Efficient and Effective Sparse Tensor Reordering

Jiajia Li
jiajia.li@pnnl.gov
Pacific Northwest National Laboratory
Richland, WA, USA

Bora Uçar
bora.ucar@ens-lyon.fr
UMR5668, CNRS and ENS Lyon
France

Ümit V. Çatalyürek
umit@gatech.edu
Georgia Institute of Technology
Atlanta, GA, USA

Jimeng Sun
jsun@cc.gatech.edu
Georgia Institute of Technology
Atlanta, GA, USA

Kevin Barker
kevin.barker@pnnl.gov
Pacific Northwest National Laboratory
Richland, WA, USA

Richard Vuduc
richie@cc.gatech.edu
Georgia Institute of Technology
Atlanta, GA, USA

ABSTRACT

This paper formalizes the problem of reordering a sparse tensor to improve the spatial and temporal locality of operations with it, and proposes two reordering algorithms for this problem, which we call BFS-MCS and Lexi-Order. The BFS-MCS method is a Breadth First Search (BFS)-like heuristic approach based on the maximum cardinality search family; Lexi-Order is an extension of doubly lexical ordering of matrices to tensors. We show the effects of these schemes within the context of a widely used tensor computation, the CANDECOMP/PARAFAC decomposition (CPD), when storing the tensor in three previously proposed sparse tensor formats: coordinate (COO), compressed sparse fiber (CSF), and hierarchical coordinate (HiCOO). A new partition-based superblock scheduling is also proposed for HiCOO format to improve load balance. On modern multicore CPUs, we show Lexi-Order obtains up to 4.14× speedup on sequential HiCOO-Mttkrp and 11.88× speedup on its parallel counterpart. The performance of COO- and CSF-based Mttkrps also improves. Our two reordering methods are more effective than state-of-the-art approaches. The code is released as part of Parallel Tensor Infrastructure (ParTI!): https://github.com/hpcgarage/ParTI.

KEYWORDS

Sparse tensor, tensor decomposition, reordering, hierarchical coordinate, HiCOO

ACM Reference Format:
Jiajia Li, Bora Uçar, Ümit V. Çatalyürek, Jimeng Sun, Kevin Barker, and Richard Vuduc. 2019. Efficient and Effective Sparse Tensor Reordering. In *2019 International Conference on Supercomputing (ICS '19), June 26–28, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3330345.3330366

1 INTRODUCTION

We consider the problem of how to improve memory reference locality for *sparse tensor computations* [4, 19]. A tensor is a multiway (N-way) array, with a vector and a matrix being the 1-way and 2-way (rows and columns) specializations thereof. It is *sparse* if most of its entries are zero and, therefore, need not be stored nor explicitly computed upon. A sparse tensor is often a natural way to represent a multifactor or multirelational dataset, and has found numerous applications in data analysis and mining [4, 7, 9, 9, 11, 13, 14, 18, 19, 21, 25, 27, 33, 33, 34, 37, 40, 43, 44, 44] for health care [14, 47], natural language processing [16], machine learning [1, 3, 30], and social network analytics [32], among many others.

Like sparse matrices, there are a variety of data structures and techniques for storing and operating efficiently with sparse tensors [4, 24, 25, 41]. Moreover, one technique is to *reorder* the tensor, which means relabeling the indices—and, therefore, reorganizing the nonzero structure of the tensor—in the hopes of improving spatial or temporal locality when operating with it. One form of reordering explored in prior work is to "sort" the input tensor [18, 25, 44]. In this paper, we formalize the reordering problem and propose and evaluate new heuristic algorithms for it.

While tensor reordering can improve locality, it can also aggravate load balance as measured by nonzeros assigned to each thread. We observe that reordering increases imbalance by as much as 6.7× in practice. In this paper, we propose improvements to the data structure and corresponding algorithms that address this problem.

Our evaluation is in the context of a widely used tensor computation known as the CANDECOMP/PARAFAC decomposition (CPD), a kind of generalization of the (truncated) singular value decomposition (SVD) from matrices to tensors [19]. Within a CPD, the most costly computational kernel is the *matricized tensor-times-Khatri-Rao product*, or Mttkrp [19, 24, 25, 44]. Thus, we examine the effects of reordering on the locality of sparse Mttkrp operations, which we then evaluate within a CPD on a modern multicore CPU platform. We also consider this evaluation when using three state-of-the-art tensor storage formats, known as coordinate (COO), compressed sparse fiber (CSF), and hierarchical COO (HiCOO). Thus, this evaluation gives practitioners a sense of when reordering pays off in commonly occurring scenarios.

The main claimed contributions of this work are as follows.

- We propose two heuristic tensor reordering schemes, BFS-MCS and Lexi-Order, to enhance data locality by relabeling mode indices. (§ 3)

ICS '19, June 26–28, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6079-1/19/06…$15.00
https://doi.org/10.1145/3330345.3330366

- We improve the state-of-the-art parallel strategy, superblock scheduling [24], by suggesting a partition-based version thereof. (§ 4)
- Lexi-Order achieves up to 4.14× speedup on sequential HiCOO-Mttkrp and 11.88× speedup on its parallel counterpart. BFS-MCS obtains up to 1.88× speedup on sequential HiCOO-Mttkrp and 1.94× speedup on its parallel case. (§ 5)
- For COO and CSF formats, Lexi-Order improves the performance of sequential COO- and CSF-Mttkrps by up to 4.29× and 2.33× and parallel COO- and CSF-Mttkrps by up to 1.48× and 1.86× respectively. BFS-MCS and Lexi-Order behave better than the state-of-the-art reordering based on graph and hypergraph partitionings [44] on CSF-Mttkrp. (§ 5)
- We further parallelize the reordering and HiCOO conversion to reduce the pre-processing overhead. When CPD iterates for tens of times, which is common, then the reordering is worthwhile. (§ 5)

Table 1: List of symbols and notation.

Symbols	Description
Data representations	
$\mathcal{X}$	A sparse tensor
$\mathbf{X}_{(n)}$	Matricized tensor $\mathcal{X}$ in mode-n
$\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$, $\mathbf{U}$	Dense matrices
$\mathbf{a_r}$, $\mathbf{b_r}$, $\mathbf{c_r}$	Dense vectors
$\boldsymbol{\lambda}$	Weight vector
Operations	
$\mathbf{A} \odot \mathbf{B}$	Khatri-Rao product between two matrices
$\mathbf{a} \circ \mathbf{b}$	Outer product between two vectors
Input parameters	
N	Tensor order
I, J, K, I_n	Tensor mode sizes
M	#Nonzeros of the input tensor $\mathcal{X}$
R	Approximate tensor rank (usually a small value)
Word sizes	
β_{int}	Bit-length of an integer
β_{long}	Bit-length of a long integer
β_{byte}	Bit-length of a byte or character
β_{float}	Bit-length of a single-precision floating point value
Data structures	
inds	Indices of COO, β_{int} bits
cinds	Indices of CSF, β_{int} bits
cptrs	Pointers of CSF, β_{int} or β_{long} bits
einds	Element indices of HiCOO, β_{byte} bits
binds	Block indices of HiCOO, β_{int} bits
bptr	Block pointers of HiCOO, β_{long} bits
lptr	Superblock pointers of HiCOO, β_{long} bits
lschr	Superblock scheduler of HiCOO, β_{int} bits
blschr	Partition-based superblock scheduler of HiCOO, β_{int} bits
plschr	Partition pointers of blschr, β_{int} bits
val	Nonzero value array of COO, CSF, HiCOO, β_{float} bits
Performance parameters	
perm_n	Permutation for a given mode n
L	Tensor superblock size
B	Tensor block size, $B \ll L$
α_b	Block ratio, $\alpha_b = \frac{n_b}{M}$, n_b is #Nonzero tensor blocks
$\overline{M_b}$	Geometric mean of #Nonzeros per tensor block
$\overline{c_b}$	Average slice size per tensor block, $\overline{c_b} = \frac{\overline{M_b}}{B}$
P	#Physical CPU cores

2 BACKGROUND

We summarize the related symbols and notation of tensors, CPD, and the three state-of-the-art tensor formats (COO, CSF, HiCOO) in Table 1.

2.1 Tensors and CPD

The *order* of a tensor, N, is the number of its dimensions or *modes*. We follow the notation in Kolda and Bader's survey [19]. A first-order tensor ($N = 1$) is a vector, denoted by a boldface lowercase letter, e.g., $\mathbf{v}$; A second-order tensor ($N = 2$) is a matrix, denoted by a boldface capital letter, e.g., $\mathbf{A}$. Higher-order tensors ($N \geq 3$) are denoted by bold capital calligraphic letters, e.g., $\mathcal{X}$. We show an example of a sparse third-order tensor, $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$, in Figure 1. In this example, a scalar entry of $\mathcal{X}$ at position (i, j, k) is x_{ijk}. We assume an example sparse tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ with M nonzeros in the following context.

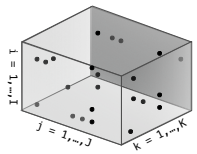


Figure 1: A third-order tensor $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$.

CPD decomposes a tensor into a sum of component rank-one tensors [19]. It approximates an Nth-order tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ as

$$\mathcal{X} \approx \sum_{r=1}^{R} \lambda_r \mathbf{a}_r^{(1)} \circ \cdots \circ \mathbf{a}_r^{(N)} \equiv [\![\boldsymbol{\lambda}; \mathbf{A}^{(1)}, \ldots, \mathbf{A}^{(N)}]\!], \tag{1}$$

where R is the canonical rank of tensor $\mathcal{X}$, taken as the number of component rank-one tensors [19]. In a low-rank approximation, R is usually chosen to be a small number less than 100. The outer product of the vectors $\mathbf{a}_r^{(1)}, \ldots, \mathbf{a}_r^{(N)}$ produces R rank-one tensors, and $\mathbf{A}^{(n)} \in \mathbb{R}^{I_n \times R}, n = 1, \ldots, N$ are the *factor matrices*, each one formed by taking the corresponding vectors as its columns, i.e., $\mathbf{A}^{(n)} = [\mathbf{a}_1^{(n)}\ \mathbf{a}_2^{(n)} \ldots \mathbf{a}_R^{(n)}]$. We normalize these vectors to unit magnitude and store the factor weights in the vector $\boldsymbol{\lambda} = \{\lambda_1, \ldots, \lambda_r\}$.

