Improving Performance of Sparse Matrix-Vector Multiplication[*]

Ali Pınar Michael T. Heath

Department of Computer Science and
Center of Simulation of Advanced Rockets

University of Illinois at Urbana-Champaign

Abstract

Sparse matrix-vector multiplication (SpMxV) is one of the most important computational kernels in scientific computing. It often suffers from poor cache utilization and extra load operations because of memory indirections used to exploit sparsity.

We propose alternative data structures, along with reordering algorithms to increase effectiveness of these data structures, to reduce the number of memory indirections. Toledo proposed handling the 1x2 blocks of a matrix separately, doing only one indirection for each block. We propose packing all contiguous nonzeros into a block to reduce the number of memory indirections further. This reduces memory indirections per block to one for the cost of an extra array in storage and a loop during SpMxV.

We also propose an algorithm to permute the nonzeros of the matrix into contiguous locations. We state this problem as the traveling salesperson problem and use associated heuristics. Experiments verify the effectiveness of our techniques.

1 Introduction

One of the most important computational kernels in scientific computing is multiplying a sparse matrix by a vector. Many algorithms use sparse matrix-vector multiplication (SpMxV) in their inner loop (iterative solvers for systems of linear equations being just one example). The repeated execution of this operation potentially amortizes the cost of a preprocessing phase, which might lead to computational savings in subsequent executions. The importance of the SpMxV operation has made this challenging problem the subject of numerous research efforts.

Data structures used for sparse matrices usually have two components: (*i*) an array that stores all the floating-point entries of the matrix, (*ii*) arrays that store the sparsity structure of the matrix, i.e., pointers to the locations of the floating-point entries in the matrix. To exploit the sparsity of the matrix the use of pointers is unavoidable but often limits the memory system performance. One reason for this is that pointers usually lead to poor cache utilization, since they lack spatial locality. The number of cache misses for the right- and/or left-hand-side vectors can dramatically increase if the sparse matrix has an irregular sparsity structure. Another important factor is that memory indirections (pointers) require extra load operations. In sparse matrix operations, the number of floating-point operations per load operation is lower than that of dense matrix operations, limiting overall performance.

[*] Research supported by the Center for Simulation of Advanced Rockets, funded by the U.S. Department of Energy through the University of California under subcontract number B341494.

We propose alternative data structures, as well as reordering algorithms to increase the effectiveness of those data structures, to reduce the number of memory indirections in SpMxV. Toledo [7] proposed identifying 1×2 blocks of a matrix and writing the matrix as the sum of two matrices, the first of which contains all the 1×2 blocks and the second contains the remaining entries. Thus, the number of memory indirections is reduced to only one for each 1×2 block. In an alternative scheme, we pack all the nonzeros in contiguous locations into a block to reduce further the number of memory indirections, because only one memory indirections is required for all the nonzeros in a block. However, we then need to know how many nonzeros are in each block, which requires an extra array and an extra loop during SpMxV. This work concentrates on the latter problem.

We also propose reordering the matrices to increase the effectiveness of our data structures. The objective of reordering is to permute the nonzeros of the matrix into contiguous locations, as much as possible, to enlarge the dense blocks. We can show that this problem is NP-Complete, and thus heuristics must be used for a practical solution. We propose a graph model to reduce the problem to the well-studied traveling salesperson problem, and thus we can use heuristics designed for that problem.

We verify experimentally the effectiveness of the proposed data structures and algorithms. Our new data structures and reordering techniques produce improvements of up to 33% and improvements of 21% on average.

The remainder of this paper is organized as follows. Section 2 discusses the shortcomings of current sparse matrix data structures and proposes new alternatives. Section 3 states a new matrix reordering algorithm to permute the nonzeros of the matrix into contiguous locations. Experimental results are presented in Section 4, and finally we conclude with Section 5.

