HiCOO: Hierarchical Storage of Sparse Tensors

Jiajia Li, Jimeng Sun, Richard Vuduc
Computational Sciences & Engineering
Georgia Institute of Technology
Atlanta, Georgia 30332
Email: jiajiali@gatech,edu, {jsun,richie}@cc.gatech.edu

***Abstract*—This paper proposes a new storage format for sparse tensors, called *Hierarchical COOrdinate* (HiCOO; pronounced: "haiku"). It derives from coordinate (COO) format, arguably the de facto standard for general sparse tensor storage. HiCOO improves upon COO by compressing the indices in units of sparse tensor blocks, with the goals of preserving the "mode-agnostic" simplicity of COO while reducing the bytes needed to represent the tensor and promoting data locality. We evaluate HiCOO by implementing a single-node, multicore-parallel version of the matricized tensor-times-Khatri-Rao product (MTTKRP) operation, which is the most expensive computational core in the widely used CANDECOMP/PARAFAC decomposition (CPD) algorithm. This MTTKRP implementation achieves up to $23.0\times$ ($6.8\times$ on average) speedup over COO format and up to $15.6\times$ ($3.1\times$ on average) speedup over another state-of-the-art format, compressed sparse fiber (CSF), by using less or comparable storage of them. When used within CPD, we also observe speedups against COO- and CSF-based implementations.**

I. Introduction

A central problem in tensor-oriented data analysis is what data structure to use to organize the tensor—the natural multiway generalization of a matrix—in a way that is compact, locality-enhancing, and easy to integrate into applications. (For a brief applications survey, see Section II.) Data tensors are usually *sparse*, meaning consisting of mostly zero entries that need not be explicitly stored or operated on. Thus, the problem of picking a sparse tensor data structure is similar to the classical one of how to choose a sparse matrix format. There are many options for sparse matrices that tradeoff size, speed, and "fit" to the nonzero structure of a given input matrix and requirements of the application [1–17]. Similarly, there are several proposals for sparse tensor storage [18–22].

Two critical issues that affect one's choice of format are *compactness* and *mode orientation*. Compactness refers to keeping the total bytes small. Mode orientation is the idea that a format favors iteration of the tensor modes in a particular order or not, as we explain next.

First consider the simpler case of mode orientation for sparse matrices. We say that a matrix has two *modes*, or "axes", namely, its rows and its columns. Let us number the modes, referring to the rows as the first mode, or "mode 1," and the columns as "mode 2". If the matrix is sparse, one might store it in compressed sparse row (CSR) format [1], where each row is a sparse vector and rows are packed contiguously one after the other. One can randomly access any row i from only its index in $\mathcal{O}(1)$ time; however, to find an (i, j) element, you must search the sparse vector representing row i's nonzeros to find j. If one wishes simply to iterate over all nonzeros, as a sparse matrix-vector multiply might do, it is the most efficient to have an outer-loop over rows and an inner-loop over the sparse nonzeros within the row. Therefore, we say CSR is oriented first toward mode 1, then mode 2; its mode orientation may be denoted by $1 \prec 2$ (1 precedes 2), reflecting this loop-order structure. A compressed sparse column (CSC) matrix orients modes as $2 \prec 1$. Per the terminology of Baskaran et al., we say CSR and CSC formats are *mode-specific* [20].

The idea of mode orientation generalizes for tensors. Consider an Nth-order tensor to be an N-way array, meaning it has N modes. (A matrix, with its two modes, is a 2-way tensor.) CSR generalizes to N modes naturally; in the literature, this format is known as *compressed sparse fiber* (CSF) [19], which, like CSR, is mode-specific. For instance, a CSF-1 tensor orients as $1 \prec 2 \prec 3 \prec \cdots \prec N$, with some ordering convention for other cases, CSF-k.

