row and same column of matrix **A** does not match with the row and column of matrix **B** directly the value is sent to the output matrix.

To avoid the extra computation imposed by the majority of zero elements found in a sparse matrix, the norm to store the nonzero elements employ auxiliary data structures proposed. The method of employing a row and column pointer to start the index of each row and column with the array of nonzero elements as shown in Figure 23. However, the other structures like CSR and CSC introduces load operations, extra traffic for memory subsystem and cache interference. Access to the input matrix **A** and **B** is irregular and totally depends on the sparsity pattern of the inbound matrix. This eliminates the possibility of exploiting spatial and temporary reuse. Many sparse matrices contain higher number of rows and columns with just zeros resulting in workload imbalance. The proposed optimization is designed to address the corresponding bottleneck, where the other proposed methodology in literature review wither targets a specific bottleneck or a specific sparse structure of matrix. To ensure good performance, the starting and ending index is temporary stored in memory to avoid data overload. However, if the matrix is larger and sparser, the performance bottleneck could not be achieved. Understanding these effects and performance the result analysis is carried out which is explained later as a part of this thesis.

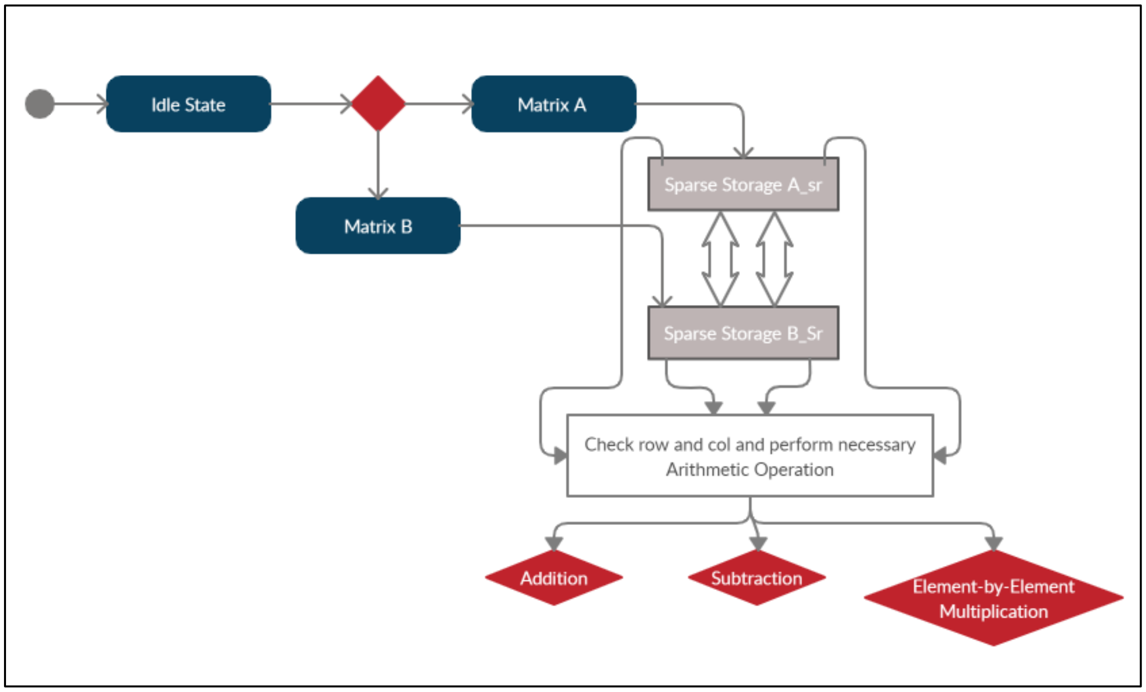

Idle State
Matrix A
Matrix B
Sparse Storage A_sr
Sparse Storage B_Sr
Check row and col and perform necessary Arithmetic Operation
Addition
Subtraction
Element-by-Element Multiplication

Figure 25. FSM Transition States for Sparse Matrix Arithmetic Operation

Figure 25 shows how the state machines are implemented in the Verilog design. The first state is Idle which sets the reset to high. Once the elements are obtained, only the non-zero values get stored using sparse matrix storage format in the order of **ROW**, **COL**, and **VAL** in separate arrays. Once the sparse matrix storage format is generated, the design checks the **ROW** and **COL** and performs addition if both are equal, else the design sends the values directly to output since addition is not required there. With this operation, only the non zeros are involved in the required arithmetic operation.

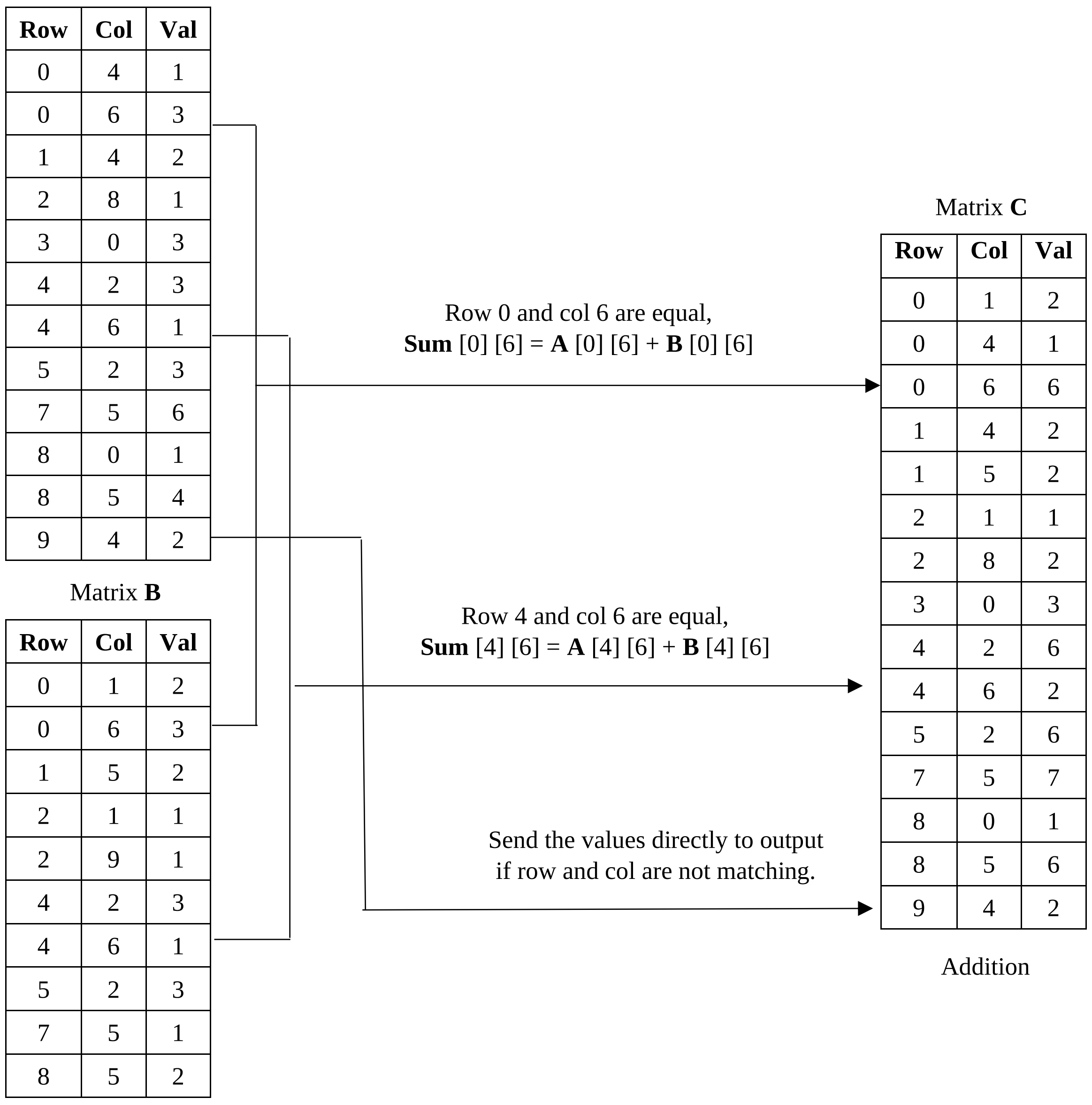
Consider two matrices **A** and **B** of size 10×10 shown in Figure 26. The first matrix **A** has nine nonzero values, and second matrix **B** has ten nonzero values. But the location of the nonzero values in both the matrices are not the same. They are distributed randomly and using the algorithm designed sparse matrix addition operation is implemented.

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	4	0	0	0	0
0	0	0	0	2	0	0	0	0	0

Matrix **A**

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	1	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	0	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

Matrix **B**



Matrix **A**

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
8	5	4
9	4	2

Matrix **B**

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

Matrix **C**

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

Addition

Row 0 and col 6 are equal,
Sum [0] [6] = **A** [0] [6] + **B** [0] [6]
Row 4 and col 6 are equal,
Sum [4] [6] = **A** [4] [6] + **B** [4] [6]
Send the values directly to output
if row and col are not matching.

Figure 26. Sparse Matrix Addition Operation Methodology

Figure 27 shows how the nonzero values are stored in separate arrays in terms of a row, col, and value. And explains how the addition operation is executed depending on the row and col arrays of the two-input matrix **A** and **B**. The operation is illustrated in detail in Figure 27 how the output sum is calculated.

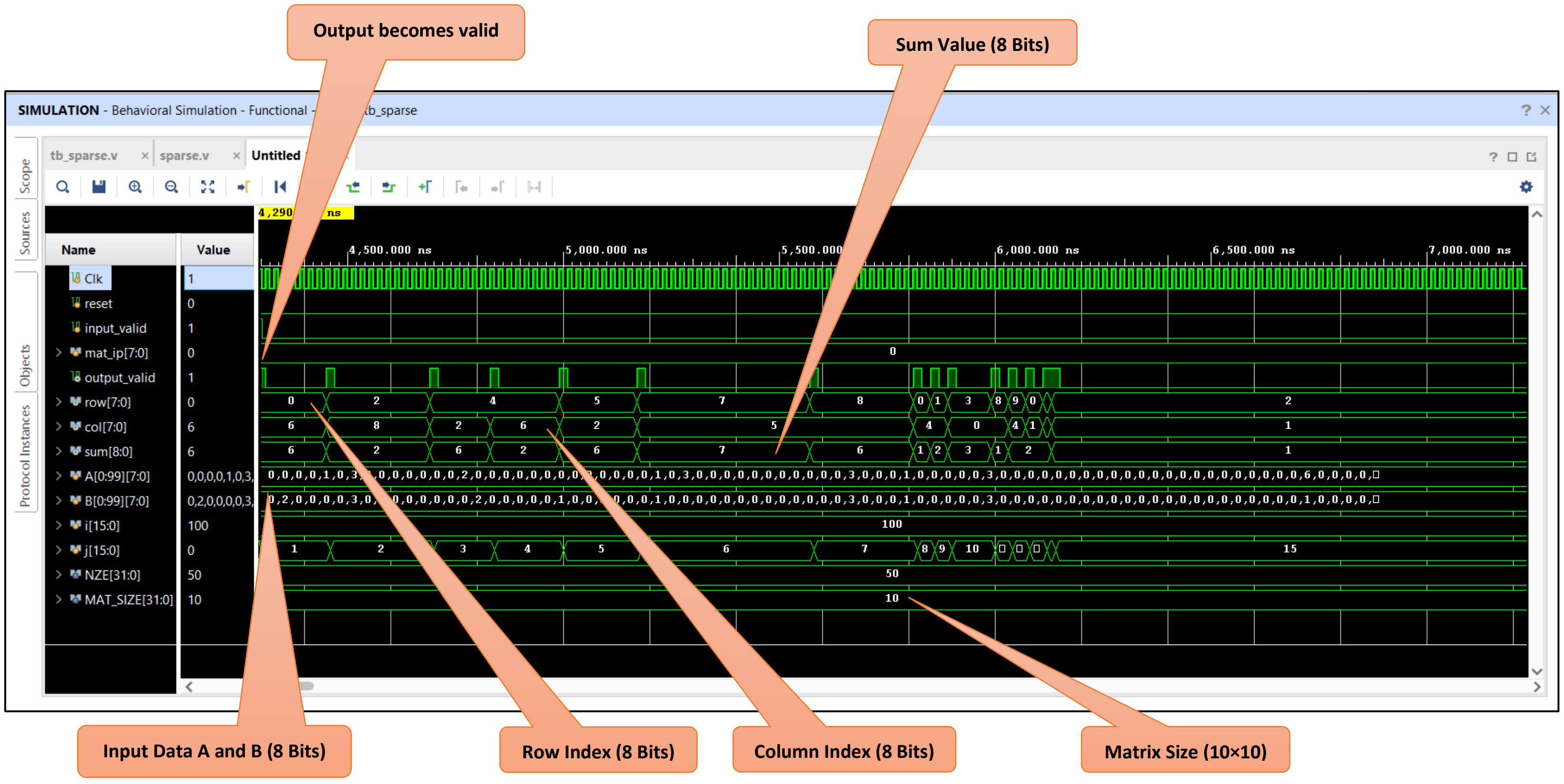


Figure 27. Design Simulation: Sparse Matrix Addition Operation

Figure 27 shows the simulation output of the input matrices $\mathbf{A}$ and $\mathbf{B}$ which are stored in three separate arrays $\boldsymbol{A\_sr}$, $\boldsymbol{A\_sc}$, $\boldsymbol{A\_sv}$ and $\boldsymbol{B\_sr}$, $\boldsymbol{B\_sc}$, $\boldsymbol{B\_sv}$ respectively. This represents how the nonzero values are checked from the input matrix and are being stored in memory. The second part of the shows the output sum calculated from the stored nonzero values accordingly.

Figure 28 captures the schematic diagram of the System-Level Optimized Design of Sparse Matrix Arithmetic Operation Design Engine. The main computation core is designed with IO blocks, memory and register for holding the data. Figure 29 shows the schematic diagram of Input available Interconnected Design. The diagram highlights the clock being reset and made available when the Input bit is ready. The design is capable irrespective of the number of bits of input elements (i.e.) either 8bit, 16bit, or 32bit. Figure 30 shows the schematic diagram of the Interconnected Design when output becomes valid, when the required arithmetic operation is performed for every clock cycle, depending upon the input valid in a detailed picture.

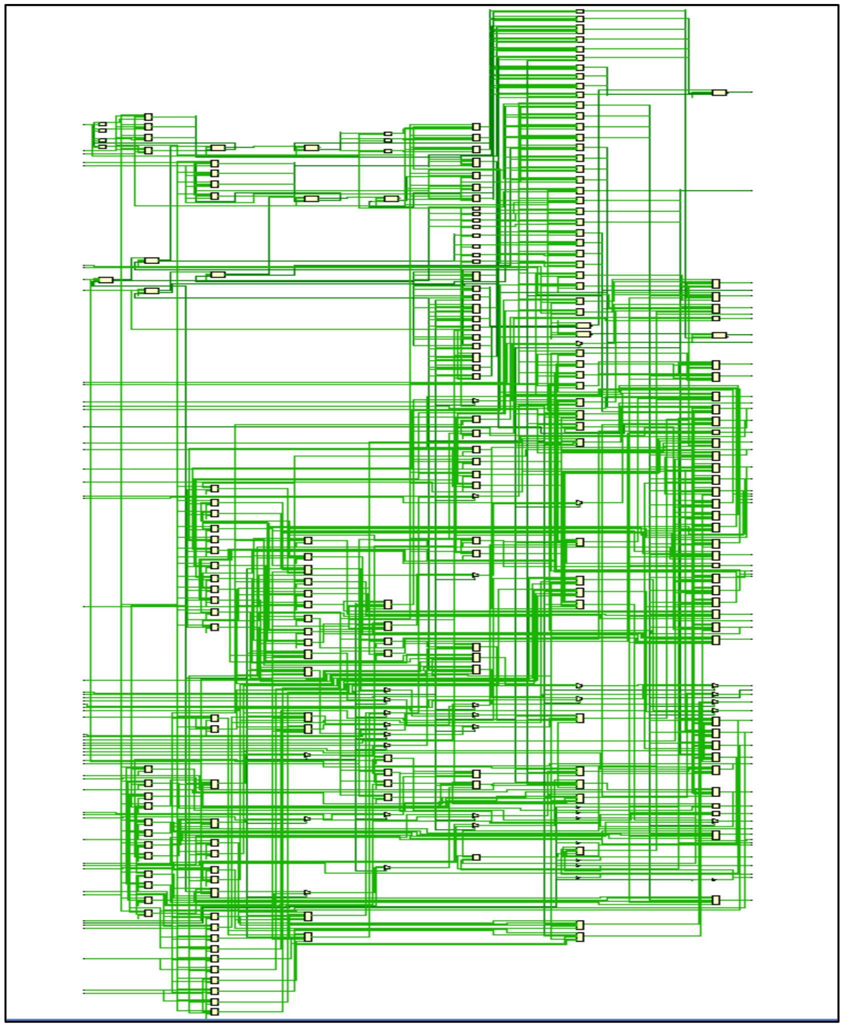


Figure 28. Implemented Design: Arithmetic Operation Engine

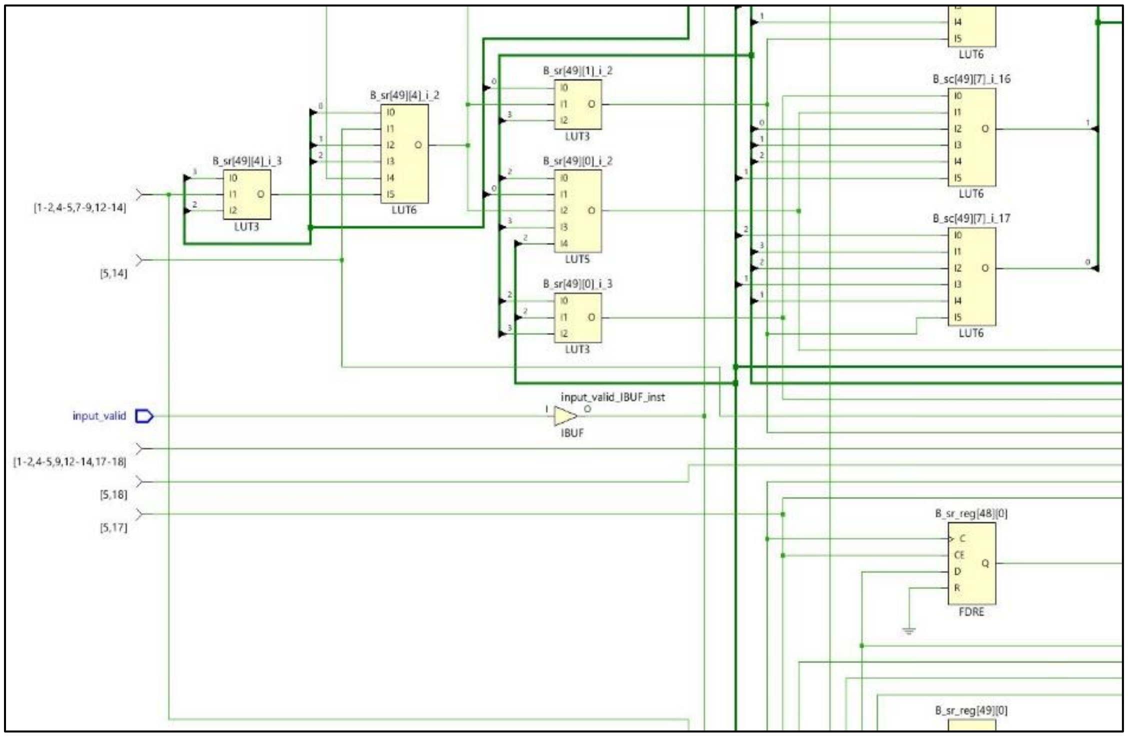


Figure 29. Interconnect Design: Input Valid Schematic

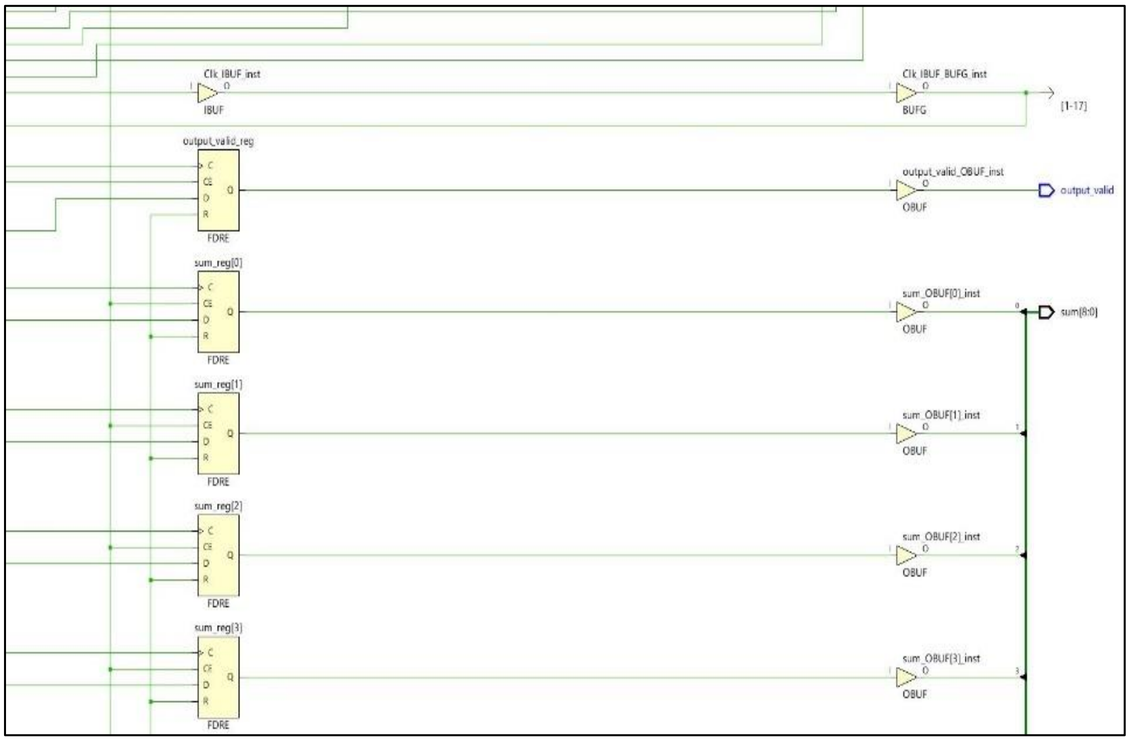


Figure 30. Interconnect Design: Output Valid Schematic

4.1.2 Sparse Matrix Multiplication. Matrix multiplication is a fundamental operation of linear algebra. It is a primitive operation in many data-analytic, graph analytic algorithms and algebraic multigrid methods. So far, many algorithms have been developed for optimizing the performance depending on the objective[34][35]. But achieving high performance for sparse matrix multiplication is quite challenging[36]. Compared to standard algorithm, sparse algorithms attempt to handle only the non-zero elements available in the matrix to remove multiplications and additions of zeros to improve performance. There are many reasons why achieving high performance with this operation is challenging because of:

- **Low Arithmetic Intensity**

The arithmetic intensity is the ratio of the number of arithmetic operations to the number of data elements accessed for computing the product [37].

- **Index Matching**

The index-matching problem is because the structure of the sparsity of the resultant matrix is unknown, and it is not possible to locate the product of $\boldsymbol{a_{i,j}}$ with $\boldsymbol{b_{i,j}}$ for the resultant $\boldsymbol{c_{i,j}}$.

There have been several approaches proposed, but each approach requires additional data access and computational problems.

- **Load Balancing**

Usually, sparse matrices that arise in practice exhibit non-uniformity and are irregular in their sparsity structure, it can be inferred those different matrices will be requiring a different distribution of work.

Consider two matrices **A**= [$\boldsymbol{a_{i,j}}$] and **B**= [$\boldsymbol{b_{i,j}}$] with size $\boldsymbol{M \times N}$ and $\boldsymbol{N \times L}$ respectively. The resultant multiplication of matrices **A** and **B** will be **C= [**$\boldsymbol{c_{i,j}}$**]**, with size $\boldsymbol{M \times L}$ as given below in equation

$$\boldsymbol{c_{i,j} = \sum_{k=1}^{N} a_{i,k} . b_{k,j}}$$

where **i** =1, 2…, M and **j**=1, 2…, L.

Sparse matrices are stored in a specific storage format taking advantage of the sparsity of the matrices[38][39]. Due to storage of nonzero values, matrix multiplication is no longer a straightforward operation. The column address of the current row being multiplied must correspond with existing row address of the other matrix. If there is a match between the two matrices multiplied, the corresponding values can be multiplied together. It operates by multiplying each nonzero element of row **A** with each nonzero element of column **B** and then repeats the process for every row and column of the matrix. Figure 31 shows the hardware architecture of the design component with a control unit, multiply and accumulate unit.

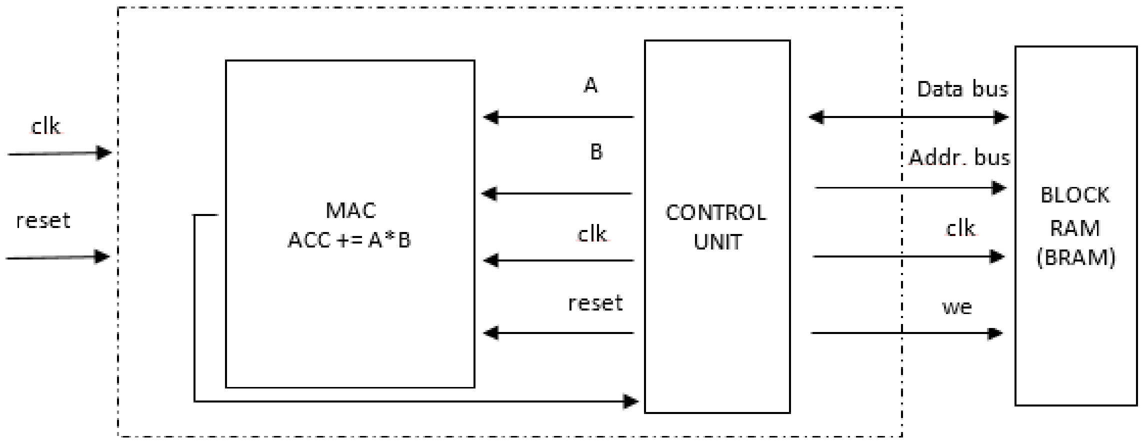


Figure 31. Matrix Multiplication Hardware Architecture

The proposed architecture in this thesis is based on minimizing the hardware resources utilization in the implemented design. The design can accomplish high performance with low execution time for large, irregular sparse matrices by reducing the number of adders and multipliers significantly. The values are arranged in streams which is a group of data like arrays permitting efficient parallelism and it maps well with the FPGA logic. This exploits data which benefits the architecture for implementing matrix multiplication on hardware device. A larger matrix will contain large number of multiplication and addition, so an adequate software for a computation we will have limited capacity. However, this proposed architecture minimizes the computation time independent of the matrix dimensions. The multiplication computation will be serial, indexing the nonzero from the matrix into an input buffer streamed and multiplied by the corresponding component of the other matrix. The output values are accumulated and sent back for error analysis. Compared to the regular multiplication, this method reduces the number of multiplication and addition. Figure 32 shows the overall design flow with hardware blocks, describing how the operation is carried out and the temporary results are accumulated to the final output.

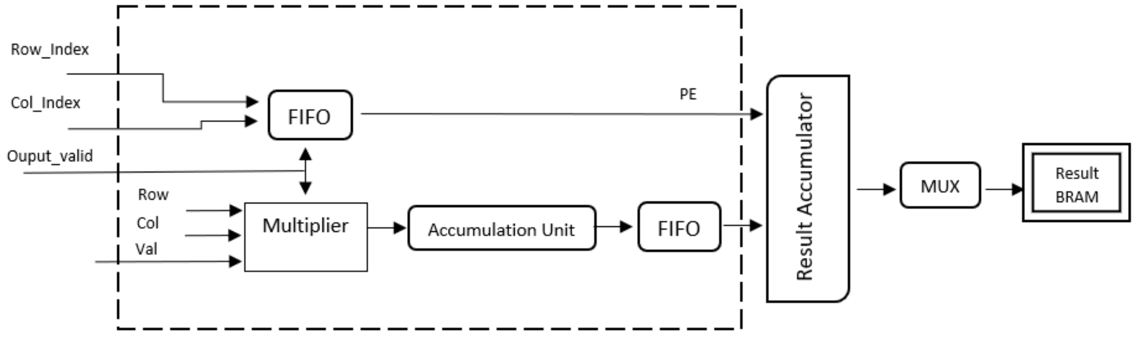


Figure 32. Overall Design Flow

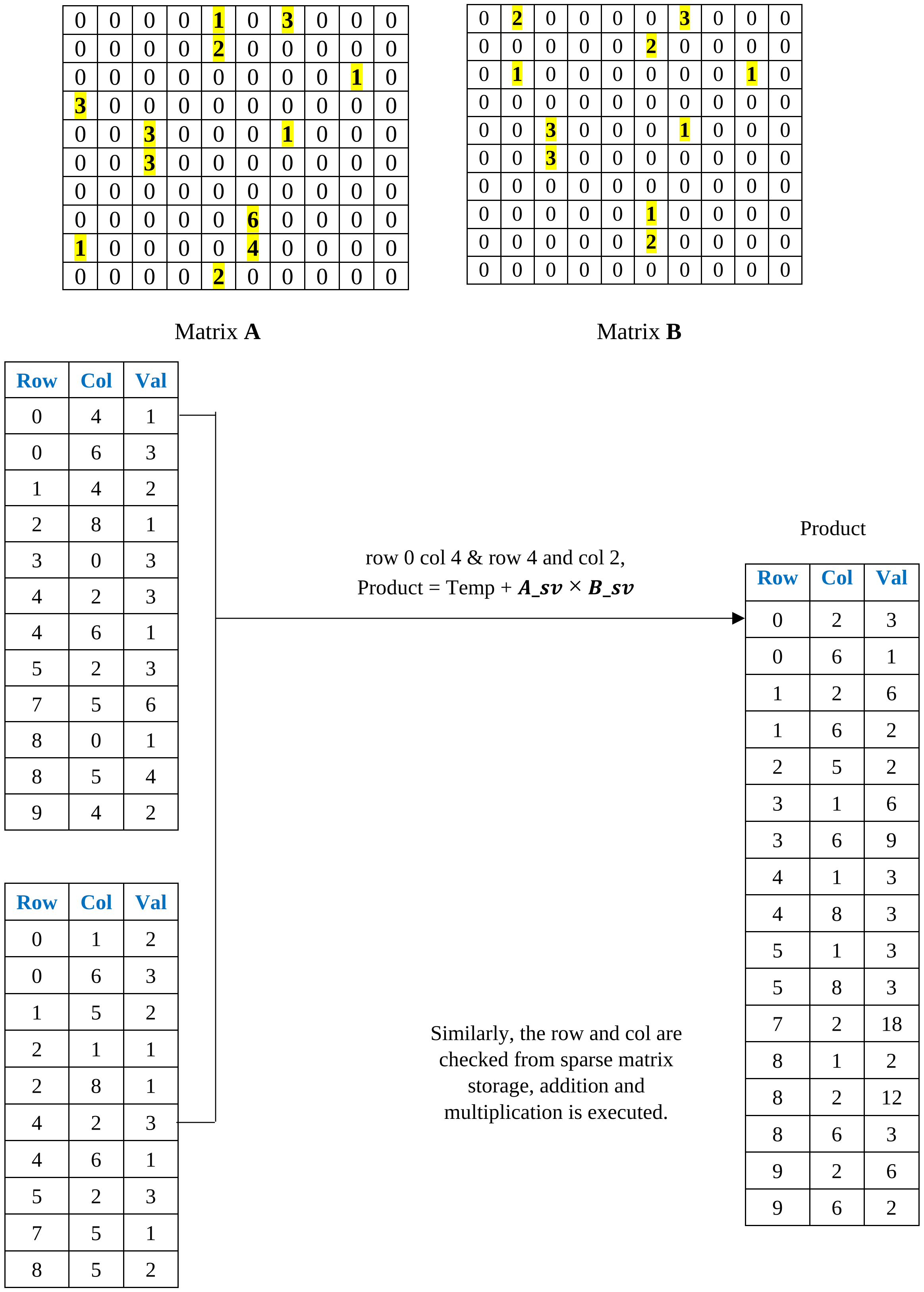
Various design techniques were considered and incorporated to optimize the performance of sparse matrix-matrix multiplication. The proposed systolic architecture consists of identical processing elements, where the number of PEs for processing depends on the size of the matrix. Each necessary multiply-accumulate operation is performed by each PE. The hardware utilization is greatly reduced as each PE operates independently with corresponding input and output thereby greatly reducing the interconnections between each PE. High throughput is achieved in the proposed architecture through pipelining and parallel processing technique by computing the intermediate product at every clock cycle. This was made possible by inserting necessary registers at appropriate places. The whole computation is divided into smaller segments which are executed in parallel, accumulating all the partial results to the Result BRAM resulting in higher frequency of operation. Every time a row index from matrix **A** is coming inside FIFO, it is compared against col index of matrix **B** present within the BRAM, so it finds a possible match for multiplication. As soon as the first element of **A** is fetched from memory and wrote in FIFO, the FSM starts comparison which makes it efficient without waiting for the entire matrix to be written in FIFO.

```
for A_index = 0: A_count
    if (A_sr > row_output)
        done_for_current m and n.
    else if (A_sr == row_output)
        //now check if a matching value is in B.
for B_index = 0: B_count
    if (B_sr > A_sc) //remember sr is always in ascending order.
        B_index <= 0; //exit from this loop.
        else if ((B_sc == n) && (B_sr == A_sc) begin //a match.
            temp = temp + A_sv × B_sv; //do multiply and add operation.
```

Listing 2. Sparse matrix multiplication algorithm

The proposed algorithm for performing matrix multiplication of sparse matrices is shown in Listing 2.

Consider two sparse input matrices **A** and **B** which are shown in figure 33. The matrix **A** consists of only nine nonzero and matrix **B** consists of only ten nonzero when the total size of the matrix elements if 100. In these cases, it would not be necessary to perform addition and multiplication on all the zero which involves a lot of hardware resources. Instead, the nonzero values are stored in sparse matrix storage format and then multiplication operation is carried out. Figure 16 illustrates how the matrix multiplication is performed based on stored nonzero values. The same procedure for sparse matrix addition is used here for sparse matrix storage. Once separate arrays are created as **ROW**, **COL**, and **VAL** the sparse matrix multiplication algorithm is performed. First the row of first matrix storage is compared with the column of second matrix storage, and the intermediate results are stored in a temporary array, and finally, the output product is sent back. When the ***output_valid*** signal is high, the "**row**", "**col**", and "**val**" are streamed into PEs which synchronizes the components with data flow. As the partial multiplication results are calculated, they are fed into an adder. Continuous computation over time, accumulates the partial sum but results are not available on next clock following the input due to the pipelined nature of the adder. Therefore, the results are temporary stored in a buffer until next result is available to be added with.



