Evaluate Metadata of Sparse Matrix for SpMV on Shared Memory Architecture

Nazmul Ahasan Maruf[1], Waseem Ahmed[2]
Faculty of Computing and Information Technology
King Abdulaziz University, Jeddah, Saudi Arabia

***Abstract*—Sparse Matrix operations are frequently used operations in scientific, engineering and high-performance computing (HPC) applications. Among them, sparse matrix-vector multiplication (SpMV) is a popular kernel and considered an important numerical method for science, engineering and in scientific computing. However, SpMV is a computationally expensive operation. To obtain better performance, SpMV depends on certain factors; choosing the right storage format for the sparse matrix is one of them. Other things like data access pattern, the sparsity of the matrix data set, load balancing, sharing of the memory hierarchy, etc. are other factors that affect performance. Metadata, that describes the substructure of the sparse matrix, like shape, density, sparsity, etc. of the sparse matrix also affects performance efficiency for any sparse matrix operation. Various approaches presented in literature over the last few decades given good results for certain types of matrix structures and don't perform as well with others. Developers thus are faced with a difficulty in choosing the most appropriate format. In this research, an approach is presented that evaluates metadata of a given sparse matrix and suggest to the developers the most suitable storage format to use for SpMV.**

Keywords*—Sparse matrix vector multiplication; sparse matrix metadata; sparse matrix vector multiplication parallelization; shared memory architecture; sparse matrix storage formats; high performance computing

I. Introduction

Sparse matrix-vector multiplication (SpMV) is an essential and frequently used kernel in high-performance computing (HPC), scientific and engineering applications. The operation $y = Ax$ is performed by the SpMV kernel, where A is a sparse matrix of size $M \times N$, and y and x are dense vectors of size M. Although SpMV is one of the most popular and essential kernels, it usually performs poorly for large sparse matrices. As has been shown by Goumas and others, SpMV achieves less than 10% of the peak performance of microprocessors [1]. Higher performance for SpMV depends on various factors - the choice of the right sparse matrix storage format is one of them. Other factors like data access pattern, the sparsity of the matrix data set, load balancing, sharing of the memory hierarchy, etc. are important when we work on shared memory architecture. In shared memory architectures, performance degradation might happen when all the processors try to access the memory simultaneously. Applications on shared memory systems that have no data dependencies and have good temporal locality tend to perform well. On the other hand, contention on memory subsystems in other shared memory applications with streaming access patterns results in poor performance [2]. In literature, we found that the SpMV kernel performs poorly on a shared memory system because of its streaming access patter [2]. In shared memory architecture, memory bandwidth also is a performance bottleneck for SpMV operations when operating on large matrices [3]. Each element of the matrix is only used once in the SpMV operation. Irregular data access is another performance bottleneck for SpMV operation when we use sparse matrix storage formats like COO and CSR. For example, when using Coordinate format (COO) for large scale SpMV, high cache miss rate and poor performance were noticed as a result of indirect addressing. Irregular data accessing in SpMV also results in reduction of performance.

To address these kinds of problems, researchers have proposed different sparse matrix storage formats [2], [4]–[6]. For example, Compressed Sparse Row (CSR) has been proposed to address space overhead but CSR does not reduce irregular data accesses completely if non-zero elements (*nnz*) elements are mostly in a certain row. Also, when parallelizing CSR on shared-memory architectures for operations on such matrices, load balancing becomes an issue. Additionally, matrix substructures also affect performance efficiency for all sparse matrix storage format; storage formats that perform well with certain substructures do not perform well with others. Take for example, ELLPACK [7]. The ELLPACK storage format is very suitable for diagonal substructures but not for horizontal substructures. SpMV performance also depends on other metadata like diagonal density, row or column-major order, max non-zero values (row thickness) per row, etc. Another example is that if the sparse matrix is not pattern symmetric and the row thickness is very large, CSR performs better than ELLPACK. Thus, it's important for a developer to know beforehand the metadata of a sparse matrix to help him decide which storage format to use. In this research, we propose new metrics to describe sparse matrices to add to existing metadata description which will help developers to take decisions easily. Our work was motivated by two key observations

1) Different storage formats give higher performance for particular substructures of matrices and lower on others.
2) Most storage formats are not based on metadata or substructures of the sparse matrices.

A tool was developed to generate the metrics to help developers find the optimal storage format to use for any given sparse matrix. The rest of the paper is organized as follows.

Section 2 presents some popular and relevant storage formats.Related work discussed in Section 3. Section 4 describes the motivation and approach for our metrics, tool,and also described the sparse matrix benchmarks used in this research.

Section 5 presents the empirical analysis using metrics and experimental evaluation. Section 6 discusses future work and concludes the paper.

II. Storage Formats

There are many storage formats for sparse matrices described in literature [8] [9]–[14] for the SpMV operation. Some of the important and popular storage formats are described in the next sub sections.

To better understand the storage formats an example sparse matrix A of size 5×5 and with elements described in Fig. 1 is used.

$$A_{5\times5} = \begin{bmatrix} 0 & 4 & 0 & 7 & 0 \\ 2 & 0 & 3 & 0 & 6 \\ 0 & 5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 \\ 1 & 0 & 0 & 6 & 0 \end{bmatrix}$$

Fig. 1. Sample Matrix for study

A. Coordinate (COO)

COO is one of the earliest and most basic format for storing sparse matrices [15] [16] and is very simple and reliable. This format stores the row index, column index and number of non-zero (nnz) values in three one-dimensional array - one for storing non zero values, one for storing row indices and one for column indices. Fig. 2 shows the COO format for matrix A given in Fig. 1.

$$Data = \begin{bmatrix} 4 & 7 & 2 & 3 & 6 & 5 & 2 & 1 & 6 \end{bmatrix}$$

$$Row_i = \begin{bmatrix} 0 & 0 & 1 & 1 & 1 & 2 & 3 & 4 & 4 \end{bmatrix}$$

$$Col_i = \begin{bmatrix} 1 & 3 & 0 & 2 & 4 & 1 & 4 & 0 & 3 \end{bmatrix}$$

Fig. 2. COO Format With Data, Row And Column Field

The storage requirement for COO to store a matrix of dimension $M \times N$ with NNZ non-zero values is

$$COO_{storage} = 3 \times NNZ$$

B. Compressed Sparse Row (CSR)

Compressed sparse row (CSR) format is the most popular and widely used sparse formats [17]. SpMV with CSR format gives good performance and is used in libraries like BLAS and LAPACK. Like COO, CSR also needs three 1-D arrays for storing data; one holds the *nnz* values, another one is for the number of *nnz* values per row, and one for column indices. Fig. 3 shows the CSR format for matrix A given in Fig. 1.