The bottleneck of CPD is the matricized tensor-times-Khatri-Rao product (Mttkrp). Given an Nth-order tensor $\mathcal{X}$ and matrices $\mathbf{A}^{(1)}, \ldots, \mathbf{A}^{(N)}$, the mode-$n$ Mttkrp is

$$\tilde{\mathbf{A}}^{(n)} \leftarrow \mathbf{X}_{(n)} \left(\mathbf{A}^{(N)} \odot \cdots \odot \mathbf{A}^{(n+1)} \odot \mathbf{A}^{(n-1)} \odot \cdots \odot \mathbf{A}^{(1)} \right), \tag{2}$$

where $\mathbf{X}_{(n)}$ is the mode-n matricization (or unfolding) of tensor $\mathcal{X}$, $\odot$ is the Khatri-Rao product. Mode-n matricization reshapes a tensor into an equivalent matrix [19]. The Khatri-Rao product is a "matching column-wise" Kronecker product between two matrices. Given matrices $\mathbf{A} \in \mathbb{R}^{I \times R}$ and $\mathbf{B} \in \mathbb{R}^{J \times R}$, their Khatri-Rao product is denoted by $\mathbf{C} = \mathbf{A} \odot \mathbf{B}$ where $\mathbf{C} \in \mathbb{R}^{(IJ) \times R}$,

$$\mathbf{C} = \mathbf{A} \odot \mathbf{B} = [\mathbf{a}_1 \circ \mathbf{b}_1, \mathbf{a}_2 \circ \mathbf{b}_2, \ldots, \mathbf{a}_R \circ \mathbf{b}_R], \tag{3}$$

where $\mathbf{a}_r$ and $\mathbf{b}_r$, $r = 1, \ldots, R$ are columns of $\mathbf{A}$ and $\mathbf{B}$, $\circ$ is the outer product of vectors, a special case of Kronecker product [19]. In particular for a third-order tensor, Mttkrp multiples each nonzero entry $x_{i,j,k}$ with the R-vector formed by the entry-wise product of the jth row of $\mathbf{A}^{(2)}$ and kth row of $\mathbf{A}^{(3)}$ when computing $\tilde{\mathbf{A}}^{(1)}$. Detailed description can be found in Kolda and Bader's survey [19]. In this paper, *sparse Mttkrp* will mean an Mttkrp between a sparse tensor and dense matrices.

2.2 Sparse Tensor Formats

We consider the three state-of-the-art tensor formats (COO, CSF, HiCOO) which are all for general unstructured sparse tensors.

2.2.1 COO. The coordinate (COO) format is the simplest yet arguably most popular format by far. It stores each nonzero value along with all of its position indices, shown in Figure 2 (a). i, j, k

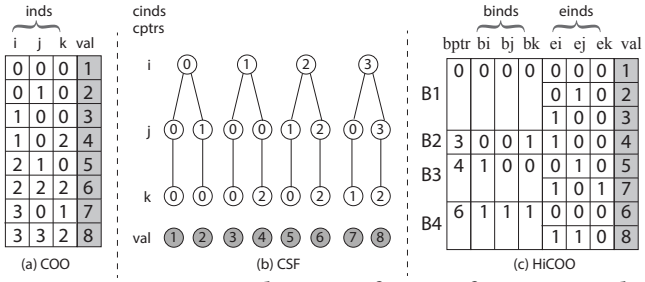


Figure 2: COO, CSF, and HiCOO formats for an example third-order tensor (This example is originally from [24]).

are indices (inds) of the nonzeros stored in the val array. Generally, 32-bit (β_{int}) integers are large enough to represent tensor indices.

2.2.2 CSF. Compressed Sparse Fiber (CSF) is a hierarchical, fiber-centric format that effectively generalizes the CSR matrix format to tensors. An example of its representation appears in Figure 2 (b). Conceptually, CSF organizes nonzero indices into trees (a forest). Each level corresponds to a tensor mode, and each nonzero is a path from a root to a leaf. The indices of CSF are stored in cinds with the pointers stored in cptrs to indicate the locations of nonzeros at the next level. Since cptrs needs to show the range up to M, we use β_{int} or β_{long} accordingly for differently sized-tensors .

2.2.3 HiCOO. Hierarchical Coordinate (HiCOO) [24] format derives from COO format, but improves upon it by compressing the indices in units of sparse tensor blocks. HiCOO stores a sparse tensor in a sparse-blocked pattern with a pre-specified block size B, meaning in $B\times\cdots\times B$ blocks. It represents every block by compactly storing its nonzero triples using fewer bits. Figure 2 (c) shows the example tensor given $2\times 2\times 2$ blocks ($B=2$). For a third-order tensor, bi, bj, bk are *block indices* in β_{int} bits, indexing tensor blocks; ei, ej, ek are *element indices* in β_{byte} bits, indexing nonzeros within a tensor block. A bptr array in β_{long} bits stores the pointers of every block's beginning locations, and val saves all the nonzero values.

Compare the three tensor formats: HiCOO and COO treat every mode equally and do not assume any mode order, these preserve the mode-generic orientation [24]. CSF has a strong mode-specificity, since the tree structure implies a mode ordering for enumeration of nonzeros. Besides, compared to COO format, HiCOO and CSF save storage space and memory footprints whereas achieve higher performance generally.

3 PROBLEM DEFINITION AND OUR SOLUTIONS

Our objective is to improve the performance of Mttkrp and therefore CPD algorithms based on the three tensor storage formats described above. We will achieve this objective by reordering (or relabeling) the indices in one or more modes of the input tensor so as to improve the data locality in the tensor and the factor matrices of an Mttkrp operation. Take for example two nonzeros (i_2, j_2, k_1) and (i_2, j_2, k_2) of a third-order tensor in Figure 3 (a). Relabeling k_1 and k_2 to other two indices close to each other will potentially improve cache hits for both tensor and their corresponding rows of factor matrices in the Mttkrp operation [24, 44]. Besides, it may also influence the tensor storage for some formats. (Analysis will be illustrated in § 3.3.)

We propose two heuristics for reordering indices. The aim of the heuristics is to arrange the nonzeros close to each other, in all modes. If we were to look at matrices, this would correspond to reordering the rows and columns so that all nonzeros are clustered around the diagonal. This way, nonzeros in a row or column would be close to each other, and any blocking (by imposing fixed sized blocks) would have nonzero blocks only around the diagonal. The proposed heuristics are based on these observations and try to obtain similar behavior for tensors. The output of a reordering algorithm is the permutations for all modes being used to relabel tensor indices of them.

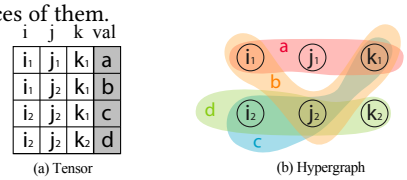


Figure 3: A hypergraph example of a sparse tensor.

3.1 BFS-MCS

BFS-MCS is a Breadth First Search (BFS)-like heuristic approach based on the maximum cardinality search family [46]. We first construct a hypergraph for a sparse tensor, where vertices are tensor indices in all modes and hyperedges represent its nonzero entries. For a third-order sparse tensor $\mathcal{X}\in\mathbb{R}^{I_1\times I_2\times I_3}$ with M nonzeros, its hypergraph $H=(V,E)$ consists of $|V|=I_1+I_2+I_3$ vertices and $|E|=M$ hyperedges. A nonzero entry $x_{i_1 i_2 i_3}$ connects the three vertices i_1, i_2, i_3. Figure 3 (b) shows an example of the hypergraph for a sparse tensor. Vertices are blank circles, and hyperedges are represented by grouping vertices.

For an Nth-order sparse tensor, we need to find the permutations for N modes. We determine a permutation for a given mode n (perm_n) of a tensor $\mathcal{X}\in\mathbb{R}^{I_1\times\cdots\times I_N}$ as follows. Suppose some of the indices of mode n are already ordered. Then, BFS-MCS picks the next to-be-ordered index as the one with the strongest connection to the currently ordered index set. In the case of ties, it selects the index with the smallest number of nonzeros in the corresponding sub-tensor. Intuitively, in the hypergraph of a sparse tensor, a stronger connection represents more common indices in the modes other than n among an unordered vertex and the already ordered ones. This means more data from factor matrices can be reused in Mttkrp, if found in cache.

This process is implemented by maintaining a max-heap (H_n, shown in Line 3 in Algorithm 1) for all mode-n indices (e.g., i_n) with two keys. The primary key of an index is the number of connections to the currently ordered indices. The secondary key is the number of nonzeros in the corresponding $(N-1)$th-order sub-tensor, e.g., $\mathcal{X}(:,\ldots,:,i_n,:,\ldots,:)$, where the smaller secondary key values signify higher priority. Note that the secondary keys are static. Initially, H_n is constructed according to the secondary keys of mode-n indices. For each mode-n index (e.g., i_n) of this max-heap, BFS-MCS traverses all connected tensor indices in the modes except n, calculates the number of connections of mode-n indices, and then updates the heap (H_n) using the primary and secondary keys.

