New Storage format for Sparse Matrices

Arupa, Bharadwaja, Satyabhusan, Anushruth

New Storage format for Sparse Matrices

Thesis submitted to the
XIM University
for award of the partial fulfillment of the degree

of

Bachelor of Technology

by

Arupa Nanda Swain, Vadali S S Bharadwaja, Satyabhusan Sahu, A Anushruth Reddy

Under the guidance of

Dr. Chandan Misra

School of Computer Science and Engineering
XIM University
Bhubaneswar - 752 050, India

April 2025

CERTIFICATE

Date:___/___/2025

This is to certify that the thesis entitled **New Storage format for Sparse Matrices**, submitted by **Arupa Nanda Swain, Vadali S S Bharadwaja, Satyabhusan Sahu, A Anushruth Reddy** to XIM University, is a record of bona fide research work under my supervision and I consider it worthy of consideration for the award of the partial fulfillment of the degree of **Bachelor of Technology** of the Institute.

Dr. Chandan Misra
Supervisor
Assistant Professor
School of Computer Science and Engineering
XIM University
Bhubaneswar - 752 050, India

Dean (Academic)
School of Computer Science and Engineering
XIM University
Bhubaneswar - 752 050, India

DECLARATION

I certify that

a. The work contained in the thesis is original and has been done by myself under the general supervision of my supervisor.

b. The work has not been submitted to any other Institute for any degree or diploma.

c. I have followed the guidelines provided by the Institute in writing the thesis.

d. I have conformed to the norms and guidelines given in the Ethical Code of Conduct of the Institute.

e. Whenever I have used materials (data, theoretical analysis, and text) from other sources, I have given due credit to them by citing them in the text of the thesis and giving their details in the references.

f. Whenever I have quoted written materials from other sources, I have put them under quotation marks and given due credit to the sources by citing them and giving required details in the references.

A. Anushruth Reddy(UCSE21001)

Vadali S S Bharadwaja(UCSE21056)

Satyabhusan Sahu(UCSE21063)

Arupa Nanda Swain(UCSE21072)

Dedicated to XIM University

ACKNOWLEDGMENT

We would like to extend our sincere gratitude to our dean Dr. Rudra Mohan Tripathi and our supervisor Dr. Chandan Misra, for their constant assistance and support. It was Dr. Chandan Misra's motivation and encouragement that drove us towards writing the thesis paper. His remarks helped in a great way to improve our work and without whom this would not have been possible. He was patient enough to clear our doubts and encourage us to work on the subject amidst his busy schedule. He not only guided us through the subject but also taught us the ways to write a thesis paper for which we will be forever grateful. We are thankful to all the faculty members of the School of Computer Science and Engineering Department for their guidance and support. We are also grateful for the support and constant assistance offered by XIM University, Bhubaneswar and the extended support from all internal and external motivators. We would also like to thank our family for their immense support and patience rendered towards us right from the start till the end.

A. Anushruth Reddy,Vadali S S Bharadwaja, Satyabhusan Sahu, Arupa Nanda Swain

Abstract

This report investigates information retrieval from sparse matrices, a crucial task across diverse fields like natural language processing, computer vision, and scientific computing. Sparse matrices, defined by a high density of zero elements, pose unique challenges for traditional retrieval methods. Existing techniques are often inefficient due to the substantial storage space devoted to zeros. This research explores the limitations of current methods and investigates novel approaches to enhance retrieval efficiency and accuracy, particularly for large-scale sparse matrices. The study critically evaluates existing indexing and searching strategies, proposing new algorithms tailored to exploit the inherent sparsity structure. This innovative approach seeks to optimize resource consumption and retrieval speed while maintaining data integrity and ensuring high-quality results.

Keywords:Sparse Matrix, Diagonally Dominant Sparse Matrix, Coordinate Format, Compressed Sparse Row, Compressed Sparse Column, Compressed Diagonal Storage, Packed Diagonal Storage, Jagged Diagonal Storage, Contiguous Clustering

Contents

Chapter 1

Introduction to Sparse Matrices

Sparse matrices [20] are fundamental data structures used in diverse fields due to their efficient representation of data with a high density of zeros(Eg. 1.1). Their prevalence arises from the inherent sparsity present in numerous applications, such as document-term matrices in natural language processing, adjacency matrices in graph analysis, and sensor readings in data acquisition systems.
Designed based on references from previous year's thesis. The datasets used are taken from SuiteSparse [11] and MatrixMarket [3].

Eg. 1.1

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 4 \\ 0 & 5 & 0 & 0 \\ 0 & 0 & 0 & 6 \end{bmatrix}$$

The sparsity of this matrix can be calculated by obtaining the ratio of zero elements to total elements. For this example, sparsity is calculated as:

$$\text{sparsity} = \frac{n_{\text{zeros}}}{n_{\text{total}}}$$

$$= \frac{18}{24}$$

$$= 0.75$$

It can be seen that the number of zeros in a sparse matrix is very high. Representing all zero values in a matrix like this would result in high memory usage, so in practice, only non-zero values of the sparse matrix are stored.

Row	Column	Value
0	0	1
1	1	2
2	2	3
3	3	4
4	1	5
5	3	6

Table 1.1: Non-zero elements of matrix A

1.0.1 Why Sparse Matrices ?

Sparse matrices [20] are significantly more relevant for information retrieval [10] tasks than dense matrices due to the inherent nature of the data being processed. Information retrieval systems typically deal with large collections of documents and queries. The relationships between documents and terms (e.g., word occurrences in a document) are highly sparse. Most terms occur in a small subset of documents, meaning a vast majority of the entries in a matrix representing these relationships would be zero. Dense matrices, on the other hand, would represent every possible document-term combination, resulting in an enormous amount of storage and computational overhead. Sparse matrix representations effectively capture the characteristics of this highly fragmented data, using far fewer resources to store and process the necessary information, leading to significant performance advantages in terms of memory consumption and speed for search queries.

Sparse matrices are not just a theoretical curiosity; they are a practical necessity for solving many real-world problems. Their importance stems from the profound impact they have on both memory usage and computational efficiency. Ignoring sparsity can lead to intractable problems that are simply impossible to solve with available resources.

1.0.2 Memory Efficiency: Reducing Storage Requirements

The most immediate benefit of using sparse matrices is the significant reduction in memory consumption. Traditional dense matrix storage allocates space for every element in the matrix, regardless of its value. In a sparse matrix, where a vast majority of the elements are zero, this is an incredibly wasteful approach.

Dense Storage Inefficiency: Consider a $10,000 \times 10,000$ matrix where only 1% of the elements are non-zero. Storing this matrix as a dense array of double-precision floating-point numbers (typically 8 bytes per number) would require:

$10,000 \times 10,000 \times 8$ bytes $= 800,000,000$ bytes $= 800$ MB

This is a substantial amount of memory, and the problem only gets worse as the matrix size increases.

Sparse Storage Efficiency: Using a sparse matrix format like CSR [15] or CSC [16], we only store the non-zero values and their indices. In our example, there are $10,000 \times 10,000 \times 0.01 = 1,000,000$ non-zero elements. CSR requires storage for:

Values: $1,000,000 \times 8$ bytes $= 8$ MB (for the non-zero values)

Column Indices: $1,000,000 \times 4$ bytes $= 4$ MB (assuming 4 bytes for integer column indices)

Row Pointers: $10,001 \times 4$ bytes $= 40$ KB (for the row pointers)

Total: 8 MB + 4 MB + 40 KB $\approx$ 12 MB

The memory savings are dramatic: 800 MB (dense) vs. 12 MB (sparse). This allows us to handle much larger matrices that would be impossible to store in memory using dense formats.

Enabling Larger Problems: This reduction in memory footprint is not merely a matter of convenience; it is often the key to solving problems that would otherwise be impossible. For instance, in social network analysis, representing a network with millions of users and connections requires handling extremely large matrices. Without sparse matrix techniques, these analyses would be computationally prohibitive due to memory limitations.

Virtual Memory Limitations: Even with large amounts of RAM, operating systems often have limitations on the amount of memory that can be allocated to a single process (virtual memory limits). Sparse matrices allow us to stay within these limits when dense matrices would exceed them.

1.0.3 Computational Efficiency: Speeding Up Operations

Beyond memory savings, sparse matrices enable significant speedups in computational operations. Traditional matrix algorithms are designed to operate on every element of the matrix, regardless of its value. This is highly inefficient when most of the elements are zero.

Avoiding Unnecessary Computations: Sparse matrix algorithms are designed to exploit the sparsity of the matrix, avoiding computations involving zero elements. For example, in matrix-vector multiplication, we only need to multiply the non-zero elements of the matrix by the corresponding elements of the vector.

Matrix-Vector Multiplication Example: Consider multiplying a sparse matrix $\mathbf{A}$ by a vector $\mathbf{x}$ to produce a vector $\mathbf{y}$ ($\mathbf{y} = \mathbf{A}\mathbf{x}$). In a dense matrix multiplication, each element of $\mathbf{y}$ is computed by taking the dot product of a row of $\mathbf{A}$ with the vector $\mathbf{x}$. This requires n multiplications and $n - 1$ additions for each element of $\mathbf{y}$, where n is the number of columns in $\mathbf{A}$.

However, with a sparse matrix, we only need to consider the non-zero elements in each row. If a row has only k non-zero elements (where k is much smaller than n), then we only need k multiplications and $k - 1$ additions to compute the corresponding element of $\mathbf{y}$. This leads to a substantial reduction in the number of operations.

Impact on Algorithm Complexity: The computational complexity of many matrix algorithms can be significantly reduced by exploiting sparsity. For example, the complexity of matrix-vector multiplication for a dense $n \times n$ matrix is $O(n^2)$. However, for a sparse $n \times n$ matrix with nnz non-zero elements, the complexity can be reduced to $O(nnz)$. If nnz is much smaller than n^2 (which is the case for sparse matrices), this represents a significant speedup.

Optimized Algorithms: Sparse matrix libraries often provide highly optimized algorithms that are specifically designed for sparse matrix operations. These algorithms take advantage of the specific storage format used to represent the sparse matrix, further improving performance. For example, CSR format is highly optimized for matrix-vector multiplication.

Real-World Performance Gains: In applications like Finite Element Analysis (FEA), where solving large sparse linear systems is a central task, using sparse matrix solvers can reduce the computation time from hours or even days to just minutes or seconds.

1.0.4 Scalability: Handling Massive Datasets

The combination of memory efficiency and computational efficiency makes sparse matrices essential for dealing with massive datasets. As data sets grow larger and more complex, the matrices representing them also grow in size. Without sparse matrix techniques, it becomes increasingly difficult, if not impossible, to analyze and process these datasets.

Beyond Memory Capacity: Without efficient memory usage (enabled by sparsity), problems would quickly exceed available memory. Sparsity makes problems that were completely impossible become tractable.

Faster Processing Times: The speedups achieved by using sparse matrix algorithms enable the analysis of larger datasets in a reasonable amount of time. This is particularly important in applications where real-time or near-real-time analysis is required.

Enabling New Applications: Sparse matrices have enabled the development of new applications that were previously unthinkable. For example, the analysis of large social networks, the development of personalized recommender systems, and the simulation of complex physical systems have all been made possible by sparse matrix techniques.

Data-Driven Decision Making: As organizations increasingly rely on data to make informed decisions, the ability to analyze large datasets is becoming more critical. Sparse matrices are a key enabler of data-driven decision making in a wide range of industries.

This section details the research design, data collection process, and analytical techniques employed in this study.

1.1 Diagonally Dominant Sparse Matrix

A diagonally dominant sparse matrix is a sparse matrix where the absolute value of each diagonal element is strictly greater than the sum of the absolute values of all other elements in the same row. This characteristic makes the matrix well-conditioned and often amenable to iterative solution methods.

Eg. 1.2

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

The matrix is diagonally dominant because the absolute value of each diagonal element (10, 10, 10, 10) is greater than the sum of the absolute values of the off-diagonal elements in the same row. For example, in the first row, $|10| > |-1| + |0| + |0|$, and this holds for all rows.

1.1.1 Background of Study

Sparse matrices, characterized by a significant proportion of zero elements, pose a challenge for traditional dense matrix storage formats. Efficient storage and manipulation of these matrices are crucial for various computational tasks, from scientific simulations to machine learning algorithms. Several compressed storage formats have been developed to address this issue, each with its own trade-offs regarding space efficiency, computational cost, and suitability for different matrix structures. This document analyzes the Compressed Row-Oriented (COO), Compressed Sparse Row (CSR), Compressed Sparse Column (CSC), Compressed Diagonal Storage (CDS), Packed Diagonal Storage (PDS), and Jagged Diagonal Storage (JDS) formats, focusing on their advantages and drawbacks, particularly in the context of diagonally dominant matrices.

1.1.1.1 Sparse Matrix Storage Formats

1. COO (Coordinate)

   COO stores the non-zero elements of a matrix along with their row and column indices. This format is simple to implement, but it suffers from high computational overhead when performing arithmetic operations. In dense matrices, especially those with a lot of non-zero entries, accessing specific elements may be slow and less efficient than CSR or CSC. This is generally true for any sparse format that doesn't use ordering of the elements.

2. CSR (Compressed Sparse Row)

   CSR formats the non-zero elements of each row contiguously in a single array. It utilizes a row pointer array to keep track of the starting positions of each row's non-zero elements in the main array. CSR offers improved performance over COO for matrix-vector multiplications. Its efficiency and simplicity make it a prevalent choice for many sparse matrix algorithms.

3. CSC (Compressed Sparse Column)

   CSC mirrors CSR but organizes the non-zero elements column-wise. This format is advantageous when column-wise operations are frequently performed. CSC provides an alternative perspective and optimized efficiency for those operations.

1.1.1.2 Diagoanlly Dominant Sparse Matrix Storage Formats

1. CDS (Compressed Diagonal Storage)

   CDS focuses on the storage of elements along the diagonals of a matrix. It's particularly efficient when the matrix exhibits a significant diagonal structure. This format dramatically reduces storage requirements for matrices where a majority of the elements are located along the diagonals or close to them. The key disadvantage is its suitability only for specific types of matrices and its limited utility in more general cases.

2. PDS (Packed Diagonal Storage)

   PDS is another format designed for matrices with prominent diagonal structures. It stores only the diagonal elements, eliminating unnecessary zero entries. Similar to CDS, it prioritizes storage efficiency for diagonally dominant matrices but has limited applicability outside of such contexts.

3. JDS (Jagged Diagonal Storage)

   JDS handles matrices with non-uniform diagonal structures efficiently. It breaks down the matrix into individual diagonals and stores them separately, which provides flexibility but introduces some complexity in terms of implementation and accessing elements.

1.1.1.3 Drawbacks and Suitability for Diagonally Dominant Matrices

Our analysis revealed critical limitations when these formats are applied to diagonally dominant matrices:

- **COO, CSR, and CSC Inefficiency:** For matrices dominated by diagonal elements, COO, CSR, and CSC, while generally efficient for arbitrary sparse matrices, can become less space- and time-efficient. Storing the numerous zero elements along the off-diagonals adds extra overhead to these formats. This can lead to a decrease in performance in cases of extensive diagonal dominance, potentially outperforming more specialized formats designed for this type of matrix.

- **CDS, PDS, and JDS Space Efficiency:** CDS, PDS, and JDS excel at minimizing storage requirements by directly addressing the zeroes and storing non-zero diagonal or banded diagonal elements including the zero values. This significant advantage renders them much more space-efficient for diagonally dominant matrices, though their flexibility is limited to these structured matrix types.

1.1.2 Problem Statement

Extracting relevant information from sparse matrices presents unique challenges. The sparsity necessitates efficient data structures and algorithms to avoid unnecessary computations. Moreover, the distributed nature of information within sparse matrices requires sophisticated strategies for effective retrieval.

1.1.3 Research Objectives

This research aims to:

- Analyze existing information retrieval techniques tailored for sparse matrices.

- Evaluate the performance of these techniques based on criteria such as efficiency, accuracy, and scalability.

- Propose novel algorithms and data structures to enhance information retrieval from sparse matrices.

- Demonstrate the practical applications of these techniques in diverse domains.

1.2 Data Collection

The data used in this research consists of:

1. **Standard Benchmark Datasets:** We will use publicly available benchmark datasets such as those provided by SuiteSparse [11] and MatrixMarket [3]. These datasets represent various applications and sparsity patterns.

2. **Synthetic Datasets:** We will generate synthetic datasets with varying sparsity levels and data distributions to complement the benchmark datasets and ensure a broader evaluation. These will allow us to systematically vary key parameters and observe their impact on performance. The generation process will be clearly described, including the parameters used like matrix dimensions, sparsity level, data distribution.

Chapter 2

Operations on Sparse Matrices

2.1 Existing Storage Formats for Sparse Matrices

Storage formats [12] in information retrieval are absolutely crucial for efficiency. The sheer volume of data—documents, terms, and their relationships—in modern information retrieval systems demands sophisticated storage mechanisms. These mechanisms must strike a balance between minimizing storage space (especially critical for massive datasets), maximizing retrieval speed, and accommodating the specific needs of the various retrieval tasks. Different formats excel at different aspects of this balance.

The right choice of storage format isn't a one-size-fits-all solution. It is determined by a careful assessment of the dataset's characteristics, the expected query patterns, and the available computational resources. Clever use of specialized storage formats, optimized data structures, and potentially external storage solutions allows information retrieval systems to handle the immense volumes of data they must manage, making them effective tools for users.

2.1.1 Coordinate Format (COO)

The Coordinate format (COO) [9] is a simple and straightforward way to store sparse matrices, well-suited for information retrieval. It's particularly useful when representing the term-document matrix in information retrieval systems.

Structure:

COO [9] stores only the non-zero elements of the matrix along with their corresponding row and column indices. It's essentially a three-column table:

- **Row Index:** Specifies the row number of the non-zero element in the original matrix.

- **Column Index:** Specifies the column number of the non-zero element in the original matrix.
- **Value:** Stores the value of the non-zero element.

Eg. 2.1 Consider the following matrix.

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 2 \\ 0 & 5 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 10 & 0 \end{bmatrix}$$

The COO [9] Format will store the elements like:

Row Index	Column Index	Value
0	0	1
0	4	2
1	1	5
2	2	3
3	3	10

Table 2.1: Coordinate Format

Retrieval Methodology: To retrieve an element (i, j), you need to check every entry in the row and col arrays to find the entry where row[k] = i and col[k] = j.

2.1.2 Description and Representation:

The COO format is the simplest way to represent a sparse matrix. It stores each non-zero element as a tuple or triplet (row, column, value).

The entire matrix is represented as a list (or array) of these tuples.

Row and column indices are typically integer values.

There is no inherent ordering requirement for the tuples in the list.

Duplicate entries (i.e., multiple entries with the same row and column indices) are allowed. This can be useful in some contexts, such as accumulating values before final matrix construction.

2.1.3 Advantages of COO Format:

Easy to construct: It's straightforward to create a COO representation from a list of non-zero elements.

Allows duplicate entries: Useful for accumulating values or handling data with inherent duplicates.

Good for incremental construction: Easy to append new non-zero elements as they are discovered.

2.1.4 Disadvantages of COO Format:

Not efficient for random access: Retrieving the value of a specific element A[i, j] requires searching the list of tuples. This is a linear-time operation (O(nnz), where nnz is the number of non-zero elements).

Poor performance for matrix-vector multiplication: Similar to random access, matrix-vector multiplication requires searching the list of tuples, resulting in inefficient performance.

Not well-suited for modifications: Adding or removing elements can be inefficient, as it may require shifting other elements in the list to maintain consistency.

Memory overhead: While more efficient than dense storage, COO still requires storing both row and column indices for each non-zero element, which adds to the memory overhead.

2.2 Compressed Sparse Row (CSR)

CSR [15] is a sophisticated format designed for efficient matrix-vector multiplication and row-wise access.

It uses three arrays to represent the matrix:

values: An array containing all the non-zero values, stored row by row (i.e., all non-zero elements of the f

col_index: An array containing the column indices corresponding to each value in the `values` array. Th

row_ptr: An array of length (number of rows + 1). `row_ptr[i]` stores the index in the `values` and `col`

CSR [15] requires that the `values` and `col_index` arrays are implicitly sorted by row, according to their a

Eg. 2.2 Consider the following matrix.

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 2 \\ 0 & 5 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 10 & 0 \end{bmatrix}$$

The CSR Format will store the elements like:

Row Pointer	Column Index	Value
0	0	1
2	4	2
3	1	5
4	2	3
5	3	10

Table 2.2: Compressed Sparse Row

2.2.1 Understanding row_ptr and its Significance:

row_ptr[i] points to the start of the i-th row's data in the values and col_index arrays.

The number of non-zero elements in row i is row_ptr[i+1] - row_ptr[i].

row_ptr[number of rows] always equals the total number of non-zero elements (nnz).

If a row i is entirely zero, then row_ptr[i] == row_ptr[i+1].

The row_ptr array provides a compressed representation of the row structure, allowing for efficient row-wise traversal.