The storage requirement for CSR to store a matrix of dimension $M \times N$ with NNZ non-zero values is

$$CSR = 2 \times NNZ + M + 1$$

$$Data = \begin{bmatrix} 4 & 7 & 2 & 3 & 6 & 5 & 2 & 1 & 6 \end{bmatrix}$$

$$Rowptr = \begin{bmatrix} 0 & 2 & 5 & 6 & 7 & 9 \end{bmatrix}$$

$$Col_i = \begin{bmatrix} 1 & 3 & 0 & 2 & 4 & 1 & 4 & 0 & 3 \end{bmatrix}$$

Fig. 3. CSR Format With Data, Row And Column Field

C. Compressed Sparse Column (CSC)

The Compressed Sparse Column (CSC) [18] [19] is similar to the Compressed sparse row (CSR). The main diference between them is that CSC uses column pointer instead of row pointer. *CSC* uses three 1-D array for storage; one for nnz values, one for column pointer and one for row indices. Fig. 4 shows the CSC format for matrix A given in Fig. 1.

$$Data = \begin{bmatrix} 4 & 7 & 2 & 3 & 6 & 5 & 2 & 1 & 6 \end{bmatrix}$$

$$colptr = \begin{bmatrix} 0 & 2 & 4 & 5 & 7 & 9 \end{bmatrix}$$

$$row_i = \begin{bmatrix} 0 & 0 & 1 & 1 & 1 & 2 & 3 & 4 & 4 \end{bmatrix}$$

Fig. 4. CSC Format With Data, Row And Column Field

The storage requirement for CSC to store a matrix of dimension $M \times N$ with NNZ non-zero values is

$$CSC = 2 \times NNZ + N + 1$$

D. ELLPACK

Another commonly referenced format is ELLPACK. It is well suited for semi-structured and unstructured meshes and for vector architectures [7]. It is also a good storage format for diagonal matrices. It is particularly suited in cases where the maximum number of non-zero values per row does not differ more with the average of non-zero elements in all rows. ELLPACK uses two 2D matrices where one is for storing the *nnz* values in row-wise order and another is for storing column indices of the *nnz* values. If our matrix size is $M \times N$ and $max(N_{nzr})$ presenting the maximum non zero values per row then the column indices 2D array has a size of $M \times max(N_{nzr})$. So the storage for ELLPACK shows in Fig. 5 for matrix A given in Fig. 1.

$$Data = \begin{bmatrix} 4 & 7 & \\ 2 & 3 & 6 \\ 5 & & \\ 2 & & \\ 1 & 6 & \end{bmatrix} Col_i = \begin{bmatrix} 1 & 3 & \\ 0 & 2 & 4 \\ 1 & & \\ 4 & & \\ 0 & 3 & \end{bmatrix}$$

Fig. 5. ELLPACK Format With Data And Column Field

At the implementation level, there are two ways to store the ELLPACK format. One way is to use 2-D arrays. In this

case, the storage requirement for ELLPACK to store a matrix of dimension $M \times N$ with NNZ non-zero values is

$$ELLPACK_{storage} = 2(max(N_{nzr}) \times m)$$

E. Compressed Sparse Row-DU (CSR-DU)

In the CSR-DU [4] format, instead of using col_index and row_ptr (or column_ptr) like in CSR (or CSC), *ctl* a single byte array is used. This array consists of four elements: *uflags*, *usize*, *ujump*, *ucis*. *uflags* and *usize* (to identify the type of unit and size), *ujump* (a variable length which denotes the first column index of each unit), and *ucis* (an array which denotes the distance between the column index of the first element of the unit and the column index of the previous element). Fig. 6 shows the CSR-DU format for matrix A given in Fig. 1.

units	uflags	usize	ujump	ucis
0	U8,NR	2	1	2
1	U8,NR	3	0	2,2
2	U8,NR	1	1	-
3	U8,NR	1	4	-
4	U8,NR	2	0	3

$Data = [\ 4\ \ 7\ \ 2\ \ 3\ \ 6\ \ 5\ \ 2\ \ 1\ \ 6\ ]$

Fig. 6. CSR-DU Format with data and ctl values

F. Compressed Sparse Row-VI(CSR-VI)

This format is extension of the CSR format. In CSR-VI [4], the *values* array of CSR is replaced with two arrays, *val_unique*: which contains the unique values of the matrix and *val_ind*: the index of the value in the vals_unique array for each of the nnz matrix element. Fig. 7 shows the CSR-VI format for matrix A given in Fig. 1.

$val - indx$ = [1 3 0 2 4 1 4 0 3]
$val - unique$ = [4 7 2 3 6 5 1]

Fig. 7. CSR-VI Format with Values

G. Compressed Sparse eXtended (CSX)

CSX [2] is also an extension of CSR-DU where CSX support different classes of regularities. In CSX, run time code generation method is employed. In CSX, five types of substructures are considered: horizontal, vertical, diagonal, anti-diagonal, and 2-D substructures. For any substructures, CSX uses run length encoding on delta values for more aggressive index compression. For 2-D substructures, transformation is used to convert different substructures to horizontal substructures. Fig. 8 shows the CSX format for matrix A given in Fig. 1.

$delta - values$ = [2 2 2 3]
$indices$ =[1 3 0 2 4 1 2 0 3]

Directions	Elements	
	y	x
Horizontal	y_o	$x_o + i\delta$
Vertical	$y_o + i\delta$	x_o
Diagonal	$y_o + i\delta$	$x_o + i\delta$
Anti-Diagonal	$y_o + i\delta$	$x_o - i\delta$

Fig. 8. CSX with data directions and transformation

III. Related Works

Here presenting some of related works, where researcher want to optimize SpMV with respect to sparse storage format.

Kenli Li, analysis and optimize SpMV using probabilistic method, they present probability mass function (PMF) for non zero elements in sparse matrix, they also evaluate the performance and efficiency of COO, CSR, ELL and HYB and find out the optimistic solution for SpMV using probability mass function (PMF), they also observed that different most of the matrices have their own most appropriate storage format for achieving best performance [15].