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	**4**	0	0	0	0
0	0	0	0	**2**	0	0	0	0	0

Matrix **A**

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	**1**	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	0	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

Matrix **B**

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
8	5	4
9	4	2

row 0 col 4 & row 4 and col 2,
Product = Temp + $\boldsymbol{A\_sv} \times \boldsymbol{B\_sv}$

Product

Row	Col	Val
0	2	3
0	6	1
1	2	6
1	6	2
2	5	2
3	1	6
3	6	9
4	1	3
4	8	3
5	1	3
5	8	3
7	2	18
8	1	2
8	2	12
8	6	3
9	2	6
9	6	2

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

Similarly, the row and col are checked from sparse matrix storage, addition and multiplication is executed.

Figure 33. Sparse Matrix Multiplication Methodology

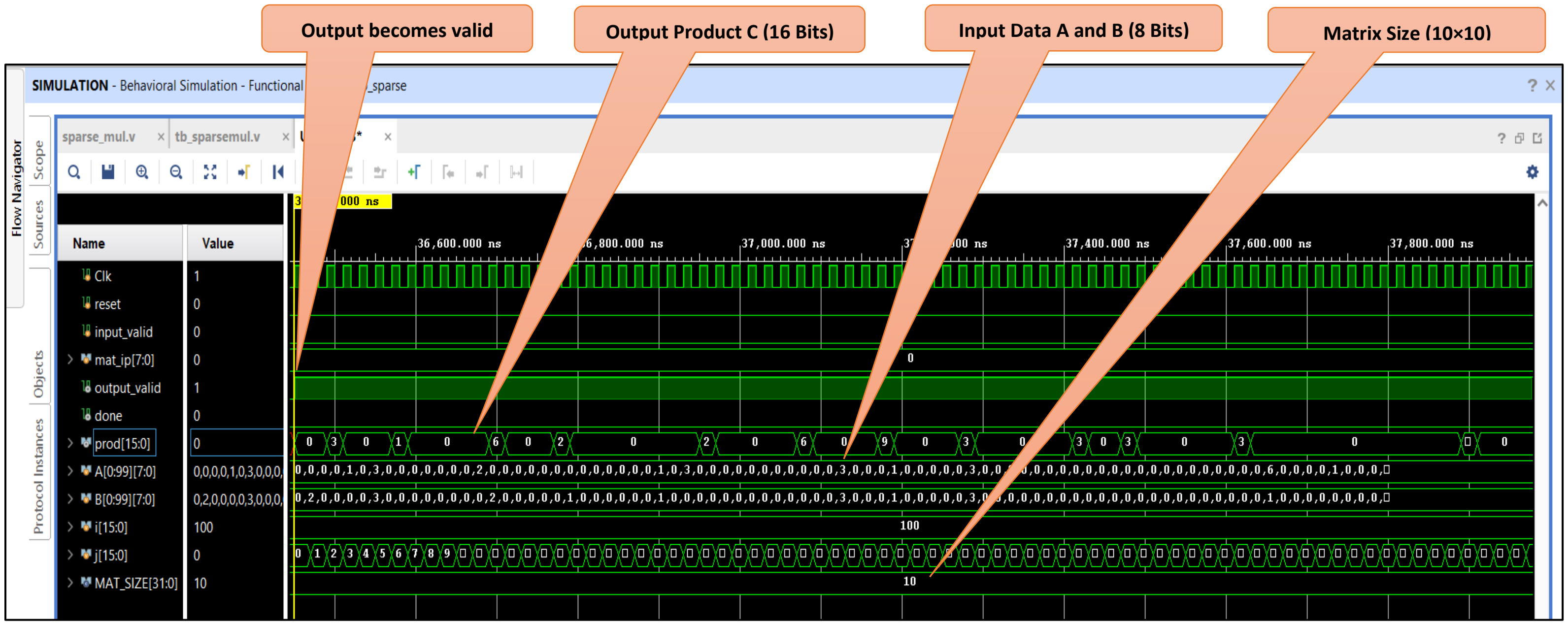


Figure 34. Design Simulation: Sparse Matrix Multiplication

Figure 34 shows the simulation results of the proposed matrix multiplication algorithm in Vivado design suite. The product is calculated from the nonzero values stored from the given input matrix after performing the storage of the sparse matrices. The product is obtained when the ***output_valid*** signal is high. From the figure, it can be inferred that the results obtained are equal to the result from regular matrix multiplication algorithm. The results are also verified with the MATLAB results; to ensure we have achieved the correct results.

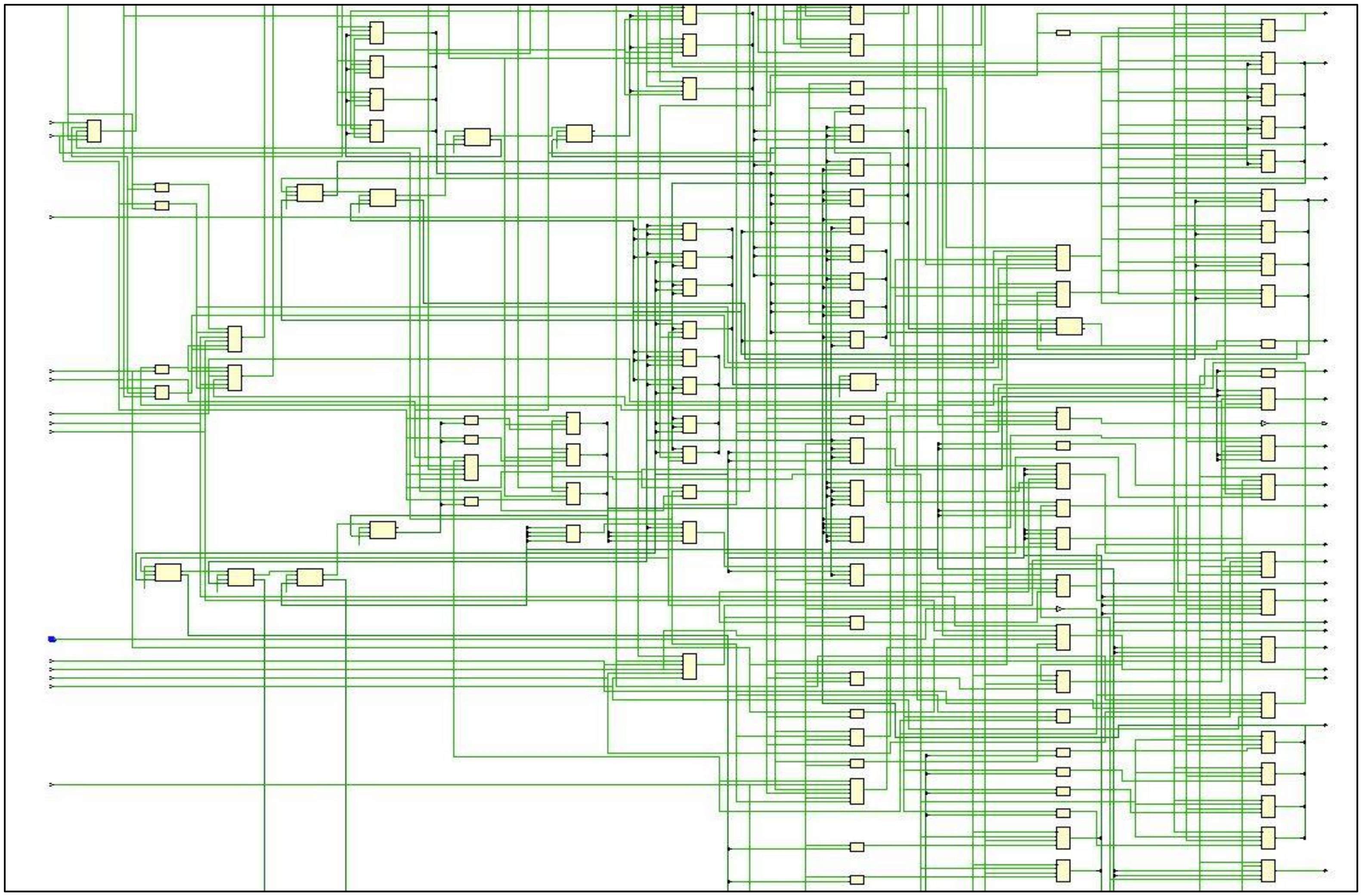
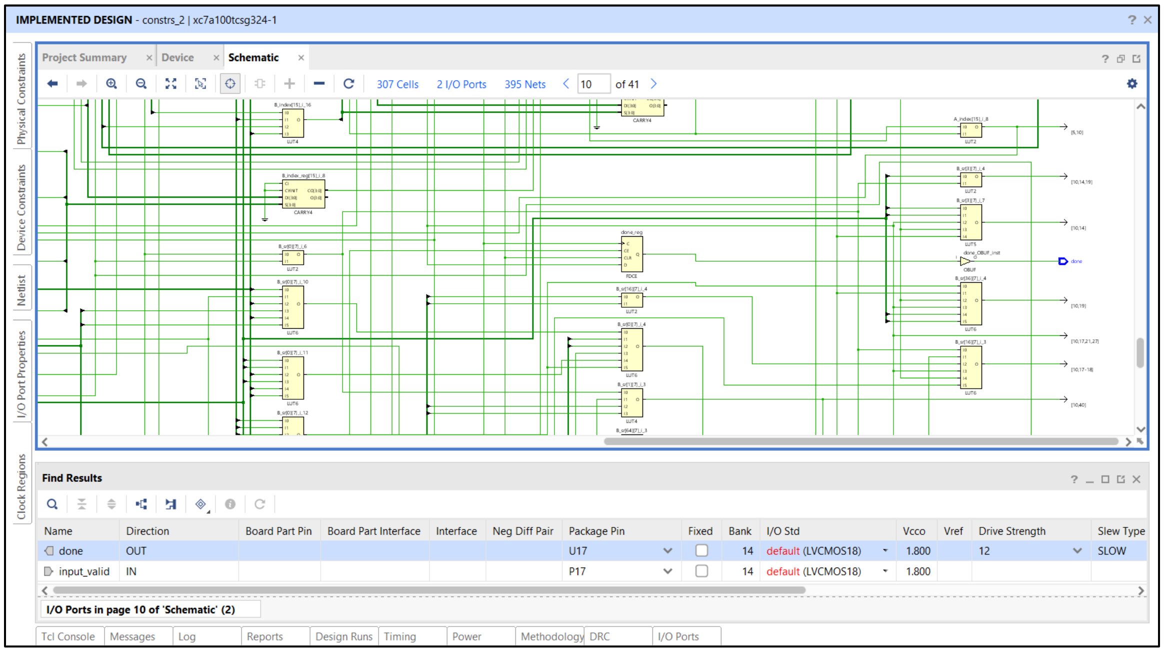


Figure 35. Elaborated Implemented Design: Sparse Matrix Multiplication

Figure 35 captures the Elaborated Implemented Design of Sparse Matrix Multiplication Operation from Vivado Design Suite. The diagram highlights the number of interconnects which are being used between the processing elements and memory controller for calculating and storing the intermediate data which is later processed for output.



Name	Direction	Board Part Pin	Board Part Interface	Interface	Neg Diff Pair	Package Pin	Fixed	Bank	I/O Std	Vcco	Vref	Drive Strength	Slew Type
done	OUT					U17	☐	14	default (LVCMOS18)	1.800		12	SLOW
input_valid	IN					P17	☐	14	default (LVCMOS18)	1.800			

I/O Ports in page 10 of 'Schematic' (2)

Figure 36. Interconnect Design: Sparse Matrix Multiplication

Figure 36 shows the Interconnect Design for Sparse Matrix Multiplication Operation which highlights the logic and cells involved in fetching the data when ***input_valid*** becomes available with high clock and when the required operation is done. Number of registers are involved, in storing the intermediate results which are accumulated for calculating the output.

The effectiveness of the algorithm varies depending upon the sparsity of the matrix, as the number of nonzero increases, the number of calculations also increases. The algorithm involves additional temporary registers to avoid overhead issues, which is negligible as comparatively single clock cycle is employed for each data whereas for a floating-point multiplication or addition operation several clock cycles are needed depending on the amount of data. Each sub processing elements generates its own matrix in a separate area in the local memory to avoid write conflicts.

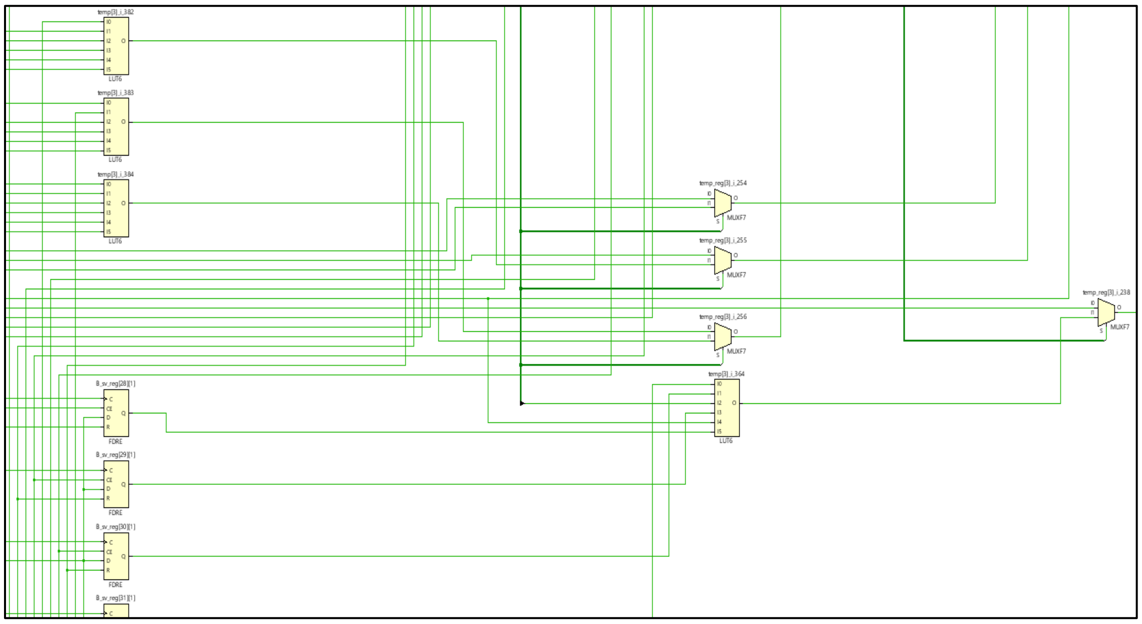


Figure 37. Interconnect Design: Temporary Registers for Intermediate Output

Figure 37 shows a part of the schematic for a single PE where the interconnects have the necessary register needed. In the proposed design, almost 60% of the total computation time are the operators and temporary registers which occur at every step or even multiple time per time step, making it hard to optimize. The number of non zeros of the matrix usually dominates the memory overhead, hence a precise number of register allocation is impossible before real execution. To achieve load balancing, each PE is partitioned into multiple sub-PE for extra irregularity to cut down the computational overhead throughout every stage.

4.2 Matrix Decomposition

Numerous engineering and machine learning applications rely primarily on matrix decomposition due to rapid development in the field of Mathematics and Computation [40]. In linear algebra, matrix decomposition is decomposing a matrix into a product of two matrices. Matrix decomposition provides an efficient means to compute the matrix inverse. Matrix inverse has several valuable applications in engineering practice, which also provides a means for evaluating system condition. The computational complexity indicated how the number of operations scales with the size of the problem data.

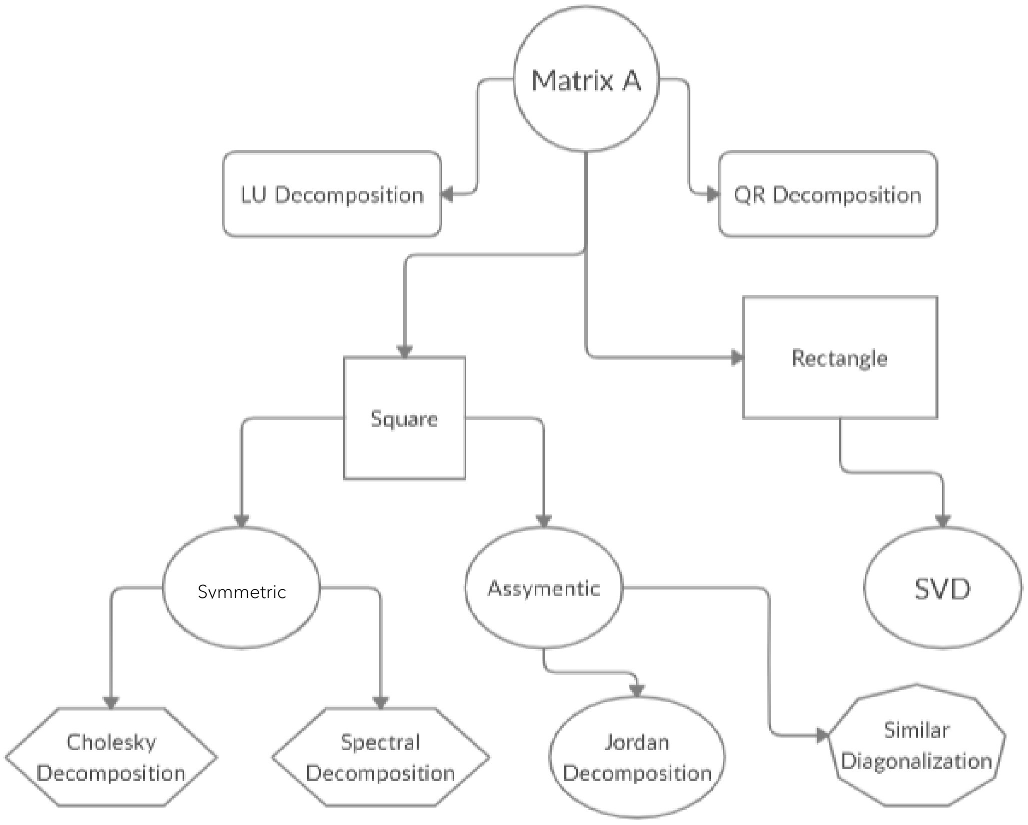


Figure 38. Matrix Decomposition

Figure 38 shows the different types of decomposition available depending upon the property of the matrix.

4.2.1 LU Decomposition.

4.2.1 LU Decomposition. LU decomposition is widely used in numerical analysis and engineering science [41]. It factors a matrix as a product of lower triangular matrix (**L**) whose diagonal elements are equal to 1, and all the elements above are equal to 0; and an upper triangular matrix (**U**) whose elements below the diagonal are equal to 0. If **A** is a square matrix, LU decomposes **A** with proper row and/or column orderings or permutations into two factors.

$$\boldsymbol{A = LU}$$

$$\begin{pmatrix} \boldsymbol{A_{11}} & \boldsymbol{A_{12}} & \boldsymbol{A_{13}} \\ \boldsymbol{A_{21}} & \boldsymbol{A_{22}} & \boldsymbol{A_{23}} \\ \boldsymbol{A_{31}} & \boldsymbol{A_{32}} & \boldsymbol{A_{33}} \end{pmatrix} = \boldsymbol{P} \begin{pmatrix} \mathbf{1} & \mathbf{0} & \mathbf{0} \\ \boldsymbol{L_{21}} & \mathbf{1} & \mathbf{0} \\ \boldsymbol{L_{31}} & \boldsymbol{L_{32}} & \mathbf{1} \end{pmatrix} \times \begin{pmatrix} \boldsymbol{U_{11}} & \boldsymbol{U_{12}} & \boldsymbol{U_{13}} \\ \mathbf{0} & \boldsymbol{U_{22}} & \boldsymbol{U_{23}} \\ \mathbf{0} & \mathbf{0} & \boldsymbol{U_{33}} \end{pmatrix}$$

LU decomposition is a direct method that can solve large systems of linear equations that arises from many essential application areas like circuit simulation, power networks, structural analysis, etc. To ensure stability during LU decomposition, pivoting operations are performed to remove zero elements from the diagonal of matrix **A**. Without proper pivoting, the decomposition may fail to materialize. Partial Pivoting refers to the proper permutation in rows or columns for LU decomposition. This approach is suitable for the square matrix, and it is numerically stable in practice.

$$\boldsymbol{PA = LU}$$

LU decomposition with full pivoting involves both row and column permutations.

$$\boldsymbol{PAQ = LU}$$

Where **Q** is a permutation matrix which reorders the columns of **A**.

Another useful method is the LDU decomposition,

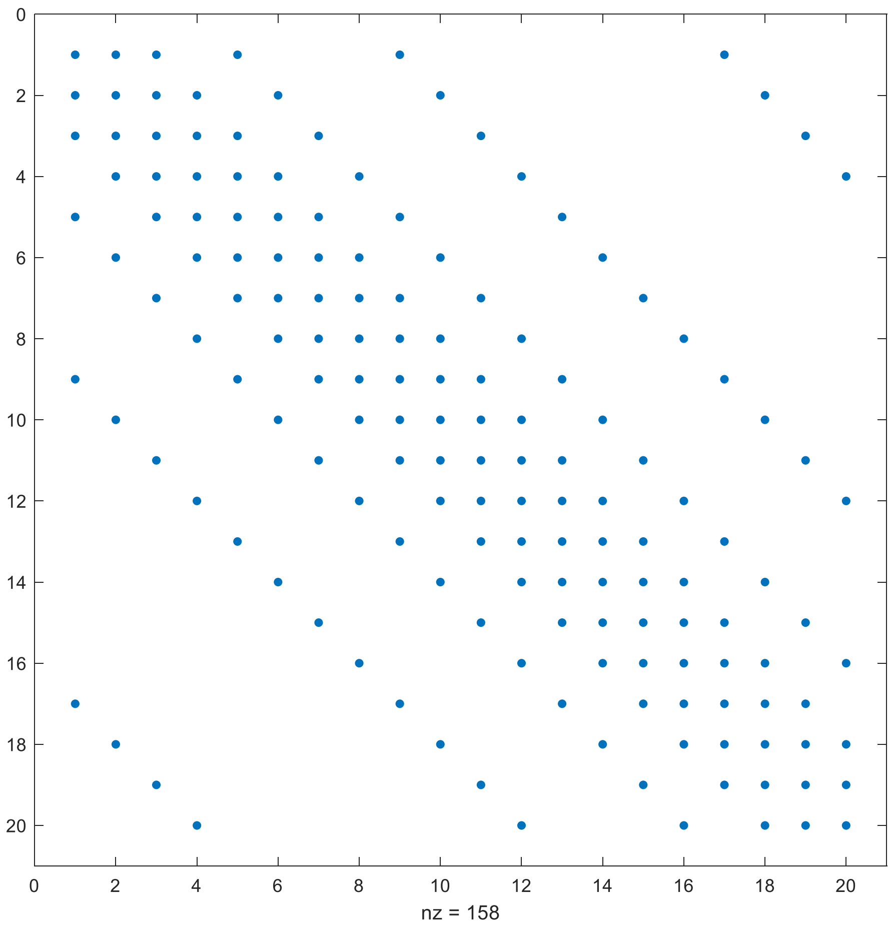
$$\boldsymbol{A = LDU}$$

Where **D** is a diagonal matrix, where all the entries on the diagonals of **L** and **U** are one.

LU decomposition introduces a permutation matrix $\boldsymbol{P}$ to ensure numerical stability leading to $\boldsymbol{PLUx} = \boldsymbol{b}$. The equation represents LU decomposition of an $\boldsymbol{n} \times \boldsymbol{n}$ matrix which has a computation time complexity of $\boldsymbol{O}(\frac{\mathbf{2}}{\mathbf{3}}\boldsymbol{n}^{\mathbf{3}})$.

LU decomposition followed by forward reduction and backward substitution technique is more stable compared to matrix inverses to solve systems of linear equations because every nonsingular matrix possesses an LU decomposition. Also, LU decomposition saves space storing either $\mathbf{L}$ or $\mathbf{U}$ matrix in the space required for input **A** matrix. On the contrast, standard matrix inversion needs much more space. The two most common methods employed are Doolittle LU decomposition algorithm and Crout decomposition algorithm. However, the optimization and generalization of the sparse matrix is required before factorization. For easy realization, all PEs are connected to a central PE along with the finite state machine where one PE can access the rows while the other PE is performing normalization.

If a matrix is nonsingular for each $\mathbf{L}$ the upper triangular matrix is unique, but the LU decomposition is not unique. There can be more than one such LU decomposition for a matrix. There is also generalization of LU to non-square and singular matrices, such as rank-level LU factorization. Figure 39 represents the matrix spy plot of LU Factorization of 20×20 Combinatorial Problem from Suite Sparse Matrix Collection.



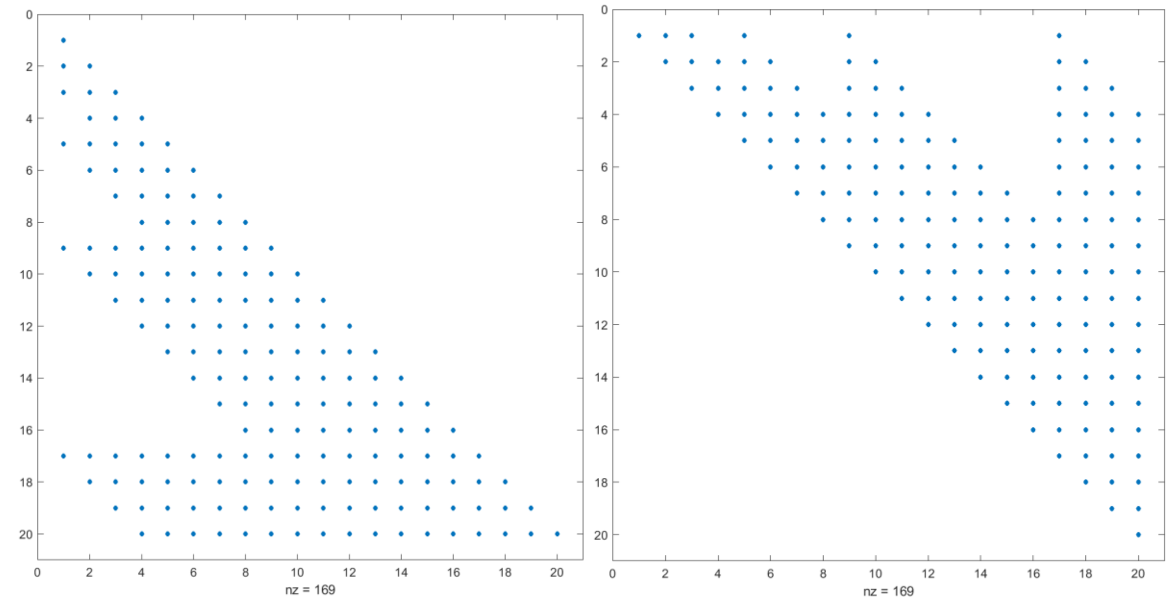


Figure 39. MATLAB representation of LU Decomposition of 20×20 Combinatorial Problem from Suite Sparse Matrix Collection

4.2.1.1 Sparse LU Decomposition Architecture. Sparse matrices compared with regular matrices can benefit from algorithms that reduce the number of operations which is required to calculate **L** and **U**. But the disadvantage involves sparse methods suffer from irregular computation patters as it is dependent on the nonzero structure of the matrix.

The proposed algorithm implements a row-wise, right-looking form of Gaussian elimination with partial row pivoting. The design of the hardware algorithm can be broken into partitions. The control system is implemented as a Finite State Machine (**FSM**), which tracks the progress of the units for synchronization. The pivoting and logic implement performs the necessary computations required for sparse LU decomposition. In addition, the last partition handles the sparse matrix storage and retrieval for the pivot search. The approach for sparse LU decomposition consists of the following operation: Pivoting strategy when **A** has nonzero entries, which are at fill-up locations. Symbolic decomposition, which estimates the memory requirements for **L** and **U** factors. Numerical calculation, which is computed using Gaussian elimination.

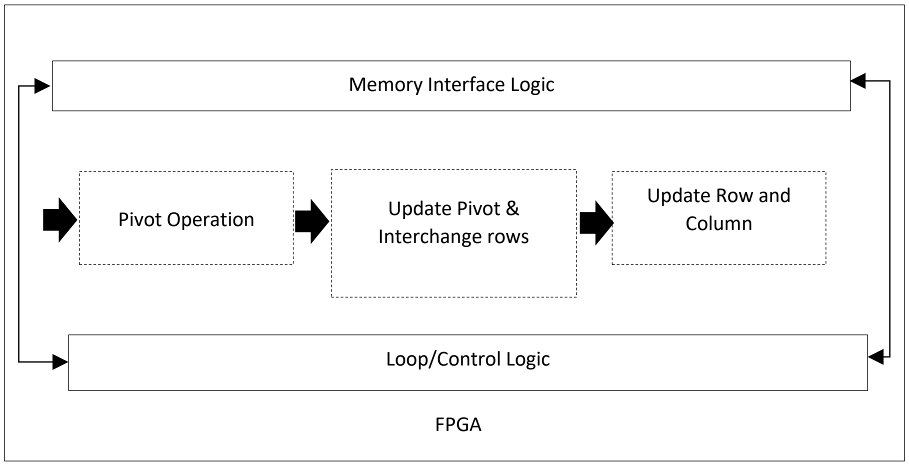


Figure 40. Control Logic for LU Decomposition Hardware

For maximizing the performance, LU hardware is designed to focus on maintaining regular computation and memory access pattern. Figure 40 shows a block diagram of the proposed hardware algorithm. The control and memory access handle the operations performed for decomposing the matrix. The design ensures the memory will have enough space to store the values. The performance of LU decomposition of the sparse matrix depends heavily on the quality of the placement tool. The initial design algorithm is inspired from Doolittle and the right-looking algorithm for sparse LU decomposition.

A. Pivot Operation

When decomposition is executed in parallel, it often tries to avoid pivoting using threshold pivoting or static pivoting beforehand. So right-looking algorithm is implemented for sparse LU decomposition including pivoting. This design produces one column of **L** and one row of **U** simultaneously and is referred by the order of loops[41]. The initial step of partial pivoting is performed by choosing a specific element from the column of **A**. To perform pivoting operation, the design includes usage of lookup tables and memory pointers to keep track of the memory mapping. It conducts pivot search for each matrix elimination step. Index pointers are created for each pivoting to store the row and column physical address accordingly. These physical addresses are then used for fetching the values from memory. These values as they arrive are sequentially checked for the absolute maximum values with index. Using a register, it gets stored as pivot element. The minimum amount of memory utilized is proportional to the size of the matrix. Once the pivoting is complete, an update is sent back to lookup tables.

After choosing the pivot, the specific row is swapped with the current row **j** and **i** being collected in the permutation matrix. To perform full pivoting, one would choose a pivot for the entire matrix[37]. The algorithm yields $\boldsymbol{LU} = \boldsymbol{PA}$, where the matrix overwrites **A** with $\boldsymbol{LU} - \boldsymbol{I}$, $\boldsymbol{I}$ is an identity matrix. The first half of the algorithm will be triangular solving, leaving behind the pivoting and scaling. In the case of sparse, it will be inefficient for swapping rows and due to having a single unreduced row or column full pivoting is not easily achievable.

B. Update Pivot & Interchange rows

The Processing Elements in Update State will be responsible for computing the core computation of the right looking algorithm method. This logic performs normalization before elimination for the pivot values of row and column requested from memory. The necessary data such as pivot index, values and column are inferred from the previous state. The updates row and column values and the normalized row and column values are then stored in registers.

C. Update row and column

The remaining computations required are performed during this transition state. First, it indicates if the given row or column should be updated. Secondly, it manages the addresses of nonzero that are to be stored. This unit contains the necessary floating-point multipliers and adders for performing the required arithmetic operations. This unit operates in parallel for maximizing the utilization of all logic units. This will update the number of update logic that fits in FPGA chip. There are enough resources available in the FPGA which can accommodate all the units. The algorithm for sparse matrix LU decomposition is given below.

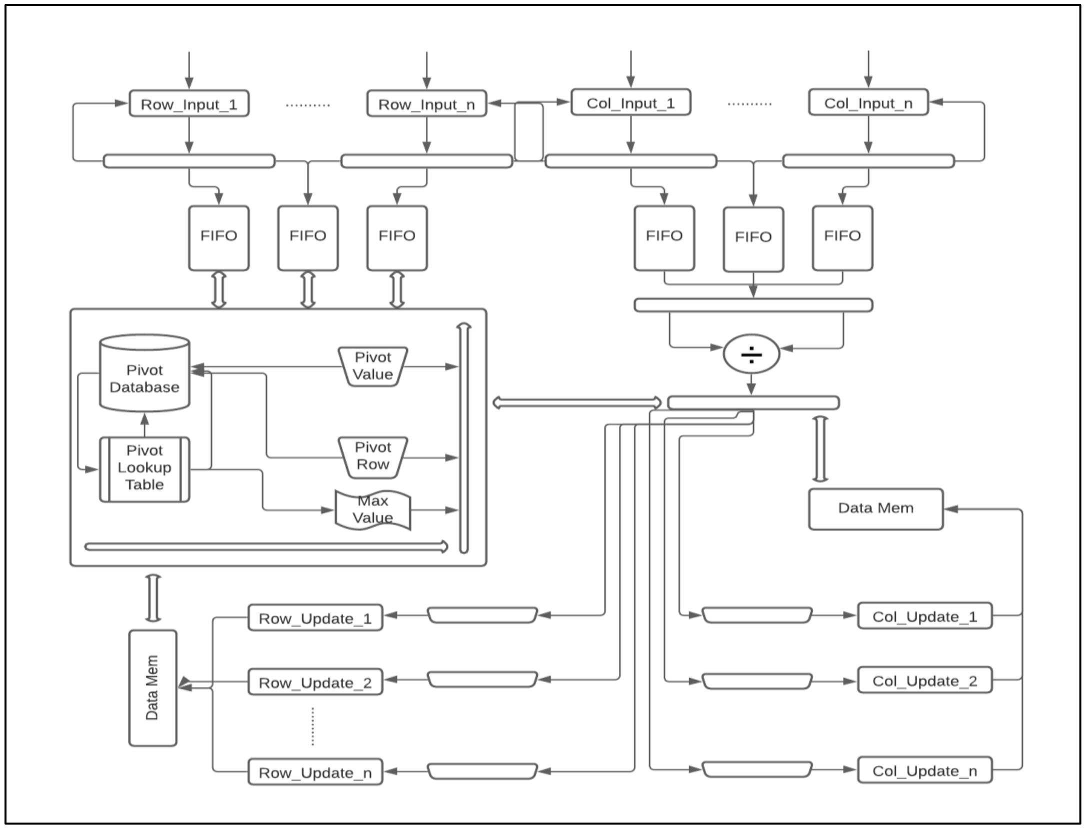


Figure 41. Block Diagram of Proposed LU Decomposition Hardware

Figure 41 shows the block diagram of the LU decomposition Hardware which consists of the Input, Output and Pivot Lookup for swapping the elements.

```
U = A
L = P = I_{n*n}

[Perform pivoting operation]
function pivot (A, P, i)
        P = choose pivot (A_i: end, i)
        if (P ≠ k) then
                SWAP (A_i, *, A_p, *)
                SWAP (P_i, *, P_p, *)
        end if
        return (A, P)
end function
[Interchanging rows in matrix]
If m≠ j
  U ([m, j], :) = U ([j, m], :)
  P ([m, j], :) = P ([j, m], :)
  If j<=2
    L ([m, j], 1: j-1) = L ([j, m], 1: j-1)
  end
end
[Update row and column entries]
for i = j+1 to n
  for j = 1 to n
    $L_{i,j} = U_{i,j} / U_{j,j}$
      for k = j+1 to n-1
        U (i, *) = U (i, *) - L (i, j) × U (j, *)
      end
  end
end
```

Listing 3. LU Decomposition Algorithm

The optimization strategies involve prefetching data when input is available to keep the control unit busy while the multiply and accumulate unit performs delayed normalization to achieve the required clock cycle to improve throughput of the overall system. Both run01 and run02 are similar in terms of data blocking, prefetching, pipeline execution units and communication with the finite state machine for computation except for the following difference.