Algorithm 1 makes BFS-MCS more precise. Let $\mathbf{m}_V$ denote a size-$|V|$ array that tracks whether a vertex has been visited. They are

Algorithm 1 BFS-MCS ordering based on maximum cardinality search for a given mode.

Input: An Nth-order sparse tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$, hypergraph $G = (V, E)$ with vertex weights $\mathbf{w}$, mode n;
Output: Permutation $perm_n$;
1: Initialize $\mathbf{w}_p$ to zeros.
2: Initialize $\mathbf{w}_s$ to the minus of number of nonzeros of each corresponding sub-tensor as a static secondary key.
3: Build max-heap H_n for mode-n indices with $\mathbf{w}_p$ and $\mathbf{w}_s$ as the primary and secondary keys respectively.
4: Initialize $\mathbf{m}_V$ to zeros.
5: **for** $i_n = 1, \ldots, I_n$ **do**
6: $v_n^{(0)}$ = GetHeapMax(H_n);
7: $perm_n(v_n^{(0)}) = i_n$;
8: $\mathbf{m}_V(v_n^{(0)}) = 1$;
9: **for** $e_n \in$ hyperedges of $v_n^{(0)}$ **do**
10: **for** $v \in e_n$, v is not in mode n and $\mathbf{m}_V(v) == 0$ **do**
11: $\mathbf{m}_V(v) = 1$;
12: **for** $e \in$ hyperedges of v and $e \neq e_n$ **do**
13: v_n = vertex in mode n of e;
14: **if** inHeap(v_n) **then**
15: $\mathbf{w}_p(v_n) + +$;
16: heapUpdateKey(H_n, $\mathbf{w}_p(v_n)$);
17: **return** $perm_n$;

initialized to zeros (unvisited) and changed to 1s once after being visited. We record a mode-n index $v_n^{(0)}$, obtained from the max-heap H_n, to the permutation array perm_n. The algorithm visits all its hyperedges (i.e., all nonzeros with i_n, Line 9) and their connected and unvisited vertices from the other $(N-1)$ modes except n (Line 10). For these vertices, it again visits their hyperedges (e in Line 12) and then checks if the connected vertices in mode n (v_n) are in the heap (Line 14). If so, the primary key (connectivity) of v_n is increased by 1 and the max-heap (H_n) is updated. To summarize, this algorithm locates the sub-tensors of the neighbor vertices of $v_n^{(0)}$ to increase the primary key values of the occurred mode-n indices in H_n.

The BFS-MCS heuristic has low time complexity. We explain it using tensor terms. The innermost heap update operation (Line 16) costs $O(\log(I_n))$. Each sub-tensor is only visited once with the help of the marker array $\mathbf{m}_V$. For all the sub-tensors in one tensor mode, the heap update is only performed for M nonzeros (hyperedges). Overall, the time complexity of BFS-MCS is $O(NM\log(I_n))$ for an Nth-order tensor with M nonzeros, when computing perm_n for mode n.

BFS-MCS does not exactly catch the memory access pattern of an Mttkrp. It treats the contribution from the indices of all modes except n equally to the connectivity, thus the connectivity might not match the actual data reuse. Also, it uses a greedy strategy to determine the next-level vertices, which could miss the optimal global orderings, as is common to greedy heuristics for hard problems.

3.2 Lexi-Order

Lexi-Order is an extension of doubly lexical ordering of matrices [26, 31] to tensors. A lexicographic ordering of an integer vector is the standard dictionary ordering of its elements, defined as follows. Given two equal-length vectors, $\mathbf{x}$ and $\mathbf{y}$, we say $\mathbf{x} \leq \mathbf{y}$ iff either (i) all elements are the same, i.e., $\mathbf{x} = \mathbf{y}$; or (ii) there exists an index j such that $\mathbf{x}(j) < \mathbf{y}(j)$ and $\mathbf{x}(i) = \mathbf{y}(i)$ for all $0 \leq i < j$. For example, Figure 4 (a) shows the vector comparison of two zero-one vectors, $\mathbf{x} \equiv (1, 0, 1, 0)$ and $\mathbf{y} \equiv (1, 1, 0, 0)$. $\mathbf{x} \leq \mathbf{y}$ because $\mathbf{x}(1) < \mathbf{y}(1)$ and $\mathbf{x}(i) = \mathbf{y}(i)$ for all $0 \leq i < 1$.

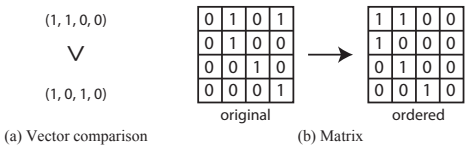


Figure 4: A doubly lexical ordering of a zero-one matrix and its vector comparison operation.

A *doubly lexical ordering* of a matrix [26] is an ordering of the rows and columns of the matrix so that both the row and column vectors are in non-increasing lexicographic order. A row vector is read from left to right, and a column vector is read from top to bottom. Every real-valued matrix has a doubly lexicographic ordering, and such an ordering is not unique (the proof can be found in [26]). Figure 4 (b) shows a doubly lexical ordering of a zero-one example matrix. The row vectors are in non-increasing lexicographic order from top to bottom, and the column vectors are in non-increasing lexicographic order from left to right. Assume the ordered matrix in Figure 4 (b) is $\mathbf{A}$, then rows $\mathbf{a}(1,:) \geq \mathbf{a}(2,:) \geq \mathbf{a}(3,:) \geq \mathbf{a}(4,:)$ and columns $\mathbf{a}(:,1) \geq \mathbf{a}(:,2) \geq \mathbf{a}(:,3) \geq \mathbf{a}(:,4)$. Even from this simple example, the ordered matrix shows better data locality than the original one. Doubly lexical ordering has a number of applications, including the efficient recognition of totally balanced, subtree, and plaid matrices and (strongly) chordal graphs. To the best of our knowledge, the merits of this ordering for high performance have not been investigated for sparse matrices.

We propose an iterative algorithm called `matLexiOrder` for doubly lexical ordering of matrices, where in an iteration either rows or columns are sorted. The known doubly lexical ordering algorithms for matrices, which were first explored by Lubiw [26] and then improved by Paige and Tarjan [31], are "direct" ordering methods with a non-linear runtime of $O(M\log(I+J)+J)$ and $O(M+I+J)$ space, for an $I \times J$ matrix with M nonzeros. We find the time complexity of these algorithms to be too high for our purpose. Furthermore, the data structures are too complex to allow an efficient generalized implementation for tensors. By appealing to a result of Lubiw [26, Claim 2.2], one can show that our `matLexiOrder` algorithm finds a doubly lexical ordering in a finite number of iterations. We do not show the proof, since we do not aim to obtain an exact doubly lexical ordering; a close-by ordering will likely suffice to improve the Mttkrp performance.

We first describe `matLexiOrder` algorithm for a sparse matrix. This aims to show the efficiency of our iterative approach compared to others. A partition refinement technique is used to order the rows and columns alternatively. Given an ordering of the rows, the columns can be sorted lexically. This is achieved by an order preserving variant of the partition refinement method [31], which is called `orderlyRefine`. We briefly explain the partition refinement technique for ordering the columns of a matrix. Given an $I \times J$ matrix $\mathbf{A}$, all columns are initially put into a single part. Then, $\mathbf{A}$'s nonzeros are visited row by row. At a row i, each column part C is split into two parts $C_1 = C \cap \mathbf{a}(i,:)$ and $C_2 = C \setminus \mathbf{a}(i,:)$, and

these two parts replace C in the order $C_1 \succ C_2$ (empty sets are discarded). Note that this algorithm keeps the parts in a particular order which generates an ordering of the columns. `orderlyRefine` is used to refine all parts that have at least one nonzero in row i in $O(|\mathbf{a}(i,:)|)$ time, where $|\mathbf{a}(i,:)|$ is the number of nonzeros of row i. Overall, `matLexiOrder` costs a linear total time of $O(M+I+J)$ (for rows and columns ordering) per iteration and $O(J)$ space. We also observe that only a small number of iterations will be enough (will be shown in § 5.8 for tensors), yielding a more storage-efficient algorithm compared to the prior doubly lexical ordering methods [26, 31]. `matLexiOrder`, in particular the use of `orderlyRefine` routine a few times, is sufficient for our needs. This is so, as we do not need a full lexicographic order, thereby making our approach faster and simpler to implement.

Algorithm 2 Lexi-Order for a given mode.

Input: An Nth-order sparse tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$, mode n;
Output: Permutation perm_n;

▷ Sort all nonzeros along with all but mode n.
1: quickSort($\mathcal{X}$, coordCmp);
▷ Matricize $\mathcal{X}$ to $\mathbf{X}_{(n)}$.
2: r = compose (inds ([$-n$]), 1);
3: **for** $m = 1, \ldots, M$ **do**
4: c = inds(n, m); ▷ Column index of $\mathbf{X}_{(n)}$
5: **if** coordCmp($\mathcal{X}$, m, $m-1$) == 1 **then**
6: r = compose (inds ([$-n$]), m); ▷ Row index of $\mathbf{X}_{(n)}$
7: $\mathbf{X}_{(n)}(r, c)$ = val(m);
▷ Use a variation of partition refinement in [31]
8: perm_n = `orderlyRefine` ($\mathbf{X}_{(n)}$);
9: **return** perm_n;
▷ Comparison function for two indices of $\mathcal{X}$
10: Function: coordCmp($\mathcal{X}$, m_1, m_2)
11: **for** $n' = 1, \ldots, N$ **do**
12: **if** $n' != n$ **then**
13: **if** $m_1(n') < m_2(n')$ **then**
14: return -1; ▷ Entry m_1 < entry m_2
15: **if** $m_1(n') > m_2(n')$ **then**
16: return 1; ▷ Entry m_1 > entry m_2
17: **return** 0; ▷ Entry m_1 = entry m_2

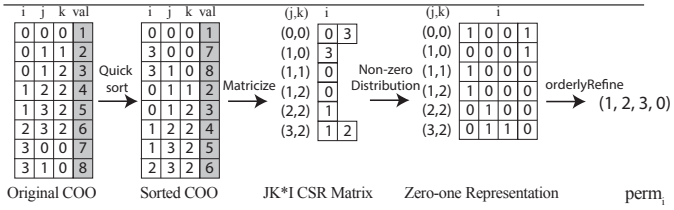


Figure 5: The steps of Lexi-Order illustrated for mode 1.

