Group 7

NEW STORAGE FORMAT FOR SPARSE MATRIX

THESIS

Under the guidance of
Dr. Chandan Misra

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

RECAP….

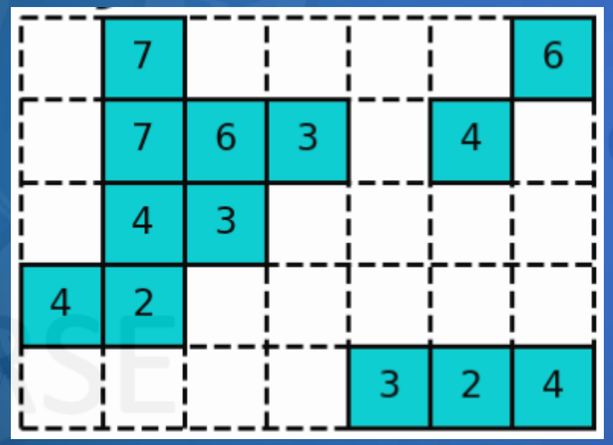
Results/Analysis

- **Results:**

- Proposed method outperforms traditional formats in memory usage and speed.
- Significant reduction in storage overhead.
- Achieved higher GFLOPS, indicating better computational performance.

- **Analysis:**

- Effective handling of matrices with specific sparsity patterns.
- Demonstrates scalability and applicability to real-time systems.
- Versatility in adapting to diverse application needs (e.g., NLP, graph analysis).



Maximizing insights from research exploration

Compressed Diagonal Storage (CDS)

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

Compressed Diagonal Storage (CDS) 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)

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

0	2	3	-2	-3
1	2	2	0	0
5	0	0	0	3
6	0	6	0	0
10	0	0	10	0
12	0	0	0	12

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

Jagged Diagonal Storage (JDS)

Jagged diagonal storage format 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.

$$\begin{bmatrix} a_{11} & a_{12} & 0 & a_{14} & 0 & 0 \\ 0 & a_{22} & a_{23} & 0 & a_{25} & 0 \\ a_{31} & 0 & a_{33} & a_{34} & 0 & 0 \\ 0 & a_{42} & 0 & a_{44} & a_{45} & a_{46} \\ 0 & 0 & 0 & 0 & a_{55} & a_{56} \\ 0 & 0 & 0 & 0 & a_{65} & a_{66} \end{bmatrix}$$

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).**

$$\begin{bmatrix} a_{42} & a_{44} & a_{45} & a_{46} & 0 & 0 \\ a_{11} & a_{12} & a_{14} & 0 & 0 & 0 \\ a_{22} & a_{23} & a_{25} & 0 & 0 & 0 \\ a_{31} & a_{33} & a_{34} & 0 & 0 & 0 \\ a_{55} & a_{56} & 0 & 0 & 0 & 0 \\ a_{65} & a_{66} & 0 & 0 & 0 & 0 \end{bmatrix}$$

Packed Diagonal Storage (PDS)

Packed Diagonal Storage (PDS) 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.

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

PDS Storage Structure

PDS 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:

-1	0	1
0	1	2
3	4	5
6	7	8
9	10	11
12	13	0

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

Space

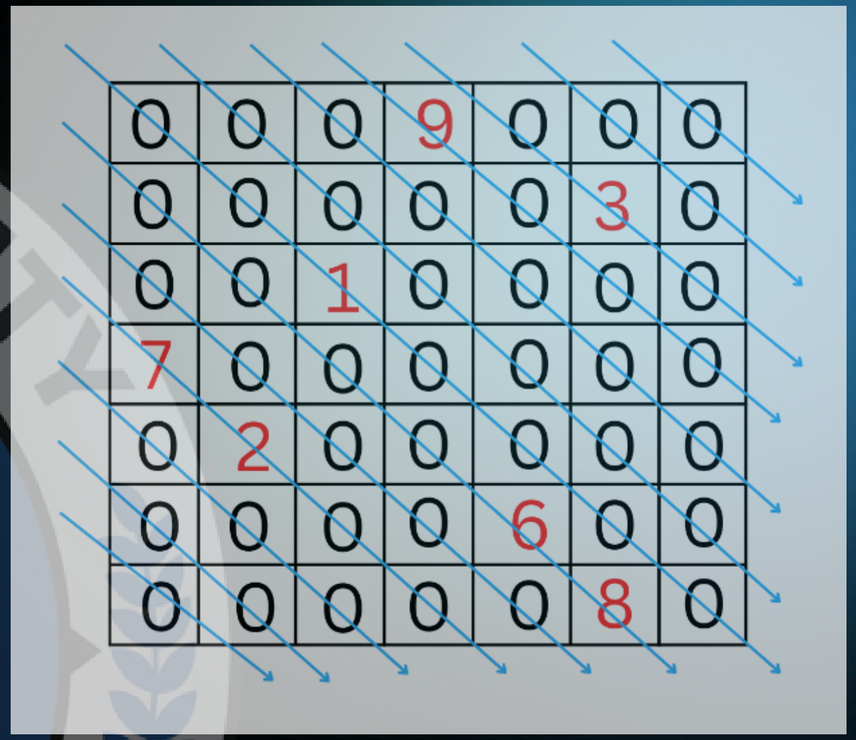
Formats
CDS
JDS
PDS
BDS
CC
COO

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)	$\frac{NnZ \times 16}{1024 \times 1024}$

Contiguous Clustering(CC)

Contiguous Clustering(CC)

Our Algorithm uses a Store Diagonal 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 . 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.



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.

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.

numOfClusters
clusterSizes
startRowClus
startColClus
storeValues

numOfClusters

5 (one for each diagonal)

- It contains the count of diagonal occurs in a matrix.

1 :- [34], 2 :- [5, 7], 3 :- [1], 4 :- [8, 9], 5:- [1], 6 :- [2, 3, 6], 7 :- [45]

1	0	2	0	45
0	0	0	3	0
5	0	0	0	6
0	7	0	8	1
34	0	0	0	9

clusterSizes

[1, 2, 1, 2, 1, 3, 1] (size of each diagonal)

- Stores the size of each cluster.

1 :- 1, 2 :- 2, 3 :- 1, 4:- 2, 5:- 1, 6 :- 3, 7 :- 1

startRowClus

[4, 2, 0, 3, 3, 0, 0] (starting rows of each diagonal

- Starting row index for each cluster.

1 :- 4, 2 :- 2, 3 :- 0, 4:- 3, 5:- 3, 6 :- 0, 7 :- 0

startColClus

[1, 1, 1, 3 , 4, 2, 4] (starting rows of each diagonal

- Starting row index for each cluster.

