Technical Report

BU-CE-1017

Hypergraph-Partitioning-Based Models and Methods for Exploiting Cache Locality in Sparse-Matrix Vector Multiplication

Kadir Akbudak, Enver Kayaaslan and Cevdet Aykanat

November 5, 2010

Bilkent University
Computer Engineering Department
Faculty of Engineering
06800 Ankara, Turkey
http://cs.bilkent.edu.tr
HYPERGRAPH-PARTITIONING-BASED MODELS AND METHODS FOR EXPLOITING CACHE LOCALITY IN SPARSE-MATRIX VECTOR MULTIPLICATION

KADIR AKBUDAK*, ENVER KAYAASLAN†, AND CEVDET AYKANAT‡

Abstract. The sparse matrix-vector multiplication (SpMxV) is a kernel operation widely used in iterative linear solvers. The same sparse matrix is multiplied by a dense vector repeatedly in these solvers. Matrices with irregular sparsity patterns make it difficult to utilize cache locality effectively in SpMxV computations. In this work, we investigate single- and multiple-SpMxV frameworks for exploiting cache locality in SpMxV computations. For the single-SpMxV framework, we propose two cache-size-aware top-down row/column-reordering methods based on 1D and 2D sparse matrix partitioning by utilizing the column-net and enhancing the row-column-net hypergraph models of sparse matrices. The multiple-SpMxV framework depends on splitting a given matrix into a sum of multiple nonzero-disjoint matrices so that the SpMxV operation is computed as a sequence of multiple input- and output-dependent SpMxV operations. For an effective matrix splitting required in this framework, we propose a cache-size-aware top-down approach based on 2D sparse matrix partitioning by utilizing the row-column-net hypergraph model. For this framework, we also propose a TSP formulation for an effective ordering of individual SpMxV operations. The primary objective in all of the three methods is to maximize the exploitation of temporal locality. We evaluate the validity of our models and methods on a wide range of sparse matrices. Experimental results show that proposed methods and models outperform state-of-the-art schemes.

Key words. cache locality, sparse matrix, matrix-vector multiplication, matrix reordering, computational hypergraph model, hypergraph partitioning, traveling salesman problem

AMS subject classifications. 65F10, 65F50, 65Y20

1. Introduction. Sparse matrix-vector multiplication (SpMxV) is an important kernel operation in iterative linear solvers used for the solution of large, sparse, linear systems of equations. In these iterative solvers, the SpMxV operation $y \leftarrow Ax$ is repeatedly performed with the same large, irregularly sparse matrix $A$. Irregular access pattern during these repeated SpMxV operations causes poor usage of cpu caches in today’s deep memory hierarchy technology. However, SpMxV operation has a potential to exhibit very high performance gains if temporal and spatial localities are respected and exploited properly.

In this work, we investigate two distinct frameworks for cache-oblivious SpMxV: single-SpMxV and multiple-SpMxV frameworks. In the single-SpMxV framework, the $y$-vector results are computed by performing a single SpMxV operation $y \leftarrow Ax$, whereas in the multiple-SpMxV framework, $y \leftarrow Ax$ operation is computed as a sequence of multiple input- and output-dependent SpMxV operations. For the single-SpMxV framework, we propose two cache-size-aware row/column reordering methods based on top-down 1D and 2D partitioning of a given sparse matrix. The 1D-partitioning-based method relies on transforming a sparse matrix into a singly-bordered block-diagonal form by utilizing the column-net hypergraph model [7, 8, 9]. The 2D-partitioning-based method relies on transforming a sparse matrix into a doubly-bordered block-diagonal form by utilizing the row-column-net hypergraph model [7, 11]. We provide upper bounds on the total number of cache misses based on these transformations, and show that the objectives in the transformations based on partitioning the respective hypergraph models correspond to minimizing these upper bounds. In the 1D-partitioning-based method, the column-net hypergraph model correctly encapsulates the minimization of the respective upper bound. For the 2D-partitioning-based method, we propose an enhancement to the row-column-net hypergraph model to encapsulate the mini-

*Computer Engineering Department, Bilkent University, Ankara, Turkey (kadir@cs.bilkent.edu.tr).
†Computer Engineering Department, Bilkent University, Ankara, Turkey (enver@cs.bilkent.edu.tr).
‡Computer Engineering Department, Bilkent University, Ankara, Turkey (aykanat@cs.bilkent.edu.tr).
mization of the respective upper bound on cache misses. The primary objective in both methods is to maximize the exploitation of the temporal locality due to the access of \textit{x-vector} entries, whereas exploitation of the spatial locality due to the access of \textit{x-vector} entries is a secondary objective. In this paper, we claim that exploiting temporal locality is more important than exploiting spatial locality (for practical line sizes) in SpMxV operations that involve irregularly sparse matrices.

The multiple-SpMxV framework depends on splitting a given matrix into a sum of multiple nonzero-disjoint matrices so that the SpMxV operation is computed as a sequence of multiple dependent SpMxV operations. For an effective matrix splitting required in this framework, we propose a cache-size-aware top-down approach based on 2D sparse matrix partitioning by utilizing the row-column-net hypergraph model \cite{7, 11}. We provide an upper bound on the total number of cache misses based on this matrix-splitting, and show that the objective in the hypergraph-partitioning (HP) based matrix partitioning exactly corresponds to minimizing this upper bound. The primary objective in this method is to maximize the exploitation of the temporal locality due to the access of both \textit{x-vector} and \textit{y-vector} entries. For this framework, we also propose a traveling salesman problem (TSP) formulation for an effective ordering of individual SpMxV operations. We provide a lower bound on the total number of cache misses based on the ordering of individual SpMxV operations, and show that the objective in the proposed TSP formulation exactly corresponds to minimizing this lower bound.

We evaluate the validity of our models and methods on a wide range of sparse matrices. Experimental results show that proposed methods and models outperform state-of-the-art schemes and also these results conform to our expectation that temporal locality is more important than spatial locality in SpMxV operations that involve irregularly sparse matrices.

The rest of the paper is organized as follows: Background material is introduced in Section 2. In Section 3, we review some of the previous works about iteration/data reordering and matrix transformations for exploiting locality. The two frameworks along with our contributed models and methods are described in Sections 4 and 5. We present the experimental results in Section 6. Finally, the paper is concluded in Section 7.

2. Background. Several sparse-matrix storage schemes utilized in SpMxV are summarized in Section 2.1. Data locality issues during SpMxV operations are discussed in Section 2.2. Section 2.3 summarizes the HP problem, whereas Section 2.4 discusses hypergraph models and methods for sparse-matrix partitioning. Finally, bipartite graph model for sparse matrices is given in Section 2.5.

2.1. Sparse-matrix storage schemes. There are two standard sparse-matrix storage schemes for SpMxV operation: Compressed Storage by Rows (CSR) and Compressed Storage by Columns (CSC) \cite{14, 33}. In this paper, we restrict our focus on cache-oblivious SpMxV operation using the CSR storage scheme without loss of generality. In the following paragraphs, we review the standard CSR scheme and two CSR variants.

The compressed Storage by Rows (CSR) scheme contains three 1D arrays: \textit{nonzero}, \textit{colIndex} and \textit{rowStart}. The values and the column indices of nonzeros are respectively stored in row-major order in the \textit{nonzero} and \textit{colIndex} arrays in a one-to-one manner. That is, \textit{colIndex}[k] stores the column index of the nonzero stored in \textit{nonzero}[k]. The \textit{rowStart} array stores the index of the first nonzero of each row in the \textit{nonzero} and \textit{colIndex} arrays.

Algorithm 1 shows SpMxV utilizing the CSR storage scheme for an \(m \times n\) sparse matrix. Each outer for-loop iteration of Algorithm 1 corresponds to the inner product of the respective sparse row with the dense input vector \(x\).

The Zig-zag CSR (ZZCSR) scheme is recently proposed to reduce end-of-row cache misses \cite{42}. In this scheme, nonzeros are stored in increasing column index order in even-
Algorithm 1 SpMxV using CSR scheme

Require: nonzero, colIndex and rowStart arrays of an \( m \times n \) sparse matrix  

\( A \)

dense input vector  

\( x \)

Output: dense vector  

\( y \)

\begin{algorithm}
1: for \( i \leftarrow 1 \to m \) do
2: \( \text{sum} \leftarrow 0.0 \)
3: for \( k \leftarrow \text{rowStart}[i] \to \text{rowStart}[i+1] - 1 \) do
4: \( \text{sum} \leftarrow \text{sum} + \text{nonzero}[k] \times \text{colIndex}[k] \)
5: end for
6: \( y[i] \leftarrow \text{sum} \)
7: end for
8: return  
\end{algorithm}

numbered rows, whereas they are stored in decreasing index order in odd-numbered rows, or vice versa.

The Incremental Compressed Storage by Rows (ICSR) scheme [27] which is given in Algorithm 2 is reported to decrease instruction overhead by using pointer arithmetic. In ICSR, the colIndex array is replaced with the colDiff array, which stores the increments in the column indices of the successive nonzeros stored in the nonzero array. The rowStart array is replaced with the rowJump array which stores the increments in the row indices of the successive nonzero rows. The beginning of a new row is signalled by causing an increment value  \( j \) to overflow  \( n \) so that \( j - n \) shows the column index of the first nonzero in the next row. For this purpose, nonzeros of each row are stored in increasing column index order. The ICSR scheme has the advantage of handling zero rows efficiently since it avoids the use of the rowStart array. Consequently, this feature of ICSR is exploited in our multiple-SpMxV framework since this scheme introduces many zero rows in the individual sparse matrices.

Algorithm 2 SpMxV using ICSR scheme [27]

Require: nonzero, colDiff and rowJump arrays of an \( m \times n \) sparse matrix  

\( A \) with \( nnz \) nonzeros,  

dense input vector  

\( x \)

Output: dense vector  

\( y \)

\begin{algorithm}
1: \( i \leftarrow \text{rowJump} \)
2: \( j \leftarrow \text{colDiff}[0] \)
3: \( k \leftarrow 0 \)
4: \( r \leftarrow 1 \)
5: \( \text{sum} \leftarrow 0.0 \)
6: for \( k \leftarrow 1 \to nnz \) do
7: \( \text{sum} \leftarrow \text{sum} + \text{nonzero}[k] \times [j] \)
8: \( k \leftarrow k + 1 \)
9: \( j \leftarrow j + \text{colDiff}[k] \)
10: if \( j \geq n \) then
11: \( y[i] \leftarrow \text{sum} \)
12: \( \text{sum} \leftarrow 0.0 \)
13: \( j \leftarrow j - n \)
14: \( i \leftarrow i + \text{rowJump}[r] \)
15: \( r \leftarrow r + 1 \)
16: end if
17: end for
18: return  
\end{algorithm}

2.2. Data locality in SpMxV. Here, we will briefly mention about the data locality characteristics of the SpMxV operation  \( y \leftarrow Ax \) using the CSR scheme as also discussed
in [41]. In terms of the \( A \)-matrix stored in CSR format, temporal locality is not feasible since the elements of each of the nonzero, colIndex (colDiff in ICSR) and rowStart (rowJump in ICSR) arrays are accessed only once. Spatial locality is feasible and it is achieved automatically by nature of the CSR scheme since the elements of each of the three arrays are stored and accessed consecutively.