Mode orientation matters because sparse tensor analysis methods may require iterating over the modes in several orientations during the same computation [18]. Thus, for a fixed storage format, iteration might be fast in one orientation but slow when the orientation switches. For a low-order (small-N) tensor, e.g., a matrix, it might be feasible to store multiple copies of the matrix to mitigate this effect. For example, one might store the matrix in both CSR and CSC formats and use the appropriate one when it applies, so the order no longer matters. But for tensors, as the order N grows, a multiple-copy strategy can become infeasible.

A format needs not be mode-specific. The simplest, and arguably most popular storage format for sparse tensors is COOrdinate (COO) format [18, 23, 24]. COO records each nonzero as a tuple, $(i_1, i_2, \ldots, i_N; v)$, where $i_k, k = 1, \ldots, N$ is an index coordinate and v is the nonzero value. Rather than being mode-specific, it is *mode-generic* [20].[1] However, the price is that it is less compact than formats like CSF, which exploit mode-specificity to reduce the average amount of index metadata (i.e., the i_k values) per nonzero.

We wish to propose a new format that tries, heuristically, to be *both* compact *and* mode-generic. We refer to it as *hierarchical coordinate* format, or HiCOO (pronounced "haiku"). By

[1] Baskaran et al. use the terms "mode-specific" and "mode-generic" to refer both to the general types of mode orientation *and* to their particular proposal for two formats to exploit dense tensor substructure. In our paper, we use these terms as general concepts.

SC18, November 11-16, 2018, Dallas, Texas, USA
978-1-5386-8384-2/18/$31.00 ©2018 IEEE

contrast, essentially all proposed formats for general sparse tensors [18, 19, 22] can achieve compactness but not, simultaneously, mode-generic orientation, at least not without paying a performance penalty. This situation includes CSF [19] from above, as well as the more recent flagged-coordinate (F-COO) format, which is also mode-specific [22]. For example, consider the commonly occuring tensor operation known as the "MTTKRP" (Section II). Performing it in a given mode using the CSF representation oriented toward a different mode can suffer a 3× slowdown for tensor `choa` in mode 2.

Our claimed contributions of HiCOO and the present study may be summarized as follows.

- We first compare and analyze COO, CSF, and F-COO formats, along the criteria of compactness and mode orientation, as well as their expected behavior on real tensor computations, such as the matricized tensor-times-Khatri-Rao product (MTTKRP), which motivates this work (Section III).
- We describe HiCOO, which compresses tensor indices in units of sparse tensor blocks and exploits shorter integer types to express offsets within the blocks. Since HiCOO has a mode-generic orientation, only one HiCOO representation is needed (Section IV).
- We accelerate MTTKRP on multicore CPU architectures based on HiCOO. Using a superblock scheduler and two parallelization strategies, our parallel HiCOO-MTTKRP exhibits better thread scalability than COO- and CSF-based MTTKRPs (Section V).
- Overall, HiCOO achieves up to 23.0× (6.8× on average) speedup over COO and 15.6× (3.1× on average) speedup over CSF for a single MTTKRP operation; it can also use up to 2.5× less storage than COO format and comparable storage with only one CSF representation. When MTTKRP is integrated into a complete tensor decomposition algorithm (known as "CPD"), the HiCOO-based implementation is also faster than COO- and CSF-based implementations (Section VI).

II. Background

In applications of tensor-based analysis and mining, an input dataset is represented as a tensor, and the primary computational task is to factorize it in some way, analogous to matrix factorization [25]. These factors may then be interpreted to discover some property of the underlying data, or perhaps exploit it for some task (such as compression [26, 27]). One most commonly used factorization is the CANDECOMP/PARAFAC decomposition (CPD), which is, roughly speaking, analogous to the singular value decomposition (SVD) for matrices in that it seeks to uncover global low-rank structure. The most expensive computational kernel of CPD is the *matricized tensor-times-Khatri-Rao product* (MTTKRP) [20, 22, 28–32]. Thus, the context of this paper is to speed up sparse MTTKRP and then CPD. Symbols and notation used in this paper are summarized in Table I.

