Exploiting Locality in Sparse Matrix-Matrix Multiplication on Many-Core Architectures

Kadir Akbudak and Cevdet Aykanat

Abstract—Exploiting spatial and temporal localities is investigated for efficient row-by-row parallelization of general sparse matrix-matrix multiplication (SpGEMM) operation of the form $C = A\,B$ on many-core architectures. Hypergraph and bipartite graph models are proposed for 1D rowwise partitioning of matrix A to evenly partition the work across threads with the objective of reducing the number of B-matrix words to be transferred from the memory and between different caches. A hypergraph model is proposed for B-matrix column reordering to exploit spatial locality in accessing entries of thread-private temporary arrays, which are used to accumulate results for C-matrix rows. A similarity graph model is proposed for B-matrix row reordering to increase temporal reuse of these accumulation array entries. The proposed models and methods are tested on a wide range of sparse matrices from real applications and the experiments were carried on a 60-core Intel Xeon Phi processor, as well as a two-socket Xeon processor. Results show the validity of the models and methods proposed for enhancing the locality in parallel SpGEMM operations.

Index Terms—Data locality, sparse matrix, sparse matrix-matrix multiplication, SpGEMM, computational hypergraph model, hypergraph partitioning, hypergraph clustering, graph model, bipartite graph model, graph partitioning, graph clustering, many-core architecture, Intel Xeon Phi

1 Introduction

1.1 Applications Involving SpGEMM

General sparse matrix-matrix multiplication (SpGEMM) of the form $C = A\,B$ is an important kernel in various applications such as molecular dynamics simulations [1], [2], [3], [4], [5], [6], [7], finite element simulations based on domain decomposition [8], [9], linear programming (LP) [10], [11], [12], multigrid interpolation and restriction [13], breadth-first search from multiple source vertices [14], finding all-pair shortest-paths [15], similarity join [16], data summarization [17], and item-to-item collaborative filtering in recommendation systems [18].

Some of the above-mentioned applications require repeated SpGEMM operations which involve matrices with same sparsity patterns but differing numerical values: Solution of LP problems through iterative interior point methods that use normal equations constitutes a typical example of such applications. These methods solve positive-definite linear systems of the form $(AD^2A^T)\,x = b$ at each iteration, where A is the original constraint matrix and D is a positive diagonal matrix which varies with each iteration. Direct solvers [10], [11], [12] that utilize Cholesky factorization as well as iterative solvers [11] that utilize preconditioners require explicitly forming the coefficient matrix at each iteration through the SpGEMM computation $C = A\,B$, where the sparsity patterns of both A and $B = D^2A^T$ remain the same throughout the iterations.

Similarity join and collaborative filtering applications may also involve repeated SpGEMM operations. In the self similarity join application that involves $C = AA$ and in the similarity join of two different sparse data sets that involve $C = AB$, different weightings can be defined on the relative importance of features that will incur repeated SpGEMM operations of the forms $C = AWA$ and $C = AWB$ for different W matrices. Here, W is a diagonal matrix that contains values for relative ranking of the features in the data sets. In item-to-item collaborative filtering in recommendation systems, given a similar-items table, which is A, SpGEMM is performed in order to find items similar to each of the user's preferences, which are stored in B, and then the system aggregates those items and recommends the most popular or correlated items. This application may incur repeated SpGEMM operations of the form of $C = AWB$, where W is a diagonal matrix that adjusts importance of items in filtering.

In this work, we propose matrix partitioning and row/column reordering schemes for exploiting temporal and spatial localities in row-by-row parallelization of SpGEMM on many-core architectures. All of the above-mentioned applications will benefit from these matrix partitioning and reordering methods, whereas the preprocessing overhead due to these intelligent partitioning operations will amortize especially for the applications that involve repeated SpGEMM operations.

1.2 Parallel SpGEMM Algorithms

Parallel SpGEMM schemes can be broadly classified into four categories depending on the following 1D GEMM formulations [19]: Outer product, inner product, column-by-column, and row-by-row. In the outer-product parallelization, an atomic task is defined as the outer product of column i of A with row i of B, so that each thread is held responsible for computing one or more outer products. The outer-product formulation is reported to lead to scalable

• *The authors are with Computer Engineering Department, Bilkent University, Ankara 06800, Turkey. E-mail: {kadir, aykanat}@cs.bilkent.edu.tr.*

Manuscript received 22 Dec. 2015; revised 15 Jan. 2017; accepted 16 Jan. 2017. Date of publication 23 Jan. 2017; date of current version 14 July 2017. Recommended for acceptance by T. Hoefler.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2017.2656893

parallelization on distributed-memory architectures [20], [21], [22]. However, this SpGEMM scheme requires sparse accumulation operations for obtaining the output matrix from the partial results of concurrent thread computations on shared-memory architectures. These operations necessitate complex indexing schemes to efficiently identify the contributions to the same matrix entries by different threads. Although there exists an indexing scheme proposed for outer-product kernel on distributed-memory architectures [20], this scheme does not seem to be viable for many-core architectures due to the overhead of many concurrent writes.

In the inner-product parallelization, an atomic task is defined as the inner product of row i of A with column j of B, so that each thread is held responsible for computing one or more inner products. This scheme requires merging two sparse vectors which is not efficient because of large number of element accesses per floating-point operation (Flop) [23]. Therefore, this scheme is also not viable for current many-core architectures with deep cache hierarchies.

In the column-by-column parallelization, an atomic task is defined as the post-multiply of the whole A matrix with a column of B, so that each thread is held responsible for computing one or more post-multiplies. In the row-by-row parallelization, an atomic task is defined as the pre-multiply of a row of A with the whole B matrix, so that each thread is held responsible for computing one or more pre-multiplies. Both schemes share the nice property of not requiring concurrent writes as opposed to the outer-product formulation. The former and latter schemes can complete the computation of a column and a row of the output matrix at a time, respectively. Both schemes also share the nice property of performing all multiplications related with a nonzero of one of the two input matrices as opposed to the inner-product formulation. So, both of these two schemes are found to be viable for shared-memory parallelism.

1.3 Related Work

There exist several recent works on SpGEMM parallelization on distributed-memory architectures [7], [20], [21], [22], [24], [25], [26], [27]. Here, we discuss the related work on SpGEMM parallelization on shared-memory architectures according to the above-mentioned categorization.

Sulatcyke and Ghose [28] examine the impact of data reuse on the SpGEMM performance. They consider outer-product and row-by-row formulations and report that the row-by-row formulation is more amenable for data reuse opportunities. They also report that the row-by-row parallelization does not need any synchronization among threads so it is more eligible for shared-memory parallelization.

There are successful GPU libraries, cuSPARSE [29] and Cusp [30], which perform SpGEMM. cuSPARSE uses row-by-row formulation as its top-level parallelism. A hash table is used for accumulating partial results for each row of the output matrix. In Cusp [31], elements of input matrices are accessed via row-by-row scheme, but the partial results for the output matrix are stored in an intermediate list consisting of row, column, and value tuples. This list is sorted and duplicate column indices are compressed to compute the final output matrix. The SpGEMM operation in Cusp is further enhanced by the works [32] and [33].

Matam et al. [34] investigate row-by-row and outer-product parallelization schemes, which utilize blocking to decrease the size of the temporary accumulation array used to accumulate the result of each pre-multiply and outer product, respectively. They report that the row-by-row parallelization performs better than the outer-product parallelization. A similar blocking approach is also utilized in [35] to decrease the size of the temporary accumulation array.

Liu and Vinter [36] use row-by-row parallelization. They utilize progressive allocation of C matrix for memory efficiency, computing final C-matrix rows via merging partial results, and partitioning rows of C with respect to their nonzero density for better load balancing.

Gremse et al. [37] uses row-by-row parallelization. Instead of using accumulation array, W (subwarp size) results are merged at a time via utilizing a warp reduction method, which uses only hardware registers.

Siegel et al. [38] use inner-product formulation for coarse-grained parallelism among tasks and row-by-row formulation for fine-grained parallelism within a task. Task sharing approach is used in [38] for dynamic load balancing among nodes and GPUs of a node.

The above-mentioned works on SpGEMM do not exploit sparsity patterns of input or output matrices for efficient parallelization via reducing cache misses. Although there exist models and methods that utilize sparsity pattern for exploiting locality in sparse matrix-vector multiplication [39], [40], [41], the literature lacks locality exploiting methods for SpGEMM operations. In this work, we propose models and matrix partitioning/reordering methods, which utilize matrix sparsity pattern in order to exploit data locality in parallel SpGEMM operation.

1.4 Contributions

We propose a hypergraph model and a bipartite graph model for encapsulating computational and B-matrix transfer requirements of thread-level row-by-row parallelization of SpGEMM operation that is based on rowwise A-matrix partitioning. We show that the minimization objectives of partitioning the hypergraph and bipartite graph models relate to reducing data transfers from memory or another cache. We devise fast bottom-up clustering methods utilizing both proposed hypergraph and bipartite graph models in order to decrease the preprocessing overhead.

We also investigate how to exploit locality in accumulation of partial results for C-matrix rows. For this purpose, we propose a spatial hypergraph model and a temporal graph model for reordering B-matrix columns and rows, respectively. For the same purpose, we also show how to encode an existing A-matrix partition as a row/column reordering of the B matrix for the special cases of $C = AA$ and $C = AA^T$.

The validity of the proposed matrix partitioning and row/column reordering methods are extensively experimented on real SpGEMM instances using many-core architectures Intel Xeon Phi and Xeon.

We should note that the column-by-column and row-by-row parallelization schemes, both of which are found to be viable for shared-memory parallelization, can be considered as dual of each other. So, the discussion for the column-by-column parallelization directly follows the discussion given for the row-by-row parallelization.

The rest of the paper is organized as follows: The row-by-row parallelization of SpGEMM and its data locality properties are given in Sections 2 and 3, respectively. The hypergraph and bipartite graph models for exploiting temporal locality in accessing B-matrix rows are introduced in Section 4. In Section 5, we present the models and methods for exploiting spatial and temporal localities in accessing entries of accumulation arrays. The experimental results are presented in Section 6. Finally, we conclude the paper in Section 7. The supplemental material contains Appendix Sections A–D, Algorithm A.1, Tables A.1–A.4, and Fig. A.1, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2017.2656893.

2 The Parallel Algorithm

The row-by-row parallelization is based on one-dimensional conformable rowwise partitioning of the input matrix A and the output matrix C as follows:

$$\hat{A} = PA = \begin{bmatrix} A_1 \\ A_2 \\ \vdots \\ A_K \end{bmatrix} \quad \text{and} \quad \hat{C} = PC = \begin{bmatrix} C_1 \\ C_2 \\ \vdots \\ C_K \end{bmatrix}. \tag{1}$$

In (1), submatrices A_k and C_k denote the kth row slices of the reordered A and C matrices, respectively, where K denotes the number of parts. Here, P denotes the permutation matrix obtained from partitioning. The use of the same permutation matrix for row reorderings of A and C shows that a rowwise partition on A induces a conformable rowwise partition on C. The reordered $\hat{A}$ and $\hat{C}$ will be referred to as A and C.