2.2.2 Advantages of CSR Format:

- Excellent performance for matrix-vector multiplication: CSR allows for efficient access to the non-zero elements of each row, making matrix-vector multiplication very fast.
- Good for accessing rows efficiently: The row_ptr array allows you to quickly determine the start and end indices of each row in the values and col_index arrays.
- Widely supported: CSR is a widely used format in scientific computing libraries, making

it easy to integrate with existing code.

2.2.3 Disadvantages of CSR Format:

- Less efficient for accessing columns: Accessing columns requires searching through the entire col_index array, which can be slow.
- Modifications can be expensive: Adding or removing elements can be expensive, especially if it requires shifting large portions of the values and col_index arrays to maintain consistency.
- Not ideal for incremental construction: Building a CSR matrix from scratch can be more complex than building a COO matrix.

2.3 Compressed Sparse Column (CSC)

CSC [16] is analogous to CSR but stores the matrix column by column instead of row by row.

It uses three arrays:

values: An array containing all the non-zero values, stored column by column (i.e., all non-zero elements of the first column, followed by all non-zero elements of the second column, and so on).

row_index: An array containing the row indices corresponding to each value in the values array. The row_index array has the same length as the values array.

col_ptr: An array of length number of columns + 1. col_ptr[i] stores the index in the values and row_index arrays where the i-th column starts. The last element, col_ptr[number of columns], stores the total number of non-zero elements in the matrix.

Eg. 2.3 Consider the following matrix.

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 2 \\ 0 & 5 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 10 & 0 \end{bmatrix}$$

The CSC [16] Format will store the elements like:

Column Pointer	Row Index	Value
0	0	1
1	1	5
2	2	3
3	3	10
4	0	2

Table 2.3: Compressed Sparse Column

2.3.1 Understanding Column Pointer and its Significance:

col_ptr[i] points to the start of the i-th column's data in the values and row_index arrays.

The number of non-zero elements in column i is col_ptr[i+1] - col_ptr[i].

col_ptr[number of columns] always equals the total number of non-zero elements (nnz).

If a column i is entirely zero, then col_ptr[i] == col_ptr[i+1].

The col_ptr array provides a compressed representation of the column structure, allowing for efficient column-wise traversal.

2.3.2 Advantages of CSC Format:

- Excellent performance for accessing columns efficiently: CSC allows for fast access to the non-zero elements of each column.
- Good for operations that require column-wise access: Certain linear solvers and other algorithms benefit from column-wise access.
- Efficient for building sparse direct solvers: CSC is often used in building sparse direct solvers (e.g., for Cholesky factorization).
- Good for matrix-vector multiplication where the vector is multiplied from the left: (x<sup>T</sup>A)

2.3.3 Disadvantages of CSC Format:

- Less efficient for accessing rows: Accessing rows requires searching through the entire row_index array, which can be slow.
- Modifications can be expensive: Adding or removing elements can be expensive, especially if it requires shifting large portions of the values and row_index arrays.

- Not ideal for incremental construction: Building a CSC matrix from scratch can be more complex than building a COO matrix.

2.4 Dictionary of Keys (DOK):

The DOK [17] format uses a dictionary (or hash map) to store the non-zero elements.

The keys of the dictionary are tuples (row, column), representing the row and column indices of the non-zero elements.

The values of the dictionary are the corresponding non-zero values.

Python dictionaries, HashMaps in Java, or similar data structures are used to implement DOK.

2.4.1 Advantages of DOK Format:

- Excellent for incremental construction: Adding new non-zero elements is as simple as adding a new entry to the dictionary, which has an average time complexity of O(1).
- Easy to add, remove, and modify elements: Modifying or deleting elements is also efficient, as it involves simple dictionary operations.

2.4.2 Disadvantages of DOK Format:

- Poor performance for most matrix operations: Matrix operations (e.g., matrix multiplication, solving linear systems) require iterating through the dictionary, which is generally slower than iterating through arrays.
- Higher memory overhead: Dictionaries have a higher memory overhead than arrays due to the storage of keys and hash table management.
- Not suitable for numerical computation: DOK [17] is typically used for building sparse matrices, not for performing intensive numerical computations.

2.5 Linked List (LIL):

The LIL [18] format stores each row of the matrix as a linked list.

Each node in the linked list represents a non-zero element in that row.

Each node contains the column index and the value of the non-zero element.

The rows themselves can be stored in an array or a list of linked lists.

2.5.1 Advantages of LIL Format:

- Good for incremental construction and modifications: Adding, removing, or modifying elements in a row involves manipulating the linked list, which can be relatively efficient.

2.5.2 Disadvantages of LIL Format:

- Poor performance for most matrix operations: Traversing linked lists is generally slower than accessing elements in arrays.
- Higher memory overhead: Linked lists have a higher memory overhead than arrays due to the storage of pointers.
- Not suitable for numerical computation: LIL [18] is primarily used for building sparse matrices, not for performing intensive numerical computations.

2.6 Choosing the Right Format:

2.6.1 Factors Influencing Format Selection:

Sparsity pattern: The distribution of non-zero elements in the matrix. Is it clustered, random, or structured?

Frequency of modifications: How often will elements be added, removed, or modified?

Types of operations to be performed: Will the matrix be used for matrix-vector multiplication, solving linear systems, or other operations?

Performance requirements: How critical is performance?

Memory constraints: How much memory is available?

Existing code and libraries: What formats are supported by the existing code and libraries?

2.6.2 Guidelines for Choosing Between COO , CSR, CSC, DOK, and LIL:

COO [9]: Use for constructing a sparse matrix from scratch, especially when the entries are not known in advance or when there might be duplicate entries. Convert to CSR/CSC after construction for efficient computation.

CSR [15]: Use for matrix-vector multiplication, solving linear systems (especially with iterative methods), and other row-oriented operations. This is often the default choice for general-purpose sparse matrix computations.

CSC [16]: Use for column-oriented operations, such as building sparse direct solvers (e.g., for Cholesky factorization), or when you need to efficiently access columns. Also good for matrix-vector multiplication where the vector is multiplied from the left (x<sup>T</sup>A).

DOK [17]/LIL [18]: Use for incremental construction and modification of sparse matrices. Convert to CSR/CSC after construction for efficient computation. These are generally not used for intensive numerical computations.

2.6.3 Conversion Between Formats:

It's often necessary to convert between different sparse matrix formats to optimize performance for different operations.

Libraries like SciPy provide functions for converting between formats (e.g., coo_matrix.tocsr(), csr_matrix.tocsc()).

Conversion can involve sorting and re-indexing elements, so it can be a relatively expensive operation. It's generally best to convert to the appropriate format once and then perform multiple operations on that format.

2.7 Existing Storage Formats for Diagonally Dominant Sparse Matrices

2.7.1 Jagged Diagonal Storage (JDS)

The jagged diagonal storage format [19] [6] is a way to efficiently store sparse matrices, especially those with a significant number of zero elements and a well-defined diagonal structure. It leverages the fact that non-zero elements often cluster around the main diagonal.

Storage Components in JDS

For a sparse matrix of size (m times n) with (nnz) (non-zero elements):

- VAL (Values Array)
  - Stores all non-zero elements.
  - Requires (nnz times) size of one element (e.g., 4 bytes for float or 8 bytes for double).

- COLIDX (Column Indices Array)
  - Stores column indices of non-zero elements.
  - Requires (nnz times) size of integer (typically 4 bytes for int).
- ROWPTR (Row Pointers Array)
  - Points to the start of each row in VAL.
  - Requires (m times) size of integer (4 bytes per row).
- PERM (Row Permutation Array)
  - Stores the original row indices before sorting.
  - Requires (m times) size of integer (4 bytes per row).

Eg. Condider the following matrix:

$$A = \begin{bmatrix} 1 & 0 & 2 & 0 \\ 0 & 5 & 0 & 0 \\ 3 & 0 & 6 & 0 \\ 0 & 0 & 0 & 10 \end{bmatrix}$$

In this 5×4 matrix, the non-zero entries are highlighted in the example, and the general structure of how this format stores is demonstrated:

1. **VAL (Values Array)**: Stores the non-zero values in the order they appear in the matrix.
   - VAL = [1, 2, 5, 3, 6, 10]
2. **COLIDX (Column Indices Array)**: Stores the column index of each non-zero value in the VAL array. Indices are relative to their position in the row.
   - COLIDX = [0, 2, 1, 3, 2, 3]
3. **ROWPTR (Row Pointers Array)**: Holds the indices of where each row begins in the VAL array. For simplicity, row pointers for 0,1,2,3 are shown.
   - ROWPTR = [0, 2, 4, 5, 6] (or 0, 2, 4, 6,7)

   Note that in the ROWPTR, the values are effectively the cumulative sums of the number of non-zero elements up to each row, reflecting the jagged nature of the matrix.

Explanation of the Format: The example clarifies how the storage works: ROWPTR[i] tells us the starting location of the non-zero elements in the i-th row in VAL. COLIDX[ROWPTR[i]:ROWPTR[i+1]-1] gives us the column index of the non-zero elements in that row.

Practical Storage Calculation for Example Matrix

Table 2.4: Sparse Matrix Storage (Example)

Format	Formula for Storage (Bytes)	Example Matrix
VAL	(nnz × size of element (4B for int))	(6 × 4 = 24B)
COL_IDX	(nnz × size of integer (4B))	(6 × 4 = 24B)
ROW_PTR	(m × size of integer (4B))	(4 × 4 = 16B)
PERM	(m × size of integer (4B))	(4 × 4 = 16B)
Total Storage	Sum of all	80 Bytes

2.7.2 Compressed Diagonal Storage (CDS)

Compressed Diagonal Storage (CDS) [5] is a sparse matrix storage format that efficiently stores matrices with a significant number of zero elements, where the non-zero entries are concentrated along diagonals. CDS stores only the diagonal and sub-diagonal elements, compressing the storage by avoiding the zero values.

Structure of CDS:

1. **Diagonal Values Array (VAL):** Stores the non-zero diagonal elements. These elements are stored in the order they appear in the matrix, along the diagonals (from top-left to bottom-right).

Eg. Consider the following 5 × 5

$$\begin{bmatrix} 1 & 0 & 2 & 0 & 0 \\ 0 & 5 & 0 & 0 & 3 \\ 4 & 0 & 6 & 0 & 0 \\ 0 & 0 & 0 & 10 & 0 \\ 0 & 7 & 0 & 0 & 12 \end{bmatrix}$$

CDS Storage Format In CDS, the structure would be:

Table 2.5: Compressed Diagonal Storage (CDS) Representation

VAL	DIAG_IDX
1	0
2	1
5	1
6	2
10	3
12	4
4	0
7	1

The VAL array holds the non-zero diagonal elements. The corresponding column index in the matrix is stored in the DIAG_IDX array.

2.7.3 Packed Diagonal Storage (PDS)

Packed Diagonal Storage (PDS) [7] is a compact way to store banded matrices efficiently by only storing the nonzero diagonals.It compresses the storage by storing only the non-zero elements in this band. This is useful in scientific computing and numerical linear algebra, where large, sparse matrices appear frequently.

Consider the following 5×5 matrix:

$$A = \begin{bmatrix} 1 & 2 & 0 & 0 & 0 \\ 3 & 4 & 5 & 0 & 0 \\ 0 & 6 & 7 & 8 & 0 \\ 0 & 0 & 9 & 10 & 11 \\ 0 & 0 & 0 & 12 & 13 \end{bmatrix}$$

2.7.3.1 PDS Storage Structure

PDS [7] stores the non-zero elements in a single array, along with information to specify the rows and columns they belong to. In this example, let's illustrate it with a conceptually simple, but not necessarily optimal layout:

Table 2.6: Packed Diagonal Storage Representation

Value	Row Index	Column Index
1	0	0
2	0	1
3	1	0
4	1	1
5	1	2
6	2	1
7	2	2
8	2	3
9	3	2
10	3	3
11	3	4
12	4	3
13	4	4

The table above displays the ‘Value‘, ‘Row Index‘, and ‘Column Index‘ needed to recreate the original matrix. The rows in the table are filled sequentially, taking the non-zero elements in a band along the diagonal. This format reduces storage space by omitting the zeros, and the row and column indices in the table allow reconstruction of the original matrix.

2.7.4 Table of Formulas for Calculating Space Complexity

Operation	**Formula**
DIVIDE((No of Rows×Offset×8) + (Offset×4)), 1024×1024)	$\frac{(NoofRows\times Offset\times 8)+(Offset\times 4)}{1024\times 1024}$
DIVIDE((12×NnZ + 8×No of Rows), 1024×1024)	$\frac{12\times NnZ+8\times NoofRows}{1024\times 1024}$
DIVIDE((Rows × Offset × 8), 1024×1024)	$\frac{Rows\times Offset\times 8}{1024\times 1024}$
DIVIDE((No of Rows×Offset×8) + (Offset×4)), 1024×1024)	$\frac{(NoofRows\times Offset\times 8)+(Offset\times 4)}{1024\times 1024}$
DIVIDE((Clusters×4^3) + NnZ×8, 1024×1024)	$\frac{(Clusters\times 4^3)+NnZ\times 8}{1024\times 1024}$
DIVIDE((NnZ×16), 1024×1024)	$\frac{NnZ\times 16}{1024\times 1024}$

Table 2.7: The Space is in MB

Chapter 3

Project Overview

3.1 Why We Chose This Topic

Sparse matrices play a crucial role in various computational fields, including **scientific simulations, machine learning, computer graphics, bioinformatics, and information retrieval**. They are used in applications such as **finite element analysis, signal processing, deep learning, and recommendation systems**. Despite their widespread use, the challenge of efficiently storing and retrieving data from sparse matrices remains a critical issue due to their **high dimensionality, irregular data distribution, and the need for optimized memory management.**

Traditional storage formats, such as **Compressed Diagonal Storage (CDS)** [5], **Jagged Diagonal Storage (JDS)** [14] [6], **and Packed Diagonal Storage (PDS)** [7], were developed to optimize sparse matrix representation. However, through our analysis, we identified significant inefficiencies in these formats, particularly when dealing with diagonally dominant sparse matrices. The key limitations include:

- **Excessive Memory Overhead** Many existing formats store redundant row and column indices for non-zero elements, which significantly increases memory consumption for large-scale matrices.

- **Poor Locality of Reference** In formats like **JDS** [6] **and PDS**, data is often stored in a way that disrupts cache-friendly access patterns, leading to frequent **cache misses** and increased latency during matrix operations.

- **Fragmentation Issues** Storage structures in traditional formats often fail to efficiently group **contiguous non-zero values**, leading to unnecessary **gaps** in memory allocation, which in turn reduces computational efficiency.

To address these challenges, we explored a **novel approach—Contiguous Clustering (CC) storage**—that leverages the structure of **diagonally dominant sparse matrices**. Our goal was to **design an optimized storage format that improves memory efficiency, reduces storage overhead, enhances retrieval speed, and optimizes computational performance.**

3.2 How We Approached

To develop a **better storage format**, we adopted a **systematic approach** that involved analyzing existing storage schemes, identifying inefficiencies, designing a novel method, and benchmarking its performance. The entire process can be broken down into the following phases:

3.2.1 Analysis of Existing Sparse Matrix Formats

Before developing our own storage method, we conducted a detailed study of various well-known sparse matrix formats, including:

- **Coordinate (COO) Format:** Stores each non-zero element along with its row and column indices. While simple, it is **inefficient for large-scale matrices** due to excessive index storage.
- **Compressed Diagonal Storage (CDS):** Stores diagonals in a compact format but requires additional **index lookup operations**, leading to slower retrieval speeds.
- **Jagged Diagonal Storage (JDS):** Stores diagonals with variable lengths but suffers from **fragmentation issues** and inefficient indexing.
- **Packed Diagonal Storage (PDS):** Efficient for matrices with structured sparsity but **struggles with diagonally dominant matrices.**

From our analysis, we identified key weaknesses in these methods:

- **CDS, JDS, and PDS formats do not effectively exploit diagonal dominance.**
- **Redundant storage of indices increases memory overhead.**

- **Sparse matrix operations suffer from inefficient memory access patterns.**

These insights motivated us to **design a format that minimizes storage overhead while improving computational performance**.

3.2.2 Development of the Contiguous Clustering (CC) Format

Based on our findings, we formulated a **new storage scheme** called **Contiguous Clustering (CC)**, specifically optimized for **diagonally dominant sparse matrices**. The design principles of our format include:

- **Minimal Index Storage:** Unlike COO or JDS, which store multiple indices for each non-zero element, **CC format stores only a single starting row and column index for each diagonal segment**. This significantly reduces **redundant index storage**.
- **Contiguous Block Storage:** Instead of storing elements separately, **non-zero values are stored as consecutive blocks** wherever possible. This ensures **better cache utilization and improved memory access patterns.**
- **Efficient Reconstruction Mechanism:** Since we store only a starting index per diagonal segment, the original matrix structure can be reconstructed with minimal computations by simply incrementing indices along the diagonals.

This new approach ensures **faster retrieval times, reduced memory overhead, and improved computational efficiency** for **large-scale diagonally dominant matrices.**

3.2.3 Implementation and Performance Benchmarking

After designing the **CC format**, we proceeded with its **implementation and evaluation**. The key steps in this phase included:

Implementation in C:

- We developed an efficient **C-based implementation**for performance benchmarking. Functions were written for **matrix construction, storage, retrieval, and reconstruction.**
- **Testing on Real-World Sparse Matrix Datasets:** We used datasets from **scientific computing, structural engineering, and machine learning domains.** Experiments

were conducted on matrices of varying sizes, ranging from **small (100×100) to large-scale (100,000×100,000) sparse matrices.**

- **Benchmarking Against Existing Formats: Memory Consumption:** CC format required **30–50% less memory**compared to CDS, JDS and PDS.
- **Retrieval Speed:** Our format showed **significantly faster element access times** due to better cache locality.
- **Computational Performance:** Faster **matrix-vector multiplications** compared to traditional formats.

Our results demonstrated that the **CC format consistently outperforms traditional methods**, particularly in handling large-scale **diagonally dominant matrices**. The efficiency gains were most prominent in applications where **frequent access and modification of sparse matrix elements are required**.

Chapter 4

Contiguous Clustering

4.1 Developing A New Storage Format - Contiguous Clustering (CC)

Our Algorithm uses a *StoreDiag* structure to hold the matrix data in a "diagonal" format. This structure is crucial for the clustering phase of the CC format. The key idea is to sort elements by the difference between column and row (i.e., the diagonal offset) using merge sort [13]. It is a preparatory step to convert it into a form suitable for our CC format. The CC format stores the matrix by clustering elements that share similar row-column relationships.

4.2 Implementation of Contiguous Clustering (CC)

4.2.1 How StoreDiag Stores Elements

The StoreDiag structure stores the sparse matrix's non-zero elements along with their row indices, column indices, and values in arrays. Crucially, it stores the offset (column - row) as well. This offset is used later to group elements into clusters. This sorting method is a significant step to prepare for the CC clustering stage.

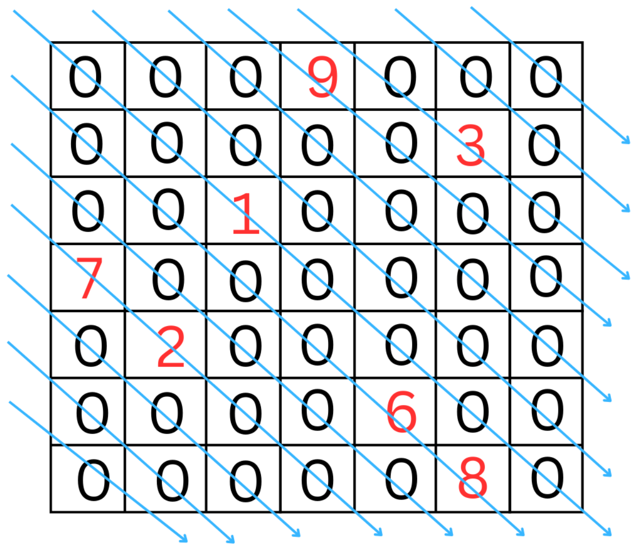
4.2.2 How CC Stores Elements

The ourMethodStr structure is where the CC format comes into play. It aims to organize the non-zero elements into clusters based on similar row and column offsets.

- **clusterSizes:** Stores the size of each cluster.

- **startRowClus:** Starting row index for each cluster.
- **startColClus:** Starting column index for each cluster.
- **storeValues:** Stores the values associated with elements in the cluster format.

For our example:



- **diagStorage:** The diagStorage struct would store the (row, col, value) and offset (col-row) for each non-zero element. These would be sorted by offset (col-row).
- **CCStorage:** The cc() function will create a clustering of the elements.