2 Sparse Matrix Data Structures

The most widely used sparse matrix storage scheme is *Compressed Row Storage* (CRS). As its name implies, this scheme stores the sparse matrix as a sequence of compressed rows. Three arrays are employed: A_f, $Colind$ and $Rowptr$. The nonzeros of the sparse matrix A are compressed into an array A_f in a rowwise manner. The column index of each nonzero entry is stored in the array $Colind$, i.e., $Colind[i]$ is the column index of the nonzero entry stored in $A_f[i]$. Finally, $Rowptr$ stores the index of the first nonzero of each row. Figure 1 presents a SpMxV algorithm using compressed row storage.

```
for $i \leftarrow 1$ to $m$ do
    $y[i] \leftarrow 0$;
    for $j \leftarrow Rowptr[i]$ to $Rowptr[i+1]-1$ do
        $y[i] \leftarrow y[i] + A_f[j] * x[Colind[j]]$;
```

Figure 1: SpMxV algorithm in compressed row storage

Sparse matrix-vector multiplication (SpMxV) algorithms tend to suffer from poor memory performance. One reason for this is ineffective cache utilization. Temporal locality is limited to right- and left-hand-side vectors and $Rowptr$ array. No temporal locality is present in arrays A_f and $Colind$. In CRS, there is good temporal locality in the left-hand-side vector y, but the access pattern for the right-hand-side vector can be quite irregular, causing excessive cache misses. Reordering the matrix to reduce cache misses was proposed by Das. et al. [3], who suggested reducing the bandwidth of the matrix. Temam and Jalby [6] analyzed the number of cache misses as a function of the bandwidth for various cache parameters. Burgess and Giles [2] experimented with various ordering algorithms and found that reordering the matrix improves performance compared with random ordering, but they did not detect a notable sensitivity to the particular ordering method used. Toledo [7] studied reordering, along with other techniques, and reported that Reverse Cuthill-McKee ordering yielded slightly better performance, but the differences were not significant.

Another problem with SpMxV is that the ratio of load operations is higher than with dense matrix operations. One extra load operation is required to find the column index of each nonzero entry for each multiply-add operation. The innermost statement in the algorithm in Figure 1 requires three load operations for two floating-point operations. This not only increases the total number of load instructions, but also can cause the load units to be a bottleneck, especially in recent architectures as discussed in [7].

The following two sections present data structures that can decrease the number of memory indirections during SpMxV operation.

2.1 Fixed-Size Blocking

In this approach, the matrix is written as sum of several matrices, some of which contain the dense blocks of the matrix with a prespecified size. For instance, given a matrix A, we can decompose it into two matrices A_{12} and A_{11}, such that $A = A_{12} + A_{11}$, where A_{12} contains the 1×2 dense blocks of the matrix and A_{11} contains the remainder. An example is illustrated in Figure 2. A simple greedy algorithm is sufficient to extract the maximum number of $1 \times l$ blocks in a matrix, where $1 < l \leq n$. However, the problem is more difficult for $k \times l$ blocks for $k > 1$.

$$\begin{pmatrix} x & x & x & & \\ & x & & x & \\ x & x & & & \\ & & & x & x \\ & & x & x & x \end{pmatrix} = \begin{pmatrix} x & x & & & \\ & & & & \\ x & x & & & \\ & & & x & x \\ & & x & x & \end{pmatrix} + \begin{pmatrix} & & x & & \\ & x & & x & \\ & & & & \\ & & & & \\ & & & & x \end{pmatrix}$$

$$A \qquad = \qquad A_{12} \qquad + \qquad A_{11}$$

Figure 2: Fixed size blocking with 1×2 blocks

Exploiting dense blocks can reduce the number of load operations, as well as the total memory requirement, because only one index per block is required. Moreover, the entry of the right-hand-side vector x can be used multiple times—as opposed to once in the conventional scheme—after a load operation.

A similar problem has been studied in the context of vector processors [1], but those efforts concentrate on finding fewer blocks of larger size, whereas we are interested in much smaller blocks.

2.2 Blocked Compressed Row Storage

In this section we propose a new sparse matrix storage scheme designed to reduce the number of load operations. The idea of this scheme is to exploit the nonzeros in contiguous locations by packing them. Unlike fixed-size blocking, the blocks will have variable lengths. This enables longer nonzero strings to be packed into a block. As in fixed-size blocking, if we know the column index of the first nonzero in a block, then we will also know the column indices of all its other nonzeros. In other words, only one memory indirection (extra load operation) is required for each block.