TABLE I
List of symbols and notation.

Symbols	Description
$\mathcal{X}$	A sparse tensor
inds	indices of COO, β_{int} bits
val	nonzero value array of COO and HiCOO, β_{float} bits
$\mathbf{X}_{(n)}$	Matricized tensor $\mathcal{X}$ in mode-n
$\mathbf{A}, \mathbf{B}, \mathbf{C}, \tilde{\mathbf{A}}$	Dense matrices
$\mathbf{a_r}, \mathbf{b_r}, \mathbf{c_r}$	Dense vectors
λ	Weight vector
$\odot$	Khatri-Rao product between two matrices
$\circ$	Outer product between two vectors
$*$	Hadamard or element-wise product between two vectors
N	Tensor order
I, J, K, I_n	Tensor mode sizes
M	#Nonzeros of the input tensor $\mathcal{X}$
M_l	#Nodes at level l of a CSF tree of tensor $\mathcal{X}$
R	Approximate tensor rank (usually a small value)
P	#CPU threads
S_{cache}	Cache size
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
L	Tensor superblock size
B	Tensor block size, $B \ll L$
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
$\overline{c}$	Average slice size, $\overline{c} = \frac{M}{I_n}$
n_l	#Nonzero tensor superblocks
n_b	#Nonzero tensor blocks
α_b	Block ratio, $\alpha_b = \frac{n_b}{M}$
$\overline{M_b}$	Geometric mean of #Nonzeros per tensor block
$\overline{c_b}$	Average slice size per tensor block, $\overline{c_b} = \frac{\overline{M_b}}{B}$

Regard a tensor as a multi-way array. Its *order*, N, is the number of its dimensions or *modes*. We follow the notation in Kolda and Bader's survey [25]. 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}.

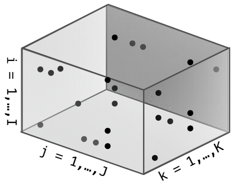


Fig. 1. A third-order tensor $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$.

CPD decomposes a tensor into a sum of component rank-one tensors [25]. It approximates an Nth-order tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ as

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

where R is the canonical rank of tensor $\mathcal{X}$, taken as the number of component rank-one tensors [25]. 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 $\mathbf{A}^{(n)} \in \mathbb{R}^{I_n \times R}, n = 1, \dots, N$ are the *factor matrices*, each one formed by taking the corresponding vectors as its columns, i.e., $\mathbf{A}^{(n)} = [\mathbf{a}_1^{(n)} \; \mathbf{a}_2^{(n)} \dots \mathbf{a}_R^{(n)}]$. We normalize these vectors to unit magnitude and store the factor weights in the vector $\lambda = \{\lambda_1, \dots, \lambda_r\}$.

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

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

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 by arranging all mode-n fibers to be the columns of the matrix. For example, mode-1 matricization of a tensor $\mathcal{X} \in \mathbb{R}^{3 \times 4 \times 5}$ would result in a matrix $\mathbf{X}_{(1)} \in \mathbb{R}^{3 \times 20}$.

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 $\mathbf{C} \in \mathbb{R}^{(IJ) \times R}$,

$$\mathbf{C} = \mathbf{A} \odot \mathbf{B} = [\mathbf{a}_1 \circ \mathbf{b}_1, \mathbf{a}_2 \circ \mathbf{b}_2, \dots, \mathbf{a}_R \circ \mathbf{b}_R], \quad (3)$$

where $\mathbf{a}_r$ and $\mathbf{b}_r$, $r = 1, \dots, R$ are columns of $\mathbf{A}$ and $\mathbf{B}$, $\circ$ is the outer product of vectors, a special case of Kronecker product. Detailed description can be found in Kolda and Bader's survey [25].

In this paper, *sparse MTTKRP* will mean an MTTKRP between a sparse tensor and dense matrices. We can compute it *without* explicit matricization, as will be explained in Section III.

