# Multiplication of Sparse Matrices and their Transpose using Compressed Sparse Diagonals

1st Sardar Anisul Haque
*School of Computing and Data Science*
*Oryx Universal College in Partnership with LJMU (UK)*
Doha, Qatar
haque.sardar@gmail.com

2nd Mohammad Tanvir Parvez
*Department of Computer Engineering, College of Computer*
*Qassim University*
Buraydah 52571, Saudi Arabia
m.parvez@qu.edu.sa

3rd Shahadat Hossain
*Department of Computer Science*
*University of Northern British Columbia*
Prince George, BC V2N 4Z9, Canada
shahadat.hossain@unbc.ca

***Abstract*—Matrix-matrix multiplication is one of the most important kernel in linear algebra operations with a multitude of applications in scientific and engineering computing. Sparse matrix computation on modern High-performance Computing (HPC) architecture presents challenges such as load balancing, data locality optimization, and computational scalability. Data structures to store sparse matrices are designed to minimize overhead information as well as to optimize the operations count and memory access. In this study, we present a new data structure, "compressed sparse diagonal" (CSD), to efficiently store and compute with general sparse matrices. The CSD builds upon the previously developed diagonal storage - an orientation-independent uniform scheme to compute with "structured" matrices [1]. Compared with the widely used compressed sparse row/column (CSR/CSC), the CSD scheme avoids explicit transposition operation when multiplying a matrix with its transpose. The results from preliminary numerical experiments with the aforementioned types of matrices demonstrate the CSD scheme's effectiveness in matrix-transposed matrix multiplication.**

***Index Terms*—Matrix Multiplication, Compressed Diagonal, Sparse Matrix, Matrix Transpose**

## I. Introduction

Matrix-matrix multiplication where the arguments are sparse is a fundamental computational kernel in numerous scientific and graph applications such as computer graphics, network analytics, algebraic multigrid solvers, triangle counting, multi-source breadth-first searching, shortest path problems, colored intersecting, and subgraphs matching [2]–[5]. An efficient implementation of this ubiquitous operation on modern computing architectures is challenging because of irregular memory access and often becomes the performance bottleneck. In particular, fast algorithms for computing strongly connected component of a sparse graph [6], [7] (represented as a sparse matrix $A$) require access to $A$ and $A^\top$. It has been experimentally shown that transposition operation does not scale well on multicore processors [8]. In this paper we present a novel data structure to compute products of the form $AA^\top, A^\top A$, for a general sparse matrix $A$ where explicit transposition of $A$ is avoided. Built upon its predecessor for efficient computation with "banded (structured)" matrices [1], the proposed sparse matrix data structure exhibit algorithmic and data abstraction properties similar to the widely used CSR/CSC.

There exists a wide variety of storage formats to match various features of sparse matrices [9]. In the case of general sparse matrices, traditional triple-loop matrix multiplication employs row-wise access with Compressed Sparse Row (CSR) or the transposed access with Compressed Sparse Column (CSC). The diagonal storage format [10] stores elements in every diagonal and is found to be suitable for storing and computing with banded matrices. A variant of it, Ellpack-ltpack [11], is a generalized diagonal format that uses an $n \times k$ array where $k$ denotes the maximum number of non-zero elements in a row.

The advantages of the orientation-neutral matrix storage scheme of [1], [12], along with efficient cache usage and reduced storage requirements, as identified in [13] are:

1) a one-dimensional storage alternative without padding thus improving the previous diagonal storage schemes that utilize a two-dimensional array to store the diagonals,
2) efficient scaling over increased bandwidth of the matrices and parallel processing of the independent diagonal calculations.

The main contributions of the paper are summarized below.

- To the best of our knowledge the compressed sparse diagonal storage scheme is the first sparse matrix data structure for general sparse matrices that is neutral of column/row-oriented data layout.
- A detailed algorithmic description of CSD matrix-matrix multiplication is given in terms of the sum-product calculations that occurs in the innermost loop of the triple-loop algorithm. This procedure effectively provides conversion between CSD and CSR/CSC formats.