In terms of output vector \( y \), temporal locality is not feasible since each \( y \)-vector result is written only once to the memory. As a different view, temporal locality can be considered as feasible but automatically achieved especially at the register level because of the summation of scalar nonzero and \( x \)-vector entry product results to the temporary variable \( \text{sum} \). Spatial locality is feasible and it is achieved automatically since the \( y \)-vector entry results are stored consecutively.

In terms of input vector \( x \), both temporal and spatial locality are feasible. Temporal locality is feasible since each \( x \)-vector entry may be accessed multiple times. However, exploiting the temporal and spatial locality for the \( x \)-vector is the major concern in the CSR scheme since \( x \)-vector entries are accessed through a colIndex array (colDiff in ICSR) in a non-contiguous and irregular manner.

These locality issues can be solved by reordering rows/columns of matrix \( A \) and the exploitation level of these data localities depends both on the existing sparsity pattern of matrix \( A \) and the effectiveness of reordering heuristics.

### 2.3. Hypergraph partitioning.

A hypergraph \( \mathcal{H} = (\mathcal{V}, \mathcal{N}) \) is defined as a set \( \mathcal{V} \) of vertices and a set \( \mathcal{N} \) of nets (hyperedges). Every net \( n_j \in \mathcal{N} \) connects a subset of vertices, i.e., \( n_j \subseteq \mathcal{V} \). Weights and costs can be associated with vertices and nets, respectively. We use \( w(v_i) \) to denote the weight of vertex \( v_i \) and \( \text{cost}(n_j) \) to denote the cost of net \( n_j \).

Given a hypergraph \( \mathcal{H} = (\mathcal{V}, \mathcal{N}) \), \( \Pi = \{V_1, \ldots, V_K\} \) is called a \( K \)-way partition of the vertex set \( \mathcal{V} \) if parts of \( \Pi \) are mutually disjoint and exhaustive. A \( K \)-way vertex partition of \( \mathcal{H} \) is said to satisfy the partitioning constraint if

\[
W_k \leq W_{\text{avg}}(1 + \varepsilon), \quad \text{for } k = 1, 2, \ldots, K
\]

Here, the weight \( W_k \) of a part \( V_k \) is defined as the sum of weights of vertices in that part (i.e., \( W_k = \sum_{v_i \in V_k} w(v_i) \)), \( W_{\text{avg}} \) is the average part weight (i.e., \( W_{\text{avg}} = (\sum_{v_i \in \mathcal{V}} w(v_i))/K \)), and \( \varepsilon \) represents a predetermined, maximum allowable imbalance ratio.

In a partition \( \Pi \) of \( \mathcal{H} \), a net that connects at least one vertex in a part is said to connect that part. Connectivity set \( \Lambda(n_j) \) of a net \( n_j \) is defined as the set of parts connected by \( n_j \). Connectivity \( \lambda(n_j) = |\Lambda(n_j)| \) of a net \( n_j \) denotes the number of parts connected by \( n_j \). A net \( n_j \) is said to be cut if it connects more than one part (i.e., \( \lambda(n_j) > 1 \)), and uncut otherwise (i.e., \( \lambda(n_j) = 1 \)). The set of cut nets of a partition \( \Pi \) is denoted as \( \mathcal{N}_{\text{cut}} \). The partitioning objective is to minimize the cutsize defined over the cut nets. There are various cutsize definitions. Two relevant definitions are the cut-net metric

\[
\text{cutsize}(\Pi) = \sum_{n_j \in \mathcal{N}_{\text{cut}}} \text{cost}(n_j)
\]

and the connectivity metric ([6]):

\[
\text{cutsize}(\Pi) = \sum_{n_j \in \mathcal{N}_{\text{cut}}} (\lambda(n_j) - 1) \text{cost}(n_j)
\]

In the cut-net metric, each cut net \( n_j \) incurs the cost of \( \text{cost}(n_j) \) to the cutsize, whereas in the connectivity metric, each cut net incurs the cost of \( (\lambda(n_j) - 1) \text{cost}(n_j) \) to the cutsize. The HP problem is known to be NP-hard [28]. There exists several successful HP tools such
as hMeTiS [26], PaToH [10] and Mondriaan [40], all of which apply the multilevel framework. The recursive bisection (RB) paradigm is widely used in K-way HP and known to be amenable to produce good solution qualities. In the RB paradigm, first, a two-way partition of the hypergraph is obtained. Then, each part of the bipartition is further bipartitioned in a recursive manner until the desired number K of parts is obtained or part weights drop below a given part-size threshold $W_{\text{max}}$. In RB-based HP, the cut-net removal and cut-net splitting schemes [9] are used to capture the cut-net and connectivity cutsize metrics, respectively. The RB paradigm is inherently suitable for partitioning hypergraphs when K is not known in advance. Hence, the RB paradigm can be successfully utilized in clustering rows/columns for cache-size-aware row/column reordering.

2.4. Hypergraph models for sparse matrix partitioning. Recently, several successful hypergraph models and methods are proposed for efficient parallelization of SpMxV operations [9, 7]. The relevant ones are row-net, column-net, and row-column-net models.

In the row-net hypergraph model [8, 9, 7] $\mathcal{H}_{RN}(A) = (\mathcal{V}_c, \mathcal{N}_R)$ of matrix $A$, there exist one vertex $v_j \in \mathcal{V}_c$ and one net $n_i \in \mathcal{N}_R$ for each column $c_j$ and row $r_i$, respectively. The weight $w(v_j)$ of a vertex $v_j$ is set to the number of nonzeros in column $c_j$. The net $n_i$ connects the vertices corresponding to the columns that have a nonzero entry in row $r_i$. Every net $n_i \in \mathcal{N}_R$ has unit cost, i.e., $\text{cost}(n_i) = 1$. In the column-net hypergraph model [8, 9, 7] $\mathcal{H}_{CN}(A) = (\mathcal{V}_r, \mathcal{N}_C)$ of matrix $A$, there exist one vertex $v_i \in \mathcal{V}_r$ and one net $n_j \in \mathcal{N}_C$ for each row $r_i$ and column $c_j$, respectively. The weight $w(v_i)$ of a vertex $v_i$ is set to the number of nonzeros in row $r_i$. Net $n_j$ connects the vertices corresponding to the rows that have a nonzero entry in column $c_j$. Every net $n_j$ has unit cost, i.e., $\text{cost}(n_j) = 1$.

In the row-column-net model [11] $\mathcal{H}_{RCN}(A) = (\mathcal{V}_Z, \mathcal{N}_RC)$ of matrix $A$, there exists one vertex $v_{ij} \in \mathcal{V}_Z$ corresponding to each nonzero $a_{ij}$ in matrix $A$. In net set $\mathcal{N}_RC$, there exists a row-net $n^r_{ij}$ for each row $r_i$, and there exists a column-net $n^c_{ij}$ for each column $c_j$. Every row net and column net have unit cost. Row-net $n^r_{ij}$ connects the vertices corresponding to the nonzeros in row $r_i$, and column-net $n^c_{ij}$ connects the vertices corresponding to the nonzeros in column $c_j$. Note that each vertex is connected by exactly two nets. $\mathcal{H}_{RCN}(A)$ is also called as the fine-grain model.

The use of these three hypergraph models in sparse-matrix partitioning for parallelization of SpMxV operations is described in detail in [7, 9]. The row-net and column-net models are used for 1D columnwise and 1D rowwise partitioning of sparse matrices, whereas row-column-net model is used for 2D nonzero-based (fine-grain) partitioning. It has been shown that the partitioning objective (2.3) corresponds to the total communication volume when the point-to-point interprocessor communication scheme is used, whereas the partitioning objective (2.2) corresponds to the total communication volume when the collective communication scheme is used. In these models, the partitioning constraint (2.1) corresponds to maintaining a computational load balance for a given number $K$ of processors.

In [3], it is shown that row-net and column-net models can also be used for transforming a sparse matrix into a K-way singly-bordered block-diagonal (SB) form through row and column reordering. In particular, the row-net model can be used for permuting a matrix into a rowwise SB form, whereas the column-net model can be used for permuting a matrix into a columnwise SB form. Here we will briefly describe how a K-way partition of the column-net model can be decoded as a row/column reordering for this purpose and a dual discussion holds for the row-net model.

A K-way vertex partition $\Pi = \{\mathcal{V}_1, \ldots, \mathcal{V}_K\}$ of $\mathcal{H}_{CN}(A)$ is considered as inducing a $(K+1)$-way partition $\{\mathcal{N}_1, \ldots, \mathcal{N}_K, \mathcal{N}_{\text{cut}}\}$ on the net set of $\mathcal{H}_{CN}(A)$. Here $\mathcal{N}_k$ denotes the set of uncut nets of vertex part $\mathcal{V}_k$, for each $k = 1, 2, \ldots, K$, whereas $\mathcal{N}_{\text{cut}}$ denotes the set of cut nets. The vertex partition is decoded as a partial row reordering of matrix $A$ such that
the rows associated with vertices in $V_{k+1}$ are ordered after the rows associated with vertices $V_k$, $k = 1, 2, \ldots, K - 1$. The net partition is decoded as a partial column reordering of matrix $A$ such that the columns associated with nets in $N_{k+1}$ are ordered after the columns associated with nets in $N_k$, $k = 1, 2, \ldots, K - 1$, whereas the columns associated with the cut nets are ordered last to constitute the column border.

2.5. Bipartite graph model for sparse matrices. In the bipartite graph model $B(A) = (V, E)$ of matrix $A$, there exists one row vertex $v_r^i \in R$ representing row $r_i$, and there exists one column vertex $v_c^j \in C$ representing column $c_j$, where $R$ is the set of row vertices and $C$ is the set of column vertices. These vertex sets $R$ and $C$ form the vertex bipartition $V = R \cup C$. There is an edge between vertices $v_r^i \in R$ and $v_c^j \in C$ if and only if the respective matrix entry $a_{ij}$ is nonzero.

3. Related work. The main focus of this work is to perform iteration and data reordering, without changing the conventional CSR-based SpMxV codes, whereas cache aware techniques such as prefetching, blocking, etc. are out of the scope of this paper. So we summarize the related work on iteration and data reordering for irregular applications which usually use index arrays to access other arrays. Iteration and data reordering approaches can also be categorized as dynamic and static. Dynamic schemes [13, 15, 12, 35, 19] achieve runtime reordering transformations by analyzing the irregular memory access patterns through adopting inspector/executor strategy [29]. Reordering rows/columns of irregularly sparse matrices to exploit locality during SpMxV operations can be considered as a static case of such general iteration/data reordering problem. We call it a static case [38, 41, 32, 42] since the sparsity pattern of matrix $A$ together with the CSR- or CSC-based SpMxV scheme determines the memory access pattern. In the CSR scheme, iteration order corresponds to row order of matrix $A$ and data order corresponds to column order, whereas a dual discussion applies for CSC.

Dynamic and static transformation heuristics mainly differ in the preprocessing times. Fast heuristics are usually used for dynamic reordering transformations, whereas much more sophisticated heuristics are used for static case. The preprocessing time for the static case can amortize the performance improvement during repeated computations with the same memory access pattern. Repeated SpMxV computations involving the same matrix or matrices with the same sparsity pattern constitute a very typical case of such static case.

Ding and Kennedy [15] propose the locality grouping and consecutive packing (CPACK) heuristics for runtime iteration and data reordering, respectively. The locality grouping heuristic traverses the data objects in a given order and clusters all the iterations that access the first data item, then the second, and etc. The CPACK heuristic reorders the data objects on a first-touch-first basis. The locality grouping heuristic is also referred to as consecutive packing for iterations (CPACKIter) in [35] and this heuristic is equivalent to the iteration reordering heuristic proposed by Das et al. [13] As also mentioned in [15, 19], these heuristics suffer from not explicitly considering different reuse patterns of different data objects because the data objects and iterations are traversed in a given order.