Lastly, in addition to sparsity we will sometimes refer to the concept of *hypersparsity* [33]. We will regard a tensor as hypersparse if its average number of nonzeros per mode is less than one. (For instance, a matrix is hypersparse if there are fewer than one nonzeros per row or column.)

III. Format Comparison

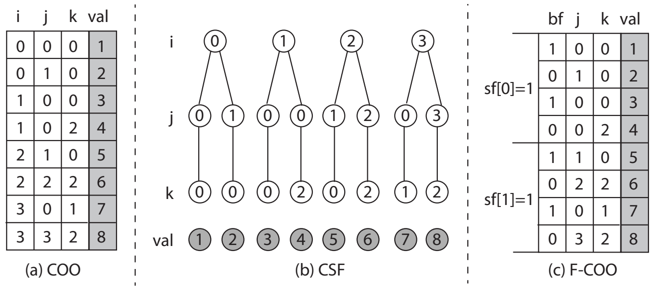
Let us regard COO as the baseline storage format and compare it analytically to two state-of-the-art formats: CSF [19] and F-COO [22]. We consider general unstructured sparse tensors and assess the formats in terms of their storage and behavior for the MTTKRP operation, e.g., floating-point operations or flops, memory traffic, and arithmetic intensity. The results motivate the present work on HiCOO.

For simplicity, this analysis assumes an Nth-order sparse tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ with M nonzeros. We assume an integer index needs β_{int} bits and a nonzero floating-point value takes β_{float} bits. Lastly, we use β_{long} bits for a pointer to index all nonzeros in a very large tensor. The summarized comparison of Table II substitutes values for them that reflect typical choices based on standard primitive types.

A. Summary

We begin with an overall summary of our observations, followed by the detailed analysis.

- **Observation 1:** *CSF generally achieves the best compression for a single representation, but it is worse than COO by storing multiple representations for all modes.* However, using only one CSF representation could suffer a performance penalty. (Table II, column "Index Space")
- **Observation 2:** *CSF and F-COO need extra time and space to construct another representation for the same tensor operation but in a different mode.* That is, neither format is, on its own, mode-generic. (See Table II, "Update Needed?")
- **Observation 3:** *MTTKRP implementations based on COO, CSF, and F-COO formats are expected to have arithmetic intensities of approximately 0.25 flops per byte, which means they will be memory-bound on most platforms.* Reducing memory traffic by enhancing data locality could be beneficial for MTTKRP. (See Table II, "Arithmetic Intensity")



i	j	k	val
0	0	0	1
0	1	0	2
1	0	0	3
1	0	2	4
2	1	0	5
2	2	2	6
3	0	1	7
3	3	2	8

(a) COO

(b) CSF

	bf	j	k	val
sf[0]=1	1	0	0	1
	0	1	0	2
	1	0	0	3
	0	0	2	4
sf[1]=1	1	1	0	5
	0	2	2	6
	1	0	1	7
	0	3	2	8

(c) F-COO

Fig. 2. COO, CSF, and F-COO formats for an example third-order tensor. This CSF tree and F-COO representation are both for mode-1 MTTKRP.

B. COO Format

The COOrdinate (COO) format is the simplest yet arguably most popular format. It stores each nonzero value along with all of its position indices. A COO representation for a third-order example appears in Figure 2(a). As mentioned in Section I, COO does not favor any mode over the others, which gives it a mode-generic orientation.

a) COO format analysis: Storing all the indices uses

$$S_{\text{COO}} = N \cdot M \cdot \beta_{\text{int}} \quad (4)$$

bits.

b) COO-MTTKRP: For illustration purposes, the pseudocode of COO-MTTKRP for third-order tensors appears in Algorithm 1 in Appendix A. MTTKRP multiplies every nonzero entry at position (i, j, k) with row-j of $\mathbf{B}$ and row-k of $\mathbf{C}$, then reduces the rows to row-i of $\tilde{\mathbf{A}}$. The three rows may be irregularly distributed in the matrices depending on the sparsity pattern of $\mathcal{X}$.