Daniele Buono, works with shared memory multiprocessor, they evaluate a new methodology to implement SpMV on shared memory multiprocessor which has two phases, one is for building a scalar matrix and on the second phase at first they reduce the scalar matrix by row by providing numerous opportunities to memory location, in this paper they use CSR as there baseline because CSR is inefficient with very sparse and graphs [20].

Zhang, on there paper they want to optimize data locality of sparse matrix vector multiplication algorithm by improving the sparse matrix storage format, they proposed cache oblivious extension quadtree storage structure (COEQT), where the sparse matrix is recursively divided into sub module and that will also fit into the cache by doing this it will improve the data locality [21].

Xiaowen Feng, proposed a new storage format for Sparse matrix name Segmented Interleave Combination (SIC), Instead of using compressed sparse row (CSR) they proposed this format because they find out a problem if the non zero variables are very high then thread divergence can cause and that will affect the performance while using CSR, so they combined the CSR values and proposed a new storage format name SIC [22].

Arash Ashari, proposed Blocked row column (BRC) storage format that optimize the Sparse Matrix vector with addressing the thread divergence, redundant computation in GPUs in the paper they also find out the optimizing challenges which are thread divergence, load imbalance, non-coalesced, and memory access [10].

Yang, introduce a SpMV that represent large graphs which has noticeable performance, they marge the ideas form Transposed Jagged Diagonal storage (TJDS) [23] with coo [24].

Francisco Vazquez, evaluated a new approach based on ELLR-T kernel on GPU using CUDA where they find out the performance of this specific kernel based on the thread block size and the number of accumulated threads, they also proposed a auto tuning model based on memory access for GPU for ELLR-T kernels [25].

Baskaran, they optimize SpMV on compilation and run time basis, in compilation time they include synchronization

free parallelism, thread mapping, optimize memory access and data redundancy on their optimization they proposed a new Blocked storage format and implement it on GPUs [26].

Shizhao Chen, perform a comprehensive study on representation of sparse matrix on Intel Knights landing XeonPhi and ARM-based FT-200PLUS architecture, they found that best representation of sparse relay on architecture and the unit of program, in their paper they use very well known CSR,CSR5,ELL,SELL and HYB sparse matrix storage format [27].

Weifeng Liu, they introduce a new storage format CSR5 (compressed sparse row) which provides high throughput SpMV on CPUs,GPUs and Xeon Phi, in the paper they also mention an efficient storage format should agree with two criteria, one is it should avoiding structure-dependent parameter tuning, another one is it should support fast sparse matrices vector for regular and irregular matrices [28].

IV. Metric

A. Motivation

For the SpMV kernel, the actual and maximum performance strongly depends on the characteristics and nature (substructure) of a sparse matrix. Any generalized storage format like CSR, COO, ELLPACK, etc. does not make any assumption about the shape and metadata of an input sparse matrix [2]. From our research, we found that some storage formats perform better on some sets of sparse matrices and perform poorly on some. In Fig. 9, performance evaluation for the basic and essential storage formats with GFLOPS (Floating point operation per second) is presented. As described earlier, performance of the SpMV operation depends on data access pattern, data dependency among storage arrays in some formats, load balancing when parallelizing the operation on shared memory architecture, sharing of the memory hierarchy, choosing the right storage format for specific sparse matrices. For choosing the right storage format, we need to know the nature of the sparse matrix, but storage format do not make any assumptions about the sparse matrix substructure.

In our research, we find out that shape and substructure of the sparse matrix plays an important role in choosing the appropriate storage format. For example, any horizontal shaped sparse matrix, CSR will perform better than ELLPACK. Another example is ELLPACK which will perform better on the diagonal shape of data. Additionally, we need to use the thickness of the diagonal. If the thickness of diagonal increases then the performance of ELLPACK will be decreased, because for ELLPACK the 2D array has a size of *m*×*Nmnzr* where *Nmnzr* is the maximum non zero values per row. If the number of non-zero values differs more with the average then ELLPACK performance will be decreased. Other characteristics (described in later sections) like symmetry, density, sparsify, etc. also helps in deciding the most appropriate storage formats for sparse matrix.

Besides COO, CSR (and its variances), CSC and ELLPACK, this extends to other proposed storage formats in literature. Table III describe, proposed sparse storage formats and their sparse matrix characteristics. From Table III Bell and Garland's HYBRID(HyB) format which is a hybrid of ELLPACK and COO, is generally the fastest format for unstructured matrices [7]. Hiroki Yoshizawa claims that performance of SpMV with compressed sparse row (CSR) storage format depends on selection of parameter. In the paper, they also mention that conjugate gradient method is one of the popular iterative methods for solving SpMV; they put their focus on optimizing thread mapping for the CSR; they propose an efficient algorithm for automatic selection of optimal parameter on GPU, which is generally performing better then CUSPARSE when the diagonal density is higher [29]. F. Vazquez [25] proposed and evaluated implementation of ELLR-T which is based on ELLPACK storage format, and the evaluation goes with some parameters like number of rows, total number of *nnz* elements, average number of entries with respect to rows, difference of the maximum number of entries in a row and average entries with respect to row, percentage standard deviation of entries. Their proposed approach performs better with less diagonal density data.

B. Bench Mark

In this paper, we will use a wide range of sparse matrix with different structural and numerical properties. We collect our sparse matrix from University of Florida sparse matrix collection (UFSMC) [30], which is the standard and most popular benchmark in SpMV research. Table I shows the benchmarks used for the experimental evaluation.

C. Metrics

To better understand the substructure of a sparse matrix and to automate the decision of choosing the best storage format given a sparse matrix, a set of newer metrics are needed to complement the existing metadata used to describe the substructure of a sparse matrix. This subsection describes these newer metrics in more detail.

1) *rowThickness* - In a given sparse matrix, the maximum number of *nnz* values per row is defined here as *rowThickness* and is given by Equation1, where r_i indicates the number of *nnz* values in row *i*.