Space-filling curves such as Hilbert and Morton as well as recursive storage schemes such as quadtree are used for iteration reordering in improving locality in dense matrix operations [16, 24, 17] and in sparse matrix operations [18]. Space-filling curves [12] and hierarchical graph clustering algorithms (GPART) [19] are utilized for data reordering in improving locality in n-body simulation applications.

Strout et al. [34] integrate run-time data and iteration reordering transformations such as lexicographically grouping, CPACK and GPART into a compile time framework and they show that sparse tiling may improve performance of these transformations depending on the underlying architecture. Strout and Hovland [35] extend the work in [34] and propose
hypergraph-based models for data and iteration reordering transformations. They introduce a
temporal locality hypergraph model for ordering iterations to exploit temporal locality. They
also generalize spatial locality graph model to spatial locality hypergraph model to encompass
the applications having multiple arrays that are accessed irregularly. Additionally, they pro-
pose a modified algorithm like Breadth-First Search (BFS) for ordering data and iterations
simultaneously, whereas Breadth-First Search is used for only data ordering in [2]. Strout
and Hovland [35] also propose metrics to determine which reordering heuristic is expected to
yield better performance.

Das et al. [13] use reordering techniques in their implementation of three-dimensional
unstructured grid Euler-solver to improve cache utilization. They reorder unstructured mesh
dges incident on the same node consecutively. They also use Reverse Cuthill McKee (RCM)
method to reorder nodes of the mesh. Burgess and Giles [5] examine effects of reordering
techniques in unstructured grid applications They report that reordering meshes that are gen-
erated without any cache optimization may result increase in performance according to appli-
cation: Original orderings give better results in Jacobi solver, whereas reordered meshes give
better results in conjugate gradients method.

Al-Furaih and Ranka [2] introduce interaction graph model to investigate optimizations
for unstructured iterative applications in which the computational structure remains static or
changes only slightly through iterations. They compare several methods to reorder data el-
ements through reordering the vertices of the interaction graph. They report that BFS, as
a fast reordering heuristic, can be applied to a static structure once or to a dynamic struc-
ture between tens of iterations. The other reordering methods are based on top-down graph
partitioning, BFS ordering after graph partitioning and reordering via finding connected com-
ponents that can fit into cache.

In the rest of this section, we discuss the related work on improving locality in SpMxV
operations. Agarwal et al. [1] try to improve SpMxV by extracting dense block structures.
Their methods consist of examining row blocks to find dense subcolumns and reorder these
subcolumns consecutively. Temam and Jalby [36] analyze the cache miss behaviour of Sp-
MxV. They report that cache hit ratio decreases as bandwidth of sparse matrix increases be-
yond the cache size and conclude that bandwidth reduction algorithms improve cache utiliza-

Toledo [38] compares several techniques to reduce cache misses in SpMxV. He uses
graph theoretic methods such as Cuthill McKee (CM), RCM and top-down graph partition-
ing for reordering matrices and other improvement techniques such as blocking, prefetching
and instruction-level-related optimization. They report that they cannot improve SpMxV per-
formance through row/column reordering over original matrices. White and Sadayappan [41]
discuss data locality issues in SpMxV in detail. They compare SpMxV performance of CSR,
CSC and blocked versions of CSR and CSC. They also propose a graph-partitioning-based
row/column reordering method which is similar to that of Toledo. They report that they
can not achieve performance improvement over the original ordering as also reported by
Toledo [38]. Haque and Hossain [20] propose a column reordering method based on Gray

There are several works on row/column reordering based on similar TSP formulations.
Heras et al. [23] define four distance functions for edge weighting depending on the similarity
of sparsity patterns between row/columns. Pichel et al. [30] use TSP-based reordering and
blocking technique to show improvements in both single processor performance and multi-
computer performance. Pichel et al. [31] compare the performance of a number of reordering
techniques which utilize TSP, top-down graph partitioning, RCM, Approximate Minimum
Degree on simultaneous multithreading architectures. Pinar and Heath [32] propose a TSP-
based column reordering for permuting nonzeros of a given matrix into contiguous blocks with the objective of decreasing the number of indirections in the CSR-based SpMxV. They compare the performance of their method to that of the RCM technique.

In a very recent work, Yzelman and Bisseling [42] propose a row/column reordering scheme based on partitioning row-net hypergraph representation of a given sparse matrix for CSR-based SpMxV. They achieve spatial locality on \textit{x-vector} entries by clustering the columns with similar sparsity pattern. They also exploit temporal locality for \textit{x-vector} entries by using zig-zag property of ZZCSR and ZZICSR schemes mentioned in Section 2.1.

4. Single-SpMxV framework. In this framework, the \textit{y-vector} results are computed by performing a single SpMxV operation, i.e., \( y \leftarrow Ax \). The objective in this scheme is to reorder the columns and rows of matrix \( A \) for maximizing the exploitation of temporal and spatial locality in accessing \textit{x-vector} entries. That is, the objective is to find row and column permutation matrices \( P_r \) and \( P_c \) so that \( y \leftarrow Ax \) is computed as \( \hat{y} \leftarrow \hat{Ax} \), where \( \hat{A} = P_r AP_c \), \( \hat{x} = xP_c \), and \( \hat{y} = P_r y \). For the sake of simplicity of presentation, reordered input and output vectors \( \hat{x} \) and \( \hat{y} \) will be referred to as \( x \) and \( y \) in the rest of the paper.

Recall that temporal locality in accessing \textit{y-vector} entries is not feasible, whereas spatial locality is achieved automatically because \textit{y-vector} results are stored and processed consecutively. Reordering the rows with similar sparsity pattern nearby increases the possibility of exploiting temporal locality in accessing \textit{x-vector} entries. Reordering the columns with similar sparsity pattern nearby increases the possibility of exploiting spatial locality in accessing \textit{x-vector} entries. This row/column reordering problem can also be considered as a row/column clustering problem and this clustering process can be achieved in two distinct ways: top-down and bottom-up. In this section, we propose and discuss cache-size-aware top-down approaches based on 1D and 2D partitioning of a given matrix. Although a bottom-up approach based on hierarchical clustering of rows/columns with similar patterns is feasible, such a scheme is not discussed in this work.

4.1. Row/column reordering based on 1D matrix partitioning. We consider a row/column reordering which permutes a given matrix \( A \) into a \( K \)-way columnwise singly-bordered block-diagonal (SB) form

\[
\hat{A} = A_{SB} = P_r AP_c = \begin{bmatrix}
A_{11} & A_{1B} \\
A_{22} & A_{2B} \\
\vdots & \vdots \\
A_{KK} & A_{KB}
\end{bmatrix} = \begin{bmatrix}
R_1 \\
R_2 \\
\vdots \\
R_K
\end{bmatrix} = \begin{bmatrix}
C_1 \\
C_2 \\
\vdots \\
C_K \\
C_B
\end{bmatrix}.
\]

(4.1)

Here, \( A_{kk} \) denotes the \( k \)th diagonal block of \( A_{SB} \). \( R_k = [0 \ldots 0 A_{kk} 0 \ldots 0 A_{kB}] \) denotes the \( k \)th row slice of \( A_{SB} \), for \( k = 1 \ldots K \). \( C_k = [0 \ldots 0 A_{kk}^T 0 \ldots 0]^T \) denotes the \( k \)th column slice of \( A_{SB} \), for \( k = 1 \ldots K \), and \( C_B \) denotes the column border as follows

\[
C_B = \begin{bmatrix}
A_{1B} \\
A_{2B} \\
\vdots \\
A_{KB}
\end{bmatrix}.
\]

(4.2)

Each column in the border \( C_B \) is called a \textit{row-coupling column} or simply a \textit{coupling column}. Let \( \lambda(c_j) \) denote the number of submatrices that contain at least one nonzero of column \( c_j \)
of matrix $A_{SB}$, i.e.,

$$\lambda(c_j) = |\{ R_k : c_j \in R_k \}| \quad (4.3)$$

In other words, $\lambda(c_j)$ denotes the row-slice connectivity or simply connectivity of column $c_j$ in $A_{SB}$. In this notation, a column $c_j$ is a coupling column if $\lambda(c_j) > 1$.

The individual $y \leftarrow Ax$ can be equivalently represented as $K$ output-independent but input-dependent SpMxV operations, i.e., $y_k \leftarrow R_k x$ for $k = 1 \ldots K$, where each submatrix $R_k$ is assumed to be stored in CSR scheme. These SpMxV operations are input dependent because of the $x$-vector entries corresponding to the coupling columns. The following theorem gives the guidelines for a “good” cache-size-aware row/column reordering based on 1D partitioning.

**Theorem 1.** Given a $K$-way SB form of matrix $A$ such that every submatrix $R_k$ fits into the cache, then the number $\Phi(A_{SB})$ of cache misses due to the access of $x$-vector entries can be upperbounded as

$$\Phi(A_{SB}) \leq \sum_{c_j \in A_{SB}} \lambda(c_j) \quad (4.4)$$

under the fully-associative cache assumption.

**Proof.** Since each submatrix $R_k$ fits into the cache, each $x$-vector entry corresponding to a nonzero column of matrix $R_k$ will be loaded to the cache at most once during the $y_k \leftarrow R_k x$ multiply, under the full-associativity assumption. Therefore for a column $c_j$, the maximum number of cache misses that can occur is bounded above by $\lambda(c_j)$ due to the access of the corresponding $x$-vector entry $x_j$. Thus, the number $\Phi(A_{SB})$ of cache misses due to the access of $x$-vector entries cannot exceed $\sum c_j \lambda(c_j)$. \[\Box\]

Theorem 1 leads us to a cache-size-aware top-down row/column reordering through an $A$-to-$A_{SB}$ transformation that minimizes the sum $\sum c_j \lambda(c_j)$ of the connectivity values of columns. Here, minimizing this sum relates to minimizing the cache misses due to the loss of temporal locality. More precisely, under the assumption that there is no empty column, since there has to be at least one cache miss for each column $c_j$, the column $c_j$ brings $\lambda(c_j) - 1$ extra cache misses due to temporal locality in the worst case.

**Corollary 1.** Given a $K$-way SB form of matrix $A$ such that every submatrix $R_k$ fits into the cache, then the number $\Phi_{\text{additional}}(A_{SB})$ of additional cache misses due to the access of $x$-vector entries can be upperbounded as

$$\Phi_{\text{additional}}(A_{SB}) \leq \sum_{c_j \in A_{SB}} (\lambda(c_j) - 1) \quad (4.5)$$

under the fully-associative cache assumption.

As discussed in [3], this $A$-to-$A_{SB}$ transformation problem can be formulated as an HP problem using the column-net model of matrix $A$ with the part size constraint of cache size and the partitioning objective of minimizing cutsize according to the connectivity metric definition given in Equation 2.3.