- Each block of data is mapped into single processing element making the implementation scalable and can be computed independently in a single block.

- BRAM's are used to support larger matrices which cannot completely fit in the registers of the processing elements.
- To allow prefetching of data the finite state machine loops are restructured in subsequent iteration to fully exploit the pipelined units.
- The multiply and accumulate unit pipelined to achieve low latency and high throughput for every clock cycle.

4.2.1.2 Implementation and Error Analysis. Various arbitrary matrices with different sparsity patterns are generated using MATLAB and tested using the hardware architecture. A parameter **n** is included along with the design to get the size of the matrix to be decomposed, and the simulated waveform from Xilinx ISE design suite for a 10×10 matrix **A** is shown in Figure 42.

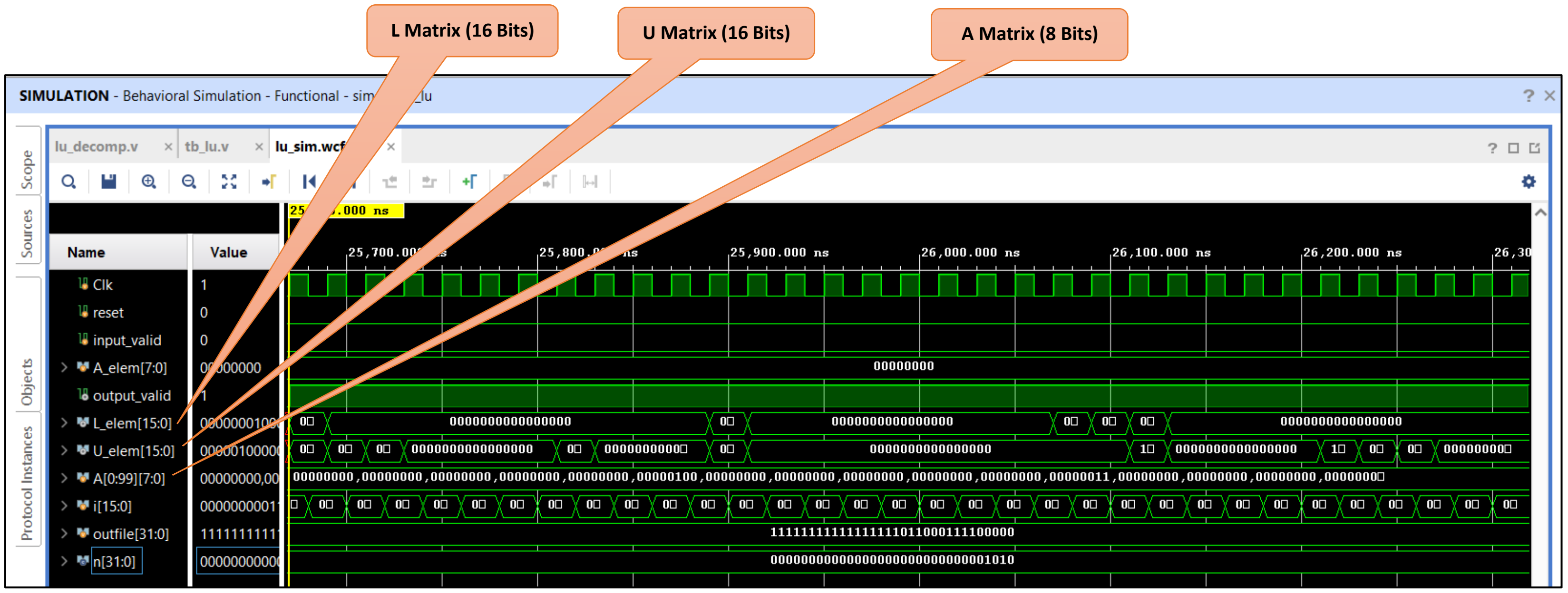


Figure 42. Design Simulation: LU Decomposition

In this case of simulation, matrix **A** is a 10×10 matrix with 10% sparsity. The ***L_elem*** is the data after LU decomposition which denotes the Lower Triangular part of the matrix, whereas ***U_elem*** is the data which denotes the Upper Triangular part of the matrix. The input matrix becomes available when the clock becomes high, meaning when ***input _valid*** is valid utilizing the maximum frequency and the operation is completed when ***output_valid*** becomes low. n denotes the size of the matrix, in this case n=10 and this is an 8-bit data

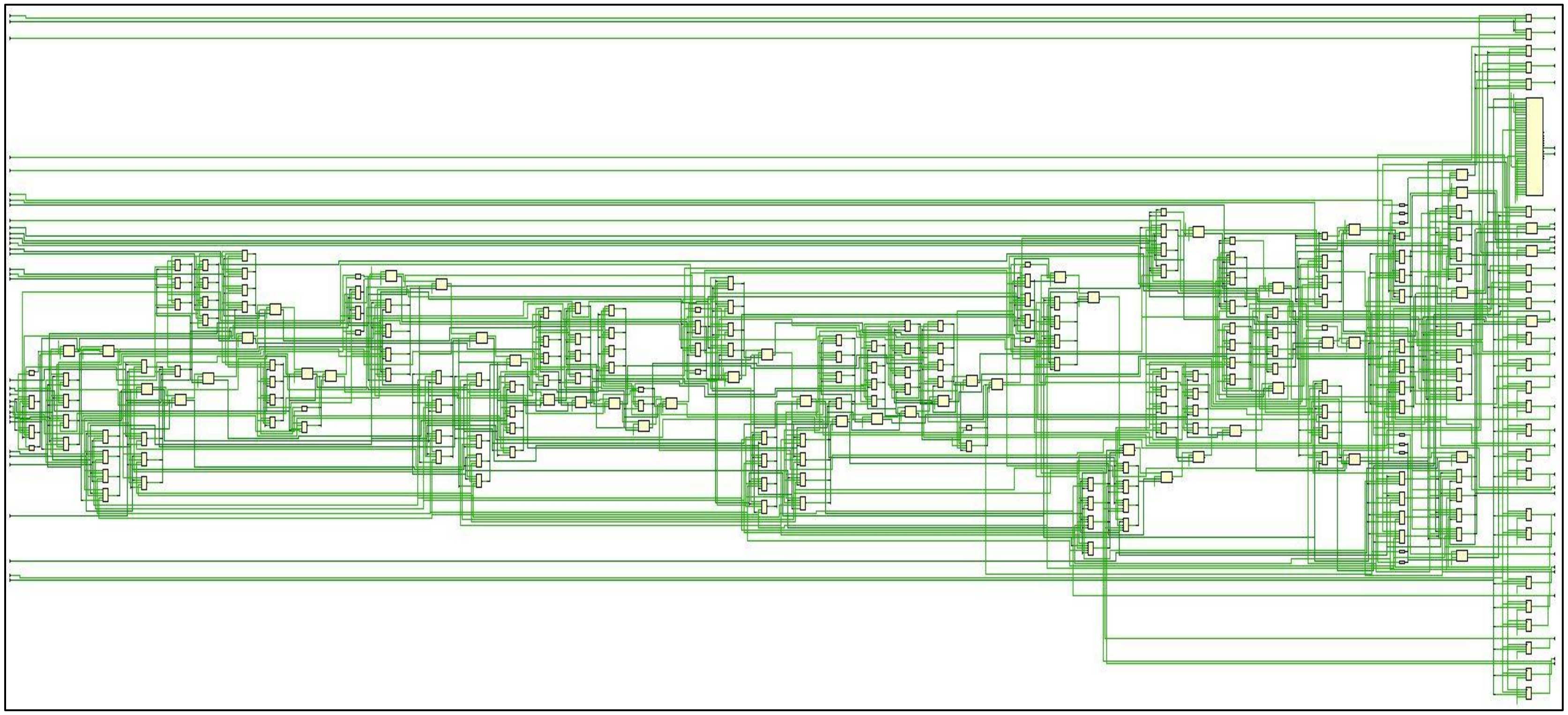


Figure 43. Implemented Design Engine: LU Decomposition

The overall implemented architecture of the LU decomposition schematic is represented in Figure 43. The proposed and implemented design is analyzed for the usage of Slice LUTs, memory, Slice Registers, and IO. The Vivado Design Tool provides the nets and logic which is used to check for all design violations, and for a better optimization.

The error analysis is carried out by comparing the software results from MATLAB and the hardware results from Vivado Design Suite. This is the precision of error for decomposing an input matrix **A** into resultant **L** and **U** matrices, respectively. The Mean Error(**ME**) is the average of all errors. The formula for calculating the Mean Error is:

$$\boldsymbol{Mean\ Error = \frac{1}{n}\sum_{i=1}^{n}|x_i - x|}$$

Where $\boldsymbol{n}$ – the number of errors, $\boldsymbol{|x_i - x|}$ – the absolute errors.

Table 4. Error Analysis for LU Decomposition Operation

10 x 10 Matrix			
Sparsity	Matrix	Min	Max
10%	L	0	0.0022
	U	-0.0087	0.0078
20%	L	-0.0049	0.0022
	U	-0.0117	0.0292
30%	L	-0.0114	0.0074
	U	-0.0566	0.0626
40%	L	-0.0144	0.0206
	U	-0.0807	0.0781
50%	L	-0.0229	0.0062
	U	0.0799	0.0643
Mean Error	**L**	**-0.01072**	**0.00772**
	U	**-0.01556**	**0.0484**

Table 4 represents the data for while decomposing a matrix of size 10x10 with varying the sparsity of the matrices from 10% to 50%. The Min and Max value of errors are calculated which is the difference between the MATLAB software results and the Vivado Hardware results. Once these are tabulated, the Mean Error is calculated from the above-mentioned formula to investigate the precision loss.

4.2.2 QR Decomposition. QR decomposition plays a vital role in computing solution of linear systems of equations, computing Eigenvalues and solving least square problems. Some of the applications of MIMO technologies and adaptive filtering require high-throughput QR decomposition for small size matrix. QR decomposition can be employed in machine learning in the automatic removal of an object from an image. To crop an image of a car from a video clip, using a single value decomposition make it relatively simple. In short by splitting a video into its individual frames, creating a matrix of vectors corresponding to each image, the decomposition allows simple separation of foreground objects from the background space. Preprocessing of QR decomposition makes the decoding in signal processing simple and to implement data detection helps to reduce the complexity of spatial multiplexing MIMO-OFDM detection. Many works have addressed the parallel hardware implementation of QR decomposition on Field Programmable Gate Arrays. It is referred to as Orthogonal matrix triangularization, which decomposes a given matrix **A** of size $\boldsymbol{m}\times\boldsymbol{n}$ into an orthogonal matrix (**Q**) of size $\boldsymbol{m}\times\boldsymbol{m}$ such that $Q^T.Q = I$ and an upper triangular matrix(**R**) of size $\boldsymbol{m}\times\boldsymbol{n}$.

$$\boldsymbol{A} = \boldsymbol{QR}$$

There are many methods for performing QR decomposition algorithms such as Givens Rotation (GR), Householder Transform (HT), Modified Gram-Schmidt (MGS) and Cholesky QR. The computation of eigenvalues is simplified using QR decomposition method.

Figure 44 shows the matrix representation of QR decomposition of 100×100 Structural Problem from Suite Sparse Matrix Collection.

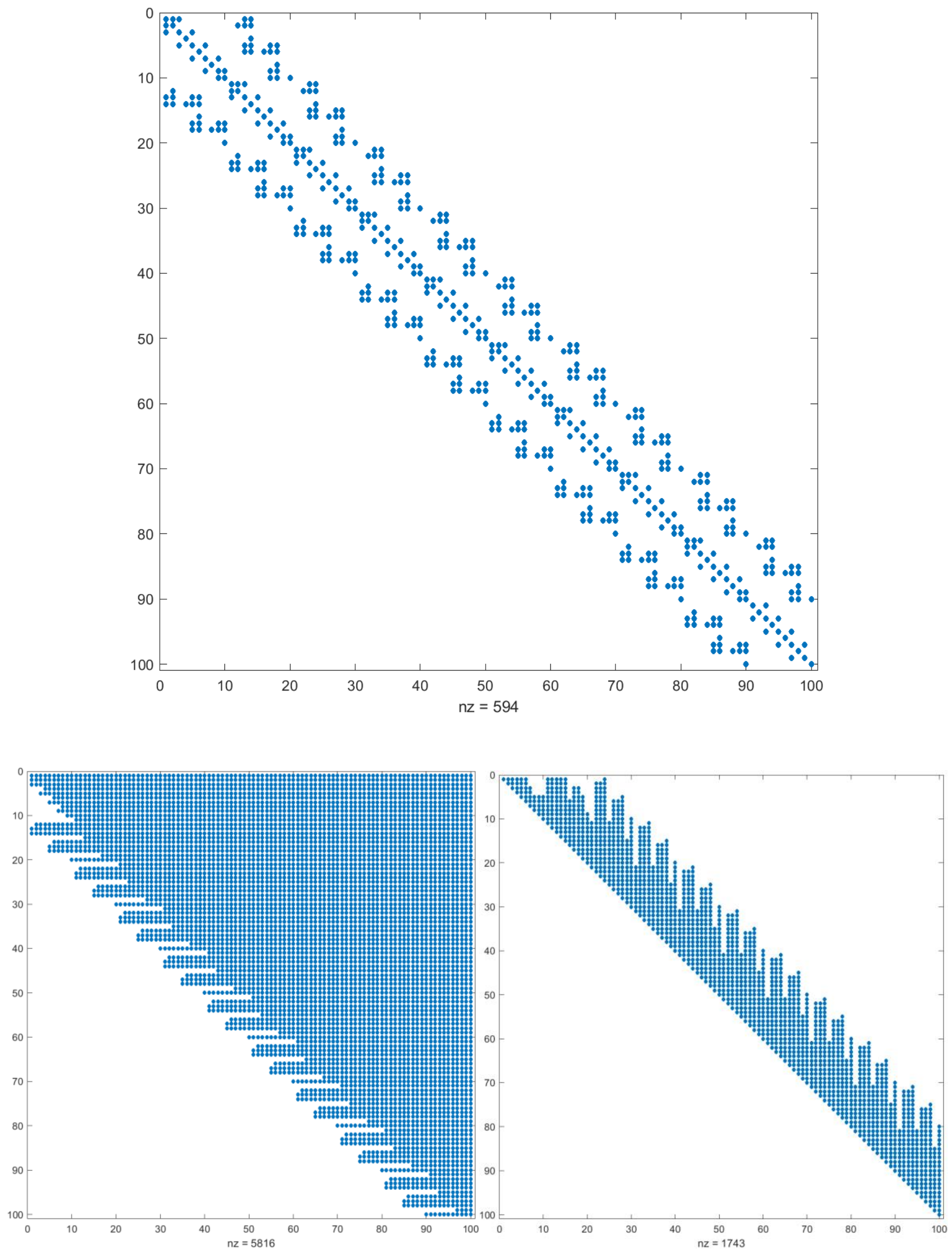


Figure 44. MATLAB representation of QR Decomposition of 100×100 Structural Problem from Suite Sparse Matrix Collection

For solving linear simultaneous equations, **A×x=b** using Gaussian elimination, elementary row transformations of the matrix **A** are applied. This is equivalent as pre-multiplying **A** by non-singular matrix **P** to solve the triangular system of equations **P×Ax = P×b** by back substitution. To improve numerical stability, Householder transformation which is orthogonal is used for QR decomposition.

4.2.2.1 Householder Transformation. Householder Transformation is a sophisticated algorithm which zeros all the elements required in a column at once. This method uses reflection method for performing zeroing operation. It performs a series of orthogonal transformations on any arbitrary matrix to convert into an upper triangular matrix. It is a linear process representing a vector through a plane containing the origin. The matrix which is transformed has the same norm as original vector.

Consider a sparse matrix **A**, with **x** representing the non-zero elements given as:

$$\boldsymbol{A} = \begin{bmatrix} \boldsymbol{x} & \boldsymbol{x} & \boldsymbol{x} & \boldsymbol{x} \\ \boldsymbol{x} & \boldsymbol{x} & \boldsymbol{x} & \boldsymbol{x} \\ \boldsymbol{x} & \boldsymbol{x} & \boldsymbol{x} & \boldsymbol{x} \\ \boldsymbol{x} & \boldsymbol{x} & \boldsymbol{x} & \boldsymbol{x} \end{bmatrix}$$

Considering the **A** matrix, we computer $\mathbf{H_1}$ such that product of $\mathbf{H_1}$ and **A** results in first column zero except for the first element as:

$$\boldsymbol{H_1 A} = \begin{bmatrix} \boldsymbol{x} & \boldsymbol{x} & \boldsymbol{x} & \boldsymbol{x} \\ \mathbf{0} & \boldsymbol{x} & \boldsymbol{x} & \boldsymbol{x} \\ \mathbf{0} & \boldsymbol{x} & \boldsymbol{x} & \boldsymbol{x} \\ \mathbf{0} & \boldsymbol{x} & \boldsymbol{x} & \boldsymbol{x} \end{bmatrix}$$

Similarly, the same procedure is carried out after multiplying it with the product and not disturbing 1st row and 1st column and zeroing all the remaining elements as follows:

$$\boldsymbol{H_2 H_1 A} = \begin{bmatrix} \boldsymbol{x} & \boldsymbol{x} & \boldsymbol{x} & \boldsymbol{x} \\ \mathbf{0} & \boldsymbol{x} & \boldsymbol{x} & \boldsymbol{x} \\ \mathbf{0} & \mathbf{0} & \boldsymbol{x} & \boldsymbol{x} \\ \mathbf{0} & \mathbf{0} & \boldsymbol{x} & \boldsymbol{x} \end{bmatrix}$$

$$H_3H_2H_1\mathbf{A} = \begin{bmatrix} x & x & x & x \\ 0 & x & x & x \\ 0 & 0 & x & x \\ 0 & 0 & 0 & x \end{bmatrix}$$

The process results in an upper triangular matrix **R**. And for any matrix **A** of size $\boldsymbol{m}$ **x** $\boldsymbol{n}$ QR decomposition can be written as follows:

$$(\boldsymbol{H_n H_{n-1} \cdots H_3 H_2 H_1})\boldsymbol{A} = \boldsymbol{R}$$

$$(\boldsymbol{H_3 H_2 H_1})\boldsymbol{A} = \boldsymbol{H^T A} = \boldsymbol{R}$$

$$\boldsymbol{Q} = \boldsymbol{H_1 H_2 \cdots H_n}$$

The Householder Transformation of a matrix with normal vector $\boldsymbol{v}$ takes the form:

$$\boldsymbol{H} = \boldsymbol{I} - \boldsymbol{2vv^T}$$

We need to build **H** from the above-mentioned mathematical calculations so that $\boldsymbol{Hx} = \boldsymbol{\alpha e_i}$for some constant $\boldsymbol{\alpha}$ and $\boldsymbol{e_1}$ = $[\mathbf{1\ 0\ 0}\ ]^{\boldsymbol{T}}$.

Since $\boldsymbol{H}$ is orthogonal, $||\boldsymbol{Hx}|| = ||\boldsymbol{x}||$ and $||\boldsymbol{\alpha e_1}|| = |\boldsymbol{\alpha}|||\boldsymbol{e_1}|| = |\boldsymbol{\alpha}|$. So $\boldsymbol{\alpha} = \pm||\boldsymbol{x}||$, the sign is selected for vector $\boldsymbol{u}$ as:

$$\boldsymbol{u} = \begin{bmatrix} \boldsymbol{x_1 + sign(x_1)||x_1||} \\ \boldsymbol{x_2} \\ \vdots \\ . \\ \boldsymbol{x_n} \end{bmatrix}$$

With his unit vector $\boldsymbol{u}$ defined as $\boldsymbol{u} = \frac{\boldsymbol{v}}{||\boldsymbol{v}||}$. The corresponding Householder transformation is:

$$\boldsymbol{H(x)} = \boldsymbol{I} - \boldsymbol{2vv^T} = \boldsymbol{I} - \boldsymbol{2}\frac{\boldsymbol{uu^T}}{\boldsymbol{u^T u}}$$

The following merits were important for considering Householder transformation for the proposed architecture:

1. Better Numerical stability compared with Gram-Schmidt.

2. Using Householder Transformation, we were able to save memory space within the original matrix **A** for **Q** or **R** matrix rather than an explicit memory space.
3. Arithmetic operations are less.

The above Householder transformation can be realized using hardware architectures for calculating performance results. The Householder Transformation is a common approach in practice as Gram-Schmidt approach causes inaccuracy in computation which may result in non-orthogonal **Q** matrix. The algorithm for computation of proposed design is provided in Listing 4.

```
$A \rightarrow n \times n$ sparse matrix
$Q \rightarrow n \times n$ Orthogonal identity matrix
$R \rightarrow n \times n$ Upper Triangular matrix
$[m, n] = size(A)$
$R = A$
$Q = I$
for k=1 to m-1
        for i=1 to i-1
            $x_{ij}=0$
        end
        for i=1 to m
            $x_{ij} = R_{ij}$
        end
g= sqrt ( $\sum_{i=0}^{m} x_i$)
$x_i=x_{ij} + g$
s= sqrt ( $\sum_{i=0}^{m} x_i$)
if $s \neq 0$
    $x = \frac{x}{s}$
    $u = 2 * R' * x$
    $R = R- x * u'$
    $Q = Q-2*Q*x*x'$
end
```

Listing 4. QR Decomposition Algorithm

4.2.2.2 Design Flow and Optimization. The core of the decomposition process which is shown in Figure 45 is optimized in the finite state machine process while designing the algorithm. Pipelining overlaps the execution of instructions in parallel improving the performance. Pipelining refers to the parallel implementation of the proposed algorithm, which increases the throughput. For QR decomposition, multiple iterations need to be calculated, so pipelining could be implemented for better performance, which could also reduce latency. However, the resources utilized for the hardware architecture will be significant, as more resources should run in parallel.

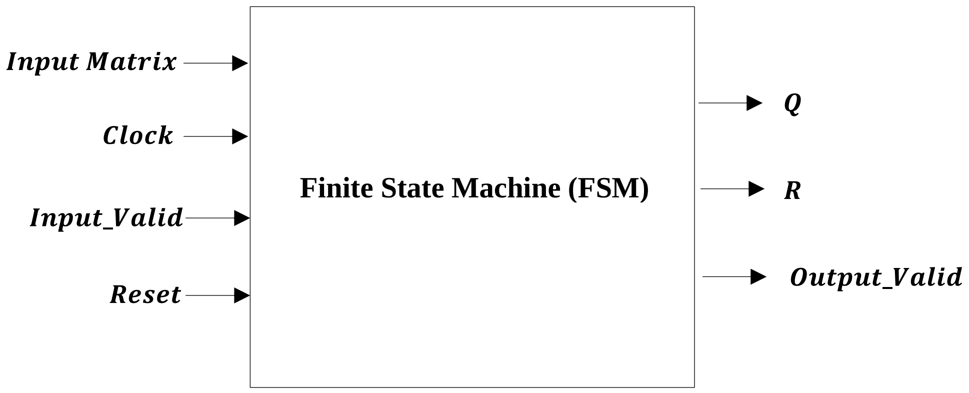


Figure 45. QR Decomposition Core

Several intensive computations are needed to be implemented for calculating QR decomposition in parallel as hardware, to avoid bottlenecks. To achieve timing results, more hardware is used resulting in a larger design area.

QR decomposition of an $\boldsymbol{n}\times\boldsymbol{n}$ matrix usually requires three times $\boldsymbol{n}\times\boldsymbol{n}$ storage space in the register, comprising the input matrix, resultant orthogonal matrix, an upper triangular matrix. Instead, once the input matrix is read decomposition takes place column by column storing the resulting **R** matrix above input matrix. So, only two times $\boldsymbol{n}\times\boldsymbol{n}$ matrix storage will be used in the architecture removing the extra storage. The usage of storage space for a 2×2 matrix is shown in Figure 46 how matrix elements are stored.

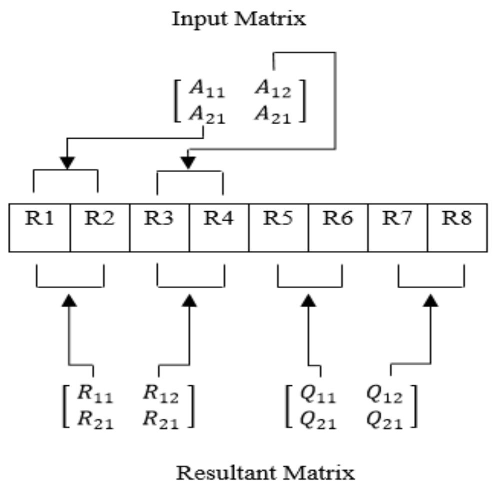

Input Matrix
$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{21} \end{bmatrix}$
R1
R2
R3
R4
R5
R6
R7
R8
$\begin{bmatrix} R_{11} & R_{12} \\ R_{21} & R_{21} \end{bmatrix}$
$\begin{bmatrix} Q_{11} & Q_{12} \\ Q_{21} & Q_{21} \end{bmatrix}$
Resultant Matrix

Figure 46. Usage of Registers for QR decomposition of 2×2 matrix

QR decomposition can be summarized in 3 steps: Preprocessing, Decomposition, and Matrix update. The preprocessing step is responsible for computation of square root norms which utilizes more hardware. The Top module is equipped with **n** multipliers, under which log(n) level of adders are employed. To save the hardware resources, the preprocessing is implemented as part of matrix update computation. The next step decomposes the matrix in which FSM-based control units are employed to synchronize the factorization. Additions are performed in parallel by a pair of adders to calculate the coefficient which is first value of the input element. A multiplier is used to multiply, which is followed by the subtraction. The last stage of updating the matrix has arithmetic calculations, and Householder Transformation process ends with subtractions. The number of matrix columns that can be held on the chip is determined as per matrix size and on-chip resources. The architectural implementation minimizes the excessive delay and processing overhead produced by the norm calculation and sorting operations. This led to more regular processing flow which increases the throughput.

An important design consideration is the calculation of square root, as it affects the precision and timing. Besides the addition, subtraction, multiplication operations for QR decomposition square root operation are mandatory for every column. Especially if the matrix size is large, the number of operations reduces the timing of the design. Considering these limitations, the square root module is designed separately along with the top module. Also, performing more iterations for finding the square root will give better results. The algorithm for square root for each column is provided in Listing 5. Coordinate Rotation Computer (**CORDIC**) algorithm which depends on the shift-add operation is used to calculate the square root operation. To increase the speed of execution multipliers are used to obtain low latency and this stage is design to work in parallel.

```
begin
for i in 0 to 15 loop
        right (0) := '1'
        right (1) := r(17)
        right (17 down to 2) := q
        left (1 down to 0) := a (31 down to 30)
        left (17 down to 2) :=r (15 down to 0)
        a (31 down to 2) := a (29 down to 0)    --shifting by 2 bit.
if ( r (17) = '1') then
        r := left + right
else
        r := left - right
end if;
        q (15 down to 1) := q (14 down to 0)
        q (0) := not r(17)
end loop
return q
```

Listing 5. Pseudo code for Square Root Algorithm

For testing the algorithm, fixed-point number representation is used as input specification. Randomly generated matrices with different sparsity patterns are used as input and a test bench is developed in MATLAB for verification. The simulation results obtained from Xilinx ISE design suite for 10×10 matrix is shown in Figure 48.

For implementing the design on FPGA, we have set the output data as a 32-bit integer. The ***dec*** and ***frac*** parameters specify the number of bits allotted for decimal and fractional part of the input. The input bit is scalable and can be reduced for smaller values of **n**. The input, and output matrix is stored in the Block RAM controllers. The architecture is synthesized and implemented to find the timing and power estimates from the Vivado design Suite. Figure 47 shows the Block Diagram of QR decomposition.

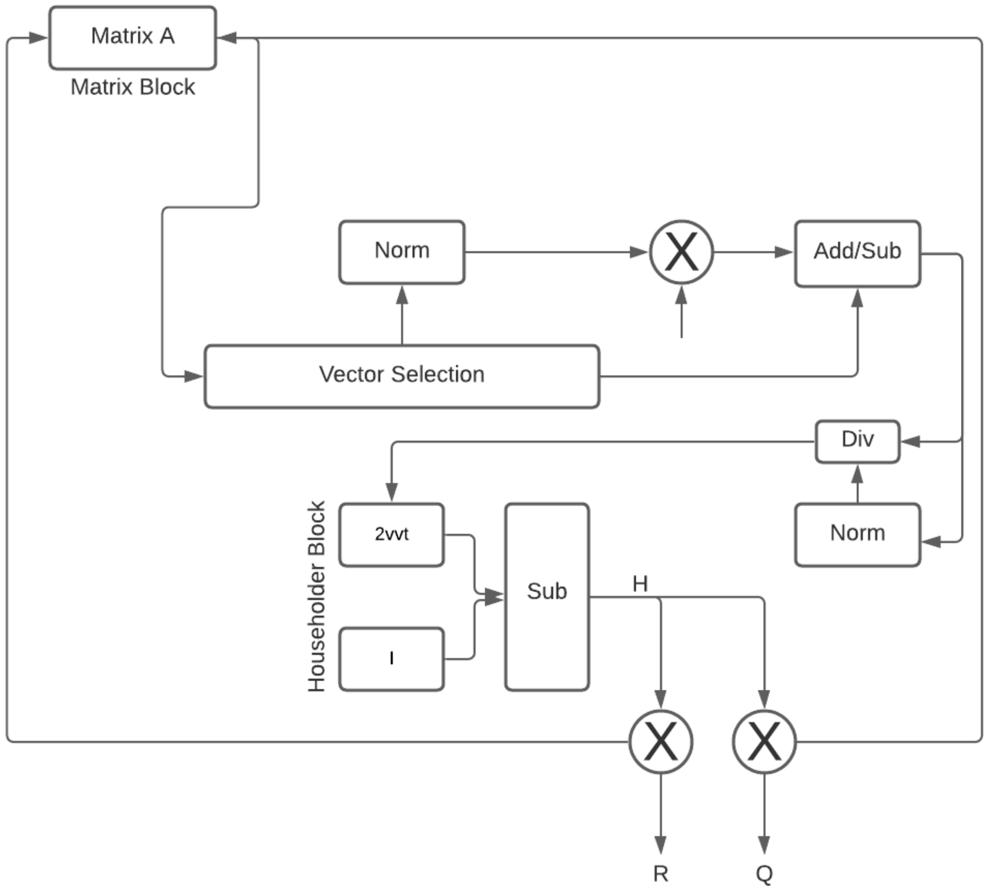


Figure 47. Block Diagram for QR decomposition using Householder Transformation

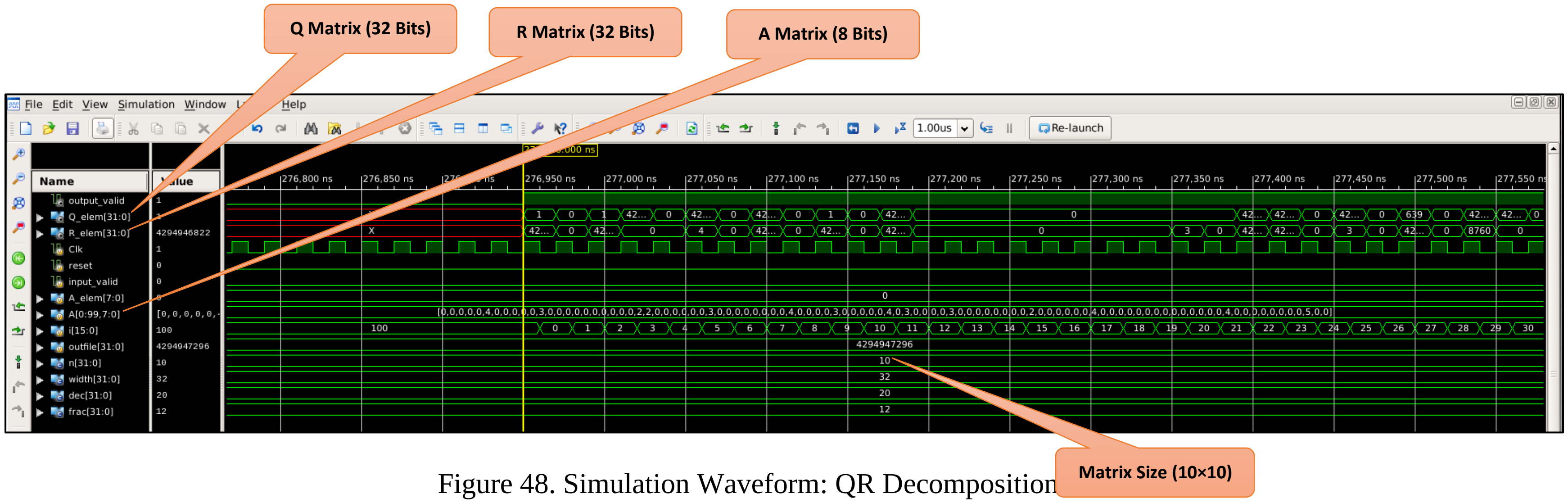


Figure 48. Simulation Waveform: QR Decomposition

As different decomposition technique leading to different solutions, the right choice decomposition technique depends on the problem and the matrix to be decomposed. This approach is mainly focused on the performance improvement of QR decomposition using Householder transformation by implementing on FPGA. This decomposition technique is mostly used to solve linear square problems, OFDM-MIMO, adaptive beamforming.