Algorithm 1. The Thread-Level Row-by-Row Parallelization of SpGEMM Operation for Numeric Multiply

Require: A_k, B, **and** C_k **matrices in CSR format**
1: //Each iteration performs $C_k = A_k B$
2: **for** $k = 1$ **to** K **in parallel do**
3: //Each iteration performs $c_{i,*} = a_{i,*} B$
4: **for each** row $a_{i,*}$ of A_k **do**
5: //Init thread-private accumulation array
6: **for each** nonzero $c_{i,j}$ in row $c_{i,*}$ **do**
7: $acc[j] = 0.0$
8: **for each** nonzero $a_{i,j}$ in row $a_{i,*}$ **do**
9: $acc = acc + a_{i,j}\ b_{j,*}$
10: //Gather dense array to sparse storage
11: **for each** nonzero $c_{i,j}$ in row $c_{i,*}$ **do**
12: $c_{i,j} = acc[j]$

The input and output data partitioning given in (1) leads to the row-by-row algorithm as shown in Algorithm 1, where the output matrix C is computed as,

$$C_k = A_k B, \quad \text{for} \quad k = 1, 2, \ldots, K. \tag{2}$$

Each submatrix-matrix multiply $C_k = A_k B$ is assigned to a thread that will be executed by an individual core as also depicted by the "for ... in parallel do" construct in line 2 of Algorithm 1. This algorithm uses CSR (compressed storage by rows) scheme for storing both input matrices and the output matrix. In lines 6–12, row i of A_k ($a_{i,*}$) is pre-multiplied by B and the result of this multiplication is written to row i of C ($c_{i,*}$). Each thread allocates a private accumulation array acc (i.e., a simple 1D dense array) of size N for efficient computation of pre-multiplies, where N is the number of columns of B. In line 6, acc is initialized to zero and then a pre-multiply is performed by the *for-loop* starting in line 8. In line 11, the final values for $c_{i,*}$ are gathered from acc and stored in compressed row storage of C. A symbolic SpGEMM operation is performed prior to the numeric multiplication for identifying the column indices of nonzeros in row $c_{i,*}$. The pseudo-code of symbolic multiplication is given in Algorithm A.1 of Appendix A, available in the online supplemental material.

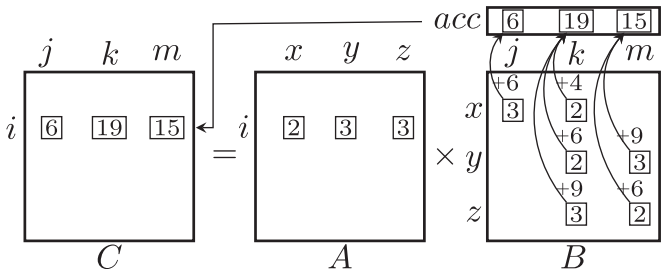


Fig. 1. Row-by-row formulation based SpGEMM.

Fig. 1 shows an SpGEMM instance to demonstrate the atomic task of computing row i of the C matrix according to row-by-row formulation. As seen in the figure, row i of A contains three nonzeros at columns x, y, and z. Each of B-matrix rows x, y, and z contains two nonzeros and these six nonzeros are spread over three columns j, k, and m. Thus, they incur the addition of the results of six scalar multiply-and-add operations to the jth, kth, and mth entries of the acc array. Each of three B-matrix rows has a nonzero at column k, which incurs the addition of the results of three scalar multiply-and-add operations to the same entry of the acc array.

3 Data Locality

We present spatial and temporal locality characteristics of the row-by-row parallelization given in Algorithm 1. Discussions presented here are valid for private caches and also for processors with private and shared caches existing at different levels of the memory hiearchy.

Spatial locality in accessing A-matrix nonzeros is simply achieved by storing nonzeros of each row consecutively using CSR and processing nonzeros of A consecutively. Temporal locality in accessing A-matrix nonzeros is not feasible since each A-matrix nonzero is accessed once.

Spatial locality in accessing B-matrix nonzeros is partially achieved by storing nonzeros of each row consecutively using CSR. However, rows of B are not processed consecutively and the processing order is determined by sparsity patterns and processing order of A-matrix rows. The performance improvement due to exploiting spatial locality is expected to be negligible because at most one extra cache miss can be avoided for each B-matrix row. So this locality is not considered in the paper.

Temporal locality in accessing B-matrix nonzeros (together with their indices) is feasible since B-matrix nonzeros are accessed multiple times. Each nonzero in row x of B is accessed $nnz(a_{*,x})$ times. Here, $nnz(\cdot)$ denotes the

number of nonzeros in a row, a column or the whole of the matrix. Temporal locality in accessing B-matrix nonzeros can be exploited through clustering A-matrix rows that are similar in terms of the number of nonzeros of required B-matrix rows, where each cluster constitutes a row-slice of the reordered A matrix. Reuse of B-matrix nonzeros can be achieved at a coarse level of reuse of B-matrix rows.

The access pattern to the accumulation array is determined by the sparsity pattern of B as well as the access order of B-matrix rows, which is induced by the processing order of A-matrix rows. Spatial locality in accessing the accumulation array entries can be exploited through reordering B-matrix columns with similar sparsity patterns nearby. Temporal locality in accessing the accumulation array entries can be exploited through reordering of B-matrix rows that are accessed within the same coarse-grain thread-level task with similar sparsity patterns nearby. This is because B-matrix rows with similar sparsity patterns are likely to access the same accumulation array entries during the execution of the coarse-grain task.

4 A-matrix Partitioning for Temporal Locality in Accessing B Matrix

For exploiting temporal locality in accessing B-matrix rows, we propose and discuss two models for conformable row-wise partitioning of A and C matrices.

4.1 Hypergraph Model $\mathcal{H}_A^T$

A given SpGEMM computation $C = A\,B$ is represented as a temporal hypergraph $\mathcal{H}_A^T(A, \{nnz(b_{x,*})\}_x) = (\mathcal{V}, \mathcal{N})$. Here, $\mathcal{H}_A^T(A, \{nnz(b_{x,*})\}_x)$ refers to the fact that the sparsity pattern of matrix A determines the topology of the proposed hypergraph model, whereas the nonzero counts of B-matrix rows determine vertex and net weights. The superscript "T" denotes temporal locality.

In $\mathcal{H}_A^T$, $\mathcal{V}$ contains a vertex v_i for each row i of A so that it represents the atomic task

$$c_{i,*} = a_{i,*}B, \tag{3}$$

of pre-multiplying row i ($a_{i,*}$) of A with the B matrix to compute row i ($c_{i,*}$) of C. Note that this atomic task definition effectively means that vertex v_i also represents row i of the C matrix according to owner computes rule. Hence, we associate vertex v_i with a weight $w(v_i)$ proportional to the computational load of this pre-multiply in terms of scalar multiply-and-add operations. That is,

$$w(v_i) = \sum_{x \in cols(a_{i,*})} nnz(b_{x,*}), \tag{4}$$

where $cols(a_{i,*})$ denotes the column indices of the nonzeros in row i of A.

$\mathcal{N}$ contains a net (hyperedge) n_x for each row x of B. Net n_x connects vertices corresponding to rows that have nonzeros at column x of A. So, the vertices connected by n_x correspond to the atomic tasks that need to access row x of B. Hence, we associate net n_x with a weight $w(n_x)$ proportional to the cost of transferring B-matrix row x from memory or another cache, that is,

$$w(n_x) = nnz(b_{x,*}). \tag{5}$$

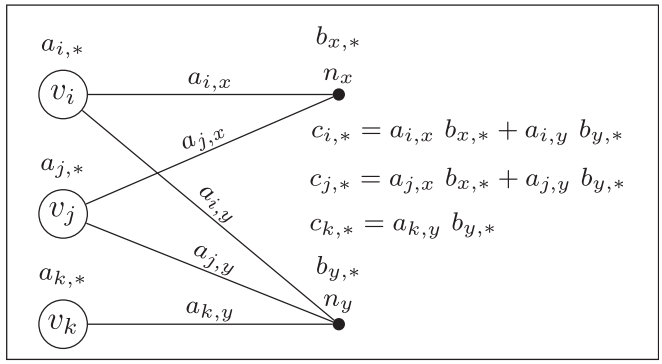


Fig. 2. Hypergraph model for exploiting temporal locality in accessing B-matrix rows during thread-level row-by-row parallelization of SpGEMM operation.

Fig. 2 illustrates the input and output dependency of the proposed hypergraph model. In this figure, as well as in Fig. 4, circles represent vertices, whereas dots represent nets. As seen in Fig. 2, n_x connects vertices v_i and v_j, whereas n_y connects vertices v_i, v_j, and v_k. So net n_x encodes the need of accessing B-matrix row x for the atomic tasks of computing rows i and j of C, whereas net n_y encodes the need of accessing B-matrix row y for the atomic tasks of computing rows i, j, and k of C. Vertex v_i connected by both n_x and n_y shows that the atomic task of computing row i of C needs to access both rows x and y of B, which in turn shows the need for an accumulation operation depending on the sparsity patterns of rows x and y of B.

Consider a K-way partition $\Pi(\mathcal{V}) = \{\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_K\}$ of vertices of $\mathcal{H}_A^T$, where parts are mutually exclusive and exhaustive. In $\Pi(\mathcal{V})$, the weight $W(\mathcal{V}_k)$ of a part k is defined as the sum of the weights of the vertices in part k. That is, $W(\mathcal{V}_k) = \sum_{v \in \mathcal{V}_k} w(v)$.

A K-way partition $\Pi(\mathcal{V})$ is decoded in such a way that the set of atomic tasks (fine-grain tasks) corresponding to the vertices in a part of $\Pi(\mathcal{V})$ constitutes a coarse-grain task to be executed by a distinct thread. That is, vertex part $\mathcal{V}_k$ denotes the submatrix-matrix multiply $C_k = A_k B$ to be executed by a distinct thread, where A_k and C_k are the submatrices respectively formed by A-matrix and C-matrix rows that are represented by the vertices in $\mathcal{V}_k$. Without loss of generality, we assume that each core executes only one distinct thread. K is selected such that the storage size of the B-matrix rows required by each submatrix-matrix multiply $C_k = A_k B$ together with the storage sizes of C_k and A_k matrices are below the size of the cache of a single core. Thus, the working set of each iteration of the *for-loop* starting at line 2 of Algorithm 1 fits into the cache of a single core due to thread-level coarse-grain task definition induced by the partitioning.

The weight of a part corresponds to the computational load of a thread-level coarse-grain task. So, the partitioning constraint of maintaining balance on the part weights corresponds to maintaining balance on the computational loads of thread-level coarse-grain tasks. This constraint corresponds to reducing overall execution time for arbitrary coarse-grain task scheduling.

In a K-way partition $\Pi(\mathcal{V})$ of $\mathcal{H}_A^T$, a net n_x is said to connect a part if n_x connects at least one vertex in that part. The connectivity $con(n_x)$ of n_x is the set of parts connected by n_x. The cutsize of $\Pi(\mathcal{V})$ is defined as

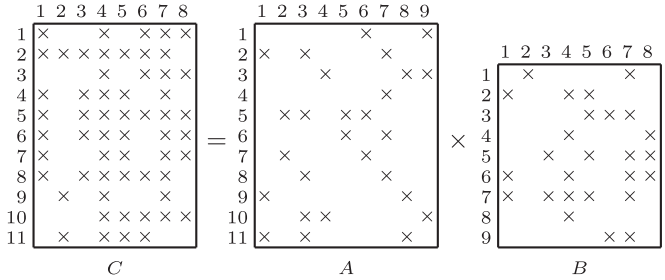
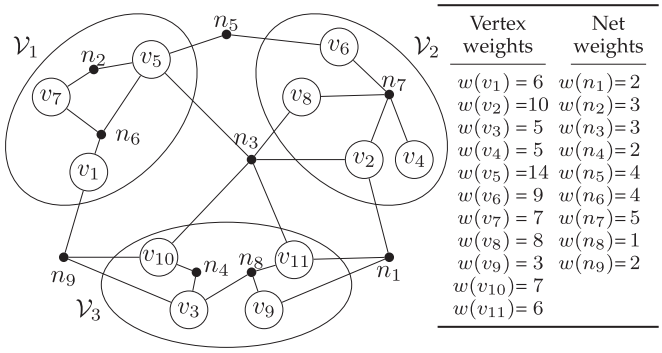


Fig. 3. A sample SpGEMM computation.