1 :- 1, 2 :- 1, 3 :- 1, 4:- 3, 5:- 4, 6 :- 2, 7 :- 4

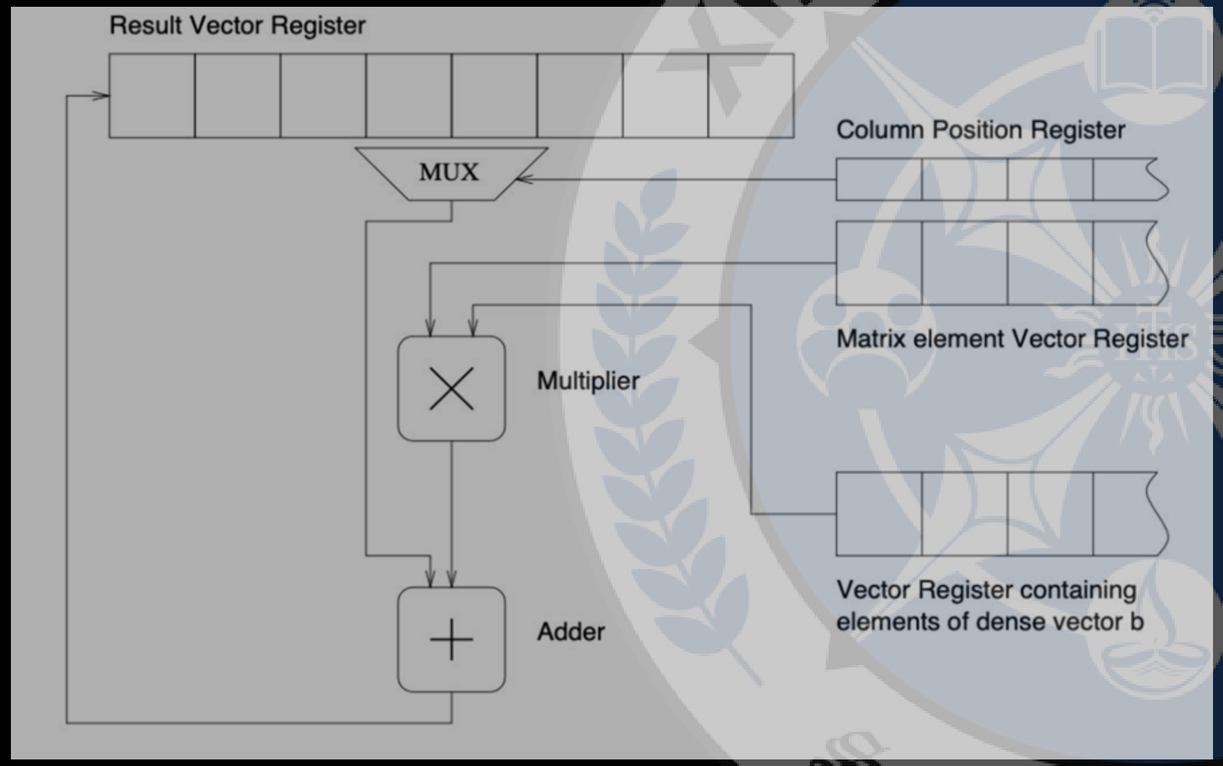
storeValues

[34, 5, 7, 1, 8, 9, 1, 2, 3, 6, 45] (values corresponding to the structure)

- Stores the values associated with elements in the cluster format.

1 :- [34], 2 :- [5, 7], 3 :- [1], 4 :- [8, 9], 5:- [1], 6 :- [2, 3, 6], 7 :- [45]

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.

WHY:

1. **Efficiency:** Handles massive sparse matrices (mostly zeros) efficiently, saving memory and computation.
2.Modeling Power: Enables practical work with large-scale models representing complex real-world systems.
3.Optimization Target: The thesis itself might focus on creating faster or more energy-efficient SPMV methods.

RESULT ANALYSIS

CC vs COO

Mtx Name: Names/identifiers of the test sparse matrices.

Mtx Size: Size of the Matrix

CC (Overall Header): Category for metrics related to a 'CC' method/process.

Clusters (under CC): Number of clusters resulting from the 'CC' process.

time (under CC/COO): Execution time (likely seconds) for the 'CC' or 'COO;process.

COO (Overall Header): Category for metrics related to the COO storage format.

Space (MB) (under COO): Memory required (in MB) to store the matrix in COO format.

GFlops (under COO): Performance (in GFLOPS) achieved, likely during computation (e.g., SpMV) with the COO format.

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

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

CC vs Diagonal Storage Formats

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

Mtx Name: Identifiers/names of the test sparse matrices.

CDS Space (MB): Memory required (in MB) to store the matrix using the CDS (Compressed Diagonal Storage or similar) format.

Offset: Likely an offset value related to the CDS data structure or an associated process.

CC Clusters: Number of clusters resulting from a 'CC' (likely Connected Components or a clustering) process.

CC Time (s): Execution time (in seconds) for the 'CC' process.

Mtx Name: Identifiers/names of the test sparse matrices.

JDS Space (MB): Memory required (in MB) to store the matrix using the JDS (Jagged Diagonal Storage) format.

GFlops: Computational performance (in GigaFLOPS), likely measured during a benchmark (e.g., SpMV) using the JDS format.

CC Time (s): Execution time (in seconds) for the 'CC' process (repeated for comparison).

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

CC vs Diagonal Storage Formats

Mtx Name: Identifiers/names of the test sparse matrices.

PDS Space (MB): Memory required (in MB) to store the matrix using the PDS (likely Packed Diagonal Storage or another 'PDS' variant) format.

CC Clusters: Number of clusters resulting from the 'CC' process (repeated for comparison).

CC Time (s): Execution time (in seconds) for the 'CC' process (repeated for comparison).