4.2. Row/column reordering based on 2D matrix partitioning. We consider a row/column reordering which permutes a given matrix $A$ into a $K$-way doubly-bordered block-
diagonal (DB) form

\[
\hat{A} = A_{DB} = P_t A P_c = \begin{bmatrix}
A_{11} & A_{1B} \\
A_{22} & A_{2B} \\
\vdots & \vdots \\
A_{K\bar{K}} & A_{K\bar{B}} \\
A_{B1} & A_{B2} & \cdots & A_{BK} & A_{BB}
\end{bmatrix} = \begin{bmatrix}
R_1 \\
R_2 \\
\vdots \\
R_K \\
R_B
\end{bmatrix} = \begin{bmatrix}
A'_{SB} \\
R_B
\end{bmatrix}
\]

Here, \( R_B = [A_{B1} \ A_{B2} \ \ldots \ A_{BK} \ A_{BB}] \) denotes the row border. Each row in \( R_B \) is called a column-coupling row or simply a coupling row. \( A'_{SB} \) denotes the columnwise SB part of \( A_{DB} \) excluding the row border \( R_B \). \( R_k \) denotes the \( k \)th row slice of both \( A'_{SB} \) and \( A_{DB} \). \( \lambda(c_j) \) denotes the connectivity of column \( c_j \) in \( A'_{SB} \). \( C_B \) denotes the column border of \( A'_{SB} \), whereas \( C_B = [C_B^T \ A_{BB}^T]^T \) denotes the column border of \( A_{DB} \). \( C_k = [0 \ldots 0 \ A_{kB}^T 0 \ldots 0 \ A_{BB}^T]^T \) denotes the \( k \)th column slice of \( A_{DB} \).

The following theorem gives the guidelines for a “good” cache-size-aware row/column reordering based on 2D partitioning.

**Theorem 2.** Given a K-way DB form of matrix \( A \) such that every submatrix \( R_k \) of \( A'_{SB} \) fits into the cache, then the number \( \Phi(A_{DB}) \) of cache misses due to the access of \( x \)-vector entries can be upperbounded as

\[
\Phi(A_{DB}) \leq \sum_{c_j \in A'_{SB}} \lambda(c_j) + \sum_{r_i \in R_B} \text{nnz}(r_i)
\]

under the fully-associative cache assumption.

**Proof.** We can consider the \( y \leftarrow Ax \) multiply as two output-independent but input-dependent SpMxVs: \( y_{SB} \leftarrow A'_{SB} x \) and \( y_B \leftarrow R_B x \), where \( y = [y_{SB}^T \ y_B^T]^T \). Thus \( \Phi(A_{DB}) \leq \Phi(A'_{SB}) + \Phi(R_B) \). By proof of Theorem 1, we already have \( \Phi(A'_{SB}) \leq \sum_{c_j} \lambda(c_j) \). In the \( y_B \leftarrow R_B x \) multiply, we have at most \( \text{nnz}(r_i) \) \( x \)-vector access for each column-coupling row \( r_i \) of \( R_B \). Hence, \( \Phi(R_B) \leq \sum_{r_i \in R_B} \text{nnz}(r_i) \) thus concluding the proof. \( \square \)

Theorem 2 leads us to a cache-size-aware top-down row/column reordering through an \( A \)-to-\( A_{DB} \) transformation that minimizes the right-hand side of the inequality given in (4.7). Here, minimizing this sum relates to minimizing the cache misses due to temporal locality. More precisely, under the assumption that there is no empty column, there has to be at least one cache miss for each column \( c_j \), which concludes the following corollary.

**Corollary 2.** Given a K-way DB form of matrix \( A \) such that every submatrix \( R_k \) of \( A'_{SB} \) fits into the cache, then the number \( \Phi_{\text{additional}}(A_{DB}) \) of cache misses due to the access of \( x \)-vector entries can be upperbounded as

\[
\Phi_{\text{additional}}(A_{DB}) \leq \sum_{c_j \in A'_{SB}} \lambda(c_j) - 1 + \sum_{r_i \in R_B} \text{nnz}(r_i)
\]

under the fully-associative cache assumption.

Here we propose to formulate the above-mentioned \( A \)-to-\( A_{DB} \) transformation problem as an HP problem using the row-column-net model of matrix \( A \) with a part size constraint of cache size. In the proposed formulation, column nets are associated with unit cost (i.e., \( \text{cost}(n_j^c) = 1 \) for each column \( c_j \)) and the cost of each row net is set to the number of
nonzeros in the respective row (i.e., $\text{cost}(n'_j) = \text{nnz}(r_i)$). However, existing HP tools do not handle the cutsize definition given in Equation 4.7, because the connectivity metric should be enforced for column nets, whereas the cut-net metric should be enforced for row nets. In order to encapsulate this different cutsize definition, we adapt and enhance the cut-net removal and cut-net splitting techniques adopted in RB algorithms utilized in HP tools. The connectivity of a column net should be calculated in such a way that it is as close as possible to the connectivity of the respective coupling column in the $A'_{SB}$ part of $A_{DB}$. For this purpose, after each bipartitioning step, each cut row-net is removed together with all of its vertices in both sides of the bipartition. Recall that the vertices of a cut net are not removed in the conventional cut-net removal scheme [9]. After applying the proposed removal scheme on the row nets on the cut, the conventional cut-net splitting technique [9] is applied to the column nets on the cut of the bipartition. This enhanced row-column-net model will be abbreviated as the “eRCN” model and the resulting reordering method will be referred to as “sHP$_{eRCN}$”.

The K-way partition $\Pi = \{V_1, \ldots, V_K\}$ of $\mathcal{H}_{RCN}(A)$ obtained as a result of the above-mentioned RB process is decoded as follows to induce the desired DB form of matrix $A$. The rows corresponding to the cut row-nets are permuted to the end to constitute the coupling rows of the row border $R_B$. The rows corresponding to the uncut row-nets of part $V_k$ are permuted to the $k$th row slice $R_k$. The columns corresponding to the uncut column-nets of part $V_k$ are permuted to the $k$th column slice $C_k$. It is clear that the columns corresponding to the cut column-nets remain in the column border $C_B$ of $A_{DB}$ and hence only those columns have the potential to remain in the column border $C'_B$ of $A'_{SB}$. Some of these columns may be permuted to a column slice $C_k$ if all of its nonzeros become confined to row slice $R_k$ and row border $R_B$. Such cases may occur as follows: Consider a cut column-net $n'_j$ of a bipartition obtained at a particular RB step. If the row nets corresponding to the rows that contain the nonzeros corresponding to $n'_j$’s vertices that lie on one part of the bipartition all become cut nets in the following RB steps, then column $c_j$ is no longer a coupling column and it can be safely permuted to column slice $C_k$. For such cases, the proposed scheme fails to correctly encapsulate the column connectivity cost in $A'_{SB}$. The proposed cut row-net removal scheme avoids such column-connectivity miscalculations that may occur in the future RB steps due the cut row-nets of the current bipartition. However, it is clear that our scheme cannot avoid such possible errors (related to the cut column-nets of the current bipartition) that may occur due to the row nets to be cut in the future RB steps.

5. Multiple-SpMxV framework. In this framework, we assume that the nonzeros of matrix $A$ are partitioned arbitrarily among $K$ $A^k$ matrices such that each matrix $A^k$ matrix contains a mutually disjoint subset of nonzeros. Then matrix $A$ can be written as the sum

$$A = A^1 + A^2 + \cdots + A^K.$$  \hspace{1cm} (5.1)

In this framework, $y \leftarrow Ax$ operation is computed as a sequence of $K$ input- and output-dependent SpMxV operations as shown in Algorithm 3. This splitting of matrix $A$ is not necessarily row disjoint or column disjoint. Thus, the individual SpMxV operations are input dependent because of the shared columns among the $A^k$ matrices, whereas they are output dependent because of the shared rows among the $A^k$ matrices.

Since a global row and column ordering is assumed in Algorithm 3, $A^k$ matrices are likely to contain empty rows. Hence, each individual SpMxV operation $y \leftarrow y + A^k x$ is performed using the ICSR scheme. As seen in Algorithm 3, individual SpMxV results are accumulated in the output vector $y$ on the fly in order to avoid additional write operations.

The partitioning of matrix $A$ into $A^k$ matrices should be done in such a way that the temporal and spatial locality of individual SpMxVs are exploited in order to minimize cache
Algorithm 3 SpMxV algorithm utilizing the multiple-SpMxV framework

Require: \( A = A_1 + A_2 + \cdots + A_K \) partitioning of matrix \( A \) and dense input vector \( x \)

Output: dense vector \( y \)

1: \( y \leftarrow 0^T \)
2: for \( k \leftarrow 1 \) to \( K \) do
3: \( y \leftarrow y + A_k x \)
4: end for
5: return \( y \)

misses. This goal is similar to that of the single-SpMxV framework discussed in Section 4. On the contrary, this framework requires the splitting of matrix \( A \) into \( A_k \) matrices, whereas the single-SpMxV framework uses the method of reordering rows and columns. We discuss pros and cons of this framework compared to the single-SpMxV framework in Section 5.1. In Section 5.2, we also show that splitting matrix \( A \) into \( A_k \) matrices can be formulated as 2D partitioning of matrix \( A \) by utilizing the row-column-net hypergraph model. The order of individual SpMxV operations is also important to exploit temporal locality. We state this ordering problem as an instance of TSP in Section 5.3.

5.1. Pros and cons compared to single-SpMxV framework. The single-SpMxV framework can be considered as a special case of multiple-SpMxV framework in which \( A_k \) matrices are restricted to be row disjoint. Thus, the multiple-SpMxV framework brings an additional flexibility for exploiting the temporal and spatial locality. Clustering \( A \)-matrix rows/subrows with similar sparsity pattern into the same \( A_k \) matrices increases the possibility of exploiting temporal locality in accessing \( x \)-vector entries. Clustering \( A \)-matrix columns/subcolumns with similar sparsity pattern into the same \( A_k \) matrices increases the possibility of exploiting spatial locality in accessing \( x \)-vector entries as well as temporal locality in accessing \( y \)-vector entries.

It is clear that single-SpMxV framework utilizing the CSR scheme severely suffers from dense rows. Dense rows causes loading large number of \( x \)-vector entries to the cache thus disturbing the temporal locality of accessing \( x \)-vector entries. The multiple-SpMxV framework may overcome this deficiency of the single-SpMxV framework through utilizing the flexibility of distributing the nonzeros of dense rows among multiple \( A_k \) matrices in such a way to exploit the temporal locality in the respective \( y \leftarrow y + A_k x \) operations.

However, this additional flexibility comes at a cost of disturbing the following localities compared to single SpMxV approach. There is some disturbance in the spatial locality in accessing the nonzeros of the \( A \) matrix due to the division of three arrays associated with nonzeros into \( K \) parts. However, this disturbance in spatial locality is negligible since elements of each of the three arrays are stored and accessed consecutively during each SpMxV operation. That is, at most \( 3(K - 1) \) extra cache misses occur compared to the single SpMxV scheme due to the disturbance of spatial locality in accessing the nonzeros of \( A \)-matrix. More importantly, multiple read/writes of the individual SpMxV results might bring some disadvantages compared to single SpMxV scheme. These multiple read/writes disturb the spatial locality of accessing \( y \)-vector entries as well as introducing a temporal locality exploitation problem in \( y \)-vector entries.

The following theorem gives the guidelines for a “good” matrix splitting based on 2D partitioning.