Vertex weights	Net weights
$w(v_1) = 6$	$w(n_1) = 2$
$w(v_2) = 10$	$w(n_2) = 3$
$w(v_3) = 5$	$w(n_3) = 3$
$w(v_4) = 5$	$w(n_4) = 2$
$w(v_5) = 14$	$w(n_5) = 4$
$w(v_6) = 9$	$w(n_6) = 4$
$w(v_7) = 7$	$w(n_7) = 5$
$w(v_8) = 8$	$w(n_8) = 1$
$w(v_9) = 3$	$w(n_9) = 2$
$w(v_{10}) = 7$	
$w(v_{11}) = 6$	

Fig. 4. Hypergraph model for the sample SpGEMM instance given in Fig. 4 and its 3-way partition.

$$cutsize(\Pi(\mathcal{V})) = \sum_{n_x \in \mathcal{N}} w(n_x)|con(n_x)|. \quad (6)$$

Here, we present the following discussion to show that the cutsize of a given partition $\Pi(\mathcal{V})$ of $\mathcal{H}_A^T$ is equal to the number of B-matrix words to be transferred from the memory and between different caches under the following assumptions: single-word line, fully associative cache, and single thread per core.

Consider a net n_x with connectivity $con(n_x)$ in Π. Let $\mathcal{T}_x$ denote the set of $|\mathcal{T}_x| = |con(n_x)|$ threads that will execute the coarse-grain tasks corresponding to the vertex parts in $con(n_x)$ on different cores of the system. Without loss of generality, let t be the first thread to be executed in $\mathcal{T}_x$. Thread t will transfer the B-matrix row x from the memory, whereas the other threads in $\mathcal{T}_x$ will transfer B-matrix row x either from the memory or from the cache of another core. Hence the amount of data transfer due to accessing B-matrix row x, which is represented by net n_x, will be equal to $w(n_x)|con(n_x)|$ words. Here, the costs of transferring from memory and another cache are assumed to be equal based on the results reported in [43]. Thus $cutsize(\Pi)$ according to (6) will show the total amount of data transferred due to accessing B-matrix rows under the above-mentioned assumptions. It is clear that the amount of data transfer will reduce for the general cases of multi-word line and multiple threads per core. On the other hand, the amount of data transfer will increase for the general case of set associative caches because of the possibility of the conflict misses, whereas fully associative caches will incur only capacity misses. So, for the general case, the partitioning objective of minimizing the cutsize relates to minimizing the amount of data transfer due to accessing B-matrix rows.

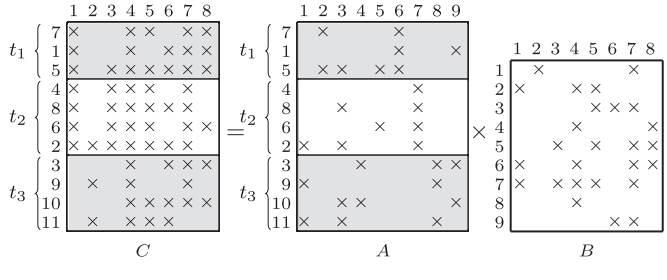


Fig. 5. Matrices A and C partitioned according to the partition $\Pi(\mathcal{V})$ of $\mathcal{H}_A^T$ given in Fig. 4 and matrix B.

In Appendix B, available in the online supplemental material, we also show that the objective of minimizing the cutsize also relates to maximizing B-matrix row reuse.

Fig. 3 shows a sample SpGEMM computation $C = A\,B$, where A and B are 11 by 9 and 9 by 8 matrices with 27 and 26 nonzeros, respectively, and C is an 11 by 8 matrix with 57 nonzeros. Fig. 4 shows the hypergraph model $\mathcal{H}_A^T$ that represents the sample SpGEMM computation instance given in Fig. 3. As seen in Fig. 4, $\mathcal{H}_A^T$ contains 11 vertices, 9 nets, and 27 pins. Note that a connection between a net and a vertex is said to be *pin* of that net. As also seen in the figure, $w(v_5) = 14$ since row 5 of A contains nonzeros at columns 2, 3, 5, and 6, and $nnz(b_{2,*}) + nnz(b_{3,*}) + nnz(b_{5,*}) + nnz(b_{6,*}) = 3 + 3 + 4 + 4 = 14$.

Fig. 4 also shows a 3-way partition $\Pi(\mathcal{V})$ of $\mathcal{H}_A^T$, and Fig. 5 shows the 3-way partition of the sample input and output matrices induced by this $\Pi(\mathcal{V})$. In $\Pi(\mathcal{V})$, $W(\mathcal{V}_1) = w(v_1) + w(v_5) + w(v_7) = 6 + 14 + 7 = 27$. Similarly, $W(\mathcal{V}_2) = 22$ and $W(\mathcal{V}_3) = 31$. So $\Pi(\mathcal{V})$ incurs 16 percent load imbalance.

In the 3-way partition $\Pi(\mathcal{V})$ given in Fig. 4, there are four cut nets n_1, n_3, n_5, and n_9, whereas the remaining five nets are uncut. Note that a net is said to be *cut* if it connects more than one part, *uncut* otherwise. As seen in the figure, net n_3 connects all vertex parts, and hence $|con(n_3)| = 3$. Consequently, cut net n_3 incurs $w(n_3)|con(n_3)| = 3 \cdot 3 = 9$ to the cutsize, which is equal to the amount of data transfer due to accessing B-matrix row 3, under the mentioned assumptions.

4.1.1 Hypergraph Partitioning (HP)

The state-of-the-art hypergraph partitioning tools such as PaToH [44] and hMeTiS [45] support the following cutsize metrics: hyperedge/net-cut [44], [45], "connectivity-1" metric [44], [46], [47] and sum of external degrees (SOED) [45]. None of these metrics directly encode the cutsize definition given in (6) for the proposed hypergraph model. On the other hand, it can easily be shown that there exist a constant difference of $nnz(B)$ between the cutsize definition in (6) and the cutsize definition according to the "connectivity-1" metric. So, "connectivity-1" cutsize metric of PaToH can be safely used for minimizing the connectivity cutsize definition given in (6).

4.1.2 Fast Bottom-Up Hypergraph Clustering (HC)

In order to reduce the preprocessing overhead due to the top-down partitioning using PaToH, we propose a method that performs bottom-up clustering on the proposed hypergraph model. This bottom-up clustering method exploits the multi-level coarsening phase of PaToH in such a way

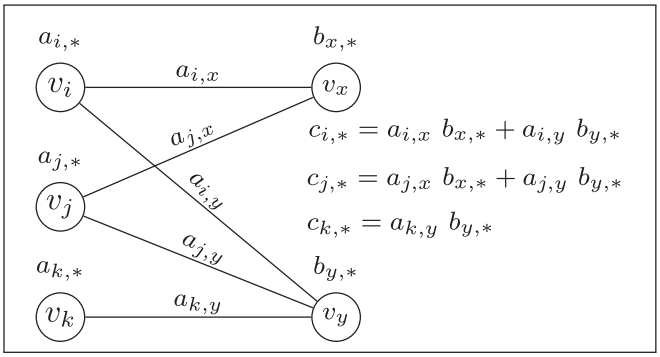


Fig. 6. Bipartite graph model for exploiting temporal locality in accessing B-matrix rows during thread-level row-by-row parallelization of SpGEMM operation.

that coarsening is continued until the number of supervertices in a level becomes smaller than or equal to $K' = 1.30K$. Note that each supervertex in the last level is decoded as a vertex part/cluster in such a way that the set of atomic tasks (fine-grain tasks) constituting that supervertex is taken as a coarse-grain task to be executed by a distinct thread.

The proposed method uses a multi-level matching-based clustering scheme. As the matching selection criteria, "absorption metric" and a variant of "scaled heavy connectivity metric" (SHCM) are implemented. The SHCM-variant used in this experimentation corresponds to the generalized Jaccard similarity metric [48]. This bottom-up method establishes a tradeoff between the solution quality and the preprocessing time and it will be referred to as hypergraph clustering HC.

HC aims at clustering vertices with similar net connectivities. This corresponds to clustering A-matrix rows that are similar in terms of the number of nonzeros of the required B-matrix rows to the same cluster, thus exploiting temporal locality in accessing B-matrix nonzeros.

HC does not enforce any explicit constraint for maintaining balance on clusters. So this scheme may incur load imbalance for instances with high atomic-task weight variation. In order to partially alleviate this bottleneck, the coarse-grain tasks obtained from partitioning are sorted in decreasing order with respect to their coarse-grain task weights and they are given to the OpenMP scheduler in this order.

4.2 Bipartite Graph Model $\mathcal{B}_A^T$

A given SpGEMM computation $C = A\,B$ is represented as a temporal bipartite graph $\mathcal{B}_A^T(A, \{nnz(b_{x,*})\}_x) = (\mathcal{V} = \mathcal{V}^A \cup V^B, \mathcal{E})$. $\mathcal{V}^A$ contains a vertex v_i for each row i of A and $\mathcal{V}^B$ contains a vertex v_x for each row x of B. $\mathcal{E}$ contains an edge $e_{i,x}$ between $v_i \in \mathcal{V}^A$ and $v_x \in \mathcal{V}^B$ if $a_{i,x}$ is nonzero. So the adjacency lists of the row and column vertices represent the sparsity patterns of the respective rows and columns. Note that $\mathcal{B}_A^T$ and $\mathcal{H}_A^T$ are topologically equivalent, where there exist one-to-one correspondences between vertices of $\mathcal{V}^A(\mathcal{B}_A^T)$ and vertices of $\mathcal{V}(\mathcal{H}_A^T)$, between vertices of $\mathcal{V}^B(\mathcal{B}_A^T)$ and nets of $\mathcal{N}(\mathcal{H}_A^T)$, and between edges of $\mathcal{E}(\mathcal{B}_A^T)$ and pins of $Pins(\mathcal{H}_A^T)$. Fig. 6 illustrates the input and output dependency of the proposed bipartite graph model.

Atomic task and weight definitions associated with the vertices of $\mathcal{V}(\mathcal{H}_A^T)$ in (3) and (4) directly apply to the vertices of $\mathcal{V}^A(\mathcal{B}_A^T)$, and vertices of $\mathcal{V}^B(\mathcal{B}_A^T)$ are associated with zero weight since they do not incur any computation. That is,

$$v_i \in \mathcal{V}^A\text{: } w(v_i) = \sum_{x \in cols(a_{i,*})} nnz(b_{x,*}), \quad v_x \in \mathcal{V}^B\text{: } w(v_x) = 0. \tag{7}$$

A K-way vertex partition of $\mathcal{B}_A^T$ induces a K-way partition on both $\mathcal{V}^A$ and $\mathcal{V}^B$. $\Pi(\mathcal{V}^A)$ of $\mathcal{B}_A^T$ is decoded in a way similar to decoding $\Pi(\mathcal{V})$ of $\mathcal{H}_A^T$. That is, the set of atomic tasks (fine-grain tasks) corresponding to the vertices in a part of $\Pi(\mathcal{V}^A)$ constitutes a coarse-grain task to be executed by a distinct thread. So, the partitioning constraint of maintaining balance on the part weights corresponds to maintaining balance on the computational loads of thread-level coarse-grain tasks. $\Pi(\mathcal{V}^B)$ is not decoded since vertices in $\mathcal{V}^B$ do not signify any computation.