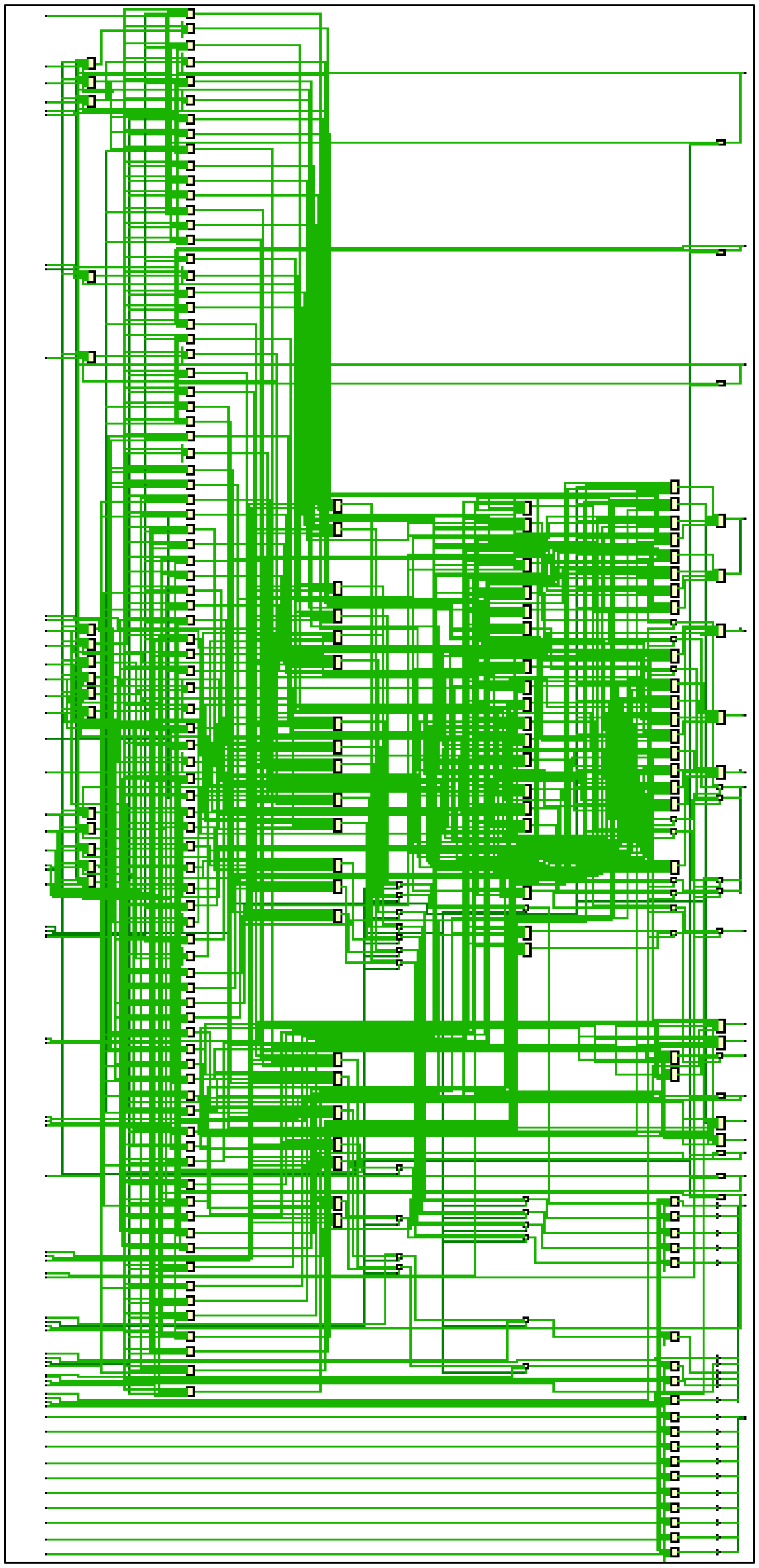


Figure 49. Implemented Design: QR Decomposition

Figure 49 shows the schematic diagram of the implement QR decomposition from Vivado Design Suite. The schematic is analyzed to find if any logic or nets are not interconnected. The overall engine is implemented using the finite state control machine which consists of the memory unit, and the necessary operators.

Figure 50 captures the interconnect schematic for the Output engine. Once all the normalization is performed the finite state machine transfers the output data when the ***output_valid*** is available. The registers work is parallel to keep the temporary factorization values, which is being used by each row and column shifting with the square root module. This eliminates the loss of data, thereby keeping the precision loss to a minimum.

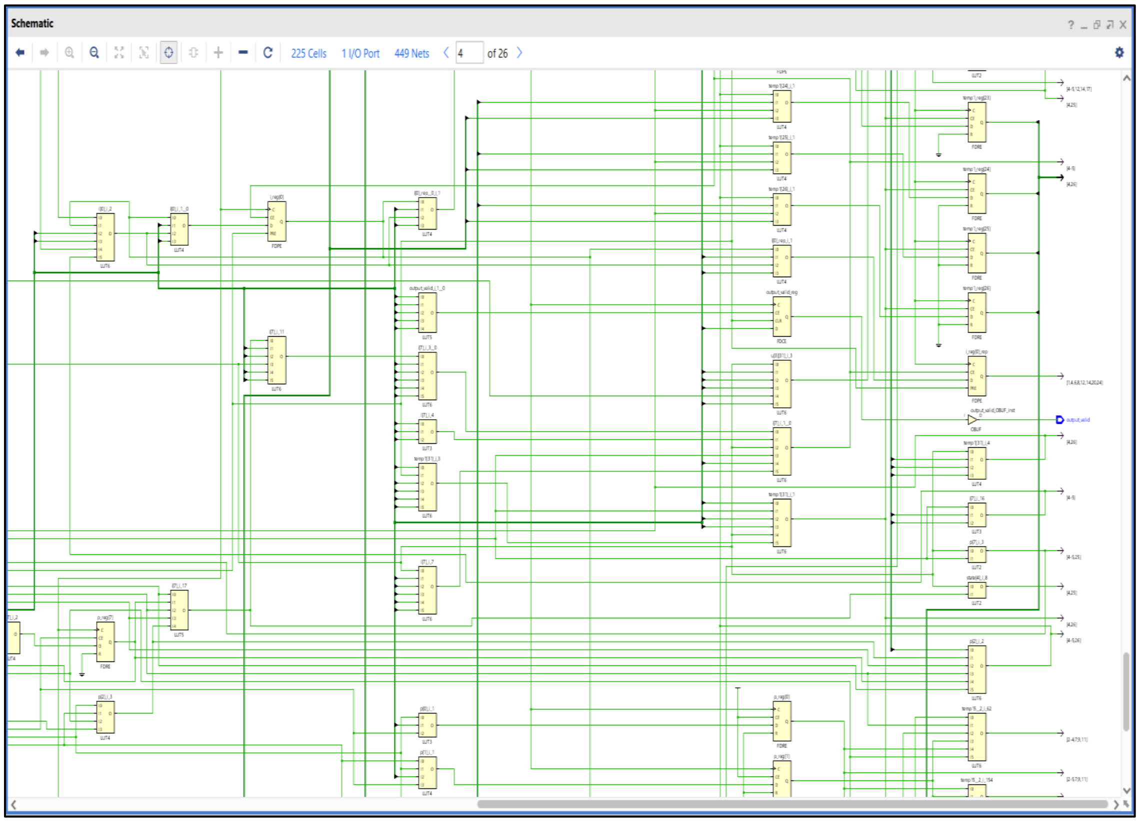


Figure 50. Interconnect Schematic: Output FSM

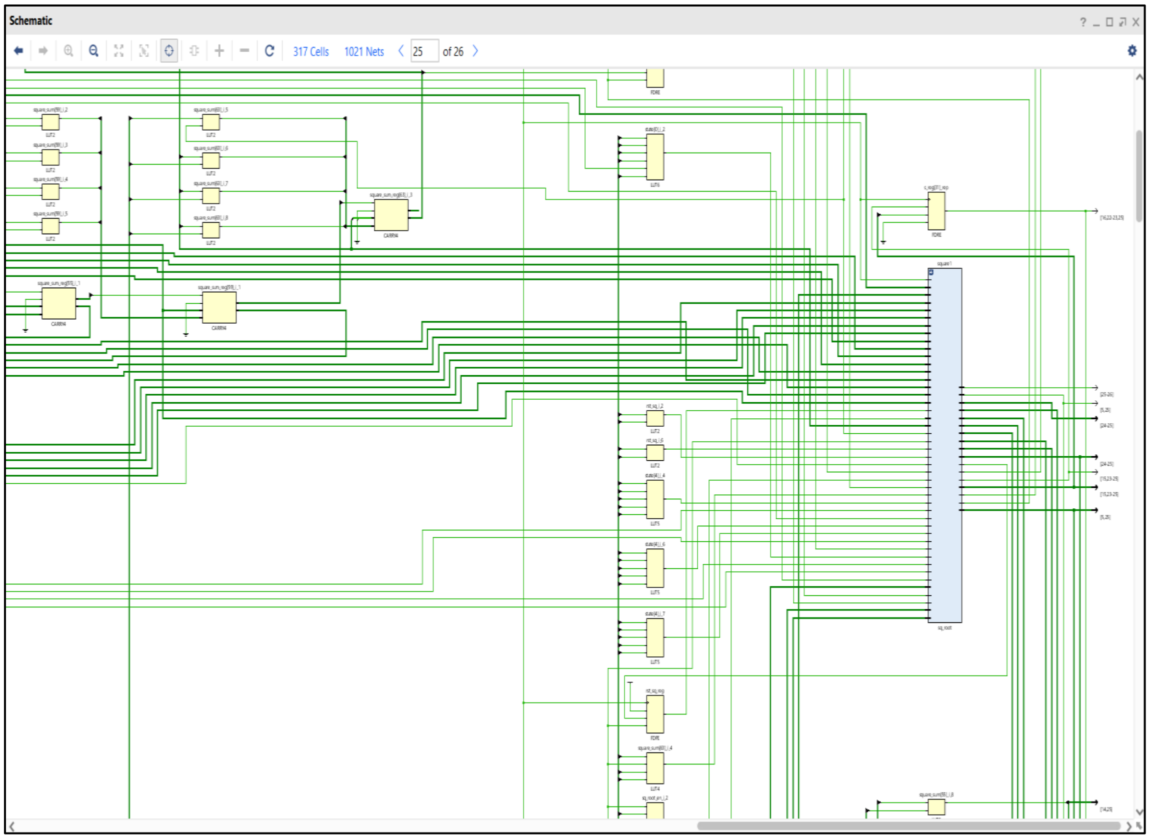


Figure 51. Interconnect Schematic: Square Root Module

Figure 51 shows the interconnect schematic of the square root module, which is designed as a sperate module, but works in parallel along with the factorization. This engine is designed with finite state machine and as a part of the module, shifting with **CORDIC** algorithm is implemented to reduce the multipliers used. This operation is performed when the clock high is reset, and an extra parameter is employed to take care of the fractional part of the bit.

The error analysis is carried out by comparing the software results from MATLAB and the hardware results from Vivado Design Suite. This is the precision of error for decomposing an input matrix **A** into resultant **Q** and **R** matrices, respectively. The Mean Error(**ME**) is the average of all errors. The formula for calculating the Mean Error is:

$$\boldsymbol{Mean\ Error = \frac{1}{n}\sum_{i=1}^{n}|x_i - x|}$$

Where $\boldsymbol{n}$ – the number of errors, $|\boldsymbol{x_i - x}|$ – the absolute errors.

Table 5. Error Analysis for QR Decomposition Operation

10 x 10 Matrix			
Sparsity	Matrix	Min	Max
10%	Q	-2	0.0015
	R	-1	0.0035
20%	Q	-0.002	0.0021
	R	-0.0037	0.0051
30%	Q	-0.0037	0.002
	R	-0.005	0.0106
40%	Q	-0.0021	0.0028
	R	-0.0085	0.0121
50%	Q	-0.0033	0.0023
	R	-0.0108	0.0121
Mean Error	**Q**	**-0.40222**	**0.00214**
	R	**-0.2056**	**0.00868**

Table 5 represents the data for while decomposing a matrix of size 10×10 with varying the sparsity of the matrices from 10% to 50%. The Min and Max value of errors are calculated which is the difference between the MATLAB software results and the Vivado Hardware results. Once values these are tabulated, the Mean Error is calculated from the above-mentioned formula to investigate the minimum and maximum precision loss.

5. ALGORITHM PERFORMANCE RESULTS

Once the optimized sparse matrix operation algorithms had been developed using Verilog code, we can compare its efficiency with the conventional matrix operation algorithms to confirm our performance estimates and do further optimization. The codes are tested for simulation and synthesize using Vivado Design Suite on 64-bit Intel® Core™ i-3-5005U CPU processor operating at 2.00GHz. Further the results are checked with MATLAB code to ensure we are getting the same output.

Once the FPGA hardware design is ready to be tested, test runs with various matrices with different sparsity structure, and density is used to evaluate the performance. These results are plotted in forms of graphs and tables for detailed analysis. The performance of sparse matrix operations was determined by the number of the nonzero present in the matrices. Each of these test matrices sizes were 10×10, 20×20, 30×30, 40×40, 50×50, 60×60 and 100×100 with different density ranging between 1% and 10%.

5.1 Sparse Matrix Addition vs. Regular Matrix Addition

The designed sparse matrix addition algorithm is simulated and the necessary measurements at what time the input matrix is taken (**t1**) and what time the output matrix is produced (**t2**) are calculated for latency and throughput calculations. These metrics calculated for the proposed algorithm are compared with the regular algorithm and the comparison is plotted in graphs. It is evident from Figure 52 for matrix dimension of 10×10 the latency of sparse matrix addition with different sparsity percentage of the matrix ranging from 1% to 10% is significantly reduced. As the matrix size keeps increasing the latency increases but beyond the regular algorithm.

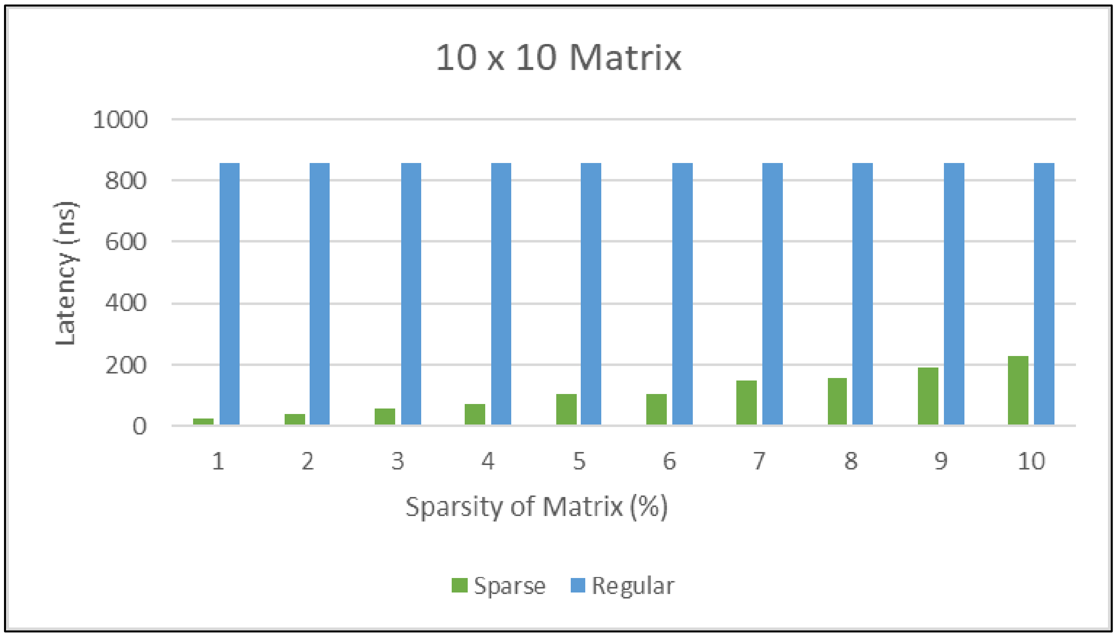


Figure 52. Latency Comparison: Proposed vs Regular for 10×10 Matrix

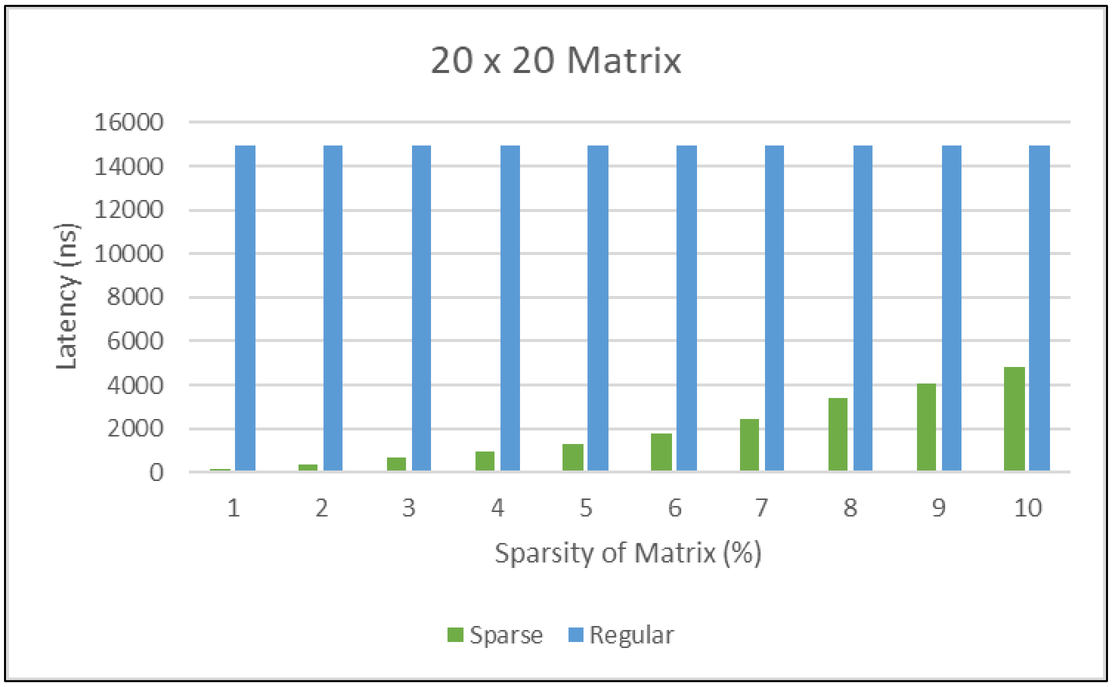


Figure 53. Latency Comparison: Proposed vs Regular for 20×20 Matrix

Figure 53 represents the latency comparison of proposed algorithm with the regular algorithm for 20×20 matrix with sparsity range varying from 1% to 10%. When we compare the latency performance of different sizes of the matrix with different sparsity, we were able to produce improvements in latency for sparse matrix algorithm.

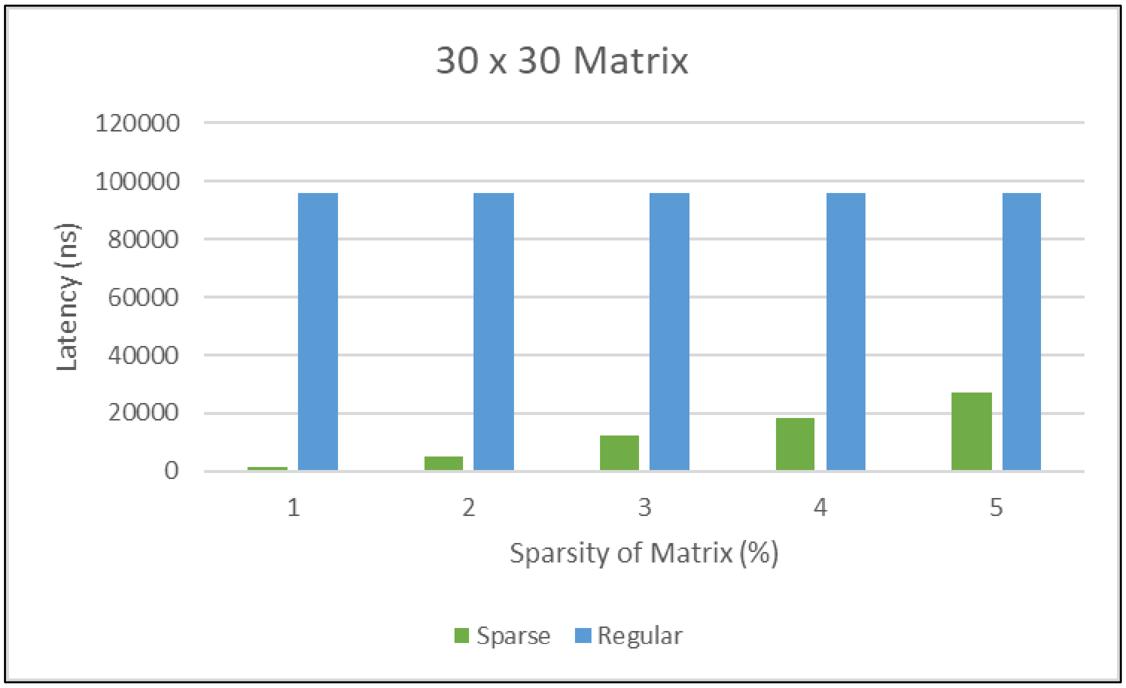


Figure 54. Latency Comparison: Proposed vs Regular for 30×30 Matrix

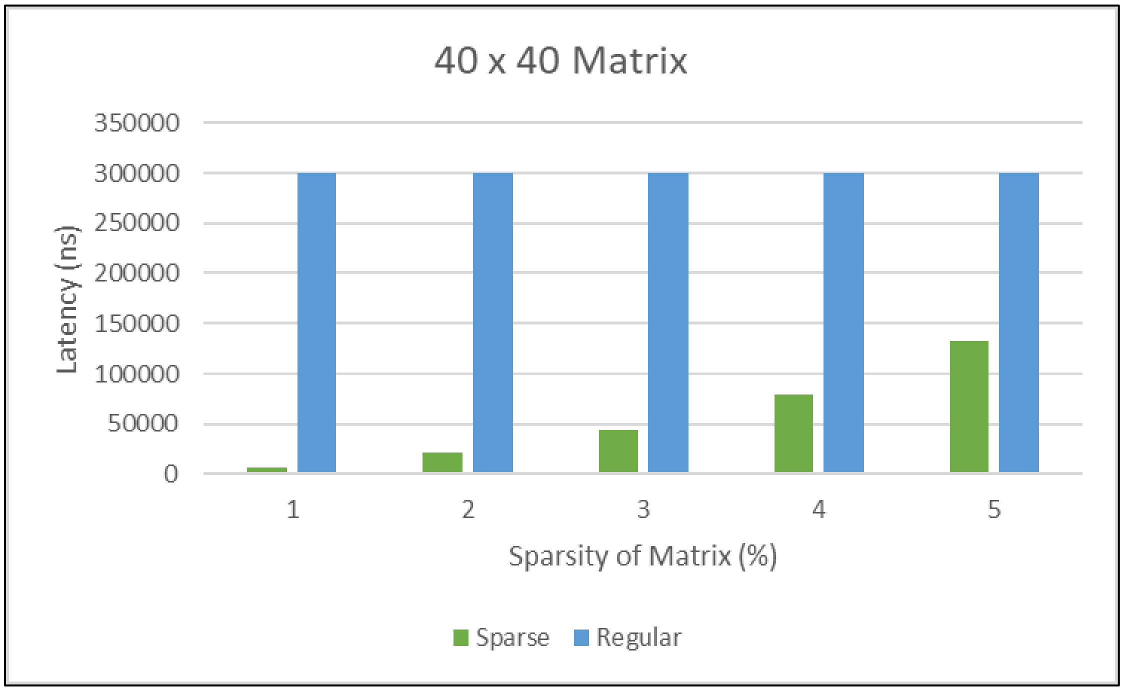


Figure 55. Latency Comparison: Proposed vs Regular for 40×40 Matrix

Figure 54 and 55 represents the latency comparison for 30×30 and 40×40 matrix with sparsity range varying from 1% to 5%.

The data plot shows latency is directly proportional to the size of the matrix. The lower the size of the matrix, consumes a smaller number of clock cycles as compared with the higher the size of the matrix.

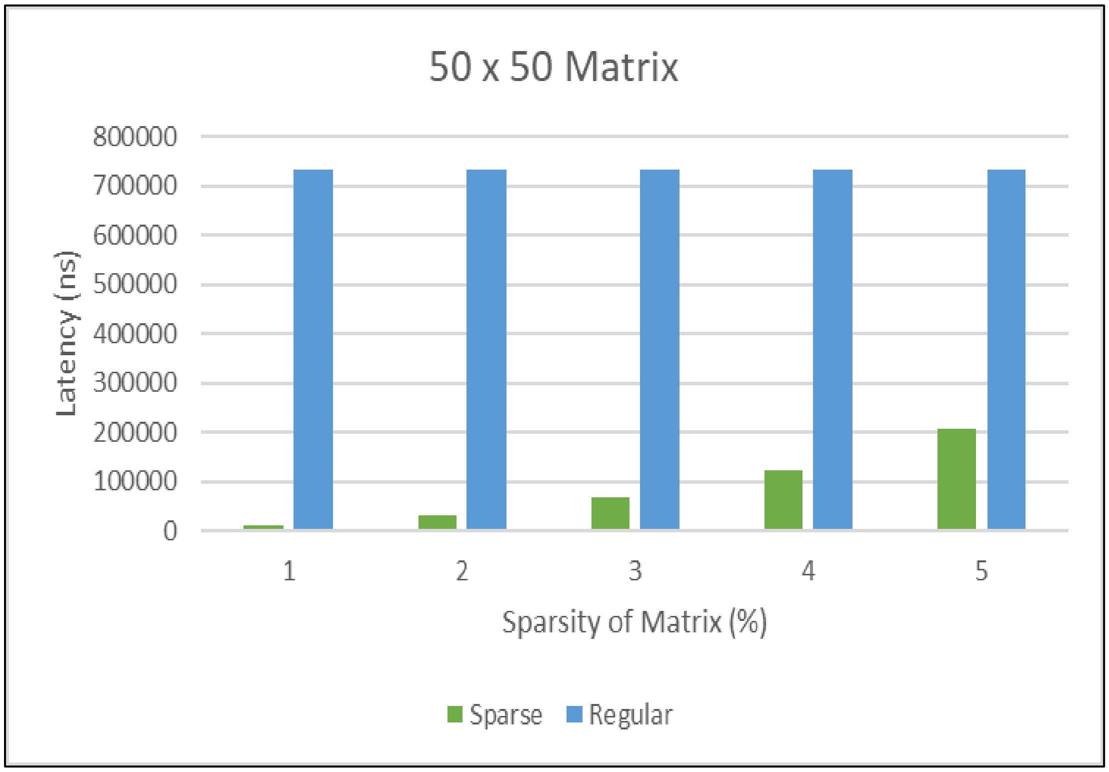


Figure 56. Latency Comparison: Proposed vs Regular for 50×50 Matrix

Figure 56 shows the latency comparison of a 50×50 matrix size, with different sparsity ranging between 1 to 5% between the regular operation and proposed operation. From the graph, the decrease in latency for each operation by the proposed method is evident and was able to achieve comparatively lower latency leading to improved throughput. Each test matrices were imported from the Suite Sparse Matrix Collection from Texas A&M University dataset with different varying properties and patters which are laten detailed towards the end of the thesis discussion.

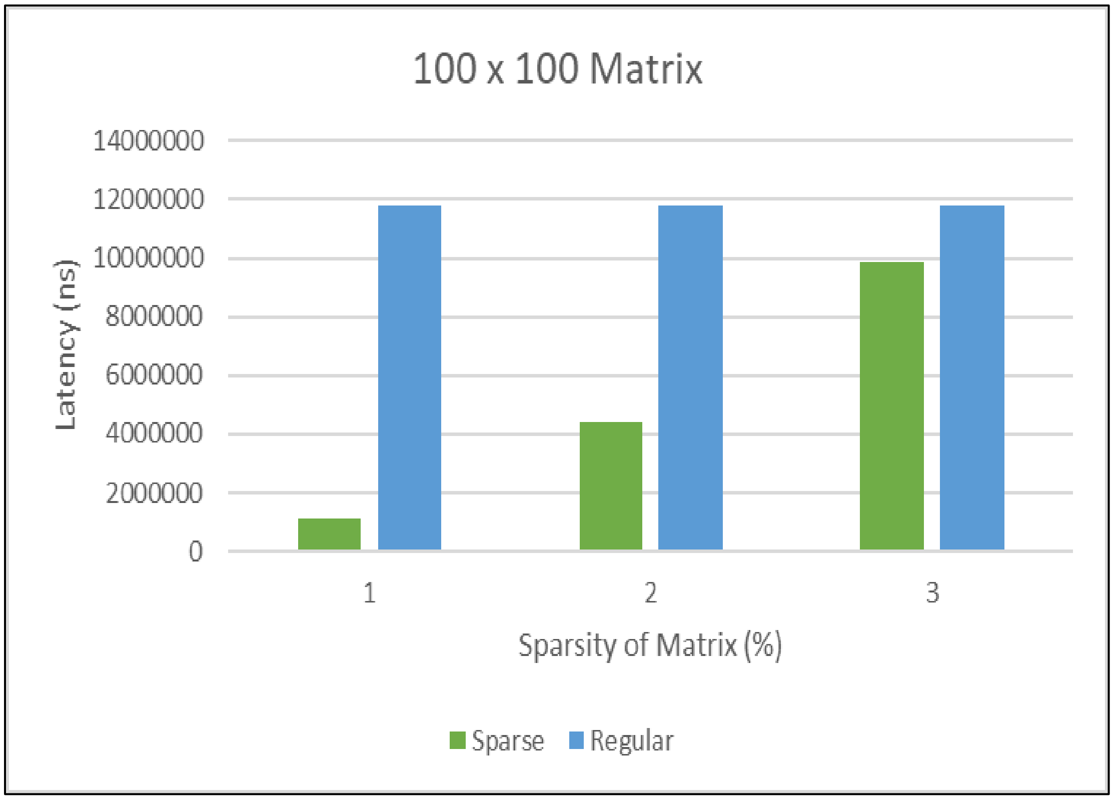


Figure 57. Latency Comparison: Proposed vs Regular for 100×100 Matrix

The overall arithmetic operation throughput is calculated from the clock cycle latency and the frequency achieved by the proposed design. Numerous varying sizes of matrix valves are involved, with differing sparsity percentage and the results are plotted in graphs for comparison.

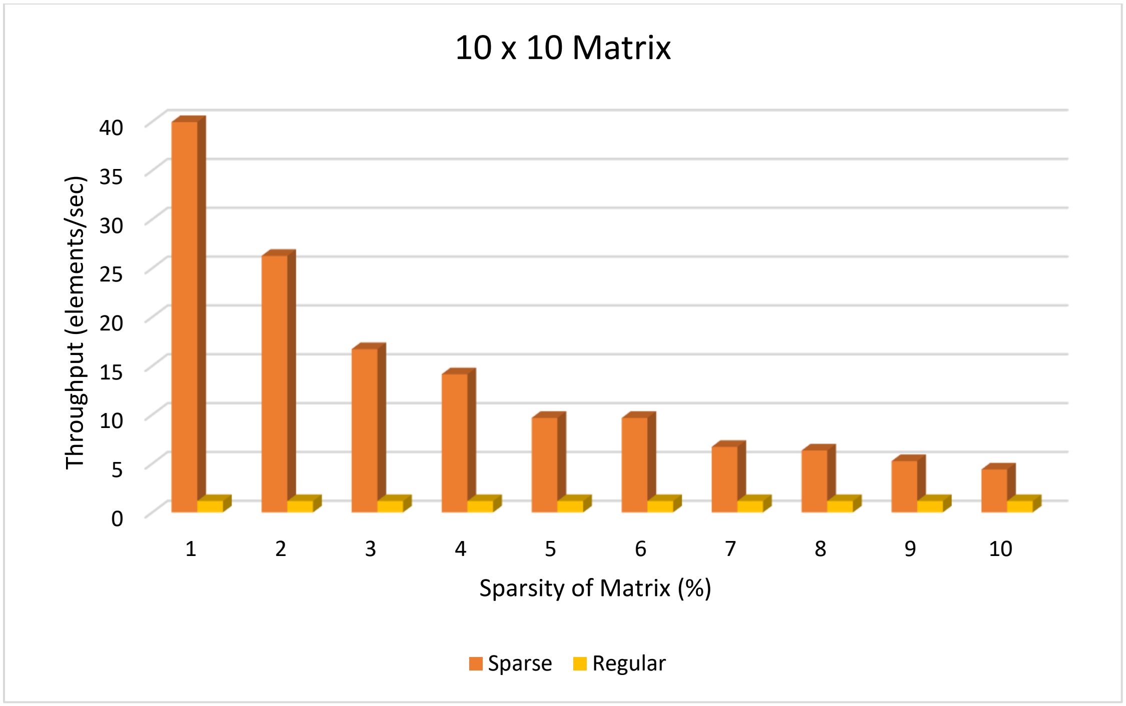


Figure 58. Throughput Comparison: Proposed vs Regular for 10×10 Matrix

Figure 58 shows the increase in throughput for the proposed method, in comparison with the regular method in terms of bits/sec. For a 10×10 matrix, the design was able to achieve a maximum of 4.2% increase with 10% of the elements being non zeros. Figure 59 shows the comparison of throughput calculated for 20×20 matrix size, and Figure 60 shows the comparison of throughput calculated for 30×30 matrix size where the proposed method can get a maximum of 2.3% increase in throughput. From the data analysis, throughput is directly proportional to the size of the matrix.