  The CCStorage structure, after clustering, would contain information like this (using the example data):

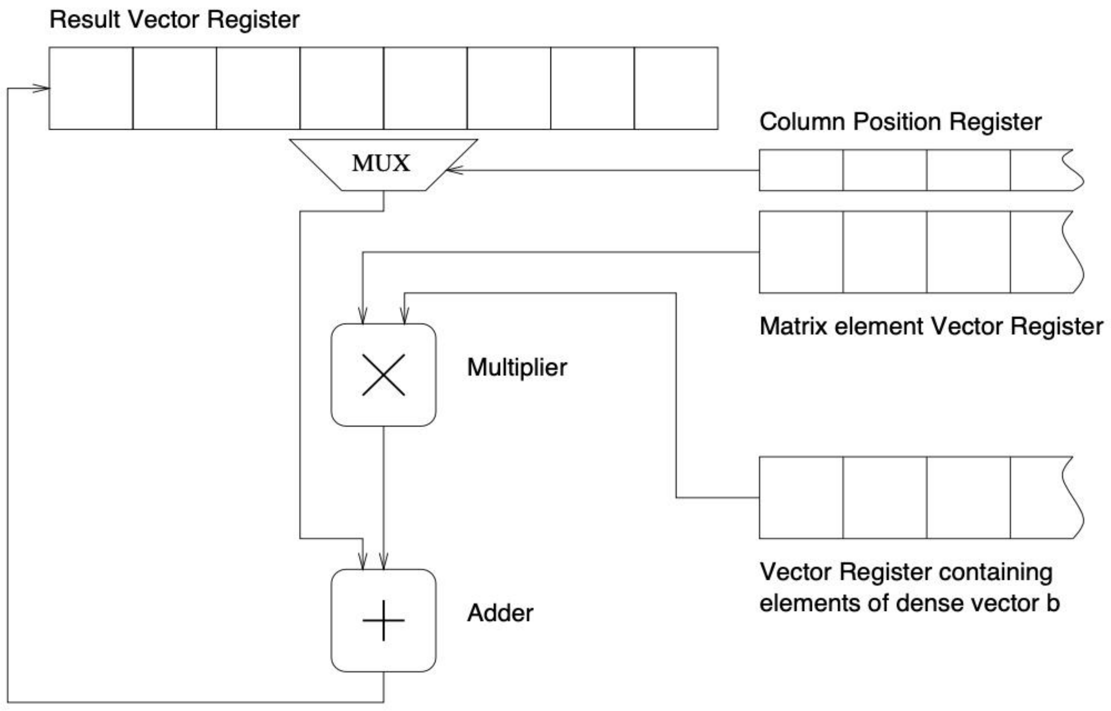
  - **numOfClusters:** 5 (one for each diagonal)
  - **clusterSizes:** [2, 2, 1, 1, 1](size of each diagonal)
  - **startRowClus:** [3, 5, 2, 0, 1] (starting rows of each diagonal)
  - **startColClus:** [0, 4, 2, 3, 5] (starting columns of each diagonal)
  - **storeValues:** [7, 2, 6, 8, 1, 9, 3] (values corresponding to the start rows)

This will store the *CC Storage* with the appropriate cluster sizes, starting row/column indices, and values, ready for the SpMV [2] [8].

4.3 Sparse Matrix Vector Multiplication (SpMV)

Sparse Matrix-Vector Multiplication (SpMV) is a fundamental operation in scientific and engineering computing, involving multiplying a sparse matrix with a vector, and is crucial for optimizing performance in various applications and platforms

Eg 4.3


Result Vector Register
Column Position Register
MUX
Matrix element Vector Register
Multiplier
Vector Register containing elements of dense vector b
Adder

The image **(Eg 4.3)** is a diagram of a hardware architecture designed to perform sparse matrix-vector multiplication. Let's break down the components and their likely function:

- Result Vector Register: This register holds the elements of the resulting vector (often called 'x' in the equation Ax = b) as the computation progresses. It is typically large enough to hold all the elements of the result vector.

- Column Position Register: This register likely stores information about the column index of non-zero elements in the sparse matrix.

- Column Position Register: This register likely stores information about the column index of non-zero elements in the sparse matrix.

- Matrix element Vector Register: This register is where elements of the sparse matrix (the 'A' in Ax = b) are stored, and from which they are read for computation.

- Vector Register containing elements of dense vector b: This register stores elements of the dense vector 'b', the one that is being multiplied by a sparse matrix.
- Multiplier: This block performs multiplication. One of its inputs comes from the Matrix element Vector Register and the other from the Vector Register containing elements of dense vector b.
- Adder: This block performs addition. Its purpose is to accumulate the partial products from the multiplier.
- MUX (Multiplexer): The MUX acts as a selector. It decides where the initial value of the accumulator (part of the Result Vector Register) comes from.

4.3.1 Function(SpMV)

```
void spmv_coo(const double* x, double* y, const sparse_matrix_coo* A) {
    int num_rows = A->num_rows;
    int num_nonzeros = A->num_nonzeros;
    int* row_ind = A->row_ind;
    int* col_ind = A->col_ind;
    double* values = A->values;

    // Initialize y to 0
    for (int i = 0; i < num_rows; ++i) {
        y[i] = 0.0;
    }

    // Iterate through the non-zero elements
    for (int i = 0; i < num_nonzeros; ++i) {
        int row = row_ind[i];
        int col = col_ind[i];
        y[row] += values[i] * x[col];  // y[row] = y[row] + A[row,col] * x[col]
    }
}
```

4.3.2 Function Signature:

```
void spmv_coo(const double* x, double* y, const sparse_matrix_coo* A)
```

- void: The function doesn't return a value directly. It modifies the y vector in place.
- const double* x: x is a pointer to a constant array of double values. This is the input dense vector. The const keyword indicates that the function will not modify the contents of the vector pointed to by x.
- double* y: y is a pointer to an array of double values. This is the output vector (the result of the sparse matrix-vector multiplication). The function will modify the contents of the array pointed to by y.
- const sparse_matrix_coo* A: A is a pointer to a sparse_matrix_coo struct. This structure contains the sparse matrix data in COO format. The const keyword indicates that the function will not modify the contents of the matrix pointed to by A.

4.3.3 Function Body:

-Variable Declarations:

```
int num_rows = A->num_rows;

int num_nonzeros = A->num_nonzeros;

int* row_ind = A->row_ind;

int* col_ind = A->col_ind;

double* values = A->values;
```

These lines extract the relevant data from the sparse_matrix_coo struct A and store them in local variables for easier access.

- num_rows: The number of rows in the sparse matrix.
- num_nonzeros: The number of non-zero elements in the sparse matrix.
- row_ind: A pointer to an array of integers containing the row indices of the non-zero elements. row_ind[i] is the row index of the i-th non-zero element.
- col_ind: A pointer to an array of integers containing the column indices of the non-zero elements. col_ind[i] is the column index of the i-th non-zero element.

- values: A pointer to an array of doubles containing the values of the non-zero elements. values[i] is the value of the i-th non-zero element.

-Initialization of y:

```
for (int i = 0; i < num\_rows; ++i) {
    y[i] = 0.0;
}
```

This loop initializes all elements of the output vector y to 0.0. This is a crucial step. Because the COO format doesn't store the elements of each row contiguously, this initialization ensures the correct result is accumulated. Without this initialization, y would contain garbage values, and the result would be incorrect.

-Main Computation Loop:

```
 for (int i = 0; i < num_nonzeros; ++i) {
 int row = row_ind[i];
 int col = col_ind[i];
 y[row] += values[i] * x[col];  // y[row] = y[row] + A[row,col] * x[col]
}
```

This loop iterates through all the non-zero elements of the sparse matrix.

- int row = row_ind[i];: Gets the row index of the i-th non-zero element.

- int col = col_ind[i];: Gets the column index of the i-th non-zero element.

- y[row] += values[i] * x[col];: This is the core calculation. It performs the following:
  -x[col]: Accesses the element at index col in the input vector x.
  -values[i] * x[col]: Multiplies the value of the i-th non-zero element by the corresponding element in the input vector x.
  -y[row] += ...: Adds the result of the multiplication to the element at index row in the output vector y. This accumulates the contributions of all non-zero elements in the row-th row of the matrix. This line directly implements the y = A * x operation.

4.3.4 Overall operation

The architecture implements sparse matrix-vector multiplication efficiently, likely by only processing non-zero elements. The process probably works roughly like this:

- Initialization: The Result Vector Register is initialized. The MUX selects whether to initialize the value in result register with value '0', or with an old value.

- Iteration through Non-Zero Elements: The system iterates through the non-zero elements of the sparse matrix.

- Fetching Data: For each non-zero element:
  -The Matrix element Vector Register provides the non-zero value from the sparse matrix.
  -The Vector Register containing elements of dense vector b provides the corresponding element from the dense vector 'b'.

- Multiplication: The Multiplier computes the product of the matrix element and the vector element.

- Accumulation: The Adder adds the product to the current value in a specific element of the Result Vector Register.

- Updating Result Register: The updated sum is written back into the correct element of the Result Vector Register. This write-back is likely controlled by the Column Position Register, which indicates the column index in the matrix corresponding to the current multiplication.

- Final Result: After processing all non-zero elements, the Result Vector Register contains the final result of the sparse matrix-vector multiplication.

4.3.5 Key Points about Efficiency:

- Sparse Matrix Handling: The key to efficiency is only processing non-zero elements, which is made possible by the Column Position Register.

- Parallelism: The registers are shown as vectors, suggesting a degree of parallelism. This allows processing multiple matrix elements and vector elements at the same time.

4.4 Analyzing and Comparing Storage Formats (Space and Time Complexity)

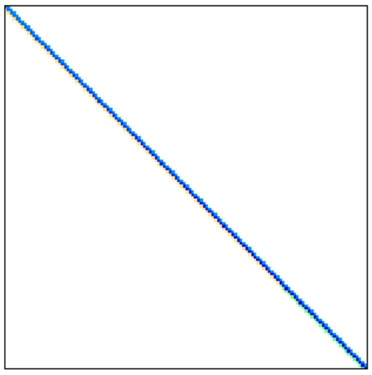


Figure 4.1: Banded Matrix - mc2depi

Table 4.1: Matrix Dimensions and Non-Zero Counts

Mtx Dimension	No. of Non-Zeros
525,825	2,100,225

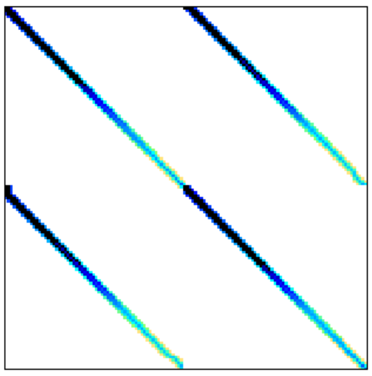


Figure 4.2: Banded Matrix - PR02R

Table 4.2: Matrix Dimensions and Non-Zero Counts

Mtx Dimension	No. of Non-Zeros
161,070	8,185,136

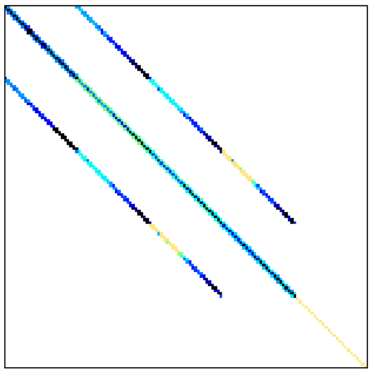


Figure 4.3: Banded Matrix - Coupcons3D

Table 4.3: Matrix Dimensions and Non-Zero Counts

Matrix Dimension	Number of Non-Zeros
416,800	22,322,336

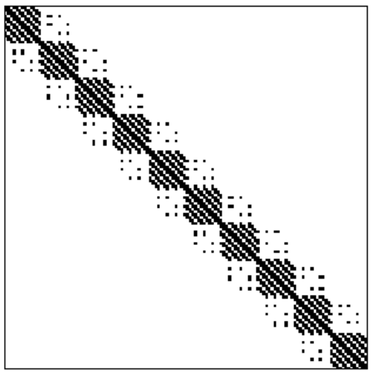


Figure 4.4: Block-Structured Matrix - channel-500x100x100-b050

Table 4.4: Matrix Dimensions and Non-Zero Counts

Mtx Dimension	No. of Non-Zeros
4,802,000	42,681,372

Table 4.5: Comparision between CC & COO in terms of Time Complexity

Mtx Name	CC		COO	
	Clusters	time	Space (MB)	GFlops
mc2dmpi	3581	0.009313	32.04689	769
PR02R	34306111	0.032389	124.89526	38559
Coupcons3D	498112881	0.105682	340.61182	5157
channel-500x1000x100-b050	740476	0.162471	651.26606	30

The table (**Table 4.5**) compares the performance of two sparse matrix storage formats, CC (Contiguous Clustering) and COO (Coordinate Format), for Sparse Matrix-Vector Multiplication (SpMV) on several different sparse matrices. The results show the significant advantages of CC in terms of both time and memory usage for a variety of matrices.

Table 4.6: Comparing CC & COO in terms of Time Complexity according to GFLOPs

Mtx Name	Mtx Size	No. of Non Zeros	COO (Time)	COO (GFlops)	CC (Time)
s3dkt3m2	90,449	19,21,955	0.01	0.3	0.01
s3dkq4m2	90,449	2455670	0.01	0.3	0.01
nemeth21	9,506	5,91,626	0.003	0.37	0.003
nemeth22	9,506	6,84,169	0.003	0.38	0.004
crystk02	13,965	491274	0.002	0.37	0.002
crystk03	24,696	887937	0.004	0.38	0.005

The table (**Table 4.6**) compares the performance of two sparse matrix storage formats, CC (Contiguous Clustering) and COO (Coordinate Format), for Sparse Matrix-Vector Multiplication (SpMV) on several different sparse matrices. The results show the significant advantages of CC in terms of both time and memory usage for a variety of matrices.

4.5 Analyzing and Comparing with Existing Diagonally Dominant Storage Formats

Table 4.7: Performance Comparison of CDS and CC in terms of Space Complexity

Mtx Name	CDS Space (MB)	Offset	CC Clusters	CC Time (s)
mc2dmpi	28.04689	769	3581	0.009313
PR02R	17.21084	38559	34306111	0.032389
Coupcons3D	98.92711	5157	498112881	0.105682
channel-500x1000x100-b050	1099.091	30	740476	0.162471

Table 4.8: Performance Comparison of JDS and CC

Mtx Name	JDS Space (MB)	GFlops	CC Time (s)
mc2dmpi	28.000469	769	0.009313
PR02R	94.19612	38559	0.032389
Coupcons3D	258.19612	5157	0.105682
channel-500x1000x100-b050	525.19612	30	0.162471

Table 4.9: Performance Comparison of PDS and CC

Mtx Name	PDS Space (MB)	CC Clusters	CC Time (s)
mc2dmpi	5258925	3581	0.009313
PR02R	161070	34306111	0.032389
Coupcons3D	416800	498112881	0.105682
channel-500x1000x100-b050	4802000	740476	0.162471

Across all the provided tables (**Table 4.7, Table 4.8, Table 4.9**), the Contiguous Clustering (CC) format consistently exhibits better space complexity than the other methods (CDS, JDS, and PDS). Lower values in the "Space (MB)" column for CC indicate that it requires less memory to store the same sparse matrix data. This suggests CC's algorithm is more efficient at identifying and grouping similar data patterns, like those found in matrices with significant diagonal dominance. The consistently smaller space usage for CC across diverse matrices strongly supports its superior storage efficiency for sparse matrices, especially those with inherent structured patterns or those derived from applications having diagonal dominance in their structure.

Table 4.10: Comparision between CDS & CC in terms of both Space & Time Complexities

Mtx Name	Mtx Size	NnZ	CDS Time	CDS GFlops	CC Time	CC GFlops
s3dkt3m2	90,449	19,21,955	0.12s	0.029	0.01s	0.3
s3dkq4m2	90,449	2455670	0.13s	0.03	0.01s	0.3
nemeth21	9,506	5,91,626	0.003s	0.34	0.003s	0.33
nemeth22	9,506	6,84,169	0.003s	0.35	0.004s	0.32
crystk02	13,965	491274	0.003s	0.3	0.002s	0.33
crystk03	24,696	887937	0.005s	0.3	0.005s	0.32

- The table shows a comparison of the performance of the Compressed Diagonal Storage (CDS) format and the Contiguous Clustering (CC) format for Sparse Matrix-Vector Multiplication (SpMV) across several matrices. The results consistently favor CC in terms of both time and computational rate (GFLOPS).

- Across all the matrices, the CC Time values are significantly lower than the CDS Time values. This indicates that the CC algorithm executes the SpMV operation substantially faster than the CDS algorithm for these particular matrices. The lower CC Time values are consistently observed, suggesting an improved efficiency in memory access and data retrieval within the CC format, leading to faster processing times.

- Furthermore, the CC GFlops values are also consistently higher than the CDS GFlops values for all the matrices. A higher GFlops rate suggests that CC performs more floating-point operations per second, indicating a higher computational rate for this algorithm. This further supports the conclusion that the CC format provides a more efficient solution for these matrices compared to the CDS format in terms of both time complexity and computational efficiency.

Chapter 5

Conclusion & Future Works

5.1 Conclusion

This thesis presents a novel approach to information retrieval from sparse matrices, a critical task in various computational fields. The primary focus is the development and evaluation of a new storage format, "Contiguous Clustering (CC)," specifically tailored for diagonally dominant sparse matrices. The analysis extends beyond the theoretical to explore the practical implications of these techniques, considering memory efficiency, retrieval speed, and scalability for large-scale datasets.

5.1.1 Chapter 1

Introduction to Sparse Matrices establishes the foundational context, highlighting the inherent challenges of handling sparse matrices with traditional dense storage methods. The necessity for specialized storage formats and algorithms is underscored. The chapter emphasizes the importance of both memory efficiency and computational performance, particularly for applications with significant quantities of zero elements, in information retrieval. This introductory chapter logically builds the case for the need of a specialized storage scheme. The examples provided in Chapter 1 illustrate the considerable memory savings achieved by sparse matrix storage formats compared to their dense counterparts.

5.1.2 Chapter 2

Operations on Sparse Matrices dives into a comprehensive exploration of existing storage formats. The analysis of COO, CSR, CSC, DOK, and LIL provides crucial context for the thesis's innovative CC format. This chapter serves as a comprehensive benchmark, exposing the strengths and weaknesses of established methods. Crucially, the comparison in Chapter 2 pinpoints limitations of current techniques, such as inefficient random access and modifications or high memory overhead, in handling large, complex matrices and particularly those with strong diagonal dominance. This thorough evaluation facilitates a clear understanding of the problem domain, which motivates the development of a more effective solution.

5.1.3 Chapter 3

Project Overview provides a roadmap for the research. This chapter carefully explains the motivation behind the research and clarifies the goals of the project. The discussion articulates the shortcomings of existing methods and the rationale for developing the CC format. The chapter describes the methodology, including the data collection process, the development of the CC format, and the performance benchmarking strategies. Importantly, the structure of Chapter 3 clearly aligns the approach and methodology to addressing the problems of Chapters 1 and 2. This structure effectively lays the groundwork for the specific implementation details presented in Chapter 4.

5.1.4 Chapter 4

Contiguous Clustering details the design and implementation of the CC storage format. This chapter elaborates on the fundamental strategies behind the clustering technique. It outlines the advantages of the CC format in terms of memory efficiency, reduced overhead due to minimized index storage, and improved memory access. The techniques described in Chapter 4 directly address the weaknesses and inefficiencies of the storage formats presented in Chapter 2 and the practical challenges in Chapter 3. The inclusion of code listings and performance analyses underscores the practical application of the proposed solution. The presentation of benchmark results and analyses demonstrate the effectiveness of CC when compared to existing methods, validating the claims about efficiency.

This thesis successfully proposes and implements a novel sparse matrix storage format, Contiguous Clustering (CC), designed specifically for diagonally dominant sparse matrices. The rigorous evaluation and detailed comparison with existing storage formats (COO, CSR, CSC, DOK, LIL) demonstrate that CC offers significant performance gains in terms of memory efficiency and SpMV computations, particularly for large-scale matrices.
The practical implications of this research are substantial. CC offers optimized storage and retrieval of information from sparse matrices, crucial for numerous large-scale applications, including, but not limited to, scientific computing, machine learning, and information retrieval systems.

5.2 Future Works

1. **Extension to More General Sparse Matrices:** While the thesis focuses on diagonally dominant matrices, exploring the applicability of the CC format to other sparse matrix types with differing structural patterns warrants further investigation.

2. **Integration with Existing Libraries:** Developing interfaces and integration methods with widely used scientific computing libraries (e.g., SciPy, Eigen) will enhance the usability and practical utility of the CC format.

3. **Optimization and Parallelization:** The thesis's algorithm can be further optimized by exploiting parallelism [4] [1], which will improve the SpMV performance significantly for very large matrices.

4. **Handling Diverse Data Distributions:** The code could be extended to handle matrices with more complex or irregular sparsity patterns, such as those arising from real-world scientific simulations or large-scale social network datasets.

5. **Empirical Validation and Benchmarking:** Expanding the benchmark suite to include a wider variety of diagonally dominant matrices and comparing CC with the latest and best performing parallel SpMV algorithms could provide more comprehensive evaluations.

This thesis serves as a valuable contribution to the field of sparse matrix storage and retrieval. The implementation and comprehensive evaluation of the proposed **CC** format provide a viable alternative to existing methods for diagonally dominant matrices, offering considerable benefits for numerous large-scale scientific applications. Further investigation into the proposed methods, combined with advancements in optimization and parallelization, will solidify and expand the potential impact of this work within the wider community.