4.2.1 $Edgecut$ Formulation (BGPe)

We associate each edge $e_{i,x}$ with a weight equal to

$$w(e_{i,x}) = nnz(b_{x,*}). \tag{8}$$

So the partitioning objective of minimizing

$$edgecut(\Pi) = \sum_{v_i \in \mathcal{V}_k \wedge v_x \in \mathcal{V}_{\ell \neq k}} w(e_{i,x}), \tag{9}$$

relates to minimizing the amount of B-matrix rows to be transferred from the memory and between different caches under the assumption that accessing each B-matrix row always incurs compulsory miss(es), i.e., there is no reuse of B-matrix rows. In other words, minimizing the $edgecut$ corresponds to minimizing a loose upper bound on the number of cache misses due to accessing B-matrix rows.

4.2.2 $Totalv$ Formulation (BGPv)

The above-mentioned $edgecut$ formulation is based on the fact that this objective is widely supported by almost all graph partitioning tools. Here, we propose a formulation that encodes the partitioning objective of minimizing the "total communication volume" ($totalv$) supported by MeTiS [49],

$$totalv(\Pi) = \sum_{v \in \mathcal{V}_b} s(v) Nadj(v). \tag{10}$$

Here, $\mathcal{V}_b$ denotes the set of boundary vertices, where a vertex is said to be *boundary* if it is incident to at least one cut edge. For a vertex $v \in \mathcal{V}_k$, $Nadj(v)$ denotes the number of parts other than $\mathcal{V}_k$ that the vertices adjacent to v (i.e., $Adj(v)$) belong to. In (10), $s(v)$ denotes the size of vertex v so that MeTiS assumes a weight $w(v)$ and a size $s(v)$ for each vertex v, where edges are unweighted.

In the proposed $totalv$ formulation, the vertex weight assignment is as same as in the $edgecut$ formulation (see (7)), whereas vertex size assignment is as follows:

$$v_i \in \mathcal{V}^A : \ s(v_i) = 0, \quad v_x \in \mathcal{V}^B : \ s(v_x) = nnz(b_{x,*}). \tag{11}$$

This vertex size assignment is based on establishing a one-to-one correspondence between vertices of $\mathcal{V}^B(\mathcal{B}_A^T)$ and nets of $\mathcal{N}(\mathcal{H}_A^T)$. Note that $Nadj(v_x)$ is equal to $|con(v_x)| - 1$ for $v_x \in \mathcal{V}^B$, where $con(v_x)$ to be the set of parts that the vertices in $\{v_x \cup Adj(v_x)\}$ reside.

It can easily be shown that, for the same partition Π on $\mathcal{V}(\mathcal{H}_A^T) \equiv \mathcal{V}^A(\mathcal{B}_A^T)$, we have a constant difference of $nnz(B)$ between $cutsize(\Pi(V(\mathcal{H}_A^T)))$ and $totalv(\Pi(\mathcal{V}(\mathcal{B}_A^T)))$. So, the $totalv$-based partitioning objective of the bipartite graph model becomes equivalent to the partitioning objective of the hypergraph model given in Section 4.1.

4.2.3 Fast Bottom-Up Graph Clustering (BGCe)

In order to reduce the preprocessing overhead due to the top-down partitioning, we propose a method that performs bottom-up clustering on the proposed bipartite graph models. Similar to the HC method proposed in Section 4.1.2, this method exploits only the multi-level coarsening phase of the multi-threaded graph partitioning tool mt-MeTiS [50]. The initial bipartitioning and refinement phases of mt-MeTiS are omitted. This method will be referred to as bipartite graph clustering BGCe since mt-MeTiS performs vertex matching according to weights of the connecting edges.

5 B-Matrix Row/Column Reordering for Locality in Accessing *acc* Array

In Sections 5.1 and 5.2, we respectively propose a spatial graph model and a temporal hypergraph model for preserving locality in accessing acc array. A K''-way vertex partition of both models is used to induce a partial ordering on either the rows or columns of the B matrix as follows: The rows/columns corresponding to the vertices in $\mathcal{V}_{k+1}$ are ordered after the rows/columns corresponding the vertices in $\mathcal{V}_k$ for $k = 1, 2, \ldots, K'' - 1$. In both reordering schemes, K'' is selected such that each vertex part is sufficiently small.

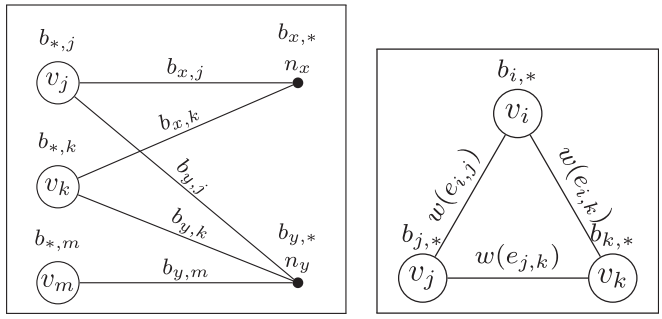
5.1 Spatial Hypergraph Model $\mathcal{H}_B^S$ for $C = A\,B$

In order to exploit spatial locality in accessing acc-array entries, the B-matrix columns with similar sparsity patterns should be reordered nearby. Furthermore, the sparsity pattern similarity among B-matrix columns should be weighted according to the access counts of the B-matrix rows that contain nonzeros in those columns. So, we propose to represent the pattern of accessing acc-array entries as a spatial hypergraph model $\mathcal{H}_B^S(B, \{nnz(a_{*,x})\}_x) = (\mathcal{V}, \mathcal{N})$ for column reordering of B matrix. Here, the superscript "S" stands for spatial locality. In $\mathcal{H}_B^S$, $\mathcal{V}$ contains a vertex v_j for each column j of matrix B. We associate vertices with unit weights. $\mathcal{N}$ contains a net n_x for each row x of matrix B. We associate each net n_x with a weight $w(n_x)$ equal to the number of times B-matrix row x will be accessed during $C = A\,B$. That is,

$$w(n_x) = nnz(a_{*,x}). \tag{12}$$

Fig. 7a illustrates the proposed model.

Let a single cache line contain L words. Consider a partition Π of vertices of $\mathcal{H}_B^S$, where each part contains L vertices. Assume that L number of B-matrix columns represented by vertices in a part are ordered consecutively so that corresponding acc-array entries are stored in consecutive locations of the memory (i.e., in the same cache line) and accessed consecutively by the SpGEMM algorithm. Consider a net $n_x \in \mathcal{H}_B^S$ with connectivity set $con(n_x)$ and with weight $w(n_x)$. This net n_x means that the B-matrix row x will be transferred from memory or another cache $w(n_x)$ times. Each time this row is transferred and accessed during multiplication, $|con(n_x)|$ number of lines containing acc-array entries will be accessed also. So, in total, the number of accesses to acc-array entries will be equal to $\sum_{n_x \in \mathcal{N}} w(n_x)|con(n_x)|$. Therefore, the cutsize according to the connectivity metric (6) corresponds to the number of accesses to the cache lines containing acc-array entries during $C = A\,B$. Hence, the proposed model exploits spatial locality in accessing acc-array entries.



(a) Spatial hypergraph model (b) Temporal graph model

Fig. 7. Models for exploiting spatial and temporal localities in accessing acc array.

5.2 Temporal Graph Model $\mathcal{G}_B^T$ for $C = A\,B$

In order to exploit temporal locality in accessing acc-array entries, B-matrix rows with similar sparsity patterns should be reordered nearby. Furthermore, the sparsity pattern similarity among B-matrix rows should be weighted according to the number of accesses to the same acc-arrray entries within the same coarse-grain thread-level tasks. So, exploiting this locality depends on coarse-grain task set induced by the rowwise A-matrix partition, $\Pi(A) = \{A_1, A_2, \ldots, A_K\}$, obtained via using the models proposed in Section 4. Thus, B-matrix-row access pattern is represented as a B-matrix-row similarity graph $\mathcal{G}_B^T(B, \Pi(A)) = (\mathcal{V}, \mathcal{E})$ for conformable column and row reordering of matrices A and B, respectively.

In $\mathcal{G}_B^T$, $\mathcal{V}$ contains a vertex v_i for each row i of matrix B. We associate vertices with unit weights. $\mathcal{E}$ contains an edge $e_{i,j}$ if B-matrix rows i and j have nonzeros in at least one common column and these rows are accessed together by at least one coarse-grain task executed by a thread. We associate each edge $e_{i,j}$ with a weight $w(e_{i,j})$ equal to the number of common columns of B-matrix rows i and j that are accessed together in the same coarse-grain task, i.e.,

$$w(e_{i,j}) = |\{x : a_{*,i}, a_{*,j} \in A_k; x \in cols(b_{i,*}) \cap cols(b_{j,*})\}|. \tag{13}$$

Fig. 7b illustrates the proposed temporal graph model.

Consider a partition $\Pi(\mathcal{G}_B^T)$ of vertices of the similarity graph $\mathcal{G}_B^T$ and an edge $e_{i,j} \in \mathcal{G}_B^T$ with weight $w(e_{i,j})$. This edge $e_{i,j}$ means that the B-matrix rows i and j involve accesses to $w(e_{i,j})$ same entries of acc array. If B-matrix rows i and j are always processed close in time, which in turn corresponds to the case where edge $e_{i,j}$ is uncut and parts of $\Pi(\mathcal{G}_B^T)$ are small enough, the $w(e_{i,j})$ different acc-array entries are likely to be reused. If $e_{i,j}$ is cut, these $w(e_{i,j})$ acc-array entries probably will not be reused. So, in total, the amount of cache misses

due to loss of *acc*-array entry reuse is proportional to $\sum_{e_{i,j} \in \mathcal{E}} w(e_{i,j})$. Therefore, partitioning objective given in (9) relates to maximizing the amount of reuse.

5.3 Using A-matrix Partitioning for $C=AA^T$ and $C=AA$

Here, we will show how a rowwise partition of matrix A obtained by partitioning the two models proposed in Section 4 can be used to exploit spatial and temporal localities in accessing *acc*-array entries during the SpGEMM operations of the forms $C=AA^T$ [10], [11], [12] and $C=AA$ [3], [7], [15], [16], [17]. We should here mention that in both cases, second input matrices A^T and A are stored separately from the first input matrix A.

A rowwise partition on the A matrix induces a partial ordering R^A_{row} on the rows of A, where rows in row slice A_{k+1} are ordered after the rows in A_k for $k = 1, 2, \ldots, K-1$. In the bipartite graph model $\mathcal{B}^T_A$, the partition $\Pi(\mathcal{V}^B)$ on the A-matrix columns also induces a partial column ordering R^A_{col} on the A-matrix columns in a similar manner. In the hypergraph model $\mathcal{H}^T_A$, the partition $\Pi(\mathcal{V})$ on the A-matrix rows induces a partial ordering R^A_{col} on the A-matrix columns as follows: The A-matrix columns corresponding to the uncut nets of $\mathcal{V}_{k+1}$ are ordered after the columns corresponding to the uncut nets of $\mathcal{V}_k$ for $k = 1, 2, \ldots, K-1$, whereas the columns corresponding to the cut nets are ordered last.