For a general N, a rank-R COO-MTTKRP performs

$$\text{Flops}_{\text{COO}} = N \cdot M \cdot R \quad (5)$$

flops, where N operations per nonzero for a matrix column consist of $(N-1)$ multiplications and one addition. Assume no reuse exists in $\tilde{\mathbf{A}}$, $\mathbf{B}$, and $\mathbf{C}$,[2] its memory traffic is

$$\text{Bytes}_{\text{COO}} = \frac{M}{8}(\underbrace{N\beta_{\text{int}}}_{\text{indices}} + \underbrace{\beta_{\text{float}}}_{\text{values}} + \underbrace{NR\beta_{\text{float}}}_{\text{matrices}}) \quad (6)$$

bytes, by counting the instant read and write operations of $\tilde{\mathbf{A}}$ as a one-time memory access. Therefore, its arithmetic

[2] Even though this is the worst case of memory traffic, it is reasonable as an approximation of real sparse tensors which typically have a very small degree of data reuse in MTTKRP. Similar assumption is also made for the following CSF and F-COO analysis.

TABLE II
THE ANALYSIS OF TENSOR FORMATS AND THEIR MTTKRP ALGORITHMS FOR A THIRD-ORDER TENSOR ($N = 3$) WITH M NONZERO ENTRIES. THE WORD SIZE PARAMETERS ARE $\beta_{\text{INT}} = 32$, $\beta_{\text{LONG}} = 64$, $\beta_{\text{BYTE}} = 8$, AND $\beta_{\text{FLOAT}} = 32$ BITS FOR SINGLE-PRECISION FLOATING-POINT VALUES AND DISCARDING INSIGNIFICANT ITEMS.

	Data Structure		MTTKRP Behavior		
Format	Index Space (Bits)	Update Needed?	Work (Flops)	Memory Access (Bytes)	Arithmetic Intensity (AI)
COO	$96M$	NO	$3MR$	$12MR$	$1/4$
F-COO	$65M$	YES	$3MR$	$12MR$	$1/4$
CSF	$[32M, 128M]$	YES[1]	$[2MR, 4MR]$	$[8MR, 16MR]$	$1/4$
HiCOO	$[24M, 184M]$	NO	$3MR$	$min\{\frac{12}{\overline{c_b}}, 12\}MR$	$max\{\frac{1}{4}, \frac{\overline{c_b}}{4}\}$

[1] MTTKRPs in all tensor modes can use less CSF representations than tensor order or even only one CSF representation with performance payoff.

intensity (AI), or ratio of the number of flops to the number of memory accesses, is about $1/4$ when $\beta_{\text{int}}, \beta_{\text{float}} = 32$ bits (as listed in Table II).

C. CSF Format

CSF (compressed sparse fiber) is a hierarchical, fiber-centric format that effectively generalizes the CSR matrix format to tensors, and is implemented in the SPLATT package [34]. An example of its structure appears in Figure 2(b). Conceptually, CSF organizes nonzero values into a tree. Each level corresponds to a tensor mode, and each nonzero is a path from the root to a leaf. Because the path implies a mode ordering for enumeration of nonzeros, CSF has a strong mode-specificity.