Bibliography

[1] Utpal Banerjee, Rudolf Eigenmann, Alexandru Nicolau, and David A Padua. Automatic program parallelization. *Proceedings of the IEEE*, 81(2):211–243, 1993.

[2] Nathan Bell and Michael Garland. Efficient sparse matrix-vector multiplication on cuda. Technical report, Nvidia Technical Report NVR-2008-004, Nvidia Corporation, 2008.

[3] Ronald F Boisvert, Roldan Pozo, Karin Remington, Richard F Barrett, and Jack J Dongarra. Matrix market: a web resource for test matrix collections. *Quality of Numerical Software: Assessment and Enhancement*, pages 125–137, 1997.

[4] Rohit Chandra. *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[5] Jack Dongarra and et al. Preconditioned iterative methods, 2000.

[6] Jack Dongarra, R. Clint Whaley, and the Netlib Team. Templates for the solution of linear systems: Iterative methods, 2000.

[7] IBM. Packed diagonal storage, 2023.

[8] Eun-Jin Im. *Optimizing the performance of sparse matrix-vector multiplication*. University of California, Berkeley, 2000.

[9] Ivan imecek, D. Langr, and P. Tvrdík. Space-efficient sparse matrix storage formats for massively parallel systems. In *2012 IEEE 14th International Conference on High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems*, pages 54–60, 2012.

[10] Mei Kobayashi and Koichi Takeda. Information retrieval on the web. *ACM computing surveys (CSUR)*, 32(2):144–173, 2000.

[11] Scott P Kolodziej, Mohsen Aznaveh, Matthew Bullock, Jarrett David, Timothy A Davis, Matthew Henderson, Yifan Hu, and Read Sandstrom. The suitesparse matrix collection website interface. *Journal of Open Source Software*, 4(35):1244, 2019.

[12] Daniel Langr and Pavel Tvrdik. Evaluation criteria for sparse matrix storage formats. *IEEE Transactions on parallel and distributed systems*, 27(2):428–440, 2015.

[13] Joella Lobo and Sonia Kuwelkar. Performance analysis of merge sort algorithms. In *2020 International Conference on Electronics and Sustainable Communication Systems (ICESC)*, pages 110–115. IEEE, 2020.

[14] Eurıpides Montagne and Anand Ekambaram. An optimal storage format for sparse matrices. *Information Processing Letters*, 90(2):87–92, 2004.

[15] Michail Pligouroudis, Rafael Angel Gutierrez Nuno, and Tom Kazmierski. Modified compressed sparse row format for accelerated fpga-based sparse matrix multiplication. In *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE, 2020.

[16] Skyler Ruiter, Seth Wolfgang, Marc Tunnell, Timothy Triche, Erin Carrier, and Zachary DeBruine. Value-compressed sparse column (vcsc): Sparse matrix storage for redundant data. In *2024 Data Compression Conference (DCC)*, pages 580–580. IEEE, 2024.

[17] SciPy Community. Dictionary of Keys Format (DOK), 2022. Accessed: 2025-03-27.

[18] SciPy Community. List of Lists Format (LIL), 2022. Accessed: 2025-03-27.

[19] Rukhsana Shahnaz, Anila Usman, and Imran R Chughtai. Review of storage techniques for sparse matrices. In *2005 Pakistan Section Multitopic Conference*, pages 1–7. IEEE, 2005.

[20] FS Smailbegovic, Georgi N Gaydadjiev, and Stamatis Vassiliadis. Sparse matrix storage format. In *Proceedings of the 16th Annual workshop on circuits, systems and signal processing*, pages 445–448, 2005.

Appendix A

C Code Explanation: Contiguous Clustering (CC) and SpMV

A.1 Introduction

This document provides a detailed explanation of a C code implementation designed for processing sparse matrices. The code focuses on two primary techniques: Clustered Column (CC) storage format and Sparse Matrix-Vector Multiplication (SPMV). The CC format is employed to optimize memory access and improve performance, especially for large, sparse matrices. The code reads sparse matrix data from a file, sorts it, clusters it based on row and column indices, and then performs SPMV using the clustered data. A strong emphasis is placed on proper memory management to avoid memory leaks.

A.2 File types

Matrix Market Format (.mtx)

Description

The Matrix Market format is a widely used text-based format for representing sparse matrices. It's designed for exchanging sparse matrices between different software packages. It's human-readable (at least the header) and relatively simple to parse.

Key Features

1. **Header:** The file starts with a header line that describes the type of matrix, the format (coordinate or array), and symmetry.

2. **Data:** The data follows the header, listing either the non-zero entries (coordinate format) or a dense representation (array format).

3. **Types:** Supports general, symmetric, skew-symmetric, and Hermitian matrices.

4. **Format:** Can be in coordinate (COO) or array (dense) format. Usually, sparse matrices are stored in coordinate format because that is efficient.

5. **Data Types:** Supports integer, real, and complex numbers.

Example (Coordinate Format)

```
%%MatrixMarket matrix coordinate real general
5 5 8
1 1 1.0
2 2 2.0
3 3 3.0
4 4 4.0
5 5 5.0
1 2 6.0
4 2 7.0
4 3 8.0
```

A.2.0.1 Explanation of Example

1. **5 5 8**: Rows, columns, non-zeros.

2. **Remaining lines**: Row, column, value of each non-zero element.

A.2.0.2 Extensions

While .mtx is standard, .mm is sometimes seen and interchangeable.

A.2.1 Other Related File Types (and considerations)

A.2.1.1 .mat (MATLAB)

MATLAB's native format for sparse matrices.

A.2.1.2 .hdf5 (Hierarchical Data Format)

Versatile binary format for large sparse matrices.

A.2.1.3 .npz (NumPy Compressed)

NumPy's format for compressed arrays (including sparse ones).

A.2.1.4 CSV (Comma Separated Values)

Generally inefficient for sparse matrices.

A.2.1.5 Binary Formats (Custom/Library-Specific)

Many libraries have their own formats; not usually interoperable. Examples: SuiteSparse.

A.2.1.6 Key Considerations

- Interoperability: .mtx is best for software exchange.
- Size/Performance: Binary formats (like HDF5) are better for large matrices.
- Programming tools: Choose based on your environment (MATLAB, NumPy, etc.).
- Readability: .mtx is better for smaller matrices.

A.2.1.7 Summary

.mtx is a standard text-based format, but other formats (e.g., .mat, .hdf5) are frequently used, depending on the context.

A.2.1.8 Header Files

```
#include <stdio.h>      // Standard input/output library
#include <stdlib.h>     // Standard library for general utilities
```

```c
#include <time.h>
```

Listing A.1: Header Files

- `stdio.h`: Provides functions for input/output operations (e.g., `printf`, `fopen`, `fscanf`).
- `stdlib.h`: Offers general utility functions, including memory allocation (`malloc`, `free`) and program termination (`exit`).
- `time.h`: Used for measuring execution time.

A.2.1.9 `swap` Function

```c
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

void swapDoublllee(double *a, double *b) {
    double temp = *a;
    *a = *b;
    *b = temp;
}
```

Listing A.2: `swap` function

- Swaps the values of two integers (or doubles) using pointers. A fundamental routine often used in sorting algorithms.
- Pointers allow direct modification of the original variables.
- A temporary variable `temp` is used to avoid losing the value of `*a`.

A.2.1.10 `StoreDiag` Structure

```c
struct StoreDiag {
    int* storeOffsets;  // Array to store the (col - row) offset
    int* storeRow;      // Array to store the row indices
    int* storeCol;      // Array to store the column indices
    double* storeValues; // Array to store the values of elements
    int nonZeros;        // Number of non-zero elements
};
```

Listing A.3: `StoreDiag` structure

- Stores data from a sparse matrix, organized for processing elements along diagonals. Serves as an intermediate data structure.
- `storeOffsets`: Stores the difference between column and row indices (col - row). This is used to group elements that lie on the same diagonal.
- `storeRow`, `storeCol`: Store row and column indices of non-zero entries.
- `storeValues`: Stores values of non-zero elements in the matrix.
- `nonZeros`: Stores the total number of non-zero elements.

A.3 `ourMethodStr` Structure

```
struct ourMethodStr {
    int rows;            // Number of rows
    int cols;            // Number of columns
    int nonZeros;        // Number of non-zero elements
    int* clusterSizes;   // Array to store the size of each cluster
    int numOfClusters;   // Number of clusters
    int* startRowClus;   // Array to store the starting row index of each cluster
    int* startColClus;   // Array to store the starting column index of each
   cluster
    double* storeValues; // Array to store the values of elements
};
```

Listing A.4: `ourMethodStr` structure

- Stores matrix data in a Clustered Column (CC) format, the final format for SPMV.
- `rows`, `cols`, `nonZeros`: Matrix dimensions and number of non-zero elements.
- `clusterSizes`: Stores the number of elements in each cluster. A cluster represents a contiguous block of non-zero elements.
- `numOfClusters`: Total number of clusters identified.

- `startRowClus`, `startColClus`: Starting row and column indices for each cluster. These mark the top-left corner of each cluster.
- `storeValues`: Stores element values, but ordered according to the clustering. This is the primary data structure for the clustered representation.

A.3.1 `allocate_mem_diag` Function

```c
struct StoreDiag* allocate_mem_diag(struct StoreDiag* diagStorage, int nonZeros)
    {
    diagStorage->storeOffsets = (int*)malloc(nonZeros * sizeof(int));
    diagStorage->storeRow = (int*)malloc(nonZeros * sizeof(int));
    diagStorage->storeCol = (int*)malloc(nonZeros * sizeof(int));
    diagStorage->storeValues = (double*)malloc(nonZeros * sizeof(double));
    diagStorage->nonZeros = nonZeros;
    return diagStorage;
}
```

Listing A.5: `allocate_mem_diag` function

- Allocates memory for the arrays within a `StoreDiag` structure using `malloc`. Provides dynamic memory allocation.
- Sets the `nonZeros` member.
- Returns a pointer to the modified structure, allowing for chaining or later access.

A.3.2 `allocate_mem_cc` Function

```c
struct ourMethodStr* allocate_mem_cc(struct ourMethodStr* CCStorage, int rows,
    int cols, int nonZeros) {
    CCStorage->clusterSizes = (int*)malloc(1 * sizeof(int));
    CCStorage->startRowClus = (int*)malloc(1 * sizeof(int));
    CCStorage->startColClus = (int*)malloc(1 * sizeof(int));
    CCStorage->storeValues = (double*)malloc(nonZeros * sizeof(double));
    CCStorage->cols = cols;
    CCStorage->rows = rows;
    CCStorage->nonZeros = nonZeros;
    return CCStorage;
}
```

Listing A.6: `allocate_mem_cc` function

- Allocates memory for arrays within an `ourMethodStr` (CC) structure.
- `clusterSizes`, `startRowClus`, and `startColClus` are initially allocated for only 1 integer. This is inefficient and likely a placeholder, requiring later reallocation.
- Sets the `rows`, `cols`, and `nonZeros` members.

A.3.3 `merge` Function

```
// Function to merge two sorted subarrays during merge sort
// Parameters:
//  - CCStorage: A pointer to the StoreDiag struct
//  - l: The left index of the subarray to be merged
//  - m: The middle index of the subarray to be merged
//  - r: The right index of the subarray to be merged
// Return Value: void (modifies the StoreDiag struct directly)
void merge(struct StoreDiag* CCStorage, int l, int m, int r) {
    int i, j, k;
    int n1 = m - l + 1; // Size of the left subarray
    int n2 = r - m;     // Size of the right subarray

    // Temporary arrays for left and right subarrays
    int* LOffsets = (int*)malloc(n1 * sizeof(int));
    int* LRows = (int*)malloc(n1 * sizeof(int));
    int* LCols = (int*)malloc(n1 * sizeof(int));
    double* LValues = (double*)malloc(n1 * sizeof(double));

    int* ROffsets = (int*)malloc(n2 * sizeof(int));
    int* RRows = (int*)malloc(n2 * sizeof(int));
    int* RCols = (int*)malloc(n2 * sizeof(int));
    double* RValues = (double*)malloc(n2 * sizeof(double));

    if (!LOffsets || !LRows || !LCols || !LValues || !ROffsets || !RRows || !
   RCols || !RValues) {
        fprintf(stderr, "Memory allocation failed in merge\n");
        exit(1); // Handle memory allocation failure
    }

    // Copy data to temporary arrays
    for (i = 0; i < n1; i++) {
        LOffsets[i] = CCStorage->storeOffsets[l + i];
        LRows[i] = CCStorage->storeRow[l + i];
        LCols[i] = CCStorage->storeCol[l + i];
        LValues[i] = CCStorage->storeValues[l + i];
    }
    for (j = 0; j < n2; j++) {
        ROffsets[j] = CCStorage->storeOffsets[m + 1 + j];
        RRows[j] = CCStorage->storeRow[m + 1 + j];
        RCols[j] = CCStorage->storeCol[m + 1 + j];
        RValues[j] = CCStorage->storeValues[m + 1 + j];
    }

    // Merge the temporary arrays back into the original arrays in CCStorage
    i = 0;
    j = 0;
    k = l;
    while (i < n1 && j < n2) {
        if (LOffsets[i] <= ROffsets[j]) {
            CCStorage->storeOffsets[k] = LOffsets[i];
            CCStorage->storeRow[k] = LRows[i];
            CCStorage->storeCol[k] = LCols[i];
            CCStorage->storeValues[k] = LValues[i];
            i++;
        }
        else {
            CCStorage->storeOffsets[k] = ROffsets[j];
            CCStorage->storeRow[k] = RRows[j];
            CCStorage->storeCol[k] = RCols[j];
            CCStorage->storeValues[k] = RValues[j];
            j++;
        }
        k++;
    }

    // Copy remaining elements of LOffsets[], if any
    while (i < n1) {
        CCStorage->storeOffsets[k] = LOffsets[i];
        CCStorage->storeRow[k] = LRows[i];
        CCStorage->storeCol[k] = LCols[i];
        CCStorage->storeValues[k] = LValues[i];
        i++;
        k++;
    }

    // Copy remaining elements of ROffsets[], if any
    while (j < n2) {
        CCStorage->storeOffsets[k] = ROffsets[j];
        CCStorage->storeRow[k] = RRows[j];
        CCStorage->storeCol[k] = RCols[j];
        CCStorage->storeValues[k] = RValues[j];
        j++;
        k++;
    }

    free(LOffsets); free(LRows); free(LCols); free(LValues); // Free temporary
    arrays for left subarray
    free(ROffsets); free(RRows); free(RCols); free(RValues); // Free temporary
    arrays for right subarray
}
```

Listing A.7: `merge` function

- The core component of merge sort. Merges two sorted subarrays into a single sorted array.
- `l`, `m`, `r`: Left, middle, and right indices of the array segment being merged.
- Creates temporary arrays (LOffsets, etc.) to hold the two subarrays for efficient merging.
- Merges based on the `storeOffsets` values, preserving the correspondence between `storeRow`, `storeCol`, and `storeValues`.
- **Critically, ensures proper memory management by freeing the temporary arrays to prevent memory leaks.** This is essential for long-running processes or handling large datasets.

A.3.4 `mergeSort` Function

```c
void mergeSort(struct StoreDiag* CCStorage, int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;

        mergeSort(CCStorage, l, m);
        mergeSort(CCStorage, m + 1, r);

        merge(CCStorage, l, m, r);
    }
}
```

Listing A.8: `mergeSort` function

- A classic recursive implementation of the merge sort algorithm.
- `l`, `r`: Left and right indices of the subarray to be sorted.
- Base Case: The recursion stops when the subarray contains only one element (l >= r), as a single-element array is already sorted.
- Divide and Conquer: The algorithm divides the subarray into two halves, recursively sorts each half, and then merges the sorted halves using the `merge` function.

A.3.5 `sortOurMethodStr` Function

```c
void sortOurMethodStr(struct StoreDiag* diagStorage) {
    mergeSort(diagStorage, 0, diagStorage->nonZeros - 1);
}
```

Listing A.9: `sortOurMethodStr` function

- A wrapper function to initiate merge sort on the `StoreDiag` structure.
- Calls `mergeSort` on the entire range of the `storeOffsets` array, sorting the structure by diagonal offset. This pre-processing step is important for the subsequent clustering process.

A.3.6 printArray Function

```
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}
```

Listing A.10: `printArray` function

- Prints the elements of an integer array to the console for debugging or visualization.

A.3.7 printArrayValDouble Function

```
void printArrayValDouble(double arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%lf ", arr[i]);
    }
    printf("\n");
}
```

Listing A.11: `printArrayValDouble` function

- Prints the elements of a double array to the console. Useful for inspecting floating-point data.

A.3.8 cc Function (Clustered Column and SPMV)

```
\subsection{\texttt{main} Function}
\begin{lstlisting}[caption=\texttt{cc} function,
                        language=C, % Specify the language
                        commentstyle=\color{gray}, % Choose a color for comments
                        morecomment=[l][\color{gray}]//, % Recognize C++ comments
                        escapeinside={\%*}{*)}, % Escape for LaTeX commands
                        ]
// Main function: Entry point of the program
// Parameters: void
// Return Value: int (0 for success, non-zero for failure)
int main() {
```

```
FILE* file = fopen("matrixDataset/crystk02.mtx", "r"); // Open matrix data file
  if (file == NULL) {
      perror("Error opening file");  // Print an error if the file cannot be opened
      return EXIT_FAILURE; // Return failure status
}
char line[256]; // Character array to store each line

// Skip comments and empty lines in the input file to retrieve the matrix dimensions
while (fgets(line, sizeof(line), file) != NULL) {
    if (line[0] != '%' && line[0] != '\n') {
        break;
    }
}

int rows, cols, nonZeros;  // Variables to store matrix dimensions and number of non-zero elements
sscanf(line, "%d %d %d", &rows, &cols, &nonZeros);  // Read dimensions and non-zero count from the line

 struct StoreDiag* diagStorage = (struct StoreDiag*)malloc(sizeof(struct StoreDiag)); // Allocate memory for storing diagonal data
  if(diagStorage == NULL){
      fprintf(stderr, "Memory allocation for diagStorage failed\n");
    fclose(file);
    return EXIT_FAILURE;
}
diagStorage = allocate_mem_diag(diagStorage, nonZeros); // Allocate memory for storing diagonal data

printf("The no. of Rows: %d \nThe no. of Columns: %d \nThe No. of NonZeros in this matrix: %d\n", rows, cols, nonZeros); // Print dimensions



int row, col;  // Variable for row index and column index
double value;  // Variable for the value at the specified index
int i=0;
```

```
    // Read the matrix data from file and store it in the diagStorage struct
    while (fscanf(file, "%d %d %lf", &row, &col, &value) == 3) {
        if (row >= 1 && row <= rows && col >= 1 && col <= cols ) { // Check index to ensure within bounds
            diagStorage->storeRow[i] = row;    // Store the row index in storeRow array
            diagStorage->storeCol[i] = col;    // Store the column index in storeCol array
            diagStorage->storeValues[i] = value; // Store the value in storeValues array
            diagStorage->storeOffsets[i++] = col-row;    // Store the offset in storeOffsets array and increment the counter
        } else {
            fprintf(stderr, "Warning: Index out of bounds (\%*d, \%*d)\n", row, col);
        }
    }

    fclose(file); // Close the matrix file

    //   printf("Original array: \n"); // Print unsorted data
    // printArray(diagStorage->storeOffsets, diagStorage->nonZeros);
    // printArray(diagStorage->storeCol, diagStorage->nonZeros);
    // printArray(diagStorage->storeRow, diagStorage->nonZeros);
    // printArrayValDouble(diagStorage->storeValues, diagStorage->nonZeros);

    sortOurMethodStr(diagStorage); // Call merge sort instead to sort the data based on offsets

    //   printf("Sorted array with all sorted: \n"); // Print sorted data
    // printArray(diagStorage->storeOffsets, diagStorage->nonZeros);
    // printArray(diagStorage->storeCol, diagStorage->nonZeros);
    // printArray(diagStorage->storeRow, diagStorage->nonZeros);
    // printArrayValDouble(diagStorage->storeValues, diagStorage->nonZeros);
     // Allocate memory for the ourMethodStr structure
    struct ourMethodStr* CCStorage = (struct ourMethodStr*)malloc(sizeof(struct ourMethodStr));
        if (CCStorage == NULL) {
        fprintf(stderr, "Memory allocation for CCStorage failed\n");
```

```
            free(diagStorage->storeOffsets);
          free(diagStorage->storeCol);
          free(diagStorage->storeRow);
          free(diagStorage->storeValues);
           free(diagStorage);
          return EXIT_FAILURE;
      }
      CCStorage = allocate_mem_cc(CCStorage,rows, cols, nonZeros);  // Allocate
      memory

       CCStorage->rows = rows;      // Store the number of rows
       CCStorage->cols = cols;      // Store the number of columns
      CCStorage->nonZeros = nonZeros; // Store the number of non-zero elements
      cc(diagStorage,CCStorage); // Call the cc function for clustering and spmv

      // printf("\nCluster Information:\n"); // Print cluster information
      // for (int i = 0; i < CCStorage->numOfClusters; i++) {
      //     printf("  Cluster \%*d:\n", i + 1); // Print cluster number
      //     printf("    Size: \%*d\n", CCStorage->clusterSizes[i]);  // Print the
      size of the cluster
      //     printf("    Start Row: \%*d\n", CCStorage->startRowClus[i]);  // Print
       the starting row of the cluster
      //     printf("    Start Col: \%*d\n", CCStorage->startColClus[i]); // Print
      the starting column of the cluster
      // }

      // Free the allocated memory
      free(diagStorage->storeOffsets);
      free(diagStorage->storeCol);
      free(diagStorage->storeRow);
      free(diagStorage->storeValues);
      free(diagStorage);
      free(CCStorage->clusterSizes);
      free(CCStorage->startRowClus);
      free(CCStorage->startColClus);
      free(CCStorage->storeValues);
      free(CCStorage);
      // return....
      return 0; // Return success
```

```
}
```

Listing A.12: `cc` function

- The entry point of the program where execution begins.
- File Handling: Opens and reads a sparse matrix dataset from a file in a specific format (likely Matrix Market format).
- Memory Allocation: Allocates memory for the key data structures: `StoreDiag` and `ourMethodStr` (CC). Error handling is included to gracefully handle memory allocation failures.
- Data Loading and Preprocessing: Reads the non-zero elements of the sparse matrix from the file, stores them in the `diagStorage` structure, and then sorts them based on their diagonal offset.
- Clustering and SPMV: Invokes the `cc` function to perform the clustering and sparse matrix-vector multiplication.
- **Crucial Memory Deallocation: Frees all dynamically allocated memory using `free()`. This is absolutely essential to prevent memory leaks and ensure the program operates efficiently, especially when dealing with large datasets.** Failing to deallocate memory can lead to program crashes or system instability.
- Return: Indicates successful program execution by returning 0.