This storage scheme requires an array $Nzptr$ (of length the number of blocks) in addition to the other three arrays used in CRS: a floating-point array A_f (of length the number of nonzeros) to store the nonzero values, an array $Colind$ (of length the number of blocks) to store the column number of the first nonzero for each block, and an array $Rowptr$ (of length the number of rows) to point to the position where the blocks of each row start. $Nzptr$ stores the location of the first nonzero of each block in array A_f. We refer to this storage scheme as *blocked compressed row storage* (BCRS). Figure 3 presents an example of BCRS, and the SpMxV operation using this storage is presented in Figure 4.

$$A = \begin{pmatrix} 5. & 1. & 7. & 0 & 0 \\ 0 & 1. & 0 & 2. & 3. \\ 0 & 2. & 4. & 0 & 0 \\ 0 & 0 & 1. & 3. & 0 \\ 0 & 6. & 0 & 0 & 3. \end{pmatrix} \qquad \begin{array}{rcl} A_f & = & (5., 1., 7., 1., 2., 3., 2., 4., 1., 3., 6., 3.) \\ Colind & = & (1, 2, 4, 2, 3, 2, 5) \\ Rowptr & = & (1, 2, 4, 5, 6, 8) \\ Nzptr & = & (1, 4, 5, 7, 9, 11, 12, 13) \end{array}$$

Figure 3: Example of blocked compressed row storage

This storage scheme reduces extra load operations but requires an extra loop during the SpMxV operation and thus suffers additional loop overhead. If the sizes of the blocks are small, the overhead due to the extra loop will dominate the gain due to decreased load operations. Thus, the effectiveness of this storage scheme depends directly on the sizes of the blocks in the matrix.

The total volume of storage has a similar tradeoff. Two numbers (one in $Colind$ and one in $Nzptr$) are stored for each block in the matrix, as opposed to one number for each nonzero. Thus, if the blocks are large enough, the total storage size can be significantly reduced.

3 Reordering to Enlarge Dense Blocks

The previous section described our data structures to exploit dense blocks of a matrix. The effectiveness of these data structures depend directly on the availability of dense blocks. In this section we describe

```
**for** $i \leftarrow 1$ **to** $m$ **do**
    $y[i] \leftarrow 0;$
    **for** $j \leftarrow Rowptr[i]$ **to** $Rowptr[i+1]-1$ **do**
        $startcol \leftarrow Colind[j];$
        $t \leftarrow 0;$
        **for** $k \leftarrow Nzptr[j]$ **to** $Nzptr[j+1]-1$ **do**
            $y[i] \leftarrow y[i] + A_f[k] * x[startcol + t];$
            $t \leftarrow t + 1;$
```

Figure 4: SpMxV with blocked compressed row storage

reordering algorithms to enlarge dense blocks of a matrix. First, we will formally define the problem and propose a graph model to reduce the problem to the traveling salesperson problem (TSP). Then we will discuss briefly heuristics we used for solving TSP.

3.1 Problem Formulation

Our objective in reordering the matrix is to increase the sizes of the dense blocks in a row, i.e., to align the nonzeros in a row in consecutive locations as much as possible. This requires reordering of columns and is not affected by ordering of rows. Thus, rows of the matrix can be reordered without disturbing the aligned nonzeros within a row (e.g., the same reordering can be applied to rows to preserve symmetry). These techniques can also be used to align nonzeros within a column by interchanging the roles of columns and rows. A formal description of the problem follows.

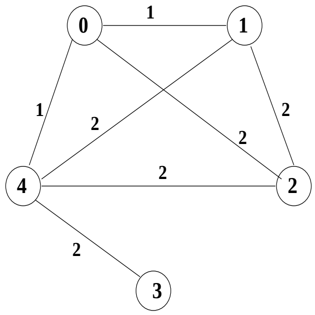
Given an $m \times n$ matrix $A = (a_{ij})$, find an ordering of columns to maximize the number of (i, j) pairs satisfying $a_{ij} \neq 0$ and $a_{ij+1} \neq 0$, i.e.,

$$Maximize \quad \#((i,j) : 1 \leq i \leq m, 1 \leq j < n : a_{ij} \neq 0 \;\; and \;\;\; a_{ij+1} \neq 0).$$

In the Appendix, we show that the problem is NP-Complete by using reduction from Hamiltonian Path problem. Thus, we must resort to heuristics for a practical solution. Such an ordering problem is naturally close to TSP. However, we are looking for a path, not a tour (cycle) of vertices. Although this is a slightly different version of the problem, heuristics designed to find a tour can still be used or adapted to find a path of vertices. Thus, our strategy will be first to define a graph model that reduces our matrix reordering problem to the well-studied TSP and use heuristics already designed for TSP to reorder our matrices. For the sake of presentation, we will refer to the maximization version of TSP, i.e.,