For $C=AA^T$ and $C=AA$ cases, R^A_{row} and R^A_{col} can be used for reordering rows and columns of B, where $B=A^T$ in the former case and $B=A$ in the latter case.

Here, we will briefly discuss how closely R^A_{row} and R^A_{col} serve the purpose of clustering B-matrix rows and columns for exploiting localities in $C=AA^T$ and $C=AA$. In the hypergraph model, the objective of minimizing cutsize directly and closely relates to clustering vertices with similar net connectivity to the same parts. So, R^A_{row} induced by the vertex partition $\Pi(\mathcal{V}^A)$ closely relates to clustering A-matrix rows with similar sparsity patterns. However, the partitioning objective of the hypergraph model indirectly and hence loosely relates to clustering nets with similar vertex connectivity to the same parts as uncut nets. So, R^A_{col} induced by the net reordering described above loosely relates to clustering A-matrix columns with similar sparsity patterns.

The bipartite graph model has the nice property of inducing R^A_{row} and R^A_{col} based on the vertex reordering obtained from the respective vertex partitions $\Pi(\mathcal{V}^A)$ and $\Pi(\mathcal{V}^B)$. However it may suffer from the relatively loose relation between the objective of graph partitioning and clustering vertices with similar adjacency patterns.

$C=AA^T$ Case. We exploit the simple fact of rows and columns of $B = A^T$ being columns and rows of A, respectively, as follows. For exploiting spatial locality, R^A_{row} is used for reordering A^T-matrix columns. For exploiting temporal locality, R^A_{col} is used for reordering A^T-matrix rows. Note that the ordering obtained on the rows of A^T is conformably applied on the columns of A.

$C=AA$ Case. We exploit the simple fact that the first and second input matrices are the same. For exploiting spatial locality, R^A_{col} is used for reordering columns of second A matrix. For symmetric matrices, R^A_{row} can be a better alternative for hypergraph models to avoid the disadvantage of using net ordering. For exploiting temporal locality, there are two options: In the first option, R^A_{col} is used for reordering the rows of second A matrix, whereas in the second option, R^A_{row} is used. Note that in both options, the ordering obtained on the rows of the second A is conformably applied on the columns of the first A matrix.

6 Experiments

6.1 Data Set

The validity of the proposed methods are tested on three different categories: $C=AA^T$, $C=AA$, and $C=A\,B$.

The $C=AA^T$ category contains 12 LP constraint matrices, all of which are selected from the UFL sparse matrix collection [51].

The $C=AA$ category contains 17 sparse matrices. Two of these matrices, `cp2k-h2o-e6` and `cp2k-h2o-.5e7`, are obtained from the simulation of H_2O molecules via using CP2K's implementation of KohnSham density functional theory calculations [3], [21]. The remaining 15 matrices are selected from the UFL collection. The two matrices, `144` and `cage12`, represent graphs, which can be used in finding all-pairs shortest-paths [15], self similarity joins of sparse datasets [16], and summarization of sparse data [17]. Some of the remaining matrices are included in this dataset because of their use in recent works [36], [37], [42] as synthetic applications.

The $C=A\,B$ category for the general SpGEMM case contains 4 SpGEMM instances. This general case arises in two applications: First application is item-to-item collaborative filtering [18] in recommendation systems. The first two A matrices in the $C=A\,B$ category, i.e., `amazon0302` and `amazon0312`, represent similarities between items, and they are obtained from the UFL collection. The corresponding B matrices are randomly generated according to Zipf distribution with exponent equal to 3.0. The second application arises in similarity joins of two different sparse datasets [16]. The remaining two SpGEMM instances in the $C=A\,B$ category represent sparse networks obtained from the DIMACS Implementation Challenges [52].

In the appendix, available in the online supplemental material, Table A.1 displays the properties of the 33 SpGEMM instances. For each category, the SpGEMM instances are listed in alphabetical order by name of the first input matrix. The properties of SpGEMM instances are displayed in terms of total number of rows, columns, and nonzeros of the input matrices, as well as their average and maximum number of nonzeros per row and column. The properties of SpGEMM instances are also displayed in terms of the number of thread-level coarse-grain tasks (K), and statistics of atomic tasks in terms of number of multiply-and-adds and kilo bytes (KB). K values are introduced to show the amount of parallelism in each SpGEMM instance. The "cov" (coefficient of variation) value of an SpGEMM instance is given as an indication of the level of irregularity in the atomic task weights. As seen in Table A.1, available in the online supplemental material, $C=AA^T$ instances have much higher "cov" values than $C=AA$ instances in general, thus showing the higher irregularity of LP instances.

For each of the 33 SpGEMM instances, the properties of the hypergraph and bipartite graph models can also be extracted from the information available in Table A.1,

available in the online supplemental material. Statistics given for rows of A matrices correspond to statistics for vertices in $\mathcal{V}(\mathcal{H}_A^T)$ and $\mathcal{V}^A(\mathcal{B}_A^T)$. Similarly, statistics given for columns of A matrices correspond to statistics for nets in $\mathcal{N}(\mathcal{H}_A^T)$ and vertices $\mathcal{V}^B(\mathcal{B}_A^T)$. Statistics given for rows of B matrices correspond to statistics for nets $\mathcal{N}(\mathcal{H}_B^S)$, whereas statistics given for columns correspond to statistics for vertices in $\mathcal{V}(\mathcal{H}_B^S)$.

6.2 Implementation of Partitioning Methods

6.2.1 Proposed Algorithms

A-matrix Partitioning. The hypergraph and bipartite graph models of the test SpGEMM instances are generated as described in Sections 4.1 and 4.2. The state-of-the-art tools PaToH [44] and MeTiS [49] are used for K-way partitioning of these hypergraphs and bipartite graphs, respectively. PaToH is used with the `PATOH_SUGPARAM_SPEED` parameter in order to trade off between the partitioning quality and the preprocessing overhead due to partitioning. MeTiS is used with default values except it is made to use multilevel recursive bisectioning. For the partitioning constraint, the maximum allowed imbalance threshold is set to be equal to 0.30 for both PaToH and MeTiS. For the partitioning objective, "connectivity-1" cutsize metric is used with PaToH, whereas both $edgecut$ and $totalv$ metrics are used with MeTiS.

As the selection criteria in the HC method, "absorption metric" and a variant of "scaled heavy connectivity metric" (SHCM) are respectively used for the $C = AA$ (and $C = A\,B$) and $C = AA^T$ categories.

The cache size threshold utilized for calculating K values for each SpGEMM instance is selected as half of the effective cache size per thread. This cache size thresholding scheme is used to alleviate the possibility of capacity misses due to the small set-associativity (8 ways) during the execution of a coarse-grain task.

PaToH, MeTiS, HC, and BGCe involve randomized algorithms. So for each SpGEMM instance, these tools/methods are run five times and the average results are reported in the following tables.

B-Matrix Row/Column Reordering. For each test SpGEMM instance, the spatial hypergraph model proposed in Section 5.1 is partitioned by PaToH using the same parameters mentioned above. For hypergraph models, K'' is calculated such that each part has about 10 vertices. For each rowwise A-matrix partition of each test SpGEMM instance, the temporal graph model proposed in Section 5.2 is partitioned using MeTiS three times, and the average result is reported. MeTiS is used with default values except it is made to use multilevel recursive bisectioning. For graph models, K'' is calculated such that each part has about 10 vertices.

6.2.2 Baseline Algorithms

We use two baseline algorithms MKL and BinP, neither of which exploits locality.

MKL. We use `mkl_dcsrmultcsr` function [53] of the latest MKL library (version 11.3). In the parallelization strategy adopted by MKL [54], the first input matrix is divided into chunks with more or less equal number of rows, and every row chunk is assigned to a thread. Since the number of chunks is equal to the number of threads, the chunk size cannot be set by the user outside MKL. So MKL considers balancing the computational loads of threads in a very rough manner.

BinP. The BinP algorithm is implemented to achieve a much better balancing of the computational loads of the threads as follows: BinP is a binpacking-based algorithm and it adapts the best-fit-decreasing heuristic used in solving the K-feasible binpacking problem [55]. In adapting this heuristic for our purpose, A-matrix rows are considered for assignment into one of the K bins in decreasing number of multiply-and-add operations incurred by the pre-multiply of this row with the B matrix. The best-fit criterion is the assignment of the A-matrix row to the minimally loaded bin (part). The bin capacity constraint is not used in BinP. At the termination of the algorithm, each bin represents a coarse-grain task to be executed by a distinct thread. The number of resulting parts becomes much larger than the number of threads thus enabling the utilization of dynamic part-to-thread scheduling for further load balancing.

The cache size thresholding scheme described in Section 6.2.1 for the proposed algorithms is also used to determine the K value in the BinP algorithm.

6.3 Parallel Systems and Implementation

We conducted experiments on a single Xeon Phi 5110P coprocessor. We used the offload mode instead of the native mode to enable future vertical integration that involves hybrid parallelization on a Xeon Phi cluster. The Xeon Phi coprocessor has 8 GB on-device RAM and provides 59 cores in the offload mode and each core can handle up to four hardware threads. Each core has 32 KB 8-way set associative L1 data cache with 64-byte lines, and 512 KB 8-way set associative L2 with 64-byte lines.

We also conducted experiments on a two-socket Xeon server. Each X5650 Xeon processor clocked at 2.67 GHz has 6 cores and 12 MB 16-way set associative L3 cache. Each core has 32 KB 8-way associative L1 cache with 8-byte lines and 256 KB 8-way set associative L2 with 8-byte lines. Only Tables 4, A.2, and A.4, available in the online supplemental material, contain results on the two-socket Xeon server, whereas all other tables contain results for Xeon Phi.

For evaluating the performance of the proposed models as well as BinP, SpGEMM routines are implemented and integrated into `shoc-mic` benchmark [56], which is compiled with `-O3` flag. OpenMP's dynamic scheduler is used with the default chunk size. The best results for 59, 118, 177, and 236 threads are reported for Xeon Phi and results for 12 threads are reported for the Xeon server.

6.4 Parallel SpGEMM Performance

This section compares the SpGEMM performance of the methods without considering the preprocesing overhead, whereas Section 6.5 gives an overall discussion including the preprocessing overheads.

6.4.1 Sorting Coarse-Grain Tasks for Computational Load Balancing in HC

Table 1 displays the performance of the sorting-based coarse-grain task scheduling scheme described for HC at the end of Section 4.1.2 for the $C = AA^T$ category. In the

TABLE 1
Coarse-Grain Task Sorting in HC for $C = AA^T$

Matrix	Coarse grain task: count K	Coarse grain task: weight avg	Coarse grain task: weight max	Coarse grain task: weight cov	#of threads T	Percent impr.
e18	411	8606	15060	0.22	236	12.7%
fxm3_16	766	5301	44310	1.44	177	27.8%
fxm4_6	556	5591	17268	0.93	236	30.2%
rlfdual	259	8000	20787	0.53	177	15.9%
rlfprim	3821	8701	23632	0.53	236	3.1%
sc205-2r	4260	4900	25709	0.63	118	8.7%
scfxm1-2b	278	6176	25415	1.12	118	16.4%
scfxm1-2r	1319	6560	44432	1.28	236	19.8%
scrs8-2r	2327	5922	25160	0.71	177	11.2%
scsd8-2r	4393	7363	15660	0.37	236	5.9%
sctap1-2b	1754	6098	22924	0.90	236	20.1%
testbig	1006	5198	19412	0.81	177	22.0%

HC: Fast bottom-up clustering method (see Section 4.1.2)

table, the properties of the coarse-grain tasks are given in terms of the number K of tasks, the average and maximum task weights, and the covariance of task weights. Each T value shows the number of threads that achieve the best reported result for the respective SpGEMM instance. The last column displays the percent performance improvement achieved by using the sorting-based scheduling scheme.