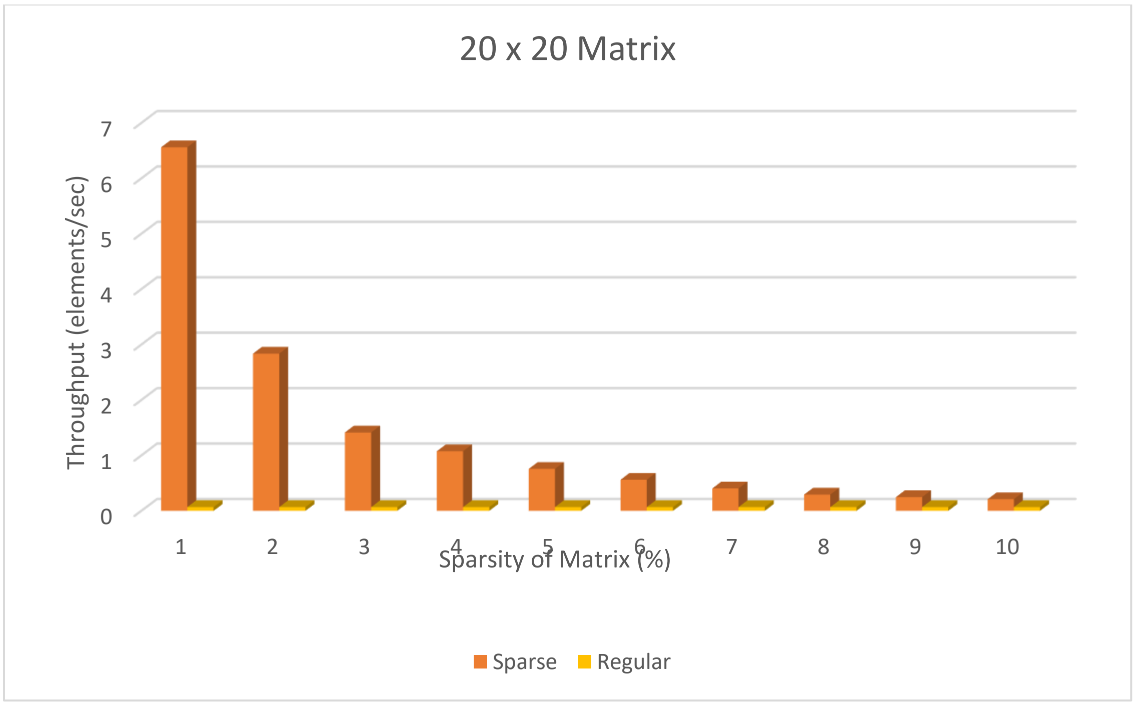


Figure 59. Throughput Comparison: Proposed vs Regular for 20×20 Matrix

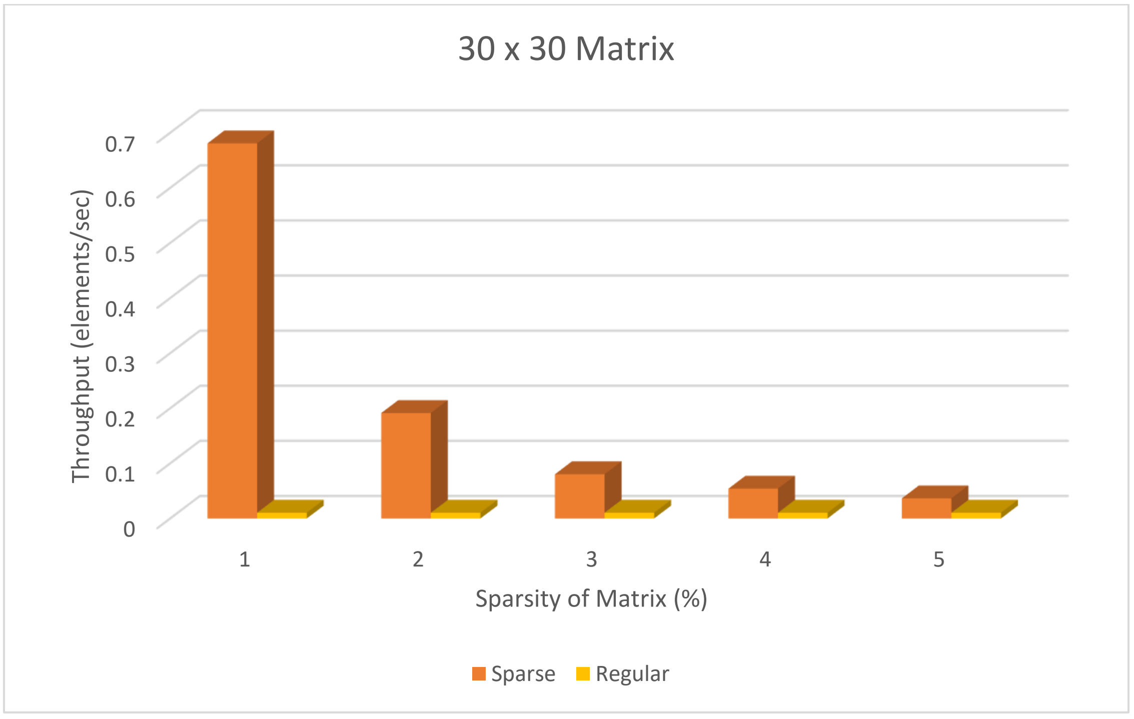


Figure 60. Throughput Comparison: Proposed vs Regular for 30×30 Matrix

Figure 61 shows the comparison of throughput between the two methods for 40×40 matrix size, containing 1% to 5% sparsity range.

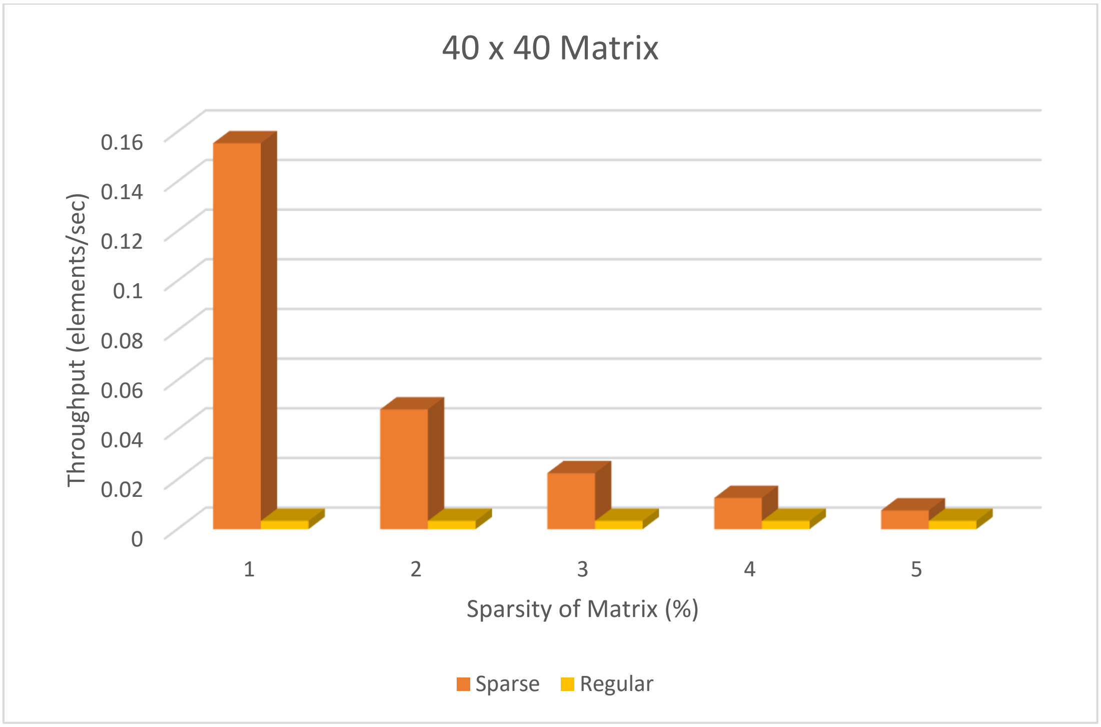


Figure 61. Throughput Comparison: Proposed vs Regular for 40×40 Matrix

Figure 62 is plotted to compare the throughput results between the proposed and the regular method, which shows the maximum increase in throughput is achieved for a matrix size of 50×50 with varying sparsity percentage from 1% to 5%. Figure 63 shows the comparison of throughput between the proposed and regular method for 100×100 matrix with sparsity range varying between 1% to 3%.

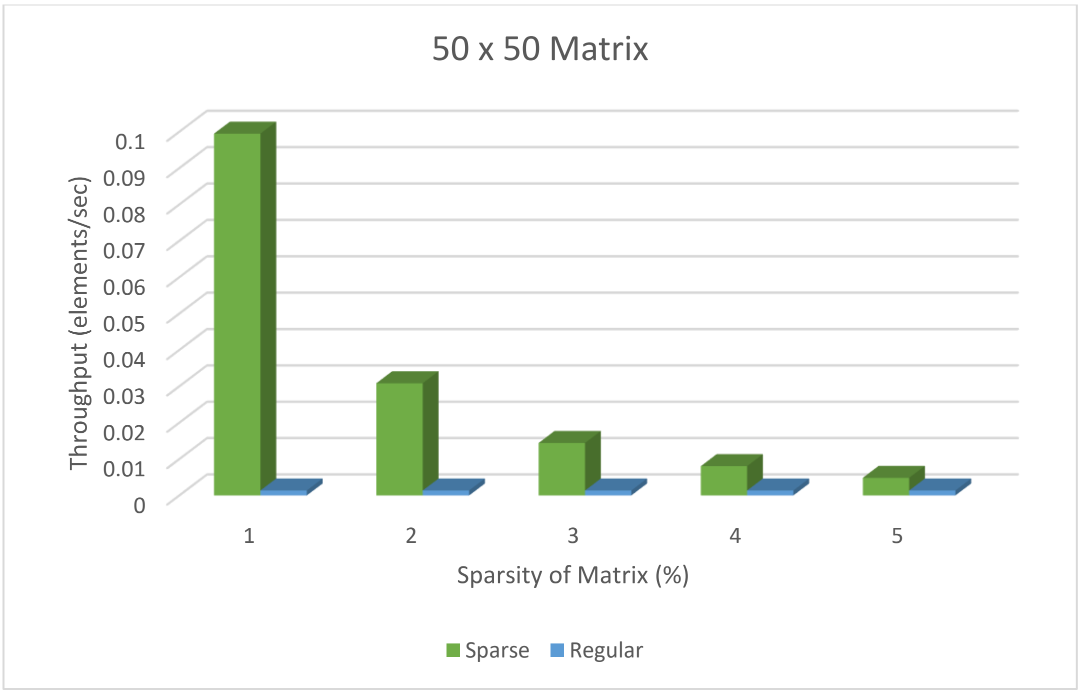


Figure 62. Throughput Comparison: Proposed vs Regular for 50×50 Matrix

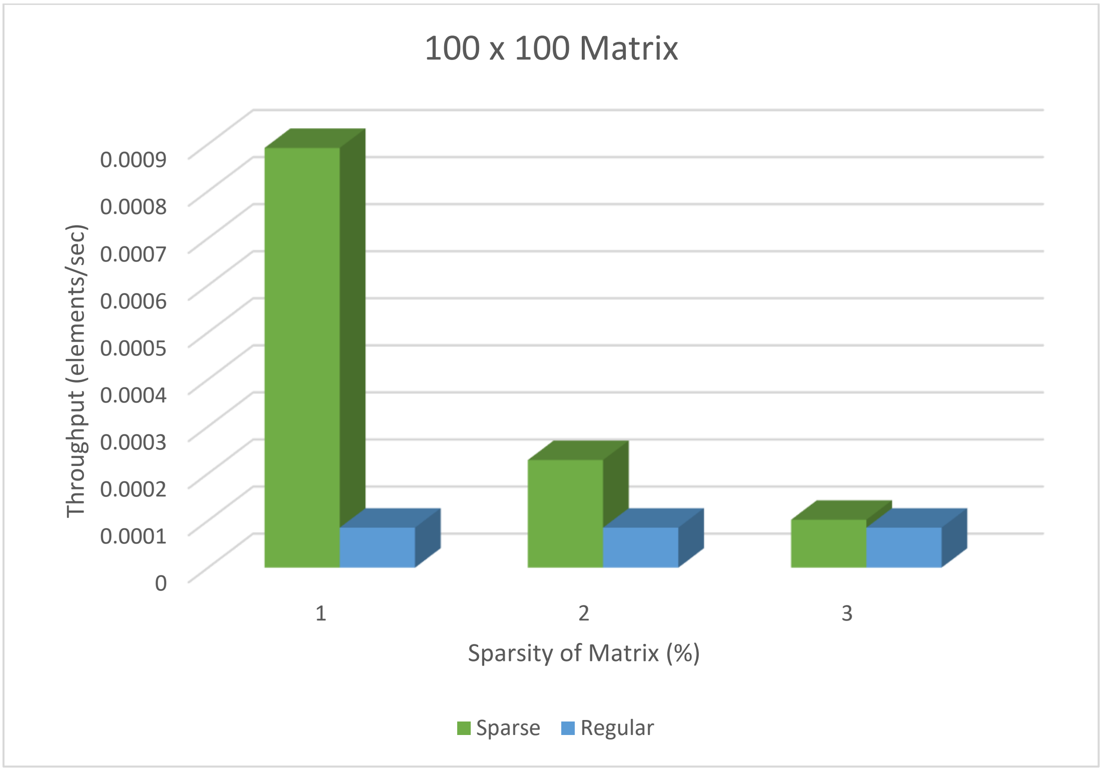


Figure 63. Throughput Comparison: Proposed vs Regular for 100×100 Matrix

From figure 63, the throughput obtained from sparse matrix algorithm and regular matrix algorithm are plotted as a graph to show the improvements in performance. Figure 63 illustrates a large increase in throughput from small size matrix with less sparsity while it gradually decreases as the size of the matrix dimension increases. This is due to the increase in number of operations to be performed when the nonzero value increases with the size of the matrix. But however, even for a 100×100 matrix with 10% sparsity distribution, a significant increase in throughput is proved from the comparison.

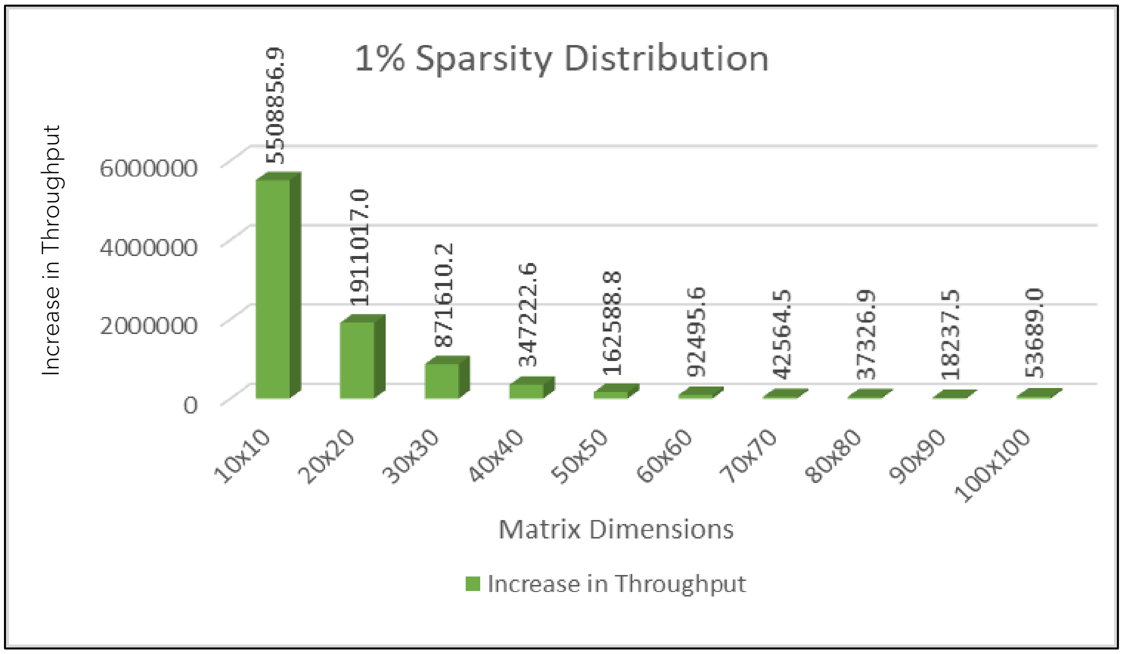


Figure 64. Increase in Throughput for Different Matrix Dimensions with 1% Sparsity

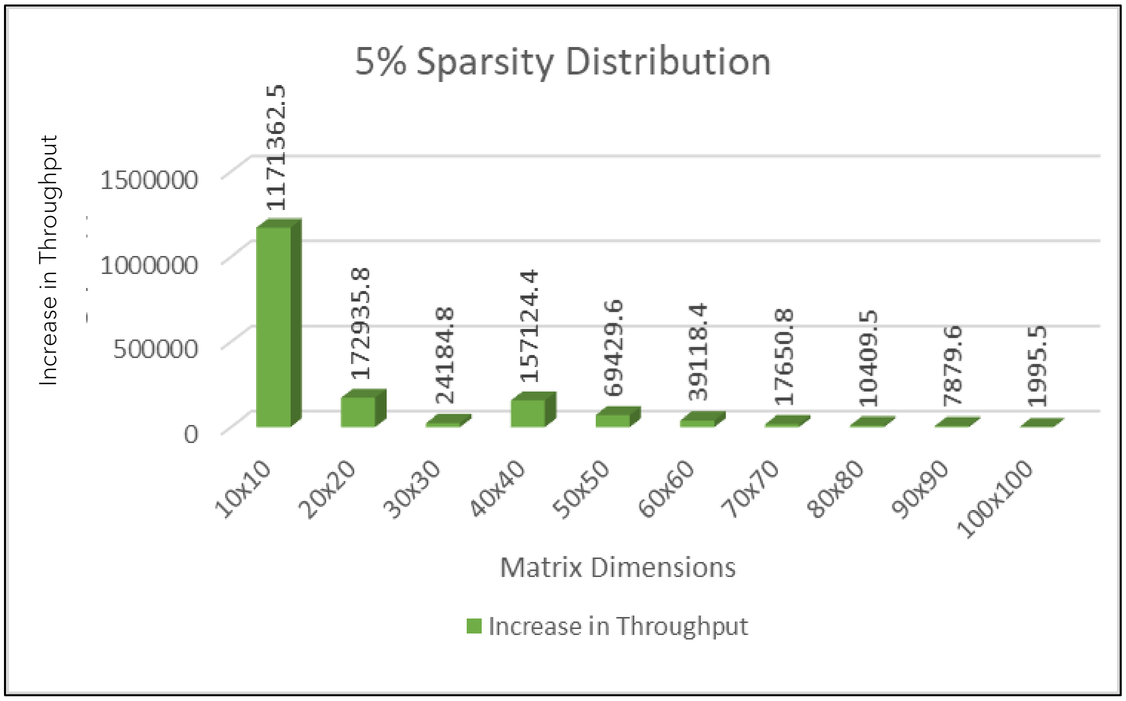


Figure 65. Increase in Throughput for Different Matrix Dimensions with 5% Sparsity

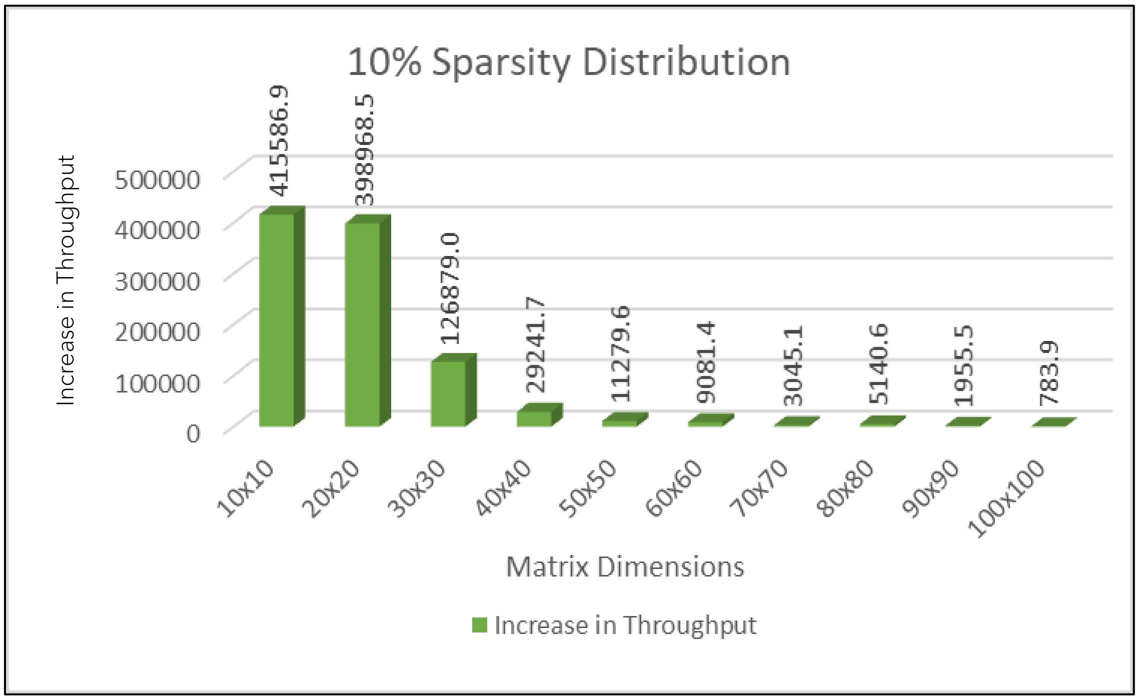


Figure 66. Increase in Throughput for Different Matrix Dimensions with 10% Sparsity

With the use of Vivado Design Suite, the hardware resources utilized for both sparse algorithm and regular algorithm are compared in Figure 67.

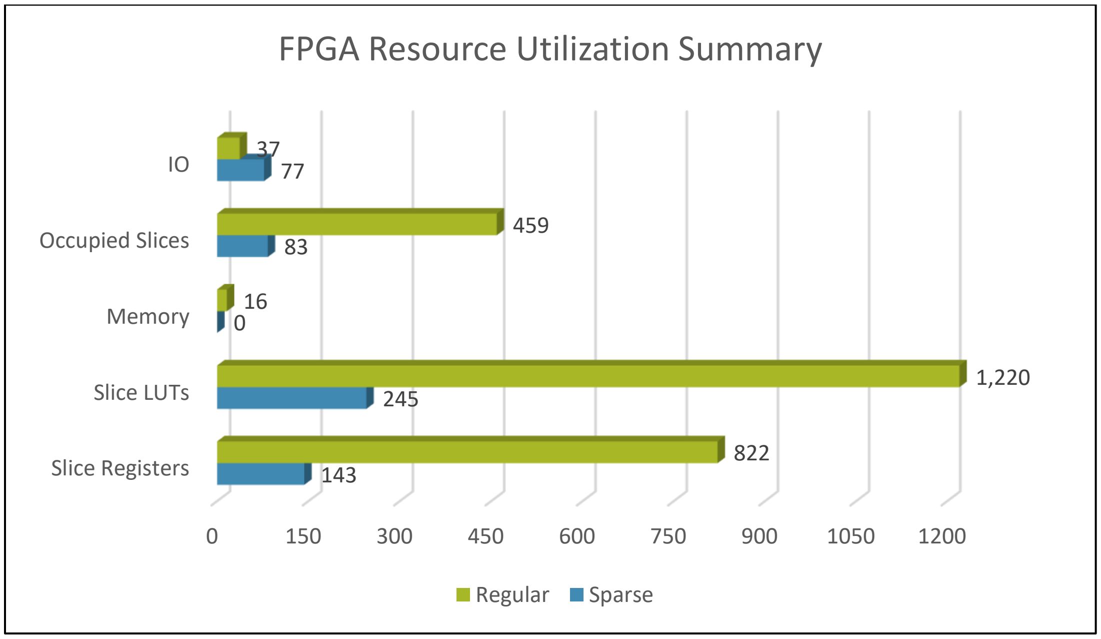


Figure 67. FPGA Resource Utilization: Regular vs Sparse-Sparse Matrix Addition Operation

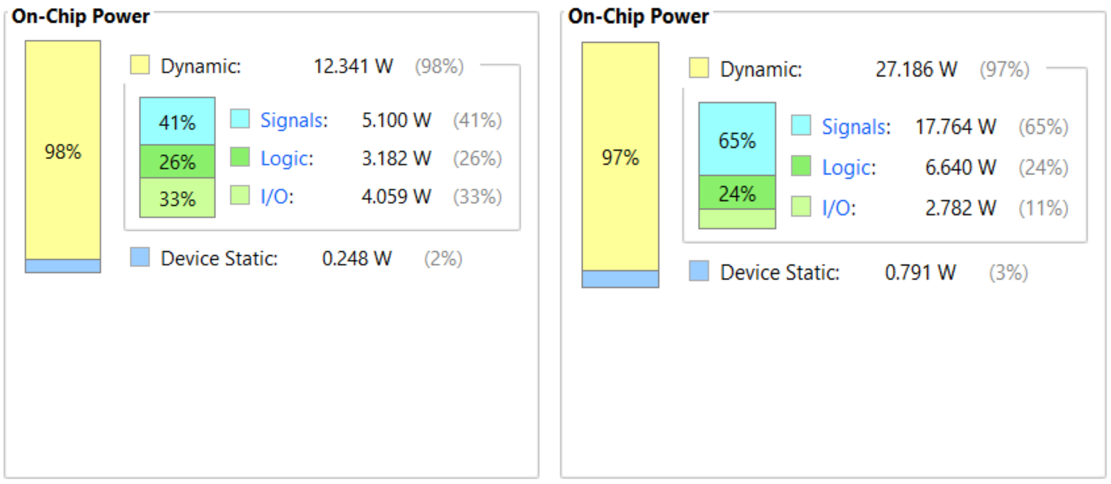
Figure 67 provides the hardware utilization for a 10×10 matrix dimension with 10% sparsity. We can see the amount of the hardware utilized is significantly reduced in the proposed sparse algorithm as the addition operation is performed only for the nonzero values saving more hardware resources.

Table 6 summarizes the best and worst-case operational delays. These values are used to compare with software implementation. It shows the best- and worst-case delays achievable for fetching the input and calculating the output. The design was able to achieve a min of 3.76ns to process input data, and max of 4.34ns for the same. Once the necessary operation is performed, the design restricts to a min of 4.82ns for the output to be available and a max of 5.96ns.

Table 6. Hardware Implementation: Sparse Matrix Arithmetic Operation

	Input (ns)	Output (ns)
Best Case Delay (min)	3.76	4.82
Worst Case Delay (max)	4.34	5.96

The power analysis tool articulates the power consumed by the design. Figure 68(a) shows the estimated static and dynamic power consumed by the proposed design, and Figure 68(b) shows the estimated power consumed by the regular design. This shows a comparison of the power, and clearly the proposed design is consuming less power.



On-Chip Power

🟨 Dynamic:	12.341 W	(98%)
🟦 Signals:	5.100 W	(41%)
🟩 Logic:	3.182 W	(26%)
🟩 I/O:	4.059 W	(33%)
🟦 Device Static:	0.248 W	(2%)

On-Chip Power

🟨 Dynamic:	27.186 W	(97%)
🟦 Signals:	17.764 W	(65%)
🟩 Logic:	6.640 W	(24%)
🟩 I/O:	2.782 W	(11%)
🟦 Device Static:	0.791 W	(3%)

(a) Proposed Method (b) Regular Method

Figure 68. Power Analysis: Sparse Matrix Arithmetic Operation

5.2 Sparse Matrix Subtraction vs. Regular Matrix Subtraction

Since subtraction is equivalent to the addition of matrix with signs of its nonzero reversed, the same algorithm is used for analyzing the performance of subtraction operation. Table 5 presents us the Latency and Throughput calculations of sparse matrices of different sizes ranging between 10×10 to 100×100, with different sparsity pattern and sparsity percentage ranging from 1% to 10%.

Table 7 also provides the number of nonzero elements present in each matrix. We were able to achieve low latency and high throughput for the proposed sparse algorithm when compared with the regular algorithm for subtraction of two matrices. The number of nonzero is the determining factor for the number of operations to be performed by the algorithm.

Table 7. Latency and Throughput Comparison from Implemented Design

Matrix Size (n*n)	**Number of nonzero (nnz)**	**Sparsity (%)**	**Sparse Algorithm**		**Regular Algorithm**	
			Latency (ns)	**Throughput (s)**	**Latency (ns)**	**Throughput (s)**
10×10	1	1%	119.7495	8350765.56	3205.434	311970.284
	2	2%	182.2275	5487645.94		
	3	3%	286.3575	3492138.32		
	4	4%	338.4225	2954886.27		
	5	5%	494.6175	2021764.29		
	6	6%	494.6175	2021764.29		
	10	10%	1088.1585	918983.769		
20×20	4	1%	378.7845	2640023.55	19199.6	52084.4279
	8	2%	876.2925	1141171.47		
	12	3%	1758.2385	568751.054		
	16	4%	2312.2815	432473.295		
	20	5%	3284.6835	304443.335		
	24	6%	4437.9975	225326.851		
	40	10%	11889.311	84109.1668		

	9	1%	832.902	1200621.44		
	18	2%	2963.318	337459.564		
	27	3%	7065.502	141532.76		
30×30	36	4%	10487.766	95349.1907	53294.37	18763.7089
	45	5%	15530.506	64389.4024		
	54	6%	19598.694	51023.808		
	90	10%	47452.75	21073.5943		
	16	1%	2121.745	471310.172		
	32	2%	6841.901	146158.21		
	48	3%	14619.695	68400.8798		
40×40	64	4%	26222.443	38135.2721	38884.05	25717.4857
	80	5%	43638.191	22915.707		
	96	6%	60647.029	16488.8539		
	160	10%	164490.46	6079.37989		
	25	1%	5007.3435	199706.691		
	50	2%	17691.746	56523.5352		
	75	3%	39908.984	25057.015		
50×50	100	4%	52020.374	19223.2376	148899.9	6715.92006
	125	5%	109712.26	9114.75133		
	150	6%	162650.76	6148.14225		
	250	10%	436771.89	2289.52465		
	36	1%	9174.7035	108995.348		
	72	2%	32160.299	31094.2388		
	108	3%	84721.127	11803.4313		
60×60	144	4%	126004.04	7936.25369	214411.5	4663.92837
	180	5%	211929.79	4718.5438		
	216	6%	287007.39	3484.2309		
	360	10%	850915.66	1175.20461		
	100	1%	70001.865	14285.3337		
	200	2%	273098.98	3661.67608		
	300	3%	613428.28	1630.18243		
100×100	400	4%	1089617.6	917.753182	589749.8	1695.63423
	500	5%	1701666.9	587.659084		
	600	6%	2449576.2	408.233885		
	1000	10%	6786227.4	147.357279		

We were able to reduce the number of resources utilized from the proposed sparse algorithm than the regular algorithm for sparse matrix subtraction operation. This is represented in Figure 69, which was calculated for a 10×10 matrix size implemented from the design proposed. Further optimizations were performed on the design of hardware to reduce the Look Up Tables utilized for the operation, and we were able to achieve the performance expected. From the implementation report, the amount of slice registers used for the proposed algorithm is greatly reduced by 8.19% from the conventional algorithm. Similarly, the Look Up Tables utilized were also greatly reduced in number from the implemented design. But the number of I/O used for implementation is slightly high as parallel computations are running for efficient results.

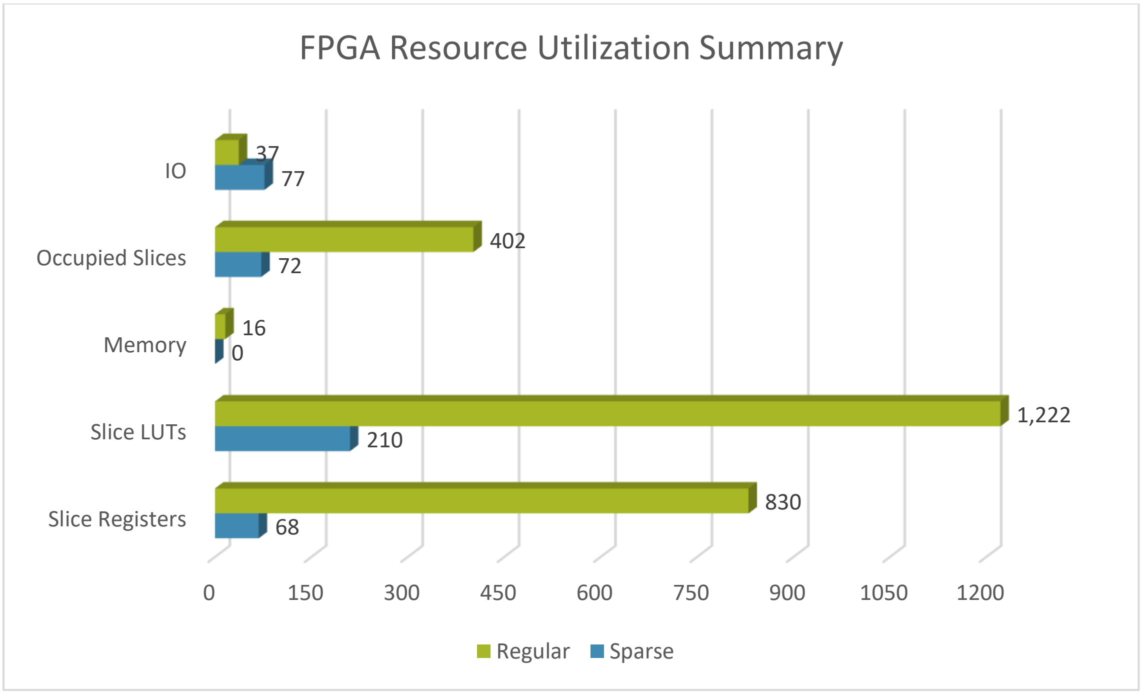


Figure 69. FPGA Resource Utilization: Regular vs Sparse-Sparse Matrix Subtraction Operation

5.3 Element-By-Element Multiplication

The performance comparison for latency achieved between proposed and regular algorithm for element-by-element multiplication is shown in Figure 70.

The Figure 70 shows how greatly the latency of the proposed algorithm is reduced with various graphs of different matrix dimensions. The conclusion of the results is like matrix addition operation except with multiplier instead of adder with the same algorithm.