Given a graph $G = (V, E)$ and a weighting function w on its edges, find a tour $\langle v_0, v_1, \ldots v_{|V|-1} \rangle$ to maximize

$$\begin{array}{c}\begin{pmatrix} 0 & 1 & 2 & 3 & 4 \end{pmatrix} \\ \begin{pmatrix} x & x & x & & x \\ & x & x & & x \\ x & & x & & \\ & & & x & x \\ & & & x & x \end{pmatrix}\end{array}$$



$$\begin{array}{c}\begin{pmatrix} 0 & 2 & 1 & 4 & 3 \end{pmatrix} \\ \begin{pmatrix} x & x & x & x & \\ & x & x & x & \\ x & x & & & \\ & & & x & x \\ & & & x & x \end{pmatrix}\end{array}$$

Figure 5: Edge weight computation. Graph shows edge weights of matrix on left-hand side. Matrix after reordering is presented on right.

$$\sum_{0 \leq i < |V|} w(v_i, v_{i+1}).$$

Notice that the weight of the edge $(v_{|V|}, v_0)$ is not included, since we need a path, not a tour.

Since we are trying to reorder columns of the matrix, the graph that reduces the matrix reordering problem to TSP has vertices representing the columns of the matrix. The weight of an edge between two vertices is defined as the number of rows where both respective columns have a nonzero, i.e.,

$$w(u, v) = \#(i : 1 \leq i \leq m : a_{iu} \neq 0 \;\; and \quad a_{iv} \neq 0)$$

An alternative definition, which also gives the basics of an algorithm, is as follows. Let $S_A = (s_{ij})$ be a matrix of zeros and ones with the same nonzero structure as A, i.e.,

$$s_{ij} = 1 \quad \Leftrightarrow \quad a_{ij} \neq 0.$$

Let $W = S_A S_A^T$. The weight function can be defined by the matrix $W = (w_{ij})$ as $w(u, v) = w_{uv}$. Figure 5 illustrates an example of edge weight computation. In this example the weight of the $(0, 1)$ edge is equal to 1 because columns 0 and 1 share a nonzero only in the first row, whereas the weight of the $(1, 4)$ edge is equal to 2 because columns 1 and 4 share nonzeros in the first and second rows.

Notice that although a weight (possibly zero) is assigned to any pair of vertices in the graph (any ordering is a feasible solution for the matrix reordering problem), W is expected to be sparse, which must be exploited for the sake of efficiency (or even existence) of a practical solution.

If the vertices u and v are in consecutive locations in the TSP solution, then the respective columns will be in consecutive locations in the ordered matrix, and $w(u, v)$ nonzeros will be in contiguous locations. The larger the weights of the edges in the tour, the greater the number of nonzeros in contiguous locations will be. In general, finding a tour with maximum weight in this graph corresponds to finding an ordering with maximum number of contiguous nonzeros. To be more precise, the number of blocks (as in BCRS scheme) in the reordered matrix is equal to the number of nonzeros in the matrix minus the total weight of the TSP tour, since each unit of weight corresponds to locating a nonzero adjacent to another one, thus decreasing the number of blocks by one. Consequently, a tour with maximum weight describes an ordering with minimum number of blocks.

In the example of Figure 5, the matrix has 13 nonzeros. TSP solution used to reorder the matrix is $0 \rightarrow 2 \rightarrow 1 \rightarrow 4 \rightarrow 3$, the total weight of which is $2+2+2+2=8$. The number of blocks in the reordered matrix is $13-8=5$.

3.2 Heuristics for TSP

As in many other combinatorial optimization problems, heuristics for TSP can be classified into two groups: constructive and improvement. Constructive heuristics directly construct a solution for the problem, whereas improvement heuristics start with a given solution and try to improve that solution by searching the neighborhood. A comprehensive discussion of TSP solution methods can be found in [4, 5]. In this work, we adopted heuristics from the literature instead of designing our own. Here we discuss briefly the heuristics used in our experiments. Since matrix reordering is proposed as a preprocessing step, we focused on computationally efficient heuristics.

The simplest constructive heuristic is to use the initial ordering of the matrix. Matrices often have some natural ordering that can be used as an initial solution. We also used a vertex insertion procedure. The process starts with an initial path of one random vertex, and vertices are inserted one by one to the path until all are included in the path. At each step, a vertex that is not in the current path is randomly chosen and inserted into the path, so that the sum of weights of edges to its successor and predecessor is maximum. The final constructive heuristic we used is to start with a random vertex as the initial path and proceed by inserting vertices at the end of the path. At each step, the vertex that is connected to the last vertex of the current path with the heaviest edge weight is inserted at the end of the path.