- We present preliminary numerical results for serial and multi-core shared memory parallel implementation of the operation $AA^\top$ on large sparse benchmark instances from the literature. The numerical results show that our non tuned baseline implementation can be highly efficient as it avoids explicit transposition.

The rest of the paper is organized as follows. Section II introduces the compressed sparse diagonal and discusses it under the traditional triple-loop matrix multiplication scenario. The sparse matrix multiplication algorithm using CSD data structure is described in Section III. Performance comparisons that demonstrate the efficiency of our approach is given in Section IV. Section V summarizes the paper with notes on future research directions.

## II. Conversion from Compressed Sparse Row to Diagonal Storage Scheme

Given a sparse matrix in Compressed Sparse Row (CSR) format, one can find out the column and row indices of each nonzero element by looking its column index and the row offsets respectively. Logically, the conversion from CSR to dense coordinate storage scheme is similar to conversion from coordinate representation of a matrix to its diagonal storage scheme. In this section, we are going to show such formula. In addition, we will introduce a CSR type data structure for diagonal storage scheme for sparse matrices.

In a square matrix of size $n \times n$, we have $2n-1$ diagonals. We can divide these diagonals into two groups:

1) Non-negative diagonals: this group of diagonals includes the principle diagonal and all diagonals above the principle diagonal (also termed super diagonals). The diagonals in this group are labelled or numbered as $0, 1, \ldots, n-1$, where $0$ is the label of the principal diagonal.
2) Sub or negative diagonals: this group of diagonals includes all diagonals below the principle diagonal. The diagonals in this group are labelled as $-1, -2, -3, -(n-1)$, where $-1$ is the label of the diagonal immediately below the principal diagonal.

The number of elements in the $k^{th}$ diagonal is $n - |k|$, $k = -(n-1), -(n-2), ..., -1, 0, 1, ..., (n-1)$. Below is an example of a square matrix $A$ of size $4 \times 4$ and its 7 diagonals.

$$A = \begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix}$$

$$\mathbf{diag}[\mathbf{0}] = [a_{00}, a_{11}, a_{22}, a_{33}]$$

$$\mathbf{diag}[\mathbf{1}] = [a_{01}, a_{12}, a_{23}]$$

$$\mathbf{diag}[\mathbf{2}] = [a_{02}, a_{13}]$$

$$\mathbf{diag}[\mathbf{3}] = [a_{03}]$$

$$\mathbf{diag}[-\mathbf{1}] = [a_{10}, a_{21}, a_{32}]$$

$$\mathbf{diag}[-\mathbf{2}] = [a_{20}, a_{31}]$$

$$\mathbf{diag}[-\mathbf{3}] = [a_{30}]$$

Note that, elements in a diagonal of $A$ are listed in the order of row numbers of the elements in $A$.

Now, we have two observations regarding the diagonal representation given above. First, if $a_{i,j}$ appears in diagonal $diag[k]$ of a matrix $A$, then $a_{i+1,j+1}$ is the element after $a_{i,j}$ in $diag[k]$. Second, $a_{0,i}$ and $a_{i,0}$ are the first elements in $i^{th}$ and $-i^{th}$ diagonals, respectively.

We claim that $a_{i,j}$ is contained in $(j-i)^{th}$ diagonal at $\min\{i,j\}^{th}$ position. To see this, consider the index pair $(i,j)$. If $i = j$, the element is located on the principle diagonal labelled $j - i = 0$ at position $l = i = j$ where $l = 0, 1, \ldots, (n-1)$ as can be directly verified. If $i < j$, then $(i,j)^{th}$ element is located on diagonal $k = (j-i) > 0$. The elements on the $k^{th}$ diagonal are indexed $i = 0, 1, \ldots, (n-k-1)$. Also $i < j$ implies $\min\{i,j\} = i$. Therefore, $(i,j)^{th}$ entry of matrix $A$ is the $i^{th}$ entry of diagonal $k$. The case for $j < i$ is analogous.