As seen in Table 1, the proposed sorting scheme considerably improves the performance in all SpGEMM instances. As seen in the table, sorting scheme achieves relatively better performance for small K/T and large covariance values, as expected. For example, sorting does not improve the performance much for the `rlfprim`, `sc205-2r` and `scsd8-2r` instances, which have large K/T ratios of 3821/236≈16, 4260/118≈36, and 4393/236≈19, respectively.

On the average, the proposed sorting scheme improves HC by 12.9, 0.9, and 3.8 percent for $C = AA^T$, $C = AA$, and $C = A\,B$ categories, respectively. The significant amount of improvement in the $C = AA^T$ category can be attributed to the relatively smaller average K/T ratio of $(K/T)_{avg}$=10 and higher average covariance value of cov_{avg}=0.71 of this category, whereas $(K/T)_{avg}$=24 and cov_{avg}=0.29 for $C = AA$, and $(K/T)_{avg}$=14 and cov_{avg}=0.64 for $C = A\,B$.

We should note here that, for the BGCe method, the sorting scheme achieves only slight improvement of 1.5 percent in $C = AA^T$ whereas it does not achieve any improvement for the other two categories.

As expected, the sorting scheme does not considerably improve the performance of partitioning-based methods since they explicitly enforce balance among coarse-grain task weights.

6.4.2 Exploiting Locality in Accessing acc Array

Here, we evaluate the validity of the B-matrix row/column reordering algorithms $\mathcal{H}_B^S$, $\mathcal{G}_B^T$, R_{col}^A, and R_{row}^A proposed in Section 5. For this purpose, we report the performance improvement induced by these reordering algorithms on each of the five methods HP, HC, BGPe, BGPv, and BGCe.

Table 2 displays the average performance improvement over the original orderings of A-matrix columns and B-matrix rows and columns. The "Spatial" and "Temporal" columns respectively refer to the separate use of B-matrix-column and B-matrix-row reorderings described in Section 5.

As seen in Table 2, the spatial $\mathcal{H}_B^S$ model achieves significant performance improvement in all categories. The temporal $\mathcal{G}_B^T$ model achieves significant improvement in $C = AA$ and $C = A\,B$, whereas it achieves relatively small improvement in $C = AA^T$. These experimental findings show the validity of $\mathcal{H}_B^S$ and $\mathcal{G}_B^T$ models.

Here, for the $C = AA^T$ and $C = AA$ cases, we compare the performance of R_{row}^A/R_{col}^A against the performance of $\mathcal{H}_B^S$ and $\mathcal{G}_B^T$. As seen in Table 2, although $\mathcal{H}_B^S$ performs better than R_{col}^A/R_{row}^A in general, the performance gap is rather small for HP, BGPe, and BGPv, whereas it is significant in HC and BGCe. For example, for HC and BGCe on the $C = AA^T$ category, while $\mathcal{H}_B^S$ achieves 38.5 and 38.9 percent performance improvement, R_{row}^A achieves only -7.7 and 8.3 percent, respectively. This may be attributed to the high sensitivity of exploiting spatial locality to sparsity patterns of B-matrix columns and less sophistication of HC and BGCe compared to HP and BGPe, respectively. A similar discussion follows for the comparison of R_{row}^A/R_{col}^A against $\mathcal{G}_B^T$ for exploiting temporal locality. So for B-matrix row/column reordering in the $C = AA^T$ and $C = AA$ cases, we recommend the use of $\mathcal{H}_B^S$ and $\mathcal{G}_B^T$ only for HC and BGCe, whereas we recommend the use of R_{row}^A/R_{col}^A, which are induced by existing A-matrix partitionings, for all other methods.

6.4.3 Exploiting Temporal Locality in Accessing B Matrix

Table 3 compares the performance (in terms of GFlops) of the proposed locality-aware HP, HC, BGPe, BGPv, and BGCe methods against the performance of the baseline MKL and BinP methods on Xeon Phi. For all SpGEMM instances, the

TABLE 2
Performance Improvement of B-matrix Row/Column Reordering for Locality in Accessing acc on Xeon Phi

	$C = AA^T$ (LP)				$C = AA$						$C = AB$	
	Spatial		Temporal		Spatial			Temporal			Spatial	Temporal
	R_{row}^A	$\mathcal{H}_B^S$	R_{col}^A	$\mathcal{G}_B^T$	R_{col}^A	R_{row}^A	$\mathcal{H}_B^S$	R_{col}^A	R_{row}^A	$\mathcal{G}_B^T$	$\mathcal{H}_B^S$	$\mathcal{G}_B^T$
HP	34.6%	38.0%	0.8%	0.1%	0.4%	18.6%	19.3%	1.6%	8.5%	8.8%	20.4%	4.8%
HC	-7.7%	38.5%	0.8%	1.7%	0.8%	10.3%	18.9%	1.8%	5.7%	8.4%	18.4%	1.6%
BGPe	30.8%	36.2%	1.0%	0.6%	19.7%	19.4%	19.2%	9.4%	8.8%	9.1%	20.4%	4.5%
BGPv	29.2%	37.8%	1.8%	2.7%	19.3%	18.0%	18.7%	8.5%	8.7%	9.1%	20.4%	4.6%
BGCe	8.3%	38.9%	0.8%	2.8%	6.0%	5.5%	20.4%	5.0%	4.6%	10.4%	20.7%	2.5%

Reordering for spatial locality: reordering B-matrix columns. Reordering for temporal locality: reordering B-matrix rows.

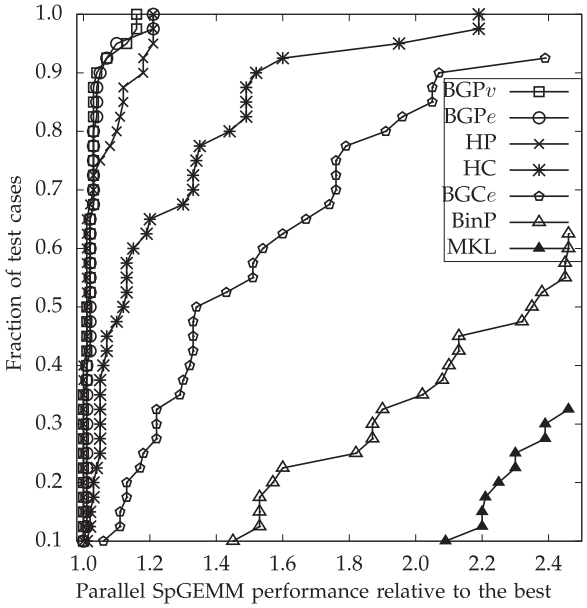


Fig. 8. GFlops performance profiles on Xeon Phi.

proposed methods use the best-performing B-matrix row and column reordering methods as discussed in Section 5 for exploiting locality in accessing acc array. In the table, for each method, performance averages and the number of best results attained over each of the three different categories are listed at the bottom of each part, whereas overall averages and number of best results are displayed at the bottom of the table. A bold value in a row of the table indicates the highest GFlops performance attained for that instance.

In Fig. 8, we present a performance profile, which is a generic tool introduced by Dolan and Moré [57], in order to give a more comprehensive view of the runtime results of the proposed, as well as the baseline methods. The test set in this figure consists of all instances listed in Table A.1, available in the online supplemental material.

We will first briefly discuss the relative performance of the two baseline methods MKL and BinP. As seen in Table 3, the proposed baseline algorithm BinP performs significantly better than MKL both in $C=AA^T$ and $C=A\,B$ categories, whereas BinP and MKL display comparable performance in the $C=AA$ category. On average, BinP performs 2.32x and 1.39x better than MKL for $C=AA^T$ and $C=A\,B$ categories, respectively, whereas BinP performs only 1.04x better than MKL for the $C=AA$ category. This is because atomic tasks of the SpGEMM instances in the $C=AA^T$ and $C=A\,B$ categories are much more irregular compared to those in $C=AA$ category as mentioned in Section 6.1. So, by assigning equal number of A-matrix rows to threads, MKL may attain reasonable load balancing among threads in the $C=AA$ category. This significant performance improvement of BinP over MKL for the $C=AA^T$ and $C=A\,B$ categories shows the merits of using load balancing alone.

As seen in Table 3, all of the proposed locality exploiting methods perform significantly better than both baseline methods for all of the three categories. Among the proposed methods, top-down partitioning-based methods HP, BGPe, and BGPv perform better than the bottom-up clustering methods HC and BGCe as expected. As seen in Table 3, top-down partitioning-based methods HP, BGPe, and BGPv display comparable performance for all categories. As seen in the performance profile curves given in Fig. 8, in almost 75 percent of the instances, BGPe, BGPv, and HP perform nearly same.

As seen in Table 3, top-down partitioning-based methods HP, BGPe and BGPv performs 3.14x, 3.18x, and 3.19x better than MKL on the overall average. As also seen in the table, HP, BGPe, and BGPv perform 2.18x, 2.21x, and 2.21x better than BinP on the overall average. These relative performance improvements of HP, BGPe, and BGPv over BinP show the benefit of exploiting locality.