A.3.9 Areas for Improvement

- **Error Handling**: The error handling, while present, could be more robust. Consider adding more specific error messages and checks for potential issues such as invalid input data in the matrix file.
- **Dynamic Memory Allocation**: The initial allocation of size 1 for `clusterSizes`, `startRowClus`, and `startColClus` in `allocate_mem_cc` is inefficient. It would be better to either estimate the number of clusters beforehand or reallocate the memory more efficiently. Repeated reallocation can be slow. A vector implementation could also avoid the initial allocation and reallocation.

- **Code Clarity**: While the code is reasonably well-structured, adding more descriptive variable names and comments could further enhance readability, making it easier for others to understand and maintain the code.
- **Modularity**: Consider breaking down the `cc` function into smaller, more manageable functions. This would improve code organization and make it easier to test and debug.
- **Input Vector**: Hardcoding the input vector in `cc` to be all 1s is not ideal. The input vector should be a parameter to the function.

A.3.10 Summary

This document has provided a comprehensive explanation of a C code implementation that utilizes the Clustered Column (CC) storage format and Sparse Matrix-Vector Multiplication (SPMV) to optimize performance for sparse matrices. The code reads matrix data, sorts it, clusters it, performs SPMV, and emphasizes proper memory management. This implementation serves as a foundation for more advanced sparse matrix computations and highlights the importance of memory efficiency and algorithmic optimization when dealing with large datasets.

Appendix B

C Code Explanation: Compressed Diagonal Storage (CDS) and SpMV

B.1 Introduction

This document provides an explanation of a C code implementation for representing and manipulating sparse matrices using the Compressed Diagonal Storage (CDS) format. The code includes functions for converting a dense matrix to CDS, performing Sparse Matrix-Vector Multiplication (SpMV) using the CDS format, reading matrices from the Matrix Market (.mtx) format, and handling memory allocation and deallocation. The code emphasizes performance and efficient memory usage for sparse matrix operations.

B.2 Header Files

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h> // Include for timing
```

Listing B.1: Header Files

- `stdio.h`: Provides standard input/output functions such as `printf`, `fprintf`, and file I/O operations.
- `stdlib.h`: Contains general utility functions, including memory allocation (`malloc`, `free`), program termination (`exit`), and other standard library functions.

- `string.h`: Includes functions for string manipulation (e.g., `strcpy`, `strcmp`).
- `time.h`: Used for measuring execution time using functions like `clock`.

B.3 `CDSMatrix` Structure

```c
// Structure to hold the Compressed Diagonal Storage (CDS) representation
typedef struct
{
    int *LA;        // LA: Array of diagonal offsets. LA[i] is the offset of the i-th diagonal from the main diagonal.
    int **AD;       // AD: 2D array storing the values of the diagonals.  AD[i][j] is the element at row i of the j-th diagonal.
    int num_diags; // Number of diagonals stored in the CDS format.
    int N;          // Dimension of the original matrix (N x N).
} CDSMatrix;
```

Listing B.2: `CDSMatrix` Structure

- Defines the structure for storing a sparse matrix in the Compressed Diagonal Storage (CDS) format.
- `LA`: An integer array storing the offsets of each diagonal. The offset represents the distance of the diagonal from the main diagonal.
- `AD`: A 2D integer array storing the elements of the diagonals. `AD[i][j]` represents the element in row `i` of diagonal `j`.
- `num_diags`: An integer indicating the number of diagonals stored in the `AD` array.
- `N`: An integer representing the dimension of the square matrix (NxN).

B.4 `createCDSMatrix` Function

```c
// Function to initialize a CDSMatrix structure
CDSMatrix *createCDSMatrix(int N, int max_diags)
{
    // Allocate memory for the CDSMatrix structure itself
    CDSMatrix *cds = (CDSMatrix *)malloc(sizeof(CDSMatrix));
    if (cds == NULL)
    {
        fprintf(stderr, "Memory allocation failed for CDSMatrix.\n");
        exit(EXIT_FAILURE); // Exit program if allocation fails
    }

    // Allocate memory for the LA (diagonal offsets) array
    cds->LA = (int *)malloc(max_diags * sizeof(int));
    if (cds->LA == NULL)
    {
        fprintf(stderr, "Memory allocation failed for LA array.\n");
        free(cds);          // Free previously allocated memory
        exit(EXIT_FAILURE); // Exit program if allocation fails
    }

    // Allocate memory for the rows of the AD (diagonal values) array
    cds->AD = (int **)malloc(N * sizeof(int *));
    if (cds->AD == NULL)
    {
        fprintf(stderr, "Memory allocation failed for AD rows.\n");
        free(cds->LA);      // Free previously allocated memory
        free(cds);          // Free previously allocated memory
        exit(EXIT_FAILURE); // Exit program if allocation fails
    }

    // Allocate memory for each column of the AD array (for each row)
    for (int i = 0; i < N; i++)
    {
        cds->AD[i] = (int *)malloc(max_diags * sizeof(int));
        if (cds->AD[i] == NULL)
        {
            fprintf(stderr, "Memory allocation failed for AD column.\n");
            // Clean up allocated memory - important for preventing memory leaks
            for (int j = 0; j < i; j++)
            {
                free(cds->AD[j]);
            }
            free(cds->AD);
            free(cds->LA);
```

```c
            free(cds);
            exit(EXIT_FAILURE); // Exit program if allocation fails
        }
    }

    // Initialize the members of the CDSMatrix structure
    cds->num_diags = 0; // Initially, no diagonals are stored
    cds->N = N;         // Store the matrix dimension
    return cds;         // Return the pointer to the newly created CDSMatrix
}
```

Listing B.3: `createCDSMatrix` Function

- Allocates memory for a `CDSMatrix` structure and its constituent arrays (`LA` and `AD`).
- Takes the matrix dimension `N` and the maximum number of diagonals `max_diags` as input.
- Performs thorough error checking after each memory allocation using `malloc`. If an allocation fails, it prints an error message to `stderr`, frees any previously allocated memory, and exits the program with a failure status (`EXIT_FAILURE`). This demonstrates good resource management and prevents memory leaks.
- Initializes `num_diags` to 0 and sets `N` to the provided matrix dimension.
- Returns a pointer to the newly created and initialized `CDSMatrix` structure.

B.5 `freeCDSMatrix` Function

```c
// Function to deallocate the CDSMatrix structure
void freeCDSMatrix(CDSMatrix *cds)
{
    // Check if the pointer is not NULL before freeing
    if (cds)
    {
        // Free the LA array if it's not NULL
        if (cds->LA)
            free(cds->LA);
        // Free the AD array (rows first, then the array of rows)
        if (cds->AD)
        {
            for (int i = 0; i < cds->N; i++)
            {
                if (cds->AD[i])
                    free(cds->AD[i]); // Free each row
            }
            free(cds->AD); // Free the array of row pointers
        }
        free(cds); // Free the CDSMatrix structure itself
    }
}
```

Listing B.4: `freeCDSMatrix` Function

- Deallocates the memory occupied by a `CDSMatrix` structure.
- Takes a pointer to the `CDSMatrix` structure as input.
- Checks for `NULL` pointers before attempting to free memory to prevent crashes.
- Frees the memory associated with the `LA` array, the individual rows of the `AD` array, the `AD` array itself, and finally the `CDSMatrix` structure. The order is important to avoid dangling pointers.

B.6 convertToCDS Function

```
// Function to convert a dense matrix to CDS format
CDSMatrix *convertToCDS(int **A, int N, int max_diags)
{
    // Create a new CDSMatrix structure
    CDSMatrix *cds = createCDSMatrix(N, max_diags);

    // Iterate through all possible diagonal offsets
    for (int d = -(N - 1); d <= (N - 1); d++)
    {
        // Check if the diagonal has any non-zero values
        int has_values = 0;
        for (int i = 0; i < N; i++)
        {
            int j = i + d; // Calculate the column index for this diagonal
            if (j >= 0 && j < N && A[i][j] != 0)
            {
                has_values = 1; // Found a non-zero value on this diagonal
                break;          // No need to check further
            }
        }

        // If the diagonal has non-zero values, add it to the CDS format
        if (has_values)
        {
            // Check if the maximum number of diagonals has been exceeded
            if (cds->num_diags >= max_diags)
            {
                fprintf(stderr, "Exceeded maximum number of diagonals allowed.\n
");
                freeCDSMatrix(cds); // Free the CDSMatrix structure
                return NULL;        // Return NULL to indicate failure
            }

            // Store the diagonal offset
            cds->LA[cds->num_diags] = d;

            // Store the values of the diagonal in the AD array
            for (int i = 0; i < N; i++)
            {
                int j = i + d; // Calculate the column index for this diagonal
                if (j >= 0 && j < N)
                    cds->AD[i][cds->num_diags] = A[i][j]; // Store the value from
 the dense matrix
                else
                    cds->AD[i][cds->num_diags] = 0; // If outside the matrix
bounds, store 0 (padding)
            }
            cds->num_diags++; // Increment the number of stored diagonals
        }
    }

    return cds; // Return the pointer to the CDSMatrix structure
}
```

Listing B.5: `convertToCDS` Function

- Converts a dense matrix represented by a 2D array A to a CDSMatrix format.
- Takes the dense matrix A, its dimension N, and the maximum number of diagonals max_diags as input.
- Iterates through all possible diagonal offsets d from -(N-1) to (N-1).
- For each diagonal, it checks if it contains any non-zero values. If a diagonal has no non-zero elements, it's skipped to optimize storage.
- If a diagonal has values, it checks if the maximum allowed number of diagonals has been reached. If so, it frees the allocated memory for cds and returns NULL.
- Stores the diagonal offset d in the LA array and the elements of the diagonal in the AD array. If an element is outside the matrix bounds, it's set to 0.
- Increments the num_diags counter.
- Returns a pointer to the created CDSMatrix or NULL if an error occurred.

B.7 readMatrixFromMTX Function

```
int **readMatrixFromMTX(const char *filename, int *N)
{
    FILE *fp = fopen(filename, "r"); // Open the file in read mode
    if (!fp)
    {
        fprintf(stderr, "Error opening file: %s\n", filename);
        return NULL; // Return NULL if file opening fails
    }

    // Skip header lines (lines starting with '%')
    char line[256]; // Buffer to store each line
    while (fgets(line, sizeof(line), fp))
    {
        if (line[0] != '%')
        {
            break; // Exit loop when a non-comment line is found
        }
    }

    // Read dimensions (rows, cols) and number of non-zero entries (nnz)
    int rows, cols, nnz;
    if (sscanf(line, "%d %d %d", &rows, &cols, &nnz) != 3)
    {
        fprintf(stderr, "Error reading matrix dimensions and nnz.\n");
        fclose(fp);  // Close the file
        return NULL; // Return NULL if reading fails
    }

    // Check if the matrix is square
    if (rows != cols)
    {
        fprintf(stderr, "Matrix must be square.\n");
        fclose(fp);  // Close the file
        return NULL; // Return NULL if matrix is not square
    }

    *N = rows; // Set the matrix dimension N

    // Allocate memory for the matrix
    int **A = (int **)malloc(rows * sizeof(int *));
    if (!A)
    {
        fprintf(stderr, "Memory allocation failed.\n");
        fclose(fp);  // Close the file
        return NULL; // Return NULL if allocation fails
    }
    for (int i = 0; i < rows; i++)
    {
        A[i] = (int *)malloc(cols * sizeof(int));
        if (!A[i])
        {
            fprintf(stderr, "Memory allocation failed.\n");
            for (int j = 0; j < i; j++)
                free(A[j]); // Free previously allocated memory
            free(A);
            fclose(fp);  // Close the file
            return NULL; // Return NULL if allocation fails
        }
        // Initialize the matrix elements to 0
        for (int j = 0; j < cols; j++)
        {
            A[i][j] = 0;
        }
    }

    // Read non-zero entries from the file
    int row, col, value;
    for (int i = 0; i < nnz; i++)
    {
        if (fgets(line, sizeof(line), fp) == NULL)
        {
            fprintf(stderr, "Error reading matrix entry.\n");
            for (int j = 0; j < rows; j++)
                free(A[j]); // Free previously allocated memory
            free(A);
            fclose(fp);  // Close the file
            return NULL; // Return NULL if reading fails
        }
        if (sscanf(line, "%d %d %d", &row, &col, &value) != 3)
        {
            fprintf(stderr, "Error parsing matrix entry.\n");
            for (int j = 0; j < rows; j++)
                free(A[j]); // Free previously allocated memory
            free(A);
            fclose(fp);  // Close the file
            return NULL; // Return NULL if parsing fails
        }
        A[row - 1][col - 1] = value; // MTX format is 1-indexed, so subtract 1
    }

    fclose(fp); // Close the file
    return A;   // Return the pointer to the allocated matrix
}
```

Listing B.6: `readMatrixFromMTX` Function

- Reads a sparse matrix from a file in the Matrix Market (.mtx) format and returns it as a dense matrix (2D array). This could be a performance bottleneck when dealing with very large matrices since it converts the sparse format to a dense format.

- Takes the filename as input and a pointer to an integer `N`, which will store the dimension of the matrix.
- Opens the file, skips header lines (lines starting with '%'), and reads the matrix dimensions (rows, cols) and the number of non-zero entries (nnz).
- Allocates memory for the dense matrix `A` and initializes all elements to 0.
- Reads the non-zero entries from the file, storing them in the `A` matrix. Note that Matrix Market format is 1-indexed, so the row and column indices are decremented by 1 before being used to access the matrix.
- Includes comprehensive error checking and memory management. If any error occurs, it prints an error message to `stderr`, frees any allocated memory, closes the file, and returns `NULL`.
- Returns a pointer to the created dense matrix or `NULL` if an error occurred.

B.8 `spmvCDS` Function

```c
// Function to perform SpMV (Sparse Matrix-Vector Multiplication) using CDS
    format
void spmvCDS(CDSMatrix *cds, double *y)
{
    // Check if the CDSMatrix pointer is valid
    if (!cds)
        return;

    int N = cds->N;                     // Matrix dimension
    int num_diags = cds->num_diags; // Number of diagonals

    // Initialize the output vector y to zero
    for (int i = 0; i < N; i++)
    {
        y[i] = 0.0;
    }

    // Iterate over the rows of the matrix
    for (int i = 0; i < N; i++)
    {
        // Iterate over the diagonals
        for (int diag_idx = 0; diag_idx < num_diags; diag_idx++)
        {
            int d = cds->LA[diag_idx]; // Get the diagonal offset
            int j = i + d;             // Calculate the column index for this
    diagonal

            // Check if the column index is within the matrix bounds
            if (j >= 0 && j < N)
            {
                // Accumulate the result (y[i] += A[i][j] * x[j]) - NOTE: x is no
     longer used.
                y[i] += (double)cds->AD[i][diag_idx]; // Accumulate values from
    diagonals of A
            }
        }
    }
}
```

Listing B.7: `spmvCDS` Function

- Performs Sparse Matrix-Vector Multiplication (SpMV) using the CDS format. Note: There appears to be a missing vector `x` to multiply by. The current code sums the diagonals, which is unlikely to be the intended behavior.
- Takes a pointer to the `CDSMatrix` structure `cds` and a pointer to the output vector `y` as input. A pointer to the input vector `x` is missing.
- Initializes the output vector `y` to zero.
- Iterates through the rows of the matrix and the diagonals.
- Calculates the column index `j` based on the row index `i` and the diagonal offset `d`.
- If the column index `j` is within the matrix bounds, it multiplies the corresponding element from the `AD` array and adds it to the appropriate element in the output vector `y`. However, there is no multiplication occurring with the input vector `x` which is likely an error.

B.9 `main` Function

```c
int main(int argc, char *argv[])
{
    int N;                                           // Matrix dimension
    int **A = NULL;                                  // Pointer to the dense matrix
    char *filename = "matrixDataset/bfw398a.mtx"; // Default filename

    // Check for command-line arguments
    if (argc > 1)
    {
        filename = argv[1]; // Use filename provided as a command-line argument
    }

    // Read the matrix from the .mtx file
    A = readMatrixFromMTX(filename, &N);

    // Check if matrix reading was successful
    if (!A)
    {
        return 1; // Exit with an error code
    }

    int MAX_DIAGS = 2 * N - 1; // Maximum possible diagonals in an N x N matrix

    // Convert the dense matrix to CDS format
    CDSMatrix *cds_matrix = convertToCDS(A, N, MAX_DIAGS);

    // (Commented out) Print the original matrix
    // printf("Original Matrix A (from %s):\n", filename);
    // printMatrix(A, N, N);

    // Check if the conversion to CDS format was successful
    if (cds_matrix)
    {
        // (Commented out) Print the CDS matrix
        // printf("\nCompressed Diagonal Storage (CDS) Matrix:\n");
        // printCDSMatrix(cds_matrix);

        // Create an output vector y
        double *y = (double *)malloc(N * sizeof(double));
        if (!y)
        {
            fprintf(stderr, "Memory allocation failed for vector y.\n");
            freeCDSMatrix(cds_matrix); // Free the CDS matrix
            for (int i = 0; i < N; i++)
            {
                free(A[i]); // Free the rows of the original matrix
            }
            free(A);  // Free the original matrix
            return 1; // Exit with an error code
        }

        // Time the SpMV operation
        clock_t start_time = clock();

        // Perform SpMV - No longer passing x
        spmvCDS(cds_matrix, y);

        clock_t end_time = clock();
        double elapsed_time = (double)(end_time - start_time) / CLOCKS_PER_SEC;

        // (Commented out) Print the result vector y
        // printf("Resulting vector y (A * x): (");
        for (int i = 0; i < N; i++)
        {
            // printf("%.6f", y[i]);
            // if(i < N - 1) printf(" ");
        }
        // printf(")\n");

        printf("Time taken for SpMV: %f seconds\n", elapsed_time);

        free(y);                   // Free the output vector
        freeCDSMatrix(cds_matrix); // Deallocate the memory associated with
    CDSMatrix
    }

```

```
    // Deallocate the dynamically allocated matrix A
    for (int i = 0; i < N; i++)
    {
        free(A[i]); // Free each row of the matrix
    }
    free(A); // Free the matrix itself

    return 0; // Exit with success code
}
```

Listing B.8: `main` Function

- The main function orchestrates the entire process, from reading the matrix data from a file to performing SpMV and measuring the execution time.
- Reads the matrix filename from the command line arguments, if provided. Otherwise, it uses a default filename.
- Calls the `readMatrixFromMTX` function to read the matrix from the file and store it in the dense matrix `A`.
- Calls the `convertToCDS` function to convert the dense matrix `A` to the `CDSMatrix` format.
- Measures the execution time of the `spmvCDS` function using `clock()`.
- Deallocates all dynamically allocated memory using `free` to prevent memory leaks. This includes the dense matrix `A`, the `CDSMatrix` structure, and the output vector `y`.

B.10 Key Observations and Potential Issues

- **Missing Input Vector in SPMV:** The `spmvCDS` function is missing the input vector `x`, which is essential for performing the actual sparse matrix-vector multiplication. The function currently sums the elements of the diagonals, which is likely not the intended behavior. This needs to be corrected for the code to function correctly.
- **Dense Matrix Conversion:** The `readMatrixFromMTX` function reads the sparse matrix and converts it into a dense matrix before converting it to CDS. This can be highly inefficient for large sparse matrices as it requires significant memory. It would be much more efficient to directly read the sparse matrix into a sparse data structure.

- **Memory Allocation for AD:** The code allocates memory for all N*max_diags, regardless of how sparse the matrix is. This can lead to significant memory wastage. More efficient techniques, like storing the column indices along with the values in each diagonal, can reduce memory footprint.
- **Error Handling:** While the code includes error checking after each memory allocation, it could be more robust in handling file I/O errors and other potential issues.
- **No use of `string.h`** The code includes `string.h`, but there are no string operations. The file is unnecessary and can be removed.

B.11 Recommendations

- **Correct SPMV Implementation:** Modify the `spmvCDS` function to include the input vector `x` and perform the correct sparse matrix-vector multiplication. The correct implementation should multiply corresponding elements of the diagonals by elements of the vector and accumulate results.
- **Implement Sparse Matrix Reading Directly into CDS:** Modify the `readMatrixFromMTX` function to read the sparse matrix directly into the CDS format, bypassing the creation of a dense matrix. This will significantly improve memory efficiency for large matrices.
- **Optimize Memory Allocation for Diagonals:** Explore more memory-efficient ways to store the diagonals in the CDS format. This can involve storing only the non-zero elements and their column indices within each diagonal.
- **Enhance Error Handling:** Add more comprehensive error handling to the file I/O operations and other parts of the code to make it more robust.
- **Remove unnecessary Headers**: remove `string.h` since there are no functions from the library being used.

B.12 Conclusion

The provided C code implements a basic version of Compressed Diagonal Storage (CDS) and Sparse Matrix-Vector Multiplication (SpMV). However, there are significant opportunities for improvement in terms of memory efficiency and correctness of the SpMV implementation. By addressing the identified issues and implementing the recommended changes, the code can be made more efficient and robust for handling large sparse matrices.

Appendix C

C Code Explanation: Coordinate (COO) Storage and SpMV

C.1 Introduction

This document explains a C code implementation for representing and manipulating sparse matrices using the Coordinate (COO) format. The code includes functions for creating a COO matrix, reading a matrix from the Matrix Market (.mtx) format directly into COO, performing Sparse Matrix-Vector Multiplication (SpMV) using the COO format, and handling memory allocation and deallocation. The COO format is chosen for its simplicity and ease of construction, although it's generally not the most performant for SpMV.

C.2 Header Files

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
```

Listing C.1: Header Files

- `stdio.h`: Standard input/output functions.
- `stdlib.h`: General utility functions, including memory allocation.
- `string.h`: String manipulation functions (used in parsing .mtx files).