An element $a$ in $A$ can be identified with two indices $(k,p)$ in the diagonal storage scheme. Here, $k$ is the diagonal number or label and $p$ is the position or index of $a$ in the diagonal. Therefore, given an element $a_{i,j}$ of $A$ (in coordinate system), we represent $a_{i,j}$ in diagonal storage scheme as $a^d(k,p)$, where $k = j - i$ and $p = \min\{i,j\}$. Thus, the elements of, for example, $-1^{st}$ diagonal of $A$ can be written as follows.

$$\mathbf{diag}[-\mathbf{1}] = [a_{1,0}, a_{2,1}, a_{3,2}] = [a^d_{-1,0}, a^d_{-1,1}, a^d_{-1,2}]$$

We now propose a new data structure for general sparse matrix. Our data structure is based on diagonal storage scheme and is similar to CSR. We call our scheme *Compressed Sparse Diagonal* or *CSD*. We can describe *CSD* structure using three arrays:

1) An array `nonZeros`: Its size is equal to the number of nonzeros in the sparse matrix $A$. The nonzeros of $A$ are laid out in the following order. As mentioned earlier, for a matrix $A$ of size $n \times n$, the diagonals of $A$ are labelled as $-(n-1), -(n-2), ..., -1, 0, 1, ..., (n-1)$. To build the array of nonzero elements in $A$, we consider the elements sequentially in each diagonal of $A$, where the diagonals of $A$ are considered in this order: $0, 1, ..., (n-1), -1, -2, ..., -(n-1)$. That means, the elements of the principle diagonal are considered first, followed by the super diagonals and then sub diagonals.
2) `posID`: it is an integer array of size equal to the number of nonzero elements in the sparse matrix. This array stores the indices of each nonzero element of $A$ in the corresponding diagonal it appears (the $p$ value in $a^d(k,p)$ above).
3) `diagOffsets`: it is an integer array of size $2n$. The $i^{th}$ entry of this array is the index of a value in `nonZeros` array, which is the first nonzero value in diagonal number $i$ (if $i < n$) or in diagonal number $i - n - 1$ (if $i \geq n$).

To illustrate, consider the following matrix $A$ shown in the coordinate scheme. For clarity, we list the elements (including zeros) in each diagonal of $A$.

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

$$\textbf{diag}[\mathbf{0}] = [1, 0, 0, 6]$$

$$\textbf{diag}[\mathbf{1}] = [0, 3, 0]$$

$$\textbf{diag}[\mathbf{2}] = [0, 0]$$

$$\textbf{diag}[\mathbf{3}] = [2]$$

$$\textbf{diag}[-\mathbf{1}] = [0, 0, 5]$$

$$\textbf{diag}[-\mathbf{2}] = [4, 0]$$

$$\textbf{diag}[-\mathbf{3}] = [0]$$

The matrix $A$ represented in CSD scheme consists of three arrays as shown below.

$$\texttt{nonZeros} = [1, 6, 3, 2, 5, 4]$$

$$\texttt{posID} = [0, 3, 1, 0, 2, 0]$$

$$\texttt{diagOffsets} = [0, 2, 3, 3, 4, 5, 6, 6]$$

### *A. Getting to the Transpose*

As discussed in the Introduction, matrix transposition is an important computational kernel and arises as a building block of graph algorithms [4]. In mobile robotics, one of the most computationally intensive operations is the computation of so called information matrix $\mathcal{I} = A^\top A$ where $A$ is sparse and the sparsity changes in each of the outer iteration [14]. In the accompanying numerical experiments the authors report that the computation of $\mathcal{I}$ accounts approximately six times the cost of computing the Cholesky factorization of matrix $A$.

The transpose of $A$ in CSD is readily available. For each index pair $(i, j)$, its transposed position is $(j, i)$ such that entries in diagonal $k = (j-i)$ of matrix $A$ is found in diagonal $(-k = i - j)$. Thus, no additional storage or calculation is needed to obtain the transposed access in CSD.