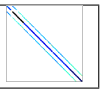
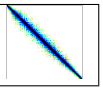
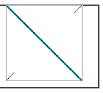
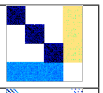
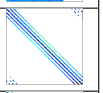
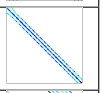
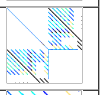
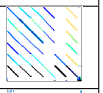
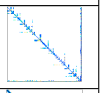
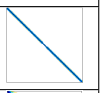
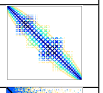
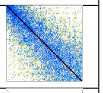
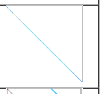
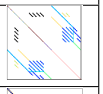
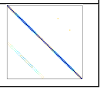
$$rowThickness = max(r_1, r_2, r_3, ...r_N) \quad (1)$$

The SpMV performance for any storage format depends on *rowThickness*.For example, if $rowThickness \ll avg(r_1, r_2, r_3, ...r_N)$ the performance of ELLPACK storage format decreases. In this case, CSR, COO or other formats will be the better choice for developers. On the other hand when $rowThickness \approx avg(r_1, r_2, r_3, ...r_N)$ ELLPACK performs better.

2) *Pattern Symmetry* - The sparse (square) matrix will be pattern symmetric if the existence of *nnz* entries match across the diagonal [30]. To better understand pattern symmetry, let's consider a square matrix P of size $M \times M$. This matrix will be pattern symmetric if $\forall i, jP(i, j) \neq 0 \rightarrow P(j, i) \neq 0$.

3) *Numeric Symmetry* - The sparse matrix will be numeric symmetric if the *nnz* entries numerically match across the diagonal [30]. Consider a square sparse matrix P of size $M \times$M. Matrix P will be pattern symmetric if $\forall i, jP(i, j) = P(j, i)$ or $P = P^T$.

TABLE I. THE BENCHMARKS USED FOR THE EXPERIMENTAL EVALUATION.

No.	Matrix Name	Rows	Columns	Non-zeros	Plot
1	sherman3	5,005	5,005	20,033	
2	airfoil_2d	14,214	14,214	259,688	
3	bbmat	38,744	38,744	177,1722	
4	ill_stokes	20,896	20,896	131,368	
5	gt01r	7,980	7,980	430,909	
6	cavity26	4,562	4,562	138,040	
7	lowthrust	18,476	18,476	224,897	
8	windtunnel	40,816	40,816	803,978	
9	circuit	12,127	12,127	48,137	
10	qa8fm	66,127	66,127	1,660,579	
11	ga3as3h12	61,349	61,349	5,970,947	
12	nd24k	72,000	72,000	28,715,634	
13	bcsstm	15,439	15,439	15,439	
14	robot	2,358	2,358	18,218	
15	dw2048	2,048	2,048	10,114	

4) *Horizontal Symmetry* - A sparse matrix P of size $M \times N$ will be horizontal symmetric if $\forall i \forall j P(i,j) = P(M-i, M-j)$.
5) *Vertical Symmetry* - A sparse matrix P of size $M \times N$ will be vertical symmetric if $\forall i \forall j P(i,j) = P(N-i, N-j)$.
6) *Row or Column-major Order* - A Sparse matrix entries stored in a file like $\forall e =< row, col, val >$. Row-major order for that stored sparse matrix will be $\forall e_{row}(e_{row} \leq e_{row+1})$ and column-major order will be $\forall e_{col}(e_{col} \leq e_{col+1})$.
7) *Density and Sparsity* - Sparsity of a sparse matrix will be $sparsity = zeroElements/(M \times N)$.On the other hand, $Density = nnz/(M \times N)$ will give us density of a sparse matrix. Density and sparsity are related as $sparsity = 1 - density$. Thus, any matrix with $sparsity \geq 0.5$ is generally considered as sparse otherwise the matrix is considered dense.
8) *Diagonal Density* - The *diagonal density* for any square sparse matrix is calculated by finding out the count of *nnz* values on the diagonal. Equation2 describe the diagonal density of a sparse matrix.

$$count(\forall i P(i,i) \neq 0) \tag{2}$$

9) *Upper Triangle Density* - The upper triangle density for any sparse matrix calculated by finding out the count of upper triangle *nnz* values where row indices are less then column indices. Equation 3 describe the upper triangle density of a sparse matrix.

$$count(\forall i \forall j P(i,j) = P(i,j) : j > i) \tag{3}$$

10) *Lower Triangle Density* - The lower triangle density for any sparse matrix calculated by finding out the count of lower triangle *nnz* values where row indices are greater then column indices. Equation describe the lower triangle density of a sparse matrix.

$$count(\forall i \forall j P(i,j) = P(i,j) : i > j) \tag{4}$$

D. Metric Generation

In this subsection, we describe the algorithms for generating the metrics to describe the substructure and shape of the sparse matrix. These metrics help to suggest the best storage format for a given sparse matrix. For generating the metrics, at first we will find out the *rowThickness*, in *rowThickness* we will find the maximum number of non zero values per row. Algorithm 1 describe rowThickness function.

Algorithm 1 Maximum Number per rows(*rowThickness*)

```
1:  function ROWTHICKNESS(col)
2:      count[col]++
3:      maxcol ← 0
4:      for i ← 1, col do
5:          if count[i] > maxCol then
6:              maxCol ← count[i]
7:          end if
8:      end for
9:      return maxCol
10: end function
```

Next, for getting more specific knowledge about the sparse matrix we need to find out the information about *pattern symmetric*, our algorithm will find the pattern symmetric by matching the upper triangle and lower triangle value, if it is the same the sparse matrix will be *pattern symmetric*. Algorithm 2 describe *pattern Symmetric* function.

After that, we will find out the percentage of *numeric, horizontal and vertical symmetric*. In *numeric symmetric*, our algorithm will find the percentage of the same upper triangle and lower triangle *nnz* values which are separated by a diagonal. Algorithm 3 describe *numeric Symmetric* function.

Algorithm 2 Pattern Symmetric

```
1: function PATTERNSYMMETRIC(row_indices, col_indices)
2:     for row ← 1, row_indecies do
3:         for col ← 1, col_indecies do
4:             if row < col then
5:                 up ← up + 1
6:             end if
7:             if row > col then
8:                 lw ← lw + 1
9:             end if
10:        end for
11:    end for
12:    if up != lw then
13:        patternSym == 1
14:    else
15:        patternSym == 0
16:    end if
       return patternSym
17: end function
```

Algorithm 3 Numeric Symmetric

```
1: function NUMERICSYMMETRIC(row_indices, col_indices)
2:     for row ← 1, row_indecies do
3:         for col ← 1, col_indecies do
4:             if data[row][col] != data[col][row] then
5:                 nCount ← nCount + 1
6:             end if
7:         end for
8:     end for
9:     numeric = (nCount/nnz) * 100
10:    return numeric
11: end function
```

In *horizontal and vertical* algorithms, we will check the percentage of the same value which is presented horizontally and vertically. Algorithm 4, 5 describes the horizontal and vertical symmetric function.