TABLE 3
Performance Results (in GFlops) on Xeon Phi

	Baseline methods (no locality)		Proposed methods (locality aware)				
			Hypergraph		Bipartite graph		
			part.	cluster.	partitioning		cluster.
	MKL	BinP	HP	HC	BGPe	BGPv	BGCe
$C=AA^T$ (Linear programming)							
e18	0.65	1.60	**3.41**	2.54	3.10	3.34	2.26
fxm3_16	2.48	3.53	5.79	4.24	6.19	6.25	**6.41**
fxm4_6	1.75	4.29	5.84	5.16	5.66	6.07	**6.85**
rlfdual	0.51	2.76	**4.00**	2.50	3.94	3.86	1.68
rlfprim	1.84	2.90	5.52	4.13	5.50	5.42	3.88
sc205-2r	1.32	1.41	4.86	3.94	5.75	**5.86**	1.45
scfxm1-2b	1.04	3.06	4.73	2.61	**5.70**	4.92	3.71
scfxm1-2r	1.20	3.47	7.30	5.67	7.62	8.13	4.68
scrs8-2r	1.15	2.42	**5.59**	4.66	5.46	5.26	2.70
scsd8-2r	1.54	9.00	**11.78**	11.29	11.71	11.51	8.88
sctap1-2b	2.59	3.73	**5.69**	4.81	5.60	5.56	2.90
testbig	1.33	2.59	5.03	4.18	5.35	5.39	2.83
Average	1.31	3.04	5.52	4.26	5.68	5.68	3.48
# of bests	–	–	6	–	1	3	2
$C=AA$							
144	1.66	1.75	6.28	6.55	6.81	**6.88**	2.67
2cubes_sphere	3.17	2.84	6.93	6.97	6.91	**6.98**	5.23
cage12	2.04	1.91	**4.67**	4.26	4.51	4.54	2.66
conf5_4-8x8-05	5.10	7.96	10.61	**10.65**	10.24	10.47	9.10
cop20k_A	2.46	3.93	**9.65**	9.27	9.64	9.54	5.51
cp2k-h2o-.5e7	2.99	3.43	8.11	7.95	8.05	**8.14**	7.21
cp2k-h2o-e6	2.72	2.53	**6.69**	6.47	6.61	6.66	6.03
filter3D	3.75	3.48	**9.73**	9.66	9.52	9.63	6.11
mac_econ_fwd500	1.67	1.24	3.31	3.02	3.34	**3.39**	2.92
majorbasis	2.89	3.09	**6.49**	6.22	6.42	6.34	4.95
mario002	1.27	1.22	**4.41**	3.94	4.41	4.41	2.48
mc2depi	1.04	0.83	**3.42**	3.27	3.33	3.35	2.63
offshore	2.83	2.17	6.99	7.02	6.99	**7.05**	4.24
poisson3Da	2.03	3.20	8.39	7.92	8.21	**8.42**	4.11
scircuit	1.29	1.72	4.95	3.36	4.84	**4.98**	3.75
tmt_sym	2.23	1.90	5.90	5.65	5.90	**5.93**	4.62
torso2	2.84	3.23	**6.53**	6.36	6.44	6.52	6.40
Average	2.28	2.38	6.31	5.97	6.27	6.32	4.41
# of bests	–	–	8	1	–	8	–
$C=AB$							
amazon0302	1.36	1.89	2.54	2.25	**2.58**	2.56	2.12
amazon0312	1.60	1.90	2.86	2.57	2.87	**2.89**	2.39
rgg_n_2_16_s0	0.74	0.65	1.69	1.74	**1.76**	1.71	1.66
smallworld	0.28	0.72	0.95	0.58	**1.12**	1.11	1.00
Average	0.82	1.14	1.85	1.55	1.95	1.94	1.70
# of bests	–	–	–	–	3	1	–
Overall average	1.65	2.38	5.18	4.48	5.25	5.27	3.61
Overall # of bests	–	–	14	1	4	12	2

TABLE 4
Average Performance Results (in GFlops) on Two-Socket Xeon Server

	Baseline methods (no locality)		Proposed methods (locality aware)				
			Hypergraph		Bipartite graph		
			part.	cluster.	partitioning		cluster.
	MKL	BinP	HP	HC	BGPe	BGPv	BGCe
$C = AA^T$ (Linear programming)							
Average	3.32	4.31	6.06	4.85	6.01	6.04	4.94
# of bests	–	–	5	1	2	4	–
$C = AA$							
Average	2.22	1.63	4.61	4.39	4.60	4.62	3.24
# of bests	–	–	5	–	2	10	–
$C = AB$							
Average	0.86	1.18	1.90	1.85	1.92	1.94	1.57
# of bests	–	–	–	1	–	3	–
Overall average	2.29	2.23	4.57	4.10	4.56	4.58	3.46
Overall # of bests	–	–	10	2	4	17	–

Despite the inferior performance of HC and BGCe over the top-down partitioning-based methods, they still perform significantly better than BinP, where HC is the clear winner. HC and BGCe perform 1.88x and 1.52x better than BinP on the overall average. The better performance of HC over BGCe can be attributed to the use of nets that encode multi-way similarity instead of edges that encode only two-way similarity during the bottom-up clustering process.

Table 4 displays results of experiments conducted on Xeon server. In Table 4, GFlops performance results are displayed as averages over the three different SpGEMM categories. We refer the reader to Table A.2 of the appendix for instance-based detailed performance results, available in the online supplemental material. Comparison of Tables 3 and 4 show that the performance gap between the proposed locality aware methods and the baseline method BinP slightly reduces on Xeon compared to Xeon Phi. For example, the performance improvement of HP over BinP decreases from 2.18x on Xeon Phi to 2.05x on Xeon on the overall average. The average performance gap between HP and BinP remains almost the same (1.62x versus 1.61x) for the $C = A\,B$ category. The average performance improvement of HP over BinP decreases from 1.82x on Xeon Phi to 1.41x on Xeon for the $C = AA^T$ category, whereas the average improvement of HP over BinP increases from 2.65x on Xeon Phi to 2.83x on Xeon for the $C = AA$ category. This experimental finding may be attributed to the reduced cache miss overhead in Xeon due to out-of-order execution capability as opposed to the in-order execution of Xeon Phi.

6.4.4 Reducing Data Transfer

In the appendix, Table A.3, available in the online supplemental material, is introduced in order to compare the improvements provided by the proposed partitioning-based methods against BinP in terms of cutsize. For each SpGEMM instance, the cutsize (computed according to (6)) of the respective method is divided by the cutsize of the baseline method BinP. As seen in the table, the normalized cutsize values in general conform with the relative GFlops performance values of the respective methods given in Table 3. For example, as seen in Table A.3, available in the online supplemental material, for the $C = AA^T$ and $C = AA$ categories, HP, which aims at exploiting locality, respectively achieves 3.22x and 8.33x less cutsize than the baseline partitioning method BinP which only considers load-balancing. As seen in Table 3, HP achieves respectively 1.82x and 2.65x speedups over BinP for the $C = AA^T$ and $C = AA$ categories, on average.

TABLE 5
Partitioning Overhead in Terms of Parallel MKL SpGEMM Times

		Proposed methods (locality aware)				
		Hypergraph		Bipartite graph		
		partition.	cluster.	partitioning		cluster.
	BinP	HP	HC	BGPe	BGPv	BGCe
$C = AA^T$	0.14	18.58	2.33	17.07	95.99	1.14
$C = AA$	0.47	80.18	7.33	58.81	219.23	2.02
$C = AB$	0.34	36.81	6.37	31.21	165.97	0.99
Overall	0.29	42.87	4.75	34.74	156.98	1.50

We also conducted experiments with `likwid` [58], which enables counting data transfers in a multi-threaded setting, to measure data transfer between L2 caches and last level caches of the Xeon server. Table A.4, available in the online supplemental material, displays the data transfer amounts incurred by the proposed partitoning-based methods normalized with respect to those of BinP. As seen in Table A.4, available in the online supplemental material, the proposed locality-exploiting methods achieve up to 2.63x less data transfers than BinP, on the overall average. Comparison of Tables A.4, available in the online supplemental material, and 4 show that data transfer amounts incurred by the partitioning-based methods correlate with the attained GFlops performances. For example, on the overall average, BinP incurs 2.44x more data transfers than HP, whereas HP performs 2.18x better than BinP. In conclusion, the cutsize minimization objective in the proposed methods successfully achieve reducing pressure on memory and caches.

6.5 Partitioning Overhead versus SpGEMM Performance

Table 5 is introduced to compare the partitioning overheads of the proposed methods, as well as BinP. For each SpGEMM instance, the partitioning time of the respective method in the host machine (Xeon) is divided by the parallel SpGEMM time obtained by MKL on Xeon Phi and averages of these normalized values over the matrix categories are reported in the table. For BGCe, which uses multi-threaded implementation, running times for the number of threads that achieve the lowest time are used.

As seen in Tables 3 and 5, BinP performs considerably better than MKL in terms of parallel SpGEMM performance, whereas the preprocessing overhead of BinP amortizes for only one SpGEMM operation. Hence BinP is a very simple yet effective heuristic that can easily be integrated into existing libraries.

The comparison of HP and BGPv is as follows: As seen in Table 5, BGPv incurs significantly higher partitioning

overhead than both HP and BGPe. Although BGPv achieves a similar parallel SpGEMM performance as HP, it is not recommended because of its significantly higher preprocessing overhead.

The comparison of HP and BGPe is as follows: Although bipartite graph and hypergraph models are of similar size (same number of edges and pins), HP incurs higher preprocessing overhead than BGPe as seen in Table 5. This is due to the fact that graph partitioning is considerably faster than hypergraph partitioning in general. As BGPe shows a close SpGEMM performance to HP on the average, BGPe can be considered as a good alternative to HP because of its considerably lower preprocessing overhead.

The comparison of HP and HC is as follows: In terms of parallel SpGEMM performance, for $C = AA$ and $C = A\,B$ categories, the average performance of HC is close to that of HP as seen in Table 3. In terms of partitioning time, on the average, HC runs approximately 10x faster than HP as seen in Table 5. Hence the fast bottom-up clustering approach is recommended for $C = AA$ and $C = A\,B$ categories instead of HP.

The comparison of BGPe and BGCe is as follows: In terms of parallel SpGEMM performance, BGCe performs significantly worse (31 percent worse) than BGPe, on the overall average. On the other hand, BGCe runs approximately 23x faster than BGPe, on the overall average. For example, running time of BGCe is no more than the running time of a single MKL SpGEMM time for the $C = AB$ category.

7 Conclusion

We investigated row-by-row formulation of the sparse matrix-matrix multiplication (SpGEMM) operation of the form $C = A\,B$ for locality-aware parallelization on many-core architectures. We proposed a hypergraph model and a bipartite graph model for 1D rowwise A-matrix partitioning to exploit locality in accessing B-matrix rows. In the partitioning methods utilizing these models, the partitioning constraint corresponds to maintaining balance on the computational loads of threads, whereas the partitioning objective relates to reducing data transfer amount from memory and between caches. For both hypergraph and bipartite graph models, we proposed bottom-up clustering methods, which were experimentally shown to be producing reasonably good partitions while being significantly faster than the respective partitioning methods.

We also investigated how to exploit locality in accessing entries of temporary arrays utilized by threads during accumulation of results for C-matrix rows. We proposed a hypergraph and a graph model for B-matrix column and row reordering to exploit spatial and temporal localities in these operations, respectively.

We tested the validity of the proposed methods on a wide range of realistic SpGEMM instances from three different categories of $C = AA^T$, $C = AA$, and $C = A\,B$. Experimental results showed the validity of the proposed methods. These results also showed that Xeon Phi and Xeon processors can benefit from locality enhancements for sparse and irregular applications through intelligent partitioning and reordering.

Acknowledgment

This work was partially supported by the Scientific and Technological Research Council of Turkey (TUBITAK) under Grant EEEAG-115E212. The authors would also like to acknowledge the contribution of the COST Action IC1406 High-Performance Modeling and Simulation for Big Data Applications (cHiPSet).

References

[1] M. Challacombe, "A general parallel sparse-blocked matrix multiply for linear scaling SCF theory," *Comput. Phys. Commun.*, vol. 128, no. 12, pp. 93–107, 2000.

[2] S. Itoh, P. Ordejón, and R. M. Martin, "Order-N tight-binding molecular dynamics on parallel computers," *Comput. phys. Commun.*, vol. 88, no. 2, pp. 173–185, 1995.

[3] J. VandeVondele, U. Borstnik, and J. Hutter, "Linear scaling self-consistent field calculations with millions of atoms in the condensed phase," *J. Chem. Theor. Comput*, vol. 8, no. 10, pp. 3565–3573, 2012.

[4] N. Hine, P. Haynes, A. Mostofi, and M. Payne, "Linear-scaling density-functional simulations of charged point defects in Al2O3 using hierarchical sparse matrix algebra," *J. Chemical Phys.*, vol. 133, no. 11, 2010, p. 114111.

[5] C. Saravanan, Y. Shao, R. Baer, P. N. Ross, and M. Head-Gordon, "Sparse matrix multiplications for linear scaling electronic structure calculations in an atom-centered basis set using multiatom blocks," *J. Comput. Chemistry*, vol. 24, no. 5, pp. 618–622, 2003.