## III. Sparse Matrix-Matrix Multiplication Using CSD

Given two matrices $A$ and $B$, where both of them are of size $n \times n$, we need $n^3$ scalar multiplications for computing $C = A * B$ using traditional nested three for-loops algorithm. The innermost for-loop executes the following code: $C(i, k) \mathrel{+}= A(i, j) * B(j, k)$. We can rewrite this code when the matrices are stored in CSD format. The details are discussed below.

### *A. Matrix Multiplication in CSD*

When all the matrices are stored in CSD format, the code $C(i, k) \mathrel{+}= A(i, j) * B(j, k)$ takes the form: $c^d(k - i, \min\{i, k\}) \mathrel{+}= a^d(j - i, \min\{i, j\}) * b^d(k - j, \min\{j, k\})$. Here, all three indices $i, j, k$ have the same range: $[0, 1, \ldots, n-1]$.

For dense matrix-matrix multiplication in coordinate storage scheme, the result $C_{ik}$ is obtained as the inner product of row $i$ of matrix $A$ and column $k$ of matrix $B$. With diagonal storage scheme, the corresponding elements are identified by their diagonal index and the position in the diagonal. We now show six inequalities among $i, j$ and $k$ that cover all possible scenarios.

1) ***Case 1***: Multiplying an element from a non-negative diagonal of $A$ with an element from a non-negative diagonal of $B$. Therefore, $i \le j \le k$. In this case $k - i \ge 0$. So, this type of multiplication will contribute to a non-negative diagonal of matrix $C$. We can simplify the computation as $c^d(k-i, i) \mathrel{+}= a^d(j-i, i) * b^d(k-j, j)$. We can rewrite it as $c^d(x+y, i) \mathrel{+}= a^d(x, i) * b^d(y, x+i)$ where $x = j - i \ge 0$ and $y = k - j \ge 0$. Note that, $x$ and $y$ represent the diagonal numbers of the elements from $A$ and $B$, respectively. The result of the multiplication is stored in an element of $C$ whose diagonal number is $x + y$. Therefore, Case 1 type of multiplication occurs only when $0 \le (x + y) < n$.
The location of the elements in their respective diagonals from the three matrices are obtained from the indices $i, j, x, y$ as shown earlier.
2) ***Case 2 and 3***: Multiplying an element from a non-negative diagonal of $A$ with an element from a sub diagonal of $B$. Therefore, $i \le j$ and $j > k$. We have two scenarios based on the values of $i$ and $k$.
   a) Case 2: if $i \le k$, we can rewrite the computation of $c(i, k)$ as: $c^d(k-i, i) \mathrel{+}= a^d(j-i, i) * b^d(k-j, k)$. We can further simplify it as $c^d(x + y, i) \mathrel{+}= a^d(x, i) * b^d(y, x + y + i)$ where $x = j - i$ and $y = k - j$. Result of this multiplication contributes to an element of $C$ whose diagonal number is $(k - i) = (x + y)$, where $x$ and $y$ are the diagonal numbers of $b$ and $a$ respectively. Note that, $(k - i) \ge 0$ here, as $i \le k$. That means, Case 2 type of multiplications contribute to the elements in non-negative diagonals of $C$.
   b) Case 3: if $i > k$, we can rewrite the computation of $c(i, k)$ as: $c^d(k-i, k) \mathrel{+}= a^d(j-i, i) * b^d(k-j, k)$. We can further simplify it as: $c^d(x + y, k) \mathrel{+}= a^d(x, i) * b^d(y, x + y + i)$ where $x = j - i$ and $y = k - j$. Similar to the previous cases, result of this multiplication contributes to an element of $C$ whose diagonal number is $(k - i) = (x + y)$, where $x$ and $y$ are the diagonal numbers of $b$ and $a$ respectively. However, $(k - i) < 0$ here, as $i > k$. That means, Case 3 type of multiplications contribute to the elements in sub diagonals of $C$,

as opposed to Case 2.