Improvement heuristics have the same flavor as constructive heuristics. One of the heuristics we used depends on vertex insertion. But this time a vertex is first removed from the current path, unlike the case in constructive heuristics, and then reinserted to maximize the total weight of the tour. We also used edge or path reinsertion procedures, which are similar to vertex insertion. In these heuristics, edges (paths of two vertices) or longer paths are reinserted instead of vertices (named Or-opt procedure in [4]).

4 Experimental Results

We implemented the new data structures and reordering algorithms in C and compared their performances with that of conventional compressed row storage (CRS) scheme. We experimented with a collection of matrices selected from the Harwell-Boeing sparse-matrix collection. All experiments were performed on a Sun Enterprise 3000.

In the first set of experiments, we investigated the effectiveness of our ordering techniques. Figure 6 presents our experimental results with 1×2 blocked matrices after Reverse Cuthill-McKee (RCM) ordering, using the initial ordering, and TSP ordering (described in Section 3). TSP solutions are generated by starting with the initial ordering and then using a vertex insertion procedure to improve the initial solution. In Figure 6, SpMxV times are normalized with respect to SpMxV times with the conventional CRS scheme. 1 × 2 blocking after TSP ordering always improves performance and is always superior to the other two ordering methods. The difference becomes very significant for `dw8192` and `bcsstk28`. On average, TSP ordering reduces the runtime of a SpMxV operation to 79% of that of the conventional scheme, whereas RCM ordering and using the initial ordering are limited to only, 95% and 92%, respectively.

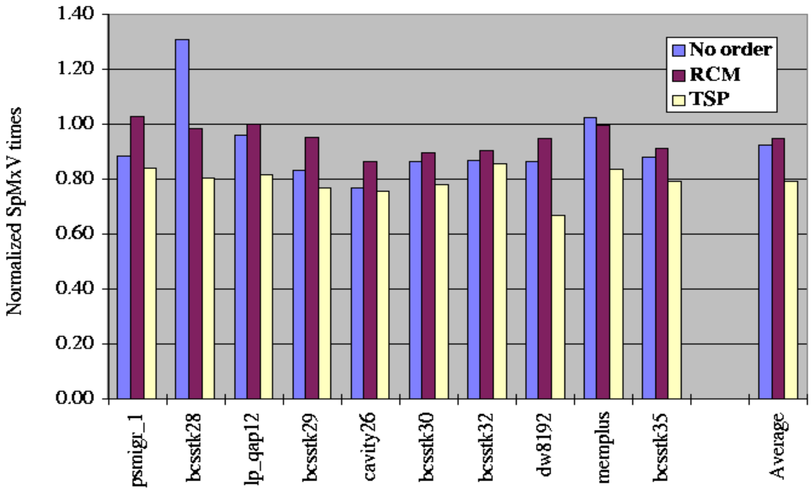


Figure 6: Effectiveness of ordering

Figure 7 presents our results with the two new data structures. Again, SpMxV times are normalized with respect to SpMxV times with the conventional CRS scheme. Fixed-size blocking is superior to BCRS for most matrices and on average. BCRS outperforms fixed-size blocking for four matrices: `bcsstk28, cavity26, bcsstk30`, and `bcsstk32`.

We also experimented with minor modifications of the data structures. For instance, using 1 × 3 blocks instead of 1 × 2 increases the performance by 3% on average, but further increasing the block size does not help. Using blocked compressed row storage only for blocks of size greater than 1 and using the conventional scheme for the remainder increased the performance by 8%, compared with blocked compressed row storage. We are in the process of tuning our data structures for maximum performance.

The results show that significant savings can be achieved by exploiting the dense blocks in a matrix. Reordering the matrices yields notable improvement in performance, offsetting the cost of the preprocessing phase, which is often amortized over repeated SpMxV operations with the same matrix.