To order tensors, we propose the Lexi-Order function as an extension of `matLexiOrder`. The basic idea of Lexi-Order is to determine the permutation of each tensor mode independently, while considering the order in other modes fixed. Lexi-Order sets the indices of the mode to be ordered as the columns of a matrix, the other indices as the rows and sorts the columns as described for matrices (with the order preserving partition refinement method). The precise algorithm appears in Algorithm 2, which we also illustrate in Figure 5 when applied to mode 1. Given a mode n, Lexi-Order first builds a matricized tensor in Compressed Sparse Row (CSR) sparse matrix format by a call to `quickSort` with the comparison function `coordCmp` and then by partitioning the nonzeros into the row segments (Lines 3–7). This comparison function `coordCmp` does a lexicographic comparison of all-but-mode-n indices, which enables efficient matricization. In other words, sorting of the tensor $\mathcal{X}$ is the same as building the matricized tensor $\mathbf{X}_{(n)}$ by rows in the fixed lexical ordering, where mode n is the column dimension and the remaining modes constitute the rows. In Figure 5, the sorting step orders the COO entries by (j,k) tuples, which then serve as the row indices of the matricized CSR representation of $\mathbf{X}_{(1)}$. Once the matrix is built, we construct zero-one row vectors in Figure 5 to illustrate its nonzero distribution, which could seamlessly call `orderlyRefine` function. Apart from the `quickSort`, the other parts of Lexi-Order are of linear time complexity (linear in terms of tensor storage). We use OpenMP Tasks to parallelize `quickSort` to accelerate Lexi-Order.

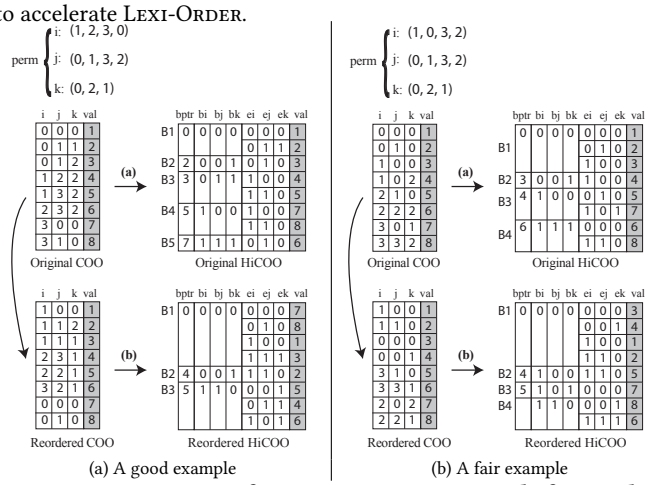


Figure 6: Comparison of HiCOO representations before and after Lexi-Order.

Like BFS-MCS approach, Lexi-Order also finds the permutations for N modes of an Nth-order sparse tensor. Figure 6 (a) illustrates the effect of Lexi-Order on an example $4 \times 4 \times 3$ sparse tensor. The original tensor is converted to a HiCOO representation with block size $B = 2$ consisting of 5 blocks, with maximum 2 nonzeros per block. After reordering with Lexi-Order, the new HiCOO has 3 nonzero blocks with up to 4 nonzeros per block. Thus, the blocks are denser, which should exhibit better locality behavior. However, this reordering scheme is heuristic. Figure 6 (b) shows HiCOO representation after reordering of the tensor shown in Figure 2. The number of blocks is unchanged (4), although the maximum number of nonzeros per block increases to 4. For this tensor, Lexi-Order may not show a big advantage.

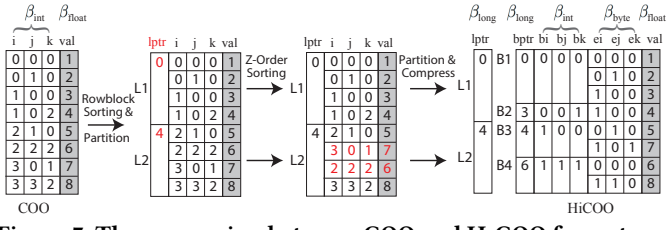


Figure 7: The conversion between COO and HiCOO formats for an example third-order tensor. HiCOO uses 2×2×2 blocks ($B = 2$) with word sizes marked above.

3.3 Analysis

We take MTTKRP operation to analyze reordering behavior for COO, CSF, and HiCOO formats. Recall that for a third-order sparse tensor $\mathcal{X}$, MTTKRP multiples each nonzero entry $x_{i,j,k}$ with the R-vector formed by the entry-wise product of the jth row of $\mathbf{A}^{(2)}$ and kth row of $\mathbf{A}^{(3)}$ when computing $\tilde{\mathbf{A}}^{(1)}$. The arithmetic intensity of MTTKRP algorithms on these formats is approximately 1/4 [24]. Thus, MTTKRP can be considered memory-bound for most computer architectures, especially CPU platforms.

We analyze the memory access of HiCOO-MTTKRP algorithm in bytes (Equation (4), details in Equation (16) of [24]) to roughly reflect its real performance. $\beta_?$ is different bit-lengths for HiCOO (Table 1). The block ratio (α_b) and the average slice size per tensor block ($\overline{c_b}$) are two critical parameters of HiCOO. Smaller α_b and larger $\overline{c_b}$ are favorable for good HiCOO-MTTKRP performance. (Readers could refer to [24] for more explanation.) Our reordering algorithms tend to closely pack nonzeros together and get denser blocks (larger $\overline{c_b}$), which potentially generates less tensor blocks (smaller α_b), thus HiCOO-MTTKRP has less memory access, more cache hits, and better performance further. Take for example two nonzeros (i_2, j_2, k_1) and (i_2, j_2, k_2) again. A HiCOO representation after reordering could combine them in a single block. (A detailed example is shown in Figure 6.) Moreover, from the HiCOO storage equation (Equation (5), Equation (13) from [24]), smaller α_b can also reduce tensor memory requirement as a side benefit. That is, for HiCOO format a reordering scheme should increase the block density, reduce the number of blocks, and increase cache locality—three related performance metrics. Reordering is more beneficial for HiCOO format than COO and CSF formats.

$$\text{Bytes}_{\text{hicoo}} \approx \frac{M}{8}[2\alpha_b \cdot \beta_{\text{int}} + \beta_{\text{byte}} + R\min\{\frac{1}{\overline{c_b}}, 1\} \cdot \beta_{\text{float}}]N \tag{4}$$

$$S_{\text{hicoo}} \approx M[\alpha_b \cdot \beta_{\text{long}} + \alpha_b N \cdot \beta_{\text{int}} + N \cdot \beta_{\text{byte}}] \tag{5}$$

COO stores all nonzero indices, so relabeling does not make a difference in its storage. The same is also true for CSF; relabeling does not change the its tree structure, while the order of nodes may change. For an MTTKRP with tensors stored in these two formats, the performance gain from reordering is only the improved data locality and therefore cache behavior. Though it is hard to do theoretical analysis for them, the potential better data locality could also brings performance advantages, but might be less than HiCOO's.

3.4 Parallelize Preprocessing

For an efficient parallel HiCOO-MTTKRP algorithm, *superblocks*, an extra blocking level above blocks, are proposed to increase the workload granularity of scheduling [24]. A superblock is essentially a "logical" subtensor that can potentially consist of many small blocks. During HiCOO format conversion in Figure 7, we first sort all nonzeros only by i indices in mode 1 ("rowblock sorting"), to ensure that a mode-1 slice is not split between superblocks. Then $L \times \cdots \times L$ nonzero superblocks are partitioned with an additional array lptr to store the beginning pointers of nonzero superblocks in size n_l, the number of superblocks. We treat them as independent subtensors to convert each to the physical HiCOO format with $B \times \cdots \times B$ blocks ($L \geq B$) through Z-Morton order sorting, partitioning, and compression steps. In the first rowblock sorting we do a quicksort for a single mode where the quicksort is parallelized. Afterwards, the operations on each superblock (Z-order sorting, partitioning, and compression) are naturally parallelized.

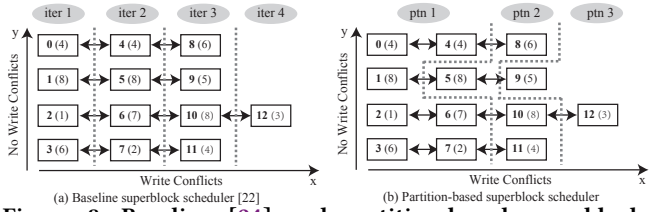


Figure 8: Baseline [24] and partition-based superblock scheduling tables for mode-1 MTTKRP. Each box represents a superblock identified by its number, along with its number of nonzeros in parentheses.

4 HYBRID PARALLEL HICOO-MTTKRP

An important side-effect of the proposed reordering schemes is that it can inadvertently create load imbalance, if one assumes the original "superblock scheduling" scheme proposed for HiCOO [24]. This section summarizes the issue and proposes a modification to HiCOO's superblock scheduler to mitigate such imbalance.