3) ***Case 4 and 5***: Multiplying an element from a sub diagonal of $A$ with an element from a non-negative diagonal of $B$. Therefore, $i > j$ and $j \leq k$. Again, analogous to Cases 2 and 3, we have two scenarios based on the values of $i$ and $k$.
    a) Case 4: if $i \leq k$ we can rewrite the computation of $c(i,k)$ as: $c^d(k-i,i) \mathrel{+}= a^d(j-i,j) * b^d(k-j,j)$. We can further simplify it as: $c^d(x+y, j-x) \mathrel{+}= a^d(x,j) * b^d(y,j)$ where $x = j-i$ and $y = k-j$. Similar to the previous cases, result of this multiplication contributes to an element of $C$ whose diagonal number is $(k-i) = (x+y)$, where $x$ and $y$ are the diagonal numbers of $b$ and $a$ respectively. Note that, $(k-i) \geq 0$ here, as $i \leq k$. That means, Case 4 type of multiplications contribute to the elements in non-negative diagonals of $C$.
    b) Case 5: if $i > k$, we can rewrite the computation of $c(i,k)$ as: $c^d(k-i,k) \mathrel{+}= a^d(j-i,j) * b^d(k-j,j)$. We can further simplify it as: $c^d(x+y, j+y) \mathrel{+}= a^d(x,j) * b^d(y,j)$ where $x = j-i$ and $yk-j$. Similar to the previous cases, result of this multiplication goes to an element of $C$ whose diagonal number is $(k-i) = (x+y)$, where $x$ and $y$ are the diagonal numbers of $b$ and $a$ respectively. However, as in Case 3, $(k-i) < 0$ here, as $i > k$. That means, Case 5 type of multiplications contribute to the elements in sub diagonals of $C$.
4) ***Case 6***: Multiplying an element from a sub diagonal of $A$ with an element from a sub diagonal of $B$. Therefore, $i > j > k$.
   We can rewrite the computation of $c(i,k)$ as: $c^d(k-i,k) \mathrel{+}= a^d(j-i,j) * b^d(k-j,k)$. We can further simplify it as: $c^d(x+y,k) \mathrel{+}= a^d(x,k-y) * b^d(y,k)$ where $x = j-i$ and $y = k-j$ with $x$ and $y$ respectively. Note that, the diagonal number of $c$ in this case is $(k-i) < 0$, as $i > k$. Therefore, the condition $-1 \geq (k-i) \geq -(n-1)$ must satisfy for Case 6. This means, Case 6 type of multiplications contribute to the elements in sub diagonals of $C$.

The above analysis effectively gives an algorithm for performing matrix-matrix multiplication where the elements are stored in CSD scheme.

## *B. Illustrative Example*

Let $A$ and $B$ be two matrices of size $3 \times 3$:

$$A = \begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{pmatrix}, \quad B = \begin{pmatrix} b_{00} & b_{01} & b_{02} \\ b_{10} & b_{11} & b_{12} \\ b_{20} & b_{21} & b_{22} \end{pmatrix}.$$

To compute $C = A*B$, we need a total of 27 multiplication operations using the traditional triple-loop algorithm. As an illustration, among these 27 required symbolic multiplications of the element of $A$ and $B$, we list below nine multiplication operations that contribute to the elements of diagonal number 1 and $-2$ of the resultant matrix $C$. For each multiplication operation, we also show its indices in CSD scheme with case number.

Table I: Mapping traditional matrix multiplications to the cases in CSD multiplications.