**Theorem 3.** Consider a partition \( \Pi(A) \) of matrix \( A \) into \( K \) nonzero-disjoint matrices \( A^1, A^2, \ldots, A^K \). Let \( \lambda(r_i) \) denote the number of \( A^k \) matrices that contain at least one nonzero of row \( r_i \) of matrix \( A \), i.e., \( \lambda(r_i) = |\{ A^k : r_i \in A^k \}|. \) Similarly let \( \lambda(c_j) \) denote...
the number of $A^k$ matrices that contain at least one nonzero of column $c_j$ of matrix $A$, i.e., $\lambda(c_j) = |\{A^k : c_j \in A^k\}|$. Let $q$ denote the size of the largest $A^k$ matrix in terms of the number of caches it can fit into. Then the number $\Phi(\Pi(A))$ of cache misses due to the access of $x$-vector and $y$-vector entries can be upperbounded as

$$\Phi(\Pi(A)) \leq \sum_{r_i \in A} \lambda(r_i) + q \sum_{c_j \in A} \lambda(c_j) \tag{5.2}$$

under the fully-associative cache assumption.

Proof. For each matrix $A^k$, each $y$-vector result of $A^k$ is written only once to the memory. For the sake of simplicity, we refer $\Phi(\Pi(A))$ as $\Phi$. Let $\Phi_x$ and $\Phi_y$ respectively denote the number of cache misses due to the access of $x$-vector and $y$-vector entries for $\Pi(A)$. Then, $\Phi = \Phi_x + \Phi_y$. The number of cache misses due to the access of $y_i$ is at most $\lambda(r_i)$ which happens when no cache-reuse occurs in accessing $y_i$, that is,

$$\Phi_y \leq \sum_{r_i \in A} \lambda(r_i). \tag{5.3}$$

Let $q_k$ denote the minimum number of caches that matrix $A^k$ can fit into. Since full-associativity is assumed, for each matrix $A^k$, each $x$-vector entry of $A^k$ is accessed at most $q_k$ times. Therefore, the number of cache misses due to the access of $x_j$ is at most $q_k$ for each matrix $A^k$ that requires $x_j$ to be accessed. Then,

$$\Phi_x \leq \sum_{c_j \in A} \sum_{k : c_j \in A^k} q_k \leq \sum_{c_j \in A} \sum_{k : c_j \in A^k} q \leq q \sum_{c_j \in A} \lambda(c_j) \tag{5.4}$$

Equations 5.3 and 5.4 together lead to Equation 5.2. \qed

**Corollary 3.** If each $A^k$ matrix fits into the cache then the number $\Phi(\Pi(A))$ of cache misses due to the access of $x$-vector and $y$-vector entries can be upperbounded as

$$\Phi(\Pi(A)) \leq \sum_{r_i \in A} \lambda(r_i) + \sum_{c_j \in A} \lambda(c_j) \tag{5.5}$$

in case of unit cache-line size and full-associativity of cache is assumed.

**5.2. Splitting $A$ into $A^k$ matrices.** Corollary 3 leads us to a cache-size-aware top-down matrix splitting which minimizes the sum $\sum_{r_i} \lambda(r_i) + \sum_{c_j} \lambda(c_j)$ of $\lambda$ values of rows and columns such that the storage of each $A^k$ matrix fits into the cache. Here, the minimization objective relates to minimizing the cache misses due to temporal locality. More precisely, under the assumption that there is no empty column, there is at least one cache miss for each row $r_i$ and each column $c_j$. Thus, row $r_i$ and column $c_j$, respectively, incurs $\lambda(r_i) - 1$ and $\lambda(c_j) - 1$ additional cache misses due to the loss of temporal locality in the worst case.

**Corollary 4.** Given a $K$-way matrix splitting $\Pi(A)$ of matrix $A$ such that every $A^k$ matrix fits into the cache, then the number $\Phi_{\text{additional}}(\Pi(A))$ of additional cache misses due to the access of $x$-vector and $y$-vector entries can be upperbounded as

$$\Phi_{\text{additional}}(\Pi(A)) \leq \sum_{r_i \in A} (\lambda(r_i) - 1) + \sum_{c_j \in A} (\lambda(c_j) - 1) \tag{5.6}$$

The matrix partitioning problem can be formulated as an HP problem using the row-column-net model [7, 11] of matrix $A$ with a part size constraint of cache size and partitioning objective of minimizing cutsize according to the connectivity metric definition given in Equation 2.3.
5.3. Ordering individual SpMxV operations. The above-mentioned objective in partitioning matrix $A$ into $A^k$ matrices is to exploit temporal and spatial locality of individual SpMxVs in order to minimize cache misses. However, when all SpMxVs are considered, data reuse between two consecutive SpMxVs must also be considered to exploit temporal locality. We give an exact lower bound for the cache misses due to the access of $x$-vector and $y$-vector entries for a given order of $A^k$ matrices.

**Theorem 4.** Consider a splitting $\hat{\Pi}(A)$ of matrix $A$ into $K$ nonzero-disjoint matrices $A^1, A^2, \ldots, A^K$ with a given ordering of the $A^k$ matrices. A subchain of $A^k$ matrices is said to cover a row $r_i$ and a column $c_j$ if each matrix in the subchain contains at least one nonzero of row $r_i$ and column $c_j$, respectively. Let $\gamma(r_i)$ and $\gamma(c_j)$ denote the number of maximal $A^k$-matrix subchains that cover row $r_i$ and column $c_j$, respectively. If no $A^k$ matrix can fit into one cache, then the number $\Phi(\hat{\Pi}(A))$ of cache misses due to the access of $x$-vector and $y$-vector entries can be lowerbounded as

$$\Phi(\hat{\Pi}(A)) \geq \sum_{r_i \in A} \gamma(r_i) + \sum_{c_j \in A} \gamma(c_j)$$

(5.7)

under the fully-associative cache and unit cache-line-size assumption.

**Proof.** We will give the proof only for the columns, since a similar proof applies for the rows; then total number of cache misses can be written as a sum of cache misses due to access of $y$-vector entries and $x$-vector entries and can be formulated as

$$\Phi(\hat{\Pi}(A)) = \Phi_r(\hat{\Pi}(A)) + \Phi_c(\hat{\Pi}(A))$$

(5.8)

Consider a column $c_j$ of matrix $A$. Then there exists $\gamma(c_j)$ maximal $A^k$-matrix subchains that cover column $c_j$. Since no $A^k$ matrix can fit into one cache, it is guaranteed that there will be no cache reuse of column $c_j$ between two different maximal $A^k$-matrix subchains that cover $c_j$. Therefore, at least $\gamma(c_j)$ cache misses will occur for each column $c_j$ which means that the number $\Phi_c(\hat{\Pi}(A))$ of cache misses due to the access of $x$-vector entries is greater than or equal to $\sum_{c_j} \gamma(c_j)$ in the case of unit cache-line size. $\square$

**Theorem 5.** Consider the TSP Instance $(G = (V, E), w)$, where vertex set $V$ denotes the $K$ $A^k$ matrices. The weight $w(k, \ell)$ of edge $e_{k\ell} \in E$ is set to be equal to the sum of the number of shared rows and the number of shared columns between $A^k$ and $A^\ell$. Then, finding an order on $V$ that maximizes the path weight corresponds to finding an order of $A^k$ matrices which minimizes $\Psi = \sum_{r_i} \gamma(r_i) + \sum_{c_j} \gamma(c_j)$.

**Proof.** Below, let $A^{\Gamma(\ell)}$ denote the $\ell$th $A^k$ matrix in the ordering $\Gamma$ of $A^k$ matrices and let $A^k$ also denote the set of rows and columns that belong to the matrix $A^k$.

$$\Psi = \sum_{r_i} \left[ |A^{\Gamma(1)} \cap \{r_i\}| + \sum_{\ell=2}^K (|A^{\Gamma(\ell)} - A^{\Gamma(\ell-1)}| \cap \{r_i\}) \right]$$

$$+ \sum_{c_j} \left[ |A^{\Gamma(1)} \cap \{c_j\}| + \sum_{\ell=2}^K (|A^{\Gamma(\ell)} - A^{\Gamma(\ell-1)}| \cap \{c_j\}) \right]$$

$$= |A^{\Gamma(1)}| + \sum_{\ell=2}^K |(A^{\Gamma(\ell)} - A^{\Gamma(\ell-1)})| = |A^{\Gamma(1)}| + \sum_{\ell=2}^K (|A^{\Gamma(\ell)}| - |A^{\Gamma(\ell)} \cap A^{\Gamma(\ell-1)}|)$$

$$= \sum_{\ell=1}^K |A^{\Gamma(\ell)}| - \sum_{\ell=2}^K |A^{\Gamma(\ell)} \cap A^{\Gamma(\ell-1)}| = \sum_{\ell=1}^K |A^{\Gamma(\ell)}| - \sum_{\ell=2}^K w(\Gamma(\ell), \Gamma(\ell - 1))$$
The maximum value of \( \sum_{\ell=2}^{K} w(\Gamma(\ell), \Gamma(\ell-1)) \) will yield the minimum value of \( \sum_{i} \gamma(r_{i}) + \sum_{j} \gamma(c_{j}) \). Then, finding an order on \( V \) that maximizes the path weight \( \sum_{\ell=2}^{K} w(\Gamma(\ell), \Gamma(\ell-1)) \) corresponds to finding an order of submatrices that minimizes \( \sum_{i} \gamma(r_{i}) + \sum_{j} \gamma(c_{j}) \).

According to Theorem 5, the lower bound \( \sum_{i} \gamma(r_{i}) + \sum_{j} \gamma(c_{j}) \) is equal to the objective function of the TSP instance constructed in the theorem. So, the maximization objective in the proposed TSP formulation exactly corresponds to minimizing the lower bound on the number of cache misses due to the access of \( x \)-vector and \( y \)-vector entries.

6. Experimental results. We tested the performance of the proposed methods against three state-of-the-art methods: sBFS [35], sRCM [13, 38, 25] and sHP_{RN} [42]. The small letter “s” used as the first letter in these abbreviations refer to the fact that all of them belong to the single-SpMxV framework described in Section 4. Here, sBFS refers to our adaptation of BFS-based simultaneous data and iteration reordering method of Strout et al. [35] to matrix row and column reordering. Strout et al.’s method depends on implementing breadth-first search on both temporal and spatial locality hypergraphs simultaneously. In our adaptation, we apply BFS on the bipartite graph representation of the matrix, so that the resulting BFS orders on the row and column vertices determine row and column reorderings, respectively. sRCM refers to applying the RCM method, which is widely used for envelope reduction of symmetric matrices, on the bipartite graph representation of the given sparse matrix. Application of the RCM method to bipartite graphs has also been used by Berry et al. [4] to reorder rectangular term-by-document matrices for envelope minimization. sHP_{RN} refers to the work by Yzelman and Bisseling [42] which utilizes top-down HP using the row-net model for CSR-based SpMxV.

The following abbreviations will be used for the proposed methods: sHP_{CN}, sHP_{RCN} and mHP_{RCN}. Here sHP_{CN} and sHP_{RCN} respectively refer to the 1D and 2D matrix partitioning schemes which are described in Sections 4.1 and 4.2 for the single-SpMxV framework. Recall that sHP_{CN} utilizes the column-net model for \( A \)-to-\( A_{SB} \) transformation, whereas sHP_{RCN} utilizes the enhanced row-column-net model that encapsulates a different cutsize metric given in (4.7) for the desired \( A \)-to-\( A_{DB} \) transformation. mHP_{RCN} refers to the method proposed in Section 5 for multiple-SpMxV framework. Note that the small letter “m” is used to indicate the multiple-SpMxV framework. Recall that mHP_{RCN} utilizes the row-column-net model for splitting \( A \) into multiple \( A^{k} \) matrices and a TSP model for ordering \( y \leftarrow y + A^{k}x \) multiplies which are described in Sections 5.2 and 5.3, respectively.