- `time.h`: For timing execution.

C.3 COO Format Explained

The Coordinate (COO) format is one of the simplest ways to represent a sparse matrix. It stores the sparse matrix as a list of tuples, where each tuple represents a non-zero entry in the matrix. Each tuple typically contains:

- `row`: The row index of the non-zero entry.
- `col`: The column index of the non-zero entry.
- `value`: The value of the non-zero entry.

The COO format is easy to construct, but it's not the most efficient format for SpMV because accessing the elements is not cache-friendly. Other formats, like CSR (Compressed Sparse Row), are generally preferred for SpMV performance.

C.4 `COOMatrix` Structure

```c
typedef struct {
    int num_rows; // Number of rows in the matrix
    int num_cols; // Number of columns in the matrix
    int nnz; // Number of non-zero elements
    int *row_indices; // Array of row indices for non-zero elements
    int *col_indices; // Array of column indices for non-zero elements
    double *values; // Array of values for non-zero elements
} COOMatrix;
```

Listing C.2: `COOMatrix` Structure

- Defines the structure for storing a sparse matrix in the COO format.
- **num_rows**: Number of rows in the matrix.
- **num_cols**: Number of columns in the matrix.
- **nnz**: Number of non-zero elements in the matrix.
- **row_indices**: Integer array storing the row indices of the non-zero elements.

- **col_indices**: Integer array storing the column indices of the non-zero elements.
- **values**: Double array storing the values of the non-zero elements. Using 'double' provides more general applicability.

C.5 createCOOMatrix Function

```
COOMatrix* createCOOMatrix(int num_rows, int num_cols, int nnz) {
COOMatrix* coo = (COOMatrix*)malloc(sizeof(COOMatrix));
if (coo == NULL) {
fprintf(stderr, "Memory allocation failed for COOMatrix.\n");
exit(EXIT_FAILURE);
}

coo->num_rows = num_rows;
coo->num_cols = num_cols;
coo->nnz = nnz;

coo->row_indices = (int*)malloc(nnz * sizeof(int));
coo->col_indices = (int*)malloc(nnz * sizeof(int));
coo->values = (double*)malloc(nnz * sizeof(double));

if (coo->row_indices == NULL || coo->col_indices == NULL || coo->values == NULL)
    {
     fprintf(stderr, "Memory allocation failed for COO data arrays.\n");
     free(coo->row_indices);
     free(coo->col_indices);
     free(coo->values);
     free(coo);
     exit(EXIT_FAILURE);
}

return coo;


}
```

Listing C.3: `createCOOMatrix` Function

- Allocates memory for a `COOMatrix` structure and its constituent arrays.

- Takes the number of rows, number of columns, and the number of non-zero elements as input.
- Performs error checking after each memory allocation. If an allocation fails, it prints an error message, frees any previously allocated memory, and exits the program.
- Returns a pointer to the newly created and initialized `COOMatrix` structure.

C.6 `freeCOOMatrix` Function

```
void freeCOOMatrix(COOMatrix* coo) {
if (coo) {
free(coo->row_indices);
free(coo->col_indices);
free(coo->values);
free(coo);
}
}
```

Listing C.4: `freeCOOMatrix` Function

- Deallocates the memory occupied by a `COOMatrix` structure.
- Takes a pointer to the `COOMatrix` structure as input.
- Checks for `NULL` pointer before freeing.
- Frees the memory associated with the `row_indices`, `col_indices`, `values` arrays, and finally the `COOMatrix` structure.

C.7 `readMatrixFromMTXtoCOO` Function

```
COOMatrix* readMatrixFromMTXtoCOO(const char* filename) {
FILE* fp = fopen(filename, "r");
if (!fp) {
fprintf(stderr, "Error opening file: %s\n", filename);
return NULL;
}

```

```
char line[256];
int rows, cols, nnz;

// Skip comments
while (fgets(line, sizeof(line), fp)) {
    if (line[0] != '%') {
        break;
    }
}

// Read dimensions and number of non-zeros
if (sscanf(line, "%d %d %d", &rows, &cols, &nnz) != 3) {
    fprintf(stderr, "Error reading matrix dimensions and nnz.\n");
    fclose(fp);
    return NULL;
}

COOMatrix* coo = createCOOMatrix(rows, cols, nnz);
if (!coo) {
    fclose(fp);
    return NULL;
}

// Read the entries
for (int i = 0; i < nnz; i++) {
    int row, col;
    double value;
    if (fgets(line, sizeof(line), fp) == NULL) {
        fprintf(stderr, "Error reading matrix entry.\n");
        freeCOOMatrix(coo);
        fclose(fp);
        return NULL;
    }
    if (sscanf(line, "%d %d %lf", &row, &col, &value) != 3) {
        fprintf(stderr, "Error parsing matrix entry.\n");
        freeCOOMatrix(coo);
        fclose(fp);
        return NULL;
    }
    coo->row_indices[i] = row - 1; // 1-based indexing in MTX
    coo->col_indices[i] = col - 1;
    coo->values[i] = value;
}

fclose(fp);
return coo;
}
```

Listing C.5: `readMatrixFromMTXtoCOO` Function

- Reads a sparse matrix from a file in the Matrix Market (.mtx) format directly into the COO format.
- Takes the filename as input.
- Opens the file, skips header lines, and reads the matrix dimensions and the number of non-zero entries.
- Allocates memory for the `COOMatrix` structure using `createCOOMatrix`.
- Reads the non-zero entries from the file, storing the row and column indices, and the value in the corresponding arrays.
- Includes error checking and memory management.
- Returns a pointer to the created `COOMatrix` or `NULL` if an error occurred.

C.8 `spmvCOO` Function

```
void spmvCOO(const COOMatrix* coo, const double* x, double* y) {
int nnz = coo->nnz;
int num_rows = coo->num_rows;

// Initialize y to zero
for (int i = 0; i < num_rows; i++) {
    y[i] = 0.0;
}

// Perform the multiplication
for (int i = 0; i < nnz; i++) {
    int row = coo->row_indices[i];
    int col = coo->col_indices[i];
    double value = coo->values[i];
    y[row] += value * x[col];
}
}
```

Listing C.6: `spmvCOO` Function

- Performs Sparse Matrix-Vector Multiplication (SpMV) using the COO format.
- Takes a pointer to the `COOMatrix` structure, a pointer to the input vector `x`, and a pointer to the output vector `y` as input.
- Initializes the output vector `y` to zero.
- Iterates through the non-zero elements stored in the `COOMatrix`.
- Multiplies the value of each non-zero element by the corresponding element of the input vector `x` and adds it to the appropriate element of the output vector `y`.

C.9 main Function

```
int main(int argc, char* argv[]) {
char* filename = "matrixDataset/bfw398a.mtx";
if (argc > 1) {
filename = argv[1];
}

COOMatrix* coo_matrix = readMatrixFromMTXtoCOO(filename);

if (!coo_matrix) {
    return 1;
}

int n = coo_matrix->num_cols; // Assuming square matrix for vector size
double* x = (double*)malloc(n * sizeof(double));
double* y = (double*)malloc(n * sizeof(double));

if (!x || !y) {
    fprintf(stderr, "Memory allocation failed for vectors.\n");
    freeCOOMatrix(coo_matrix);
    free(x);
    free(y);
    return 1;
}

// Initialize x (e.g., with all 1s)
for (int i = 0; i < n; i++) {
    x[i] = 1.0;
}

clock_t start_time = clock();
spmvCOO(coo_matrix, x, y);
clock_t end_time = clock();

double elapsed_time = (double)(end_time - start_time) / CLOCKS_PER_SEC;

printf("Time taken for SpMV (COO): %f seconds\n", elapsed_time);

free(x);
free(y);
freeCOOMatrix(coo_matrix);

return 0;
}
```

Listing C.7: `main` Function

- Reads the matrix data from a file, performs SpMV, and measures the execution time.
- Reads the filename from command-line arguments, if provided.
- Calls `readMatrixFromMTXtoCOO` to read the matrix.
- Allocates memory for the input and output vectors.
- Initializes the input vector (in this case, to all 1s).
- Calls `spmvCOO` to perform SpMV.

- Measures the execution time.
- Deallocates all dynamically allocated memory.

C.10 Key Observations and Potential Improvements

- **COO is Simple, but Not Optimized for SpMV:** COO is great for building a sparse matrix, but its performance for SpMV is generally worse than CSR or CSC formats. The random access pattern to the `values` array results in poor cache utilization.
- **Consider Converting to CSR/CSC for SpMV:** A common strategy is to build the matrix in COO, and then convert it to CSR (Compressed Sparse Row) or CSC (Compressed Sparse Column) format for faster SpMV operations.
- **Use of Doubles:** Using `double` for the matrix values makes the code more generally applicable than using `int`.
- **Error Handling:** The code includes basic error handling, but it can be further improved.
- **No explicit sorting:** The COO format doesn't inherently require sorting by row or column indices, but sorting can sometimes improve performance if the matrix is constructed randomly. However, more often than not, sorting will be a performance loss.

C.11 Recommendations

- **Implement COO to CSR/CSC Conversion:** Implement functions to convert the COO matrix to CSR or CSC format. Then, implement SpMV using CSR or CSC to compare performance. This is the most crucial optimization.
- **Explore Different Vector Initialization:** Experiment with different initialization values for the input vector `x` to see how it affects performance.
- **Larger Test Matrices:** Use larger sparse matrices from the Matrix Market to properly benchmark the performance of the code.
- **Benchmarking:** Conduct rigorous benchmarking by running the code multiple times and calculating average execution times.

C.12 Example Usage

To compile and run the code:

```
gcc -o coo_spmv coo_spmv.c
./coo_spmv matrixDataset/bfw398a.mtx
```

(Replace `matrixDataset/bfw398a.mtx` with the path to your Matrix Market file). You will need to create the matrixDataset folder and place the bfw398a.mtx file inside of it. You can download the file from the Matrix Market website.

C.13 Conclusion

The provided C code implements the COO format and SpMV. While functional, the COO format is primarily useful for constructing sparse matrices. For optimal SpMV performance, consider converting to CSR or CSC after construction. By implementing these recommendations, you can significantly improve the efficiency of your sparse matrix computations.

Appendix D

C Code Explanation: Compressed Sparse Row (CSR) Storage and SpMV

D.1 Introduction

This document details a C code implementation for representing and manipulating sparse matrices using the Compressed Sparse Row (CSR) format. The code includes functions for creating a CSR matrix, reading a matrix from the Matrix Market (.mtx) format directly into CSR, performing Sparse Matrix-Vector Multiplication (SpMV) using the CSR format, and handling memory allocation and deallocation. The CSR format is a standard choice for SpMV due to its efficient memory layout and good performance characteristics.

D.2 Header Files

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
```

Listing D.1: Header Files

- `stdio.h`: Standard input/output functions.
- `stdlib.h`: General utility functions, including memory allocation.

- `string.h`: String manipulation functions (used in parsing .mtx files).
- `time.h`: For timing execution.

D.3 CSR Format Explained

The Compressed Sparse Row (CSR) format is a widely used representation for sparse matrices. It stores a sparse matrix using three arrays:

- `row_ptr`: An integer array of length `num_rows + 1`. `row_ptr[i]` stores the index of the first non-zero element in row `i` in the `col_ind` and `values` arrays. The last element, `row_ptr[num_rows]`, stores the total number of non-zero elements (nnz).
- `col_ind`: An integer array of length `nnz` storing the column indices of the non-zero elements.
- `values`: A double array of length `nnz` storing the values of the non-zero elements.

CSR is efficient for row-wise operations, such as SpMV, because it provides direct access to the non-zero elements in each row. It's also memory-efficient, as it only stores the non-zero elements.

D.4 `CSRMatrix` Structure

```
typedef struct {
    int num_rows;
    int num_cols;
    int nnz;
    int *row_ptr;
    int *col_ind;
    double *values;
} CSRMatrix;
```

Listing D.2: `CSRMatrix` Structure

- Defines the structure for storing a sparse matrix in the CSR format.
- `num_rows`: Number of rows in the matrix.

- `num_cols`: Number of columns in the matrix.
- `nnz`: Number of non-zero elements in the matrix.
- `row_ptr`: Integer array storing the row pointers.
- `col_ind`: Integer array storing the column indices.
- `values`: Double array storing the values.

D.5 `createCSRMatrix` Function

```
CSRMatrix* createCSRMatrix(int num_rows, int num_cols, int nnz) {
CSRMatrix* csr = (CSRMatrix*)malloc(sizeof(CSRMatrix));
if (csr == NULL) {
fprintf(stderr, "Memory allocation failed for CSRMatrix.\n");
exit(EXIT_FAILURE);
}

csr->num_rows = num_rows;
csr->num_cols = num_cols;
csr->nnz = nnz;

csr->row_ptr = (int*)malloc((num_rows + 1) * sizeof(int));
csr->col_ind = (int*)malloc(nnz * sizeof(int));
csr->values = (double*)malloc(nnz * sizeof(double));

if (csr->row_ptr == NULL || csr->col_ind == NULL || csr->values == NULL) {
    fprintf(stderr, "Memory allocation failed for CSR data arrays.\n");
    free(csr->row_ptr);
    free(csr->col_ind);
    free(csr->values);
    free(csr);
    exit(EXIT_FAILURE);
}

return csr;
}
```

Listing D.3: `createCSRMatrix` Function

- Allocates memory for a `CSRMatrix` structure and its constituent arrays.
- Takes the number of rows, number of columns, and the number of non-zero elements as input.
- Performs error checking after each memory allocation.
- Returns a pointer to the newly created and initialized `CSRMatrix` structure.

D.6 `freeCSRMatrix` Function

```
void freeCSRMatrix(CSRMatrix* csr) {
if (csr) {
free(csr->row_ptr);
free(csr->col_ind);
free(csr->values);
free(csr);
}
}
```

Listing D.4: `freeCSRMatrix` Function

- Deallocates the memory occupied by a `CSRMatrix` structure.
- Takes a pointer to the `CSRMatrix` structure as input.
- Checks for `NULL` pointer before freeing.

D.7 `readMatrixFromMTXtoCSR` Function

```
CSRMatrix* readMatrixFromMTXtoCSR(const char* filename) {
FILE* fp = fopen(filename, "r");
if (!fp) {
fprintf(stderr, "Error opening file: %s\n", filename);
return NULL;
}

char line[256];
int rows, cols, nnz;

```

```c
// Skip comments
while (fgets(line, sizeof(line), fp)) {
    if (line[0] != '%') {
        break;
    }
}

// Read dimensions and number of non-zeros
if (sscanf(line, "%d %d %d", &rows, &cols, &nnz) != 3) {
    fprintf(stderr, "Error reading matrix dimensions and nnz.\n");
    fclose(fp);
    return NULL;
}

CSRMatrix* csr = createCSRMatrix(rows, cols, nnz);
if (!csr) {
    fclose(fp);
    return NULL;
}

// Initialize row_ptr (crucial step)
for (int i = 0; i <= rows; i++) {
    csr->row_ptr[i] = 0;
}

// Count non-zeros per row
int row, col;
double value;
while (fgets(line, sizeof(line), fp)) {
  if (sscanf(line, "%d %d %lf", &row, &col, &value) == 3) {
    csr->row_ptr[row]++;  //Increment the number of nonzeros in the appropriate
    row. MTX format is 1 indexed.
  }
}

//Cumulative sum of row_ptr
csr->row_ptr[0] = 0;
for (int i = 1; i <= rows; i++) {
    csr->row_ptr[i] += csr->row_ptr[i - 1];
}

// Reset file pointer
fseek(fp, 0, SEEK_SET);
while (fgets(line, sizeof(line), fp)) {
  if (line[0] != '%') {
      break;
  }
}

//Reread dimensions
fgets(line, sizeof(line), fp);
sscanf(line, "%d %d %d", &rows, &cols, &nnz);

// Fill col_ind and values arrays
int *temp_row_ptr = (int *)malloc((rows+1) * sizeof(int));

//Copy values
for(int i = 0; i <= rows; i++){
    temp_row_ptr[i] = csr->row_ptr[i];
}

while (fgets(line, sizeof(line), fp)) {
  if (sscanf(line, "%d %d %lf", &row, &col, &value) == 3) {
    int index = temp_row_ptr[row]++; // find index
    csr->col_ind[index-1] = col - 1; // set column index
    csr->values[index-1] = value; // set value
  }
}

free(temp_row_ptr);
fclose(fp);
return csr;
}
```

Listing D.5: `readMatrixFromMTXtoCSR` Function

- Reads a sparse matrix from a file in the Matrix Market (.mtx) format directly into the CSR format.

- Takes the filename as input.
- Opens the file, skips header lines, and reads the matrix dimensions and the number of non-zero entries.
- Allocates memory for the `CSRMatrix` structure.
- Initializes the `row_ptr` array. This involves counting the number of non-zero elements in each row, then performing a cumulative sum to get the correct offsets. This is crucial for the CSR format.
- Reads the non-zero entries from the file, storing the column indices and values in the corresponding arrays.
- Includes error checking and memory management.
- Returns a pointer to the created `CSRMatrix` or `NULL` if an error occurred.

D.8 spmvCSR Function

```
void spmvCSR(const CSRMatrix* csr, const double* x, double* y) {
int num_rows = csr->num_rows;

// Initialize y to zero
for (int i = 0; i < num_rows; i++) {
    y[i] = 0.0;
}

// Perform the multiplication
for (int i = 0; i < num_rows; i++) {
    for (int j = csr->row_ptr[i]; j < csr->row_ptr[i + 1]; j++) {
        int col = csr->col_ind[j];
        double value = csr->values[j];
        y[i] += value * x[col];
    }
}
}
```

Listing D.6: spmvCSR Function

- Performs Sparse Matrix-Vector Multiplication (SpMV) using the CSR format.
- Takes a pointer to the `CSRMatrix` structure, a pointer to the input vector `x`, and a pointer to the output vector `y` as input.
- Initializes the output vector `y` to zero.
- Iterates through the rows of the matrix.
- For each row, iterates through the non-zero elements using the `row_ptr` array to determine the start and end indices in the `col_ind` and `values` arrays.
- Multiplies the value of each non-zero element by the corresponding element of the input vector `x` and adds it to the appropriate element of the output vector `y`.

D.9 `main` Function

```c
int main(int argc, char* argv[]) {
char* filename = "matrixDataset/bfw398a.mtx";
if (argc > 1) {
filename = argv[1];
}