For an Nth-order tensor, N CSF trees would be needed for the best performance if the tensor operation of interest needs to be iterated over the tensor multiple times, once in each mode, as stated in [35]. An analogy would be a computation that needed both parallel matrix-vector and matrix-transpose-vector multiplies without write conflicts; a simple and fast approach would be to store the matrix in both CSR and CSC formats at the cost of doubling the space as suggested in [36]. However, storing multiple CSF trees consumes a large extra storage space (as will be shown in Table IV).

a) CSF format analysis: The CSF format uses

$$S_{\text{csf}} = \underbrace{\sum_{l=1}^{N-1} (M_l + 1) \cdot \beta_{\text{long}}}_{\text{pointers}} + \underbrace{\sum_{l=1}^{N} M_l \cdot \beta_{\text{int}}}_{\text{indices}} \tag{7}$$

bits for a *single* CSF tree, where M_l represents the number of nodes at level l in Figure 2(b). $M_l = M$ when $l = N$, and $M_l = I_n$ when $l = 1$. Besides indices for all N levels, CSF stores pointers at non-leaf levels to index the beginning locations of every sub-tree, i.e., sub-tensor. These pointers are analogous to the row pointers in CSR for sparse matrices. If a tensor has $M_l \ll M, l = 1, \dots, N-1$, CSF achieves good compression; otherwise, if $M_l \approx M$, it may need even more storage than COO because of the overhead of storing additional pointers with more bits (β_{long} instead of β_{int}). These two extreme cases are shown in Table II as lower and upper bounds. When maintaining multiple CSF trees for higher MTTKRP performance, the total storage will be their sum.

b) CSF-MTTKRP: CSF-MTTKRP performs

$$\text{Flops}_{\text{csf}} = 2R \sum_{l=2}^{N} M_l \tag{8}$$

flops [28]. It incurs

$$\text{Bytes}_{\text{csf}} = \frac{1}{8} [\underbrace{\beta_{\text{long}} \sum_{l=1}^{N-1} M_l}_{\text{pointers}} + \underbrace{\beta_{\text{int}} \sum_{l=1}^{N} M_l}_{\text{indices}} + \underbrace{M \beta_{\text{float}}}_{\text{values}} + \underbrace{(2R \sum_{l=2}^{N} M_l) \beta_{\text{float}}}_{\text{matrices}}] \tag{9}$$

bytes of memory accesses [3] [19, 28]. Therefore, its arithmetic intensity (AI) is also about $1/4$ when $\beta_{\text{int}}, \beta_{\text{float}} = 32$ bits, $\beta_{\text{long}} = 64$ bits.

D. F-COO Format

Flagged-COOrdinate (F-COO) format, recently proposed in [22], stores the indices of the mode(s) needed by a given product operation while replacing all remaining modes with two bit-arrays, a "bit-flag" (bf) and a "start-flag" (sf). Figure 2(c) shows an example of a mode-1 F-COO representation which can be used to compute the MTTKRP oriented toward mode 1 only, i.e., $\tilde{\mathbf{A}} \leftarrow \mathcal{X}_{(1)}(\mathbf{C} \odot \mathbf{B})$. The indices j and k are unchanged, and i is replaced by bf and sf bit-arrays, thereby reducing storage space. Intuitively, bf and sf indicate any changes in the index mode(s), then FCOO-MTTKRP uses segmented scan primitive to avoid locks or atomic operations.

a) F-COO format analysis: We only analyze F-COO for the MTTKRP operation in this work, interested readers could refer to [22] for other scenarios, e.g., tensor-times-matrix multiplication (TTM). The F-COO format uses

$$S_{\text{fcoo}} = M[\underbrace{(N-1)\beta_{\text{int}}}_{\text{indices}} + \underbrace{1}_{\text{bit-flag}} + \underbrace{\frac{1}{M_{thread}}}_{\text{start-flag}}] \tag{10}$$

bits to hold the indices, where M_{thread} is the number of nonzeros assigned to a thread. It is strongly mode-specific, thus an F-COO representation is required for every mode of a tensor operation.

b) F-COO MTTKRP: MTTKRP based on F-COO performs

$$\text{Flops}_{\text{fcoo}} = N \cdot M \cdot R \tag{11}$$

flops, which is the same as COO-MTTKRP. It moves

$$\text{Bytes}_{\text{fcoo}} = \frac{M}{8} \left[\underbrace{(N-1) \cdot \beta_{\text{int}} + 1 + \frac{1}{M_{thread}}}_{\text{indices and flags}} + \underbrace{N \cdot R \cdot \beta_{\text{float}}}_{\text{matrices}} \right] \tag{12}$$