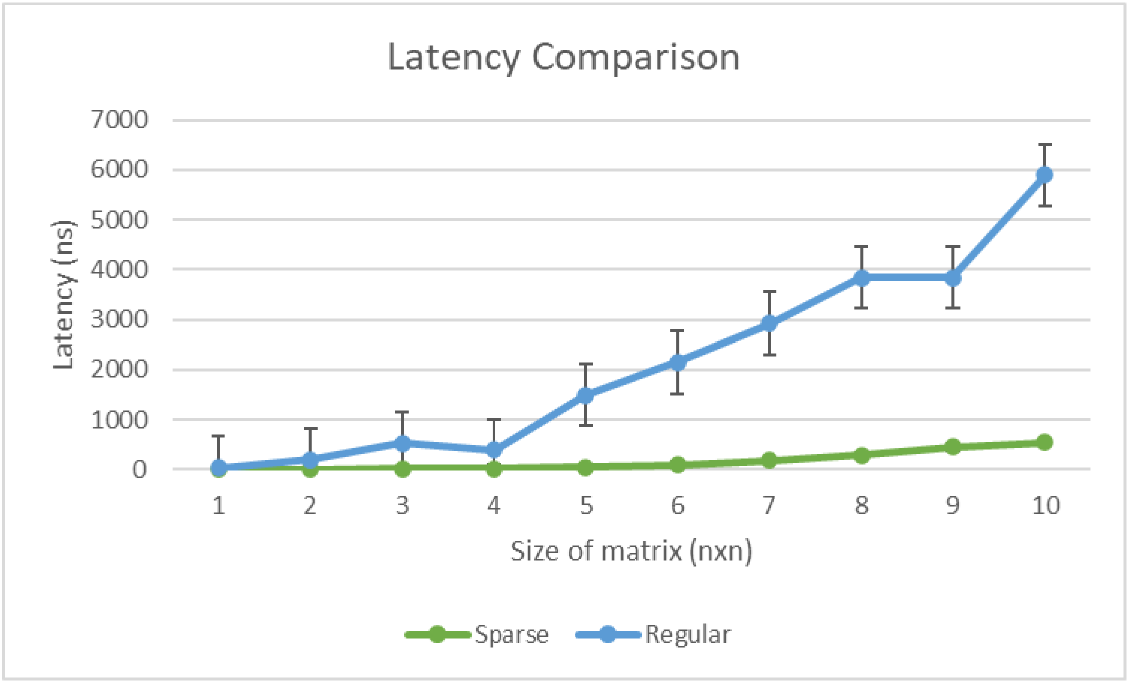


Figure 70. Latency Comparison: Proposed vs Regular for Sparse Matrix Element-by-Element Multiplication Operation

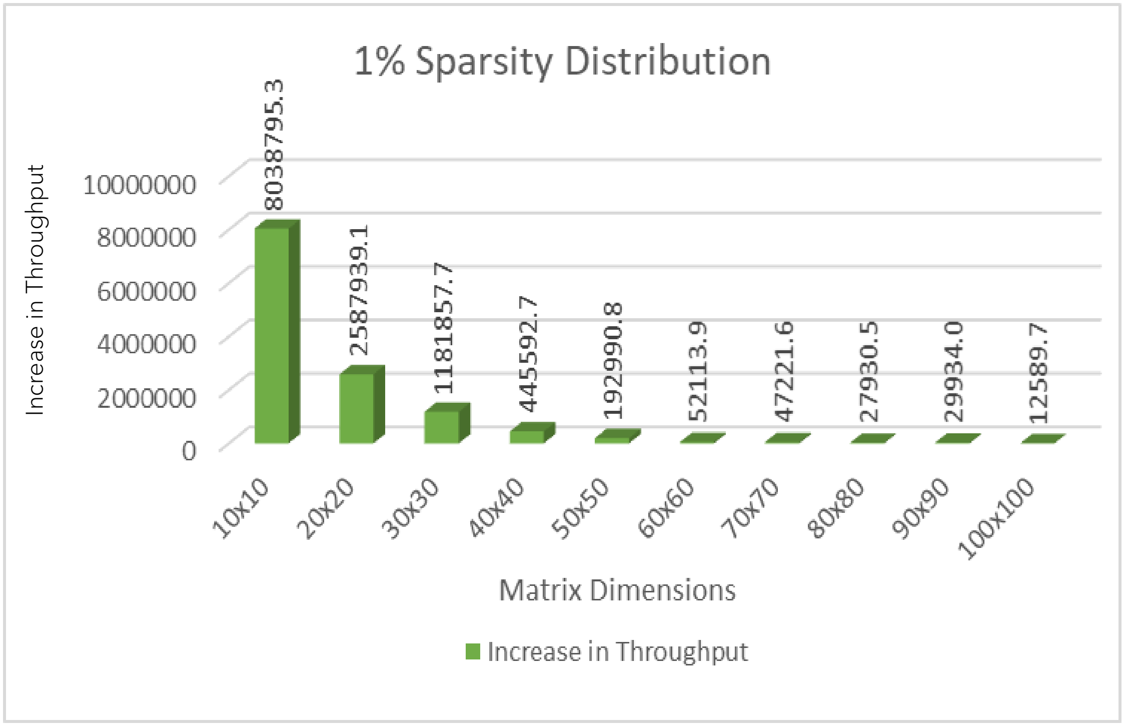


Figure 71. Increase in Throughput for Different Matrix Dimensions with 1% Sparsity

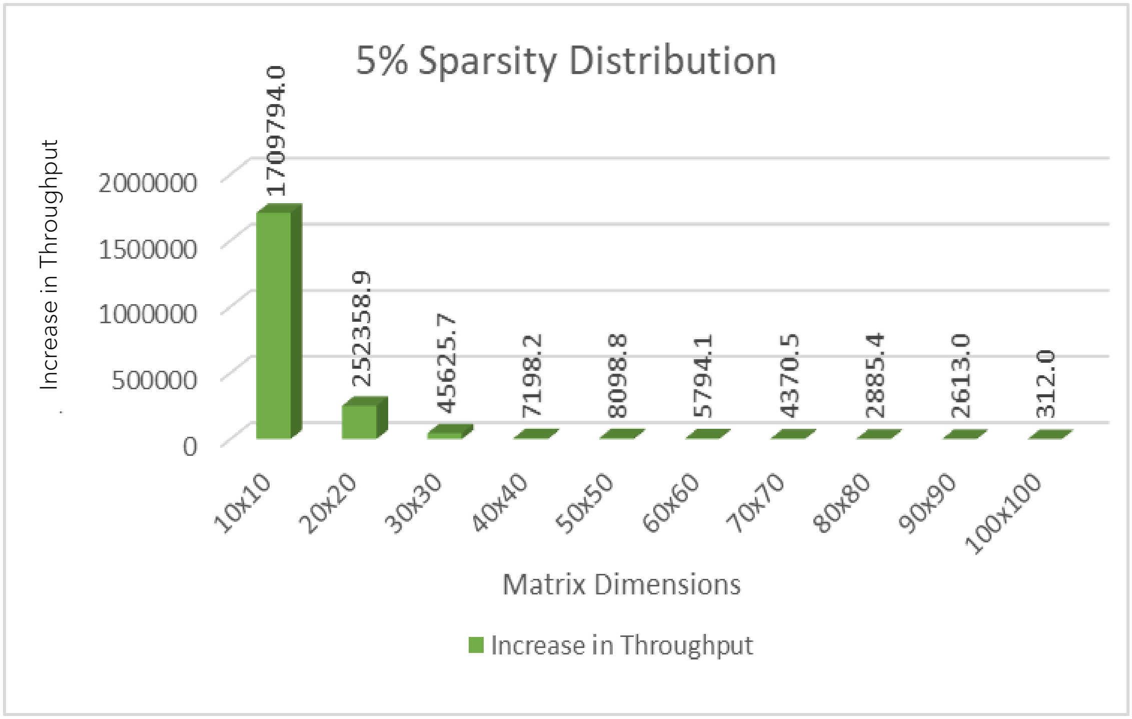


Figure 72. Increase in Throughput for Different Matrix Dimensions with 5% Sparsity

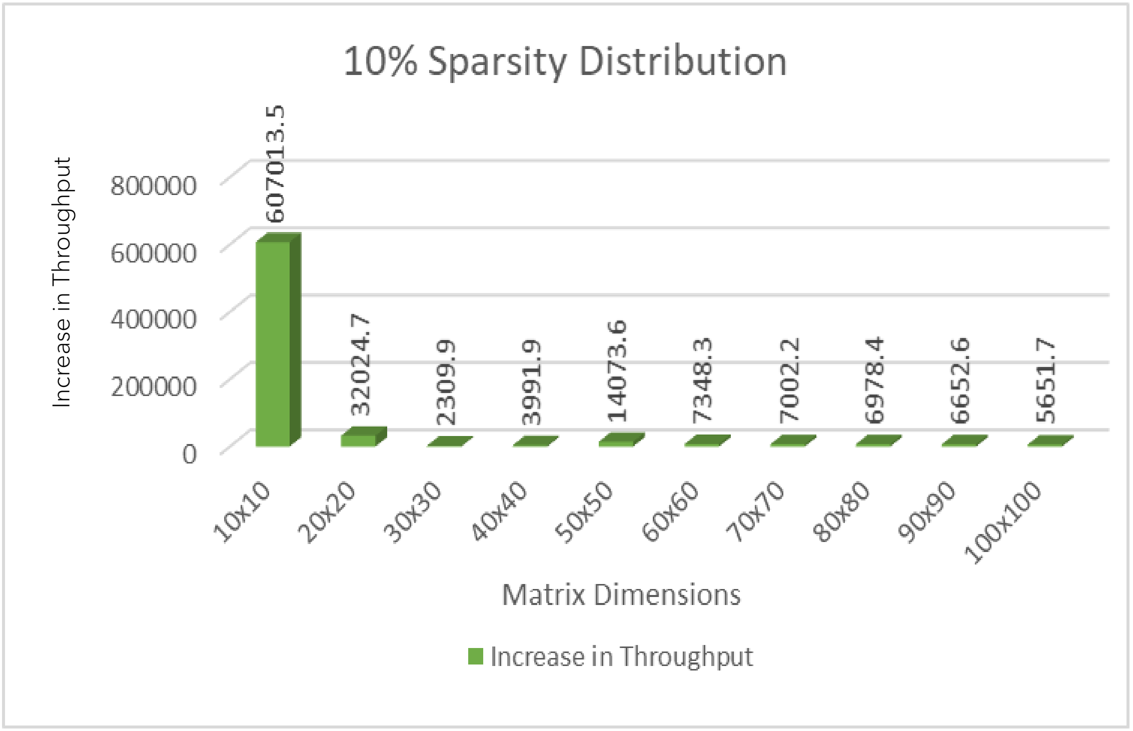


Figure 73. Increase in Throughput for Different Matrix Dimensions with 10% Sparsity

Figure 71 shows the throughput calculated for the sparse matrix subtraction operation using the sparse algorithm and regular algorithm. The values calculated are plotted as graphs to compare the throughput efficiency of the sparse based algorithm. It is also evident from the Figure 72 that we were able to achieve high throughput. Figure 73 shows high throughput is produced by the sparse algorithm for 10% sparsity. The graphs were plotted for different matrix sizes ranging from 10×10 to 100×100 with sparsity percentage ranging from 1% to 10%.

Further, the primary goal of this thesis is to reduce the storage space for matrix operations, thereby hardware utilized will be reduced. Table 8 gives a comparison of the resources used for implementing the design. The number of resources used was substantially reduced compared with the traditional algorithm for sparse matrix subtraction operation.

Table 8. Resources Utilization for Sparse Matrix Element by Element Multiplication

Device Utilization Summary						
Sparse Algorithm				**Regular Algorithm**		
Slice Logic Utilization	**Used**	**Available**	**Utilization**	**Used**	**Available**	**Utilization**
Number of Slice Registers	96	126,800	1%	825	126,800	1%
Number of Slice LUTs	246	63,400	1%	1,219	63,400	1%
Number used as Memory	0	19,000	0%	16	19,000	1%
Number of occupied Slices	81	15,850	1%	394	15,850	2%
Number of LUT Flip Flop pairs used	248			1,224		
Number of bonded IOBs	77	210	36%	37	210	17%

5.4 Sparse Matrix Multiplication vs. Regular Matrix Multiplication

The latency of sparse matrix multiplication operation calculated from test values comprising of different matrix sizes from 10×10 to 100×100 with different sparsity pattern and the sparsity percentage ranges from 1% to 10% are plotted in the form of graph which illustrates, we were able to achieve low latency for the proposed algorithm for small and large matrices and low and high sparsity range. As the matrix size grows large, the number of operations increases. To correlate these operations, parallel processing in implemented in the algorithm along with pipelining with multiple processing elements. These optimizations have helped to improve the performance of the proposed algorithm.

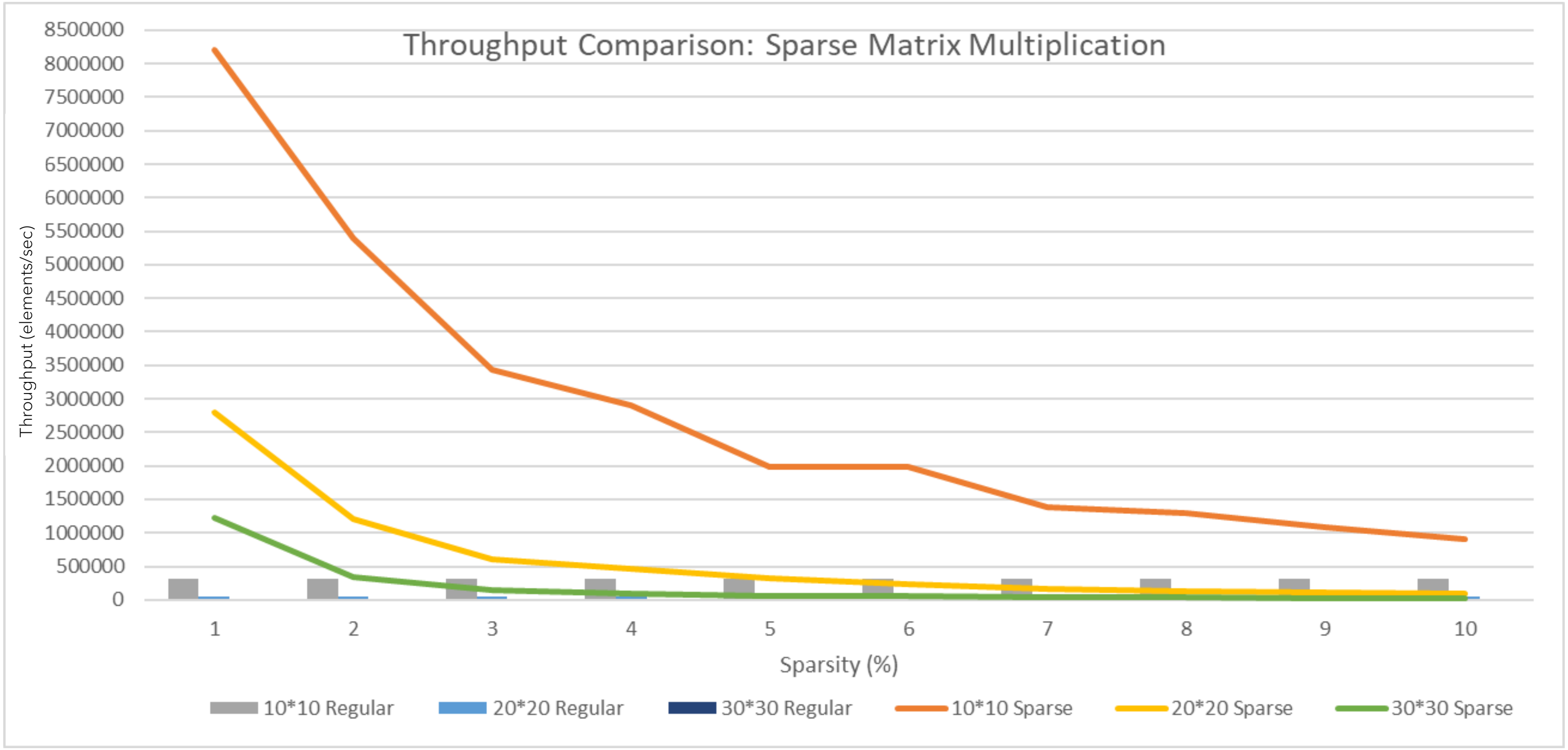


Figure 74. Throughput Comparison: Sparse Matrix Multiplication Operation

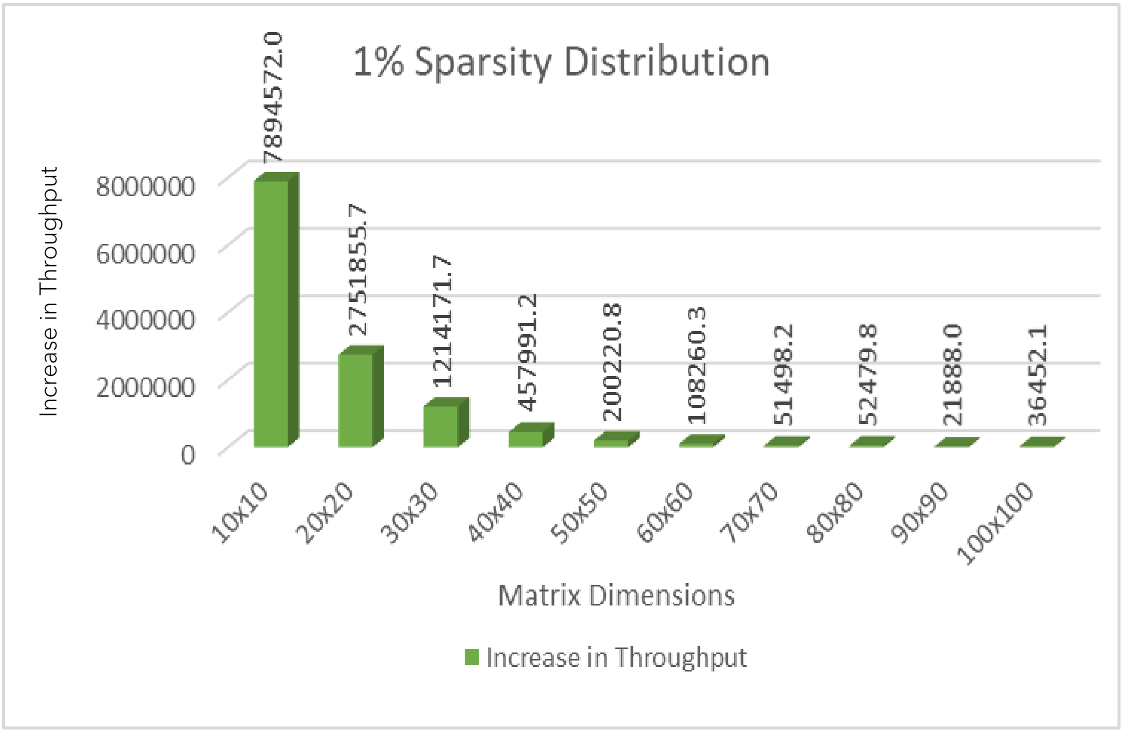


Figure 75. Increase in Throughput for Different Matrix Dimensions with 1% Sparsity

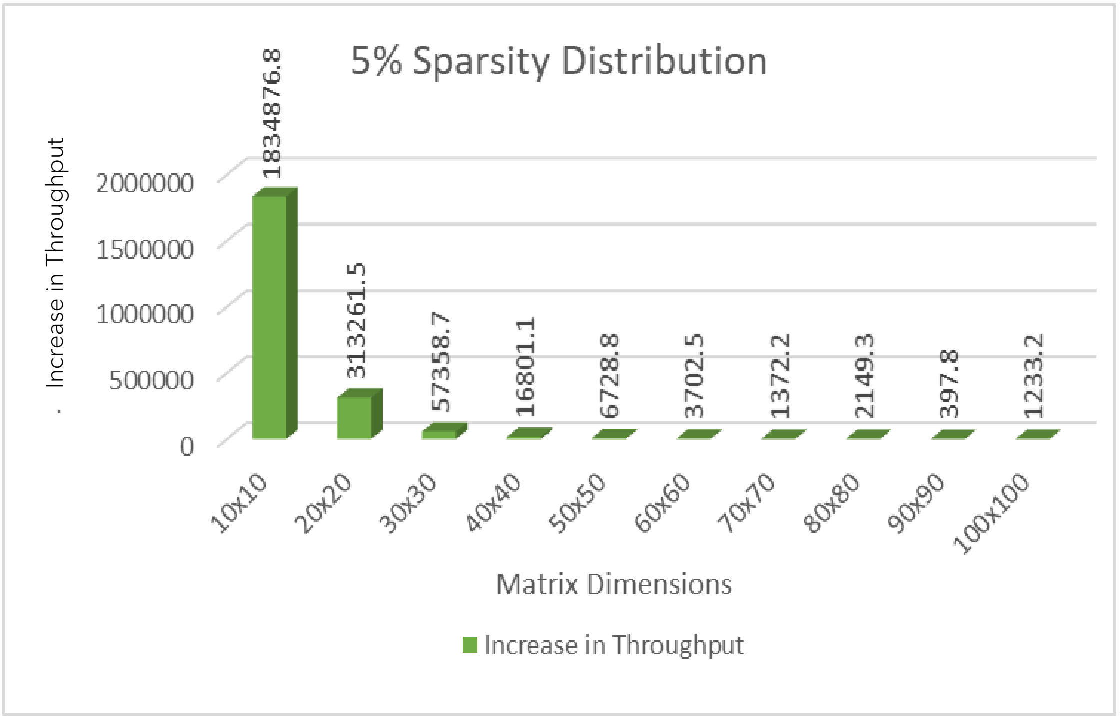


Figure 76. Increase in Throughput for Different Matrix Dimensions with 5% Sparsity

Figure 74 gives the increase in throughput for various dimensions of matrices when compared with the proposed and regular algorithm. High throughput was able to be achieved from the proposed design, which is evident from Figure 75, 76 and 77, even if the percentage of sparsity increases. The x-axis represents the increase in throughput whereas, the y-axis represents the size of the matrices used as test matrices.

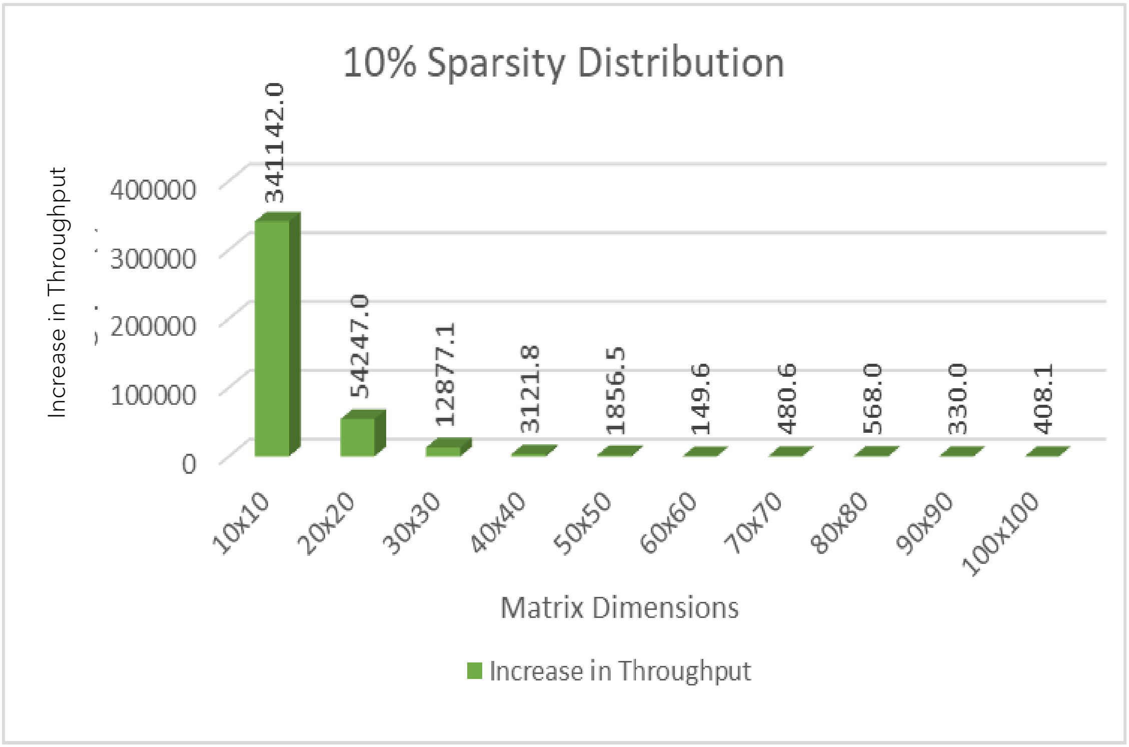


Figure 77. Increase in Throughput for Different Matrix Dimensions with 10% Sparsity

The hardware utilized on the Nexys 4 DDR Artix 7 FPGA board was reduced in the proposed sparse algorithm for matrix multiplication, and the comparison with traditional algorithm is shown in Figure 78.

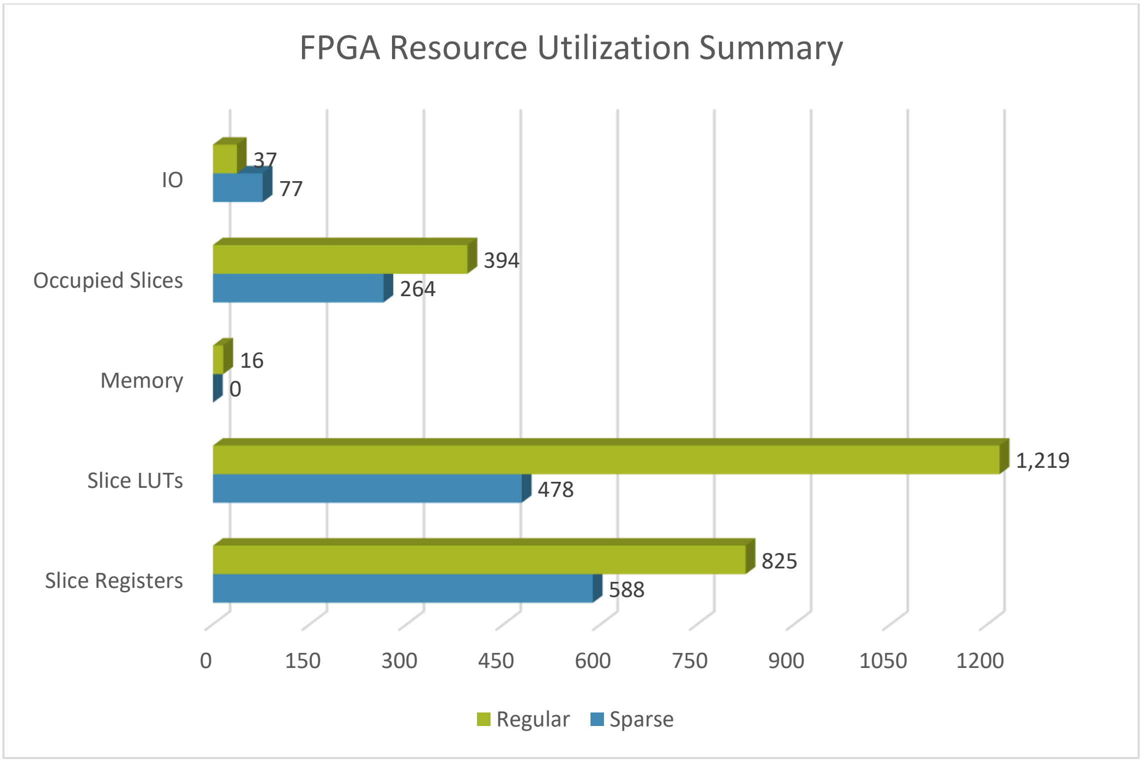
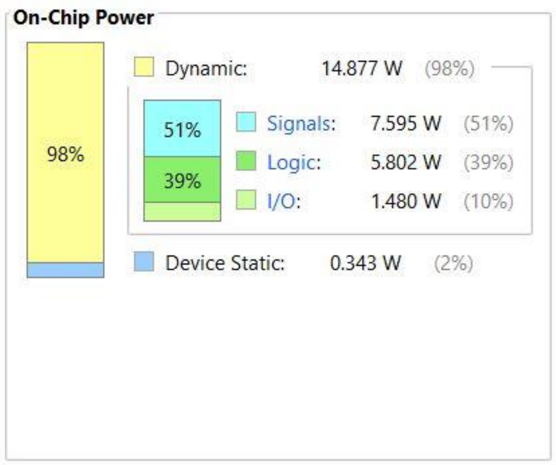


Figure 78. FPGA Resource Utilization: Regular vs Sparse-Sparse Matrix Multiplication Operation

Table 9 represents the best- and worst-case delays for the implemented design on the desired target FPGA board. The design was able to achieve a min of 3.79ns and a max of 15.03ns to fetch the input data, and this is dependent on how sparse the matrix is distributed. But once the data is fetched, and the operation is done the best- and worst-case delays are approx. 5ns.

Table 9. Hardware Implementation: Sparse Matrix Multiplication

	Input (ns)	Output (ns)
Best Case Delay (min)	3.79	5.21
Worst Case Delay (max)	15.03	5.89

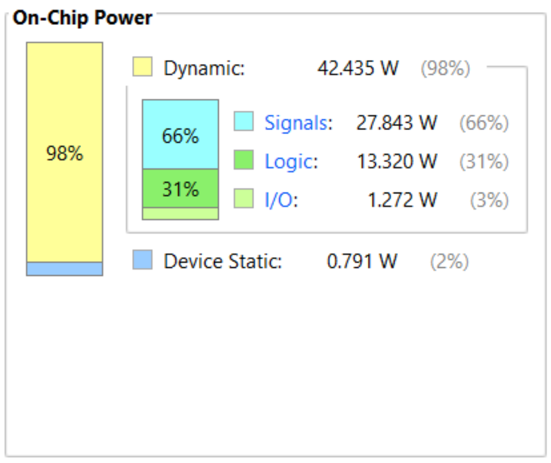


On-Chip Power

98%

Dynamic:	14.877 W	(98%)	
	Signals:	7.595 W	(51%)
	Logic:	5.802 W	(39%)
	I/O:	1.480 W	(10%)
Device Static:	0.343 W	(2%)	

(a) Proposed Method



On-Chip Power

98%

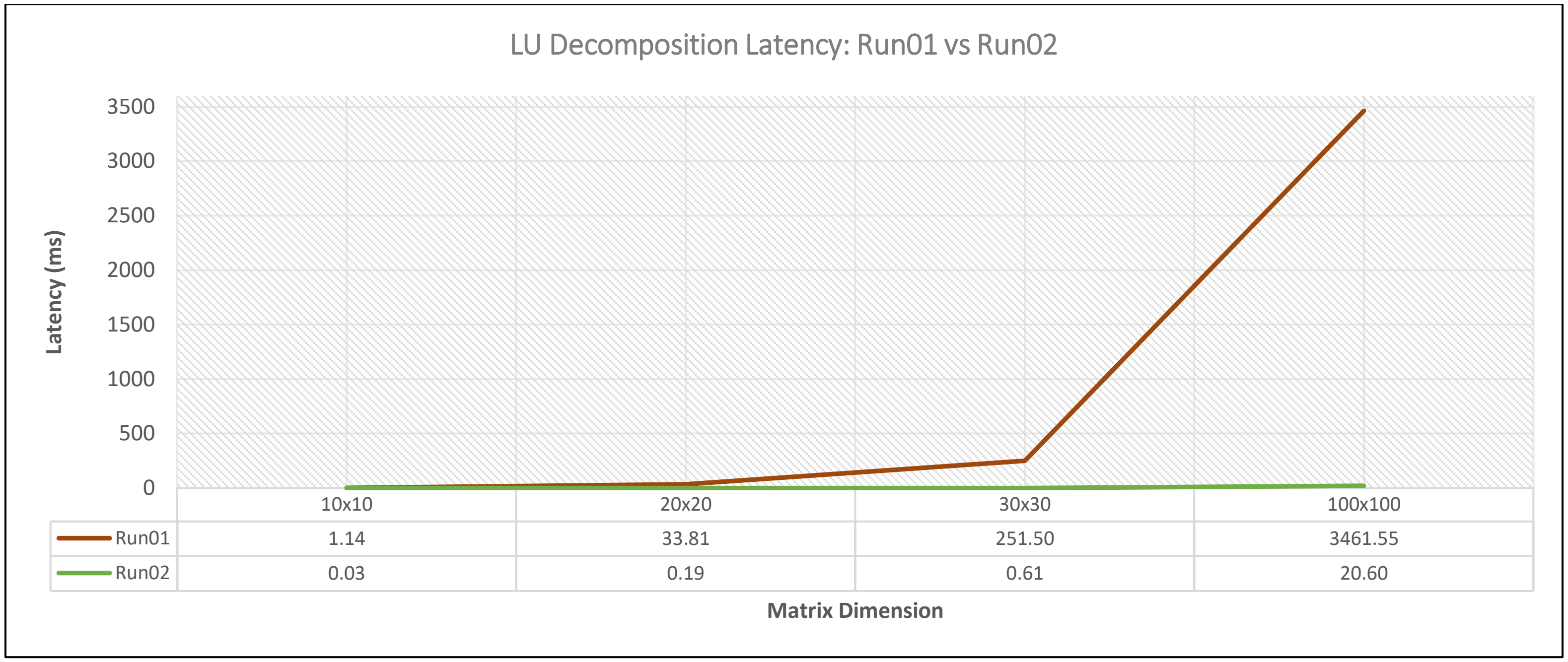
Dynamic:	42.435 W	(98%)	
	Signals:	27.843 W	(66%)
	Logic:	13.320 W	(31%)
	I/O:	1.272 W	(3%)
Device Static:	0.791 W	(2%)	

(b) Regular Method

Figure 79. Power Analysis: Sparse Matrix Multiplication Operation

Figure 79 shows a comparison of Power utilized on the FPGA board after implementation. Figure 79 (a) shows the approximate power consumption of the proposed implementation from the Vivado Power Analysis Tool, and figure 79 (b) shows the power consumption of the regular matrix multiplication operation on the same. By comparing the signal, logic and I/O power consumed, the proposed method only consumes a total of 14.877W whereas the other method consumes 42.43W. Some of the low configuration FPGAs do not support high power consumption which may lead to a failure.