The HP-based top-down reordering methods sHP_{RN}, sHP_{CN}, sHP_{RCN} and mHP_{RCN} are implemented using the state-of-the-art HP tool PaToH [10]. In these implementations, PaToH is used as a 2-way HP tool within the RB paradigm. The hypergraphs representing sparse matrices according to the respective models are recursively bipartitioned into parts until the CSR-storage size of the submatrix (together with the \( x \) and \( y \) vectors) corresponding to a part drops below the cache size. That is, the part-size threshold \( W_{max} \) is set to the cache size (64 KB) and the reordering results for this value of \( W_{max} \) are reported in Tables 6.2–6.6 where Table 6.7 displays the performance variation of HP-based reordering methods with varying \( W_{max} \). PaToH is used with default parameters except the use of heavy connectivity clustering (\( \text{PaToH}_{\text{CRS}, HCC=9} \)) in the sHP_{RN}, sHP_{CN} and sHP_{RCN} methods that belong to the single-SpMxV framework, and the use of absorption clustering using nets (\( \text{PaToH}_{\text{CRS}, ABSHCC=11} \)) in the mHP_{RCN} method that belong to the multiple-SpMxV framework. Since PaToH contains randomized algorithms, the reordering results are reported by averaging the values obtained in 10 different runs, each randomly seeded.

Performance evaluations are carried out on a wide range of matrices obtained from the University of Florida Sparse Matrix Collection [37]. Properties of these matrices are pre-
<table>
<thead>
<tr>
<th>Name</th>
<th>Rows</th>
<th>Cols</th>
<th>Nonzeros</th>
<th>Avg</th>
<th>Max</th>
<th>Cov</th>
<th>Avg</th>
<th>Max</th>
<th>Cov</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Symmetric Matrices</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ncvxq9</td>
<td>16,554</td>
<td>16,554</td>
<td>54,040</td>
<td>3</td>
<td>9</td>
<td>0.5</td>
<td>3</td>
<td>9</td>
<td>0.5</td>
</tr>
<tr>
<td>tuma1</td>
<td>22,967</td>
<td>22,967</td>
<td>87,760</td>
<td>4</td>
<td>5</td>
<td>0.3</td>
<td>4</td>
<td>5</td>
<td>0.3</td>
</tr>
<tr>
<td>bloweybl</td>
<td>30,003</td>
<td>30,003</td>
<td>120,000</td>
<td>4</td>
<td>10,001</td>
<td>14.4</td>
<td>4</td>
<td>10,001</td>
<td>14.4</td>
</tr>
<tr>
<td>bloweya</td>
<td>30,004</td>
<td>30,004</td>
<td>150,009</td>
<td>5</td>
<td>10,001</td>
<td>11.6</td>
<td>5</td>
<td>10,001</td>
<td>11.6</td>
</tr>
<tr>
<td>brainpc2</td>
<td>27,607</td>
<td>27,607</td>
<td>179,395</td>
<td>7</td>
<td>13,799</td>
<td>20.2</td>
<td>7</td>
<td>13,799</td>
<td>20.2</td>
</tr>
<tr>
<td>a5esind</td>
<td>60,008</td>
<td>60,008</td>
<td>255,004</td>
<td>4</td>
<td>9,993</td>
<td>12.7</td>
<td>4</td>
<td>9,993</td>
<td>12.7</td>
</tr>
<tr>
<td>dixmaanl</td>
<td>60,000</td>
<td>60,000</td>
<td>299,998</td>
<td>5</td>
<td>5</td>
<td>0.0</td>
<td>5</td>
<td>5</td>
<td>0.0</td>
</tr>
<tr>
<td>shallow_water1</td>
<td>81,920</td>
<td>81,920</td>
<td>327,680</td>
<td>4</td>
<td>4</td>
<td>0.0</td>
<td>4</td>
<td>4</td>
<td>0.0</td>
</tr>
<tr>
<td>c-65</td>
<td>48,066</td>
<td>48,066</td>
<td>360,528</td>
<td>8</td>
<td>3,276</td>
<td>2.5</td>
<td>8</td>
<td>3,276</td>
<td>2.5</td>
</tr>
<tr>
<td>brainpc2</td>
<td>27,607</td>
<td>27,607</td>
<td>179,395</td>
<td>7</td>
<td>13,799</td>
<td>20.2</td>
<td>7</td>
<td>13,799</td>
<td>20.2</td>
</tr>
<tr>
<td><strong>Square Nonsymmetric Matrices</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>poli_large</td>
<td>15,575</td>
<td>15,575</td>
<td>33,074</td>
<td>2</td>
<td>491</td>
<td>4.2</td>
<td>2</td>
<td>18</td>
<td>0.2</td>
</tr>
<tr>
<td>powersim</td>
<td>15,838</td>
<td>15,838</td>
<td>67,562</td>
<td>4</td>
<td>40</td>
<td>0.6</td>
<td>4</td>
<td>41</td>
<td>0.8</td>
</tr>
<tr>
<td>memplus</td>
<td>17,758</td>
<td>17,758</td>
<td>126,150</td>
<td>7</td>
<td>574</td>
<td>3.1</td>
<td>7</td>
<td>574</td>
<td>3.1</td>
</tr>
<tr>
<td>Zhao1</td>
<td>33,861</td>
<td>33,861</td>
<td>166,453</td>
<td>5</td>
<td>6</td>
<td>0.1</td>
<td>5</td>
<td>7</td>
<td>0.2</td>
</tr>
<tr>
<td>mult_dcop_01</td>
<td>25,187</td>
<td>25,187</td>
<td>193,276</td>
<td>8</td>
<td>22,767</td>
<td>18.7</td>
<td>8</td>
<td>22,774</td>
<td>18.8</td>
</tr>
<tr>
<td>jan99jac120sc</td>
<td>41,374</td>
<td>41,374</td>
<td>260,202</td>
<td>6</td>
<td>68</td>
<td>1.1</td>
<td>6</td>
<td>138</td>
<td>2.3</td>
</tr>
<tr>
<td>circuit_4</td>
<td>80,209</td>
<td>80,209</td>
<td>307,604</td>
<td>4</td>
<td>6,750</td>
<td>7.8</td>
<td>4</td>
<td>8,900</td>
<td>10.5</td>
</tr>
<tr>
<td>ckt11752_dc_1</td>
<td>49,702</td>
<td>49,702</td>
<td>333,029</td>
<td>7</td>
<td>2,921</td>
<td>3.5</td>
<td>7</td>
<td>2,921</td>
<td>3.5</td>
</tr>
<tr>
<td>poisson3Da</td>
<td>13,514</td>
<td>13,514</td>
<td>352,762</td>
<td>26</td>
<td>110</td>
<td>0.5</td>
<td>26</td>
<td>110</td>
<td>0.5</td>
</tr>
<tr>
<td>bcircuit</td>
<td>68,902</td>
<td>68,902</td>
<td>375,558</td>
<td>6</td>
<td>34</td>
<td>0.4</td>
<td>6</td>
<td>34</td>
<td>0.4</td>
</tr>
<tr>
<td>g7jac120</td>
<td>35,550</td>
<td>35,550</td>
<td>475,296</td>
<td>13</td>
<td>153</td>
<td>1.7</td>
<td>13</td>
<td>120</td>
<td>1.7</td>
</tr>
<tr>
<td>e40r0100</td>
<td>17,281</td>
<td>17,281</td>
<td>553,562</td>
<td>32</td>
<td>62</td>
<td>0.5</td>
<td>32</td>
<td>62</td>
<td>0.5</td>
</tr>
<tr>
<td><strong>Rectangular Matrices</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lp_dfl001</td>
<td>6,071</td>
<td>12,230</td>
<td>35,632</td>
<td>6</td>
<td>228</td>
<td>1.3</td>
<td>3</td>
<td>14</td>
<td>0.4</td>
</tr>
<tr>
<td>ge</td>
<td>10,099</td>
<td>16,369</td>
<td>44,825</td>
<td>4</td>
<td>48</td>
<td>0.8</td>
<td>3</td>
<td>36</td>
<td>0.9</td>
</tr>
<tr>
<td>ex3sta1</td>
<td>17,843</td>
<td>17,516</td>
<td>68,779</td>
<td>4</td>
<td>8</td>
<td>0.4</td>
<td>4</td>
<td>46</td>
<td>1.4</td>
</tr>
<tr>
<td>lp_stocfor3</td>
<td>16,675</td>
<td>23,541</td>
<td>76,473</td>
<td>5</td>
<td>15</td>
<td>0.7</td>
<td>3</td>
<td>18</td>
<td>1.0</td>
</tr>
<tr>
<td>cq9</td>
<td>9,278</td>
<td>21,534</td>
<td>96,653</td>
<td>10</td>
<td>391</td>
<td>3.5</td>
<td>5</td>
<td>24</td>
<td>1.0</td>
</tr>
<tr>
<td>pse0</td>
<td>26,722</td>
<td>11,028</td>
<td>102,432</td>
<td>4</td>
<td>4</td>
<td>0.1</td>
<td>9</td>
<td>68</td>
<td>0.7</td>
</tr>
<tr>
<td>co9</td>
<td>10,789</td>
<td>22,924</td>
<td>109,651</td>
<td>10</td>
<td>441</td>
<td>3.6</td>
<td>5</td>
<td>28</td>
<td>1.1</td>
</tr>
<tr>
<td>baxter</td>
<td>27,441</td>
<td>30,733</td>
<td>111,576</td>
<td>4</td>
<td>2,951</td>
<td>8.7</td>
<td>4</td>
<td>46</td>
<td>1.6</td>
</tr>
<tr>
<td>graphics</td>
<td>29,493</td>
<td>11,822</td>
<td>117,954</td>
<td>4</td>
<td>4</td>
<td>0.0</td>
<td>10</td>
<td>87</td>
<td>1.0</td>
</tr>
<tr>
<td>fome12</td>
<td>24,284</td>
<td>48,920</td>
<td>142,528</td>
<td>6</td>
<td>228</td>
<td>1.3</td>
<td>3</td>
<td>14</td>
<td>0.4</td>
</tr>
<tr>
<td>route</td>
<td>20,894</td>
<td>43,019</td>
<td>206,782</td>
<td>10</td>
<td>2,761</td>
<td>7.1</td>
<td>5</td>
<td>44</td>
<td>1.0</td>
</tr>
<tr>
<td>fxm4_6</td>
<td>22,400</td>
<td>47,185</td>
<td>265,442</td>
<td>12</td>
<td>57</td>
<td>1.0</td>
<td>6</td>
<td>24</td>
<td>1.1</td>
</tr>
</tbody>
</table>

The single-level cache simulator developed by Yzelman and Bisseling [42] is used for performance evaluation. The simulator is configured to have 64 KB, 2-way set-associative cache with a line size of 64 bytes (8 words). Some of the experiments are conducted to show the sensitivities of the methods to the cache-line size without changing the other cache parameters. Double precision arithmetic is used during SpMxVs computations. In the simulations, since the ICSR [27] storage scheme is to be used in the multiple-SpMxV framework...
as discussed in Section 5, ICSR is also for all other methods. The ZZCSR scheme proposed by Yzelman and Bisseling [42] is not used in the simulations, since the main purpose of this work is to show the cache miss effects of the six different reordering methods. In the following tables, the performances of the existing and proposed methods are displayed in terms of cache miss ratios. The cache miss ratios are calculated through dividing the number of cache misses for the reordered matrix by the number of cache misses for the original matrix. Only cache misses due to the access of \( x \)-vector and \( y \)-vector entries are reported, whereas compulsory cache misses due to the access of matrix nonzeros are not reported in order to better show the performance differences among the methods.