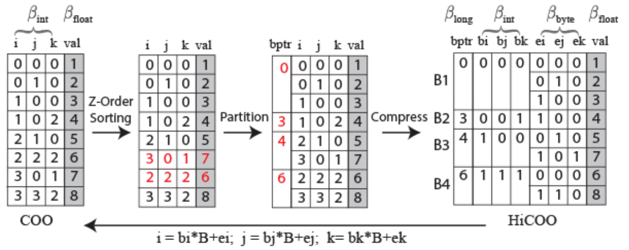
[3] CSF-MTTKRP in SPLATT uses a R-array to accumulate intermediate results of inter-nodes. Since this small R-array could be easily cached between two adjacent node levels, only one of the two reads is counted.

bytes of data to and from memory. Therefore, its AI is about $1/4$ when $\beta_{\text{int}}, \beta_{\text{float}} = 32$ bits.

From the above three observations and detailed analysis, we propose a new sparse tensor format HiCOO to overcome the drawbacks of current formats, maintain mode-generic orientation, meanwhile, pursue higher performance by optimizing memory locality.

IV. HiCOO Format

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 (only cubical blocks are considered in this work). It represents every block by compactly storing its nonzero triples using fewer bits. A tensor is sorted and then partitioned and compressed by every mode into sized-B chunks, resulting in at most $\frac{I_1}{B} \times \cdots \times \frac{I_N}{B}$ (assume all I_ns are dividable by B) nonzero tensor blocks. Figure 3 shows the same third-order tensor example as Figure 2 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, and 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, which is the same with COO's val array. HiCOO treats every mode equally and does not assume any mode order, these preserve the mode-generic orientation of COO.



β_{int} β_{float}
Z-Order Sorting
Partition
Compress
β_{long} β_{int} β_{byte} β_{float}
B1
B2
B3
B4
COO
HiCOO
i = bi*B+ei; j = bj*B+ej; k= bk*B+ek

i	j	k	val
0	0	0	1
0	1	0	2
1	0	0	3
1	0	2	4
2	1	0	5
2	2	2	6
3	0	1	7
3	3	2	8

i	j	k	val
0	0	0	1
0	1	0	2
1	0	0	3
1	0	2	4
2	1	0	5
3	0	1	7
2	2	2	6
3	3	2	8

bptr	i	j	k	val
0	0	0	0	1
	0	1	0	2
	1	0	0	3
3	1	0	2	4
4	2	1	0	5
	3	0	1	7
6	2	2	2	6
	3	3	2	8

	bptr	bi	bj	bk	ei	ej	ek	val
B1	0	0	0	0	0	0	0	1
					0	1	0	2
					1	0	0	3
B2	3	0	0	1	1	0	0	4
B3	4	1	0	0	0	1	0	5
					1	0	1	7
B4	6	1	1	1	0	0	0	6
					1	1	0	8

Fig. 3. The conversion between COO and HiCOO formats for an example third-order tensor. HiCOO uses $2 \times 2 \times 2$ blocks ($B = 2$) with word sizes marked above.

A. Conversion

Sorting, partitioning, and compression are the three steps to convert from a COO tensor to a HiCOO tensor. We first sort all nonzeros of a COO tensor in Z-Morton order [37] using a variation of quick sort. A Morton key is computed from nonzero indices and is used for the comparison of sorting. In Figure 3, the sorted COO tensor switches two nonzero entries (marked in red). We then partition the sorted tensor into sparse tensor blocks according to the given block size B and record the block pointers bptr simultaneously. Since we limit the block size to a power-of-two constant, this partitioning maintains the sorted Z-order between tensor blocks and among the nonzeros within a block. Lastly, we compress COO indices into block and element indices correspondingly. By having a HiCOO tensor, no need to explicitly convert it back to a COO tensor. The COO indices of a nonzero entry can be calculated from $i = \mathrm{bi} \cdot B + \mathrm{ei}$, $j = \mathrm{bj} \cdot B + \mathrm{ej}$, and $k = \mathrm{bk} \cdot B + \mathrm{ek}$.