| Coordinate format | CSD format | Case number |
|---|---|---|
| $a_{00}b_{01}$ contributes to $c_{01}$ | $a^d_{00}b^d_{10}$ contributes to $c^d_{10}$ | Case 1 |
| $a_{01}b_{11}$ contributes to $c_{01}$ | $a^d_{10}b^d_{01}$ contributes to $c^d_{10}$ | Case 1 |
| $a_{02}b_{21}$ contributes to $c_{01}$ | $a^d_{20}b^d_{-11}$ contributes to $c^d_{10}$ | Case 2 |
| $a_{10}b_{02}$ contributes to $c_{12}$ | $a^d_{-10}b^d_{20}$ contributes to $c^d_{11}$ | Case 4 |
| $a_{11}b_{12}$ contributes to $c_{12}$ | $a^d_{01}b^d_{11}$ contributes to $c^d_{11}$ | Case 1 |
| $a_{12}b_{22}$ contributes to $c_{12}$ | $a^d_{11}b^d_{02}$ contributes to $c^d_{11}$ | Case 1 |
| $a_{20}b_{00}$ contributes to $c_{20}$ | $a^d_{-20}b^d_{00}$ contributes to $c^d_{-20}$ | Case 5 |
| $a_{21}b_{10}$ contributes to $c_{20}$ | $a^d_{-11}b^d_{-10}$ contributes to $c^d_{-20}$ | Case 6 |
| $a_{22}b_{20}$ contributes to $c_{20}$ | $a^d_{02}b^d_{-20}$ contributes to $c^d_{-20}$ | Case 3 |

For our sparse matrix-matrix multiplication, we assume that the indices in `posID` are stored in increasing order. The algorithm works as follows.

1) We have two loops. The outer and inner loops works for each diagonal of $C$ and $A$ respectively. For each pair of diagonals in $C$ and $A$, we find the diagonal in $B$, whose nonzeros can multiply with those of $A$ and can contribute to that diagonal of $C$.
2) In the inner loop, based on the diagonals from $A$, $B$ and $C$, the multiplication rule falls into one of the 6 categories described above.
3) Before entering the inner loop, we initialize a zero vector of length equal to the length of the corresponding diagonal of matrix $C$. This vector act as a dense representation of the corresponding diagonal of $C$.
4) Once the diagonal of $C$ is computed in the inner loop by enumerating all possible pairs between $A$ and $B$, we copy the elements from the vector to the `nonZeros` array of $C$'s CSD representation.

In numerical experiments described in the next section, we assume that the sparsity pattern of the result matrix $C$ has already been computed. This is done to enable a fair performance comparison of CSD and CRS matrix-matrix multiplication on a set of standard benchmark instances.

## IV. Numerical Experiments

In this section we present test results from a preliminary computational study of our compressed sparse diagonal storage scheme for sparse matrix-matrix algorithm from Section III. All experiments are carried out on a computer with i7-13700H (14 core, 2.40 GHz base speed) processor containing 32 GB RAM and running Window 11 Pro operating system. The implementation language is VISUAL C++.

We use a select subset of test matrices from SuiteSparse Matrix Collection, also used in [3]. As a proof-of-concept implementation, we do not use any compiler optimization flag in our numerical testing and use `float` data type for test matrices to enable largest test instances possible on the

| matrix Name ($A$) | no. of rows | no. nonzeros in $A$ | no. of nonzeros in $A^\top A$ | total multiplication in computing $A^\top A$ | time (in sec) to compute $A^\top$ | time (in sec) to compute $A^\top A$ in CSR | time (in sec) to compute $A^\top A$ in CSD |
|---|---|---|---|---|---|---|---|
| Chebyshev4 | 68121 | 5377761 | 18870336 | 277056025 | 5.31 | 90.151 | 106.65 |
| Goodwin_127 | 178437 | 5778545 | 26891956 | 231353414 | 4.846 | 2.77 | **1.773** |
| marine1 | 400320 | 6226538 | 28952191 | 98739729 | 5.608 | 4.35 | 160.649 |
| ohne2 | 181343 | 11063545 | 72432047 | 755597643 | 10.761 | 20.155 | 1767.15 |
| PR02R | 161070 | 8185136 | 30969453 | 471911391 | 7.344 | 7.361 | **8.957** |
| test1 | 392908 | 12968200 | 75250370 | 468508270 | 12.811 | 62.757 | 701.495 |
| torso1 | 116158 | 8516500 | 19600778 | 3303780828 | 7.887 | 76.099 | 182.173 |
| TSOPF_RS_b678_c2 | 35696 | 8781949 | 40139105 | 69226453 | 8.292 | 141.846 | **3.199** |
| TSOPF_RS_b2052_c1 | 25626 | 6761100 | 30950093 | 53335853 | 6.316 | 89.397 | **2.76** |
| largebasis | 440020 | 5560100 | 17040280 | 70840580 | 5.132 | 1.671 | 929.294 |