We introduce Table 6.2 to show the validity of the enhanced row-column-net model proposed in Section 4.2 for the \( \text{sHP}_{eRCN} \) method. In the table, \( \text{sHP}_{RCN} \) refers to a version of the \( \text{sHP}_{eRCN} \) method that utilizes the conventional row-column-net model instead of the enhanced row-column-net model. Table 6.2 displays average performance results of \( \text{sHP}_{RCN} \) and \( \text{sHP}_{eRCN} \) over the three different matrix categories as well as the overall averages. As seen in the table, \( \text{sHP}_{eRCN} \) performs considerably better than \( \text{sHP}_{RCN} \), thus showing the validity of the proposed cutsize definition given in (4.7) according to Theorem 2.

We introduce Table 6.3 to show the merits of the TSP formulation proposed in Theorem 5 for ordering individual \( \text{SpMxV} \) operations in the \( \text{mHP}_{RCN} \) method. Table 6.3 displays average performance results of \( \text{mHP}_{RCN} \) for the random and TSP orderings over the three different matrix categories as well as the overall averages. As seen in the table, TSP ordering leads to considerable performance improvement in the \( \text{mHP}_{RCN} \) method compared to the random ordering. In the following tables, we display the performance results of the \( \text{mHP}_{RCN} \) method that utilizes the TSP ordering. The TSP implementation given in [21] is used in these experiments.

Table 6.4 displays the performance comparison of the existing and proposed methods for each test matrix. The bottom part of the table shows the geometric means of the performance results of the methods over the three different matrix categories as well as the overall averages. Among the existing methods, \( \text{sHP}_{RN} \) performs considerably better than both \( \text{sBFS} \) and \( \text{sRCM} \), whereas \( \text{sRCM} \) perform better than \( \text{sBFS} \). \( \text{sHP}_{RN} \) performs better than both \( \text{sBFS} \) and \( \text{sRCM} \) in reordering 17 test matrices out of 36 in terms of cache misses due to the access of \( x \)-vector and \( y \)-vector entries. However there are test matrices such as bloweya, brainpc, memplus and Zhao1 for which \( \text{sHP}_{RN} \) performs significantly worse than both.
### Table 6.4
Simulation results for all test matrices (cache size = part-size threshold = 64 KB)

<table>
<thead>
<tr>
<th>Existing Methods</th>
<th>Proposed Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sBFS [35]</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>x + y</td>
</tr>
<tr>
<td>Symmetric Matrices</td>
<td></td>
</tr>
<tr>
<td>ncvxqnp9</td>
<td>0.51</td>
</tr>
<tr>
<td>tuma1</td>
<td>0.42</td>
</tr>
<tr>
<td>bloweybl</td>
<td>1.00</td>
</tr>
<tr>
<td>bloweya</td>
<td>1.00</td>
</tr>
<tr>
<td>brainpc2</td>
<td>0.88</td>
</tr>
<tr>
<td>aSesindl</td>
<td>1.11</td>
</tr>
<tr>
<td>dixmaanl</td>
<td>0.33</td>
</tr>
<tr>
<td>shallow_water1</td>
<td>1.45</td>
</tr>
<tr>
<td>c-65</td>
<td>0.90</td>
</tr>
<tr>
<td>finan512</td>
<td>1.57</td>
</tr>
<tr>
<td>copter2</td>
<td>0.44</td>
</tr>
<tr>
<td>msc23052</td>
<td>0.46</td>
</tr>
<tr>
<td>Square Nonsymmetric Matrices</td>
<td></td>
</tr>
<tr>
<td>poli_large</td>
<td>1.12</td>
</tr>
<tr>
<td>powersim</td>
<td>1.02</td>
</tr>
<tr>
<td>memplus</td>
<td>0.87</td>
</tr>
<tr>
<td>Zhao1</td>
<td>0.55</td>
</tr>
<tr>
<td>mult_dcop_01</td>
<td>0.98</td>
</tr>
<tr>
<td>jan99jac120c</td>
<td>1.20</td>
</tr>
<tr>
<td>circuit_4</td>
<td>1.52</td>
</tr>
<tr>
<td>ckt11752_dc_1</td>
<td>0.79</td>
</tr>
<tr>
<td>poisson3Da</td>
<td>0.09</td>
</tr>
<tr>
<td>bcircuit</td>
<td>0.60</td>
</tr>
<tr>
<td>g7jac120</td>
<td>0.75</td>
</tr>
<tr>
<td>c40rf0100</td>
<td>0.82</td>
</tr>
<tr>
<td>Rectangular Matrices</td>
<td></td>
</tr>
<tr>
<td>lp_dif001</td>
<td>0.30</td>
</tr>
<tr>
<td>ge</td>
<td>0.40</td>
</tr>
<tr>
<td>ex3sta1</td>
<td>1.75</td>
</tr>
<tr>
<td>lp_stocfor3</td>
<td>1.74</td>
</tr>
<tr>
<td>cq9</td>
<td>0.40</td>
</tr>
<tr>
<td>pse0</td>
<td>0.45</td>
</tr>
<tr>
<td>co9</td>
<td>0.43</td>
</tr>
<tr>
<td>baxter</td>
<td>0.69</td>
</tr>
<tr>
<td>graphics</td>
<td>0.74</td>
</tr>
<tr>
<td>fome12</td>
<td>0.29</td>
</tr>
<tr>
<td>route</td>
<td>0.34</td>
</tr>
<tr>
<td>fxm4_6</td>
<td>1.54</td>
</tr>
<tr>
<td>Geometric Means</td>
<td></td>
</tr>
<tr>
<td>Symmetric</td>
<td>0.74</td>
</tr>
<tr>
<td>Nonsymmetric</td>
<td>0.74</td>
</tr>
<tr>
<td>Rectangular</td>
<td>0.60</td>
</tr>
<tr>
<td>Overall</td>
<td>0.69</td>
</tr>
</tbody>
</table>

sBFS and sRCM.

The comparison of the existing sHPRN [42] and the proposed sHPCN methods needs special attention. Both sHPRN and sHPCN belong to the single-SpMxV framework and utilize 1D matrix partitioning for row/column reordering. For the CSR-based SpMxV operation, the row-net model utilized by sHPRN corresponds to the spatial locality hypergraph model proposed by Strout et al. [35] for data reordering of unstructured mesh computations. On the
other hand, the column-net model utilized by sHP$_{CN}$ corresponds to the temporal locality hypergraph proposed by Strout et al. [35] for iteration reordering. Note that in the CSR-based SpMxV, the inner products of sparse rows with the dense input vector $x$ correspond to the iterations to be reordered. So the major difference between the sHP$_{RN}$ and sHP$_{CN}$ methods is that sHP$_{RN}$ primarily considers exploiting spatial locality and secondarily temporal locality, whereas sHP$_{CN}$ considers vice versa. This difference can also be observed by investigating the row-net and column-net models used in these two HP-based methods sHP$_{RN}$ and sHP$_{CN}$, respectively. For cutsize minimization, HP tool PaToH [10] used in sHP$_{RN}$ clusters columns with similar sparsity patterns to the same vertex parts for partial column reordering thus exploiting spatial locality, whereas PaToH used in sHP$_{CN}$ clusters rows with similar sparsity patterns to the same vertex parts for partial row reordering thus exploiting temporal locality primarily. In sHP$_{RN}$, the uncut and cut nets of a partition are used to decode the partial row reordering thus exploiting temporal locality secondarily. In sHP$_{CN}$, the uncut and cut nets of a partition are used to decode the partial column reordering thus exploiting spatial locality secondarily.

We should also note that the row-net and column-net models become equivalent for symmetric matrices. So, sHP$_{RN}$ and sHP$_{CN}$ obtain the same vertex partitions for symmetric matrices. The difference between these two methods in reordering matrices stems from the difference in the way that they decode the resultant partitions. sHP$_{RN}$ reorders the columns corresponding to the vertices in the same part of a partition successively, whereas sHP$_{CN}$ reorders the rows corresponding to the vertices in the same part of a partition successively.

As seen Table 6.4, sHP$_{CN}$ performs significantly better than sHP$_{RN}$, on the overall average. sHP$_{CN}$ performs better than sHP$_{RN}$ in all of the 36 reordering instances except a5esind1, lp_stocfactor3 and route. The significant performance gap between sHP$_{RN}$ and sHP$_{CN}$ in favor of sHP$_{CN}$ even for symmetric matrices confirm our expectation that temporal locality is more important than spatial locality in SpMxV operations that involve irregularly sparse matrices.

We introduce Table 6.5 to experimentally investigate the sensitivity of the sHP$_{RN}$ and sHP$_{CN}$ methods to the cache-line size. In the construction of the averages reported in this table, simulation results of every method are normalized with respect to those of the original ordering with the respective cache-line size. We also utilize Table 6.5 to provide fairness in the comparison of sHP$_{RN}$ and sHP$_{CN}$ methods for nonsymmetric square and rectangular matrices. Some of the nonsymmetric square and rectangular matrices may be more suitable for rowwise partitioning by the column-net model, whereas some other matrices may be more suitable for columnwise partitioning utilizing the row-net model. This is because of the differences in row and column sparsity patterns of a given nonsymmetric or rectangular matrix. Hendrickson and Kolda [22] and Ucar and Aykanat [39] provide discussions on choosing partitioning dimension depending on the individual matrix characteristics in the parallel SpMxV context. In the construction of Table 6.5, each of the sHP$_{RN}$ and sHP$_{CN}$ methods are applied on both $A$ and $A^T$ matrices and the better result is reported for the respective method on the reordering of matrix $A$. Here the performance of CSR-based SpMxV $y \leftarrow A^T x$ is assumed to simulate the performance of CSC-based $y \leftarrow A x$. Comparison of the results in Table 6.5 for the line size of 64 bytes and the average results in Table 6.4 shows that the performance of both methods increase due to the selection of better partitioning dimension (especially for rectangular matrices) while the performance gap remaining almost the same.

As seen in Table 6.5, the performance of sHP$_{RN}$ is considerably more sensitive to the cache-line size than that of sHP$_{CN}$. For nonsymmetric matrices, as the line size is increased from 8 bytes (1 word) to 512 bytes, the average normalized cache-miss count decreases from 0.70 to 0.33 in the sHP$_{RN}$ method, whereas it decreases from 0.53 to 0.30 in the sHP$_{CN}$
method. Similarly, for rectangular matrices, the average normalized cache-miss count decreases from 0.62 to 0.23 in the $sHP_{RN}$ method, whereas it decreases from 0.52 to 0.23 in the $sHP_{CN}$ method. As seen in Table 6.5, the performance of these two methods become very close for the largest line size of 512 bytes (64 words). This experimental finding conforms to our expectation that $sHP_{RN}$ exploits spatial locality better than $sHP_{CN}$, whereas $sHP_{CN}$ exploits temporal locality better than $sHP_{RN}$.