Algorithm 4 Horizontal Symmetric

```
1: function HORIZONTALSYMMETRIC(row_indices, col_indices)
2:     for row ← 1, row_indecies/2, AND, nrow ← row_indecies − 1, nrow ← nrow − 1 do
3:         for col ← 1, col_indecies do
4:             if data[row][col] != data[nrow][col] then
5:                 hCount ← hCount + 1
6:             end if
7:         end for
8:     end for
9:     horizontal = (hCount/nnz) * 100
10:    return horizontal
11: end function
```

Row and column-major order is also an important characteristic to find out the appropriate storage format for a particular sparse matrix. Our algorithm will find the row and column-major order by checking the row and column values from the data set are sorted or not. If row indices are sorted then it will print *row-major order* if column indices are sorted then it will print *column-major order*. Algorithm 6 describe

Algorithm 5 Vertical Symmetric

```
1: function VERTICALSYMMETRIC(row_indices, col_indices)
2:     for row ← 1, col_indecies/2, AND, ncol ← col_indecies − 1, ncol ← ncol − 1 do
3:         for col ← 1, col_indecies do
4:             if data[row][col] != data[ncol][col] then
5:                 vCount ← vCount + 1
6:             end if
7:         end for
8:     end for
9:     vertical = (vCount/nnz) * 100
10:    return vertical
11: end function
```

the row and column-major order. After finding the *row and column-major order*, we will approach for sparsity and density, Algorithms will find out the *sparsity* by dividing the zero-elements with row multiple column and *density* by dividing the *nnz* by row multiple column. Algorithm 7 describe the sparsity and density functions. After that we will find out the *density* of the *diagonal, upper, lower triangle*. For calculating the percentage of *diagonal density*, our algorithm at first find out the diagonal values where row and column indices are same then *nnz* elements will be divided by those diagonal values and for upper and lower triangle density our tools will find out the upper and lower values where row indices are less or greater then column indices, respectively then *nnz* elements again divided by the upper and lower values. Algorithm 8 describe the diagonal density function where upper and lower triangle also describe. After evaluating those results metrics will help to suggest developers for most suitable storage formats for their particular sparse matrix.

Algorithm 6 Row and Column major

```
1: function ROWCOLMAJOR(nnz)
2:     for row ← 1, nnz − 1 do
3:         if rowdata[row] > rowdata[row + 1] then
4:             dataorder == 1
5:         else
6:             dataorder == 0
7:         end if
8:     end for
9:     for col ← 1, nnz − 1 do
10:        if coldata[col] > coldata[col + 1] then
11:            dataorder == −1
12:        else
13:            dataorder == 0
14:        end if
15:    end for
16:    return dataorder
17: end function
```

V. Experimental Evaluation

A. Experimental Setup

The experiments have been performed on Aziz Super-Computer at King Abdulaziz University, Saudi Arabia. In experiment, one nodes with 16 processors is being requested.

Algorithm 7 Sparsity and Density of Matrix data

```
function SPARSIFY(row, col, nnz)
    rowCol = [row] * [col]
    zeroElement = rowCol − nnz
    sparsify = zeroElement / rowCol
    return sparsify
end function
function DENSITY(row, col, nnz)
    rowCol = [row] * [col]
    density = nnz / rowCol
    return density
end function
```

Algorithm 8 Upper, Lower Triangle and Diagonal Density

```
function DIAGONALDENSITY(row_indices, col_indices)
    for row ← 1, row_indecies do
        for col ← 1, col_indecies do
            if row < col then
                upper = nnz/(up ← up + 1)
            else if row > col then
                lower = nnz/(lw ← lw + 1)
            else if row = col then
                diagonal = nnz/(dia ← dia + 1)
            end if
        end for
    end for
    return upper, lower, diagonal
end function
```

TABLE II. METRICS WITH SPARSE MATRIX SPMV PERFORMANCE(GFLOPS)

mtx	Diagonal Density(%)	Numeric Symmetry(%)	Row Thickness	Avg *nnz values*	COO	CSR	IDX	ELL
sherman	4.003	25	7	4	1.514	1.788	1.715	1.718
lll_stokes	9.158	11	12	9	9.419	14	13.186	11.949
airfoil_2d	18.27	5.5	23	18	13.033	17.588	16.318	15.365
gt01r	54	1.9	75	53	20.352	26.369	24.153	20.228
cavity	30.29	3.3	62	30	9.506	11.222	10.799	9.145
bbmat	45.73	2.2	132	45	39.555	49.206	40.264	22.618
lowthrust	6.364	16	7184	6	4.641	8.726	8.947	6.360
windtunnel	66.9	0.99	93	66	17.089	59.388	47.609	98.01
circuit	3.969	25	5682	3	2.699	4.103	4.009	3.017
qa8fm	13.06	7.7	14	13	19.675	33.065	32.265	28.224
bcsstm	1	1.00E+02	1	1	0.901	1.346	1.315	1.322
robot	4.162	24	23	4	0.762	0.881	0.886	0.898
dw2048	4.938	20	8	4	0.847	0.922	0.958	0.936
nd24k	199.9	0.5	483	199	47.167	71.32	58.514	462.15
ga3as3h12	49.16	2	1024	49	29.524	81.224	43.407	105.96