The baseline HiCOO superblock scheduler operates schematically as shown in Figure 8 (a). Each box is a superblock, where the number of nonzeros inside the block is shown in parentheses. Superblocks in the same row all write to the same output area and so are dependent (arrows), while columns indicate independent superblocks. Two parallel strategies are employed in [24]: direct and privatized parallelization. Simply speaking, direct parallelization parallelizes rows while privatization strategy parallelizes iterations (or columns). Direct parallelization uses an owner-computes strategy to assign one (or more rows) to each available thread and then execute each column ("iteration") of independent superblocks in a bulk-synchronous fashion. Privatized parallelization uses thread-local buffers to keep the updates of the superblocks from distinct iterations, and then accumulates these buffers together in parallel at the end. The superblock scheduler also varies the superblocks-to-threads assignment from row-to-row or iteration-to-iteration if necessary for the two strategies respectively to achieve better load balance. These two parallel strategies are proposed to obviate the need for locks or atomic operations and to explore sufficient parallel degree for irregular sparse tensors.

However, neither of thee two strategies can obtain good load balance due to the uneven nonzero distribution of the superblocks in rows or columns, even under dynamic thread scheduling policy. This situation becomes worse after applying tensor reordering. We use the ratio of maximum number of nonzeros assigned to a thread to the average number of nonzeros per thread to represent the load balance. Under a random ordering, which might be expected to have good load balance, at the price of poor locality, most ratios are under 1.3. While after LEXI-ORDER reordering, using the baseline HiCOO superblock scheduler, the ratios become worse on most of tensors and raise to as high as 6.72, although it would be expected to have better locality. (Refer to details in Table 4.) Therefore, we are motivated to try to mitigate this imbalance.

Here is our scheme that tries heuristically to improve the balance ratio. The main idea behind our modified superblock scheduler is illustrated in Figure 8 (b). We simply aggregate superblocks in the

same row of the figure, which effectively creates "jagged" partition boundaries (partitions 1 and 2 in Figure 8 (b)), with any remainder superblocks in their own final "tail" partition (the last partition). To decide how many superblocks within each row may be aggregated at a time, there is an additional tuning parameter which is a target maximum number of nonzeros to aggregate at a time. In this example, that target is set to 8 nonzeros, superblocks 0 and 4 are aggregated, as are the pairs (2, 6) and (3, 7). Correspondingly, our scheme again uses two parallel strategies: direct and privatized parallelization. However, direct parallelization parallelizes rows in units of partitions, and privatization strategy parallelizes partitions by saving thread-local buffers. Under this aggregation scheme, the two partitions of this example combined represent a more balanced unit to schedule than the original superblock scheduler's units of rows or columns (iterations). The number of nonzeros of our new scheduling unit is around 8 for both row and partition scheduling whereas that of the baseline row and column scheduling units varies in the range of 1 - 8. Note that our partitioning scheme does not split any superblock, this may introduce some imbalance. Imbalance also occurs in the last partition for the remainders. We also observe that the number of partitions in Figure 8 (b) could be easily smaller than the number of iterations in Figure 8 (a). This could lead to insufficient parallel degree for privatization strategy.

We record these partition-based partitions as blschr, companied with plschr to record the partitioning positions. In practice, because the superblock sizes are relatively large, there are not many superblocks to schedule on our evaluation tensors. Therefore, it is not expected that these auxiliary metadata will increase the size of the overall HiCOO data structure.

5 EXPERIMENTS

5.1 Experimental Setup

Platform. We perform experiments on a Linux-based Intel Xeon E5-2698 v3 multicore server platform with 32 physical cores distributed on two sockets, each with 2.3 GHz frequency. The processor microarchitecture is Haswell, having 32 KiB L1 data cache and 128 GiB memory. The code artifact was written in the C language using OpenMP parallelization, and was compiled using icc 18.0.1.

Dataset. We use the sparse tensors, derived from real-world applications, that appear in Table 2, ordered by decreasing nonzero density separately for third- and fourth-order tensors. Most of these tensors are included in The Formidable Repository of Open Sparse Tensors and Tools (FROSTT) dataset [39]. The darpa (source IP-destination IP-time triples), fb-m, and fb-s (short for "freebase-music" and "freebase-sampled", entity-entity-relation triples) are from the dataset of HaTen2 [15], and choa is built from the electronic health records (EHRs) of pediatric patients at Children's Healthcare of Atlanta (CHOA) [34].

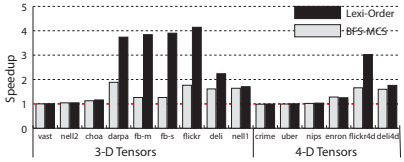
Configurations. We report the results under the best configurations for the following parameters for the highest Mttkrp performance with the three formats: (i) the number of reordering iterations, the superblock size L, and the block size B for HiCOO format; (ii) privatization or not for COO; and (iii) tiling or not for CSF. For HiCOO, $B = 128$ achieves the best results in most cases, and we use five reordering iterations which will be analyzed in § 5.8. All experiments use approximate rank of $R = 16$. The parallel experiments are run with 32 threads under dynamic scheduling strategy in units of one superblock/partition. We use the total execution time of Mttkrps in all modes for every tensor to calculate the speedup which is the ratio of the total Mttkrp time on a randomly reordered tensor over that using a specific reordering scheme. All the execution time is averaged over five runs.

Table 2: Description of sparse tensors.

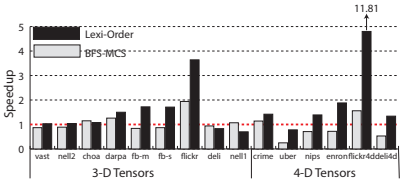
Tensors	Order	Dimensions	#Nnzs	Density
vast	3	$165K \times 11K \times 2$	26M	6.9×10^{-3}
nell2	3	$12K \times 9K \times 29K$	77M	2.4×10^{-5}
choa	3	$712K \times 10K \times 767$	27M	5.0×10^{-6}
darpa	3	$22K \times 22K \times 24M$	28M	2.4×10^{-9}
fb-m	3	$23M \times 23M \times 166$	100M	1.1×10^{-9}
fb-s	3	$39M \times 39M \times 532$	140M	1.7×10^{-10}
flickr	3	$320K \times 28M \times 2M$	113M	7.8×10^{-12}
deli	3	$533K \times 17M \times 3M$	140M	6.1×10^{-12}
nell1	3	$2.9M \times 2.1M \times 25M$	144M	9.1×10^{-13}
crime	4	$6K \times 24 \times 77 \times 32$	5M	1.5×10^{-2}
uber	4	$183 \times 24 \times 1140 \times 1717$	3M	3.9×10^{-4}
nips	4	$2K \times 3K \times 14K \times 17$	3M	1.8×10^{-6}
enron	4	$6K \times 6K \times 244K \times 1K$	54M	5.5×10^{-9}
flickr4d	4	$320K \times 28M \times 2M \times 731$	113M	1.1×10^{-14}
deli4d	4	$533K \times 17M \times 3M \times 1K$	140M	4.3×10^{-15}

5.2 HiCOO-Mttkrp with Reordering

Figure 9 (a) shows the speedup of the proposed reordering methods on sequential and parallel HiCOO-Mttkrp respectively. Lexi-Order reordering obtains 0.99–4.14× speedup (2.12× on average); while BFS-MCS reordering gets 0.99–1.88× speedup (1.34× on average). Lexi-Order and BFS-MCS do not behave as well on fourth-order tensors as on third-order tensors. Tensor flickr4d is constructed from the same data with flickr, with an extra short mode (refer to Table 2). Lexi-Order obtains 4.14× speedup on flickr while only 3.02× speedup on flickr4d. The same phenomenon is also observed on tensors deli and deli4d. This phenomenon indicates that it is harder to get good data locality on higher-order tensors, which will be justified in Table 3.



(a) Sequential



(b) Parallel

Figure 9: Reordered HiCOO-Mttkrp speedup over a random ordering implementation.

Figure 9 (b) shows the speedup of the proposed reordering methods on multicore parallel HiCOO-Mttkrp. We set the same superblock size L before and after reordering for a fair comparison. Overall, Lexi-Order results in 0.70–11.81× speedup (2.12× on average) for parallel HiCOO-Mttkrp; while BFS-MCS reordering gets 0.25–1.94× speedup (0.98× on average). Thus, the benefit from reordering using either Lexi-Order or BFS-MCS is generally less than that in the sequential case. This observation may indicate that

it is harder to pursue a good load balance after reordering. The 11.81× speedup achieved on tensor flick4d is because Lexi-Order changes the nonzero distribution and hence the optimal superblock size L. Thus, under the same L, reordering gives a big performance improvement. It indicates that automatically tuning the parameters of HiCOO will be very helpful.

5.3 HiCOO Parameters

We investigate two critical parameters of HiCOO [24]: the block ratio (α_b) and the average slice size per tensor block ($\overline{c_b}$). Smaller α_b and larger $\overline{c_b}$ are favorable for good HiCOO-Mttkrp performance. Table 3 lists the parameter values for all tensors before and after Lexi-Order, the HiCOO-Mttkrp speedup (as shown in Figure 9), and the storage ratio of HiCOO over random ordering. Generally, when α_b is reduced and $\overline{c_b}$ is increased using Lexi-Order for a tensor, we see a good performance speedup and storage ratio. (Sequential speedup reflects better by involving less factors than parallel cases.) For the same data in different orders, e.g., flickr4d and flickr, the α_b and $\overline{c_b}$ values are the same for random reordering, after Lexi-Order, these values of flickr are better than those of flickr4d. This fact justifies the phenomenon that getting good data locality is harder for higher-order tensors.

Table 3: HiCOO parameters change before and after Lexi-Order reordering.