Table II: Runtime comparison of matrix multiplication operation for CSR and CSD

hardware used. All experiments are carried on a machine i7-13700H (14 core, 2.40 GHz base speed), 32 GB RAM with Window 11 Pro operating system.

Table II displays the test matrix statistics (matrix name, number of rows (=number of columns), number of nonzeros), the time in seconds required to compute explicit transposition and to compute the product $A^\top A$, and the total number of scalar multiplications. The comparative timing results are mixed for CSD and CSR. Test instances where CSD performs better are shown in boldface. Closer examination of the test instances (see Figure 2) reveals that, the sparsity pattern of instances TSOPF_RS_b678_c1, TSOPF_RS_b678_c2, and Goodwin are "strucured" in that several "band-type" nonzero patterns can be visualized and CSR being most suitable for general sparsity pattern, the memory access overhead dominates possibly due to relatively small number of nonzeros in rows. On the other hand, CSD naturally exploits the banded sections in the matrix.

In our next experiment, we employ multicore capability of the computer under OpenMP environment to test CDS multiplication. Figure 1 displays the speedup obtained over serial execution. Only the outer for-loop is parallelized with default scheduling of the threads. The speedup over sequential execution varies between 2x and 6x. This represents a reasonably good speedup considering the implementation of CSD being "baseline".

Among the matrix instances it is evident that best speedup performance is obtained with ohne2. This is most likely due to the relatively high volume of work available to the threads as indicated by the sparsity pattern of matrix ohne in Figure 2.

Although the results from limited experiments provide some general trend on the performance of CSD based matrix multiplication, more elaborate experimentation is needed.

## V. Summary and Concluding Remarks

In this paper we have presented a new diagonally structured storage scheme, the CSD, for general sparse matrices. We have presented a matrix-matrix multiplication algorithm where the argument matrices and the result are stored using CSD scheme. The algorithm is given in relation to the standard coordinate indexing scheme and it can be used to convert between different storage schemes for dense as well as sparse matrices.

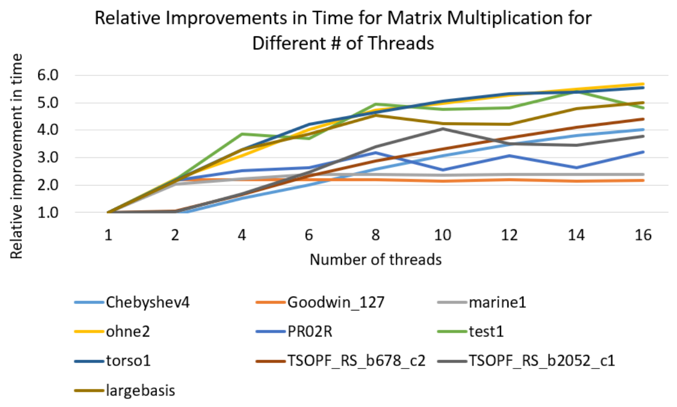

Figure 1: Relative performance in computational time by OpenMP

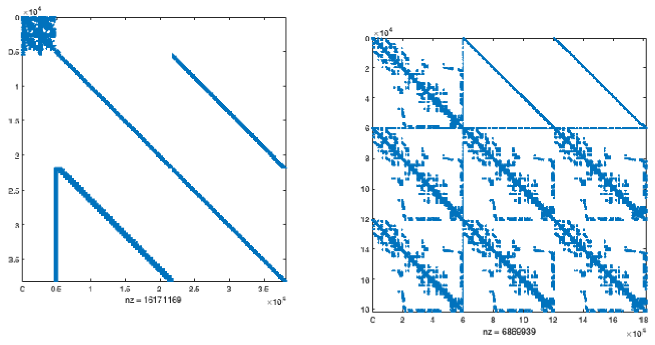

Figure 2: Sparsity pattern of TSOPF_RS_b2383 (left) and ohne2 (right)

Numerical experiments with a small yet representative benchmark test instances demonstrate that CSD is competitive with CSR with respect to matrix multiplication.