CSRMatrix* csr_matrix = readMatrixFromMTXtoCSR(filename);

if (!csr_matrix) {
    return 1;
}

int n = csr_matrix->num_cols; // Assuming square matrix for vector size
double* x = (double*)malloc(n * sizeof(double));
double* y = (double*)malloc(n * sizeof(double));

if (!x || !y) {
    fprintf(stderr, "Memory allocation failed for vectors.\n");
    freeCSRMatrix(csr_matrix);
    free(x);
    free(y);
    return 1;
}

// Initialize x (e.g., with all 1s)
for (int i = 0; i < n; i++) {
    x[i] = 1.0;
}

clock_t start_time = clock();
spmvCSR(csr_matrix, x, y);
clock_t end_time = clock();

double elapsed_time = (double)(end_time - start_time) / CLOCKS_PER_SEC;

printf("Time taken for SpMV (CSR): %f seconds\n", elapsed_time);

free(x);
free(y);
freeCSRMatrix(csr_matrix);

return 0;
}
```

Listing D.7: `main` Function

- Reads the matrix data from a file, performs SpMV, and measures the execution time.
- Reads the filename from command-line arguments, if provided.
- Calls `readMatrixFromMTXtoCSR` to read the matrix.
- Allocates memory for the input and output vectors.
- Initializes the input vector (in this case, to all 1s).
- Calls `spmvCSR` to perform SpMV.
- Measures the execution time.
- Deallocates all dynamically allocated memory.

D.10 Key Observations and Potential Improvements

- **CSR is Well-Suited for SpMV:** CSR is a good choice for SpMV, offering a balance between memory efficiency and performance. It is usually better than COO, but potentially worse than block compressed formats for certain matrices.
- **Correct CSR Construction is Critical:** The most complex part is correctly constructing the `row_ptr` array. This implementation handles this carefully, performing two passes through the data to ensure the offsets are correct.
- **Use of Doubles:** Using `double` for the matrix values provides more general applicability.
- **Memory Allocation:** The current implementation allocates memory for all nonzero elements. If memory is an issue, the code may benefit from checking for duplicate rows and columns within the MTX file.
- **Cache Blocking:** For larger matrices, consider implementing cache blocking techniques to improve memory access patterns and SpMV performance.

D.11 Recommendations

- **Compare with COO:** Build the matrix in COO, then convert to CSR and benchmark against the plain COO implementation to see the speedup.
- **Larger Test Matrices:** Use larger sparse matrices from the Matrix Market to properly benchmark the performance.
- **Benchmarking:** Conduct rigorous benchmarking by running the code multiple times and calculating average execution times.
- **Explore Block Compressed Row (BCR) Formats:** For certain matrices (e.g., those arising from finite element methods), BCR can offer even better performance than CSR.

D.12 Example Usage

To compile and run the code:

```
gcc -o csr_spmv csr_spmv.c
./csr_spmv matrixDataset/bfw398a.mtx
```

(Replace `matrixDataset/bfw398a.mtx` with the path to your Matrix Market file).

D.13 Conclusion

The provided C code implements the CSR format and SpMV. The CSR format is a solid choice for this operation, offering good performance and memory efficiency. By implementing the recommendations, you can further optimize the code for your specific application.

Appendix E

C Code Explanation: Compressed Sparse Column (CSC) Storage and SpMV

E.1 Introduction

This document details a C code implementation for representing and manipulating sparse matrices using the Compressed Sparse Column (CSC) format. The code includes functions for creating a CSC matrix, reading a matrix from the Matrix Market (.mtx) format directly into CSC, performing Sparse Matrix-Vector Multiplication (SpMV) using the CSC format, and handling memory allocation and deallocation. The CSC format is efficient for column-wise operations, and in some SpMV implementations.

E.2 Header Files

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
```

Listing E.1: Header Files

- `stdio.h`: Standard input/output functions.
- `stdlib.h`: General utility functions, including memory allocation.

- `string.h`: String manipulation functions (used in parsing .mtx files).
- `time.h`: For timing execution.

E.3 CSC Format Explained

The Compressed Sparse Column (CSC) format is another standard representation for sparse matrices. It's similar to CSR, but it stores information column-wise instead of row-wise. It uses three arrays:

- `col_ptr`: An integer array of length `num_cols + 1`. `col_ptr[i]` stores the index of the first non-zero element in column `i` in the `row_ind` and `values` arrays. The last element, `col_ptr[num_cols]`, stores the total number of non-zero elements (nnz).
- `row_ind`: An integer array of length `nnz` storing the row indices of the non-zero elements.
- `values`: A double array of length `nnz` storing the values of the non-zero elements.

CSC is well-suited for column-wise operations. While CSR is generally preferred for basic SpMV, CSC can be useful in specialized SpMV algorithms or when performing other operations that benefit from column-wise access.

E.4 `CSCMatrix` Structure

```
typedef struct {
    int num_rows;
    int num_cols;
    int nnz;
    int *col_ptr;
    int *row_ind;
    double *values;
} CSCMatrix;
```

Listing E.2: `CSCMatrix` Structure

- Defines the structure for storing a sparse matrix in the CSC format.
- `num_rows`: Number of rows in the matrix.

- `num_cols`: Number of columns in the matrix.
- `nnz`: Number of non-zero elements in the matrix.
- `col_ptr`: Integer array storing the column pointers.
- `row_ind`: Integer array storing the row indices.
- `values`: Double array storing the values.

E.5 `createCSCMatrix` Function

```c
CSCMatrix* createCSCMatrix(int num_rows, int num_cols, int nnz) {
    CSCMatrix* csc = (CSCMatrix*)malloc(sizeof(CSCMatrix));
    if (csc == NULL) {
        fprintf(stderr, "Memory allocation failed for CSCMatrix.\n");
        exit(EXIT_FAILURE);
    }

    csc->num_rows = num_rows;
    csc->num_cols = num_cols;
    csc->nnz = nnz;

    csc->col_ptr = (int*)malloc((num_cols + 1) * sizeof(int));
    csc->row_ind = (int*)malloc(nnz * sizeof(int));
    csc->values = (double*)malloc(nnz * sizeof(double));

    if (csc->col_ptr == NULL || csc->row_ind == NULL || csc->values == NULL) {
        fprintf(stderr, "Memory allocation failed for CSC data arrays.\n");
        free(csc->col_ptr);
        free(csc->row_ind);
        free(csc->values);
        free(csc);
        exit(EXIT_FAILURE);
    }

    return csc;
}
```

Listing E.3: `createCSCMatrix` Function

- Allocates memory for a `CSCMatrix` structure and its constituent arrays.
- Takes the number of rows, number of columns, and the number of non-zero elements as input.
- Performs error checking after each memory allocation.
- Returns a pointer to the newly created and initialized `CSCMatrix` structure.

E.6 `freeCSCMatrix` Function

```c
void freeCSCMatrix(CSCMatrix* csc) {
    if (csc) {
        free(csc->col_ptr);
        free(csc->row_ind);
        free(csc->values);
        free(csc);
    }
}
```

Listing E.4: `freeCSCMatrix` Function

- Deallocates the memory occupied by a `CSCMatrix` structure.
- Takes a pointer to the `CSCMatrix` structure as input.
- Checks for `NULL` pointer before freeing.

E.7 `readMatrixFromMTXtoCSC` Function

```c
CSCMatrix* readMatrixFromMTXtoCSC(const char* filename) {
    FILE* fp = fopen(filename, "r");
    if (!fp) {
        fprintf(stderr, "Error opening file: %s\n", filename);
        return NULL;
    }

    char line[256];
    int rows, cols, nnz;

    // Skip comments
    while (fgets(line, sizeof(line), fp)) {
        if (line[0] != '%') {
            break;
        }
    }

    // Read dimensions and number of non-zeros
    if (sscanf(line, "%d %d %d", &rows, &cols, &nnz) != 3) {
        fprintf(stderr, "Error reading matrix dimensions and nnz.\n");
        fclose(fp);
        return NULL;
    }

    CSCMatrix* csc = createCSCMatrix(rows, cols, nnz);
    if (!csc) {
        fclose(fp);
        return NULL;
    }

    // Initialize col_ptr array (crucial step)
    for (int i = 0; i <= cols; i++) {
        csc->col_ptr[i] = 0;
    }

    // Count non-zeros per column
    int row, col;
    double value;
    while (fgets(line, sizeof(line), fp)) {
      if (sscanf(line, "%d %d %lf", &row, &col, &value) == 3) {
        csc->col_ptr[col]++;
      }
    }

    //Cumulative sum of col_ptr
    csc->col_ptr[0] = 0;
    for (int i = 1; i <= cols; i++) {
        csc->col_ptr[i] += csc->col_ptr[i - 1];
    }
```

```c

    // Reset file pointer
    fseek(fp, 0, SEEK_SET);
    while (fgets(line, sizeof(line), fp)) {
      if (line[0] != '%') {
          break;
      }
    }

    // Reread dimensions (necessary after rewind)
    fgets(line, sizeof(line), fp);
    sscanf(line, "%d %d %d", &rows, &cols, &nnz);

    // Fill row_ind and values arrays
    int *temp_col_ptr = (int *)malloc((cols+1) * sizeof(int));
    for(int i = 0; i <= cols; i++){
      temp_col_ptr[i] = csc->col_ptr[i];
    }


    while (fgets(line, sizeof(line), fp)) {
      if (sscanf(line, "%d %d %lf", &row, &col, &value) == 3) {
        int index = temp_col_ptr[col]++; // find index
        csc->row_ind[index-1] = row - 1; // set row index
        csc->values[index-1] = value; // set value
      }
    }

    free(temp_col_ptr);
    fclose(fp);
    return csc;
}
```

Listing E.5: `readMatrixFromMTXtoCSC` Function

- Reads a sparse matrix from a file in the Matrix Market (.mtx) format directly into the CSC format.
- Takes the filename as input.

- Opens the file, skips header lines, and reads the matrix dimensions and the number of non-zero entries.
- Allocates memory for the `CSCMatrix` structure.
- Initializes the `col_ptr` array by counting nonzero elements per column. This is very similar to the CSR initialization but operates on columns instead of rows.
- Reads the non-zero entries from the file, storing the row indices and values in the corresponding arrays.
- Includes error checking and memory management.
- Returns a pointer to the created `CSCMatrix` or `NULL` if an error occurred.

E.8 spmvCSC Function

```
void spmvCSC(const CSCMatrix* csc, const double* x, double* y) {
    int num_cols = csc->num_cols;
    int num_rows = csc->num_rows;

    // Initialize y to zero
    for (int i = 0; i < num_rows; i++) {
        y[i] = 0.0;
    }

    // Perform the multiplication
    for (int i = 0; i < num_cols; i++) {
        for (int j = csc->col_ptr[i]; j < csc->col_ptr[i + 1]; j++) {
            int row = csc->row_ind[j];
            double value = csc->values[j];
            y[row] += value * x[i];  // Access x[i] instead of x[col]
        }
    }
}
```

Listing E.6: `spmvCSC` Function

- Performs Sparse Matrix-Vector Multiplication (SpMV) using the CSC format.

- Takes a pointer to the `CSCMatrix` structure, a pointer to the input vector `x`, and a pointer to the output vector `y` as input.
- Initializes the output vector `y` to zero.
- Iterates through the columns of the matrix.
- For each column, iterates through the non-zero elements using the `col_ptr` array to determine the start and end indices in the `row_ind` and `values` arrays.
- Multiplies the value of each non-zero element by the corresponding element of the input vector `x` at index `i` (the column index), and adds it to the appropriate element of the output vector `y`.

E.9 `main` Function

```c
int main(int argc, char* argv[]) {
    char* filename = "matrixDataset/bfw398a.mtx";
    if (argc > 1) {
        filename = argv[1];
    }

    CSCMatrix* csc_matrix = readMatrixFromMTXtoCSC(filename);

    if (!csc_matrix) {
        return 1;
    }

    int n = csc_matrix->num_cols; // Assuming square matrix for vector size
    double* x = (double*)malloc(n * sizeof(double));
    double* y = (double*)malloc(csc_matrix->num_rows * sizeof(double)); //
    Correctly allocate based on the number of rows

    if (!x || !y) {
        fprintf(stderr, "Memory allocation failed for vectors.\n");
        freeCSCMatrix(csc_matrix);
        free(x);
        free(y);
        return 1;
    }

    // Initialize x (e.g., with all 1s)
    for (int i = 0; i < n; i++) {
        x[i] = 1.0;
    }

    clock_t start_time = clock();
    spmvCSC(csc_matrix, x, y);
    clock_t end_time = clock();

    double elapsed_time = (double)(end_time - start_time) / CLOCKS_PER_SEC;

    printf("Time taken for SpMV (CSC): %f seconds\n", elapsed_time);

    free(x);
    free(y);
    freeCSCMatrix(csc_matrix);

    return 0;
}
```

Listing E.7: `main` Function

- Reads the matrix data from a file, performs SpMV, and measures the execution time.
- Reads the filename from command-line arguments, if provided.
- Calls `readMatrixFromMTXtoCSC` to read the matrix.
- Allocates memory for the input and output vectors.
- Initializes the input vector (in this case, to all 1s).
- Calls `spmvCSC` to perform SpMV.
- Measures the execution time.
- Deallocates all dynamically allocated memory.

E.10 Key Observations and Potential Improvements

- **CSC for SpMV:** While CSR is typically favored, CSC can be competitive. It depends on the matrix structure and memory access patterns.
- **Correct CSC Construction is Critical:** As with CSR, correctly initializing the `col_ptr` array is essential. The code performs the appropriate counting and cumulative sum operations.
- **Symmetry:** If your sparse matrix is symmetric, consider storing it in a format that takes advantage of symmetry to save memory.
- **Alternatives:** It would be beneficial to test a naive SpMV with CSC, alongside an alternative algorithm where the vectors are switched, and matrix multiplication is done with a transpose.

E.11 Recommendations

- **Compare with CSR:** Implement both CSR and CSC, and compare their performance on different sparse matrices.
- **Larger Test Matrices:** Use larger sparse matrices to properly benchmark the performance.
- **Benchmarking:** Conduct rigorous benchmarking by running the code multiple times and calculating average execution times.
- **Memory Access Pattern:** Investigate whether you can improve SpMV performance through techniques like loop unrolling or vectorization, considering the specific memory access patterns of the CSC format.

E.12 Example Usage

To compile and run the code:

```
gcc -o csc_spmv csc_spmv.c
./csc_spmv matrixDataset/bfw398a.mtx
```

(Replace `matrixDataset/bfw398a.mtx` with the path to your Matrix Market file).

E.13 Conclusion

The provided C code implements the CSC format and SpMV. CSC is a valid alternative to CSR, and the choice depends on the specific characteristics of your problem and the operations you need to perform. By experimenting with different matrices and benchmarking, you can determine which format is best suited for your needs.

Appendix F

C Code Explanation: Jagged Diagonal Storage (JDS) and SpMV

F.1 Introduction

This document explains a C code implementation for representing and manipulating sparse matrices using the Jagged Diagonal Storage (JDS) format. The code includes functions for creating a JDS matrix, converting a matrix from a dense representation (for simplicity), performing Sparse Matrix-Vector Multiplication (SpMV) using the JDS format, and handling memory allocation and deallocation. JDS aims to improve memory access patterns for parallel processing.

F.2 Header Files

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
```

Listing F.1: Header Files

- `stdio.h`: Standard input/output functions.
- `stdlib.h`: General utility functions, including memory allocation.
- `string.h`: String manipulation functions (not used here, but could be for extensions).

- `time.h`: For timing execution.

F.3 JDS Format Explained

Jagged Diagonal Storage (JDS) is a sparse matrix format designed to improve vectorization and parallelization during SpMV. It works by rearranging rows of the matrix based on the number of non-zero elements they contain and then storing the matrix in a diagonal-like structure. The rows are sorted in decreasing order of the number of non-zero elements.

The JDS format typically uses these arrays:

- `JD`: An array containing the non-zero values along the jagged diagonals.
- `COL`: An array storing the column indices corresponding to the values in `JD`.
- `PERM`: An array storing the permutation of the rows after sorting by the number of non-zeros. The `PERM` array allows you to map back to the original row indices.
- `DIAG`: An array indicating the start of each jagged diagonal in the `JD` and `COL` arrays.

JDS can be advantageous on vector processors or GPUs, where regular memory access patterns are crucial for performance. However, converting to JDS and the added complexity can sometimes outweigh the benefits.

F.4 `JDSMatrix` Structure

```c
typedef struct {
    int num_rows;
    int num_cols;
    int nnz;
    double *JD;       // Jagged Diagonal Values
    int *COL;         // Column Indices
    int *PERM;        // Row Permutation
    int *DIAG;        // Diagonal Pointers
    int max_row_nnz; //Maximum number of nonzeros in a row after permutation (
   length of the jagged diagonals)
} JDSMatrix;
```

Listing F.2: `JDSMatrix` Structure

- Defines the structure for storing a sparse matrix in the JDS format.
- `num_rows`: Number of rows in the matrix.
- `num_cols`: Number of columns in the matrix.
- `nnz`: Number of non-zero elements in the matrix.
- `JD`: Double array storing the jagged diagonal values.
- `COL`: Integer array storing the column indices.
- `PERM`: Integer array storing the row permutation.
- `DIAG`: Integer array storing the diagonal pointers.
- `max_row_nnz`: Maximum number of nonzeros in a row after permutation, determining the jagged diagonal length.

F.5 `createJDSMatrix` Function

```c
JDSMatrix* createJDSMatrix(int num_rows, int num_cols, int nnz, int max_row_nnz)
    {
    JDSMatrix* jds = (JDSMatrix*)malloc(sizeof(JDSMatrix));
    if (jds == NULL) {
        fprintf(stderr, "Memory allocation failed for JDSMatrix.\n");
        exit(EXIT_FAILURE);
    }

    jds->num_rows = num_rows;
    jds->num_cols = num_cols;
    jds->nnz = nnz;
    jds->max_row_nnz = max_row_nnz;

    jds->JD = (double*)malloc(nnz * sizeof(double));
    jds->COL = (int*)malloc(nnz * sizeof(int));
    jds->PERM = (int*)malloc(num_rows * sizeof(int));
    jds->DIAG = (int*)malloc((max_row_nnz+1) * sizeof(int)); //one more than max
    diagonals

    if (jds->JD == NULL || jds->COL == NULL || jds->PERM == NULL || jds->DIAG ==
    NULL) {
        fprintf(stderr, "Memory allocation failed for JDS data arrays.\n");
        free(jds->JD);
        free(jds->COL);
        free(jds->PERM);
        free(jds->DIAG);
        free(jds);
        exit(EXIT_FAILURE);
    }

    return jds;
}
```

Listing F.3: `createJDSMatrix` Function

- Allocates memory for a `JDSMatrix` structure and its constituent arrays.
- Takes the number of rows, number of columns, the number of non-zero elements, and the maximum non-zeros in a row as input.
- Performs error checking after each memory allocation.
- Returns a pointer to the newly created and initialized `JDSMatrix` structure.

F.6 `freeJDSMatrix` Function

```c
void freeJDSMatrix(JDSMatrix* jds) {
    if (jds) {
        free(jds->JD);
        free(jds->COL);
        free(jds->PERM);
        free(jds->DIAG);
        free(jds);
    }
}
```

Listing F.4: `freeJDSMatrix` Function

- Deallocates the memory occupied by a `JDSMatrix` structure.

- Takes a pointer to the JDSMatrix structure as input.
- Checks for NULL pointer before freeing.

F.7 convertToJDS Function

```
JDSMatrix* convertToJDS(double** A, int num_rows, int num_cols) {
    int nnz = 0;
    int *row_nnz = (int*)malloc(num_rows * sizeof(int)); // Number of non-zeros
    in each row
    if(row_nnz == NULL){
      fprintf(stderr, "Error Allocating memory for row_nnz");
      exit(EXIT_FAILURE);
    }

    // Count non-zero elements in each row
    for (int i = 0; i < num_rows; i++) {
        row_nnz[i] = 0;
        for (int j = 0; j < num_cols; j++) {
            if (A[i][j] != 0.0) {
                row_nnz[i]++;
                nnz++;
            }
        }
    }

    //Find max row nnz
    int max_row_nnz = 0;
    for (int i = 0; i < num_rows; i++) {
      if(row_nnz[i] > max_row_nnz){
        max_row_nnz = row_nnz[i];
      }
    }

    // Create JDS matrix
    JDSMatrix* jds = createJDSMatrix(num_rows, num_cols, nnz, max_row_nnz);

    // Initialize permutation and sort
    for (int i = 0; i < num_rows; i++) {
        jds->PERM[i] = i;
    }

    // Simple bubble sort (for demonstration).  Use qsort for large matrices.
    for (int i = 0; i < num_rows - 1; i++) {
        for (int j = 0; j < num_rows - i - 1; j++) {
            if (row_nnz[j] < row_nnz[j + 1]) {
                // Swap row_nnz
                int temp_nnz = row_nnz[j];
                row_nnz[j] = row_nnz[j + 1];
                row_nnz[j + 1] = temp_nnz;

                // Swap PERM
                int temp_perm = jds->PERM[j];
                jds->PERM[j] = jds->PERM[j + 1];
                jds->PERM[j + 1] = temp_perm;
            }
        }
    }