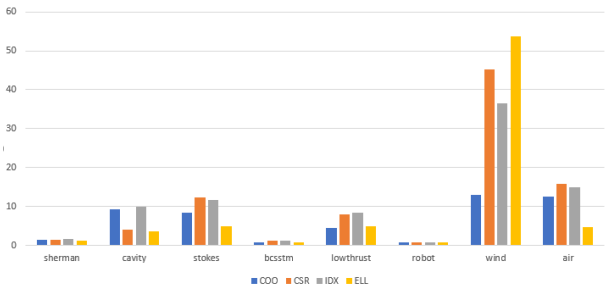


Fig. 9. SpMV performance in GFLOPS

TABLE III. PROPOSED STORAGE FORMATS BY AUTHORS

Authors name	Storage format	mtx	performance	Metrics
Bell and Garland [7]	HYB	Lowthrust	15.873	Pattern Symmetric =YES Numeric Symmetric=16% Horizontal and Vertical Sym = 7.9% Column Major Order =Yes Sparsity= 0.997 Density = 0.0003444 Diagonal Density = 6.364
		Circuit	5.987	Pattern Symmetric =YES Numeric Symmetric=25% Horizontal and Vertical Sym= 11% Column Major Order =Yes Sparsity= 0.997 Density = 0.0003273 Diagonal Density = 3.969
		Windtunnel	5.988	Pattern Symmetric =YES Numeric Symmetric=0.99% Horizontal and Vertical Sym = 7.1% Column Major Order =Yes Sparsity= 0.9984 Density = 0.001639 Diagonal Density = 66.9
Hiroki Yoshizawa [29]	CSR-T	qa8fm	5.988	Pattern Symmetric =YES Numeric Symmetric=7.7% Horizontal and Vertical Sym = 0.041% Column Major Order =Yes Sparsity= 0.9889 Density = 0.0111 Diagonal Density = 13.06
		ga3a3h2	9.001	Pattern Symmetric =YES Numeric Symmetric=2% Horizontal and Vertical Sym = 1% Column Major Order =Yes Sparsity= 1.006 Density = -0.005677 Diagonal Density = 49.16
		nd24k	12.214	Pattern Symmetric =YES Numeric Symmetric=0.5% Horizontal and Vertical Sym = 0.25% Column Major Order =Yes Sparsity= 0.9838 Density = 0.01619 Diagonal Density = 199.9
F.Vazquez [25]	Approach based on ELLR-T	rbsa480	3.4	Pattern Symmetric =YES Numeric Symmetric=2.8% Horizontal and Vertical Sym = 1.3% Column Major Order =Yes Sparsity= 0.9258 Density = 0.07417 Diagonal Density = 35.9
		dw2048	2.1	Pattern Symmetric =YES Numeric Symmetric=20% Horizontal and Vertical Sym = 7.2% Column Major Order =Yes Sparsity= 0.9976 Density = 0.002411 Diagonal Density = 4.938
		mhd32000	9.8	Pattern Symmetric =YES Numeric Symmetric=4.7% Horizontal and Vertical Sym = 2.4% Column Major Order =Yes Sparsity= 0.9934 Density = 0.006643 Diagonal Density = 21.26

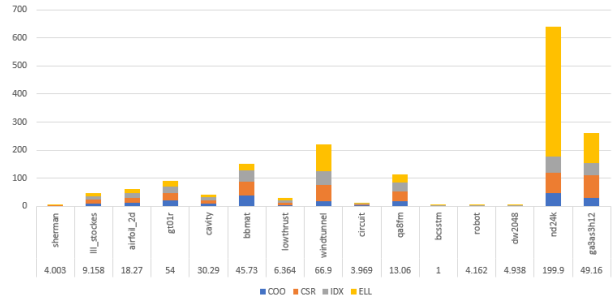


Fig. 10. SpMV performance evaluation respect to Diagonal Density

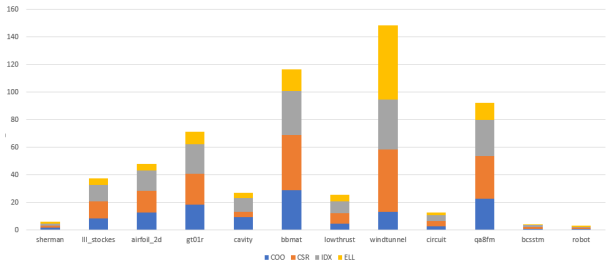


Fig. 11. SpMV performance(GFLOPS)

Each compute node has Dual socket hex-core processor running at 2.4GHz with 256GB RAM. Linux version 2.6.32-358.23.2.e16.x86_64 and GCC compiler version 4.4.7 is used in our experiment.

B. Metrics and Performance Analysis

In this section, Presents metrics evaluation using selected benchmark in previous chapter and also evaluate those similar benchmark according to the metrics behavior. In this approach, first generated metric is rowThickness. rowThickness is used to find out the maximum non-zero values per row. Row thickness is important for any sparse matrix because there is a relation between rowThickness and SpMV performance. If the row thickness does differ from the average non-zero values, then the ELLPACK storage format will not be a great choice for any data set. In this case, the experiment showed that CSR or any other storage format is more efficient then ELLPACK. From Table II section "Row Thickness" gives the generated rowThickness values for every sparse matrix. The sparse matrix characteristic's pattern symmetric is used in our approach for getting a clear visualization of the sparse matrix. If the sparse matrix is pattern symmetric any system can easily figure out that the row thickness does not very much higher from the average non-zero values.

Numeric, Horizontal and vertical symmetries are the other symmetric metrics of a sparse matrix, by using those parameters we can clearly understand the behavior and the structure of any sparse matrix. From the experiment, found that sparse matrix storage formats perform better with less numerically symmetric data. From the experiment, also found that if the percentage of numerically symmetric is less then horizontal and vertical ELLPACK perform better then other sparse matrix storage formats. In Table IV, will find out the symmetry information of selected sparse matrix. Row or column-major order showed in Table IV is another metric that is used in our approach. Experiment showed that, when a sparse matrix is in row-major order CSR and IDX perform better than other storage formats. The Table II sections "Sparsity and Density" gives the status of any matrix, those will justify the sparsity of any matrix using $sparsity = 1 - density$. Another section