[6] D. Bowler, T. Miyazaki, and M. Gillan, "Parallel sparse matrix multiplication for linear scaling electronic structure calculations," *Comput. Phys. Commun.*, vol. 137, no. 2, pp. 255–273, 2001.

[7] U. Borstnik, J. VandeVondele, V. Weber, and J. Hutter, "Sparse matrix multiplication: The distributed block-compressed sparse row library," *Parallel Comput.*, vol. 40, no. 5, pp. 47–58, 2014.

[8] Total-FETI massively parallel implementation research group. (2015). [Online]. Available: http://spomech.vsb.cz/feti/

[9] V. Hapla, D. Hork, and M. Merta, "Use of direct solvers in TFETI massively parallel implementation," in *Applied Parallel and Scientific Computing*, P. Manninen and P. ster, Eds. Berlin, Germany: Springer, 2013, pp. 192–205.

[10] R. H. Bisseling, T. M. Doup, and L. Loyens, "A parallel interior point algorithm for linear programming on a network of transputers," *Ann. Operations Res.*, vol. 43, pp. 51–86, 1993.

[11] E. Boman, O. Parekh, and C. Phillips, "LDRD final report on massively-parallel linear programming: The parPCx system," Sandia National Laboratories Albuquerque, NM, USA, Tech. Rep. SAND2004–6440, 2005.

[12] G. Karypis, A. Gupta, and V. Kumar, "A parallel formulation of interior point algorithms," in *Proc. ACM/IEEE Conf. Supercomputing*, 1994, pp. 204–213.

[13] W. L. Briggs, V. E. Henson, and S. F. McCormick, *A Multigrid Tutorial*. Philadelphia, PA, USA: SIAM, 2000.

[14] A. Buluç and J. R. Gilbert, "The Combinatorial BLAS: Design, implementation, and applications," *Int. J. High Performance Comput. Appl.*, vol. 25, no. 4, pp. 496–509, 2011.

[15] P. D'Alberto and A. Nicolau, "R-Kleene: A high-performance divide-and-conquer algorithm for the all-pair shortest path for densely connected networks," *Algorithmica*, vol. 47, no. 2, pp. 203–213, 2007.

[16] C. Ordonez, "Optimization of linear recursive queries in SQL," *IEEE Trans. Knowl. Data Eng.*, vol. 22, no. 2, pp. 264–277, Feb. 2010.

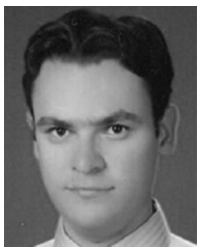
[17] C. Ordonez, Y. Zhang, and W. Cabrera, "The Gamma matrix to summarize dense and sparse data sets for big data analytics," *IEEE Trans. Knowl. Data Eng.*, vol. 28, no. 7, pp. 1905–1918, Jul. 2016.

[18] G. Linden, B. Smith, and J. York, "Amazon.com recommendations: item-to-item collaborative filtering," *IEEE Internet Comput.*, vol. 7, no. 1, pp. 76–80, Jan. 2003.

[19] J. Kepner, Ed.,*Graph Algorithms in the Language of Linear Algebra*. Philadelphia, PA, USA: SIAM, 2011.

[20] A. Buluc and J. R. Gilbert, "On the representation and multiplication of hypersparse matrices," in *Proc. IEEE Int. Symp. Parallel Distrib. Process.*, 2008, pp. 1–11.

[21] K. Akbudak and C. Aykanat, "Simultaneous input and output matrix partitioning for outer-product–parallel sparse matrix-matrix multiplication," *SIAM J. Scientific Comput.*, vol. 36, no. 5, pp. C568–C590, 2014.
[22] A. Buluc and J. R. Gilbert, "Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments," *SIAM J. Scientific Comput.*, vol. 34, no. 4, pp. C170–C191, 2012.
[23] F. G. Gustavson, "Two fast algorithms for sparse matrices: Multiplication and permuted transposition," *ACM Trans. Math. Softw.*, vol. 4, no. 3, pp. 250–269, 1978.
[24] K. Akbudak, O. Selvitopi, and C. Aykanat, "Partitioning models for scaling parallel sparse matrix-matrix multiplication," *ACM TOPC (under review)*.
[25] G. Ballard, et al., "Communication optimal parallel multiplication of sparse random matrices," in *Proc. 25th Annu. ACM Symp. Parallelism Algorithms Archit.*, 2013, pp. 222–231.
[26] G. Ballard, A. Druinsky, N. Knight, and O. Schwartz, "Brief announcement: Hypergraph partitioning for parallel sparse matrix-matrix multiplication," in *Proc. ACM Symp. Parallelism Algorithms Archit.*, 2015, pp. 86–88.
[27] A. Azad, et al., "Exploiting multiple levels of parallelism in sparse matrix-matrix multiplication," *CoRR*, vol. abs/1510.00844, 2015. [Online]. Available: http://arxiv.org/abs/1510.00844
[28] P. D. Sulatycke and K. Ghose, "Caching-efficient multithreaded fast multiplication of sparse matrices," in *Proc. 1st Merged Int. Parallel Proc. Symp. Symp. Parallel Distrib. Process.*, 1998, pp. 117–123.
[29] NVIDIA Corporation, *Cusparse Library*, NVIDIA Corporation, Santa Clara, California, 2014, https://developer.nvidia.com/cusparse
[30] N. Bell and M. Garland, "CUSP: Generic parallel algorithms for sparse matrix and graph computations," 2014, http://cusplibrary.github.io/
[31] N. Bell, S. Dalton, and L. N. Olson, "Exposing fine-grained parallelism in algebraic multigrid methods," *SIAM J. Scientific Comput.*, vol. 34, no. 4, pp. C123–C152, 2012.
[32] S. Dalton, L. Olson, and N. Bell, "Optimizing sparse matrix-matrix multiplication for the GPU," *ACM Trans. Math. Softw.*, New York, NY, USA, vol. 41, no. 4, pp. 25:1–25:20, Oct. 2015, http://doi.acm.org/10.1145/2699470
[33] L. Polok, S. V. Ila, and P. Smrvz, "Fast sparse matrix multiplication on GPU," in *Proc. Symp. High Performance Comput.*, 2015, pp. 33–40.
[34] K. Matam, S. Indarapu, and K. Kothapalli, "Sparse matrix-matrix multiplication on modern architectures," in *Proc. 19th Int. Conf. High Performance Comput.*, Dec. 2012, pp. 1–10.
[35] M. M. A. Patwary, et al., "Parallel efficient sparse matrix-matrix multiplication on multicore platforms," in *Proc. Symp. High Performance Comput.*, 2015, pp. 48–57.
[36] W. Liu and B. Vinter, "An efficient GPU general sparse matrix-matrix multiplication for irregular data," in *Proc. IEEE 28th Int. Parallel Distrib. Processing Symp.*, 2014, pp. 370–381.
[37] F. Gremse, A. Höfter, L. O. Schwen, F. Kiessling, and U. Naumann, "GPU-accelerated sparse matrix-matrix multiplication by iterative row merging," *SIAM J. Scientific Comput.*, vol. 37, no. 1, pp. C54–C71, 2015.
[38] J. Siegel, O. Villa, S. Krishnamoorthy, A. Tumeo, and X. Li, "Efficient sparse matrix-matrix multiplication on heterogeneous high performance systems," in *Proc. IEEE Int. Conf. Cluster Comput. Workshops Posters*, 2010, pp. 1–8.
[39] K. Akbudak, E. Kayaaslan, and C. Aykanat, "Hypergraph partitioning based models and methods for exploiting cache locality in sparse matrix-vector multiplication," *SIAM J. Scientific Comput.*, vol. 35, no. 3, pp. C237–C262, 2013.
[40] E. Saule, K. Kaya, and Ü. V. Çatalyürek, "Performance evaluation of sparse matrix multiplication kernels on Intel Xeon Phi," in *Parallel Processing and Applied Mathematics*. Berlin, Germany: Springer, 2014, pp. 559–570.
[41] M. O. Karsavuan, K. Akbudak, , and C. Aykanat, "Locality-aware parallel sparse matrix-vector and matrix-transpose-vector multiplication on many-core processors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 6, pp. 1713–1726, Jun. 2016.
[42] M. McCourt, B. Smith, and H. Zhang, "Sparse matrix-matrix products executed through coloring," *SIAM J. Matrix Anal. Appl.*, vol. 36, no. 1, pp. 90–109, 2015.
[43] S. Ramos and T. Hoefler, "Modeling communication in cache-coherent SMP systems: A case-study with Xeon Phi," in *Proc. 22nd Int. Symp. High-Performance Parallel Distrib. Comput.*, 2013, pp. 97–108.
[44] Ü. V. Çatalyürek and C. Aykanat, *PaToH: A Multilevel Hypergraph Partitioning Tool, Version 3.0, Comput. Eng. Dept., Bilkent Univ., Ankara, Turkey.*, 1999.
[45] G. Karypis and V. Kumar, "hMETIS: A hypergraph partitioning package, version 1.5.3," Dept. Comput. Sci., Univ. Minnesota, Minneapolis, MN, USA, 1998.
[46] U. V. Çatalyürek and C. Aykanat, "Hypergraph-partitioning based decomposition for parallel sparse-matrix vector multiplication," *IEEE Trans. Parallel Distrib. Syst.*, vol. 10, no. 7, pp. 673–693, Jul. 1999.
[47] B. U car and C. Aykanat, "Revisiting hypergraph models for sparse matrix partitioning," *SIAM Rev.*, vol. 49, no. 4, pp. 595–603, 2007.
[48] H. Späth, "The minisum location problem for the Jaccard metric," *Operations-Research-Spektrum*, vol. 3, no. 2, pp. 91–94, 1981.
[49] G. Karypis and V. Kumar, "METIS manual, version 5.1," Dept. Comput. Sci., Univ. Minnesota, Minneapolis, MN, USA, 1998.
[50] D. LaSalle and G. Karypis, "Multi-threaded graph partitioning," in *Proc. IEEE 27th Int. Symp. Parallel Distrib. Processing*, 2013, pp. 225–236.
[51] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, 2011, Art. no. 1.
[52] The DIMACS Implementation Challenges. (2012). [Online]. Available: http://dimacs.rutgers.edu/Challenges/
[53] MKL Developer Reference. (2016). [Online]. Available: software.intel.com/en-us/node/520855
[54] MKL Forum. (2014). [Online]. Available: software.intel.com/en-us/forums/topic/508188
[55] E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms*. Rockville, MD, USA: Comput. Sci. Press, 1978.
[56] The Scalable Heterogeneous Computing Benchmark Suite (SHOC) for Intel Xeon Phi. (2013). [Online]. Available: https://github.com/vetter/shoc-mic
[57] E. D. Dolan and J. J. Moré, "Benchmarking optimization software with performance profiles," *Math. Prog.*, vol. 91, pp. 201–213, 2002.
[58] J. Treibig, G. Hager, and G. Wellein, "LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments," in *Proc. 39th Int. Conf. Parallel Process. Workshops*, Sep. 2010, pp. 207–216.



Kadir Akbudak received the BS degree from Hacettepe University, Turkey, and the MS and PhD degrees from Bilkent University, Turkey, all in computer engineering. His research interests mainly include parallel scientific computing.

Cevdet Aykanat received the the BS and MS degrees from METU, Turkey, and the PhD degree from Ohio State University, Columbus, in electrical and computer engineering. Since 1989, he has been in Computer Engineering Department, Bilkent University. His research interests mainly include parallel scientific computing.