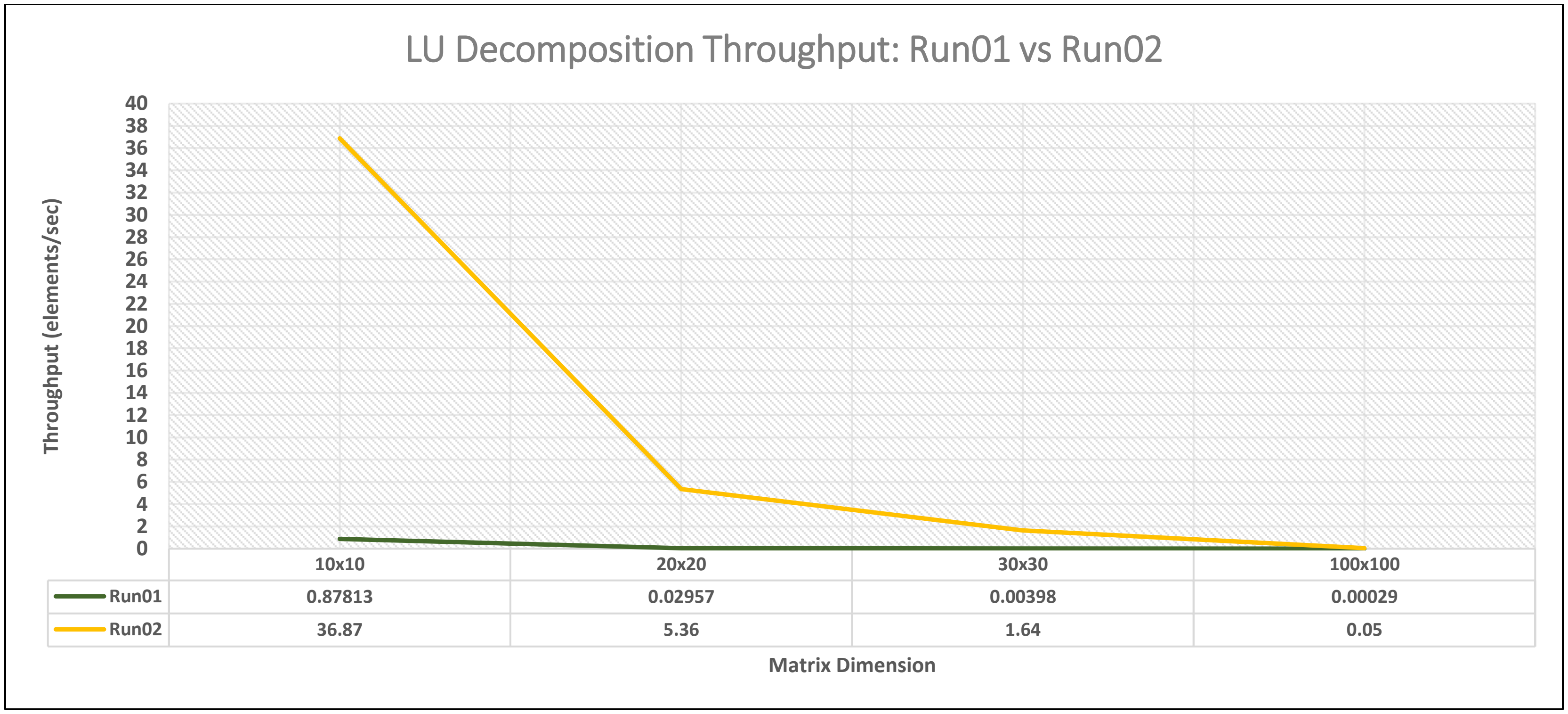
5.5 Sparse LU Decomposition vs. Regular LU Decomposition



	10x10	20x20	30x30	100x100
Run01	1.14	33.81	251.50	3461.55
Run02	0.03	0.19	0.61	20.60

Figure 80. Latency Comparison: Run01 vs Run02

Figure 80 shows a comparison of LU decomposition of the sparse matrix of size ranging from 10×10 to 100×100 with different sparsity range of 10% to 100%. The LU decomposition proposed design was able to achieve lower latency than the regular LU Decomposition algorithm. The results are also verified with the MATLAB LU Decomposition outputs for precision loss.



	10x10	20x20	30x30	100x100
Run01	0.87813	0.02957	0.00398	0.00029
Run02	36.87	5.36	1.64	0.05

Figure 81. Throughput Comparison: Run01 vs Run02

A comparison of the throughput calculated from sparse matrix algorithm and regular algorithm are plotted in the form of graph and is represented in Figure 81. As the performance needs to be high, we can infer from the graph high throughput was achieved.

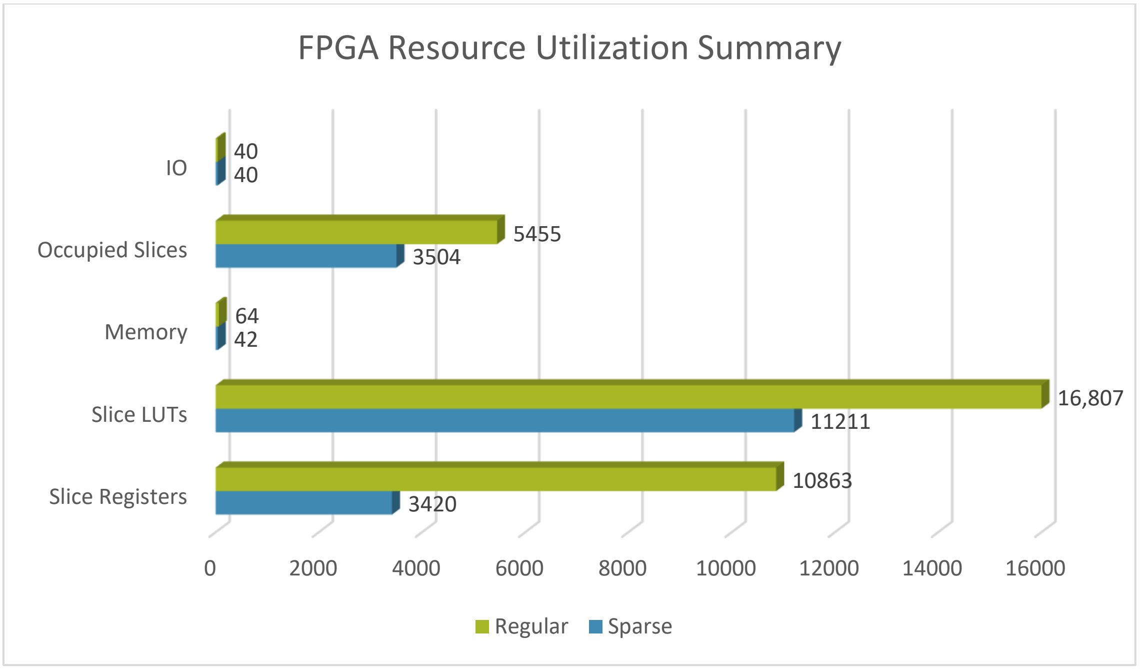
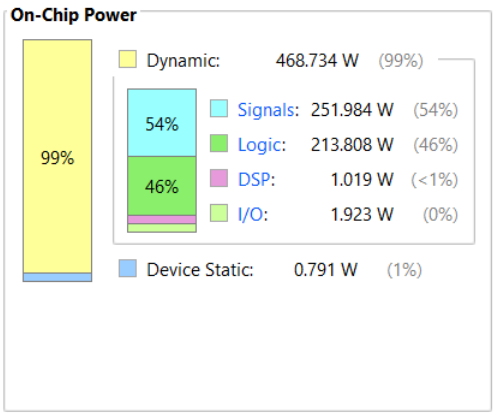


Figure 82. FPGA Resource Utilization: Regular vs Sparse-Sparse Matrix LU Decomposition

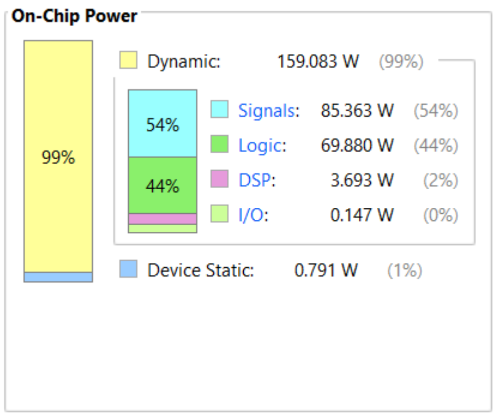
The Figure 82 represents the matrix storage format proposed in this research work was able to achieve the minimum resource utilization than the traditional regular algorithm. The resources utilized for LU decomposition was reduced with optimization throughout the HDL design programmed for the operation. A difference in about 1/3rd was achieved with the proposed algorithm.

Figure 83 depicts the power consumption of the proposed and regular design for comparison. The Figure 83(a) shows the static and dynamic power consumed by the proposed design where the dynamic power is 159.083W and Figure 83(b) shows the same for the regular method and the dynamic power is 468.734W. This clearly indicates the proposed method can achieve less than the other.



On-Chip Power

Dynamic:	468.734 W	(99%)
Signals:	251.984 W	(54%)
Logic:	213.808 W	(46%)
DSP:	1.019 W	(<1%)
I/O:	1.923 W	(0%)
Device Static:	0.791 W	(1%)



On-Chip Power

Dynamic:	159.083 W	(99%)
Signals:	85.363 W	(54%)
Logic:	69.880 W	(44%)
DSP:	3.693 W	(2%)
I/O:	0.147 W	(0%)
Device Static:	0.791 W	(1%)

(a) Regular Method (b) Proposed Method

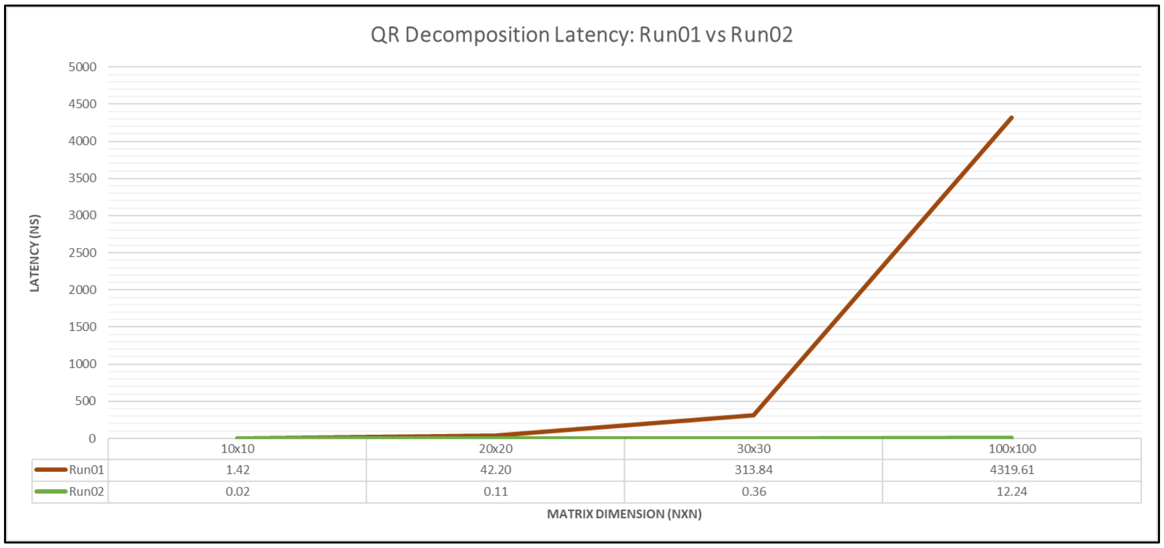
Figure 83. Power Analysis: Sparse Matrix LU Decomposition

Table 10. Hardware Implementation: LU Decomposition

	Input (ns)	Output (ns)
Best Case Delay (min)	7.41	6.54
Worst Case Delay (max)	19.68	6.71

Table 10 shows the best- and worst-case delays for the input and output to be available before and after LU factorization. These delays are calculated once the design is implemented and analyzed with the timing constraints too. Each path, with the logic and nets are analyzed from input and output to determine the maximum and minimum delay for the design.

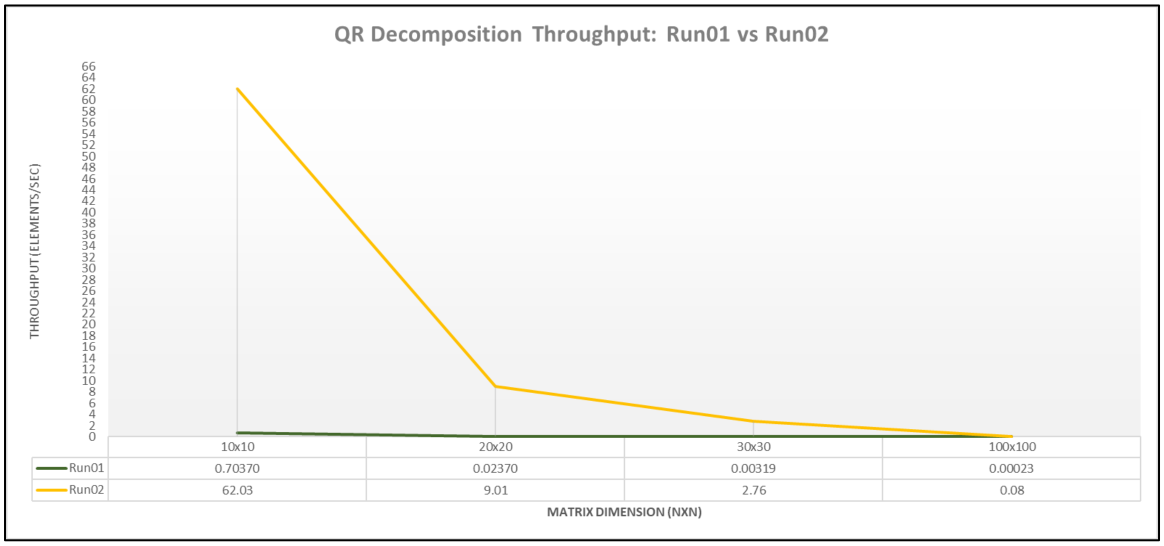
5.6 Sparse QR Decomposition vs. Regular QR Decomposition



	10x10	20x20	30x30	100x100
Run01	1.42	42.20	313.84	4319.61
Run02	0.02	0.11	0.36	12.24

Figure 84. Latency Comparison: Run01 vs Run02

From the performance metrics, latency is calculated for various matrix sizes from 10×10 to 100×100 with different sparsity percentage of 10% to 50% for both proposed algorithm and the regular algorithm, and it is evident from Figure 84 the graph indicates low latency has been achieved. In the case of a 10×10 matrix, the design was producing at least 8.5 times lower latency and for a 100×100 matrix the latency is improved. These calculations might not be the same which the same size of matrix and with different sparsity as the results are unique because of the irregular pattern of sparse matrices.



	10x10	20x20	30x30	100x100
Run01	0.70370	0.02370	0.00319	0.00023
Run02	62.03	9.01	2.76	0.08

Figure 85. Throughput Comparison: Run01 vs Run02

Figure 85 depicts the throughput comparison of the proposed algorithm and regular QR algorithm for matrices of sizes 10×10 to 100×100 with varying sparsity pattern and sparsity percentage ranging between 10% and 50%. High throughput was achieved, and it is shown in Figure 85.

Table 11. Hardware Implementation: QR Decomposition

	Input (ns)	Output (ns)
Best Case Delay (min)	6.46	8.27
Worst Case Delay (max)	16.4	5.73

Table 11 shows the comparison of minimum and maximum delay produced by the design for input and output. These results are calculated by analyzing the shortest and longest paths for the implemented design.

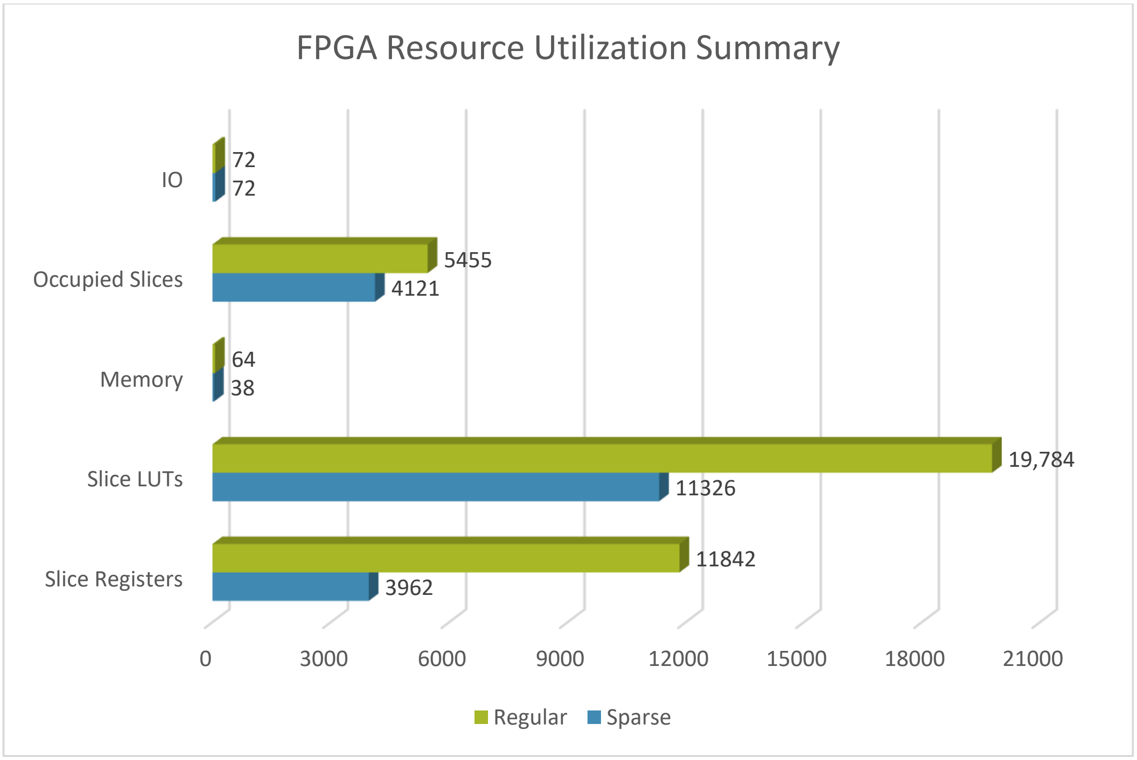


Figure 86. FPGA Resource Utilization: Regular vs Sparse-Sparse Matrix QR Decomposition

One of the focus of the thesis is to reduce the resources utilized for sparse matrix operation when implemented on FPGA board. Figure 86 shows, how much utilization was being able to be cut with the proposed algorithm from traditional QR decomposition methods. Although, the implementation of matrix operations on FPGA can be done with deep pipelining, for such decomposition algorithms like LU and QR which involves a lot of computations at each stage is a complicated process.

5.7 Execution Time Analysis

Benchmark matrices were downloaded from the University of Florida sparse matrix collection as test matrices for performance evaluation.

The Figure 87 shows the comparison the execution time of the benchmark matrices using MATLAB and FPGA implementation.

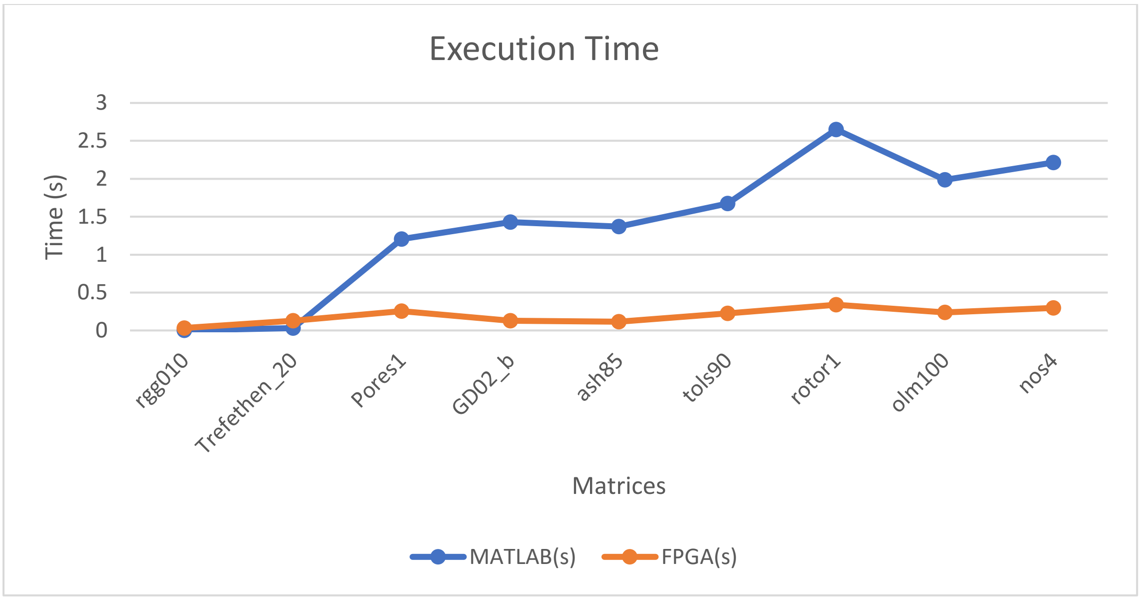


Figure 87. Comparison of execution time between MATLAB and FPGA

The Table 12 below represents the matrices and their properties.

Table 12. Benchmark matrices, properties, and pattern

Name	Dimensions	Application Domain	Nonzero	Symmetric
rgg010	10×10	Counter Example Problem	76	No
Trefethen_20	20×20	Combinatorial Problem	158	Yes
Pores1	30×30	Computational Fluid Dynamics Problem	180	No
GD02_b	80×80	Directed Graph	232	No
ash85	85×85	Least Square Problem	523	Yes
tols90	90×90	Computational Fluid Dynamics Problem	1746	No
rotor1	100×100	Structural Problem	708	No
olm100	100×100	Computational Fluid Dynamics Problem	396	No
nos4	100×100	Structural Problem	594	Yes

To quantify the efficiency of the architecture proposed, we simulated them using large varieties of sparse matrices. According to the simulation results, the execution time efficiency of the proposed FPGA Algorithmic is better than software implementation (in this case we compared with MATLAB). The efficient implementation of sparse matrix operations becomes more critical when applied to larger problems. This thesis work investigates the merits of implementing various sparse matrix operations on reconfigurable architecture. Each operation has a computational complexity, and as we can see the efficiency grows with the number of nonzero. The implementation of sparse matrix-matrix operations will benefit with further optimization. The efficiency of the proposed architecture combined with the algorithmic optimization greatly reduces the BRAM resource utilization achieving high throughput.

Opportunities for future work includes increasing the simulations for a variety of benchmark matrices from all application domains and exploring further optimization.

6. CONCLUSION AND FUTURE WORK

The overall design was successful as the results were demonstrated with data from the implementation of various sparse matrix operations. When comparing the performance to the regular algorithms and implementation, a significant achievement was made in performance and improved upon. The following sections will discuss our future work by improving areas where improvements are required for interesting applications to be designed. Finally, conclusion will be provided encapsulating the entire work.

Updating the FPGA might be significant improvement by placing on latest computers, performance improvement and speedups would be feasible in several areas. With the much available Block RAM, it is even possible for the operations to be implemented on large matrices. Adding to this, architectural improvement over the design or layout on the FPGA would add capability for allowing multiple designs based on the structure of the sparse matrices. Many of the previous designs would require pre-processing of sparse matrices on the software side and would require more research into how to implement efficiently.

The design has simple and scalable implementation that consists of a small number of input and output parameters. The University of Florida Sparse Matrix Collection contains over 1800 matrices and the other source is the Matrix Market which represents real problems that arise in various application domains such as fluid dynamics, finite element analysis, computational problems, least square problem, counter example problem, and structural problem. As large sparse matrices arise, it is difficult to find a proper and suitable algorithm and implementation for performance improvement. The following can be concluded from the thesis results.

1. The algorithm effectiveness depends on the sparsity of the matrices. When the number of nonzero is more, the number of calculations also increases.
2. Both input matrices are stored in the storage format illustrated, requiring less amount of memory.
3. For large sparse problems, parallelism is essential to reduce storage requirements, number of computations and execution of the program.

Today's applications require higher computational throughput and distributed memory approach for real-time applications. We have explored the optimizations not only for a specific application domain, but to make a generic architecture to be implemented irrespective of the application domain. This depends on several factors such as the sparsity, dimension of the matrix, irregular patterns of the nonzero elements, available resources with better clock frequency and bandwidth. The research work is primarily to design an optimized architecture for sparse matrix operations, allowing it to be more efficient than regular operations. Research improvement in this area is needed for increase in logic resources by comparable increase in I/O bandwidth and on-chip memory capacity, especially when the matrix sparsity is unstructured and randomly distributed. It would be interesting to seek further optimization to obtain efficient hybrid algorithms for different arbitrary matrices.

APPENDIX SECTION

SPARSE MATRIX ADDITION

```
`timescale 1ns / 1ps
module sparse(
    input Clk,
    input reset,
    input input_valid,
    input [7:0] A_elem,B_elem,
       input [7:0] A_r,A_c,B_r,B_c,
    output reg output_valid,
    output reg [7:0] row,col,
    output reg [8:0] sum
    );
parameter MAT_SIZE = 10;
parameter ELEMENT_SIZE = 8;
reg [7:0] A_sr [0:MAT_SIZE*MAT_SIZE-1];
reg [7:0] B_sr [0:MAT_SIZE*MAT_SIZE-1];
reg [7:0] A_sc [0:MAT_SIZE*MAT_SIZE-1];
reg [7:0] B_sc [0:MAT_SIZE*MAT_SIZE-1];
reg [ELEMENT_SIZE-1:0] A_sv [0:MAT_SIZE*MAT_SIZE-1];
reg [ELEMENT_SIZE-1:0] B_sv [0:MAT_SIZE*MAT_SIZE-1];
reg [15:0] A_count, B_count,A_index,B_index;
parameter s0 = 0,s1 = 1,s2 = 2,s3 = 3, s4 = 4, s5 = 5,s6 = 6, s7 = 7;
reg [2:0] state = 0;
reg [15:0] i;
reg [7:0] r1,c1,r2,c2,m,n;
task index2rc(input [15:0] index, output reg [7:0] r, output reg [7:0]
c);
      begin
            r = index/MAT_SIZE;
            c = index - r*MAT_SIZE;
      end
endtask
      always@(posedge Clk or posedge reset)
            begin
                  if(reset) begin
                        state <= s0;
                        A_count <= 0;
                        B_count <= 0;
                        i <= 0;
                  end else
      case (state)
            s0: begin
                  output_valid <= 0;
                  A_count <= 0;
                  B_count <= 0;
                  if(input_valid == 1)
                        state <= s1;
                  else
                        state <= s0;
            end
            s1: begin
                  if(A_elem != 0) begin
                        A_sv[A_count] <= A_elem;
                        A_sr[A_count] <= A_r;
                        A_sc[A_count] <= A_c;
                        A_count <= A_count + 1;
                  end
                  if(B_elem != 0) begin
                        B_sv[B_count] <= B_elem;
                        B_sr[B_count] <= B_r;
                        B_sc[B_count] <= B_c;
                        B_count <= B_count + 1;
                  end
                  if(input_valid == 0) begin
                        state <= s3;
                  end
                  A_index <= 0;
                  B_index <= 0;
            end
            s3: begin
                  if(A_index >= A_count) begin
                         state <= s4;
                         A_index <= 0;
                         B_index <= 0;
                         end else begin
                         if(B_index == B_count-1) begin
                              B_index <= 0;
                              A_index <= A_index + 1;
                        end else
                          B_index <= B_index + 1;
                  end
                  if(B_sv[B_index] != 0) begin
                  if( (A_sr[A_index] < B_sr[B_index]) || (
(A_sr[A_index] == B_sr[B_index]) && (A_sc[A_index] < B_sc[B_index]) ))
begin
                        B_index <= 0;
                        A_index <= A_index + 1;
                  end

                   if((A_sr[A_index] == B_sr[B_index]) &&
(A_sc[A_index] == B_sc[B_index])) begin
                        row <= A_sr[A_index];
                        col <= A_sc[A_index];
                        sum <= A_sv[A_index] - B_sv[B_index];
                        output_valid <= 1;
                        A_sv[A_index] = 0;
                        B_sv[B_index] = 0;
                        B_index <= 0;
                        A_index <= A_index + 1;
                  end else begin
                        output_valid <= 0;
                  end
                  end else begin
                        output_valid <= 0;
                  end
            end
            s4: begin
                  if(A_index == A_count) begin
                        state <= s5;
                        output_valid <= 0;
                  end else begin
                        A_index <= A_index + 1;
                        if(A_sv[A_index] != 0) begin
                        row <= A_sr[A_index];
                        col <= A_sc[A_index];
                        sum <= A_sv[A_index];
                        output_valid <= 1;
                  end else begin
                        output_valid <= 0;
                  end
                  end
            end
             s5: begin
                  if(B_index == B_count) begin
                        state <= s0;
                        output_valid <= 0;
                  end else begin
                        B_index <= B_index + 1;
                  if(B_sv[B_index] != 0) begin
                         row <= B_sr[B_index];
                        col <= B_sc[B_index];
                        sum <= B_sv[B_index];
                        output_valid <= 1;
                  end else begin
                        output_valid <= 0;
                  end
                  end
            end
      endcase
end
endmodule
```

SPARSE MATRIX MULTIPLICATION

```
`timescale 1ns / 1ps
module sparse(
    input Clk,
    input reset,
    input input_valid,
    input [7:0] mat_ip,
    output reg done,
    output reg output_valid,
    output reg [15:0] prod
    );
parameter MAT_SIZE = 10;
parameter ELEMENT_SIZE = 8;
reg [2*ELEMENT_SIZE-1:0] C [0:MAT_SIZE*MAT_SIZE-1];
reg [7:0] A_sr [0:MAT_SIZE*MAT_SIZE-1];
reg [7:0] B_sr [0:MAT_SIZE*MAT_SIZE-1];
reg [7:0] A_sc [0:MAT_SIZE*MAT_SIZE-1];
reg [7:0] B_sc [0:MAT_SIZE*MAT_SIZE-1];
reg [ELEMENT_SIZE-1:0] A_sv [0:MAT_SIZE*MAT_SIZE-1];
reg [ELEMENT_SIZE-1:0] B_sv [0:MAT_SIZE*MAT_SIZE-1];
reg [15:0] A_count, B_count,A_index,B_index;
reg [2*ELEMENT_SIZE-1:0] temp;
parameter s0 = 0,s1 = 1,s2 = 2,s3 = 3, s4 = 4, s5 = 5,s6 = 6, s7 = 7;
reg [2:0] state = 0;
reg [15:0] i;
reg [7:0] m,n;
task index2rc(input [15:0] index, output reg [7:0] r, output reg [7:0]
c);
      begin
            r = index/MAT_SIZE;
            c = index - r*MAT_SIZE;
      end
endtask
      always@(posedge Clk or posedge reset)
            begin
                  if(reset) begin
                        state <= s0;
                        done <= 1'b0;
                        A_count <= 0;
                        B_count <= 0;
                        i <= 0;
                  end else
            case (state)
                  s0: begin
                        done <= 1'b0;
                        output_valid <= 0;
                        B_count <= 0;
                        if(input_valid == 1) begin
                              state <= s1;
                              if(mat_ip != 0) begin
                                    index2rc(0,A_sr[0],A_sc[0]);
                                    A_sv[0] = mat_ip;
                                    A_count <= 1;
                              end else
                              A_count <= 0;
                              i <= 1;
                        end else begin
                        A_count <= 0;
                        state <= s0;
                        i <= 1;
                        end
                        end
                  s1: begin
                        if(input_valid == 1) begin
                              if(i != MAT_SIZE*MAT_SIZE) begin
                                    if(mat_ip != 0) begin

      index2rc(i,A_sr[A_count],A_sc[A_count]);
                                          A_sv[A_count] = mat_ip;
                                          A_count <= A_count + 1;
                                    end
                                    i <= i +1;
                              end else begin
                              state <= s2;
                              if(input_valid == 1) begin
                                    if(mat_ip != 0) begin
                                          index2rc(0,B_sr[0],B_sc[0]);
                                          B_sv[0] = mat_ip;
                                          B_count <= 1;
                                          i <= 1;
                                    end else begin
                                    i <= 1;
                                    B_count <= 0;
                              end
                              end else begin
                              i <= 0;
                              B_count <= 0;
                        end
                        end
                        end
                  end
                  s2: begin
                        if(input_valid == 1) begin
                              if(i != MAT_SIZE*MAT_SIZE-1) begin
                                    if(mat_ip != 0) begin

      index2rc(i,B_sr[B_count],B_sc[B_count]);
                                          B_sv[B_count] = mat_ip;
                                          B_count <= B_count + 1;
                                    end
                                    i <= i +1;
                                    end else begin
                                    state <= s3;
                                          if(input_valid == 1) begin
                                                if(mat_ip != 0) begin

      index2rc(i,B_sr[B_count],B_sc[B_count]);
                                                B_sv[B_count] = mat_ip;
                                                B_count <= B_count + 1;
                                                i <= 1;
                                                end
                                          end

                                    end
                                    end
                                    A_index <= 0;
                                    B_index <= 0;
                  end
                   s3: begin
                        A_sr[A_count] = 0;
                        A_sc[A_count] = 0;
                        A_sv[A_count] = 0;
                        B_sr[B_count] = 0;
                        B_sc[B_count] = 0;
                        B_sv[B_count] = 0;
                        m <= 0; //row
                        n <= 0;  //col
                        state <= s4;
                        temp <= 0;
                        A_index <= 0;
                        B_index <= 0;
                  end
                   s4: begin
                        if(A_sr[A_index] > m) begin
                              state <= s5;
                        end else if(A_sr[A_index] == m) begin
                              if(B_index < B_count)
                                    B_index <= B_index + 1;
```

```
                                        else
                                        if(A_index < A_count)
                                              A_index <= A_index + 1;
                                        else
                                        state <= s5;
                                        if(B_sr[B_index] > A_sc[A_index]) begin
                                              B_index <= 0;
                                              if(A_index < A_count)
                                              A_index <= A_index + 1;
                                        else
                                        state <= s5;
                                        end else if((B_sc[B_index] == n) &&
(B_sr[B_index] == A_sc[A_index])) begin
                                                          temp <= temp +
A_sv[A_index]*B_sv[B_index];
                                                          if(A_index < A_count)
                                                                A_index <=
                                                                
A_index + 1;
                                                          else
                                                                state <= s5;
                                                    end
                                              end else
                                                    if(A_index < A_count)
                                                          A_index <= A_index + 1;
                                                    else
                                                          state <= s5;

                                    end
                                    s5: begin
                                    //increment row_col indices to output matrix.
                                              C[m*MAT_SIZE+n] <= temp;
                                              A_index <= 0;
                                              B_index <= 0;
                                              temp <= 0;
                                              if(n == MAT_SIZE-1) begin
                                                    n <= 0;
                                                    if(m == MAT_SIZE-1) begin
                                                          m <= 0;
                                                          state <= s6;
                                                          i <= 0;
                                                    end else begin
                                                          m <= m+1;
                                                          state <= s4;
                                                    end
                                              end else begin
                                                    n <= n+1;
                                                    state <= s4;
                                              end
                                    end
                                    s6: begin
                                              if(i == MAT_SIZE*MAT_SIZE) begin
                                                     state <= s0;
                                                     done <= 1'b1;
                                                     output_valid <= 0;
                                              end else begin
                                                     i <= i + 1;
                                                     prod <= C[i];
                                        output_valid <= 1;
                                 end
                          end
                   endcase
             end


endmodule
```

LU DECOMPOSITION

```
`timescale 1ns / 1ps
module lu_decomp(
    input Clk,
    input reset,
    input input_valid,
    input [3:0] A_elem,
    output reg output_valid,
    output reg [15:0] L_elem,
    output reg [15:0] U_elem
    );
parameter n = 10;
reg signed [15:0] L [0:n*n-1];
reg signed [15:0] U [0:n*n-1];
reg signed [15:0] temp_1D [0:n-1];
parameter s0 = 0,s1 = 1,s2 = 2,s3 = 3, s4 = 4, s5 = 5,s6 = 6, s7 = 7,
s8 = 8, s9 = 9, s10 = 10, s11 = 11, s12 = 12;
reg [3:0] state = 0;
reg [7:0] i,j,p,m;
reg [15:0] L_index,U_index;
reg [15:0] pivot;
function [15:0] rc2index;
input [7:0] row,col;
begin
      rc2index = (row-1)*n+(col-1);
end
endfunction
function [15:0] abs;
input [15:0] num;
begin
if(num[15] == 1'b1)
      abs = -num;
else
      abs = num;
end
endfunction
function [15:0] resize;
input [31:0] num;
reg [31:0] num2;
reg [15:0] num3;
begin
      if(num[31] == 1'b0)
            resize = num[23:8];
      else begin
            num2 = -num;
            num3 = num2[23:8];
            resize = -num3;
      end
end
endfunction

      always@(posedge Clk or posedge reset)
            begin
                  if(reset) begin
                        state <= s0;
                        i <= 1;
                        j <= 1;
                        m <= 1;
                        p <= 1;
                        pivot <= 0;
                  end else
                        case (state)
                              s0: begin
                                    output_valid = 0;
                                    i <= 1;
                                    j <= 1;
                                    m <= 1;
                                    p <= 1;
                                    pivot <= 0;
                                    L_index = 0;
                                    if(input_valid == 1) begin
                                           state <= s1;
                                           U[0] <= A_elem*256;
                                           U_index <= 1;
                                    end else
                                           state <= s0;
                              end
                              s1: begin
                                    U[U_index] <= A_elem*256;
                                    if(U_index == n*n) begin
                                          state <= s2;
                                          U_index <= 0;
                                    end else
                                          U_index <= U_index +1;
                              end
                              s2: begin
                                    L_index = rc2index(i,j);
                                    if(i == j)
                                          L[L_index] <= 1*256;
                                    else
                                          L[L_index] <= 0;
                                    if(j == n) begin
                                          j <= 1;
                                          if(i == n) begin
                                                state <= s3;
                                          end else
                                                i <=  i+1;
                                    end else
                                          j <= j+1;
                              end
                              s3: begin
                                    pivot<= 0;
                                    m <= 1;
                                    p <= j;
                                    i <= 1;
                                    state <= s4;
                              end
                              s4: begin
                                    if(pivot < abs(U[rc2index(p,j)]))
begin
                                          pivot <=
abs(U[rc2index(p,j)]);
                                          m <= p;
                                    end
                                    if(p == n) begin
                                          p <= 1;
                                          state <= s6;
                                    end else
                                          p <= p+1;
                              end
                              s6: begin
                                    if(m != j)
                                          state <= s7;
                                    else begin
                                          if(j == n)
                                                state <= s11;
                                          else begin
                                                i <= j+1;
                                                state <= s9;
                                          end
                                    end
                              end
                              s7: begin
                                    if(p == n) begin
                                          p <= 1;
                                          if(j >= 2)
                                                state <= s8;
                                          else begin
                                                state <= s9;
                                                i <= j+1;
                                          end
                                    end else
                                          p <= p +1;
                                    U[rc2index(m,p)] <=
U[rc2index(j,p)];
                                    U[rc2index(j,p)] <=
U[rc2index(m,p)];
                              end
                              s8: begin
                                    if(p == j-1) begin
                                          p <= 1;
                                          if(j < n) begin
                                                state <= s9;
                                                i <= j+1;
                                          end else
                                                state <= s11;
                                    end else
                                          p <= p +1;
                                    L[rc2index(m,p)] <=
L[rc2index(j,p)];
```

```
                                    L[rc2index(j,p)] <=
L[rc2index(m,p)];
                              end
                              s9: begin
                                    L[rc2index(i,j)] =
(U[rc2index(i,j)]*256) / U[rc2index(j,j)];
                                    state <= s10;
                              end
                              s10: begin
                                    if(p == n) begin
                                          p <= 1;
                                          state <= s12;
                                    end else
                                          p <= p+1;
                                    temp_1D[p-1] = U[rc2index(i,p)] -
resize(L[rc2index(i,j)]*U[rc2index(j,p)]);

                                    L_index = 0;
                              end
                              s12: begin
                                    if(p == n) begin
                                          p <= 1;
                                          if(i == n) begin
                                                if(j == n) begin
                                                      state <= s11;
                                                      j <= 0;
                                                end else begin
                                                      j <= j+1;
                                                      state <= s3;
                                                end
                                                i <= 1;
                                          end else begin
                                                state <= s9;
                                                i <= i+1;
                                          end

                                    end else
                                          p <= p +1;
                                    U[rc2index(i,p)] <= temp_1D[p-1];
                              end
                              s11: begin
                                    L_elem = L[L_index];
                                    U_elem = U[L_index];
                                    if(L_index == n*n)
                                           state <= s0;
                                    else
                                           L_index = L_index + 1;
                                    output_valid = 1;
                              end
                   endcase
            end

endmodule
```

QR DECOMPOSITION

```
`timescale 1ns / 1ps
```

```
module QR_decomp(
    input Clk,
    input reset,
    input input_valid,
    input [3:0] A_elem,
    output reg output_valid,
    output reg [width-1:0] Q_elem,
    output reg [width-1:0] R_elem
    );

parameter n = 10;
parameter width = 32;
parameter dec = 20;
parameter frac = 12;

reg signed [width-1:0] Q [0:n*n-1];
reg signed [width-1:0] R [0:n*n-1];
reg signed [width-1:0] x [1:n];
reg signed [width-1:0] u [1:n];
reg signed [2*width-1:0] square_sum;
reg signed [width-1:0] temp1;
reg signed [width-1:0] temp2 [0:n*n-1];
reg signed [width-1:0] temp3 [0:n*n-1];
reg signed [width-1:0] s;
wire signed [width-1:0] sq_out;

parameter s0 = 0,s1 = 1,s2 = 2,s3 = 3, s4 = 4, s5 = 5,s6 = 6, s7 = 7,
s8 = 8, s9 = 9, s10 = 10,
                  s11 = 11, s12 = 12, s13 = 13, s14 = 14, s15 = 15, s16
= 16, s17= 17, s18 = 18;
reg [4:0] state = 0;
reg [7:0] i,j,k,p;
reg [15:0] Q_index,R_index;
reg rst_sq,sq_root_en;
wire sq_out_en;
function [15:0] rc2index;
input [7:0] row,col;
begin
      rc2index = (row-1)*n+(col-1);
end
endfunction
function signed [width-1:0] resize;
input signed [2*width-1:0] num;
reg signed [2*width-1:0] num2;
reg signed [width-1:0] num3;
begin
      resize = num[2*frac+dec-1:frac];
end
endfunction
sq_root #(2*width)
square1(Clk,rst_sq,sq_root_en,square_sum,sq_out,sq_out_en);
      always@(posedge Clk or posedge reset)
            begin
                  if(reset) begin
                        state <= s0;
                        i <= 1;
                        j <= 1;
```

```
        rst_sq <= 1;
        output_valid = 0;
end else
        case (state)
                s0: begin
                        output_valid = 0;
                        i <= 1;
                        j <= 1;
                        p <= 1;
                        rst_sq <= 1;
                        R_index = 0;
                        if(input_valid == 1) begin
                                 state <= s1;
                                 R[0] = A_elem*(2**frac);
                                 R_index = 1;
                        end else
                                 state <= s0;
                end
                s1: begin
                        R[R_index] = A_elem*(2**frac);
                        if(R_index == n*n) begin
                                state <= s2;
                                R_index = 0;
                        end else
                                R_index = R_index +1;
                end
                s2: begin
                        Q_index = rc2index(i,j);
                        if(i == j)
                                Q[Q_index] = 1*(2**frac);
                        else
                                Q[Q_index] = 0;
                        if(j == n) begin
                                j <= 1;
                                if(i == n) begin
                                        state <= s3;
                                        k <= 0;
                                end else
                                        i <=  i+1;
                        end else
                                j <= j+1;
                end
                s3: begin
                        i <= 1;
                        j <= 1;
                        if(k == n-1)
                                state <= s18;
                        else begin
                                k <= k+1;
                                state <= s4;
                        end
                        Q_index = 0;
                end
                s4: begin
                        if(i == n) begin
                                i <= k;
                                state <= s5;
```

```
                                    end else
                                          i <= i+1;
                                    x[i] = 0;
                              end
                              s5: begin
                                    if(i == n) begin
                                          i <= 1;
                                          square_sum <= 0;
                                          state <= s6;
                                    end else
                                          i <= i+1;
                                    x[i] = R[rc2index(i,k)];
                              end
                              s6: begin
                                    if(i == n) begin
                                          i <= 1;
                                          state <= s7;
                                          rst_sq <= 0;

                                    end else
                                          i <= i+1;
                                    square_sum <= square_sum +
x[i]*x[i];
                              end
                              s7: begin
                                    if(sq_out_en == 1'b1) begin
                                          state <= s9;
                                          rst_sq <= 1;
                                          sq_root_en <= 1'b0;
                                          square_sum <= 0;
                                          x[k] = x[k] + sq_out;
                                    end else
                                          sq_root_en <= 1'b1;
                              end
                              s9: begin
                                    if(i == n) begin
                                          i <= 1;
                                          state <= s10;
                                          rst_sq <= 0;

                                    end else
                                          i <= i+1;
                                    square_sum <= square_sum +
x[i]*x[i];
                              end
                              s10: begin
                                    if(sq_out_en == 1'b1) begin
                                          s = sq_out;
                                          rst_sq <= 1;
                                          if(s != 0)
                                                state <= s12;
                                          else
                                                state <= s3;
                                          sq_root_en <= 1'b0;
                                    end else
                                          sq_root_en <= 1'b1;
                              end
                              s12: begin
                                    if(i == n) begin
                                          i <= 1;
                                          state <= s13;
                                          temp1 = 0;
                                    end else
                                          i <= i+1;
                                    x[i] = (x[i]*(2**frac))/s;
                              end
                              s13: begin
                                    temp1 = temp1 +
(R[rc2index(j,i)]*x[j])/(2**frac);
                                    if(j == n) begin
                                          j <= 1;
                                          u[i] = 2*temp1;
                                          temp1 = 0;
                                          if(i == n) begin
                                                state <= s14;
                                                i <= 1;
                                          end else
                                                i <= i+1;
                                    end else
                                          j <= j+1;
                              end
                              s14: begin
                                    R[rc2index(i,j)] = R[rc2index(i,j)]
- (x[i]*u[j])/(2**frac);
                                    if(j == n) begin
                                          j <= 1;
                                          temp1 = 0;
                                          if(i == n) begin
                                                state <= s15;
                                                i <= 1;
                                          end else
                                                i <= i+1;
                                    end else
                                          j <= j+1;
                              end
                              s15: begin
                                    if(j == n) begin
                                          j <= 1;
                                          if(i == n) begin
                                                state <= s16;
                                                temp1 = 0;
                                                p <= 1;
                                                i <= 1;
                                          end else
                                                i <= i+1;
                                    end else
                                          j <= j+1;
                                    temp2[rc2index(i,j)] =
(2*x[j]*x[i])/(2**frac);
                              end
                              s16: begin
                                    temp1 = temp1 +
(Q[rc2index(i,p)]*temp2[rc2index(p,j)])/(2**frac);
                                    temp3[rc2index(i,j)] <= temp1;
                                    if(p == n) begin
                                          p <= 1;
                                          temp1 = 0;
                                          if(j == n) begin
                                                j <= 1;
                                                if(i == n) begin
                                                      state <= s17;
                                                      i <= 1;
                                                end else
                                                      i <= i+1;
                                          end else
                                                j <= j+1;
                                    end else
                                          p <= p+1;
                              end
                              s17: begin
                                    Q[rc2index(i,j)] = Q[rc2index(i,j)]
- temp3[rc2index(i,j)];
                                    if(j == n) begin
                                          j <= 1;
                                          if(i == n) begin
                                                state <= s3;
                                                i <= 1;
                                          end else
                                                i <= i+1;
                                    end else
                                          j <= j+1;

                              end
                              s18: begin
                                    Q_elem = Q[Q_index];
                                    R_elem = R[Q_index];
                                    if(Q_index == n*n-1)
                                           state <= s0;
                                    else
                                           Q_index = Q_index + 1;
                                    output_valid = 1;
                              end
                 endcase
             end

endmodule

squareroot.v
`timescale 1ns / 1ps

module sq_root(
    input Clk,
    input reset,
    input input_valid,
             input [width-1:0] A,
             output reg [width/2-1:0] sq_out,
    output reg output_valid
    );

parameter width = 32;
reg [7:0] i;
reg [width/2+1:0] left =0,right=0,r=0;
reg [width-1:0] a;

      always@(posedge Clk or posedge reset)
            begin
                  if(reset) begin
                        i <= 0;
                        output_valid <= 0;
                        sq_out = 0;
                        left = 0;
                        right = 0;
                        r = 0;
                  end else
                        if(input_valid == 1'b1) begin
                              if(i == width/2-1)
                                    output_valid <= 1'b1;
                              else begin
                                    output_valid <= 1'b0;
                                    if(i == 0) begin
                                          a = A;
                                          sq_out = 0;
                                    end
                                    i <= i+1;
                              end
                              right = {sq_out,r[width/2+1],1'b1};
                              left = {r[width/2-1:0],a[width-1:width-
2]};

                              a[width-1:2] = a[width-3:0];
                              if(r[width/2+1] == 1'b1)
                                    r = left+right;
                              else
                                    r = left-right;
                              sq_out = {sq_out[width/2-
2:0],~r[width/2+1]};
                        end
            end



endmodule
```

INPUT

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.std_logic_unsigned.all;

entity UART_TX_CTRL is
    Port ( SEND : in  STD_LOGIC;
           DATA : in  STD_LOGIC_VECTOR (7 downto 0);
           CLK : in  STD_LOGIC;
           READY : out  STD_LOGIC;
           UART_TX : out  STD_LOGIC);
end UART_TX_CTRL;

architecture Behavioral of UART_TX_CTRL is
```

```
type TX_STATE_TYPE is (RDY, LOAD_BIT, SEND_BIT);

constant BIT_TMR_MAX : std_logic_vector(13 downto 0) :=
"00100000100010"; --2082 for 20 mhz clock. 10416 = (round(100MHz /
9600)) - 1
constant BIT_INDEX_MAX : natural := 10;

--Counter that keeps track of the number of clock cycles the current
bit has been held stable over the
--UART TX line. It is used to signal when the ne
signal bitTmr : std_logic_vector(13 downto 0) := (others => '0');

--combinatorial logic that goes high when bitTmr has counted to the
proper value to ensure
--a 9600 baud rate
signal bitDone : std_logic;

--Contains the index of the next bit in txData that needs to be
transferred
signal bitIndex : natural;

--a register that holds the current data being sent over the UART TX
line
signal txBit : std_logic := '1';

--A register that contains the whole data packet to be sent, including
start and stop bits.
signal txData : std_logic_vector(9 downto 0);

signal txState : TX_STATE_TYPE := RDY;

begin

--Next state logic
next_txState_process : process (CLK)
begin
      if (rising_edge(CLK)) then
            case txState is
            when RDY =>
                  if (SEND = '1') then
                        txState <= LOAD_BIT;
                  end if;
            when LOAD_BIT =>
                  txState <= SEND_BIT;
            when SEND_BIT =>
                  if (bitDone = '1') then
                        if (bitIndex = BIT_INDEX_MAX) then
                              txState <= RDY;
                        else
                              txState <= LOAD_BIT;
                        end if;
                  end if;
            when others=> --should never be reached
                  txState <= RDY;
            end case;
      end if;
end process;
```

```
bit_timing_process : process (CLK)
begin
      if (rising_edge(CLK)) then
            if (txState = RDY) then
                  bitTmr <= (others => '0');
            else
                  if (bitDone = '1') then
                        bitTmr <= (others => '0');
                  else
                        bitTmr <= bitTmr + 1;
                  end if;
            end if;
      end if;
end process;

bitDone <= '1' when (bitTmr = BIT_TMR_MAX) else
                        '0';
bit_counting_process : process (CLK)
begin
      if (rising_edge(CLK)) then
            if (txState = RDY) then
                  bitIndex <= 0;
            elsif (txState = LOAD_BIT) then
                  bitIndex <= bitIndex + 1;
            end if;
      end if;
end process;

tx_data_latch_process : process (CLK)
begin
      if (rising_edge(CLK)) then
            if (SEND = '1') then
                  txData <= '1' & DATA & '0';
            end if;
      end if;
end process;

tx_bit_process : process (CLK)
begin
      if (rising_edge(CLK)) then
            if (txState = RDY) then
                  txBit <= '1';
            elsif (txState = LOAD_BIT) then
                  txBit <= txData(bitIndex);
            end if;
      end if;
end process;

UART_TX <= txBit;
READY <= '1' when (txState = RDY) else '0';

end Behavioral;
```

OUTPUT

```
library ieee;
use ieee.std_logic_1164.ALL;
use ieee.numeric_std.all;

entity UART_RX_CTRL is
  generic (
    g_CLKS_PER_BIT : integer := 2082     -- Needs to be set correctly
    );
  port (
    i_Clk       : in  std_logic;
    i_RX_Serial : in  std_logic;
    o_RX_DV     : out std_logic;
    o_RX_Byte   : out std_logic_vector(7 downto 0)
    );
end UART_RX_CTRL;


architecture rtl of UART_RX_CTRL is

  type t_SM_Main is (s_Idle, s_RX_Start_Bit, s_RX_Data_Bits,
                     s_RX_Stop_Bit, s_Cleanup);
  signal r_SM_Main : t_SM_Main := s_Idle;

  signal r_RX_Data_R : std_logic := '0';
  signal r_RX_Data   : std_logic := '0';

  signal r_Clk_Count : integer range 0 to g_CLKS_PER_BIT-1 := 0;
  signal r_Bit_Index : integer range 0 to 7 := 0;  -- 8 Bits Total
  signal r_RX_Byte   : std_logic_vector(7 downto 0) := (others => '0');
  signal r_RX_DV     : std_logic := '0';

begin

  -- Purpose: Double-register the incoming data.
  -- This allows it to be used in the UART RX Clock Domain.
  -- (It removes problems caused by metastabiliy)
  p_SAMPLE : process (i_Clk)
  begin
    if rising_edge(i_Clk) then
      r_RX_Data_R <= i_RX_Serial;
      r_RX_Data   <= r_RX_Data_R;
    end if;
  end process p_SAMPLE;


  -- Purpose: Control RX state machine
  p_UART_RX : process (i_Clk)
  begin
    if rising_edge(i_Clk) then

      case r_SM_Main is

        when s_Idle =>
          r_RX_DV     <= '0';
          r_Clk_Count <= 0;
          r_Bit_Index <= 0;

          if r_RX_Data = '0' then       -- Start bit detected
            r_SM_Main <= s_RX_Start_Bit;
          else
            r_SM_Main <= s_Idle;
          end if;


        -- Check middle of start bit to make sure it's still low
        when s_RX_Start_Bit =>
          if r_Clk_Count = (g_CLKS_PER_BIT-1)/2 then
            if r_RX_Data = '0' then
              r_Clk_Count <= 0;  -- reset counter since we found the
middle
              r_SM_Main   <= s_RX_Data_Bits;
            else
              r_SM_Main   <= s_Idle;
            end if;
          else
            r_Clk_Count <= r_Clk_Count + 1;
            r_SM_Main   <= s_RX_Start_Bit;
          end if;


        -- Wait g_CLKS_PER_BIT-1 clock cycles to sample serial data
        when s_RX_Data_Bits =>
          if r_Clk_Count < g_CLKS_PER_BIT-1 then
            r_Clk_Count <= r_Clk_Count + 1;
            r_SM_Main   <= s_RX_Data_Bits;
          else
            r_Clk_Count            <= 0;
            r_RX_Byte(r_Bit_Index) <= r_RX_Data;

            -- Check if we have sent out all bits
            if r_Bit_Index < 7 then
              r_Bit_Index <= r_Bit_Index + 1;
              r_SM_Main   <= s_RX_Data_Bits;
            else
              r_Bit_Index <= 0;
              r_SM_Main   <= s_RX_Stop_Bit;
            end if;
          end if;


        -- Receive Stop bit.  Stop bit = 1
        when s_RX_Stop_Bit =>
          -- Wait g_CLKS_PER_BIT-1 clock cycles for Stop bit to finish
          if r_Clk_Count < g_CLKS_PER_BIT-1 then
            r_Clk_Count <= r_Clk_Count + 1;
            r_SM_Main   <= s_RX_Stop_Bit;
          else
            r_RX_DV     <= '1';
            r_Clk_Count <= 0;
            r_SM_Main   <= s_Cleanup;
          end if;


        -- Stay here 1 clock
        when s_Cleanup =>
          r_SM_Main <= s_Idle;
```

```
         r_RX_DV   <= '0';


        when others =>
          r_SM_Main <= s_Idle;

      end case;
    end if;
  end process p_UART_RX;

  o_RX_DV   <= r_RX_DV;
  o_RX_Byte <= r_RX_Byte;

end rtl;
```

Example for Error Analysis

LU Decomposition

A Matrix -10x10 50% Sparsity

0	7	0	5	0	4	0	0	9	0
2	3	9	0	6	0	0	4	3	0
0	0	2	2	0	0	3	0	1	2
3	0	4	0	3	0	0	0	7	4
0	9	0	0	3	9	0	7	1	0
4	0	3	0	0	0	0	3	8	0
2	4	0	0	7	0	2	0	0	6
0	0	5	4	8	3	0	0	4	5
2	3	0	0	0	0	5	3	4	0
0	0	6	0	0	2	0	5	3	1

L MATLAB

1	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0
0.5	0.333333	1	0	0	0	0	0	0	0
0	0.777778	0	1	0	0	0	0	0	0
0.5	0.444444	-0.2	0	1	0	0	0	0	0
0	0	0.666667	0.8	0.98	1	0	0	0	0
0.5	0.333333	-0.2	0	0	-0.30232	1	0	0	0
0.75	0	0.233333	0	0.275	0.165015	-0.05141	1	0	0
0	0	0.266667	0.4	-0.06	0.144777	0.772273	0.347572	1	0
0	0	0.8	0	-0.6	0.137723	0.333511	-0.12799	-0.28177	1

L Vivado

1	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0
0.5	0.332031	1	0	0	0	0	0	0	0
0	0.777344	0	1	0	0	0	0	0	0
0.5	0.441406	-0.19922	0	1	0	0	0	0	0
0	0	0.664063	0.796875	0.976563	1	0	0	0	0
0.5	0.332031	-0.19922	0	0	-0.30078	1	0	0	0
0.75	0	0.230469	0	0.273438	0.160156	-0.05078	1	0	0
0	0	0.265625	0.398438	-0.05859	0.144531	0.769531	0.332031	1	0
0	0	0.796875	0	-0.59375	0.140625	0.328125	-0.14844	-0.30469	1

L Difference

0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	-0.0013	0	0	0	0	0	0	0	0
0	-0.00043	0	0	0	0	0	0	0	0
0	-0.00304	0.000781	0	0	0	0	0	0	0
0	0	-0.0026	-0.00313	-0.00344	0	0	0	0	0
0	-0.0013	0.000781	0	0	0.001537	0	0	0	0
0	0	-0.00286	0	-0.00156	-0.00486	0.000625	0	0	0
0	0	-0.00104	-0.00156	0.001406	-0.00025	-0.00274	-0.01554	0	0
0	0	-0.00313	0	0.00625	0.002902	-0.00539	-0.02045	-0.02292	0

U MATLAB

4	0	3	0	0	0	0	3	8	0
0	9	0	0	3	9	0	7	1	0
0	0	7.5	0	5	-3	0	0.166667	-1.33333	0
0	0	0	5	-2.33333	-3	0	-5.44444	8.222222	0
0	0	0	0	6.666667	-4.6	2	-4.57778	-4.71111	6
0	0	0	0	0	11.908	-1.96	8.730667	2.928	-0.88
0	0	0	0	0	0	4.407457	1.839436	0.285186	-0.26604
0	0	0	0	0	0	0	-2.37613	2.138163	2.481537
0	0	0	0	0	0	0	1.11E-16	-3.60331	1.830346
0	0	0	0	0	2.22E-16	0	3.13E-17	0	5.643263

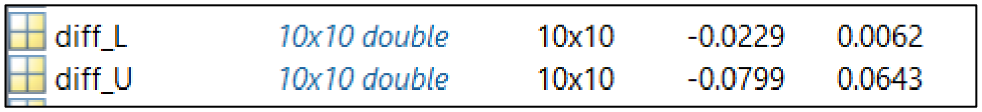
U Vivado

4	0	3	0	0	0	0	3	8	0
0	9	0	0	3	9	0	7	1	0
0	0.011719	7.5	0	5.003906	-2.98828	0	0.175781	-1.33203	0
0	0.003906	0	5	-2.33203	-2.99609	0	-5.44141	8.222656	0
0	0.027344	-0.00781	0	6.671875	-4.56641	2	-4.55859	-4.70313	6
0	-0.02734	0.023438	0.015625	0.023438	11.82813	-1.95313	8.671875	2.921875	-0.85938
0	0.003906	-0.00391	0.003906	0.003906	-0.02734	4.414063	1.8125	0.28125	-0.25781
0	0	0.023438	0	0.023438	0.042969	-0.01172	-2.33984	2.136719	2.484375
0	0.003906	0.003906	0.007813	-0.01563	0.019531	0.003906	-0.01172	-3.53906	1.84375
0	0.007813	0.019531	0	-0.02734	0.023438	0.015625	0.003906	0.007813	5.6875

U Difference

0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0.011719	0	0	0.003906	0.011719	0	0.009115	0.001302	0
0	0.003906	0	0	0.001302	0.003906	0	0.003038	0.000434	0
0	0.027344	-0.00781	0	0.005208	0.033594	0	0.019184	0.007986	0
0	-0.02734	0.023438	0.015625	0.023438	-0.07988	0.006875	-0.05879	-0.00613	0.020625
0	0.003906	-0.00391	0.003906	0.003906	-0.02734	0.006605	-0.02694	-0.00394	0.008227
0	0	0.023438	0	0.023438	0.042969	-0.01172	0.03629	-0.00144	0.002838
0	0.003906	0.003906	0.007813	-0.01563	0.019531	0.003906	-0.01172	0.06425	0.013404
0	0.007813	0.019531	0	-0.02734	0.023437	0.015625	0.003906	0.007813	0.044237

Error Analysis



```
diff_L    10x10 double    10x10    -0.0229    0.0062
diff_U    10x10 double    10x10    -0.0799    0.0643
```

REFERENCES

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

[2] A. Ching, H. Lane, M. Park, H. Lane, M. Park, H. Lane, M. Park, H. Lane, M. Park, H. Lane, and M. Park, “One Trillion Edges : Graph Processing at Facebook-Scale,” vol. 8, no. 12, pp. 1804–1815, 2015.

[3] D. Thomas, L. Howes, and W. Luk, “A comparison of CPUs, GPUs, FPGAs, and massively parallel processor arrays for random number generation,” in Proceeding of the ACM/SIGDA international symposium on field programmable gate arrays. ACM, 2009, pp. 63–72.

[4] T. Hamada, K. Benkrid, K. Nitadori, and M. Taiji, “A comparative study on ASIC, FPGAs, GPUs and general-purpose processors in the O(N2) gravitational N-body simulation,” NASA/ESA Conference on Adaptive Hardware and Systems, vol. 0, pp. 447–452, 2009.

[5] Tan, Guangming & Sun, Ninghui & R. Gao, Guang. (2007). A parallel dynamic programming algorithm on a multi-core architecture. Annual ACM Symposium on Parallelism in Algorithms and Architectures. 135-144. 10.1145/1248377.1248399.

[6] N. Bell and M. Garland, “Efficient Sparse Matrix-Vector Multiplication on CUDA,” NVIDIA Corporation, NVIDIA Technical Report NVR-2008-004, Dec. 2008.

[7] J. J. Dongarra, Jeremy Du Croz, Sven Hammarling, and I. S. Duff. 1990. A set of level 3 basic linear algebra subprograms. ACM Trans. Math. Softw.16, 1 (March 1990), 1-17. DOI: https://doi.org/10.1145/77626.79170

[8] Adrian M Caulfield, Eric S Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, et al. 2016. A cloud-scale acceleration architecture. In MICRO 2016. IEEE, 1–13.

[9] L. Zhuo and V. K. Prasanna, “Sparse Matrix-Vector Multiplication on FPGAs,” pp. 1–13.

[10] S. Jain-Mendon and R. Sass, "Performance evaluation of Sparse Matrix-Matrix Multiplication," *2013 23rd International Conference on Field programmable Logic and Applications*, Porto, 2013, pp. 1-4.

[11] S. M. Qasim, A. Ahmed, Telba, Y. Abdulhameed, and AlMazroo, "FPGA design and implementation of matrix multiplier architectures for image and signal processing applications," *International Journal of Computer Science and Network Security (IJCSNS)*, vol. 10, no. 2, pp: 169-176, Feb. 2010.

[12] M. Vucha, and A. Rajawat, "Design and FPGA Implementation of Systolic Array Architecture for Matrix Multiplication," *International Journal of Computer Aplications*, vol. 26, no. 3, pp: 18-22, Jul. 2011.

[13] Petya Vachranukunkiet. 2007. Power Flow Computation Using Field Programmable Gate Arrays. Ph.D. Dissertation. Drexel University, Philadelphia, PA, USA. Advisor(s) Prawat Nagvajara and Jeremy Johnson. AAI3261754.

[14] Siddhartha and N. Kapre, "Breaking Sequential Dependencies in FPGA-Based Sparse LU Factorization," *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, Boston, MA, 2014, pp. 60-63. doi: 10.1109/FCCM.2014.26.

[15] T. Nechma and M. Zwolinski, "Parallel Sparse Matrix Solution for Circuit Simulation on FPGAs," in *IEEE Transactions on Computers*, vol. 64, no. 4, pp. 1090-1103, April 2015. doi: 10.1109/TC.2014.2308202.

[16] Liu, Zhaohui & Mccanny, J.V.. (2003). Implementation of adaptive beamforming based on QR decomposition for CDMA. ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings. 2. II - 609. 10.1109/ICASSP.2003.1202440.

[17] L. Ma, K. Dickson, J. McAllister and J. McCanny, "Modified givens rotations and their application to matrix inversion," *2008 IEEE International Conference on Acoustics, Speech and Signal Processing*, Las Vegas, NV, 2008, pp. 1437-1440. doi: 10.1109/ICASSP.2008.4517890.

[18] Walke, Rajpal & W. M. Smith, Robert & Lightbody, G. (2000). 20 GFLOPS QR processor on a xilinx Virtex-E FPGA. Proc SPIE. 10.1117/12.406508.

[19] S. T. Alexander and A. L. Ghimikar, "A Method for Recursive Least Squares Filtering Based Upon an Inverse QR Decomposition," in *IEEE Transactions on Signal Processing*, vol. 41, no. 1, pp. 20-, January 1993. doi: 10.1109/TSP.1993.193124.

[20] K. X. Zhou and S. I. Roumeliotis, “A Sparsity-aware QR Decomposition Algorithm for Efficient Cooperative Localization,” 2012.

[21] A. Milinković, S. Milinković, and L. Lazić, “FPGA based dataflow accelerator for large matrix multiplication,” pp. 288–293.

[22] Joao Pinhao, “FPGA Multi-Processor for Sparse Matrix Applications,” pp. 1–9.

[23] A. Pınar and M. T. Heath, “Improving Performance of Sparse Matrix-Vector Multiplication.”

[24] K. Townsend and J. Zambreno, “Reduce , Reuse , Recycle (R3): a Design Methodology for Sparse Matrix Vector Multiplication on Reconfigurable Platforms,” pp. 185–191, 2013.

[25] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, “Pregel : A System for Large-Scale Graph Processing,” pp. 135–145, 2010.

[26] M. V Ryan and M. V Ryan, “FPGA Hardware Accelerators - Case Study on Design Methodologies and Trade-Offs FPGA Hardware Accelerators - Case Study on Design Methodologies and Trade-Offs by,” 2013.

[27] M. T. Shriyashi Jain, Jaikaran Singh, Neeraj Kumar, “FPGA Implementation of Latency , Computational time Improvements in Matrix Multiplication FPGA Implementation of Latency , Computational Time Improvements in Matrix Multiplication,” no. March, 2016.

[28] M. High and P. Computer, “Sparse Matrix Storage Formats 2.1,” no. 2012, pp. 20–36, 2008.

[29] P. Russek and K. Wiatr, “THE ALGORITHMS FOR FPGA IMPLEMENTATION OF SPARSE MATRICES MULTIPLICATION Ernest Jamro , Tomasz Pabi ´,” vol. 33, pp. 667–684, 2014.

[30] “7 Series FPGAs Data Sheet : Overview Summary of 7 Series FPGA Features Table 1 : 7 Series Families Comparison Spartan-7 FPGA Feature Summary,” vol. 180, pp. 1–18, 2017.

[31] S. Skalicky, C. Wood, Ł. Marcin, and M. Ryan, “High Level Synthesis : Where Are We ? A Case Study on Matrix Multiplication.”

[32] P. Grigoras, P. Burovskiy, E. Hung, and W. Luk, “Improving SpMV Performance on FPGAs through Lossless Nonzero Compression.”

[33] S. Aslan and J. Saniie, “Matrix Operations Design Tool for FPGA and VLSI Systems,” no. February, pp. 43–50, 2016.

[34] L. Zhuo and V. K. Prasanna, “Sparse Matrix-Vector Multiplication on FPGAs,” pp. 1–13.

[35] A. Azad and A. Buluc, “A work-efficient parallel sparse matrix-sparse vector multiplication algorithm.”

[36] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, “Optimization of sparse matrix – vector multiplication on emerging multicore platforms,” *Parallel Comput.*, vol. 35, no. 3, pp. 178–194, 2009.

[37] T. Mattson, I. Corporation, D. Bader, J. Berry, and S. National, “Standards for Graph Algorithm Primitives,” pp. 1–2.

[38] P. P. Letters, W. Scientific, P. Company, A. Lumsdaine, B. Hendrickson, J. Berry, S. National, L. Albuquerque, R. January, and B. Tourancheau, “Challenges in parallel graph processing,” 2007.

[39] S. Zhou, C. Chelmis, V. K. Prasanna, and A. E. G. Processing, “Graph Processing on FPGA.”

[40] X. Wang and S. G. Ziavras, “Parallel LU Factorization of Sparse Matrices on FPGA-Based Configurable Computing Engines *,” vol. 16, no. April, pp. 319–343, 2004.

[41] P. Greisen, M. Runo, P. Guillet, S. Heinzle, A. Smolic, H. Kaeslin, and M. Gross, “Evaluation and FPGA Implementation of Sparse Linear Solvers for Video Processing Applications,” no. 1, pp. 1–5.

[42] L. Polok and P. Smrz, “PIVOTING STRATEGY FOR FAST LU DECOMPOSITION OF SPARSE BLOCK MATRICES,” 2017.