We proceed with the relative performance comparison of the proposed methods. As seen in Table 6.4, on the average, 2D-partitioning-based methods $sHP_{eRCN}$ and $mHP_{RCN}$ perform better than the 1D-partitioning-based method $sHP_{CN}$. The performance gap between the 2D and 1D methods is considerably higher in reordering symmetric matrices in favor of 2D methods. This experimental finding may be attributed to the relatively restricted search space of the column-net model (as well as the row-net model) in 1D partitioning of symmetric matrices. The relative performance comparison of 2D methods shows that $sHP_{eRCN}$ and $mHP_{RCN}$ display comparable performance. $mHP_{RCN}$ performs better than $sHP_{eRCN}$ in 18 out of 36 reordering instances, whereas $sHP_{eRCN}$ performs better in 16 reordering instances. On the overall average, $mHP_{RCN}$ performs 4.3% better than $sHP_{eRCN}$ in terms of cache misses due to the access of $x$-vector and $y$-vector entries.

As seen in Table 6.4, $mHP_{RCN}$ incurs significantly less $x$-vector entry misses than $sHP_{eRCN}$ on the overall average. This is expected because the multiple-SpMxV framework utilized in $mHP_{RCN}$ enables better exploitation of temporal locality in accessing $x$-vector entries. However, the increase in the $y$-vector entry misses, which is introduced by the multiple-SpMxV framework, does not amortize in some of the reordering instances. As expected, $mHP_{RCN}$ performs better than $sHP_{eRCN}$ in the reordering of matrices that contain dense rows. For example, in the reordering of symmetric matrices $a5esindl$, $bloweya$, and $brainpc2$, which respectively contain dense rows with 9993, 10001, and 13799 nonzeros, $mHP_{RCN}$ performs significantly better than $sHP_{eRCN}$. Similar experimental findings can be observed in Table 6.4 for the following matrices that contain dense rows: square nonsymmetric matrices $circuit_4$, $ckt11752_dc_1$, $mult_dcop_01$ and rectangular matrices $baxter$, $co9$, $cq9$ and $route$. Although $shallow_water$ and $psse0$ do not contain dense rows (maximum number of nonzeros in a row is only 4 in both matrices), $mHP_{RCN}$ performs significantly better than $sHP_{eRCN}$ in reordering these two matrices. $mHP_{RCN}$ incurs significantly less cache misses in the access of $x$-vector entries while incurring very small number of additional cache misses due to the access of $y$-vector entries. The reason behind the latter finding is the very small number of shared rows among the $A^k$ matrices obtained by $mHP_{RCN}$ in splitting these two matrices. For example, in one of the splittings generated by $mHP_{RCN}$, among the 81920 rows of $shallow_water$, only 785 rows are shared and all of them are shared between only two distinct $A^k$ matrices.

<table>
<thead>
<tr>
<th>Line Size (Byte)</th>
<th>Nonsymmetric $sHP_{RN}$</th>
<th>$sHP_{CN}$</th>
<th>Rectangular $sHP_{RN}$</th>
<th>$sHP_{CN}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>0.70</td>
<td>0.53</td>
<td>0.62</td>
<td>0.52</td>
</tr>
<tr>
<td>16</td>
<td>0.68</td>
<td>0.49</td>
<td>0.58</td>
<td>0.47</td>
</tr>
<tr>
<td>32</td>
<td>0.65</td>
<td>0.45</td>
<td>0.52</td>
<td>0.41</td>
</tr>
<tr>
<td>64</td>
<td>0.61</td>
<td>0.41</td>
<td>0.44</td>
<td>0.34</td>
</tr>
<tr>
<td>128</td>
<td>0.57</td>
<td>0.38</td>
<td>0.39</td>
<td>0.28</td>
</tr>
<tr>
<td>256</td>
<td>0.52</td>
<td>0.33</td>
<td>0.36</td>
<td>0.23</td>
</tr>
<tr>
<td>512</td>
<td>0.33</td>
<td>0.30</td>
<td>0.23</td>
<td>0.23</td>
</tr>
</tbody>
</table>

TABLE 6.5
Sensitivity of $sHP_{RN}$ [42] and $sHP_{CN}$ to cache-line size
Table 6.6 shows the comparison of the sensitivities of the proposed methods $sH_{PCN}$, $sH_{RCN}$, and $mH_{RCN}$ to the cache-line size. In the construction of the averages reported in this table, simulation results of every method are normalized with respect to those of the original ordering with the respective cache-line size. In terms of cache misses due to access of $x$-vector entries, the performance of each method compared to the original ordering increases with increasing cache-line size. However, in terms of cache misses due to access of $y$-vector entries, the performance of $mH_{RCN}$ compared to the original ordering decreases with increasing cache-line size. So, with increasing cache-line size, the performance gap between $mH_{RCN}$ and the other two methods $sH_{PCN}$ and $sH_{RCN}$ increases so that $sH_{RCN}$ performs better than $mH_{RCN}$ for larger cache-line sizes of 256 and 512 bytes. This experimental finding can be attributed to the deficiency of the multiple-SpMxV framework in exploiting spatial locality in accessing $y$-vector entries. We believe that models and methods need to be investigated for intelligent global row ordering to overcome this deficiency of the multiple-SpMxV framework.

We introduce Table 6.7 to display the sensitivities (as overall averages) of the top-down HP-based reordering methods to the part-size threshold ($W_{max}$) used in terminating the RB process. The performance of each method increases with decreasing part-size threshold until the part-size threshold becomes equal to the cache size. For each method, the rate of performance increase begins to decrease as the part-size threshold becomes closer to the cache size. The performance of each method remains almost the same with decreasing part-size threshold below the cache size except $mH_{RCN}$. The slight decrease in the performance of $mH_{RCN}$ with decreasing part-size threshold below the cache size can be attributed to the increase in the number of $y$ misses with increasing number of $A^k$ matrices because of the deficiency of the multiple-SpMxV framework in exploiting spatial locality in accessing $y$-vector entries. These experimental findings show the validity of Theorems 1, 2, and 3 for the effectiveness of the proposed $sH_{PCN}$, $sH_{RCN}$, and $mH_{RCN}$ methods, respectively. Although the proposed HP-based methods are cache-size aware methods, they can easily be modified to become cache oblivious methods by continuing the RB process until the parts become sufficiently small or the qualities of the bipartitions drop below a predetermined threshold.

Table 6.8 displays the running times of the existing and proposed methods on a PC equipped with quad 2.1 GHz 6-core AMD Opteron processors with 128 GB memory. For each test matrix $A$, the running times of all methods are normalized with respect to that of the SpMxV operation $y \leftarrow Ax$ using the unordered $A$ matrix and geometric averages of these normalized values are displayed in the table. As seen in the table, top-down HP-based methods are significantly slower than the bottom-up reordering algorithms $sBFS$ and $sRCM$. As also seen in the table, the 2D-partitioning-based methods are considerably slower than the
TABLE 6.7
Sensitivity of HP-based reordering methods to the part-size threshold (cache size = 64 KB)

<table>
<thead>
<tr>
<th>Part Size (KB)</th>
<th>1D Partitioning</th>
<th>2D Partitioning</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>sHP$_{RN}$ [42]</td>
<td>sHP$_{CN}$</td>
</tr>
<tr>
<td>512</td>
<td>x</td>
<td>x + y</td>
</tr>
<tr>
<td>256</td>
<td>0.68</td>
<td>0.61</td>
</tr>
<tr>
<td>126</td>
<td>0.62</td>
<td>0.51</td>
</tr>
<tr>
<td>64</td>
<td>0.60</td>
<td>0.45</td>
</tr>
<tr>
<td>32</td>
<td>0.59</td>
<td>0.43</td>
</tr>
<tr>
<td>16</td>
<td>0.60</td>
<td>0.43</td>
</tr>
<tr>
<td>8</td>
<td>0.61</td>
<td>0.43</td>
</tr>
</tbody>
</table>

TABLE 6.8
Running times of the reordering methods in terms of SpMxV times

<table>
<thead>
<tr>
<th></th>
<th>Single SpMxV</th>
<th>Proposed Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sBFS [35]</td>
<td>sRCM [25]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sHP$_{RN}$ [42]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(1D Part.)</td>
</tr>
<tr>
<td>Modified</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>Symmetric</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nonsymmetric</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>Rectangular</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>Overall</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sHP$_{CN}$</td>
<td>sHP$_{eRCN}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(1D Part.)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(2D Part.)</td>
</tr>
<tr>
<td></td>
<td>455</td>
<td>455</td>
</tr>
<tr>
<td></td>
<td>497</td>
<td>428</td>
</tr>
<tr>
<td></td>
<td>396</td>
<td>359</td>
</tr>
<tr>
<td></td>
<td>447</td>
<td>412</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>mHP$_{RCN}$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(2D Partitioning)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1,047</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1,305</td>
<td></td>
</tr>
<tr>
<td></td>
<td>878</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1,063</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1,622</td>
<td></td>
</tr>
</tbody>
</table>

1D-partitioning-based methods as expected. The running time difference between the 1D- and 2D-partitioning-based methods becomes higher with increasing matrix density in favor of 1D methods. The running times of two 1D-partitioning-based methods sHP$_{RN}$ and sHP$_{CN}$ are comparable as expected. There exists a considerable difference in the running times of two 2D-partitioning-based methods sHP$_{eRCN}$ and mHP$_{RCN}$ in favor of sHP$_{eRCN}$. This is because of the removal of the vertices connected by the cut row nets in the enhanced row-column-net model used in sHP$_{eRCN}$ and the TSP ordering performed as a postprocessing in mHP$_{RCN}$. The relatively high preprocessing times of the top-down HP-methods are expected to amortize for large number of repeated SpMxV computations that involve A matrix with the same sparsity pattern.

7. Conclusion. Single- and multiple-SpMxV frameworks were investigated for exploiting cache locality in SpMxV computations that involve irregularly sparse matrices. For the single-SpMxV framework, two cache-size-aware top-down row/column-reordering methods based on 1D and 2D sparse matrix partitioning were proposed by utilizing the column-net and enhancing the row-column-net hypergraph models of sparse matrices. The multiple-SpMxV framework requires splitting a given matrix into a sum of multiple nonzero-disjoint matrices so that the SpMxV operation is computed as a sequence of multiple input- and output-dependent SpMxV operations. For the multiple-SpMxV framework, a cache-size aware top-down matrix splitting method based on 2D matrix partitioning was proposed by utilizing the row-column-net hypergraph model of sparse matrices. The proposed hypergraph-partitioning (HP) based methods in the single-SpMxV framework primarily aim at exploiting temporal locality in the access of input-vector entries and the proposed HP-based method in the multiple-SpMxV framework primarily aims at exploiting temporal locality in the access of both input- and output-vector entries. The performance and validity of the proposed methods were tested against three state-of-the-art methods on a wide range of test matrices. Experimental results show that the proposed methods can effectively reduce cache misses in SpMxV computations.
Experimental results confirm our expectation that temporal locality is more important than spatial locality (for practical line sizes) in SpMxV operations that involve irregularly sparse matrices. The multiple-SpMxV framework is found to be very promising, however it is suffers from the deficiency in exploiting spatial locality in accessing output-vector entries. Models and methods need to be investigated for intelligent global row reordering to overcome this deficiency of the multiple-SpMxV framework.

The sensitivity analysis conducted for the proposed top-down matrix reordering and splitting methods to the part-size threshold used in terminating the recursive bipartitioning (RB) process conforms the validity of the theoretical findings presented in this work. Although the proposed HP-based methods are cache-size aware, this sensitivity analysis show that they can easily be modified to become cache oblivious by continuing the RB process until the parts become sufficiently small or the qualities of the bipartitions drop below a predetermined threshold.

REFERENCES

[18] G. HAASE, M. LIEBMAN, AND G. PLANK, A hilbert-order multiplication scheme for unstructured sparse