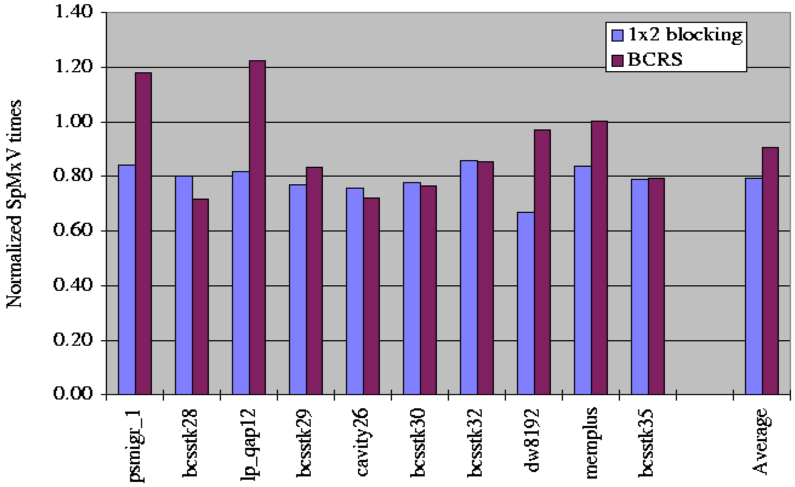


Figure 7: Comparison of data structures

5 Conclusions and Future Work

We have presented new data structures, along with a matrix reordering algorithm to increase the effectiveness of those data structures, designed to decrease the number of memory indirections during sparse matrix-vector multiplication. The data structures exploit dense blocks—nonzeros in contiguous locations—of the matrix. We also proposed a reordering algorithm to enlarge the dense blocks of a matrix. Experiments verify the effectiveness of proposed techniques, with observed performance improvement of up to 33% and an average improvement of 21%.

Currently we are working on improving the performance of the proposed techniques. We are also investigating alternative ways of exploiting the density in the sparse matrices by designing additional new data structures. Finally, we plan to experiment with various architectures to observe the performance of these techniques with a variety of architectural parameters.

References

[1] R. C. Agarwal, F. G. Gustavson, and M. Zubair, “A high performance algorithm using pre-processing for sparse matrix vector multiplication”, *Proceedings of Supercomputing ’92*, pp. 32–41.

[2] D. A. Burgess and M. B. Giles, “Renumbering unstructured grids to improve the performance of codes on hierarchical memory machines”, Tech. Rep. 95/06, Numerical Analysis Group, Oxford University Computing laboratory, May 1995.

[3] R. Das, D. J. Mavriplis, J. Saltz, S. Gupta, and R. Ponnusamy, “The design and implementation of a parallel unstructured Euler solver using software primitives”, *AIAA Journal*, No: 32, pp. 489–496, 1994.

[4] J. Perttunen, “The traveling salesman problem: the significance of initial solutions in composite heuristics”, Vaasan Korkeakoulun julkaisuja, 0358-9080; no. 139, 1989.

[5] G. Reinelt *The traveling salesman: computational solutions for TSP applications*, Lecture Notes in Computer Science, Vol: 840, Springer-Verlag, 1994.

[6] O. Temam and W. Jalby, "Characterizing the behavior of sparse algorithms on caches", *Proceedings of Supercomputing'92*, 1992.

[7] S. Toledo, "Improving Memory-System Performance of Sparse Matrix-Vector Multiplication", *Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Computing*, March 1997.

Appendix NP-Completeness of Problem

In this section, we prove that ordering a sparse matrix to increase sizes of dense blocks is NP-Complete by using reduction from the Hamiltonian Path problem. First we state a decision version of the problem:

Given an $m \times n$ matrix $A = (a_{ij})$, decide if there exists an ordering of columns, where the number of (i, j) pairs satisfying $a_{ij} \neq 0$ and $a_{i,j+1} \neq 0$ is greater than or equal to a given bound B.

Given a simple graph $G = (V, E)$ (no loops, no parallel edges), construct an $|E| \times |V|$ matrix A. In this matrix each row represents an edge and each column represents a vertex in the graph. Let the ith column represent the ith vertex. The nonzero structure matrix A is defined such that there are nonzeros at a_{ji} and a_{jk} for each edge $e_j = (v_i, v_k)$.

To increase the size of dense blocks, we have to bring the columns of adjacent vertices to consecutive locations in the reordered matrix. Notice that two adjacent columns can share nonzeros in at most one row, because there are no parallel edges. There can be at most $|V| - 1$ blocks of size 1×2 after reordering, achieved when the vertices of consequent columns share an edge in the graph, which defines a Hamiltonian path in the graph.

It is also easy to see the problem is in NP, since a given solution can be verified in polynomial time. So we can conclude that the problem is NP-Complete. □