TABLE IV. STORAGE FORMATS WITH METRICS

mtx file	metadata	mtx file	metadata
bcsstm	Pattern Symmetric =YES Numeric Symmetric=1e+02% Horizontal and Vertical Sym = 50% Column Major Order =Yes Sparsity= 0.9999 Density = 6.477e-05 Diagonal Density = 1	sherman	Pattern Symmetric =YES Numeric Symmetric=25% Horizontal and Vertical Sym = 9.7% Row Major Order =Yes Sparsity= 0.9992 Density = 0.0007997 Diagonal Density = 4.003
robot	Pattern Symmetric =YES Numeric Symmetric=24% Horizontal and Vertical Sym = 12% Column Major Order =Yes Sparsity= 0.9982 Density = 0.001765 Diagonal Density = 4.162	airfoil_2d	Pattern Symmetric =YES Numeric Symmetric=5.5% Horizontal and Vertical Sym = 2.7% Row Major Order =Yes Sparsity= 0.9987 Density = 0.001285 Diagonal Density = 18.27
gt10R	Pattern Symmetric =YES Numeric Symmetric=1.9% Horizontal and Vertical Sym = 0.93% Row Major Order =Yes Sparsity= 0.9932 Density = 0.006767 Diagonal Density = 54	bbmat	Pattern Symmetric =YES Numeric Symmetric=2.2% Horizontal and Vertical Sym = 1.1% Row Major Order =Yes Sparsity= 0.9988 Density = 0.00118 Diagonal Density = 45.73
Cavity26	Pattern Symmetric =YES Numeric Symmetric=3.3% Horizontal and Vertical Sym = 1.7% Row Major Order =Yes Sparsity= 0.9934 Density = 0.00664 Diagonal Density = 30.29	lll_stokes	Pattern Symmetric =YES Numeric Symmetric=11% Horizontal and Vertical Sym = 5.5% Row Major Order =Yes Sparsity= 0.9996 Density = 0.0004383 Diagonal Density = 9.158

from Table II diagonal density provide the description about diagonal density. Diagonal density is very important for any sparse matrix. Storage formats performance can determined by the percentage of diagonal density. Experiments showed that for higher percentage of diagonal density ELLPACK perform better then CSR. Fig. 10 showed performance of SpMV for different storage formats with different sparse matrix respect to diagonal density. Fig. 11 presents the performance graph respect to COO, CSR, IDX and ELLPACK. ELLPACK perform peak performance with *windtunnel* sparse matrix, where the percentage of numerical symmetry is 0.99%, diagonal density 66.9% and column major order. ELLPACK perform worst performance with bcsstm sparse matrix, where the percentage of numerical symmetry is 1e+02%, diagonal density 1% and column major order. After experimental analysis we found that ELLPACK perform best with higher diagonal density and give lowest performance where diagonal density is very low. CSR perform best among the sparse matrix with bbmat, where the percentage of numerical symmetry is 2.2%, diagonal density 45% and row-major order data.While the percentage of numerical symmetry 24%, diagonal density is 4.162% and column-major order then CSR perform worst.Our experimental result showed that CSR perform worst when the nature of the sparse matrix is column-major order. Any row-major order sparse matrix with better percentage of diagonal density CSR perform peak performance. For any column-major order sparse matrix we can use another solution which also compressed the data

but here the format used column pointer instead of row pointer, named CSC. On the other hand COO performed average for all the sparse matrix, COO give us its peak performance with bbmat but the performance is not greater than CSR. COO also perform worst with bcsstm, where the percentage of numerical symmetry is 1e+02%, diagonal density 1% and column-major order. For bcsstm sparse matrix CSR gives better performance than other three storage format used in our literature. Another newly storage format which we used in our tools is IDX, these storage format perform better then all others with lowthrust sparse matrix, where the percentage of numerical symmetry 16%, diagonal density is 6.364% and column-major order. IDX performance is always very close to CSR.

VI. Conclusion

SpMV operation's performance affects the efficiency of large number of applications like CFD, computer graphics, robotics, structural problems, acoustics problems, etc. The performance of SpMV operations depends on many factors. Matrix characteristics, storage formats, software and hardware implementations are most common problems for the performance of SpMV operation.

In this paper, we focused on matrix characteristics, an approach has been proposed and evaluated to assume the matrix characteristics by generating several metrics. Our approach allows the developers to choose the most suitable storage formats for given sparse matrix and improve the performance of SpMV operation. The aim of our approach is suggest the best suitable storage format for any sparse matrix by analyzing our generated metrics. We have evaluate our approach using 15 real world sparse matrices from University of Florida sparse matrix collection (UFSMC) [30] with four most popular storage formats. Along with some well-known metrics we also generate some new metrics in our approach. In our evaluation we observed that, CSR perform best when the row thickness is very large that means CSR is a better choice for horizontal substructures. ELLPACK perform best with higher diagonal density (49.16%) when the numeric symmetry (2%) is very less.

In future we will extend our research with newer metrics to describe the sparse matrix with metadata more accurately. We will incorporate other common dependency for SpMV operation and increasing both sparse matrix and storage format. We will enhance our research by incorporating machine learning.

Acknowledgment

Experiment for the work presented in this paper was supported by High Performance Computing Center (Aziz Supercomputer) at King Abdulaziz University.

References

[1] G. Goumas, K. Kourtis, N. Anastopoulos, V. Karakasis, and N. Koziris, "Performance evaluation of the sparse matrix-vector multiplication on modern architectures," *J. Supercomput.*, vol. 50, no. 1, pp. 36–77, oct 2009. [Online]. Available: https://doi.org/10.1007/s11227-008-0251-8

[2] K. Kourtis, V. Karakasis, G. Goumas, and N. Koziris, "CSX: An extended compression format for SpMV on shared memory systems," *ACM SIGPLAN Not.*, vol. 46, no. 8, pp. 247–256, 2011.

[3] W. DGROPP, D. KKAUSHIK, D. EKEYES, and G. BFSMITH, "Towards Realistic Performance Bounds for Implicit CFD Codes," *Parallel Comput. Fluid Dyn. 1999*, pp. 241–248, 2000.

[4] K. Kourtis, G. Goumas, and N. Koziris, "Optimizing Sparse Matrix-Vector Multiplication using index and value compression," *Conf. Comput. Front. - Proc. 2008 Conf. Comput. Front. CF'08*, pp. 87–96, 2008.

[5] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, "Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks," *Annu. ACM Symp. Parallelism Algorithms Archit.*, pp. 233–244, 2009.

[6] J. Willcock and A. Lumsdaine, "Accelerating sparse matrix computations via data compression," *Proc. Int. Conf. Supercomput.*, pp. 307–316, 2006.

[7] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *Proc. Conf. High Perform. Comput. Networking, Storage Anal. - SC '09*, no. 1. New York, New York, USA: ACM Press, 2009, p. 1. [Online]. Available: http://dl.acm.org/citation.cfm?doid=1654059.1654078

[8] Yousef Saad, "Iterative Methods for Sparse Linear Systems: Second Edition - Yousef Saad - Google Books," 2003.