In the HiCOO format, Z-Morton sorting contributes better data locality for tensor algorithms, while compressed indices save the storage space of a sparse tensor and also reduce the memory bandwidth of tensor access.

B. Improvement of CSB

Our proposed Hierarchical COOrdinate (HiCOO) format may be viewed as an extension of the Compressed Sparse Blocks (CSB) format for sparse matrices [33]. One distinction between HiCOO and CSB is that the latter uses relatively larger *matrix blocks*. [4] By contrast, we find that smaller blocks are more suitable for sparse tensors, both for reasons related to better cache usage and better support for higher-order tensor operations. However, small blocks pose two issues of a straightforward extension of CSB, which will be explained in Appendix B. To solve these challenges, HiCOO improves from the CSB idea in two aspects.

- First, HiCOO further compacts block indices, thus requiring even less storage space than CSB. We compact block indices in coordinate pattern to control their storage rise for small blocks and also uses fewer bits when possible.
- Second, for efficient CPU multithreading, HiCOO uses a two-level blocking strategy and a small amount of extra space to save scheduling information. We group a set of small blocks into a large yet logical *superblock*. The blocks within a superblock are always scheduled together and assigned to a single thread. Within a superblock, we physically store nonzeros in the same pattern as shown in Figure 3. This two-level blocking strategy will be better explained in Section V since it is more related to algorithm parallelization.

C. Analysis

Our analysis of HiCOO will be expressed in terms of parameters listed in Table I. We first explain these parameters and give the format analysis afterwards.

The *Average Slice Size ($\bar{c}$)* is a tensor-dependent parameter. It is an analogy of "row length" of a sparse matrix. $\bar{c}$ is the average slice size in a particular mode n, $\bar{c} = \frac{M}{I_n}$. $\bar{c}$ could vary considerably, from being a constant ($\bar{c} = \mathcal{O}(1)$) if there are only a few nonzeros per slice, to being as large as $\bar{c} = \mathcal{O}(I^2)$ for a third-order tensor if its slices are dense. The value $\bar{c}$ effectively measures nonzero density, especially for irregularly shaped sparse tensors.

The *Number of Tensor Blocks (n_b)* depends on the input tensor and HiCOO-specific block size B. The example in Figure 3 has $n_b = 4$ tensor blocks.

The *Block Ratio (α_b)* is the ratio of the number of tensor blocks to the number of total nonzero elements, $\alpha_b = \frac{n_b}{M}$. Block ratio directly affects the storage size of HiCOO, which will be shown in Equation (13). For a given sparse tensor with a fixed M, α_b is not solely determined by the block size B, but also related to nonzero distribution.

The *Geometric Mean of Numbers of Nonzeros per Block ($\overline{M_b}$)* depends on nonzero distribution and block size B.

[4] In CSB, block sizes are typically around $\sqrt{I} \times \sqrt{I}$ for an $I \times I$ sparse matrix [33].

We choose geometric mean because we expect skewed or uneven distributions. From our experiments on real tensors in Table III, the numbers of nonzeros per block are generally skewed downward, with many small values and a few large ones, and unevenly distributed. The four tensor blocks in Figure 3 consist of $3, 1, 2, 2$ nonzeros respectively, thus $\overline{M_b} = 1.9$.

The *Average Slice Size per Tensor Block* ($\overline{c_b}$) is analogous to $\overline{c}$: it is the average slice size in a block, $\overline{c_b} = \frac{\overline{M_b}}{B}$. The value $\overline{c_b}$ reflects the nonzero density of a block. It is also crucial to MTTKRP performance (details in Section V). