

CONVERSION OF SPARSE MATRIX TO BAND MATRIX USING AN FPGA FOR
HIGH-PERFORMANCE COMPUTING

by

Anjani Chaudhary, M.S.

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
December 2020

Committee Members:

Semih Aslan, Chair

Shuying Sun

William A Stapleton

COPYRIGHT

by

Anjani Chaudhary

2020

FAIR USE AND AUTHOR'S PERMISSION STATEMENT

Fair Use

This work is protected by the Copyright Laws of the United States (Public Law 94-553, section 107). Consistent with fair use as defined in the Copyright Laws, brief quotations from this material are allowed with proper acknowledgement. Use of this material for financial gain without the author's express written permission is not allowed.

Duplication Permission

As the copyright holder of this work, I, Anjani Chaudhary, authorize duplication of this work, in whole or in part, for educational or scholarly purposes only.

ACKNOWLEDGEMENTS

Throughout my research and report writing process, I am grateful to receive support and suggestions from the following people. I would like to thank my supervisor and chair, Dr. Semih Aslan, Associate Professor, Ingram School of Engineering, for his time, support, encouragement, and patience, without which I would not have made it. My committee members Dr. Bill Stapleton, Associate Professor, Ingram School of Engineering, and Dr. Shuying Sun, Associate Professor, Department of Mathematics, for their essential guidance throughout the research and my graduate advisor, Dr. Vishu Viswanathan, for providing the opportunity and valuable suggestions which helped me to come one more step closer to my goal. I would also like to thank Xilinx for their support.

I also would like to thank my family and friends who encouraged and supported me throughout this journey.

TABLE OF CONTENTS

	Page
ACKNOWLEDGEMENTS	iv
LIST OF TABLES	vii
LIST OF FIGURES	viii
LIST OF ABBREVIATIONS.....	x
ABSTRACT.....	xii
 CHAPTER	
I. INTRODUCTION.....	1
Overview.....	1
II. LITERATURE REVIEW.....	5
Band Matrix	5
Application of Band Matrix	6
Band Matrix Storage	7
BLAS-General Band Storage Mode	8
General Band Storage Mode.....	9
Bandwidth Reduction Algorithm.....	9
Cuthill Mckee Algorithm.....	12
Reverse Cuthill Mckee Algorithm.....	12
Gibbs-Poole-Stockmeyer Algorithm	16
Graph Theory	17
Pseudo-peripheral Node.....	18
King's Algorithm	21
Gu Feng Algorithm.....	21
Modified Minimum Degree/Multiple Minimum Degree.....	24
Serial Communication	25
Field Programmable Gate Array.....	25
Digilent Zedboard	26
Functional Description	27

III. PROBLEM STATEMENT	30
IV. METHODOLOGY	32
Study Area Description.....	32
Expert Suggestion and Discussion.....	32
Design and Description of the Proposed System.....	32
Software Architecture	37
Hardware Architecture.....	38
V. TESTING AND ANALYSIS.....	41
System Block Diagram in Xilinx Vivado IDE	42
Simulation Result	43
Power Summary Result.....	44
Design Summary of Resource Utilization	45
Design Timing Summary.....	47
Latency and Throughput.....	49
VI. RESULTS	52
List of Test Matrices with RCM Algorithm Application.....	52
FPGA and MATLAB Results	53
VII. CONCLUSION	58
VIII. FUTURE SCOPE.....	59
APPENDIX SECTION	60
REFERENCES	80

LIST OF TABLES

Table	Page
1. Resources Utilization Design Summary	47
2. Latency and Throughput	51
3. Input and Output Matrices Details	52
4. Comparing FPGA and MATLAB Results	53

LIST OF FIGURES

Figure	Page
1. Non-zero matrix elements comparison [5]	2
2. Band Matrix	3
3. Special Band Matrix	6
4. 8 x 8 Band Matrix and its Storage	8
5. A Bandwidth of 6x6 Matrices	10
6. Matrix Before and After Reorder of Nodes [19]	11
7. Graph representation of a matrix [19]	11
8. Breadth-First Search [22]	14
9. Graph Representation and Adjacent Matrix of a Sparse Matrix	15
10. Graph Representation and Adjacent Matrix After RCM Order	15
11. Sparse Matrix Representation of Harwell-Boeing [23]	16
12. Band Matrix Representation of Harwell-Boeing [23]	16
13. Undirected graph	17
14. Directed graph	17
15. Undirected Graph to demonstrate Distance, Eccentricity, and Diameter	18
16. Level structure of node X_6	20
17. Level structure of node X_5	20
18. Level Structure of node X_2	21
19. Undirected graph [8]	22

20. ZedBoard- FPGA [30]	28
21. Zedboard Block Diagram.....	28
22. Zynq Z7020 CLG484 Bank Assignments on ZedBoard.....	29
23. Flowchart of Proposed Work	33
24. Flowchart of the RCM Algorithm in the System.....	34
25. Matrix and its Graph Representation	34
26. Matrix and Graph after the RCM Reorder	35
27. Vivado Design Suite High-Level Design Flow	36
28. Configuration for JTAG Mode	37
29. High-Level Synthesis Design Flow	38
Figure 30. Zynq-7000 AP SoC Architecture	39
31. Flowchart of Data Transmission.....	41
32. Block Diagram of the System.....	42
33. Input Data Waveform	44
34. Output Data Waveform.....	44
35. Power Summary.....	45
36. Resources Utilization Design Summary	46
37. Timing Design Summary	48
38. Input Waveform.....	49
39. Output Waveform	50
40. Bar Graph of Bandwidth for Sparse Matrix and Band Matrix.....	57

LIST OF ABBREVIATIONS

Abbreviation	Description
ACP	Accelerator Coherency Port
ADC	Analog-to-Digital Converter
BFS	Breadth-First Search
CM	Cuthill-Mckee Algorithm
DAC	Digital-to-Analog Convertor
DMA	Direct Memory Access
FIFO	First-in/First-out
FPGA	Field Programmable Gate Array
HDL	Hardware Description Language
HLS	High-Level Synthesis
IP	Intellectual Property
LUTs	Look Up Tables
MMD	Modified Minimum Degree
PC	Personal Computer
PL	Programmable Logic
PS	Processing System
RCM	Reverse Cuthill-Mckee
SDK	Software Development Kit

SoC	System on Chip
VHDL	Very High-Speed Integrated Circuit Hardware Description Language
WHS	Worst Hold Slack
WNS	Worst Negative Slack
TNS	Total Negative Slack
XPS	Xilinx Platform Studio

ABSTRACT

Low power and high computation speed with less memory storage are essential for a real-time scientific computational application. Applications such as image processing, power system, finite element system, circuit design, data from sensors utilize a large amount of data. An arithmetic operation on a general matrix can take more time and require more memory to store the data. Band matrices could be a key component in many scientific computing applications. A special sparse matrix, i.e., band matrix, has small bandwidth and minimizes storage, efficiently leading to less computation time. This paper presents a design and hardware implementation to convert a sparse matrix to a band matrix for a minimum matrix bandwidth using an existing Reverse Cuthill-McKee algorithm (RCM).

The Field Programmable Gate Array (FPGA) hardware design helps to solve larger data problems in terms of memory storage, speeding up many sparse matrix operations. Based on the FPGA hardware, the design and implementation, and synthesis are carried out by keeping in mind the architecture, area, and power requirements. In this research, the Vivado High-Level Synthesis (HLS) language is used. Intellectual Property (IP) generated from HLS will be linked to the ZYNQ processor, which can be implemented in a large system and have flexibility in FPGA based design. For the verification and reporting of this designed system, MATLAB is used.

I. INTRODUCTION

Overview

A matrix is generally stored as a two-dimensional array. It is possible to represent many systems with less non-zero elements in matrices form. If the non-zero elements are less than 10% of the matrix's total elements, it is called a sparse matrix. The sparsity of the matrix is calculated by dividing the total zero elements by a total number of matrix elements. 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, a memory would be wasted by storing redundant zeros. For a matrix, the amount of memory required to store its elements in the same format is proportional to matrix size. Because the sparse matrix has few non-zero elements, memory management can be reduced by storing only non-zero entries [1]. There is a various application of sparse matrices such as circuit simulation, power network, structural analysis, signal processing and statistics, computer vision, tomography, finite-element methods [2][3][4], etc. and a huge amount of data is generated by these applications [5]. A sparse matrix is useful for computing large scale matrices. Figure 1 is a graph comparing the non-zero elements of an ordinary matrix and a sparse matrix.

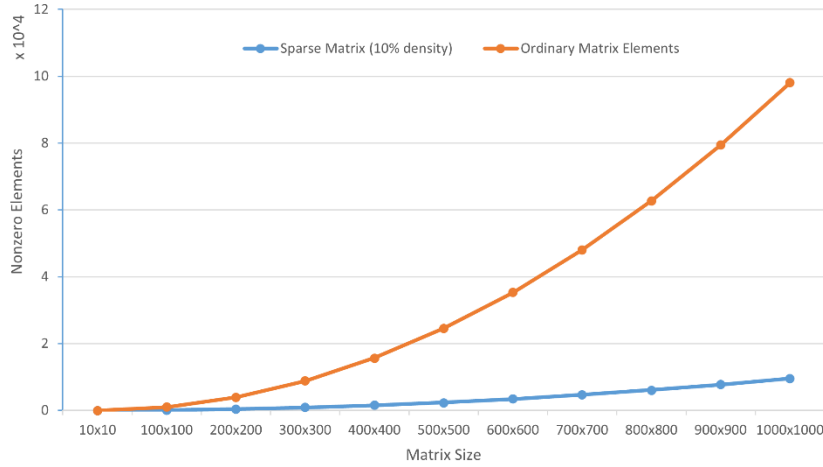


Figure 1. Non-zero matrix elements comparison [5]

The graph shows that the ordinary matrix contains more non-zero elements, making the matrix operation slow and needs more storage [5]. Sparse matrix computation can be accelerated by reducing matrix bandwidth [6]. Researchers believed and have been proved that rearranging sparse matrix vertices could reduce its bandwidth or profile. Band matrix is a special type of sparse matrix where nonzero elements are aligned to form a main diagonal and sub diagonal. In a band matrix operation, only diagonal, which contains non-zero elements, are stored. Because a band matrix has a smaller bandwidth than a sparse matrix and nonzero elements are clustered to diagonal, it is useful for scientific computing such as direct methods for solving sparse linear system and for iterative methods. In this paper, a hardware design is proposed which converts a sparse matrix to band matrix using the RCM algorithm to reduce the bandwidth of a matrix.

Large data is received from different applications such as image processing, power system, finite element system, data from sensors and is often sparse and contains more zero elements. As the matrix dimension increases, it becomes difficult for the software to compute the data. By reordering sparse matrix elements, a sparse matrix can

be converted into a band matrix to reduce its bandwidth. There are few different algorithms to reduce the bandwidth of a matrix, such as the Cuthill-McKee (CM), the Reverse Cuthill-McKee (RCM)[4], Sloan's algorithm, and the Gibbs- Poole algorithm [3][7][8]. In this research, we used the Reverse Cuthill-McKee algorithm to rearranges data of a sparse matrix because it is simple to execute, easy to parallelize relatively, shows a low computational cost, and proved to be best for the band elimination method[4], [6], [9], [10].

In this research, FPGA based hardware design is proposed for an RCM algorithm which will rearrange nonzero elements of sparse matrix diagonally to form a band matrix with smaller bandwidth or profile. A sparse matrix's bandwidth is the maximum distance between two elements in any row of a matrix [11]. A banded matrix is a sparse matrix where the non-zero element is aligned to the diagonal band (main diagonal) and zero or more diagonals on either side. Only nonzero elements of the diagonal are stored in the banded representation of the sparse matrix. The band matrix's main concept is to reduce the sparse matrix's bandwidth so the new matrix will have bandwidth smaller than the maximum possible bandwidth. Bandwidth is directly proportional to the amount of memory required to store a matrix, so proper reordering of nodes is essential to reduce the memory cost [11][4]. An example of a band matrix is shown in Figure 2.

$$\begin{array}{c}
 \begin{array}{c} \leftarrow \mathbf{K} \rightarrow \\ \updownarrow \end{array} \\
 \mathbf{A} = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & \cdots & a_{1,k} & 0 & \cdots & 0 \\ a_{2,1} & a_{2,2} & a_{2,3} & \cdots & a_{2,k} & a_{2,k+1} & \cdots & 0 \\ a_{3,1} & a_{3,2} & a_{3,3} & \cdots & a_{3,k} & a_{3,k+1} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ a_{p,1} & a_{p,2} & a_{p,3} & \cdots & a_{p,k} & \cdots & \cdots & 0 \\ 0 & a_{p+1,2} & a_{p+1,3} & \cdots & a_{p+1,k} & a_{p+1,k+1} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 0 & 0 & \cdots & a_{n,n} \end{bmatrix}
 \end{array}$$

Figure 2. Band Matrix

For an $n \times n$ matrix, A to be band matrix, all elements outside a diagonally bordered band having a range (column and row) as “k” and “p” should satisfy the given condition in Equation 1.

$$a_{i,j} = 0 \text{ if } j < i-p, \text{ or } j > i+k \quad 1$$

Where k and p are upper bandwidth and lower bandwidth, $a_{i,j}$ is matrix elements. A matrix having $k = p = 0$ and $k = p = 1$ is called diagonal and tridiagonal matrix. A matrix to be band should have a reasonably smaller bandwidth. The bandwidth of a matrix can be calculated using Equation 2.

$$\text{Bandwidth (BW)} = k+p+1 \quad 2$$

Matrix with smaller bandwidth helps minimize hardware design, smaller memory allocation storing only the non-zero elements, and decreasing the overall computational time. Reducing matrix bandwidth by reordering a sparse matrix can accelerate many sparse matrix computations[12]. For example, a direct method that allows simple data structure for solving a linear system is useful in an iterative method as nonzero elements are grouped close to diagonal enhancing data locality. The non-zero elements number of a matrix depends on the matrix size and calculated using Equation 3 below[13].

$$N_{NZ} = n + \sum_{i=1}^k n - i + \sum_{j=1}^p n - j \quad 3$$

II. LITERATURE REVIEW

Many applications generate a huge amount of data. When the number of data increases, calculation speed often decreases. It has been challenging to keep many features such as hardware size, memory storage, speed in an acceptable range, yet making data computation easy. Data generated from a different application are generally sparse. Sparse indicates data having less non-zero value and more zero values. The sparse matrix concept is usually used for larger applications as it is easy and reliable to store only non-zero elements. The sparse matrix comparatively has larger bandwidth, which means large memory is required for storage. The bandwidth and matrix profile has always been an area of great concern because of their influence in the finite element method, circuit design, hypertext layout, chemical kinetics, linear equation solution, etc.[1]. It has been found that a reduction in bandwidth and profile of a matrix can be reduced memory storage requirement and processing time of such linear systems. There are many tools proposed for the hardware design of FPGA. The High-Level Synthesis (HLS) is considered an easy and simple tool to generate hardware from C/C++ (C to Verilog/VHDL) [14]. This research uses the Vivado HLS to design a software model of the RCM algorithm and hardware implementation on FPGA.

Band Matrix

In matrix theory, a band matrix is a special type of sparse matrix whose nonzero elements lie above and below the main diagonal. The number of nonzero elements above the main diagonal gives upper bandwidth, and nonzero elements below the main diagonal give lower bandwidth of the matrix. The total bandwidth of a matrix is considered the sum of upper bandwidth, lower bandwidth, and main diagonal. The last diagonal (above

and below the main diagonal) containing non-zero elements determines the upper and lower bandwidth matrix. Matrix bandwidth plays an important role in memory requirements in a system. Let for a matrix A of elements $A_{i,j}$ with upper and lower bandwidth 'k' and 'p':

Let for a matrix A of elements $A_{i,j}$ with upper and lower bandwidth 'k' and 'p':

- The upper bandwidth 'k' is the smallest number where $A_{i,j} = 0$ whenever $j-i > k$
- The lower bandwidth 'p' is the smallest number where $A_{i,j} = 0$ whenever $i-j > p$

There are special band matrices: (1) Diagonal matrix and (2) Tridiagonal matrix.

The Diagonal matrix has all values zero except the diagonal elements. In a tridiagonal matrix, the main diagonal, the first upper diagonal, and the first below diagonal have nonzero elements, and the rest value is zero. The diagonal matrix has upper and lower bandwidth zero, and the tridiagonal matrix has upper and lower bandwidth of one. An example of a diagonal and tridiagonal matrix is shown in Figure 3 below.

$$K_a = \begin{bmatrix} \pi & \pi & & & & & & \\ \pi & \pi & \pi & & & & & \\ & \pi & \pi & \pi & & & & \\ & & \pi & \pi & \pi & & & \\ & & & \pi & \pi & & & \\ & & & & \pi & \pi & & \\ & & & & & \pi & \pi & \\ & & & & & & \pi & \pi \end{bmatrix}$$

(a) Tridiagonal Matrix

$$K_a = \begin{bmatrix} \pi & & & & & & & \\ & \pi & & & & & & \\ & & \pi & & & & & \\ & & & \pi & & & & \\ & & & & \pi & & & \\ & & & & & \pi & & \\ & & & & & & \pi & \\ & & & & & & & \pi \end{bmatrix}$$

(b) Diagonal Matrix

Figure 3. Special Band Matrix

Application of Band Matrix

In numerical analysis, for faster and efficient operation, a matrix with a smaller bandwidth/profile is preferred. Matrix operation becomes difficult and less efficient for larger matrix. A smaller bandwidth in a matrix is important to minimize the hardware

design and memory store by storing only the non-zero matrix elements and simplifying complex matrix operations such as LU factorization and QR factorization for a faster and accurate solution. It can be implemented on a large-scale system in a linear equation.

A banded matrix is a special type of sparse matrix, with non-zero elements aligned to a diagonal, forming the main diagonal with zero or more diagonals on either side. A matrix bandwidth has been the topic of research for a long time. The researcher is interested in finding ways to minimize the bandwidth of a matrix through a sparse matrix with efficient computation and less memory storage than a dense matrix. Therefore, here comes a concept of a band matrix with reduced bandwidth. Because a band matrix has small bandwidth, they are used in many real-life applications such image processing, linear system, circuit theory, finite element method for approximating solutions of a partial differential equation, chemical kinetics, numerical geophysics, hypertext layout, large scale power transmission system, etc. where collected data has many zero elements [5] [1] [15] [16] [17]. In a complex system such as a finite element system and problems in a higher dimension, matrices are often banded.

Band Matrix Storage

The real-life application produces a huge amount of data. There is always demanded to analyze and solve large and complex problems defined by a linear equation in the form $Ax = b$. The size of these data is very large, which needs much storage. In a band matrix, which is a special kind of sparse matrix, non-zero elements are aligned along the diagonal only (on the main diagonal and to sub diagonals) while all other elements are zero. The numbers of elements to be stored are only diagonal elements, therefore, minimizing memory storage. Many storage methods are proposed for storing

band matrix. The two commonly used storage techniques for a general band matrix are described below.

BLAS – General Band Storage Mode: The most popular one is a linear algebra package (LAPACK), also known as BLAS. This method is suitable for both a square matrix and a rectangular matrix. In this method, the non-zero elements of matrix A are stored in a rectangular, $(kl+ku+1)$ by n , matrix A^{band} , such that the diagonals of A become rows of A^{band} while the columns of A remain as columns of A^{band} . For example, Figure 4 shows that an $m \times n$ size matrix A having upper and lower bandwidth of q and p can be stored in a two-dimensional array as $p+q+1$ row and n column. For row storage, all row values are pushed to the left or right to align the column on an anti-diagonal forming a matrix A^{band} . For column storage, diagonals of the matrix are stored in a row array, and columns of a matrix are stored in the array's corresponding column. If $p, q < n$ then, A is stored in either a $(p+q+1)$ -by- n or a n -by- $(p+q+1)$ matrix A^{band} with $a_{i,j} = a^{band}_{i,j-i+p+1}$

$$A = \begin{matrix} & \xleftrightarrow{k} & & & & & & & \\ \begin{matrix} \uparrow p \\ \downarrow p \end{matrix} & \begin{bmatrix} a_{1,1} & a_{1,2} & 0 & 0 & 0 & 0 & 0 & 0 \\ a_{2,1} & a_{2,2} & a_{2,3} & 0 & 0 & 0 & 0 & 0 \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} & 0 & 0 & 0 & 0 \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} & a_{4,5} & 0 & 0 & 0 \\ 0 & a_{5,2} & a_{5,3} & a_{5,4} & a_{5,5} & a_{5,6} & 0 & 0 \\ 0 & 0 & a_{6,3} & a_{6,4} & a_{6,5} & a_{6,6} & a_{6,7} & 0 \\ 0 & 0 & 0 & a_{7,4} & a_{7,5} & a_{7,6} & a_{7,7} & a_{7,8} \\ 0 & 0 & 0 & 0 & a_{8,5} & a_{8,6} & a_{8,7} & a_{8,8} \end{bmatrix} \end{matrix} \quad A^{band} = \begin{bmatrix} 0 & 0 & 0 & a_{11} & a_{12} \\ 0 & 0 & a_{21} & a_{22} & a_{23} \\ 0 & a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \\ a_{52} & a_{53} & a_{54} & a_{55} & a_{56} \\ a_{63} & a_{64} & a_{65} & a_{66} & a_{67} \\ a_{74} & a_{75} & a_{76} & a_{77} & a_{78} \\ a_{85} & a_{86} & a_{87} & a_{88} & 0 \end{bmatrix}$$

Figure 4. 8 x 8 Band Matrix and its Storage

Let A is a matrix of size $m \times n$ and the number of nonzero nnz. The memory requirements in bytes are calculated as in Equation 4 [18].

$$\text{memory} = 8 * \text{nnz} + 4 * (\text{nnz} + n + 1) \quad 4$$

Below are the steps for BLAS general band matrix storage mode.

For an $m \times n$ matrix A with upper bandwidth “ml” and lower bandwidth “mu”:

Step 1: Select $j = 1:n$, where n is the size of a matrix

Step 2: $k = \mu + 1 - j$

Step 3: for $i = \max(1, j - \mu) : \min(m, j + \mu)$

Step 4: When step 3 is true, $A^{\text{new}}(k+i, j) = A(i, j)$, where A^{new} is storage form with dimension $(\mu + m + 1, n)$

Step 5: Continue until step 3 condition is true

Step 4: Repeat process until the value of “ j ” is equal to n

General Band Storage Mode: This method is applied only for a square matrix.

In this process, the band matrix elements are packed to a two-dimensional array column-wise so that each diagonal of a matrix appeared like a row in the packed array.

For a matrix A of size $m \times n$ and bandwidth μ and ν , the process is explained below. Let A^{new} be the new storage matrix. Its dimension should be greater than or equal to $(2\mu + \nu + 1, n)$.

Step 1: $md = \mu + \nu + 1$

Step 2: Select $j = 1:n$, where n is the size of a matrix

Step 3: for $i = \max(j - \mu, 1) : \min(j + \nu, n)$

Step 4: When step 3 is true, $A^{\text{new}}(i - j + md, j) = A(i, j)$, where A^{new} is storage form

Step 5: Continue until step 3 condition is true

Step 6: Repeat process until the value of “ j ” is equal to n

Bandwidth Reduction Algorithm

It is considered that the smaller the bandwidth of a matrix, the lesser is the computer storage requirement and lesser solution time. Therefore, for a larger data producing application, the matrix's profile and bandwidth play an important role. For a

given matrix A, its bandwidth is given by Equation 5.

$$A(i, j) = 0 \text{ for } |i - j| > K \quad 5$$

Where K is a non-negative integer

For example, the bandwidth of the given matrices in Figure 5(a), (b), and (c) are 3, 5, and 1, respectively.

$$\begin{bmatrix} 1 & 1 & 0 & \cdots & \cdots & 0 \\ 1 & 1 & 1 & \ddots & \ddots & \vdots \\ 0 & 1 & 1 & 1 & \ddots & \vdots \\ \vdots & \ddots & 1 & 1 & 1 & 0 \\ \vdots & \ddots & \ddots & 1 & 1 & 1 \\ 0 & \cdots & \cdots & 0 & 1 & 1 \end{bmatrix}$$

(a)

$$\begin{bmatrix} 1 & 1 & 1 & 0 & \cdots & 0 \\ 1 & 1 & 1 & 1 & \ddots & \vdots \\ 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 \\ \vdots & \ddots & 1 & 1 & 1 & 1 \\ 0 & \cdots & 0 & 1 & 1 & 1 \end{bmatrix}$$

(b)

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 8 & 0 & 0 \\ 0 & 0 & 0 & 0 & 6 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 \end{bmatrix}$$

(c)

Figure 5. A Bandwidth of 6x6 Matrices

Rows and columns reorder of a sparse matrix plays an important role in achieving certain properties that better perform the system performance. Among different properties, bandwidth is one of them [11]. There is a different algorithm proposed to reduce the bandwidth of a matrix by reordering its nodes. Figure 6 shows a matrix K_a before reordering and K_b matrix after rearranging its rows and columns. Figure 6 is a graph representation of both matrices

$$\begin{aligned}
(a) \quad K_a &= \begin{bmatrix} \pi & \pi & & \pi & \pi & & \\ \pi & \pi & \pi & & \pi & \pi & \\ & \pi & \pi & \pi & & \pi & \\ & & \pi & \pi & & & \pi \\ \pi & & & \pi & \pi & & \\ \pi & \pi & & \pi & \pi & \pi & \\ & \pi & & \pi & \pi & \pi & \\ & & \pi & & \pi & \pi & \end{bmatrix} \\
(b) \quad K_b &= \begin{bmatrix} \pi & \pi & \pi & \pi & & & \\ \pi & \pi & & \pi & & & \\ \pi & & \pi & \pi & \pi & \pi & \\ \pi & \pi & \pi & \pi & & \pi & \\ & \pi & & \pi & \pi & \pi & \pi \\ & & \pi & \pi & \pi & \pi & \pi \\ & & & \pi & & \pi & \pi \\ & & & \pi & \pi & \pi & \pi \end{bmatrix}
\end{aligned}$$

Figure 6. Matrix Before and After Reorder of Nodes [19]

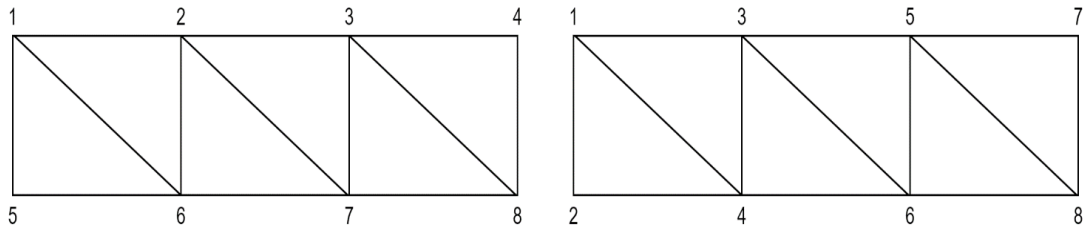


Figure 7. Graph representation of a matrix [19]

The upper and lower bandwidth of K_a is five each, and the lower and upper bandwidth of K_b is three each. Figure 7 is a graphical representation of the matrix and how the reordering of their nodes helps reduce the matrix's profile.

Different theories for reordering a sparse matrix have been proposed. Some of them are mentioned below.

- I. Cuthill-Mckee Algorithm
- II. Reverse Cuthill-Mckee Algorithm
- III. Gibbs-Poole-Stockmeyer Algorithm
- IV. Gu Feng Algorithm
- V. King's Algorithm
- VI. Modified Minimum Degree

Above mentioned algorithms represent a sparse matrix as a graph. A graph is a model made up of vertex or nodes to represent pairwise relationships between objects. Graphs are connected by edges. A graph can be directed or undirected. A sparse matrix is generally represented as an undirected graph by connecting each row and columns of the matrix. Further detail for the RCM algorithm is described below.

Cuthill-McKee Algorithm: This algorithm is named after Elizabeth Cuthill and James McKee [4], [20]. The purpose of this method is to permute a symmetric sparse matrix to a band matrix with smaller bandwidth. CM uses the concept of Breadth-First Search with a start peripheral node until all nodes are visited. The algorithm's pseudocode is given below.

Step 1: Choose vertex with the lowest degree (x) and assigned to R . $R \leftarrow \{x\}$ and let Q is an empty queue.

Then following steps is done till $|R| < n$ for $i=1,2,3 \dots, n$

Step 2: Add adjacent nodes of element ' x ' (which are not in R) to Q $\{\}$ in increasing order of their degree. Check if that number is in ' R ' or not? If not, add it to ' R '.

Step 3: Repeat this process till $Q \{\}$ is empty, and all nodes have been explored (i.e., the size of R should be equal to the number of nodes).

Step 4: Rearrange nodes according to the new index in array R .

Reverse Cuthill-McKee Algorithm: The origin of the RCM algorithm is from Cuthill-McKee (CM) algorithm named after Elizabeth Cuthill and James McKee. This is also an algorithm to permute a symmetric sparse matrix to a band matrix with small bandwidth. The Reverse Cuthill-McKee (RCM) algorithm is given by Alan George and is commonly used for bandwidth reduction [21] [12]. It is the same algorithm as RC, but

after getting results, the index number is reversed at the end. When Gaussian elimination is applied, practically, the RCM is considered superior to the CM as it reduces fill-in [11] when Gaussian elimination is applied. Fill-in process in a matrix are entries that change non-zero from initial zero during algorithm execution. To reduce arithmetic operation and memory storage for an algorithm, it is necessary to reduce fill-in by changing rows and columns of a matrix. The RCM algorithm is used in this design to reorder nonzero elements of a sparse matrix to diagonal. This algorithm works on an unlabeled graph of a matrix.

Smaller bandwidth can be achieved by converting a sparse matrix into a band matrix by rearranging elements of the sparse matrix. Among several heuristics for profile and bandwidth reduction, the RCM algorithm method is considered the most promising technique for profile reductions [10]. By an appropriate renumbering of the nodes, it is often possible to produce a matrix with much smaller bandwidth. The RCM algorithm's goal is to reduce the bandwidth of a symmetric sparse matrix, which is used in linear solvers. An important decision in the RCM is choosing a start node. Various concepts are proposed to find a start node in the RCM, such as peripheral node, pseudo-peripheral node, small degree node. Furthermore, the RCM is generally used for matrices having a higher diameter [12].

In the RCM algorithm, the node with the smallest degree in the matrix (graph) is chosen and applies Breadth-First Search (BFS) algorithm on the graph, ordering the nodes in ascending order according to their degree in each level. If any nodes are not covered in the first BFS, the same process is done again, taking the uncovered smallest degree node as the starting point. After all the nodes are covered, the result array of the

list is reversed, and the matrix is reordered according to the list to get a new band matrix.

BFS is a fundamental search algorithm to explore the nodes and edges of a graph. The

BFS algorithm is generally useful in finding the shortest path on an unweighted graph.

The BFS starts with some arbitrary node at first and explores its neighbor before moving to the next level neighbor. In the BFS algorithm, the order in which nodes are explored is shown below in Figure 8.

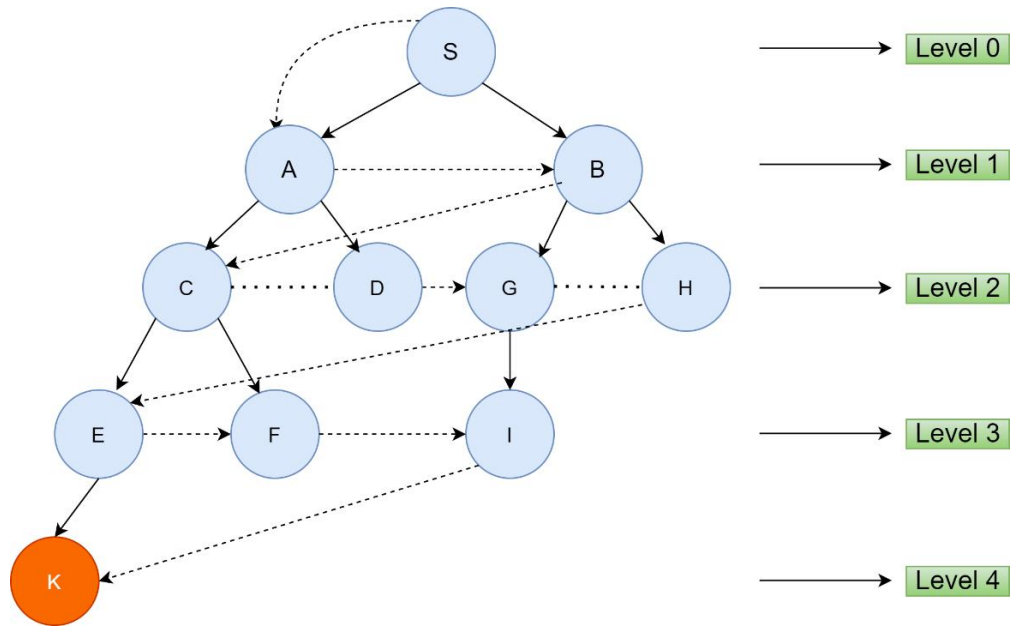


Figure 8. Breadth-First Search [22]

The algorithm for Reverse Cuthill McKee for an $m \times n$ matrix is given below.

Step 1: At first, the vertex with the lowest degree is chosen (x) and is assigned to R . $R \leftarrow \{x\}$ and let Q is an empty queue.

Then following steps is done till $|R| < n$ for $i=1,2,3,\dots, n$

Step 2: Add adjacent nodes of element ' x ' (which are not in R) to Q in increasing order of their degree. Check if that number is in ' R ' or not? If not, add it to ' R '.

Step 3: Repeat this process till Q is empty, and all nodes have been explored (i.e., the size of R should be equal to the number of nodes).

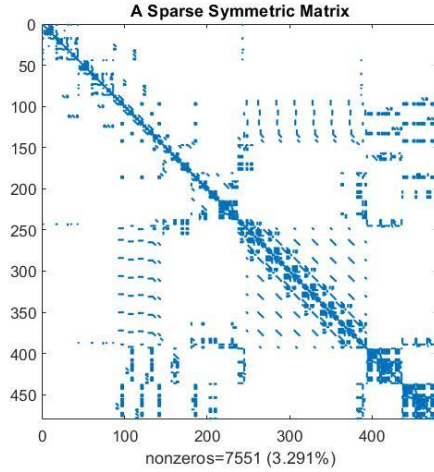


Figure 11. Sparse Matrix Representation of Harwell-Boeing [23]

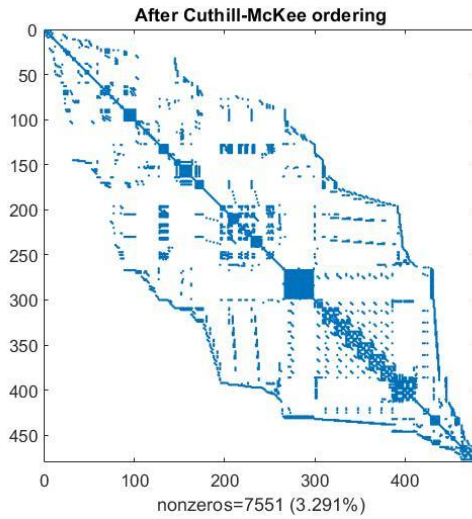


Figure 12. Band Matrix Representation of Harwell-Boeing [23]

We can see, after applying the RCM algorithm, bandwidth has decreased almost by double reducing memory storage, small hardware design, and less computation time.

Gibbs-Poole-Stockmeyer Algorithm: This is an important algorithm proposed by Gibbs-Poole-Stockmeyer [3] in reducing the bandwidth of a symmetric sparse matrix. This algorithm works with the concept of pseudo-peripheral node [3], [7], [8] to reduce both bandwidth and profile of a matrix. A matrix can be represented by a graph using graph theory. The basic concept of graph theory, peripheral node, and pseudo-peripheral

are explained below.

Graph Theory: The study of a graph is called Graph theory, a mathematical structure which also shows a relation between objects. Directed and undirected graphs are two types of graphs. A graph has nodes/points and is connected by edges. A directed and undirected graph example is shown in Figure 13 and Figure 14.

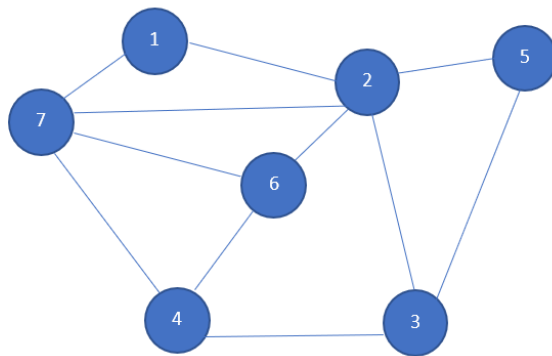


Figure 13. Undirected graph

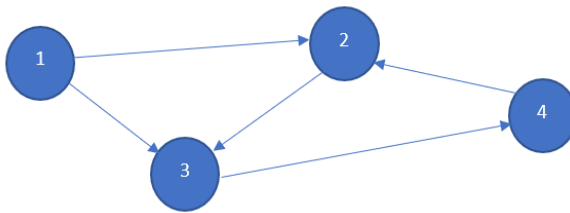


Figure 14. Directed graph

A graph has different properties, namely: distance, eccentricity, and diameter of the graph.

Distance $d(x,y)$: it is the length of the shortest path between two nodes, x and y.

Eccentricity $e(x)$: maximum distance from a specific node to reach all other nodes.

Diameter $\delta(G)$: maximum eccentricity of a graph.

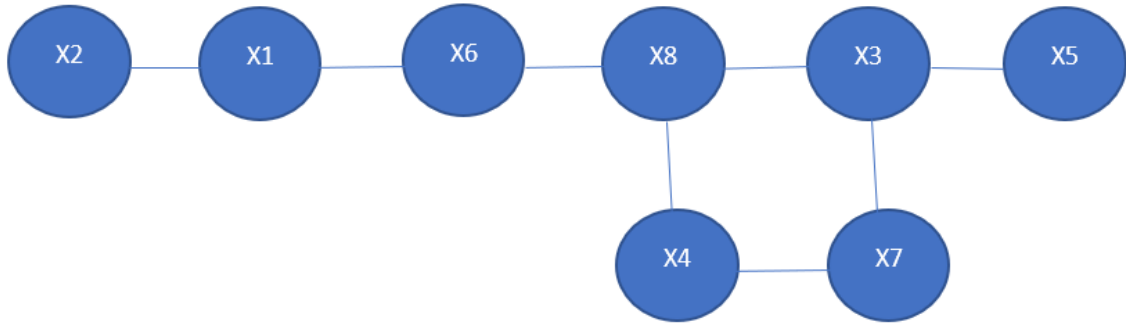


Figure 15. Undirected Graph to demonstrate Distance, Eccentricity, and Diameter

A given graph in Figure 15, diameter, eccentricity, and distance is calculated, as shown below.

$d(X_6, X_3) = 2$, smallest possible distance between X_6 and X_3

$l(X_6) = 3$, at maximum three distance, every node can be reached from node X_6

$\partial(G) = 5$, since X_2 , X_5 , and X_7 have maximum eccentricity.

Pseudo-peripheral Node: Gibbs-Poole-Stockmeyer came with a new concept of finding a start node which generally has high eccentricity called a pseudo-peripheral node. Their eccentricity can be as close as the diameter of a graph. This pseudo-peripheral node is considered a good start point for applying the RCM algorithm.

Gibbs-Poole-Stockmeyer algorithm in finding pseudo-peripheral node has mainly three steps. They are [3], [8].

Step 1: Finding two pseudo-peripheral nodes of a graph

Step 2: Minimizing level width

Step 3: Renumbering nodes level by level

The pseudocode of an algorithm given by Gibbs-Poole-Stockmeyer to find a pseudo-peripheral node of a graph is explained below[3].

Step 1: Choose a random node of minimal degree, say v .

- Step 2: Construct a level structure $L(v)$ of the given graph with “v” as a start root.
- Step 3: Note the eccentricity of “v,” $l(v)$, which is also equal to the number of levels.
- Step 4: Let M be a new set of last level nodes.
- Step 5: Arrange last level nodes in an increasing degree order and select a node, let “w” from M with a minimum degree.
- Step 6: Find the eccentricity of the selected node.
- Step 7: If $L(w) > L(v)$, new start node will be “w”
- Step 8: Repeat step (ii) till the new node's depth or level is greater than the selected one.
- Step 9: If for all $w \in M$, the level of $L(w)$ is not greater than $L(v)$, v will be a pseudo-peripheral node.

This idea given by Gibbs-Poole-Stockmeyer for bandwidth reduction highly depends on a selection of the start node. Finding a pseudo-peripheral node is further explained by the below example. Figure 16 as a reference image, the below steps are carried out to determine the pseudo-peripheral node.

- Step 1: Choose random node (let $v = X_6$).
- Step 2: Construct a level structure of a given graph, taking X_6 as a start node, as shown in Figure 16.

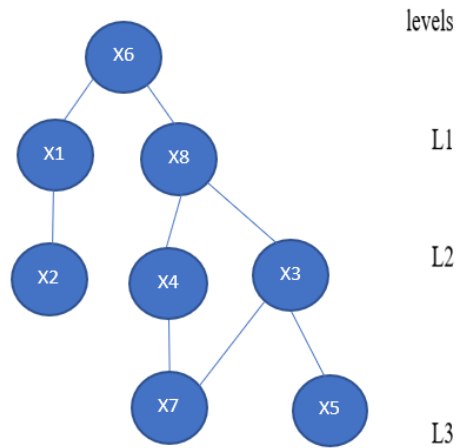


Figure 16. Level structure of node X_6

From the figure, eccentricity $l(X_6) = 3$, which is also considered the number of levels. In level 3, the degree of X_5 is 1 (number of connections), and the degree of X_7 is two.

Step 3: Choosing the next node with a minimum degree from the last level, let $u = X_5$.

Step 4: Construct level structure for the node u .

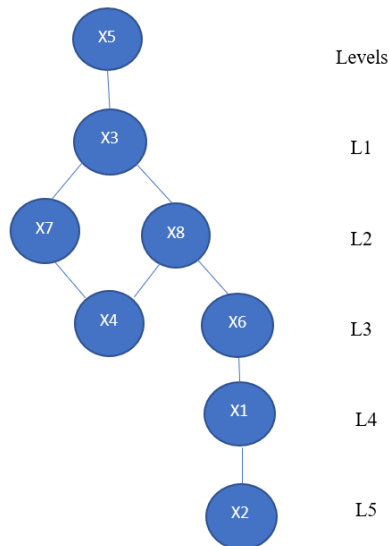


Figure 17. Level structure of node X_5

Eccentricity $l(X_5) = 5$, Since $l(X_5) > l(X_6)$, $v = X_5$.

Step 5: Since X_2 is on the last level, let $u = X_2$.

Step 6: Construct level structure for the node u .

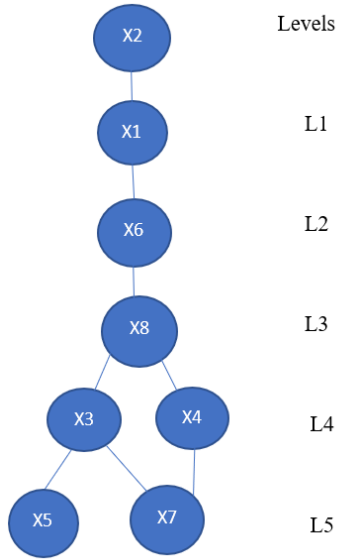


Figure 18. Level Structure of node X_2

Step 7: Eccentricity $l(X_2) = 5$, Since $l(X_5)$ is not greater than $l(X_2)$, $v = X_2$.

Step 8: X_2 and X_5 are two pseudo-peripheral nodes.

King's Algorithm: King's algorithm is a modification of the RCM algorithm where nodes/vertices are ordered based on the number of edges they must already visit vertices. The concept of this algorithm is that "if a vertex has many edges that connect it to vertices that have already been visited (ordered), then it may be a part of a local cluster and should be ordered closer to the other nodes in its cluster"[11].

Gu Feng Algorithm: This is an improvement of the Gibbs-Poole-Stockmeyer algorithm [3] given by Gu Feng to find a peripheral node [8]. A graph can have many pseudo-peripheral nodes whose eccentricity may be close enough to the graph's diameter $\partial(G)$, and it may not be a peripheral node. If the eccentricity of a node is equal to the

diameter of a graph, then it is called a peripheral node [8]. In figure 10, there are three possible peripheral nodes, X_2 , X_5 , and X_7 . Since selecting a start node is crucial in reducing matrix bandwidth, according to Gu Feng, a peripheral node is considered a good starting node for the RCM algorithm because its eccentricity is equal to the diameter of a graph. However, since it is very expensive to compute a peripheral node, it is not used often. Here is an example in Figure 19 to show that a pseudo-peripheral node may not be a peripheral node, and its eccentric distance could be less than a graph diameter [8].

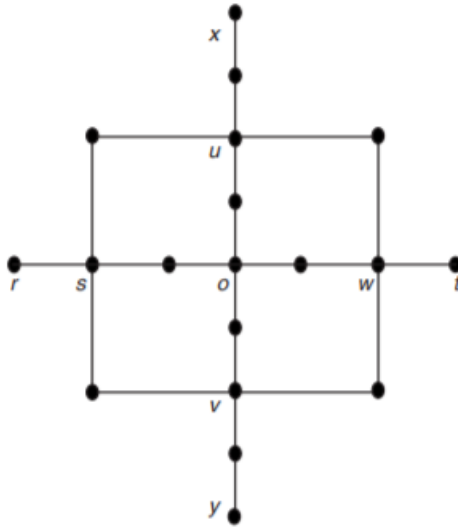


Figure 19. Undirected graph [8]

According to the Gibbs-Poole-Stockmeyer pseudo-peripheral algorithm, in Figure 19, if r is chosen as a minimum degree node as a start node, its eccentric distance will be $l(r)=6$. Constructing level structure at node ' r ,' ' t ' would be the last level node. Generating level structure at ' t ', $l(t)=6$, and ' r ' is the last level node. Since $l(r) = l(t) = 6$, therefore, we can say r and t are two pseudo-peripheral nodes of the graph. However, their eccentric distance is not equal to the diameter of a graph, which is $\partial(G) = 8$. From above figure, from definition of peripheral node, node x and y could be possible peripheral nodes as $l(x) = l(y) = \partial(G) = 8$. Thus, Gu Feng suggested that a pseudo-

peripheral node is not effective in considering it as a start node.

Gu Feng algorithm pseudocode for finding peripheral node is given below:

Step 1: Choose an $r \in A$, the closer to the center of symmetry (or pseudo center of symmetry), and with the largest possible degree, as the initial node.

Step 2: Generate G 's level structure rooted at r : Let $i = l(r)$; Let $e_{l(r)+1} = 0$.

Step 3: Ordering nodes in $L_i(r)$ decreasingly following their degrees. Calculating nodes' eccentric distance one by one. If $\exists x \in L_i(r)$ that $e(x) = 2i$ then x is a peripheral node, procedure stops; else go to Step 4.

Step 4: Seeking If $e_i \geq 2(i - 1)$, then the node whose eccentric distance is e_i is a peripheral node, the procedure stops else $i - 1 \rightarrow i$ and go to Step 3.

For Figure 19, the peripheral node can be determined in the following ways,

O, S, U, V, and W are five nodes with the highest degree, i.e., 4.

Step 1: If O is a start point, two nodes, x , and y will be at the last level with an eccentric distance of O as $l_4(O)$. The degree of x and y is 1. Eccentric distance of x and y , $e(x) = e(y) = 2i = 2*4 = 8$. x and y could be a peripheral node.

Step 2: Taking start node at S, t will be in the last level with eccentricity $e(t) = 6 \neq 2*6 = 12$. $e(t) = 6 < 2*(6-1) = 8$. Now will check $l_5(S)$, x, y, w is at the last level. $e(x) = e(y) = 8$, $e(w) = 5$. The result will be the same if taking 'W' as a start node. x and y could be a peripheral node.

Step 3: If U is a start node, y is only in the last level $L_6(U)$. $e(y) = 8 < 2*(6-1) = 10$.

Checking x in $L_5(U)$, $e(x) = 8 = 2*(5-1) = 8$. x is a peripheral node.

The above example shows that a peripheral node is a pseudo-peripheral node, but the reverse is not always true. Though the peripheral node algorithm is considered a good

start point to apply the Reverse Cuthill Mckee Algorithm (RCM), it is expensive to compute. Therefore, this algorithm is not used often.

Modified Minimum Degree/ Multiple Minimum Degree: This original algorithm is proposed by Harry Max Markowitz [24] in 1957 for a non-symmetric linear system. Tinney and Walker [25] proposed a symmetric version of Markowitz to solve large sparse systems in analyzing a power system. A theoretical graph model of Tinney and Walker's method was developed by Ross [26] and was named the Minimum Degree Algorithm. This algorithm is used to permute rows and columns of a symmetric sparse matrix before applying Cholesky Decomposition to reduce the non-zero number in the Cholesky factor [11]. The algorithm reduces nonzero elements of a sparse matrix by ordering the edges based on their degree. For a linear system of an algebraic equation, $Ax = b$, where A is a symmetric sparse square of order $n \times n$, matrix A usually suffers some fill-in [27], i.e., it will have more non-zero than the upper triangular when factorized by the Cholesky factor L . Therefore, a permutation matrix "P" is used so that the matrix (P^TAP) is also symmetric and has less possible fill-in. A new reorder equation $(P^TAP)(P^Tx) = P^Tb$ is solved instead.

For a given undirected graph $G = (V, E)$, where V =vertices and E is an edge of a graph, the pseudocode for Modified Minimum Degree (MMD) is as follows:

Step 1: For $i = 1$ to V

Step 2: Choose: the vertex v in graph G that has a minimum degree

Step 3: Remove vertex v from G

Serial Communication

In serial communication, data is sent between a PC and an external device one bit at a time. There are two types of serial communication, such as synchronous and asynchronous serial communication. For synchronous serial communication, a transmitter and a receiver are synchronized by a clock signal and running at the same rate so that the receiver and transmitter can sample data at the same time interval. On the other hand, asynchronous serial communication is a process where a transmitter and receiver are continuously synchronized by a common clock signal.

Because asynchronous serial communication has the advantage of long-distance communication and high reliability, it is generally used in data exchange between computers and peripherals [28]. A Universal Asynchronous Receiver- Transmitter (UART) is generally used for asynchronous serial communication. UART is an integrated circuit hardware device for serial communication used in computer or peripheral serial port. It mostly includes a clock generator, input, and output shift registers, read/write logic control, transmit/receive control, system data bus buffer, first-in/first-out (FIFO) buffer memory. Commonly, serial communication uses 8 data bits, no parity, and a one-stop bit. The speed/time of data transmission in a system is represented by the baud rate. A higher baud rate indicates that higher bits are transferred per second. The standard baud rate in bits per second used in serial communication are: 110, 300, 600, 1200, 2400, 4800, 9600, 14400, 19200, 38400, 57600, 115200, 128000 and 256000 [29].

Field Programmable Gate Array

To perform a complex task, there are many hardware architectures. Field Programmable Gate Array (FPGA) is one of them. It is an integrated circuit that can be

designed and programmed by the user as needed. FPGA is considered to deliver high performance, maintaining high flexibility. FPGA has a lot of simple and complex components. It contains an array of programmable logic blocks that can be configured to perform simple to complex combinational functions. FPGA includes simple components such as four-input LUTs (Look Up Tables) to implement logic functions like AND, NAND, NOT, or any user-defined function, memory elements like simple flip-flops or complex memory block. Usually, FPGA comes with software that is provided by the vendor to design and program it. Two hardware description languages (HDL) languages, such as Verilog and VHDL (Very High-Speed Integrated Circuit Hardware Description Language), are used to program FPGA.

Some FPGA has digital as well as analog features. Commonly available analog features are slew rate, quartz crystal oscillator, on-chip resistance-capacitance oscillator, phase-locked loop with embedded voltage control oscillator, etc. Mixed-signal FPGA generally has integrated peripherals such as analog-digital converter (ADC), digital to analog converter (DAC), an analog signal conditioning block helping them to behave as a system on chip (SoC). Because an FPGA is significantly faster for some applications, it can be used to solve any computable problem. Alongside using FPGA as a hardware accelerator, it can also be used to implement soft microprocessors such as Xilinx MicroBlaze or Altera Nios.

Digilent Zedboard: The proposed algorithm for the sparse matrix to band matrix conversion is implemented in the Xilinx Zynq series board, i.e., ZedBoard. It is a low-cost development board of Zynq-7000 All Programmable SoC XC7Z020-CLG484-1 (AP SoC) consisting of two ARM Cortex A9 cores and Xilinx programmable logic in a single

device. The features of this board are mentioned below. An image for Zedboard FPGA is shown in Figure 20.

Functional Description: A combination of a dual Corex-A9 with a speed of 667 MHz Processing System (PS) with 85,000 Series-7 Programmable Logic (PL) cells, therefore, makes it a powerful device for many applications. ZedBoard Zynq -7000 consists of a memory interface unit that includes a dynamic memory controller and static memory module. The ZedBoard has two Micron DDR3, 128 Megabit, each with 16 memory components creating a 32-bit interface with a total 512 MB of memory. This board consists of a 4-bit SPI (quad-SPI) serial NOR flash. The Multi-I/O SPI Flash memory helps to provide non-volatile code and data storage. It is also used to initialize the PS subsystem as well as configure the PL subsystem (bitstream). The XC7Z020 SoC contains Atrix-7 PL, which has 85k logic cells, 53200 LUT, 106400 flip-flops, 560 kB of BRAM organized to 140 units, each consisting of 2048 by 18-bit storage, and 220 DSP slices with two 12 V on-chip analog-to-digital converter (XADC). This board is compatible with Xilinx's high-performance Vivado Design Suite as well as the ISE Toolkit. ZedBoard offers more capacity, more resources, and higher performance than earlier designs.



Figure 20. ZedBoard- FPGA [30]

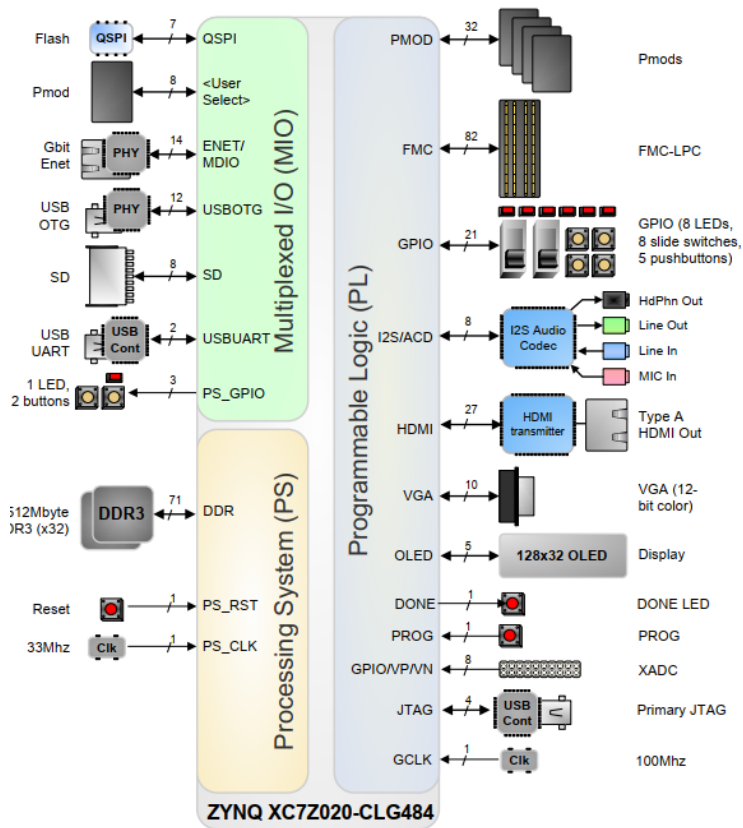


Figure 21. Zedboard Block Diagram

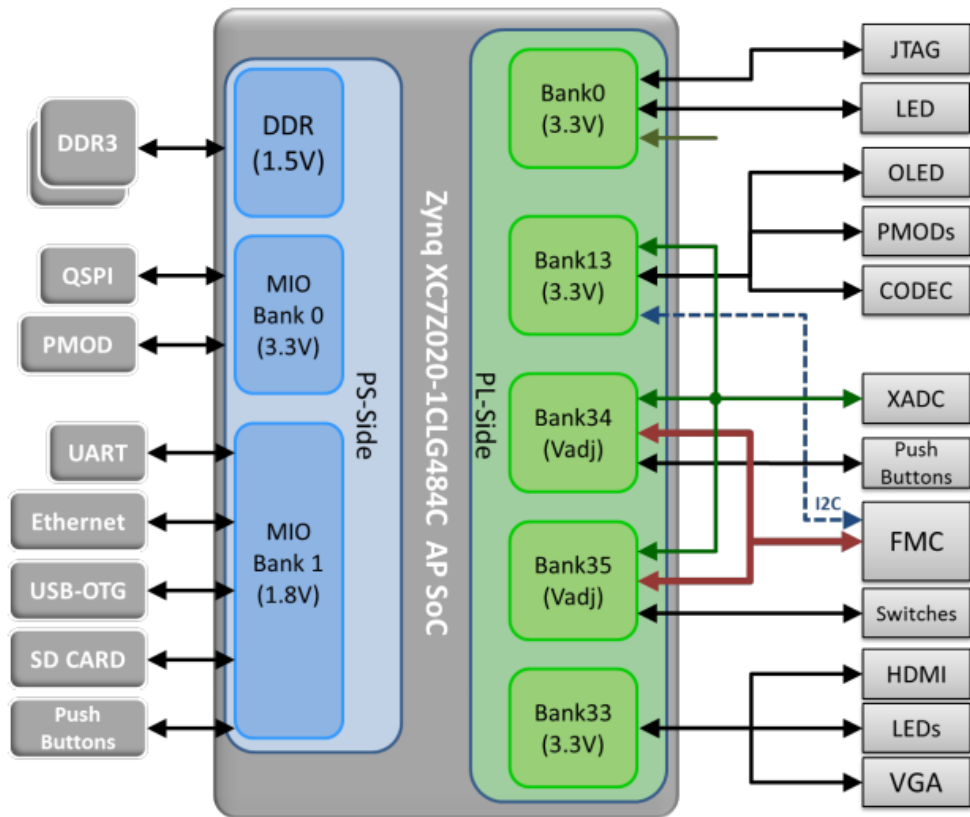


Figure 22. Zynq Z7020 CLG484 Bank Assignments on ZedBoard

III. PROBLEM STATEMENT

In the past years, multicore systems play an important role in high-performance computing. However, due to shortage of optimization techniques, some linear system is not able to utilize it in full. Lower power and high computation speed with less memory storage have always been essential criteria for a real-time scientific computational application. Data from the different applications are sparse and are generally larger in number. A sparse matrix has lesser non-zero elements, which is helpful for fast computation. Band matrix, on the other hand, is a special sparse matrix with reasonably small upper and lower bandwidth making them suitable for simpler algorithms than a sparse matrix. The difference in storage techniques of both matrices makes a band matrix preferable over a sparse matrix. Moreover, a banded matrix is compressed, which saves memory usage [31]. It is considered to have a great advantage for very large matrices.

The software has certain advantages. However, some work process is dependent on memory, CPU, and storage allocation. It might slow down if when an application does not get enough memory or available allocated space is used. Hardware has different modules designed to do dedicated tasks increasing the efficiency and overall performance of the system. Software for the Reverse Cuthill Mckee algorithm (RCM) is available in MATLAB. However, there is a limitation of matrix size in MATLAB, which is defined by the amount of available RAM in a system. Nowadays, various application fields such as image processing, telecommunication, signal processing, circuit analysis, etc. consider FPGA technology demanding and advantageous considering the fine degree of customization available which is used to direct and orchestrate data movement on and off-chip resulting in a good performance and for high data throughput[32]. There is

always a challenge to develop fine-tuned FPGA implementation, which could take months to develop to incorporate all the state of art techniques to exceed its performance. The solid step for improving the performance no longer involves proposing new expensive optimization but applying the optimizations whenever they are useful. To avoid the extra computation and storage imposed by the large majority of zero elements in a band matrix, it is standard to store only the nonzero elements in memory. Since the hardware does not need to change for varying matrices, the time required for initialization can be minimized by which the system integration is simplified. Not much work is done related to the RCM algorithm at the hardware level. This research presents an idea of the implementation of the RCM algorithm for sparse to band matrix conversion using Field Programmable Gate Array (FPGA).

IV. METHODOLOGY

Study Area Description

The design and the methodology for this research are discussed in this section. Vivado HLS takes an input matrix, synthesis, and creates RTL. Packaged IP from HLS is used in Vivado IDE for the hardware implementation. A Zynq software project is created in Xilinx SDK (software development kit). The extracted hardware design and bitstream from Vivado IDE are implemented in Xilinx Zynq – 7000 FPGA using SDK (Software Development Kit). Random square sparse matrices are used as input data. Most of the input data are undirected symmetric matrices. In this research number of different sparse matrices were tested. Each matrix was of varying density and size. To test the system's accuracy, the same input data were given to MATLAB, and the result was compared. Details about the experimental work and results have been discussed in the next section given below.

Expert Suggestion and Discussion

Guidelines and suggestions from the thesis advisor and mentor, Dr. Semih Aslan, and committee members Dr. William A Stapleton and Dr. Shuying Sun, helped carry out this research. Related research articles from the literature review were a great source to start the work.

Design and Description of the Proposed System

The overall procedure for this research is shown in the flowchart in Figure 23. The flowchart of the system design for the RCM algorithm is shown in Figure 24. This research contains two parts: software design and hardware implementation. For the software design, Xilinx Vivado High-Level Synthesis (HLS) 2016.4 software is used.

C++ programming language is used to write the main program and testbench. A sparse matrix is input data in HLS. After successful synthesis and implementation, the output from Vivado HLS can be used as an IP block for an IP catalog in either Vivado Design Suite, System Generator for DSP (Vivado, and ISE versions), or in Xilinx Platform Studio (XPS). For this research, we are using the Vivado Design Suite. A variety of design source, including (1) RTL design, (2) IP based design, and (3) Netlist designs, can be implemented in the Xilinx Vivado Design Suite, which is suitable for ultra-Scale FPGA and Xilinx 7 series FPGA design.

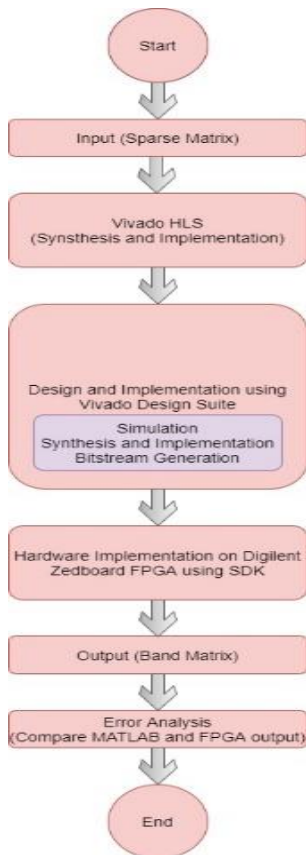


Figure 23. Flowchart of Proposed Work

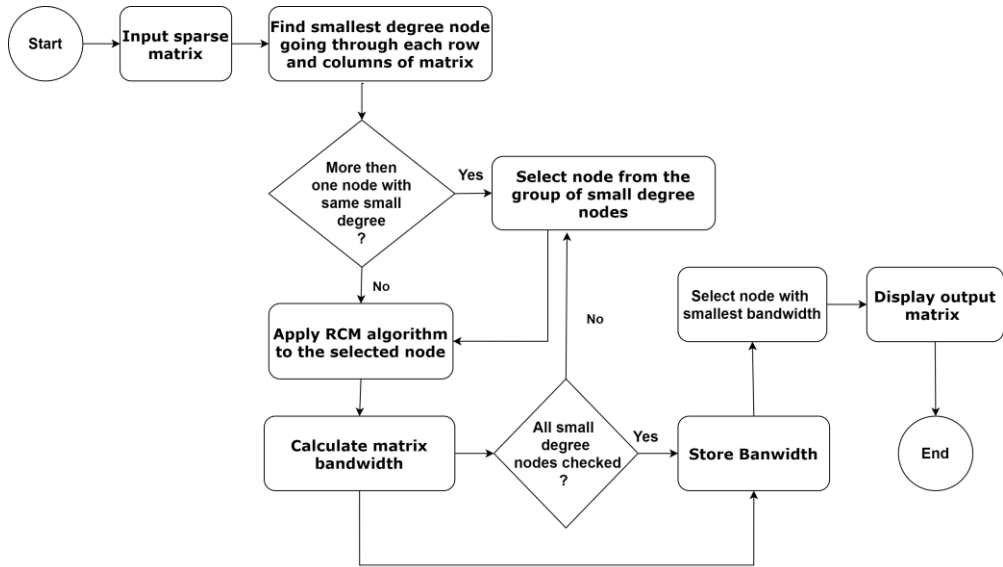
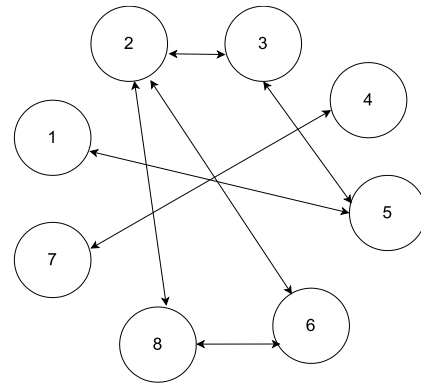


Figure 24. Flowchart of the RCM Algorithm in the System

The operation of the flowchart shown in Figure 24 is discussed in Figure 25 for an 8x8 matrix.

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \end{pmatrix}$$

(a) 8x8 input matrix



(b) Graph representation

Figure 25. Matrix and its Graph Representation

Calculate the degree of connection for each node.

1,4,7 has small degree node

Step 1: r and q array empty array (let)

Step 2: r = {1}, q = {5}

Step 3: r = {1,5}, q = {3}

Step 4: $r = \{1,5,3\}$, $q = \{2\}$

Step 5: $r = \{1,5,3,2\}$, $q = \{6,8\}$

Step 6: $r = \{1,5,3,2,6\}$, $q = \{8\}$

Step 7: $r = \{1,5,3,2,6,8\}$, $q = \{4\}$

Step 8: $r = \{1,5,3,2,6,8,4\}$, $q = \{7\}$

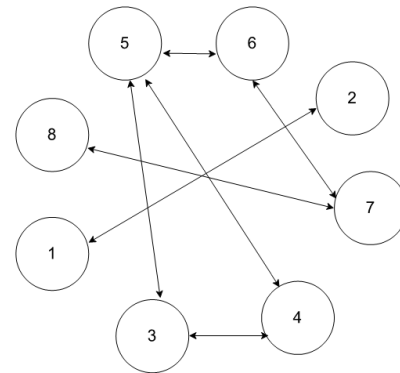
Step 9: $r = \{1,5,3,2,6,8,4,7\}$, $q = \{\}$

Reverse $r = \{7,4,8,6,2,3,5,1\}$

After reordering nodes, the below result is achieved.

$$\begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix}$$

(a) 8x8 output matrix after reorder of nodes



(b) Graph representation

Figure 26. Matrix and Graph after the RCM Reorder

After implementing the RCM algorithm, Figure 25 will be reordered as Figure 26 (a) and Figure 26 (b).

The general design flow for a High-level Vivado Design Suite is shown in Figure 27. A block diagram is designed in Vivado Design Suite with custom IP packaged from Vivado HLS, including the Zynq processor and other necessary IPs. After successful synthesis and implementation of the block diagram, a bitstream is generated for FPGA. This bitstream is used to program FPGA using a Xilinx software development kit (SDK).

SDK is used to design a software version that works with hardware designs created with Vivado Design Suite. SDK is eclipse based on an open-source standard. The result from FPGA is compared with MATLAB for error analysis.

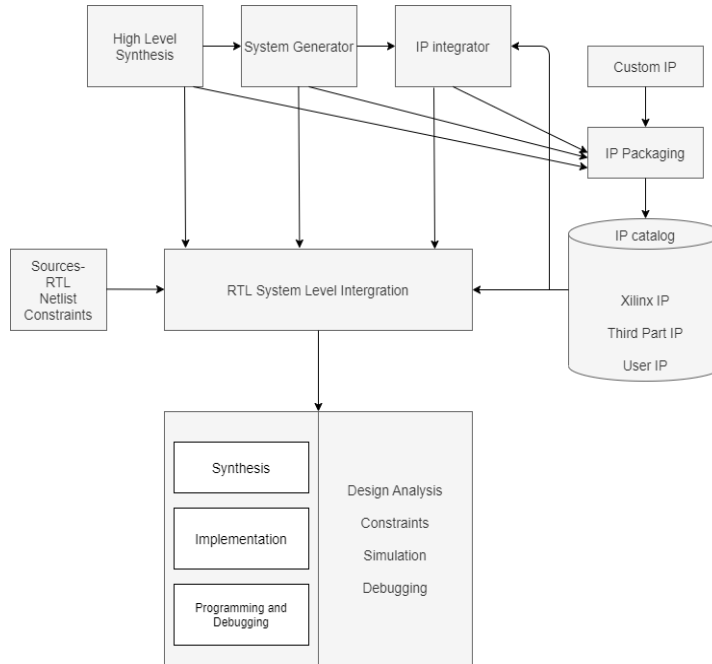


Figure 27. Vivado Design Suite High-Level Design Flow

There are three different ways to program Zedboard, i.e., JTAG, Quad SPI Flash, and SD Card. We have used a JTAG method to program the FPGA board. JTAG is generally used as programming, debugging port, and communicates through the “PROG” micro USB port. Before connecting FPGA to PC via micro USB cord, the pin configuration for JTAG mode must be done, as shown in Figure 28 below.

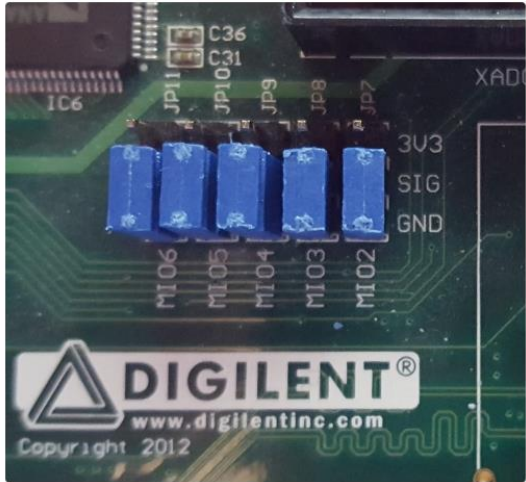


Figure 28. Configuration for JTAG Mode

Software Architecture

Software architecture is designed in Vivado HLS. High-Level Synthesis (HLS), also known as C synthesis, is a behavioral description of hardware that interprets an algorithm description of the desired behavior and creates digital hardware supporting complete bit-accurate validation of the C model. A framework for C-based IP (Intellectual Property) design is a highly preferred approach in electronic design automation for designing digital circuits. HLS is considered an easy and quick hardware programming language as one does not need to have RTL or Verilog/VHDL knowledge because HLS provides RTL IP co-simulation and implementation verification from the C level for hardware implementation. It helps to implement design based on user-defined directives or default property. High-level synthesis schedules and allocates the operations in the behavior as well as maps those operations into component libraries for hardware design. Scheduling and Binding are important properties of HLS. Scheduling helps to determine the clock cycle to perform a specific operation, and binding determines which directive/hardware unit to use for the operation. HLS supports languages such as C, C++,

and SystemC, which brings design flexibility. All programs in HLS are made up of functions to represent a design hierarchy with only one function as Top-level to determine hardware RTL interface port[33]. In Vivado HLS, there is two-step verification: (1) C Validation and (2) RTL Validation. C Validation helps to check if the algorithm is correct before synthesis. Furthermore, RTL validation is a post-synthesis process where RTL is generated from the C based program and is verified. Vivado HLS flow is shown in Figure 29.

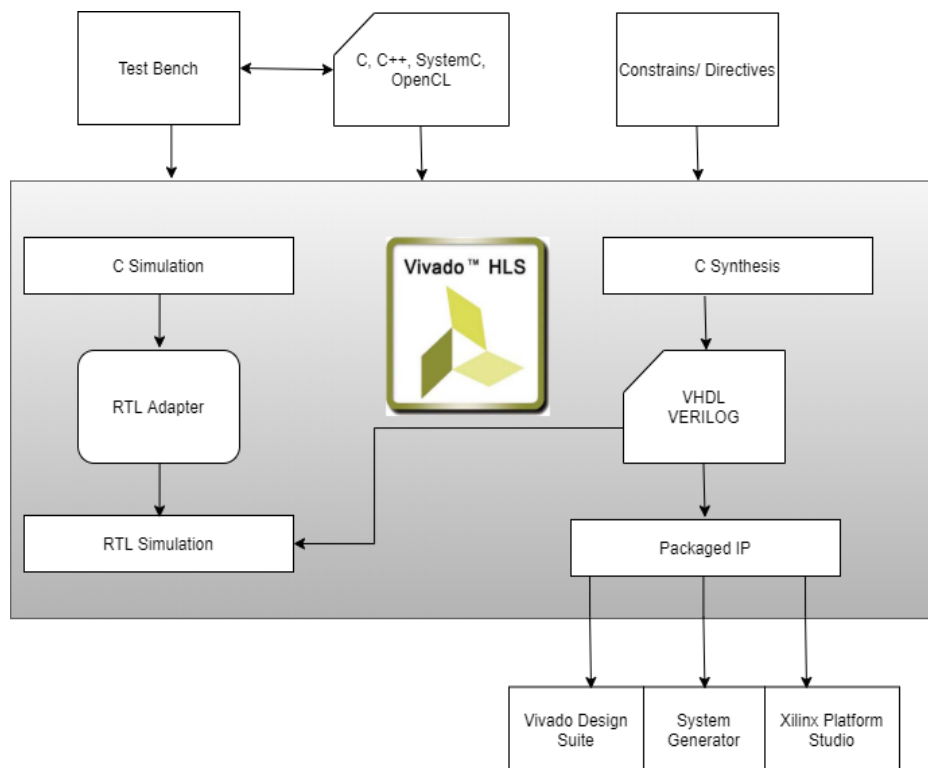


Figure 29. High-Level Synthesis Design Flow

Hardware Architecture

For hardware design, we are using an ARM CPU and coprocessor designed in Vivado HLS and are implemented in the Zynq processor. Zynq has two sections, i.e., 1) PL(Programmable Logic) and PS (Processing System). The PL is connected to an accelerator coherency port (ACP) port via a direct memory access controller (DMA)

controller. An overview of the system to be implemented on Zynq-7000 is described below in Figure 30. Figure 30 shows a streaming data transfer between a processor and FPGA.

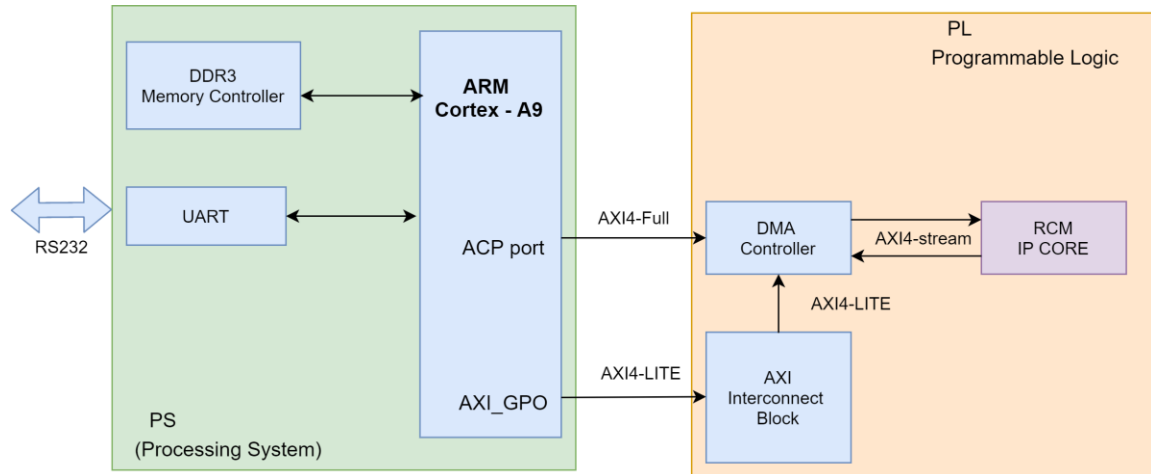


Figure 30. Zynq-7000 AP SoC Architecture

An IP integrator is used to connect the hardware accelerator to an AXI DMA peripheral in the AP SoC PL and to the ACP in the AP SoC PS. The matrix conversion core “RCM” designed in Vivado HLS is connected to the DMA controller via an AXI4-Stream interface. An AXI4-Stream interface is a communication standard for high-speed data transmission between the processor and FPGA on Zynq hardware. Generally, an AXI4-Stream is used with a DMA to transfer data. DMA is a FIFO based method of transferring a large amount of data between a source (PC) and FPGA. There is no engagement of the host processor in DMA; therefore, the data transfer is fast and with minimal interruption from the processor. Data is read by the DMA controller from memory and streams to FPGA via AXI4 Stream with one data element per sample. The FPGA can also have an AXI4-Lite interface for control signals, and they can be connected with other IPs having an AXI4-Lite interface helping them transfer data inside the FPGA.

In the designed architecture, AXI DMA is connected to the L2-cache of the ARM processor through ACP (Accelerated coherency port) port. An ACP port is a 64-bit slave interface that helps with the cache coherency issue by providing direct access to the Cortex-A9 CPU subsystem from Zynq-7000 AP SoC PL.

V. TESTING AND ANALYSIS

The idea of the whole research is to develop a hardware design that can reduce the bandwidth of a sparse matrix as smaller as possible. Different researchers proposed different theories for bandwidth reduction. As the literature review suggest RCM (Reverse Cuthill Algorithm) algorithm one of the best methods, different researchers have proposed their theory with a modified version of the RCM (Reverse Cuthill Algorithm) algorithm. In this work, the original RCM (Reverse Cuthill Algorithm) algorithm is used. The detailed process and testing, and analysis of data done in this research are discussed in other sections given below. Figure 31 is a flowchart for data transmission.

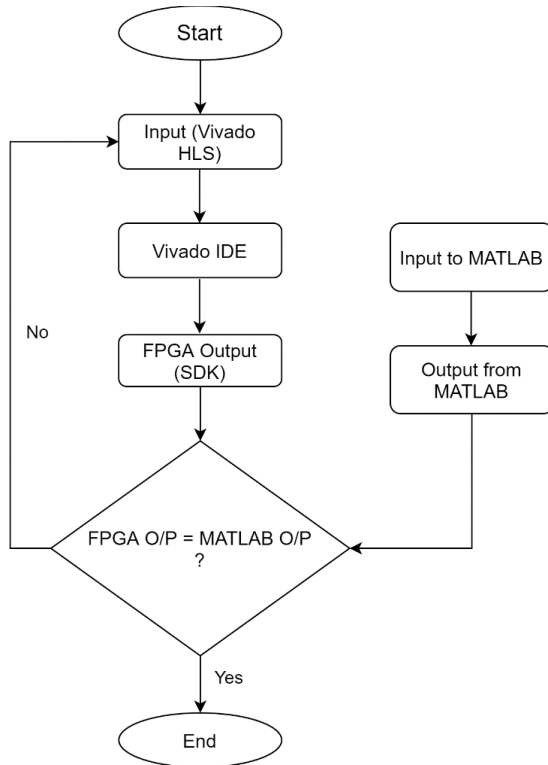


Figure 31. Flowchart of Data Transmission

A hardcoded input is given to the software model, which is in Vivado HLS. After RTL generation and custom IP, it is implemented in Vivado IDE. A bitstream is

generated and implemented into the Diligent Zedboard FPGA device. Data is sent from PC to FPGA using a USB UART serial communication. Tera term and Xilinx SDK terminals are used to display the output.

System Block Diagram in Xilinx Vivado IDE

A Zynq project is designed in Xilinx Vivado Design Suite. Figure 32. is the block diagram of the designed system.

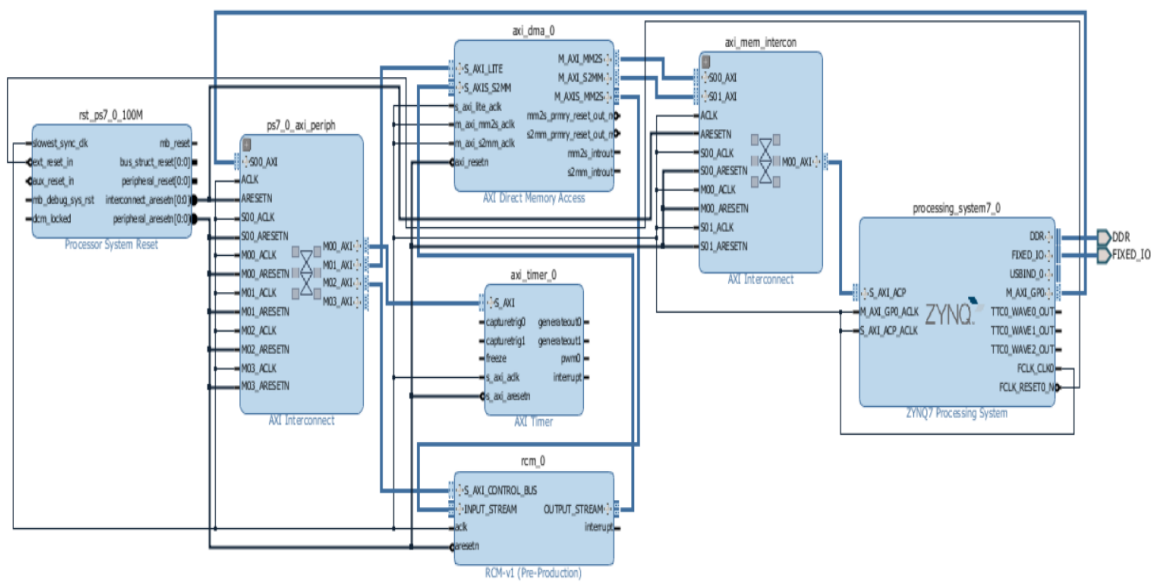


Figure 32. Block Diagram of the System

The main elements in the design have a Zynq system, custom IP generated from Vivado HLS, DMA (Direct Memory Access). After successful implementation and synthesis, an HDL wrapper of the block diagram is created, and a bitstream is generated. The generated bitstream and hardware are exported for SDK (Software Development Kit). A Zynq software is designed in SDK. Finally, FPGA is connected to the PC, and the program is loaded into it. SDK terminal is used to view the output. To verify the hardware generated output, the same input is given to MATLAB, and results are compared. The testing results of the experiments are shown in table 1 below.

Simulation Result

The function simulation of input from PC and the output from FPGA is displayed on ModelSim and Vivado Simulator. Both tools are used to monitor the waveform of simulated hardware description languages such as VHDL, Verilog, SystemC. The baud rate of 115200 bps is used in the simulation. The waveform for an 8x8 input matrix is shown below.

$$\text{8x8 Input Matrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 3 & 4 & 0 & 0 & 5 & 0 & 6 \\ 0 & 4 & 7 & 0 & 8 & 0 & 0 & 0 \\ 0 & 0 & 0 & 9 & 0 & 0 & 2 & 0 \\ 2 & 0 & 8 & 0 & 7 & 0 & 0 & 0 \\ 0 & 5 & 0 & 0 & 0 & 8 & 0 & 9 \\ 0 & 0 & 0 & 2 & 0 & 0 & 6 & 0 \\ 0 & 6 & 0 & 0 & 0 & 9 & 0 & 1 \end{bmatrix}$$

An input waveform for an 8x8 matrix is shown in Figure 33 below. One clock cycle is used to send one element of a matrix. The waveform shows all continuous zeros as one zero value. However, it can be distinguished from the clock signal. Example 100010 matrix data is shown as 1010 in the waveform. We see the clock cycle, in between two 1's three clock cycles are representing three zeros. Figure 34 displays the output simulation. For the output waveform, three clock signals are used to display one matrix element and continuous zeros as one zero value.

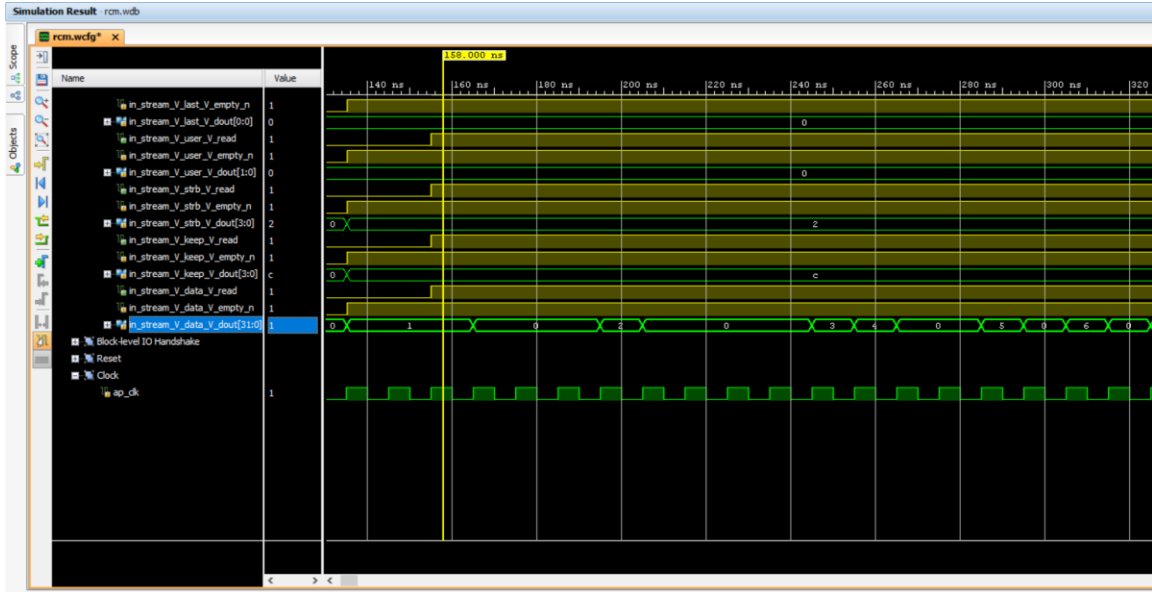


Figure 33. Input Data Waveform

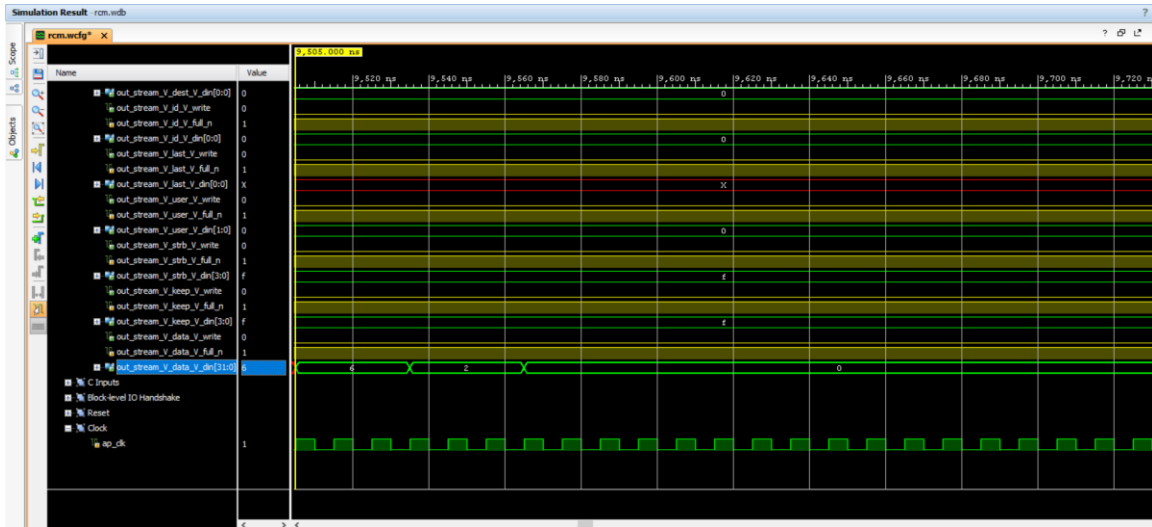


Figure 34. Output Data Waveform

Power Summary Result

In Figure 35, power analysis from the implemented netlist is shown. It can be observed that the total on-chip power is 1.751W, the junction temperature is 45.2 °C, the thermal margin is 38.8 °C, and the power supply to an off-chip device is 0 W.

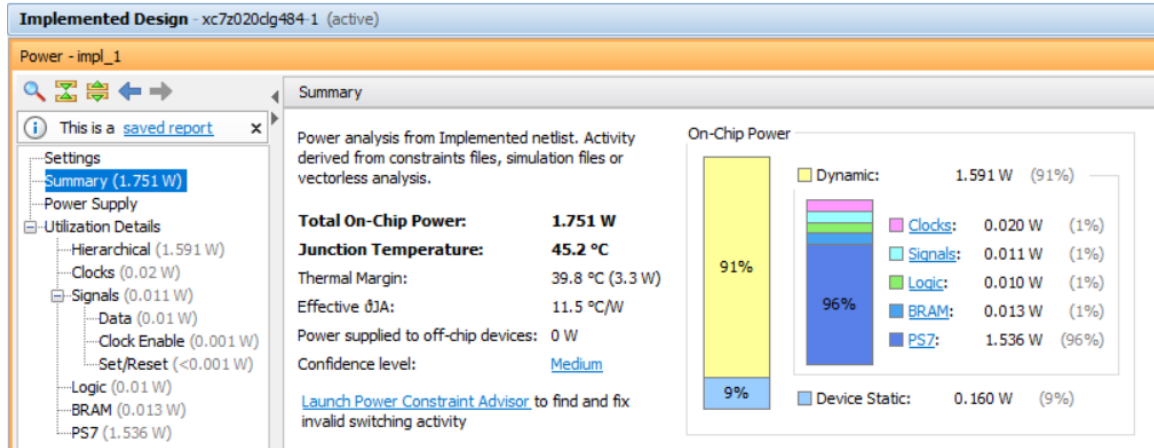


Figure 35. Power Summary

Design Summary of Resource Utilization

$$\begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{bmatrix}$$

7x7 matrix

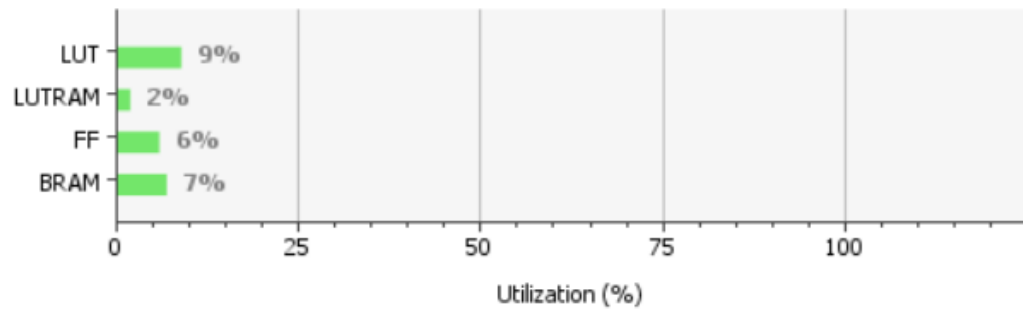
$$\begin{bmatrix} 1 & 0 & 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 3 & 4 & 0 & 0 & 5 & 0 & 6 \\ 0 & 4 & 7 & 0 & 8 & 0 & 0 & 0 \\ 0 & 0 & 0 & 9 & 0 & 0 & 2 & 0 \\ 2 & 0 & 8 & 0 & 7 & 0 & 0 & 0 \\ 0 & 5 & 0 & 0 & 0 & 8 & 0 & 9 \\ 0 & 0 & 0 & 2 & 0 & 0 & 6 & 0 \\ 0 & 6 & 0 & 0 & 0 & 9 & 0 & 1 \end{bmatrix}$$

8x8 Matrix

The synthesized design resources utilization summary for 7x7 and 8x8 matrix is shown in Figure 36 (a) and (b) below. We can see that the number of LUT is 11.12%, the number of slice flip flops is 6.23%, the BRAM is 7.14%. From the result, we can say the design requires fewer resources and is implementable. The design summary results for all input matrices are shown in Table 1 below.

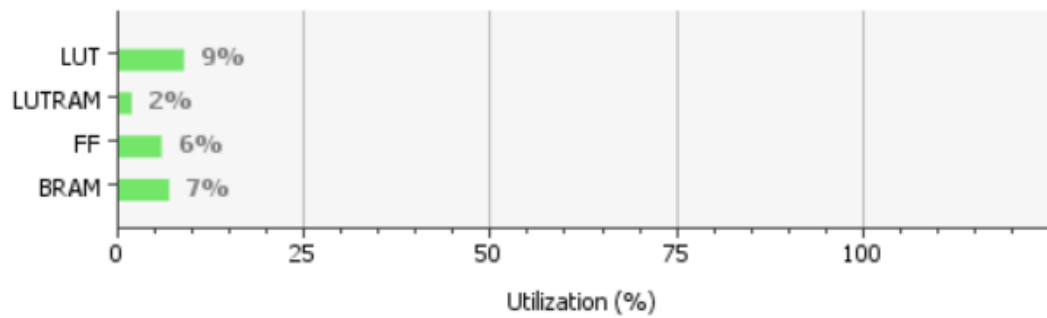
Summary

Resource	Utilization	Available	Utilization %
LUT	4912	53200	9.23
LUTRAM	276	17400	1.59
FF	6376	106400	5.99
BRAM	10	140	7.14



(a) Matrix 8x8

Resource	Utilization	Available	Utilization %
LUT	4946	53200	9.30
LUTRAM	276	17400	1.59
FF	6366	106400	5.98
BRAM	10	140	7.14



(a) Matrix 7x7

Figure 36. Resources Utilization Design Summary

Table 1. Resources Utilization Design Summary

SN	Matrix Size	Matrix Bandwidth	LUT Utilization %	LUT RAM %	FF Utilization %	BRAM Utilization %
1	7x7	9	9.3	1.59	5.98	7.14
2	8x8	15	9.3	1.59	5.99	7.14
3	9x9	17	9.3	1.59	6.01	7.14
4	10x10	13	9.31	1.59	6.02	7.14
5	14x14	23	9.32	1.59	6.02	7.14
6	20x20	38	9.40	1.62	6.06	7.14
7	30x30	53	9.46	1.62	6.07	7.86
8	50x50	87	9.48	1.67	6.15	12.14
9	100x100	189	9.7	1.68	6.2	29.29

Design Timing Summary

The timing summary report is generated automatically after the synthesis and implementation run. Timing summary gives information about a summary of the timing for a design. This summary includes three sections:

1. Setup Area (maximum delay analysis)
2. Hold Area (minimum delay analysis)
3. Pulse Width Area

Explanation of the above timing analysis is explained further below.

1. Setup Area (maximum delay analysis)
 - Worst Negative Slack (WNS): It is the worst positive slack for the maximum delay. If WNS is positive, meaning the path passes within the designed timing constraint, which is 10ns for our design system.
 - Total Negative Slack (TNS): This is the sum of the total negative slack violation. It should be '0' to meet the design timing for max delay analysis.

- Total endpoints: This is the total number of endpoints analyzed.

2. Hold Area (minimum delay analysis)

- Worst Hold Slack (WHS): This is the worst slack of timing for a minimum delay in a design. A positive value means path success, a negative meaning path failure.
- Total Hold Slack: This is the sum of total negative slack violations. It should be '0' to meet the design timing for max delay analysis.
- Total endpoints: This is the total number of endpoints analyzed.

For an 8x8 input matrix, a design timing summary is shown in Figure 37 below.

$$8 \times 8 \text{ Input Matrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 3 & 4 & 0 & 0 & 5 & 0 & 6 \\ 0 & 4 & 7 & 0 & 8 & 0 & 0 & 0 \\ 0 & 0 & 0 & 9 & 0 & 0 & 2 & 0 \\ 2 & 0 & 8 & 0 & 7 & 0 & 0 & 0 \\ 0 & 5 & 0 & 0 & 0 & 8 & 0 & 9 \\ 0 & 0 & 0 & 2 & 0 & 0 & 6 & 0 \\ 0 & 6 & 0 & 0 & 0 & 9 & 0 & 1 \end{bmatrix}$$

Design Timing Summary			
Setup	Hold	Pulse Width	
Worst Negative Slack (WNS): 1.259 ns	Worst Hold Slack (WHS): 0.013 ns	Worst Pulse Width Slack (WPWS): 3.750 ns	
Total Negative Slack (TNS): 0.000 ns	Total Hold Slack (THS): 0.000 ns	Total Pulse Width Negative Slack (TPWS): 0.000 ns	
Number of Failing Endpoints: 0	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0	
Total Number of Endpoints: 17083	Total Number of Endpoints: 17083	Total Number of Endpoints: 6809	
All user specified timing constraints are met.			

Figure 37. Timing Design Summary

Worst negative slack, i.e., the maximum delay of the system for an 8x8 input matrix is 1.259ns and, the minimum delay of the system for the input is 0.013 ns.

Latency and Throughput

Latency is the total time taken to transfer data through a system or a network. It is the time taken to perform some action. Latency can be measured by a one-way time taken to reach the destination or for a full cycle. It is measured in a unit of time like second, minute, hours, nanoseconds, or clock periods.

Throughput gives information about how much data is transferred and received for unit time.

Figure 38 and Figure 39 are input and output waveforms for the data. For the 8x8 input matrix above, the calculation for latency and throughput is shown below.



Figure 38. Input Waveform

For a given input, the time taken to transfer the first data is 135.200 ns.

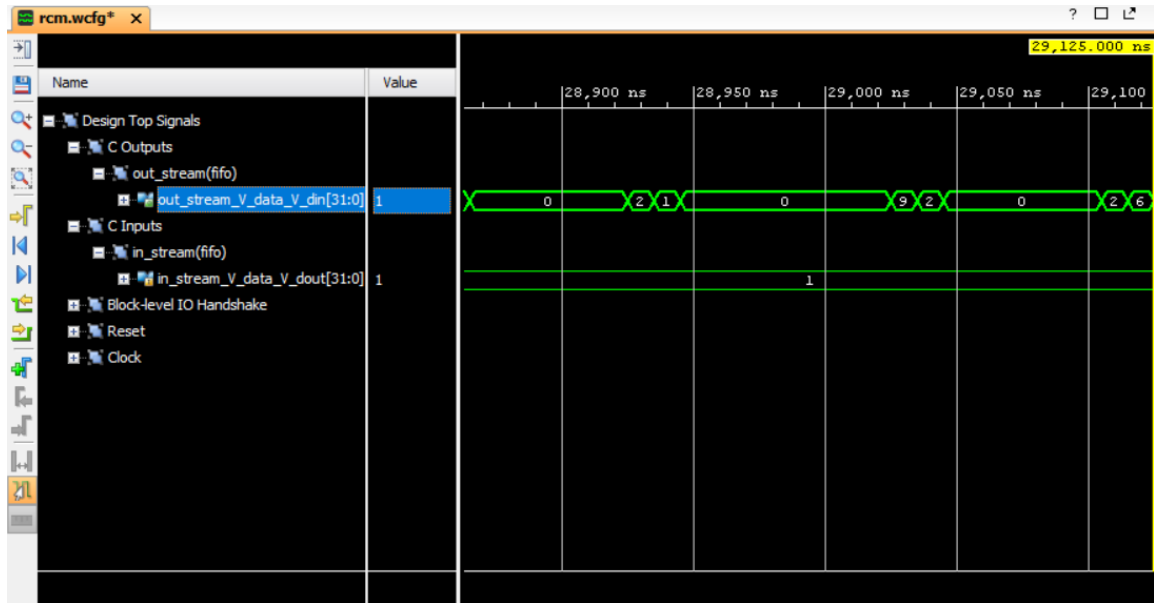


Figure 39. Output Waveform

The total Time taken to generate the final output by the system is 29125ns.

Time constraints for the design are 10 ns (constant in the system).

Therefore latency (of the system) = (last output time – first input time)* timing

constraints = $(29125 - 135.200) * 10\text{ns} = 289898 \text{ ns} = 0.29 \text{ milisecond}$

Throughput of the system = $1\text{sec} / \text{latency}$

Therefore, throughput = $1000 / 0.29 = 3.44\text{Kb/sec}$

This is the latency and throughput of my design.

Latency and throughput for an 8x8 matrix, 20x20 matrix, and 50x50 matrix with two use cases of constant density-variable bandwidth and constant bandwidth-variable density are shown in Table 2 below.

Table 2. Latency and Throughput

Matrix	Density (%)	Bandwidth	Latency	Throughput
8x8	10	12	0.36 ms	2777.7/sec
	10	9	0.28 ms	3571.4/sec
	20	13	0.23ms	4347.8/sec
	34	13	0.29 ms	3448.3/sec
20x20	10	27	0.89ms	1123.5/sec
	10	32	1.04 ms	961 /sec
	20	37	2.09 ms	478.47 /sec
	30	37	3.8 ms	263.16/sec
50x50	10	87	93.92 ms	10.65/sec
	10	76	88.87 ms	11.25/sec
	15	84	30.46	32.83/sec
	5	84	22.42	44.60/sec

VI. RESULTS

List of Test Matrices with RCM Algorithm Application

Table 3. Input and Output Matrices Details

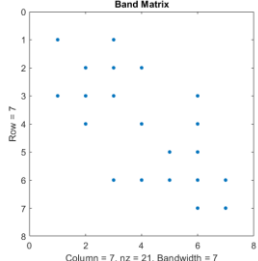
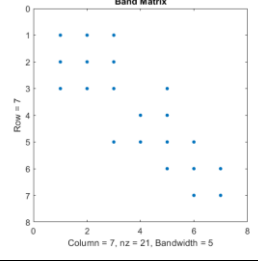
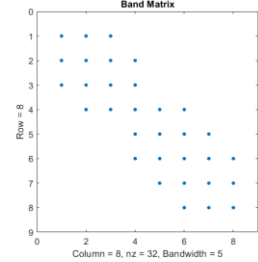
SN	Matrix Size	Total 1's	Total 0's	Density	Original Bandwidth	Bandwidth after RCM
1	7x7	21	28	42.9	9	7
2	7x7	21	28	42.9	9	5
3	8x8	32	32	50.0	15	5
4	8x8	22	42	34.4	13	5
5	9x9	33	48	40.7	17	7
6	10x10	28	72	28.0	13	5
7	10x10	20	80	20.0	15	3
8	10x10	32	68	32.0	17	5
9	10x10	32	68	32.0	16	15
10	14x14	52	144	26.5	23	19
11	14x14	61	135	31.51	23	19
12	14x14	47	149	24.0	25	14
13	15x15	61	164	27.1	29	9
14	20x20	165	235	41.3	38	29
15	20x20	118	282	29.5	37	30
16	20x20	79	321	19.8	37	21
17	24x24	160	416	27.7	43	15
18	25x25	104	521	16.6	46	28
19	25x25	154	471	24.6	46	31
20	30x30	93	807	10.3	53	37
21	30x30	269	631	29.9	57	48
22	30x30	193	707	21.4	55	45
23	50x50	150	2350	6.0	87	17
24	60x60	249	3351	6.9	113	53
25	60x60	366	3234	10.2	111	61
26	100x100	750	9250	7.5	189	121
27	100x100	550	9450	5.5	193	87
28	150x150	938	21562	4.2	285	147

Table 3. contains information for all the input matrices. Random test square sparse matrices are generated through the web. All matrix has a different density, different element location, or are of different sizes. Total elements are the size of a matrix; the density of a matrix is normally calculated by dividing the total numbers of

non-zero values in a matrix by the total elements of a matrix. Total 1's is the total non-zero element, and a total of 0's is the total number of zeros in a matrix. Matrix bandwidth is a sum of upper bandwidth + lower bandwidth+1. For our research, all matrices used are square matrix and non-directional. We can see from the table the difference between the original bandwidth and the new bandwidth calculated.

FPGA and MATLAB Results

Table 4. Comparing FPGA and MATLAB Results

Size	Input Matrix	FPGA Output Image	MATLAB Output Image
49	<pre> 1 1 0 0 0 0 0 1 1 1 0 1 0 0 0 1 1 1 0 1 1 0 0 1 1 0 0 0 0 1 0 0 1 1 0 0 0 1 0 1 1 0 0 0 1 0 0 0 1 </pre>	<pre> * - * - - - - - * * * - - - * * * - - * - - * - * - * - - - - * * * - - - * * * * * - - - - * * </pre> <p>K1 value: 3 K2 value: 3 Final bandwidth: 7</p>	
49	<pre> 1 0 1 0 1 0 0 0 1 0 1 0 1 0 1 0 1 0 1 0 0 0 1 0 1 1 0 1 1 0 1 1 1 0 0 0 1 0 0 0 1 0 0 0 0 1 0 0 1 </pre>	<pre> * * * - - - - * * * - - - - * * * - * - - - - - * * - - - - * * * * - - - - - * * * - - - - * * </pre> <p>K1 value: 2 K2 value: 2 Final bandwidth: 5</p>	
64	<pre> 1 1 0 0 0 1 0 1 1 1 1 0 1 0 0 1 0 1 1 0 1 0 1 0 0 0 0 1 0 1 0 1 0 1 1 0 1 0 1 0 1 0 0 1 0 1 0 1 0 0 1 0 1 0 1 0 1 1 0 1 0 1 0 1 </pre>	<pre> * * * - - - - * * * * - - - * * * * - - - - * * * * - - - - - * * * * - - - * * * * - - - - * * * - - - - * * </pre> <p>K1 value: 2 K2 value: 2 Final bandwidth: 5</p>	

64	<pre> 1 0 0 0 1 0 0 0 0 1 1 0 0 1 0 1 0 1 1 0 1 0 0 0 0 0 0 1 0 0 1 0 1 0 1 0 1 0 0 0 0 1 0 0 0 1 0 1 0 0 0 1 0 0 1 0 0 1 0 0 0 1 0 1 </pre>	<pre> * * * - - - - * * * - - - - * * * * - - - - - * * * - - - - - * * * - - - - - * * - - - - - - * * - - - - - * * </pre> <p> K1 value: 2 K2 value: 2 Final bandwidth: 5 </p>	
81	<pre> 1 0 1 0 0 0 0 1 1 0 1 1 0 0 1 1 1 0 1 1 1 0 0 0 0 0 0 0 0 0 1 0 0 0 1 1 0 0 0 0 1 0 1 1 0 0 1 0 0 0 1 1 1 0 0 1 0 0 1 1 1 0 0 1 1 0 1 1 0 0 1 0 1 0 0 1 0 0 0 0 1 </pre>	<pre> * * * - - - - * * * * - - - * * * - * - - - * - * - * - - - * * - * - - - * * - * * - - - * * * - - - - - * * * - - - - - * * </pre> <p> K1 value: 3 K2 value: 3 Final bandwidth: 7 </p>	
100	<pre> 1 0 0 1 0 0 0 0 0 0 0 1 1 0 0 1 0 0 0 0 0 1 1 0 0 0 0 0 1 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 1 0 1 0 0 0 1 0 1 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 1 1 0 1 1 0 0 0 1 0 0 0 0 1 1 0 0 0 0 0 1 0 1 0 0 1 </pre>	<pre> * * - - - - - * * - - - - - - - * * * - - - - * * - * - - - * * - * - - - - * * * - - - - - * * * - - - - - * * - - - - - * * </pre> <p> K1 value: 2 K2 value: 2 Final bandwidth: 5 </p>	
100	<pre> 1 0 0 1 0 0 0 1 0 0 0 1 0 0 1 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 1 0 0 1 0 0 1 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 1 0 0 1 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 </pre>	<pre> * * - - - - - * * * - - - - - - * * * - - - - * * * - - - - - * * * - - - - - * * - - - - - - * - - - - - - * - - - - - - * - </pre> <p> K1 value: 1 K2 value: 1 Final bandwidth: 3 </p>	
100	<pre> 1 0 0 0 0 1 1 0 0 0 0 1 0 0 0 0 0 0 1 1 0 0 1 0 0 0 1 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 1 0 0 1 0 1 1 0 0 0 0 1 1 0 0 0 1 0 1 0 0 1 1 0 0 0 0 0 0 1 1 0 0 1 0 1 0 1 0 0 0 0 0 0 1 1 0 1 0 0 1 0 0 1 1 1 </pre>	<pre> * * * - - - - * * * - - - - * * * * - - - - - * * * - - - - - - * * * - - - - * * * - - - - * * * - - - - * * * - - - - * * * </pre> <p> K1 value: 2 K2 value: 2 Final bandwidth: 5 </p>	

[illegible]

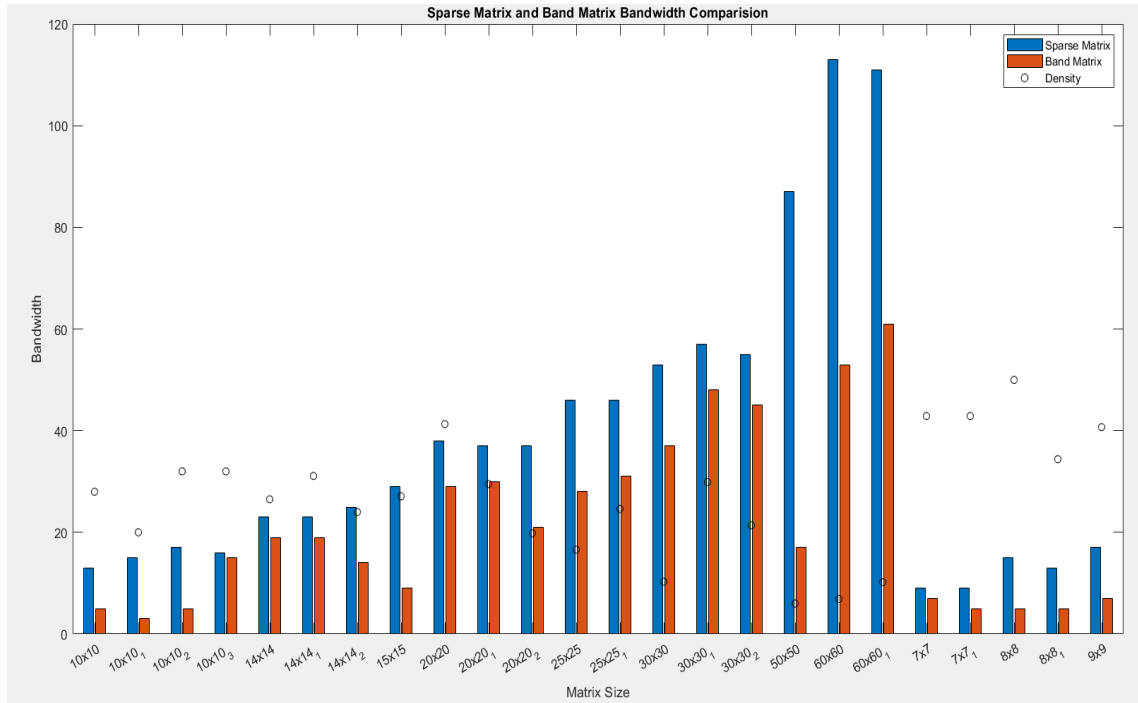


Figure 40. Bar Graph of Bandwidth for Sparse Matrix and Band Matrix

Figure 40 is a bar diagram comparing the initial bandwidth of a sparse matrix and the final bandwidth of a band matrix after using the RCM algorithm. Twenty- three matrices of different size and density are used to test the system. In Figure 40, the blue color represents a sparse matrix, the red color represents for band matrix, and the small dots represent the density of each matrix. From Figure 40, we can see there is a decrease in the bandwidth after the reordering of nodes of a matrix. From the above Figure 40, we can conclude the effectiveness and importance of the RCM algorithm is for bandwidth reduction.

VII. CONCLUSION

A real-time scientific computational application such as Image processing, power system, finite element system, circuit design, etc., utilize a large amount of data, and therefore less memory storage is preferred [1], [3]–[6], [12]. Band matrix with a comparatively small bandwidth and minimized storage are the key component in many scientific computing applications efficiently, leading to less computation time. In this research, FPGA-based hardware implementation for a sparse matrix to a band matrix conversion with a minimum matrix bandwidth using an existing algorithm Reverse Cuthill Mckee algorithm (RCM) is discussed. Furthermore, the hardware implementation is carried out with the knowledge of the hardware architecture.

From this research, the following conclusion can be drawn.

1. There are many areas where a sparse matrix is used.
2. A sparse matrix has fewer nonzero elements randomly scattered.
3. Nonzero elements of a matrix play a vital role in designing a less memory storage system and for faster computation.
4. A band matrix with smaller bandwidth could be a possible solution for memory storage reduction.
5. Matrix size could be an issue in MATLAB due to the finite number of memory it has.
6. FPGA hardware design could be a solution for the size restriction, low latency hardware optimization, and to accelerate operations such as LU, QR optimization.
7. Bandwidth reduction is important for a system such as quantum chemistry, hypertext media, image processing, circuit design, etc.
8. Applications that use sparse matrix data can use the resultant band matrix.

VIII. FUTURE SCOPE

This research is mainly focused on designing a hardware system for the Reverse Cuthill Mckee (RCM) algorithm. The method of finding the first node is like the breadth-first search technique. In this research, while choosing a start node for the RCM algorithm, only a node degree has been considered as it is defined for a sparse matrix reorder. No other heuristic is used or compared to deliver the effectiveness of other algorithms. Different other node finding algorithms are there and can be tested further and compared to see which one gives lesser matrix bandwidth. Gibbs-Poole-Stockmeyer algorithm claim that it is superior to the Reverse Cuthill-Mckee algorithm[3]. On the other hand, different heuristic had been proposed as a modification of existing algorithms mentioning they are better node order algorithm to reduce bandwidth. Therefore, further work can be done on a band matrix using these two algorithms and applying different node finding techniques. Researchers have also suggested that a combination of RCM with other new approaches could give reasonable output. This design could be implemented on large matrices, and complex mathematical operations such as QR and LU factorization can be implemented for a fast and accurate solution. In this research, latency and throughput are calculated for given test matrix values of different sizes, densities, and bandwidth. Various parallelism methods could be used to optimized the design for high throughput and low latency. The resultant output, which is a band matrix, can be used in other algorithms and other hardware for high performance.

APPENDIX SECTION

Program file (Codes):

Main File (.cpp file)

```
// *****Sparse to Band Matrix Conversion ***** //
#include "rcm.h"
#include <iostream>
#include <vector>
#include <fstream>
using namespace hls;
using namespace std;
bool contain(int checkArray[256], int checkElement, int
checkElement_size) {
    bool flag = false;
    check: for (int i = 0; i < checkElement_size; i++) {
        #pragma HLS loop_tripcount min = 1 max = 500
        if (checkArray[i] == checkElement) {
            flag = true;
        }
    }
    return flag;
}
int getPval( in initial_matrix[matrix_size][matrix_size], int
small_nodes[256], int degree[256], int small_nodes_size, int
start_node_value, int p[256], int p_size) {
    int q[256];
    int q_size = 0;
    int reverse_p[256];
    int reverse_p_size = 0;
    int bandwidth[256];
    int bandwidth_size = 0;
    int sort_array[256];
    int sort_array_size = 0;
    q[q_size] = (small_nodes[start_node_value]);
    q_size += 1;
    int ctr = 0;
    //while loop 6
    while (p_size != matrix_size && ctr <= small_nodes_size - 1) {
        #pragma HLS loop_tripcount min = 1 max = 500
        if (contain(p, q[0], p_size) == false) {
            p[p_size] = (q[0]);
            p_size += 1;
        }
        loop6_1: for (int i = 0; i < matrix_size; i++) {
            #pragma HLS loop_tripcount min = 1 max = 500
            if (initial_matrix[q[0] - 1][i] != 0) {
                if (contain(p, i + 1, p_size) == false) {
                    sort_array[sort_array_size] = (i + 1);
                    sort_array_size += 1;
                }
            }
        }
    }

    if (sort_array_size > 1) {
        int min_index;
```



```

loop6_2_row: for (int i = 0; i < sort_array_size; i++) {
    #pragma HLS loop_tripcount min = 1 max = 500
    min_index = i;
    loop6_2_col: for (int j = i + 1; j < sort_array_size; j++) {
        #pragma HLS loop_tripcount min = 1 max = 500
        if (degree[sort_array[j] - 1] < degree[sort_array[min_index]
- 1]) {
            min_index = j;
        }
    }
    swap(sort_array[min_index], sort_array[i]);
}

loop6_3: for (int i = 0; i < q_size; i++) {
    #pragma HLS loop_tripcount min = 1 max = 500
    swap(q[i], q[i + 1]);
}
q_size -= 1;

loop6_4: for (int i = 0; i < sort_array_size; i++) {
    #pragma HLS loop_tripcount min = 1 max = 500
    q[q_size] = (sort_array[i]);
    q_size += 1;
}
sort_array_size = 0;
if (q_size == 0 && p_size != matrix_size) {
    int array_to_select_all_nodes[matrix_size];
    int array_to_select_all_nodes_size = 0;
    loop6_5: for (int i = 0; i < small_nodes_size; i++) {
        #pragma HLS loop_tripcount min = 1 max = 500
        if (small_nodes[i] != small_nodes[start_node_value]) {
            array_to_select_all_nodes[array_to_select_all_nodes_size] =
(small_nodes[i]);
            array_to_select_all_nodes_size += 1;
        }
    }
    q[q_size] = (array_to_select_all_nodes[ctr]);
    q_size += 1;
    ctr += 1;
}

}

return p_size;
}

void sparse_to_band( in a[matrix_size][matrix_size], out
res[matrix_size][matrix_size]) {
    int degree[256];
    int index = 0;
    int degree_count = 0;
    bool flag = false;
    loop1_row: for (int i = 0; i < matrix_size; i++) {
        #pragma HLS loop_tripcount min = 1 max = 500
        loop1_col: for (int j = 0; j < matrix_size; j++) {
            #pragma HLS loop_tripcount min = 1 max = 500

```

```

        if (a[i][j] != 0) {
            degree_count += 1;
        }
    }
    degree[i] = (degree_count);
    degree_count = 0;
    cout << endl;
}
int smallest = degree[0];
loop2: for (int i = 1; i < matrix_size; i++) {
    if (degree[i] < smallest)
        smallest = degree[i];
}
int small_nodes[256];
int small_nodes_size = 0;
loop3: for (int i = 0; i < matrix_size; i++) {
    if (degree[i] == smallest) {
        small_nodes[small_nodes_size] = i + 1;
        small_nodes_size += 1;
    }
}
int p[256], q[256], reverse_p[256], bandwidth[256], sort_array[256];
int p_size = 0, q_size = 0, reverse_p_size = 0, bandwidth_size = 0,
sort_array_size = 0;
int ctr = 0;
int start_node_value = 0;
int array_to_select_all_nodes[256];
int array_to_select_all_nodes_size = 0;
LOOP5_1: while (start_node_value < small_nodes_size) {
    #pragma HLS loop_tripcount min = 1 max = 500
    q_size = 0;
    p_size = 0;
    reverse_p_size = 0;
    p_size = getPval(a, small_nodes, degree, small_nodes_size,
start_node_value, p, p_size);
    if (p_size != matrix_size) {
        int adtnl_array[256];
        int adtnl_array_size = 0;
        int temp = 0;

        LOOP5_2: for (int i = 1; i <= matrix_size; i++) {
            #pragma HLS loop_tripcount min = 1 max = 500
            if (contain(p, i, p_size) == false) {
                adtnl_array[adtnl_array_size] = i;
                adtnl_array_size += 1;
            }
        }
        LOOP5_3_row: for (int i = 0; i < adtnl_array_size; i++) {
            #pragma HLS loop_tripcount min = 1 max = 500
            LOOP5_3_col: for (int j = i + 1; j < adtnl_array_size; j++) {
                #pragma HLS loop_tripcount min = 1 max = 500
                if (degree[adtnl_array[i] - 1] > degree[adtnl_array[j] - 1])
{
                    temp = adtnl_array[i];
                    adtnl_array[i] = adtnl_array[j];
                    adtnl_array[j] = temp;

```

```

    }
    if (degree[adtnl_array[i] - 1] == degree[adtnl_array[j] - 1])
{
    if (adtnl_array[i] > adtnl_array[j]) {
        temp = adtnl_array[i];
        adtnl_array[i] = adtnl_array[j];
        adtnl_array[j] = temp;
    }
}
    }
    p_size = getPval(a, adtnl_array, degree, adtnl_array_size, 0, p,
p_size);
    }
    cout << "*****" << endl;
    cout << "----- HW Result-----" << endl;
    cout << endl << "Final matrix and details for node: " <<
small_nodes[start_node_value] << endl;
    cout << "values in p: ";
    LOOP5_6: for (int i = 0; i < p_size; i++) {
        #pragma HLS loop_tripcount min = 1 max = 500
        cout << p[i] << ", ";
    }
    cout << endl;
    LOOP5_7: for (int i = p_size - 1; i >= 0; i--) {
        #pragma HLS loop_tripcount min = 1 max = 500
        reverse_p[reverse_p_size] = p[i];
        reverse_p_size += 1;
    }
    cout << "values in reverse p: ";
    LOOP5_8: for (int i = 0; i < reverse_p_size; i++) {
        #pragma HLS loop_tripcount min = 1 max = 500
        cout << reverse_p[i] << ", ";
    }
    cout << endl;
    LOOP5_9_ROW: for (int i = 0; i < reverse_p_size; i++) {
        #pragma HLS loop_tripcount min = 1 max = 500
        LOOP5_9_COL: for (int j = 0; j < matrix_size; j++) {
            #pragma HLS loop_tripcount min = 1 max = 500
            res[i][j] = a[reverse_p[i] - 1][reverse_p[j] - 1];
        }
    }

    FINAL_MATRIX_ROW: for (int i = 0; i < matrix_size; i++) {
        #pragma HLS loop_tripcount min = 1 max = 500
        FINAL_MATRIX_COL: for (int j = 0; j < matrix_size; j++) {
            #pragma HLS loop_tripcount min = 1 max = 500
            cout << res[i][j] << "\t";
        }
        cout << endl;
    }
    int max_k1_val = 0;
    int k_val = 0, j_val = 0;
    BANDWIDTH_K1_ROW: for (int i = 0; i < matrix_size; i++) {
        #pragma HLS loop_tripcount min = 1 max = 500
        k_val = 0;
        j_val = 0;

```

```

BANDWIDTH_K1_COL: for (int j = i + 1; j < matrix_size; j++) {
    #pragma HLS loop_tripcount min = 1 max = 500
    if (res[i][j] != 0) {
        j_val = j;
        k_val = j - i;
    }
}
if (k_val > max_k1_val) {
    max_k1_val = k_val;
}
}
cout << "K1 value: " << max_k1_val << endl;
int max_k2_val = 0;
BANDWIDTH_K2_ROW: for (int j = 0; j < matrix_size; j++) {
    #pragma HLS loop_tripcount min = 1 max = 500
    int k_val = 0, i_val = 0;
    BANDWIDTH_K2_COL: for (int i = j + 1; i < matrix_size; i++) {
        #pragma HLS loop_tripcount min = 1 max = 500
        if (res[i][j] != 0) {
            i_val = i;
            k_val = i - j;
        }
    }
    if (k_val > max_k2_val) {
        max_k2_val = k_val;
    }
}
cout << "K2 value: " << max_k2_val << endl;
cout << "Final bandwidth: " << (max_k1_val + max_k2_val + 1) <<
endl;
bandwidth[bandwidth_size] = ((max_k1_val + max_k2_val + 1));
bandwidth_size += 1;
start_node_value += 1;
}
int low_bandwidth = 0;
LOW_BANDWIDTH: for (int i = 1; i < bandwidth_size; i++) {
    #pragma HLS loop_tripcount min = 1 max = 500
    if (bandwidth[i] < bandwidth[low_bandwidth])
        low_bandwidth = i;
}
cout << endl << "Lowest bandwidth: " << bandwidth[low_bandwidth] <<
endl;
cout << "Node with lowest bandwidth: " << small_nodes[low_bandwidth]
<< endl;
}
void rcm(stream < AXI_VALUE > & in_stream, stream < AXI_VALUE > &
out_stream) {
    #pragma HLS RESOURCE variable = in_stream core = AXIS metadata = "-
bus_bundle INPUT_STREAM"
    #pragma HLS RESOURCE variable = out_stream core = AXIS metadata = "-
bus_bundle OUTPUT_STREAM"
    #pragma HLS RESOURCE variable =
    return core = AXI4LiteS metadata = "-bus_bundle CONTROL_BUS" in
sparse[matrix_size][matrix_size];
    out band[matrix_size][matrix_size];
    AXI_VALUE aValue;
    int i, j;

```

```

read_a1: for (i = 0; i < matrix_size; i++) {
    read_a2: for (j = 0; j < matrix_size; j++) {
        #pragma HLS PIPELINE
        in_stream.read(aValue);
        union {
            unsigned int ival; in oval;
        }
        converter;
        converter.ival = aValue.data;
        sparse[i][j] = converter.oval;
    }
}
// conversion from sparse to band matrix
sparse_to_band(sparse, band);
write_res1: for (i = 0; i < matrix_size; i++) {
    write_res2: for (j = 0; j < matrix_size; j++) {
        #pragma HLS PIPELINE
        union {
            unsigned int oval;
            out ival;
        }
        converter;
        converter.ival = band[i][j];
        aValue.data = converter.oval;
        aValue.last = ((i == matrix_size - 1) && (j == matrix_size - 1))
? 1 : 0;
        aValue.strb = -1;
        aValue.keep = 15; //e.strb;
        aValue.user = 0;
        aValue.id = 0;
        aValue.dest = 0;
        out_stream.write(aValue);
    }
}
}
}

```

Test Bench File: RCM_tb.cpp

```

// Sparse to Band Matrix Conversion testbench//.
#include <iostream>

#include <fstream>

#include <string>

#include <stdio.h>

#include "rcm.h"

using namespace std;
int getP( in initial_matrix[matrix_size][matrix_size], int
small_nodes[256], int degree[256], int small_nodes_size, int
start_node_value, int p[256], int p_size);
void gen_sw_mat( in initial_matrix[matrix_size][matrix_size], out
sw_final_matrix[matrix_size][matrix_size]);
void check_mat_result(out hw_res[matrix_size][matrix_size], out
sw_res[matrix_size][matrix_size], int & err_cnt);

```

```

bool contains(int checkArray[256], int checkElement, int
checkElement_size);
bool contains(int checkArray[256], int checkElement, int
checkElement_size) {
    bool flag = false;
    check: for (int i = 0; i < checkElement_size; i++) {
        #pragma HLS loop_tripcount min = 1 max = 500
        if (checkArray[i] == checkElement) {
            flag = true;
        }
    }
    return flag;
}

int getP( in initial_matrix[matrix_size][matrix_size], int
small_nodes[256], int degree[256], int small_nodes_size, int
start_node_value, int p[256], int p_size) {
    int q[256];
    int q_size = 0;
    int reverse_p[256];
    int reverse_p_size = 0;
    int bandwidth[256];
    int bandwidth_size = 0;
    int sort_array[256];
    int sort_array_size = 0;
    q[q_size] = (small_nodes[start_node_value]);
    q_size += 1;
    int ctr = 0;
    while (p_size != matrix_size && ctr <= small_nodes_size - 1) {
        #pragma HLS loop_tripcount min = 1 max = 500
        if (contains(p, q[0], p_size) == false) {
            p[p_size] = (q[0]);
            p_size += 1;
        }
        loop6_1: for (int i = 0; i < matrix_size; i++) {
            #pragma HLS loop_tripcount min = 1 max = 500
            if (initial_matrix[q[0] - 1][i] != 0) {
                if (contains(p, i + 1, p_size) == false) {
                    sort_array[sort_array_size] = (i + 1);
                    sort_array_size += 1;
                }
            }
        }
        if (sort_array_size > 1) {
            int min_index;
            loop6_2_row: for (int i = 0; i < sort_array_size; i++) {
                #pragma HLS loop_tripcount min = 1 max = 500
                min_index = i;
                loop6_2_col: for (int j = i + 1; j < sort_array_size; j++) {
                    #pragma HLS loop_tripcount min = 1 max = 500
                    if (degree[sort_array[j] - 1] < degree[sort_array[min_index]
- 1]) {
                        min_index = j;
                    }
                }
                swap(sort_array[min_index], sort_array[i]);
            }
        }
    }
}

```

```

loop6_3: for (int i = 0; i < q_size; i++) {
    #pragma HLS loop_tripcount min = 1 max = 500
    swap(q[i], q[i + 1]);
}
q_size -= 1;
loop6_4: for (int i = 0; i < sort_array_size; i++) {
    #pragma HLS loop_tripcount min = 1 max = 500
    q[q_size] = (sort_array[i]);
    q_size += 1;
}
sort_array_size = 0;
if (q_size == 0 && p_size != matrix_size) {
    int array_to_select_all_nodes[matrix_size];
    int array_to_select_all_nodes_size = 0;
    loop6_5: for (int i = 0; i < small_nodes_size; i++) {
        #pragma HLS loop_tripcount min = 1 max = 500
        if (small_nodes[i] != small_nodes[start_node_value]) {
            array_to_select_all_nodes[array_to_select_all_nodes_size] =
(small_nodes[i]);
            array_to_select_all_nodes_size += 1;
        }
    }
    q[q_size] = (array_to_select_all_nodes[ctr]);
    q_size += 1;
    ctr += 1;
}
}
return p_size;
}
void gen_sw_mat( in initial_matrix[matrix_size][matrix_size], out
sw_final_matrix[matrix_size][matrix_size])

{
    int degree[256];
    int index = 0;
    int degree_count = 0;
    bool flag = false;
    cout <<
"*****";
    cout << endl;
    cout << "Input Sparse Matrix" << endl;
    cout << endl;
    loop1_row: for (int i = 0; i < matrix_size; i++) {
        #pragma HLS loop_tripcount min = 1 max = 500
        loop1_col: for (int j = 0; j < matrix_size; j++) {
            #pragma HLS loop_tripcount min = 1 max = 500
            if (initial_matrix[i][j] != 0) {
                degree_count += 1;
            }
            cout << initial_matrix[i][j] << " ";
        }
        degree[i] = (degree_count);
        degree_count = 0;
        cout << endl;
    }
    int smallest = degree[0];
    loop2: for (int i = 1; i < matrix_size; i++) {

```

```

    if (degree[i] < smallest)
        smallest = degree[i];
    cout << "smallest node value/degree is " << smallest << endl;
    int small_nodes[256];
    int small_nodes_size = 0;
    loop3: for (int i = 0; i < matrix_size; i++) {
        if (degree[i] == smallest) {
            small_nodes[small_nodes_size] = i + 1;
            small_nodes_size += 1;
        }
    }
    cout << "Nodes with smallest degree are: ";
    loop4: for (int i = 0; i < small_nodes_size; i++) {
        #pragma HLS loop_tripcount min = 1 max = 500
        cout << small_nodes[i] << ", ";
    }
    cout << endl;
    int p[256];
    int p_size = 0;
    int q[256];
    int q_size = 0;
    int reverse_p[256];
    int reverse_p_size = 0;
    int bandwidth[256];
    int bandwidth_size = 0;
    int sort_array[256];
    int sort_array_size = 0;
    int ctr = 0;
    int start_node_value = 0;
    int array_to_select_all_nodes[256];
    int array_to_select_all_nodes_size = 0;
    LOOP5_1: while (start_node_value < small_nodes_size) {
        #pragma HLS loop_tripcount min = 1 max = 500
        q_size = 0;
        p_size = 0;
        reverse_p_size = 0;
        p_size = getP(initial_matrix, small_nodes, degree,
small_nodes_size, start_node_value, p, p_size);
        if (p_size != matrix_size) {
            int adtnl_array[256];
            int adtnl_array_size = 0;
            int temp = 0;
            LOOP5_2: for (int i = 1; i <= matrix_size; i++) {
                #pragma HLS loop_tripcount min = 1 max = 500
                if (contains(p, i, p_size) == false) {
                    adtnl_array[adtnl_array_size] = i;
                    adtnl_array_size += 1;
                }
            }
            LOOP5_3_row: for (int i = 0; i < adtnl_array_size; i++) {
                #pragma HLS loop_tripcount min = 1 max = 500
                LOOP5_3_col: for (int j = i + 1; j < adtnl_array_size; j++) {
                    #pragma HLS loop_tripcount min = 1 max = 500
                    if (degree[adtnl_array[i] - 1] > degree[adtnl_array[j] -
1]) {
                        temp = adtnl_array[i];
                        adtnl_array[i] = adtnl_array[j];

```



```

        adtnl_array[j] = temp;
    }
    if (degree[adtnl_array[i] - 1] == degree[adtnl_array[j] -
1]) {
        if (adtnl_array[i] > adtnl_array[j]) {
            temp = adtnl_array[i];
            adtnl_array[i] = adtnl_array[j];
            adtnl_array[j] = temp;
        }
    }
}
cout << endl;
cout << "sorted additional nodes:";
LOOP5_4: for (int i = 0; i < adtnl_array_size; i++) {
    #pragma HLS loop_tripcount min = 1 max = 500
    cout << adtnl_array[i] << " ";
}
cout << endl;
p_size = getP(initial_matrix, adtnl_array, degree,
adtnl_array_size, 0, p, p_size);
}
cout <<
"*****" << endl;
cout << endl << "Final matrix and details for node: " <<
small_nodes[start_node_value] << endl;
cout << "values in p: ";
LOOP5_6: for (int i = 0; i < p_size; i++) {
    #pragma HLS loop_tripcount min = 1 max = 500
    cout << p[i] << ", ";
}
cout << endl;
// printing final matrix
LOOP5_7: for (int i = p_size - 1; i >= 0; i--) {
    #pragma HLS loop_tripcount min = 1 max = 500
    reverse_p[reverse_p_size] = p[i];
    reverse_p_size += 1;
}
cout << "values in reverse p: ";
LOOP5_8: for (int i = 0; i < reverse_p_size; i++) {
    #pragma HLS loop_tripcount min = 1 max = 500
    cout << reverse_p[i] << ", ";
}
cout << endl;
cout << endl;
LOOP5_9_ROW: for (int i = 0; i < reverse_p_size; i++) {
    #pragma HLS loop_tripcount min = 1 max = 500
    LOOP5_9_COL: for (int j = 0; j < matrix_size; j++) {
        #pragma HLS loop_tripcount min = 1 max = 500
        sw_final_matrix[i][j] = initial_matrix[reverse_p[i] -
1][reverse_p[j] - 1];
    }
}
cout << "-----SW result-----" << endl;
cout << endl;
FINAL_MATRIX_ROW: for (int i = 0; i < matrix_size; i++) {
    #pragma HLS loop_tripcount min = 1 max = 500

```

```

FINAL_MATRIX_COL: for (int j = 0; j < matrix_size; j++) {
    RCM_tb.cpp
    #pragma HLS loop_tripcount min = 1 max = 500
    cout << sw_final_matrix[i][j] << "\t";
    cout << " ";
}
cout << endl;
}
int max_k1_val = 0;
int k_val = 0, j_val = 0;
BANDWIDTH_K1_ROW: for (int i = 0; i < matrix_size; i++) {
    #pragma HLS loop_tripcount min = 1 max = 500
    k_val = 0;
    j_val = 0;
    BANDWIDTH_k1_COL: for (int j = i + 1; j < matrix_size; j++) {
        #pragma HLS loop_tripcount min = 1 max = 500
        if (sw_final_matrix[i][j] != 0) {
            j_val = j;
            k_val = j - i;
        }
    }
    if (k_val > max_k1_val) {
        max_k1_val = k_val;
    }
}
cout << "K1 value: " << max_k1_val << endl;
int max_k2_val = 0;
BANDWIDTH_K2_ROW: for (int j = 0; j < matrix_size; j++) {
    #pragma HLS loop_tripcount min = 1 max = 500
    int k_val = 0, i_val = 0;
    BANDWIDTH_K2_COL: for (int i = j + 1; i < matrix_size; i++) {
        #pragma HLS loop_tripcount min = 1 max = 500
        if (sw_final_matrix[i][j] != 0) {
            i_val = i;
            k_val = i - j;
        }
    }
    if (k_val > max_k2_val) {
        max_k2_val = k_val;
    }
}
cout << "K2 value: " << max_k2_val << endl;
cout << "Final bandwidth: " << (max_k1_val + max_k2_val + 1) <<
endl;
bandwidth[bandwidth_size] = ((max_k1_val + max_k2_val + 1));
bandwidth_size += 1;
start_node_value += 1;
}

int low_bandwidth = 0;
LOW_BANDWIDTH: for (int i = 1; i < bandwidth_size; i++) {
    #pragma HLS loop_tripcount min = 1 max = 500
    if (bandwidth[i] < bandwidth[low_bandwidth])
        low_bandwidth = i;
}
cout << endl << "Lowest bandwidth: " << bandwidth[low_bandwidth] <<
endl;

```

```

        cout << "Node with lowest bandwidth: " <<
small_nodes[low_bandwidth] << endl;

    }

    Void check_mat_result(out hw_res[matrix_size][matrix_size], out
sw_res[matrix_size][matrix_size], int & err_cnt) {
    int i, j;
    FINAL_MATRIX_ROW: for (i = 0; i < matrix_size; i++) {
        #pragma HLS loop_tripcount min = 1 max = 500
        FINAL_MATRIX_COL: for (j = 0; j < matrix_size; j++) {
            #pragma HLS loop_tripcount min = 1 max = 500
            cout << hw_res[i][j];
            cout << " ";
            if (hw_res[i][j] != sw_res[i][j]) {
                err_cnt++;
                cout << "*";
            }
        }
        cout << endl;
    }
}

// Test the streaming interface.
int test_matrix_core() {
    in A[matrix_size][matrix_size] = {
        {1, 0, 0, 0, 2, 0, 0, 0 }, {0, 3, 4, 0, 0, 5, 0,
6}, {0, 4, 7, 0, 8, 0, 0, 0}, {0, 0, 0, 9, 0, 0, 2, 0}, {2, 0, 8, 0, 7, 0, 0, 0}, {0, 5, 0, 0, 0, 8,
0, 9}, {0, 0, 0, 2, 0, 0, 6, 0}, {0, 6, 0, 0, 0, 9, 0, 1}
    };
    out sw_result[matrix_size][matrix_size],
hw_result[matrix_size][matrix_size];
    int i, j, err_cnt = 0;
    hls::stream < AXI_VALUE > in_stream;
    hls::stream < AXI_VALUE > out_stream;
    AXI_VALUE aValue;
    gen_sw_mat(A, sw_result);
    for (i = 0; i < matrix_size; i++) {
        for (j = 0; j < matrix_size; j++) {
            union {
                unsigned int oval; in ival;
            }
            converter;
            converter.ival = A[i][j];
            aValue.data = converter.oval;
            in_stream.write(aValue);
        }
    }
    rcm(in_stream, out_stream);
    for (i = 0; i < matrix_size; i++) {
        for (j = 0; j < matrix_size; j++) {
            out_stream.read(aValue);
            union {
                unsigned int ival;
                out oval;
            }
            converter;
            converter.ival = aValue.data;

```

```

        hw_result[i][j] = converter.oval;
    }
}
check_mat_result(hw_result, sw_result, err_cnt);
if (err_cnt)
    cout << "ERROR: " << err_cnt << " mismatches detected!" << endl;
else
    cout << "Test passed. No errors" << endl;
return err_cnt;
}
//-----
int main(int argc, char ** argv) {
    int err_cnt;
    err_cnt = test_matrix_core();
    return err_cnt;
}

```

Header File:

RCM.h

```

#ifndef __RCM_H_
#define __RCM_H_
#include <iostream>
#include <cmath>
#include <stdio.h>
#include <ap_axi_sdata.h>
#include <hls_stream.h>
#define matrix_size 8
typedef int in;
typedef int out;
typedef ap_axiu < 32, 2, 1, 1 > AXI_VALUE;

Void sparse_to_band( in a[matrix_size][matrix_size], out
res[matrix_size][matrix_size]);
Void rcm(hls::stream < AXI_VALUE > & in_stream, hls::stream < AXI_VALUE
> & out_stream);
#endif

```

ARM C CODE (SDK Code):

Main C CODE:

```

// *****Sparse Matrix to Band Matrix conversion*****//
#include <stdio.h>
#include <stdlib.h>
#include "platform.h"
#include "xil_io.h"
#include "platform.h"
#include "xparameters.h"
#include "xaxidma.h"
#include "xtmrctr.h"
#include "stdbool.h"
#include "sparse_to_band.h"
#include "xil_printf.h"

XAxidma AxiDma;
XTmrCtr timer_dev;
XRcm XMatcon_dev;

```

```

XRcm_Config XMatcon_config = {
    0,
    XPAR_RCM_0_S_AXI_CONTROL_BUS_BASEADDR
};
int init_dma() {
    XAxiDma_Config * cPitr;
    int status;
    CfgPitr = XAxiDma_LookupConfig(XPAR_AXI_DMA_0_DEVICE_ID);
    if (!CfgPitr) {
        print("AXI DMA config error\n\r");
        return XST_FAILURE;
    }
    status = XAxiDma_CfgInitialize( & AxiDma, CfgPitr);
    if (status != XST_SUCCESS) {
        print("DMA initializing error \n\r");
        return XST_FAILURE;
    }
}
//-----
bool contain(int checkArray[256], int checkElement, int
checkElement_size) {
    bool flag = false;
    for (int i = 0; i < checkElement_size; i++) {
        if (checkArray[i] == checkElement) {
            flag = true;
        }
    }
    return flag;
}
int getPval(int initial_matrix[DIM][DIM], int small_nodes[256], int
degree[256], int small_nodes_size, int start_node_value, int p[256],
int p_size) {
    int q[256];
    int q_size = 0;
    int reverse_p[256];
    int reverse_p_size = 0;
    int bandwidth[256];
    int bandwidth_size = 0;
    int sort_array[256];
    int sort_array_size = 0;
    q[q_size] = (small_nodes[start_node_value]);
    q_size += 1;
    int ctr = 0;
    while (p_size != DIM && ctr <= small_nodes_size - 1) {
        if (contain(p, q[0], p_size) == false) {
            p[p_size] = (q[0]);
            p_size += 1;
        }
        for (int i = 0; i < DIM; i++) {
            if (initial_matrix[q[0] - 1][i] != 0) {
                if (contain(p, i + 1, p_size) == false) {
                    sort_array[sort_array_size] = (i + 1);
                    sort_array_size += 1;
                }
            }
        }
        if (sort_array_size > 1) {

```

```

    int min_index;
    for (int i = 0; i < sort_array_size; i++) {
        min_index = i;
        for (int j = i + 1; j < sort_array_size; j++) {
            if (degree[sort_array[j] - 1] < degree[sort_array[min_index]
- 1]) {
                min_index = j;
            }
        }
        int temp = sort_array[min_index];
        sort_array[min_index] = sort_array[i];
        sort_array[i] = temp;
    }
}
int i;
for (i = 0; i < q_size; i++) {
    int temp1 = q[i];
    q[i] = q[i + 1];
    q[i + 1] = temp1;
}
q_size -= 1;
for (int i = 0; i < sort_array_size; i++) {
    q[q_size] = (sort_array[i]);
    q_size += 1;
}
sort_array_size = 0;
if (q_size == 0 && p_size != DIM) {
    int array_to_select_all_nodes[DIM];
    int array_to_select_all_nodes_size = 0;
    for (int i = 0; i < small_nodes_size; i++) {
        if (small_nodes[i] != small_nodes[start_node_value]) {
            array_to_select_all_nodes[array_to_select_all_nodes_size] =
(small_nodes[i]);
            array_to_select_all_nodes_size += 1;
        }
    }
    q[q_size] = (array_to_select_all_nodes[ctr]);
    q_size += 1;
    ctr += 1;
}
}
return p_size;
}
void sparse_to_band(int a[DIM][DIM], int res[DIM][DIM]) {
    int degree[256];
    int index = 0;
    int degree_count = 0;
    bool flag = false;
    for (int i = 0; i < DIM; i++) {
        for (int j = 0; j < DIM; j++) {
            if (a[i][j] != 0) {
                degree_count += 1;
            }
        }
        degree[i] = (degree_count);
        degree_count = 0;
    }
}

```

```

int smallest = degree[0];
for (int i = 1; i < DIM; i++) {
    if (degree[i] < smallest)
        smallest = degree[i];
}
int small_nodes[256];
int small_nodes_size = 0;
for (int i = 0; i < DIM; i++) {
    if (degree[i] == smallest) {
        small_nodes[small_nodes_size] = i + 1;
        small_nodes_size += 1;
    }
}
int p[256], q[256], reverse_p[256], bandwidth[256], sort_array[256];
int p_size = 0, q_size = 0, reverse_p_size = 0, bandwidth_size = 0,
sort_array_size = 0;
int ctr = 0;
int start_node_value = 0;
int array_to_select_all_nodes[256];
int array_to_select_all_nodes_size = 0;
while (start_node_value < small_nodes_size) {
    q_size = 0;
    p_size = 0;
    reverse_p_size = 0;
    p_size = getPval(a, small_nodes, degree, small_nodes_size,
start_node_value, p, p_size);
    if (p_size != DIM) {
        int adtnl_array[256];
        int adtnl_array_size = 0;
        int temp = 0;
        for (int i = 1; i <= DIM; i++) {
            if (contain(p, i, p_size) == false) {
                adtnl_array[adtnl_array_size] = i;
                adtnl_array_size += 1;
            }
        }
        for (int i = 0; i < adtnl_array_size; i++) {
            for (int j = i + 1; j < adtnl_array_size; j++) {
                if (degree[adtnl_array[i] - 1] > degree[adtnl_array[j] - 1])
{
                    temp = adtnl_array[i];
                    adtnl_array[i] = adtnl_array[j];
                    adtnl_array[j] = temp;
                }
                if (degree[adtnl_array[i] - 1] == degree[adtnl_array[j] - 1])
{
                    if (adtnl_array[i] > adtnl_array[j]) {
                        temp = adtnl_array[i];
                        adtnl_array[i] = adtnl_array[j];
                        adtnl_array[j] = temp;
                    }
                }
            }
        }
        p_size = getPval(a, adtnl_array, degree, adtnl_array_size, 0, p,
p_size);
    }
}

```

```

for (int i = p_size - 1; i >= 0; i--) {
    reverse_p[reverse_p_size] = p[i];
    reverse_p_size += 1;
}
for (int i = 0; i < reverse_p_size; i++) {
    for (int j = 0; j < DIM; j++) {
        res[i][j] = a[reverse_p[i] - 1][reverse_p[j] - 1];
    }
}
xil_printf("-----SW Result-----:\n");
for (int i = 0; i < DIM; i++) {
    for (int j = 0; j < DIM; j++) {
        //if(res[i][j]!=0)
        xil_printf("%d", res[i][j]);
        xil_printf("\t");
    }
    xil_printf("\n");
}
int max_k1_val = 0;
int k_val = 0, j_val = 0;
for (int i = 0; i < DIM; i++) {
    k_val = 0;
    j_val = 0;
    for (int j = i + 1; j < DIM; j++) {
        if (res[i][j] != 0) {
            j_val = j;
            k_val = j - i;
        }
    }
    if (k_val > max_k1_val) {
        max_k1_val = k_val;
    }
}
xil_printf("K1 value:\t%d", max_k1_val);
xil_printf("\n");
int max_k2_val = 0;
for (int j = 0; j < DIM; j++) {
    int k_val = 0, i_val = 0;
    for (int i = j + 1; i < DIM; i++) {
        if (res[i][j] != 0) {
            i_val = i;
            k_val = i - j;
        }
    }
    if (k_val > max_k2_val) {
        max_k2_val = k_val;
    }
}
xil_printf("K2 value:\t%d", max_k2_val);
xil_printf("\n");
xil_printf("Final bandwidth:\t%d", max_k1_val + max_k2_val + 1);
xil_printf("\n");
bandwidth[bandwidth_size] = ((max_k1_val + max_k2_val + 1));
bandwidth_size += 1;
start_node_value += 1;
}
int low_bandwidth = 0;

```



```

    for (int i = 1; i < bandwidth_size; i++) {
        if (bandwidth[i] < bandwidth[low_bandwidth])
            low_bandwidth = i;
    }
    xil_printf("Lowest bandwidth:\t%d", bandwidth[low_bandwidth]);
    xil_printf("\n");
    xil_printf("Node with lowest bandwidth:\t%d",
small_nodes[low_bandwidth]);
    xil_printf("\n");
}

void check_conversion_result(int HW_result[DIM][DIM], int
res[DIM][DIM], int * err_cnt {
    xil_printf("\n")
    for (int i = 0; i < DIM; i++) {
        for (int j = 0; j < DIM; j++) {
            if (HW_result[i][j] != 0)
                xil_printf("%d", HW_result[i][j]);
            else
                xil_printf("*");
            if (HW_result[i][j] != res[i][j]) {
                err_cnt++;
            }
            if (j == DIM - 1)
                xil_printf("\n");
            else
                xil_printf(" ");
        }
    }
    xil_printf("\n");
}

//-----

void print_mat_conversion_status(void) {
    int isDone, isIdle, isReady;
    isDone = XRcm_IsDone( & XMatcon_dev);
    isIdle = XRcm_IsIdle( & XMatcon_dev);
    isReady = XRcm_IsReady( & XMatcon_dev);
    xil_printf("Matconversion Status: is Done\t%d,\tis Idle\t%d,\tis
Ready\t%d\n", isDone, isIdle, isReady);
}

//-----

int main() {
    int i, j;
    char c;
    int status, err = 0;
    int A[DIM][DIM] = {
        {1,0,0,0,0,0,0,0},{0,3,4,0,0,5,0,6},{0,4,7,0,8,0,0,0},{0,0,0,9,0,0,2,0}
        ,{2,0,8,0,7,0,0,0},{0,5,0,0,0,8,0,9},{0,0,0,2,0,0,6,0},
        {0,6,0,0,0,9,0,1}
    };
    int res_hw[DIM][DIM];
    int res_sw[DIM][DIM];
    int dma_size = SIZE * sizeof(int);
    int run_time_sw,

```

```

    int run_time_hw = 0;
    init_platform();

xil_printf("*****\n");
    xil_printf("-----SPARSE MATRIX TO BAND MATRIX-----\n");
    status = init_dma();
    if (status != XST_SUCCESS) {
        print("\rDMA initialization error\n");
        return XST_FAILURE;
    }
    status = XTmrCtr_Initialize( & timer_dev,
XPAR_AXI_TIMER_0_DEVICE_ID);
    if (status != XST_SUCCESS) {
        sparse_to_band(A, res_sw);
        Xil_DCacheFlushRange((unsigned int) A, dma_size);
        Xil_DCacheFlushRange((unsigned int) res_hw, dma_size);
        status = XRcm_CfgInitialize( & XMatcon_dev, & XMatcon_config);
    }
    if (status != XST_SUCCESS) {
        xil_printf("Error\r\n");
        return XST_FAILURE;
    }
    XRcm_Start( & XMatcon_dev);
    print_mat_conversion_status();
    Xil_DCacheFlushRange((int) A, dma_size);
    Xil_DCacheFlushRange((int) res_hw, dma_size);
    XTmrCtr_Reset( & timer_dev, XPAR_AXI_TIMER_0_DEVICE_ID);
    status = XAxiDma_SimpleTransfer( & AxiDma, (int) A, dma_size,
XAXIDMA_DMA_TO_DEVICE);
    if (status != XST_SUCCESS) {
        xil_printf("Transfer from DMA to Vivado HLS error\n");
        return XST_FAILURE;
    }
    status = XAxiDma_SimpleTransfer( & AxiDma, (int) res_hw,
dma_size, XAXIDMA_DEVICE_TO_DMA);
    if (status != XST_SUCCESS) {
        xil_printf("Transfer from Vivado HLS to DMA error \n");
        return XST_FAILURE;
    }
    while (XAxiDma_Busy( & AxiDma, XAXIDMA_DEVICE_TO_DMA));
    print_mat_conversion_status();
    Xil_DCacheFlushRange((unsigned int) res_hw, dma_size);
    xil_printf("\r-----Result by HW-----\r\n");
    check_conversion_result(res_hw, res_sw, & err);
    if (err == 0)
        xil_printf("\r SW and HW results same\n\r");
    else
        xil_printf("\r%d results different\n\r", err);
    cleanup_platform();
    return err;
}

```

Header File:

```

//*****Matrix Conversion header file*****//

```

```
#ifndef SPARSE_TO_BAND_H
#define SPARSE_TO_BAND_H
#include "xrcm.h"
#define DIM 8
#define SIZE((DIM)*(DIM))
void print_accel_status(void);
void sparse_to_band(int A[DIM][DIM], int res[DIM][DIM]);
#endif
```

REFERENCES

- [1] L. O. Maftciu-Scai, “The Bandwidths of a Matrix. A Survey of Algorithms,” *Ann. West Univ. Timisoara - Math.*, vol. 52, no. 2, pp. 183–223, 2015, doi: 10.2478/awutm-2014-0019.
- [2] A. Fujii, R. Suda, and A. Nishida, “Parallel matrix distribution library for sparse matrix solvers,” in *Eighth International Conference on High-Performance Computing Asia-Pacific Region (HPCASIA’05)*, Jul. 2005, p. 7 pp. – 219, doi: 10.1109/HPCASIA.2005.68.
- [3] N. E. Gibbs, W. G. Poole, and P. K. Stockmeyer, “An Algorithm for Reducing the Bandwidth and Profile of a Sparse Matrix,” *SIAM J. Numer. Anal.*, vol. 13, no. 2, pp. 236–250, 1976.
- [4] E. Cuthill and J. McKee, “Reducing the Bandwidth of Sparse Symmetric Matrices,” in *Proceedings of the 1969 24th National Conference*, New York, NY, USA, 1969, pp. 157–172, doi: 10.1145/800195.805928.
- [5] S. Aslan, E. Oruklu, and J. Saniie, “Architectural design tool for low area band matrix LU factorization,” in *2011 IEEE International Conference on Electro/Information Technology*, May 2011, pp. 1–6, doi: 10.1109/EIT.2011.5978620.
- [6] I. S. Duff, J. K. Reid, and J. A. Scott, “The use of profile reduction algorithms with a frontal code,” *Int. J. Numer. Methods Eng.*, vol. 28, no. 11, pp. 2555–2568, Nov. 1989, doi: 10.1002/nme.1620281106.
- [7] J. G. Lewis, “Implementation of the Gibbs-Poole-Stockmeyer and Gibbs-King Algorithms,” *ACM Trans Math Softw*, vol. 8, no. 2, pp. 180–189, Jun. 1982, doi: 10.1145/355993.355998.
- [8] G. Feng, “An Improvement of the Gibbs-Poole-Stockmeyer Algorithm,” *J. Algorithms Comput. Technol.*, vol. 4, no. 3, pp. 325–333, Sep. 2010, doi: 10.1260/1748-3018.4.3.325.
- [9] W. Liu and A. Sherman, “Comparative Analysis of the Cuthill–McKee and the Reverse Cuthill–McKee Ordering Algorithms for Sparse Matrices,” *SIAM J. Numer. Anal.*, vol. 13, no. 2, pp. 198–213, Apr. 1976, doi: 10.1137/0713020.
- [10] S. Gonzaga de Oliveira and A. Abreu, “The use of the reverse Cuthill-McKee method with an alternative pseudo-peripheral vertice finder for profile optimization,” Feb. 2018, doi: 10.5540/03.2018.006.01.0441.

- [11] “Sparse Matrix Reordering Algorithms for Cluster Identification.”
https://www.researchgate.net/publication/240736442_Sparse_Matrix_Reordering_Algorithms_for_Cluster_Identification (accessed Sep. 18, 2019).
- [12] A. Azad, M. Jacquelin, A. Buluç, and E. G. Ng, “The Reverse Cuthill-McKee Algorithm in Distributed-Memory,” in 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS), May 2017, pp. 22–31, doi: 10.1109/IPDPS.2017.85.
- [13] V. Mani, B. Dattaguru, N. Balakrishnan, and T. S. Ramamurthy, “Parallel Gaussian elimination for a banded matrix-a computational model,” in 1990 IEEE Region 10 Conference on Computer and Communication Systems, 1990. IEEE TENCEN’90, Sep. 1990, pp. 170–174 vol.1, doi: 10.1109/TENCEN.1990.152591.
- [14] M. P. Véstias, R. P. Duarte, and H. C. Neto, “Designing Hardware/Software Systems for Embedded High-Performance Computing,” ArXiv150806832 Cs, Aug. 2015, Accessed: Mar. 18, 2019. [Online]. Available: <http://arxiv.org/abs/1508.06832>.
- [15] M. Garland, “Sparse matrix computations on manycore GPU #x2019;s,” in 2008 45th ACM/IEEE Design Automation Conference, Jun. 2008, pp. 2–6.
- [16] D. DuBois, A. DuBois, C. Connor, and S. Poole, “Sparse Matrix-Vector Multiplication on a Reconfigurable Supercomputer,” in 2008 16th International Symposium on Field-Programmable Custom Computing Machines, Apr. 2008, pp. 239–247, doi: 10.1109/FCCM.2008.53.
- [17] P. Stathis, D. Cheresiz, S. Vassiliadis, and B. Juurlink, “Sparse matrix transpose unit,” in 18th International Parallel and Distributed Processing Symposium, 2004. Proceedings., Apr. 2004, pp. 90–, doi: 10.1109/IPDPS.2004.1303033.
- [18] “Banded matrices.”
<http://www.math.chalmers.se/Math/Grundutb/CTH/tma881/0708/Assignments/MATLAB/banded.html> (accessed Mar. 04, 2018).
- [19] “Bandwidth.”
http://mae.uta.edu/~lawrence/me5310/course_materials/me5310_notes/4_Computational_Aspects/4-3_Matrix_Bandwidth/4-3_Matrix_Bandwidth.html.
- [20] “Cuthill–McKee algorithm - en.LinkFang.org.”
https://en.linkfang.org/wiki/Cuthill%E2%80%93McKee_algorithm.
- [21] A. George and J. W. Liu, Computer Solution of Large Sparse Positive Definite. Prentice-Hall Professional Technical Reference, 1981.

- [22] “Uninformed Search Algorithms - Javatpoint,” www.javatpoint.com.
<https://www.javatpoint.com/ai-uninformed-search-algorithms>.
- [23] “Sparse Matrices.”
http://bioinformatics.intec.ugent.be/MotifSuite/INCLUSive_for_users/CPU_64/Matlab_Compiler_Runtime/v79/toolbox/matlab/demos/html/sparsity.html.
- [24] “8.3 Pivoting To Preserve Sparsity.”
https://vismor.com/documents/network_analysis/matrix_algorithms/S8.SS3.php.
- [25] W. F. Tinney and J. W. Walker, “Direct solutions of sparse network equations by optimally ordered triangular factorization,” *Proc. IEEE*, vol. 55, no. 11, pp. 1801–1809, Nov. 1967, doi: 10.1109/PROC.1967.6011.
- [26] D. J. Rose, “Symmetric elimination on sparse positive definite systems and the potential flow network problem.,” 1970.
- [27] A. George and J. W. H. Liu, “The Evolution of the Minimum Degree Ordering Algorithm,” *SIAM Rev.*, vol. 31, no. 1, pp. 1–19, Mar. 1989, doi: 10.1137/1031001.
- [28] S. Irfansyah, “Design And Implementation of UART With FIFO Buffer Using VHDL On FPGA,” *ICTACT J. Microelectron.*, vol. 05, no. 01, p. 7, 2019.
- [29] “BaudRate (External Interfaces/API).” http://www.ece.northwestern.edu/local-apps/matlabhelp/techdoc/matlab_external/baudrate.html.
- [30] “ZedBoard [Digilent Documentation].”
<https://reference.digilentinc.com/reference/programmable-logic/zedboard/start>
- [31] “Optimization of Triangular and Banded Matrix Operations Using 2d-Packed Layouts | ACM Transactions on Architecture and Code Optimization.”
<https://dl.acm.org/doi/10.1145/3162016>.
- [32] “(PDF) Comparision of The FPGA Based Implementation And MATLAB/SIMULINK Simulation Torque Controller of A PMSM Machine Drive,” ResearchGate. <https://www.researchgate.net/publication/272796134>
- [33] “VivadoHLS_Overview.pdf.” Available:
http://users.ece.utexas.edu/~gerstl/ee382v_f14/soc/vivado_hls/VivadoHLS_Overview.pdf.
- [34] T. Nechma and M. Zwolinski, “Parallel Sparse Matrix Solution for Circuit Simulation on FPGAs,” *IEEE Trans. Comput.*, vol. 64, no. 4, pp. 1090–1103, Apr. 2015, doi: 10.1109/TC.2014.2308202.

- [35] C. Wang, J. Liu, W. Min, and A. Qu, “A Novel Sparse Penalty for Singular Value Decomposition,” *Chin. J. Electron.*, vol. 26, no. 2, pp. 306–312, 2017, doi: 10.1049/cje.2017.01.025.
- [36] K. Vasudevan, P. S. Rao, and K. S. Rao, “Simulation of power electronic circuits using sparse matrix techniques,” *IEEE Trans. Ind. Electron.*, vol. 42, no. 4, pp. 409–413, Aug. 1995, doi: 10.1109/41.402481.
- [37] Y. Han and Y. Igarashi, “Efficient Parallel Shortest Path Algorithms for Banded Matrices,” in *International Conference on Parallel Processing, 1993. ICPP 1993*, Aug. 1993, vol. 3, pp. 223–226, doi: 10.1109/ICPP.1993.73.
- [38] A. D. Yaghjian, “Banded-matrix preconditioning for electric-field integral equations,” in *IEEE Antennas and Propagation Society International Symposium 1997. Digest*, Jul. 1997, vol. 3, pp. 1806–1809 vol.3, doi: 10.1109/APS.1997.631610.
- [39] A. Azad, M. Jacquelin, A. Buluç, and E. G. Ng, “The Reverse Cuthill-McKee Algorithm in Distributed-Memory,” in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2017, pp. 22–31, doi: 10.1109/IPDPS.2017.85.
- [40] J.-C. Luo, “Algorithms for reducing the bandwidth and profile of a sparse matrix,” *Comput. Struct.*, vol. 44, no. 3, pp. 535–548, Jul. 1992, doi: 10.1016/0045-7949(92)90386-E.