Tensors	Random reordering		Lexi-Order		Speedup		Storage
	α_b	$\overline{c_b}$	α_b	$\overline{c_b}$	seq	omp	ratio
vast	0.004	1.758	0.004	1.562	1.01	1.03	0.999
nell2	0.020	0.314	0.008	0.074	1.04	1.04	0.966
choa	0.089	0.057	0.016	0.056	1.16	1.07	0.833
darpa	0.796	0.009	0.018	0.113	3.74	1.50	0.322
fb-m	0.985	0.008	0.086	0.021	3.84	1.21	0.335
fb-s	0.982	0.008	0.099	0.020	3.90	1.23	0.336
flickr	0.999	0.008	0.097	0.025	4.14	3.66	0.277
deli	0.988	0.008	0.501	0.010	2.24	0.83	0.634
nell1	0.998	0.008	0.744	0.009	1.70	0.70	0.812
crime	0.001	37.702	0.001	8.978	0.99	1.42	1.000
uber	0.041	0.469	0.011	0.270	1.00	0.78	0.838
nips	0.016	0.434	0.004	0.435	1.03	1.34	0.921
enron	0.290	0.017	0.045	0.030	1.25	1.36	0.573
flickr4d	0.999	0.008	0.148	0.020	3.02	11.81	0.214
deli4d	0.998	0.008	0.596	0.010	1.76	1.26	0.697

5.4 Partition-Based Scheduler

Table 4 shows the load balance under three scenarios and the corresponding performance speedup on parallel HiCOO-Mttkrp over random ordering. Load balance is represented by the ratio of maximum number of nonzeros assigned to a thread to the average number of nonzeros per thread [1], where lower numbers are favored. The target maximum number of nonzeros of a partition is set to five times of the maximum number of nonzeros per superblock. The second column shows the balance ratio under a random ordering, which might be expected to have good load balance, at the price of poor locality. Most ratios in this column are under 1.3. The third column shows the balance ratios under Lexi-Order reordering scheme, which increase for most of tensors, some even reach 6.72. Despite these balance ratios, HiCOO-Mttkrp on reordered tensors obtains speedup over random ordering on most of tensors, because of its better data locality. The resulting balance ratios under the partition-based scheduling appear in the fourth column. Only tensors vast, fb-m, fb-s, and nips get better balance ratios compared to reordering-only, but better performance is achieved on about half of tensors including these three. The improvement of the Reordered+Balanced scheme over Reordered-only is up to 42%. Two factors may harm the load balance in our new scheduling. One is whether we can get sufficient amount of partitions for parallelization; the other is that the cubical superblocks limits the parallelism for highly irregular tensors. Since the balanced superblock scheme does not always improve performance, our final scheduler selects between these two options.

[1] The same metric is used in [42].

Table 4: Load balance and its effect before and after Lexi-Order tensor reordering and partition-based scheduling.

Tensors	Max/Avg			Performance Speedup	
	Random	Reordered	Reordered +Partition	Reordered	Reordered +Partition
vast	1.25	1.21	1.14	1.03	1.01
nell2	1.18	1.24	1.36	1.04	1.04
choa	1.13	1.26	1.32	1.07	1.08
darpa	1.57	1.35	1.55	1.50	1.49
fb-m	1.16	2.07	1.70	1.21	1.72
fb-s	1.30	2.16	1.42	1.23	1.71
flickr	1.23	3.62	4.46	3.66	3.64
deli	1.28	5.00	5.90	0.83	0.82
nell1	1.28	6.72	7.10	0.70	0.70
crime	1.95	1.14	1.35	1.42	1.27
uber	1.29	1.70	1.79	0.78	0.78
nips	1.58	1.46	1.41	1.34	1.39
enron	1.19	1.61	1.65	1.36	1.88
flickr4d	1.15	2.66	2.68	11.81	10.85
deli4d	1.10	2.73	3.01	1.26	1.34

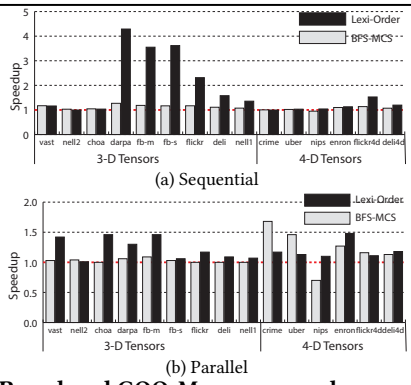


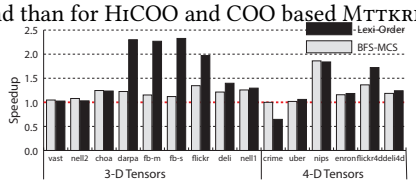
Figure 10: Reordered COO-Mttkrp speedup over a random reordering implementation.

5.5 Reordering Effect on Other Formats

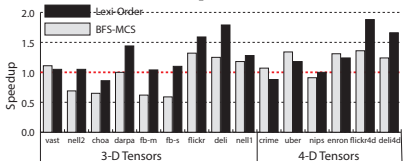
5.5.1 COO-Mttkrp with Reordering. We show the effect of the two reordering approaches on sequential and parallel COO-Mttkrp from ParTI! [22] in Figure 10. This COO-Mttkrp is implemented in C and OpenMP parallelized with or without privatization determined for different modes, using the same algorithm with Tensor Toolbox [5]. For any reordering approach, after doing a BFS-MCS, Lexi-Order, or random reordering on the input tensor, we still sort the tensor in the mode order of $1 \succ \cdots \succ N$. Observe that Lexi-Order improves sequential COO-Mttkrp performance by 1.00–4.29× (1.79× on average), while BFS-MCS gets 0.95–1.27× (1.10× on average). Also, Lexi-Order improves parallel COO-Mttkrp performance by 1.01–1.48× (1.21× on average), while BFS-MCS improves by 0.70–1.68× (1.11× on average). Note that Lexi-Order improves the performance of COO-Mttkrp for all tensors, both in sequential (79%) and parallel (21%). We conclude that this ordering is always helpful for COO-Mttkrp, while the improvements being less than what we saw for HiCOO-Mttkrp.

5.5.2 CSF-Mttkrp with Reordering. We show the effect of the two reordering approaches on sequential and parallel CSF-Mttkrp from

Splatt v1.1.1 [45] in Figure 11. CSF-Mttkrp is set to use all CSF representations (ALLMODE) for Mttkrps in all modes and with tiling option on. [2] Lexi-Order improves sequential CSF-Mttkrp performance by 0.65–2.33× (1.50× on average) and accelerates parallel CSF-Mttkrp by 0.86–1.88× (1.27× on average). BFS-MCS improves sequential CSF-Mttkrp performance by 1.00–1.86× (1.22× on average) and accelerates parallel CSF-Mttkrp by 0.59–1.36× (1.04× on average). Both ordering approaches improves the performance of CSF-Mttkrp on average. While BFS-MCS is always helpful in the sequential case, Lexi-Order is not helpful on only one tensor crime. The improvements achieved by the two reordering approaches for CSF are less than those for HiCOO and COO formats. We conclude that both reordering methods are helpful for CSF-Mttkrp, but in a lesser extend than for HiCOO and COO based Mttkrp.



(a) Sequential



(b) Parallel

Figure 11: Reordered CSF-Mttkrp speedup over a random reordering implementation.

5.5.3 Format Comparison. Figure 12 compares the parallel performance of COO-, CSF-, and HiCOO-Mttkrps all using Lexi-Order. The performance is shown by the speedup of each implementation to reordered COO-Mttkrp for every tensor. HiCOO achieves the best performance for most third-order tensors and is better than COO and CSF with tiling for most cases. In these experiments, we found that Splatt without tiling option is even faster than with tiling especially on fourth-order tensors. After reordering, HiCOO is 0.74–7.21× (3.08× on average) faster than COO and 0.25–28.84× (5.29× on average) faster than CSF-tiling. Compared to CSF with no tiling, HiCOO obtains 0.19 – 5.30× (1.40× on average) speedup. On flickr, deli, nell1, and enron data, HiCOO behaves worse than both the CSF settings because of the tensors' hypersparsity property. Although the reordering methods have improved the nonzero locality on them, due to their severe load imbalance in Table 4, their performance is still not comparable to CSF format. Overall, as also stated in [24], HiCOO works the best on short tensor modes and with reasonable sparsity of tensor blocks; while CSF is the complimentary for hypersparse tensors. [3]

5.6 Reordering Methods Comparison

As seen above, Lexi-Order improves performance more than BFS-MCS in most cases. Compared to the reordering method used in

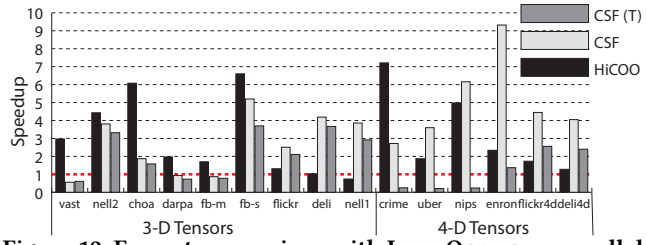


Figure 12: Format comparison with Lexi-Order on parallel Mttkrps.

Splatt [44], by setting ALLMODE (identical to the work [44]) to CSF-Mttkrp, BFS-MCS gets 1.04, 1.64, and 1.61× speedups on tensors nell2, nell1, and deli respectively, and Lexi-Order obtains 1.04, 1.70, and 2.24× speedups. By contrast, the speedups using graph partitioning [44] on these three tensors are 1.06, 1.11, and 1.19× and 1.06, 1.12, and 1.24× by using hypergraph partitioning [44] respectively. Our BFS-MCS and Lexi-Order schemes both outperform graph and hypergraph partitionings [44].

The available methods in the state-of-the-art are based on graph and hypergraph partitioning. Partitioning is a successful approach when the number of partitions is known, while the number of blocks in HiCOO is unknown ahead of time. In our case, the partitions should also be ordered for better cache reuse. Additionally, partitioners are less effective for tensors than their usual applications for matrices, as some dimensions could be very short (creating very high degree vertices). Thus, our proposed ordering based methods deliver better results.

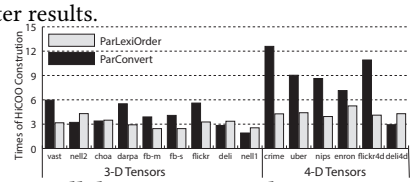


Figure 13: Parallel conversion and Lexi-Order speedup.

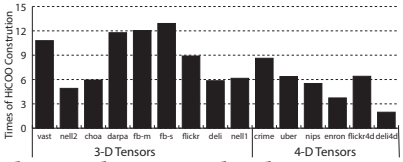


Figure 14: The reordering overhead over HiCOO construction.

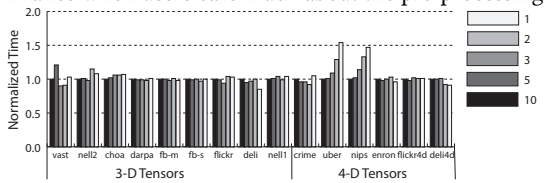
5.7 Reordering Overhead

Figure 13 shows the parallel speedup of Lexi-Order reordering and the HiCOO conversion steps respectively. Parallel Lexi-Order gains 3.61× speedup on average while parallel HiCOO conversion gets 5.84× average speedup on 32 physical cores. There is still room to further accelerate these parallel algorithms, which will be one of our future work. Figure 14 shows the overhead of Lexi-Order with 5 iterations. We specifically report the ratio of parallel Lexi-Order reordering time to parallel HiCOO construction time. (That is, we show how much more expensive it might be to reorder than to construct the HiCOO representation in the first place.) The results lie in the range of 1.97 to 12.91×. However, we could use less iterations for Lexi-Order to compromise some Mttkrp performance, which will be shown in Figure 15 (a). Thus, the reordering overhead can be further reduced.

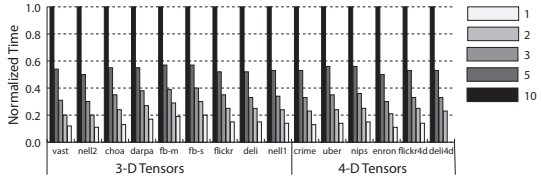
[2] Our tests using one CSF representation (ONEMODE) show quite similar results of reordering effect to the ALLMODE setting. Same for using tiling or no tiling options.
[3] HiCOO shows more performance advantage with more CPU cores as illustrated in [24].

5.8 Effect of the Number of Iterations in Lexi-Order

Since Lexi-Order improves the ordering iteratively, we evaluate the effect of the number of iterations on HiCOO-Mttkrp performance. The results appear in Figure 15 (a), which is normalized to the runtime of 10 iterations. Mttkrp on most tensors does not vary a lot by setting different number of iterations, except vast, nell2, uber, and nips. We use 5 iterations to get good Mttkrp performance similar to that of 10 iterations, with about half of the overhead (shown in Figure 15 (b)). But 3 or fewer iterations will get an acceptable performance when users care much about the pre-processing time.



(a) Sequential HiCOO-Mttkrp behavior



(b) The reordering overhead

Figure 15: The performance and overhead of different numbers of iterations.

5.9 CPD Application

Figure 16 shows the Lexi-Order effect on the parallel HiCOO-CPD algorithm using alternating least squares [19]. Similar to the numbers in Figure 9 (b), CPD after reordering achieves 0.65–10.44× speedup (1.81× on average). Reordering helps us to enhance the performance of a whole tensor application.

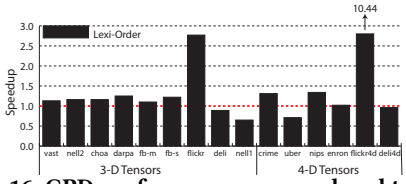


Figure 16: CPD performance on reordered tensors.

6 RELATED WORK

Plenty recent research studied the optimization of tensor algorithms [6, 8–10, 12, 13, 16, 19, 20, 23, 25, 27, 29, 38, 40]. Our work emphasizes on reordering schemes to get better nonzero structure. Various reordering methods have been considered for matrix algebra [2, 17, 28, 35, 36, 48]. For tensor operations, Smith et al. [44] proposed two reordering methods based on partitioning: the first is the partitioning of a graph that models the interactions between slices; the second is the partitioning of a hypergraph that models the memory access of the Mttkrp operation. A tensor reordering is induced such that the vertices in the same partition are set to consecutive labels. Hypergraphs have also been used to efficiently parallelize Mttkrp in distributed memory systems [18]. These two hypergraphs are constructed differently from our BFS-MCS, they use fibers and tasks as vertices respectively, while our hypergraph takes tensor indices as vertices. Besides, we propose `matLexiOrder`, an alternative doubly lexical ordering method for sparse matrices, and design Lexi-Order for sparse tensors based on it.

7 CONCLUSION

Motivated by an interest in further improving the HiCOO implementation, we investigated the problem of reordering a tensor to improve block density for tensor computations. Inspired by algorithms for sparse matrices, we proposed two heuristics, BFS-MCS and Lexi-Order. BFS-MCS is based on the maximum cardinality search principle; Lexi-Order is based on `matLexiOrder`, which we propose here as an alternative to existing doubly lexical ordering method for sparse matrices. `matLexiOrder` has near linear runtime per tensor dimension, rendering the overall tensor reordering method Lexi-Order practical. Lastly, we improve the superblock scheduling strategy of HiCOO [24] with a partition-based scheme, which eases the side effect of increased load imbalance that can occur after reordering, thereby improving Mttkrp performance for several sample inputs.

Overall, BFS-MCS is asymptotically faster while Lexi-Order is more effective. Lexi-Order obtains performance speedup on HiCOO-Mttkrp of 2.12× on average for both sequential and multicore parallel implementations. For COO and CSF-Mttkrps, the average sequential and parallel speedups are 1.79×, 1.21× and 1.50×, 1.27×, respectively. Users can control a parameter of Lexi-Order to reduce the reordering overhead without much performance drop and also tune the HiCOO parameters for the reordered tensors to pursue the highest Mttkrp performance. As future work, we plan to pursue the automatic performance tuning for different sparse tensor formats, HiCOO parameters, the tradeoff of reordering methods and parallel strategy parameters, and improve the runtime of Lexi-Order. Our proposed heuristics set a basis for improving Mttkrp performance through tensor reordering. Furthermore, we anticipate the development of additional heuristics for the general tensor reordering problem. Additionally, we expect reordered tensors will prove useful for any tensor operation in which the memory access behavior depends on the locality of tensor indices, as in Mttkrp. Examples include Tensor-times-matrix and tensor-times-vector operations in Tucker decompositions and higher-order power methods, respectively. Our approaches will be especially useful as local improvements for intra-node data locality in large HPC environment.

ACKNOWLEDGMENTS

This research was partially funded by the US Department of Energy, Office for Advanced Scientific Computing (ASCR) under Award No. 66150: "CENATE: The Center for Advanced Technology Evaluation". Pacific Northwest National Laboratory (PNNL) is a multiprogram national laboratory operated for DOE by Battelle Memorial Institute under Contract DE-AC05-76RL01830. This research was also funded in part under Defense Advanced Research Projects Agency (DARPA) contract FA8750-18-2-0108, under the DARPA MTO Software Defined Hardware program, and the Laboratory Directed Research and Development program at Sandia National Laboratories under contract DE-NA-0003525. Disclaimer: The views, opinions, and/or findings contained in this document are those solely of the author(s) and should not be interpreted as representing the official views or policies of any of its funding sources.

REFERENCES

[1] M.ín Abadi et al. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems.

[2] K. Akbudak and C. Aykanat. 2017. Exploiting Locality in Sparse Matrix-Matrix Multiplication on Many-Core Architectures. *IEEE Transactions on Parallel and Distributed Systems* 28, 8 (Aug 2017), 2258–2271.

[3] A. Anandkumar, R. Ge, D. Hsu, ShS.am M. Kakade, and M. Telgarsky. 2014. Tensor Decompositions for Learning Latent Variable Models. *J. Mach. Learn. Res.* 15, 1 (Jan. 2014), 2773–2832.

[4] B. W. Bader and T. G. Kolda. 2007. Efficient MATLAB computations with sparse and factored tensors. *SIAM Journal on Scientific Computing* 30, 1 (December 2007), 205–231.

[5] B. W. Bader, T. G. Kolda, et al. 2017. MATLAB Tensor Toolbox (Version 3.0-dev). Available online. https://www.tensortoolbox.org

[6] M. Baskaran, T. Henretty, B. Pradelle, M. H. Langston, D. Bruns-Smith, J. Ezick, and R. Lethin. 2017. Memory-efficient parallel tensor decompositions. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–7.

[7] M. Baskaran, B. Meister, N. Vasilache, and R. Lethin. 2012. Efficient and scalable computations with sparse tensors. In *High Performance Extreme Computing (HPEC), 2012 IEEE Conference on*. 1–6.

[8] Z. Blanco, B. Liu, and M. M. Dehnavi. 2018. CSTF: Large-Scale Sparse Tensor Factorizations on Distributed Platforms. In *Proceedings of the 47th International Conference on Parallel Processing (ICPP 2018)*. ACM, New York, NY, USA, Article 21, 10 pages. https://doi.org/10.1145/3225058.3225133

[9] J. Choi, X. Liu, S. Smith, and T. Simon. 2018. Blocking Optimization Techniques for Sparse Tensor Computation. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 568–577.