[9] M. Heller and T. Oberhuber, "Adaptive Row-grouped CSR Format for Storing of Sparse Matrices on GPU," 2012. [Online]. Available: http://arxiv.org/abs/1203.5737

[10] A. Ashari, N. Sedaghati, J. Eisenlohr, and P. Sadayappan, "An efficient two-dimensional blocking strategy for sparse matrix-vector multiplication on GPUs," in *Proc. 28th ACM Int. Conf. Supercomput. - ICS '14*. New York, New York, USA: ACM Press, 2014, pp. 273–282. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2597652.2597678

[11] D. Guo and W. Gropp, "Applications of the streamed storage format for sparse matrix operations," *Int. J. High Perform. Comput. Appl.*, vol. 28, no. 1, pp. 3–12, 2014.

[12] M. Maggioni and T. Berger-Wolf, "AdELL: An adaptive warp-Balancing ELL format for efficient sparse matrix-Vector multiplication on GPUs," *Proc. Int. Conf. Parallel Process.*, pp. 11–20, 2013.

[13] C. Zheng, S. Gu, T. X. Gu, B. Yang, and X. P. Liu, "BiELL: A bisection ELLPACK-based storage format for optimizing SpMV on GPUs," *J. Parallel Distrib. Comput.*, vol. 74, no. 7, pp. 2639–2647, 2014. [Online]. Available: http://dx.doi.org/10.1016/j.jpdc.2014.03.002

[14] J. Hartmanis and J. V. Leeuwen, *High Performance Embedded Architectures and Compilers 5th*, J. v. L. Gerhard Goos, Juris Hartmanis, Ed., 2010, vol. 9, no. 3.

[15] K. Li, W. Yang, and K. Li, "Performance analysis and optimization for SpMV on GPU using probabilistic modeling," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 1, pp. 196–205, 2015.

[16] R. Shahnaz, A. Usman, and I. R. Chughtai, "Review of Storage Techniques for Sparse Matrices," in *2005 Pakistan Sect. Multitopic Conf.* IEEE, dec 2005, pp. 1–7. [Online]. Available: http://ieeexplore.ieee.org/document/4133468/

[17] Y. Nagasaka, A. Nukada, and S. Matsuoka, "Adaptive Multi-level Blocking Optimization for Sparse Matrix Vector Multiplication on GPU," *Procedia Comput. Sci.*, vol. 80, pp. 131–142, 2016. [Online]. Available: http://dx.doi.org/10.1016/j.procs.2016.05.304 https://linkinghub.elsevier.com/retrieve/pii/S187705091630655X

[18] I. P. Stanimirovic and M. B. Tasic, "Performance comparison of storage formats for sparse matrices," *Ser. Math. Informatics*, vol. 24, no. 1, pp. 39–51, 2009. [Online]. Available: http://facta.junis.ni.ac.rs/mai/mai24/fumi-24_39_51.pdf

[19] E. Montagne and A. Ekambaram, "An optimal storage format for sparse matrices," *Inf. Process. Lett.*, vol. 90, no. 2, pp. 87–92, 2004.

[20] D. Buono, F. Petrini, F. Checconi, X. Liu, X. Que, C. Long, and T.-C. Tuan, "Optimizing Sparse Matrix-Vector Multiplication for Large-Scale Data Analytics," *Proc. 2016 Int. Conf. Supercomput. - ICS '16*, pp. 1–12, 2016. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2925426.2926278

[21] J. Zhang, J. Wan, F. Li, J. Mao, L. Zhuang, J. Yuan, E. Liu, and Z. Yu, "Efficient sparse matrix-vector multiplication using cache oblivious extension quadtree storage format," *Futur. Gener. Comput. Syst.*, vol. 54, pp. 490–500, 2016. [Online]. Available: http://dx.doi.org/10.1016/j.future.2015.03.005

[22] X. Feng, H. Jin, R. Zheng, K. Hu, J. Zeng, and Z. Shao, "Optimization of sparse matrix-vector multiplication with variant CSR on GPUs," *Proc. Int. Conf. Parallel Distrib. Syst. - ICPADS*, pp. 165–172, 2011.

[23] D. Hutchison and J. C. Mitchell, *Lecture Notes in Computer Science*, 1973, vol. 9, no. 3. [Online]. Available: http://www.mendeley.com/research/lecture-notes-computer-science-2/

[24] X. Yang, S. Parthasarathy, P. Sadayappan, H. Yoshizawa, D. Takahashi, E. Montagne, A. Ekambaram, M. E. Epstein, I. Rodan, G. Griffenhagen, J. Kadrlik, M. C. Petty, S. A. Robertson, and W. Simpson, "Fast sparse matrix-vector multiplication on GPUs," *Proc. VLDB Endow.*, vol. 90, no. 2, pp. 130–136, mar 2012. [Online]. Available: http://journals.sagepub.com/doi/10.1177/1098612X15572062 http://dl.acm.org/citation.cfm?doid=1938545.1938548

[25] F. Vázquez, J. J. Fernández, and E. M. Garzón, "Automatic tuning of the sparse matrix vector product on GPUs based on the ELLR-T approach," *Parallel Comput.*, vol. 38, no. 8, pp. 408–420, 2012.

[26] M. M. Baskaran, R. Bordawekar, M. Manikandan, and B. Rajesh Bordawekar, "Optimizing Sparse Matrix-Vector Multiplication on GPUs Using Compile-time and Run-time Strategies," Tech. Rep., 2008. [Online]. Available: http://domino.watson.ibm.com/library/CyberDig.nsf/home.

[27] S. Chen, J. Fang, D. Chen, C. Xu, and Z. Wang, "Adaptive Optimization of Sparse Matrix-Vector Multiplication on Emerging Many-Core Architectures," *2018 IEEE 20th Int. Conf. High Perform. Comput. Commun. IEEE 16th Int. Conf. Smart City; IEEE 4th Int. Conf. Data Sci. Syst.*, pp. 649–658, 2018.

[28] W. Liu and B. Vinter, "CSR5 : An Efficient Storage Format for Cross-Platform Sparse Matrix-Vector Multiplication Categories and Subject Descriptors," pp. 339–350.

[29] H. Yoshizawa and D. Takahashi, "Automatic Tuning of Sparse Matrix-Vector Multiplication for CRS Format on GPUs," in *2012 IEEE 15th Int. Conf. Comput. Sci. Eng.* IEEE, dec 2012, pp. 130–136. [Online]. Available: http://ieeexplore.ieee.org/document/6417285/

[30] T. A. Davis and Y. Hu, "The university of Florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1–25, 2011.