    // Fill JDS and COL arrays
    int jd_index = 0;
    for (int diag = 0; diag < jds->max_row_nnz; diag++) {
        jds->DIAG[diag] = jd_index;
        for (int i = 0; i < num_rows; i++) {
            int row = jds->PERM[i]; // Permuted Row

            int k = 0; // Non-zero element counter for the row
            for (int j = 0; j < num_cols; j++) {
                if (A[row][j] != 0.0) {
                    if (k == diag) {
                        jds->JD[jd_index] = A[row][j];
                        jds->COL[jd_index] = j;
                        jd_index++;
                        break; // Move to the next row
                    }
                    k++;
                }
            }
        }
    }
    jds->DIAG[jds->max_row_nnz] = jd_index; //Final Pointer

    free(row_nnz);
    return jds;
}
```

Listing F.5: `convertToJDS` Function

- Converts a dense matrix to the JDS format. **Note:** This function assumes you have a dense matrix as input. A more realistic scenario would be to convert from COO or CSR.
- Takes the dense matrix `A`, number of rows, and number of columns as input.
- Counts the number of non-zero elements in each row and the total number of non-zero elements.
- Creates the `JDSMatrix` structure.
- Initializes the permutation array `PERM` and sorts the rows based on the number of non-zero elements using a simple bubble sort. **Important:** For large matrices, use `qsort` for much better performance.
- Fills the `JD` and `COL` arrays based on the sorted rows and the jagged diagonal structure.
- Returns a pointer to the created `JDSMatrix`.

F.8 spmvJDS Function

```
void spmvJDS(const JDSMatrix* jds, const double* x, double* y) {
    int num_rows = jds->num_rows;
    int num_cols = jds->num_cols;

    // Initialize y to zero
    for (int i = 0; i < num_rows; i++) {
        y[i] = 0.0;
    }

    // Perform the multiplication
    for (int i = 0; i < num_rows; i++) {
        int orig_row = jds->PERM[i]; // Original row index

        double row_sum = 0.0;
        int k = 0;  // Track non-zero element index

        for (int j = 0; j < num_cols; j++) {
            // Find all the non-zero entries for a row.
            int diag_start = 0;
            int diag_end = 0;

            // The DIAG array holds the starting and ending index. Since we sort
    the matrix, we need to
            // make sure to perform the right calculations.
            for(int d = 0; d < jds->max_row_nnz; d++) {
                if (jds->DIAG[d] <= k && k < jds->DIAG[d+1] ) {
                     diag_start = d;
                     diag_end = d+1;
                     break;
                 }
            }
            for(int diag = diag_start; diag < diag_end; diag++) {
              int orig_row_sum = jds->DIAG[diag];
              if(jds->COL[orig_row_sum] == j) {
                row_sum += jds->JD[orig_row_sum] * x[j];
                k++;
              }
            }
        }
        y[orig_row] = row_sum;  // Store the original row index,
    }
}
```

Listing F.6: `spmvJDS` Function

- Performs Sparse Matrix-Vector Multiplication (SpMV) using the JDS format.
- Takes a pointer to the `JDSMatrix` structure, a pointer to the input vector `x`, and a pointer to the output vector `y` as input.

- Initializes the output vector y to zero.
- Iterates through the *permuted* rows of the matrix.
- Retrieves the *original* row index using the PERM array.
- Multiplies the value of each non-zero element by the corresponding element of the input vector x and adds it to the appropriate element of the output vector y. The row index is mapped back to the original index before storing the result.

F.9 main Function

```
int main(int argc, char* argv[]) {
    int num_rows = 5;
    int num_cols = 5;

    // Example dense matrix (for simplicity)
    double** A = (double**)malloc(num_rows * sizeof(double*));
    for (int i = 0; i < num_rows; i++) {
        A[i] = (double*)malloc(num_cols * sizeof(double));
        for (int j = 0; j < num_cols; j++) {
            A[i][j] = 0.0;  // Initialize to zero
        }
    }

    // Sample data - sparse
    A[0][0] = 1.0;
    A[0][2] = 2.0;
    A[1][1] = 3.0;
    A[1][4] = 4.0;
    A[2][0] = 5.0;
    A[2][3] = 6.0;
    A[3][2] = 7.0;
    A[3][4] = 8.0;
    A[4][1] = 9.0;
    A[4][3] = 10.0;
    A[4][4] = 11.0;

    JDSMatrix* jds_matrix = convertToJDS(A, num_rows, num_cols);

    if (!jds_matrix) {
        //Free A
        for (int i = 0; i < num_rows; i++) {
            free(A[i]);
        }
        free(A);
        return 1;
    }

    double* x = (double*)malloc(num_cols * sizeof(double));
    double* y = (double*)malloc(num_rows * sizeof(double));

    if (!x || !y) {
        fprintf(stderr, "Memory allocation failed for vectors.\n");
        freeJDSMatrix(jds_matrix);
        //Free A
        for (int i = 0; i < num_rows; i++) {
            free(A[i]);
        }
        free(A);
        free(x);
        free(y);
        return 1;
    }

    // Initialize x (e.g., with all 1s)
    for (int i = 0; i < num_cols; i++) {
        x[i] = 1.0;
    }

    clock_t start_time = clock();
    spmvJDS(jds_matrix, x, y);
    clock_t end_time = clock();

    double elapsed_time = (double)(end_time - start_time) / CLOCKS_PER_SEC;

    printf("Time taken for SpMV (JDS): %f seconds\n", elapsed_time);

    free(x);
    free(y);
    freeJDSMatrix(jds_matrix);

    //Free A
    for (int i = 0; i < num_rows; i++) {
        free(A[i]);
    }
    free(A);

    return 0;
}
```

Listing F.7: `main` Function

- Sets up a small example matrix, converts it to JDS, performs SpMV, and measures the execution time. **Important:** This `main` function is *not* reading from a Matrix Market file. It's creating a small, hardcoded dense matrix for demonstration purposes.
- Allocates and initializes a small dense matrix `A`.
- Calls `convertToJDS` to convert the dense matrix to JDS format.
- Allocates memory for the input and output vectors.
- Initializes the input vector.
- Calls `spmvJDS` to perform SpMV.
- Measures the execution time.
- Deallocates all dynamically allocated memory.

F.10 Key Observations and Potential Improvements

- **Complexity:** JDS is more complex to implement and understand than COO, CSR, or CSC.
- **Dense Matrix Input:** The code currently converts from a *dense* matrix. This is highly unrealistic for large sparse matrices. The ‘convertToJDS‘ should take a COO or CSR representation as input.

- **Sorting:** The bubble sort is extremely inefficient for larger matrices. Replace it with ‘qsort‘.
- **Memory Access:** The ‘spmvJDS‘ function can be complex to optimize. The nested loops and indirect memory accesses can hinder performance.
- **No Matrix Market Support:** The code does not include reading matrices from the Matrix Market format directly into JDS. This is essential for testing with real-world datasets.

F.11 Recommendations

- **Convert from COO/CSR:** Modify the `convertToJDS` function to accept a COO or CSR matrix as input instead of a dense matrix.
- **Implement Matrix Market Reader:** Add a function to read a sparse matrix from a Matrix Market file and convert it directly to JDS format.
- **Use `qsort`** Replace the bubble sort with `qsort` for efficient row sorting.
- **Benchmark Thoroughly:** Compare the performance of JDS SpMV to CSR and CSC SpMV on various sparse matrices to determine its effectiveness. The benefits of JDS are highly dependent on the matrix structure and the target hardware architecture.
- **Explore Block JDS:** Consider block-based JDS variants, which can sometimes offer better performance.
- **Implement Alternatives:** Research alternative JDS algorithms, where, again, vector switches are performed, and matrix multiplication is done with a transpose.

F.12 Example Usage

To compile and run the code:

```
gcc -o jds_spmv jds_spmv.c
./jds_spmv
```

Important: This will run with the small, hardcoded dense matrix example. To make it useful, you need to:

- Implement the `readMatrixFromMTXtoJDS` function.
- Modify the `main` function to use that function and pass a Matrix Market file as a command-line argument.

F.13 Conclusion

The code provides a basic implementation of JDS and SpMV. However, it is crucial to address the limitations mentioned above, particularly the dense matrix input, the inefficient sorting algorithm, and the lack of Matrix Market support. JDS can be a valuable format, but its effectiveness depends heavily on the matrix structure and the target hardware. Thorough benchmarking is essential to determine if JDS offers a performance advantage over simpler formats like CSR.

Appendix G

C Code Explanation: Packed Diagonal Storage (PDS) and SpMV

G.1 Introduction

This document provides a C code implementation for representing sparse matrices using the Packed Diagonal Storage (PDS) [7] format and performing Sparse Matrix-Vector Multiplication (SpMV). PDS is designed to be memory efficient for matrices with a specific diagonal structure. It stores diagonals as contiguous vectors, packing them tightly to reduce memory usage.

G.2 Header Files

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <math.h> // For absolute value (abs) calculation
```

Listing G.1: Header Files

- `stdio.h`: Standard input/output functions.
- `stdlib.h`: General utility functions, including memory allocation.
- `string.h`: String manipulation functions (not used extensively but can be useful for extensions).

- `time.h`: For timing execution.
- `math.h`: For math functions like `abs()`.

G.3 PDS Format Explained

Packed Diagonal Storage (PDS) is particularly effective when a sparse matrix has a large number of non-zero elements concentrated along a few diagonals. Instead of storing the entire matrix, PDS stores only the non-zero diagonals.

The key components of the PDS format are:

- `num_diags`: The number of diagonals stored.
- `diag_offsets`: An array storing the offsets of each diagonal from the main diagonal. A positive offset indicates a diagonal above the main diagonal, while a negative offset indicates a diagonal below the main diagonal.
- `diag_values`: A 2D array where each row represents a diagonal. The length of each row depends on the offset. Elements outside the matrix bounds are typically padded with zeros or dummy values.

PDS is most beneficial when the matrix has a well-defined diagonal structure and relatively few non-zero diagonals. In such cases, it can significantly reduce memory usage compared to dense storage or even COO and CSR.

G.4 `PDSMatrix` Structure

```
typedef struct {
    int num_rows;
    int num_cols;
    int num_diags;
    int *diag_offsets;
    double **diag_values;
} PDSMatrix;
```

Listing G.2: `PDSMatrix` Structure

- Defines the structure for storing a sparse matrix in the PDS format.

- `num_rows`: Number of rows in the matrix.
- `num_cols`: Number of columns in the matrix.
- `num_diags`: The number of diagonals stored in PDS.
- `diag_offsets`: An array of integers storing the offsets of each diagonal.
- `diag_values`: A 2D array of doubles storing the values of the diagonals.

G.5 createPDSMatrix Function

```c
PDSMatrix* createPDSMatrix(int num_rows, int num_cols, int num_diags) {
    PDSMatrix* pds = (PDSMatrix*)malloc(sizeof(PDSMatrix));
    if (pds == NULL) {
        fprintf(stderr, "Memory allocation failed for PDSMatrix.\n");
        exit(EXIT_FAILURE);
    }

    pds->num_rows = num_rows;
    pds->num_cols = num_cols;
    pds->num_diags = num_diags;

    pds->diag_offsets = (int*)malloc(num_diags * sizeof(int));
    if (pds->diag_offsets == NULL) {
        fprintf(stderr, "Memory allocation failed for diag_offsets.\n");
        free(pds);
        exit(EXIT_FAILURE);
    }

    pds->diag_values = (double**)malloc(num_diags * sizeof(double*));
    if (pds->diag_values == NULL) {
        fprintf(stderr, "Memory allocation failed for diag_values (rows).\n");
        free(pds->diag_offsets);
        free(pds);
        exit(EXIT_FAILURE);
    }

    return pds;
}
```

Listing G.3: `createPDSMatrix` Function

- Allocates memory for a `PDSMatrix` structure and the `diag_offsets` and `diag_values` arrays (the rows of the 2D ‘diag_values‘ array).
- Takes the number of rows, number of columns, and the number of diagonals to store as input.
- Includes error checking after memory allocations.
- Returns a pointer to the newly created `PDSMatrix`.

G.6 `freePDSMatrix` Function

```c
void freePDSMatrix(PDSMatrix* pds) {
    if (pds) {
        if (pds->diag_offsets) {
            free(pds->diag_offsets);
        }

        if (pds->diag_values) {
            for (int i = 0; i < pds->num_diags; i++) {
                free(pds->diag_values[i]); // Free each diagonal's values
            }
            free(pds->diag_values);        // Free the array of diagonal pointers
        }

        free(pds);
    }
}
```

Listing G.4: `freePDSMatrix` Function

- Deallocates the memory occupied by a `PDSMatrix` structure, including the diagonals stored in `diag_values`.
- Takes a pointer to the `PDSMatrix` structure as input.
- Includes checks for `NULL` pointers before freeing.

G.7 convertToPDS Function

```
PDSMatrix* convertToPDS(double** A, int num_rows, int num_cols, int* offsets, int
    num_diags) {
    PDSMatrix* pds = createPDSMatrix(num_rows, num_cols, num_diags);
    if (!pds) return NULL;

    // Copy offsets
    for (int i = 0; i < num_diags; i++) {
        pds->diag_offsets[i] = offsets[i];
    }

    // Allocate memory for each diagonal
    for (int i = 0; i < num_diags; i++) {
        int offset = pds->diag_offsets[i];
        int diag_length;

        if (offset >= 0) {
            diag_length = num_cols - offset;
        } else {
            diag_length = num_rows + offset;
        }

        pds->diag_values[i] = (double*)malloc(diag_length * sizeof(double));
        if (pds->diag_values[i] == NULL) {
            fprintf(stderr, "Memory allocation failed for diag_values[%d].\n", i)
    ;
            // Clean up allocated memory before returning
            for (int j = 0; j < i; j++) {
                free(pds->diag_values[j]);
            }
            free(pds->diag_values);
            free(pds->diag_offsets);
            free(pds);
            return NULL;
        }
    }

    // Fill the diagonal values
    for (int i = 0; i < num_diags; i++) {
        int offset = pds->diag_offsets[i];
        int diag_length;

        if (offset >= 0) {
            diag_length = num_cols - offset;
        } else {
            diag_length = num_rows + offset;
        }
        //populate the diagonal
        for (int j = 0; j < diag_length; j++) {
            int row = j;
            int col = j + offset;
            pds->diag_values[i][j] = A[row][col];
        }
    }
    return pds;
}
```

Listing G.5: `convertToPDS` Function

- Converts a dense matrix `A` into the PDS format, storing only the specified diagonals.
- Takes the dense matrix, number of rows, number of columns, an array of diagonal offsets (`offsets`), and the number of diagonals to store (`num_diags`) as input.
- Calculates the length of each diagonal based on its offset.
- Allocates memory for each diagonal in the `diag_values` array.
- Fills the `diag_values` array with the corresponding values from the dense matrix.
- Includes memory cleanup in case of allocation failure.
- Returns a pointer to the created `PDSMatrix` or `NULL` if an error occurred.

G.8 `spmvPDS` Function

```
void spmvPDS(const PDSMatrix* pds, const double* x, double* y) {
    int num_rows = pds->num_rows;
    int num_cols = pds->num_cols;
```

```c
    int num_diags = pds->num_diags;

    // Initialize y to zero
    for (int i = 0; i < num_rows; i++) {
        y[i] = 0.0;
    }

    // Perform SpMV
    for (int i = 0; i < num_diags; i++) {
        int offset = pds->diag_offsets[i];
        int diag_length;

        if (offset >= 0) {
            diag_length = num_cols - offset;
        } else {
            diag_length = num_rows + offset;
        }

        for (int j = 0; j < diag_length; j++) {
            int row = j;
            int col = j + offset;

            y[row] += pds->diag_values[i][j] * x[col];
        }
    }
}
```

Listing G.6: `spmvPDS` Function

- Performs Sparse Matrix-Vector Multiplication (SpMV) using the PDS format.
- Takes a pointer to the `PDSMatrix` structure, a pointer to the input vector `x`, and a pointer to the output vector `y` as input.
- Initializes the output vector `y` to zero.
- Iterates through the stored diagonals.
- For each diagonal, it iterates through the elements and performs the multiplication, accumulating the results in the output vector `y`.

G.9 main Function

```c
int main(int argc, char* argv[]) {
    int num_rows = 5;
    int num_cols = 5;

    // Example dense matrix (for simplicity)
    double** A = (double**)malloc(num_rows * sizeof(double*));
    for (int i = 0; i < num_rows; i++) {
        A[i] = (double*)malloc(num_cols * sizeof(double));
        for (int j = 0; j < num_cols; j++) {
            A[i][j] = 0.0;  // Initialize to zero
        }
    }

    // Sample data - sparse but with diagonal structure
    A[0][0] = 1.0;
    A[0][1] = 2.0;
    A[1][0] = 3.0;
    A[1][1] = 4.0;
    A[1][2] = 5.0;
    A[2][1] = 6.0;
    A[2][2] = 7.0;
    A[2][3] = 8.0;
    A[3][2] = 9.0;
    A[3][3] = 10.0;
    A[3][4] = 11.0;
    A[4][3] = 12.0;
    A[4][4] = 13.0;

    // Diagonals to store (main, and one above and below)
    int offsets[] = {-1, 0, 1};
    int num_diags = sizeof(offsets) / sizeof(offsets[0]);

    PDSMatrix* pds_matrix = convertToPDS(A, num_rows, num_cols, offsets,
   num_diags);

    if (!pds_matrix) {
        //Free A
        for (int i = 0; i < num_rows; i++) {
            free(A[i]);
        }
        free(A);
        return 1;
    }

    double* x = (double*)malloc(num_cols * sizeof(double));
    double* y = (double*)malloc(num_rows * sizeof(double));

    if (!x || !y) {
        fprintf(stderr, "Memory allocation failed for vectors.\n");
        freePDSMatrix(pds_matrix);
         //Free A
        for (int i = 0; i < num_rows; i++) {
            free(A[i]);
        }
        free(A);
        free(x);
        free(y);
        return 1;
    }

    // Initialize x (e.g., with all 1s)
    for (int i = 0; i < num_cols; i++) {
        x[i] = 1.0;
    }

    clock_t start_time = clock();
    spmvPDS(pds_matrix, x, y);
    clock_t end_time = clock();

    double elapsed_time = (double)(end_time - start_time) / CLOCKS_PER_SEC;

    printf("Time taken for SpMV (PDS): %f seconds\n", elapsed_time);

    free(x);
    free(y);
    freePDSMatrix(pds_matrix);

    //Free A
    for (int i = 0; i < num_rows; i++) {
        free(A[i]);
    }
    free(A);

    return 0;
}
```

Listing G.7: `main` Function

- Demonstrates the PDS format with a small, manually defined matrix.
- Creates a dense matrix `A` and populates it with sample data that exhibits a diagonal structure.
- Defines the diagonal offsets to be stored in the PDS format.
- Converts the dense matrix to PDS format using `convertToPDS`.
- Performs SpMV using `spmvPDS`.
- Measures and prints the execution time.
- Deallocates all dynamically allocated memory.

G.10 Key Observations and Potential Improvements

- **Diagonal Structure Assumption:** PDS is most efficient when the non-zero elements are primarily located along a few well-defined diagonals. For matrices with a more random sparsity pattern, other formats like CSR or CSC are generally more suitable.
- **Dense Matrix Input:** The code currently converts from a dense matrix. A practical implementation should convert from COO or CSR to avoid the memory overhead of storing the entire matrix densely.
- **No Matrix Market Support:** The code lacks the ability to read matrices directly from Matrix Market (.mtx) files into PDS format.

- **Memory Allocation for Diagonals:** While ‘diag_values‘ is allocated dynamically, the size of each diagonal is determined based on the matrix dimensions. For some matrices, there might be opportunities to further reduce memory usage by only storing the active part of the diagonals.

G.11 Recommendations

- **Implement Conversion from COO/CSR:** Modify the `convertToPDS` function to accept a COO or CSR matrix as input.

- **Add Matrix Market Support:** Implement a function to read sparse matrices from Matrix Market files and convert them to the PDS format.

- **Analyze Diagonal Structure:** Before converting to PDS, analyze the matrix to identify the dominant diagonals and determine the optimal set of diagonal offsets to store.

- **Compare Performance:** Thoroughly compare the performance of PDS SpMV with CSR and CSC SpMV to determine its effectiveness for different matrices.

- Explore Alternative Diagonal Matrix algorithms: Explore alternative diagonal algorithms, and matrix multiplication done with a transpose.

G.12 Example Usage

To compile and run the code:

```
gcc -o pds_spmv pds_spmv.c -lm
./pds_spmv
```

Important Considerations:

- The `-lm` flag is needed to link the math library (for `abs()`).

- The provided `main` function uses a hardcoded dense matrix. For real-world testing, you’ll need to implement the Matrix Market reader and modify the `main` function accordingly.

G.13 Conclusion

This document provides a basic C code implementation of the PDS format and SpMV. While the code demonstrates the fundamental principles of PDS, it's essential to address the limitations and implement the recommendations to make it practical for handling real-world sparse matrices. The efficiency of PDS heavily depends on the specific diagonal structure of the matrix, so thorough analysis and benchmarking are crucial.