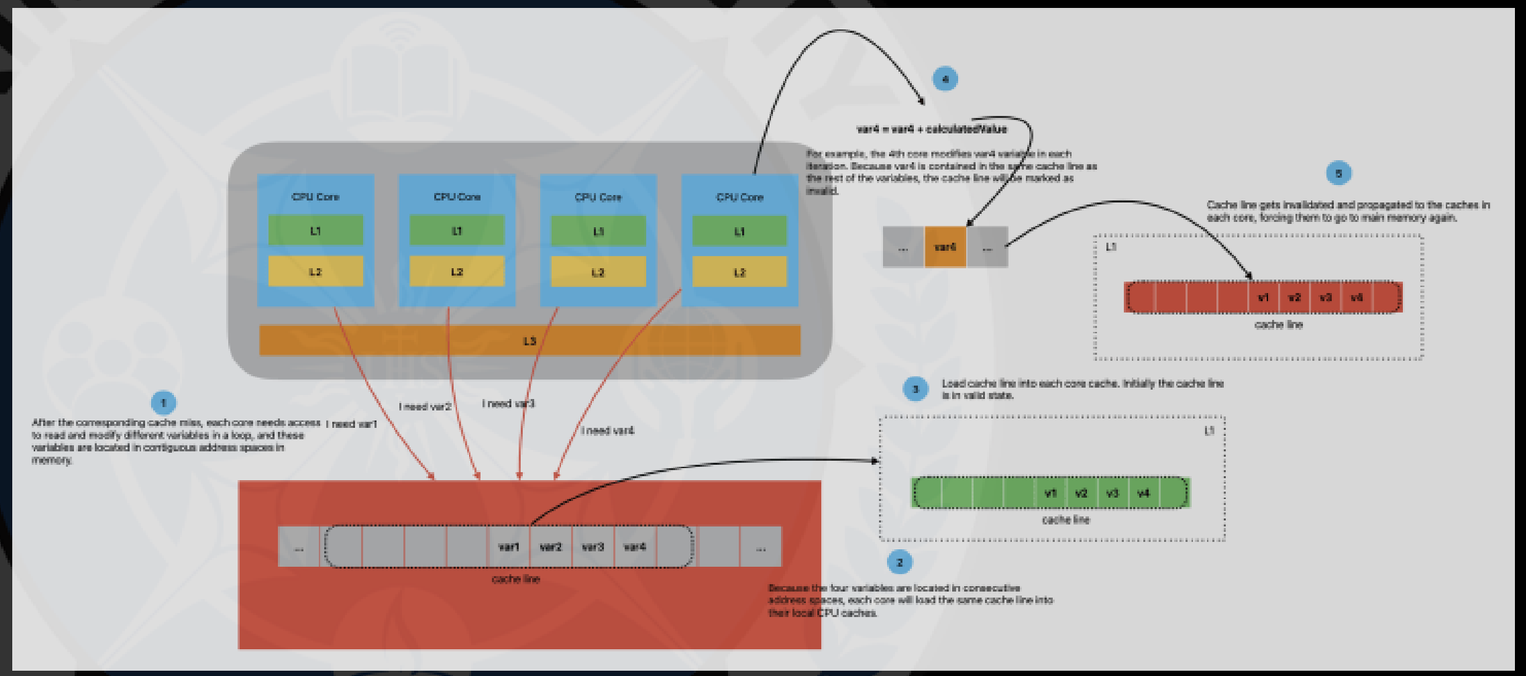
Mtx Name	PDS Space (MB)	CC Clusters	CC Time (s)
mc2dmpi	5258925	3581	0.009313
PR02R	161070	34306111	0.032389
Coupcons3D	416800	498112881	0.105682
channel-500x1000x100-b050	4802000	740476	0.162471

parallelization & Challenges

False Sharing

False sharing happens when multiple cores that share the same cache line are modifying data within this cache line very frequently. This causes the cache line to be marked as invalid, what forces every core to fetch the data from main memory (RAM) every time, leading to a serious performance degradation as CPU caching becomes pointless in this scenario.

This happens because the system is trying to keep the cache coherence among the caches present in each of the cores, so when it detects a change in one of the cache lines, enforces every core to get the most recent version from main memory to avoid inconsistencies in the data.



Thank You