The current work is an overview of early experience with the alternative storage scheme based on matrix diagonals (as opposed to rows/columns) to provide linear algebraic kernel operations in BLAS specification. More specifically, this new storage scheme avoids explicit matrix transposition and thereby has the potential to improve the performance of BLAS level-2 and level-3 operations. We are currently developing a blocked version of sparse and structured matrix-matrix multiplication for serial and hybrid massively threaded (e.g., GPU) parallel environments.

## ACKNOWLEDGMENT

This research was supported in part by Natural Sciences and Engineering Research Council of Canada Discovery Grant.

## REFERENCES

[1] S. Hossain and M. S. Mahmud, "On computing with diagonally structured matrices," in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–6, IEEE, 2019.

[2] W. M. Abdullah, D. Awosoga, and S. Hossain, "Efficient calculation of triangle centrality in big data networks," in *2022 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–7, 2022.

[3] J. Gao, W. Ji, F. Chang, S. Han, B. Wei, Z. Liu, and Y. Wang, "A systematic survey of general sparse matrix-matrix multiplication," *ACM Computing Surveys*, vol. 55, no. 12, pp. 1–36, 2023.

[4] J. Kepner and J. Gilbert, *Graph algorithms in the language of linear algebra*. SIAM, 2011.

[5] J. Kepner and H. Jananthan, *Mathematics of big data: Spreadsheets, databases, matrices, and graphs*. MIT Press, 2018.

[6] S. Hong, N. C. Rodia, and K. Olukotun, "On fast parallel detection of strongly connected components (scc) in small-world graphs," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pp. 1–11, 2013.

[7] W. Mclendon Iii, B. Hendrickson, S. J. Plimpton, and L. Rauchwerger, "Finding strongly connected components in distributed graphs," *Journal of Parallel and Distributed Computing*, vol. 65, no. 8, pp. 901–910, 2005.

[8] H. Wang, W. Liu, K. Hou, and W.-c. Feng, "Parallel transposition of sparse data structures," in *Proceedings of the 2016 international conference on supercomputing*, pp. 1–13, 2016.

[9] W. Yang, K. Li, Y. Liu, L. Shi, and L. Wan, "Optimization of quasi-diagonal matrix–vector multiplication on gpu," *The international journal of high performance computing applications*, vol. 28, no. 2, pp. 183–195, 2014.

[10] D. Langr and P. Tvrdik, "Evaluation criteria for sparse matrix storage formats," *IEEE Transactions on parallel and distributed systems*, vol. 27, no. 2, pp. 428–440, 2015.

[11] Y. Saad and K. SPARS, "A basic tool kit for sparse matrix computations," *RIACS, NA SA Ames Research Center, TR90-20, Moffet Field, CA*, 1990.

[12] S. A. Haque, M. T. Parvez, and S. Hossain, "Gpu algorithms for structured sparse matrix multiplication with diagonal storage schemes," *Algorithms*, vol. 17, no. 1, p. 31, 2024.

[13] J. Eagan, M. Herdman, C. Vaughn, N. Bean, S. Kern, and M. Pirouz, "An efficient parallel divide-and-conquer algorithm for generalized matrix multiplication," in *2023 IEEE 13th Annual Computing and Communication Workshop and Conference (CCWC)*, pp. 0442–0449, IEEE, 2023.

[14] F. Dellaert and M. Kaess, "Square root sam: Simultaneous localization and mapping via square root information smoothing," *The International Journal of Robotics Research*, vol. 25, no. 12, pp. 1181–1203, 2006.