[10] J. H. Choi and S. Vishwanathan. 2014. DFacTo: Distributed Factorization of Tensors. In *Advances in Neural Information Processing Systems 27*, Z. Ghahramani, M. Welling, C. Cortes, N.D. Lawrence, and K.Q. Weinberger (Eds.). Curran Associates, Inc., 1296–1304.

[11] A. Cichocki. 2014. Era of Big Data Processing: A New Approach via Tensor Networks and Tensor Decompositions. *CoRR* abs/1403.2048 (2014).

[12] A. Cichocki, N. Lee, I. V. Oseledets, A. Phan, Q. Zhao, and D. Mandic. 2016. Low-Rank Tensor Networks for Dimensionality Reduction and Large-Scale Optimization Problems: Perspectives and Challenges PART 1. *ArXiv e-prints* (Sept. 2016). arXiv:cs.NA/1609.00893

[13] L. De Lathauwer and D. Nion. 2008. Decompositions of a Higher-Order Tensor in Block Terms—Part III: Alternating Least Squares Algorithms. *SIAM J. Matrix Anal. Appl.* 30, 3 (2008), 1067–1083.

[14] J. C. Ho, J. Ghosh, and J. Sun. 2014. Marble: High-throughput Phenotyping from Electronic Health Records via Sparse Nonnegative Tensor Factorization. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '14)*. ACM, New York, NY, USA, 115–124. https://doi.org/10.1145/2623330.2623658

[15] I. Jeon, E. E. Papalexakis, and C. Faloutsos U Kang. 2015. HaTen2: Billion-scale Tensor Decompositions (Version 1.0). Available from http://datalab.snu.ac.kr/haten2/.

[16] U Kang, E. Papalexakis, A. Harpale, and C. Faloutsos. 2012. GigaTensor: Scaling Tensor Analysis Up by 100 Times - Algorithms and Discoveries. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '12)*. ACM, New York, NY, USA, 316–324. https://doi.org/10.1145/2339530.2339583

[17] G. Karypis and V. Kumar. 1998. A Parallel Algorithm for Multilevel Graph Partitioning and Sparse Matrix Ordering. *J. Parallel and Distrib. Comput.* 48, 1 (1998), 71 – 95.

[18] O. Kaya and B. Uçar. 2015. Scalable Sparse Tensor Decompositions in Distributed Memory Systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15)*. ACM, New York, NY, USA, Article 77, 11 pages. https://doi.org/10.1145/2807591.2807624

[19] T. Kolda and B. Bader. 2009. Tensor Decompositions and Applications. *SIAM Rev.* 51, 3 (2009), 455–500. https://doi.org/10.1137/07070111X arXiv:http://dx.doi.org/10.1137/07070111X

[20] Jiajia Li. 2018. *Scalable tensor decompositions in high performance computing environments*. Ph.D. Dissertation. Georgia Institute of Technology, Atlanta, GA, USA.

[21] J. Li, C. Battaglino, I. Perros, J. Sun, and R. Vuduc. 2015. An input-adaptive and in-place approach to dense tensor-times-matrix multiply. In *ACM/IEEE Supercomputing (SC '15)*. ACM, New York, NY, USA.

[22] J. Li, Y. Ma, and R. Vuduc. 2016. ParTI!: A Parallel Tensor Infrastructure for Multicore CPU and GPUs (Version 0.1.0). Available from https://github.com/hpcgarage/ParTI.

[23] J. Li, Y. Ma, X. Wu, A. Li, and K. Barker. 2019. *PASTA: A Parallel Sparse Tensor Algorithm Benchmark Suite*. Technical Report.

[24] J. Li, J. Sun, and R. Vuduc. 2018. HiCOO: Hierarchical Storage of Sparse Tensors. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '18)*. IEEE Press, Piscataway, NJ, USA, Article 19, 15 pages. http://dl.acm.org/citation.cfm?id=3291656.3291682

[25] B. Liu, C. Wen, A. D. Sarwate, and M. M. Dehnavi. 2017. A Unified Optimization Approach for Sparse Tensor Operations on GPUs. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. 47–57. https://doi.org/10.1109/CLUSTER.2017.75

[26] A. Lubiw. 1987. Doubly Lexical Orderings of Matrices. *SIAM J. Comput.* 16, 5 (1987), 854–879.

[27] Y. Ma, J. Li, X. Wu, C. Yan, J. Sun, and R. Vuduc. 2018. Optimizing sparse tensor times matrix on GPUs. *J. Parallel and Distrib. Comput.* (2018). https://doi.org/10.1016/j.jpdc.2018.07.018

[28] J. Mellor-Crummey, D. Whalley, and K. Kennedy. 2001. Improving Memory Hierarchy Performance for Irregular Applications Using Data and Computation Reorderings. *International Journal of Parallel Programming* 29, 3 (01 Jun 2001), 217–247.

[29] I. Nisa, J. Li, A. Sukumaran-Rajam, R. Vuduc, and P. Sadayappan. 2019. Load-Balanced Sparse MTTKRP on GPUs. arXiv:arXiv:1904.03329

[30] A. Novikov, D. Podoprikhin, A. Osokin, and D. Vetrov. 2015. Tensorizing Neural Networks. *CoRR* abs/1509.06569 (2015).

[31] R. Paige and R. E. Tarjan. 1987. Three Partition Refinement Algorithms. *SIAM J. Comput.* 16, 6 (Dec. 1987), 973–989. https://doi.org/10.1137/0216062

[32] E. E. Papalexakis, C. Faloutsos, and D. D. Sidiropoulos. 2012. ParCube: Sparse Parallelizable Tensor Decompositions. In *Proceedings of the 2012 European Conference on Machine Learning and Knowledge Discovery in Databases - Volume Part I (ECML PKDD'12)*. Springer-Verlag, Berlin, Heidelberg, 521–536.

[33] I. Perros, R. Chen, R. Vuduc, and J. Sun. 2015. Sparse Hierarchical Tucker Factorization and its Application to Healthcare. *IEEE International Conference on Data Mining (ICDM)* (2015).

[34] I. Perros, E. E. Papalexakis, F. Wang, R. Vuduc, E. Searles, M. Thompson, and J. Sun. 2017. SPARTan: Scalable PARAFAC2 for Large & Sparse Data. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '17)*. ACM, New York, NY, USA, 375–384.

[35] J. C. Pichel, F. F. Rivera, M. Fernández, and A. Rodríguez. 2012. Optimization of sparse matrix–vector multiplication using reordering techniques on GPUs. *Microprocessors and Microsystems* 36, 2 (2012), 65 – 77. https://doi.org/10.1016/j.micpro.2011.05.005

[36] A. Pothen and C.-J. Fan. 1990. Computing the Block Triangular Form of a Sparse Matrix. *ACM Trans. Math. Softw.* 16, 4 (Dec. 1990), 303–324.

[37] N. Ravindran, D. D. Sidiropoulos, S. Smith, and G. Karypis. 2014. Memory-Efficient Parallel Computation of Tensor and Matrix Products for Big Tensor Decompositions. *Proceedings of the Asilomar Conference on Signals, Systems, and Computers* (2014).

[38] N. D. Sidiropoulos, L. De Lathauwer, X. Fu, K. Huang, E. E. Papalexakis, and C. Faloutsos. 2017. Tensor Decomposition for Signal Processing and Machine Learning. *IEEE Transactions on Signal Processing* 65, 13 (July 2017), 3551–3582. https://doi.org/10.1109/TSP.2017.2690524

[39] S. Smith, J. W. Choi, J. Li, R. Vuduc, J. Park, X. Liu, and G. Karypis. 2017. FROSTT: The Formidable Repository of Open Sparse Tensors and Tools. http://frostt.io/

[40] S. Smith, K. Huang, N. D. Sidiropoulos, and G. Karypis. [n. d.]. *Streaming Tensor Factorization for Infinite Data Sources*. 81–89. https://doi.org/10.1137/1.9781611975321.10

[41] S. Smith and G. Karypis. 2015. Tensor-Matrix Products with a Compressed Sparse Tensor. In *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*. ACM, 7.

[42] S. Smith and G. Karypis. 2016. A Medium-Grained Algorithm for Distributed Sparse Tensor Factorization. In *Parallel and Distributed Processing Symposium (IPDPS), 2016 IEEE International*. IEEE.

[43] S. Smith, J. Park, and G. Karypis. 2016. An Exploration of Optimization Algorithms for High Performance Tensor Completion. *Proceedings of the 2016 ACM/IEEE conference on Supercomputing* (2016).

[44] S. Smith, N. Ravindran, N. Sidiropoulos, and G. Karypis. 2015. SPLATT: Efficient and Parallel Sparse Tensor-Matrix Multiplication. In *Proceedings of the 29th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*.

[45] S. Smith, N. Ravindran, N. Sidiropoulos, and G. Karypis. 2016. SPLATT: The Surprisingly ParalleL spArse Tensor Toolkit (Version 1.1.1). Available from https://github.com/ShadenSmith/splatt.

[46] R. E. Tarjan and M. Yannakakis. 1984. Simple Linear-Time Algorithms to Test Chordality of Graphs, Test Acyclicity of Hypergraphs, and Selectively Reduce Acyclic Hypergraphs. *SIAM J. Comput.* 13, 3 (1984), 566–579.

[47] Y. Wang, R. Chen, J. Ghosh, J. C. Denny, A. Kho, Y. Chen, B. A. Malin, and J. Sun. 2015. Rubik: Knowledge Guided Tensor Factorization and Completion for Health Data Analytics. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '15)*. ACM, New York, NY, USA, 1265–1274. https://doi.org/10.1145/2783258.2783395

[48] A. J. N. Yzelman and D. Roose. 2014. High-Level Strategies for Parallel Shared-Memory Sparse Matrix-Vector Multiplication. *IEEE Transactions on Parallel and Distributed Systems* 25, 1 (Jan 2014), 116–125.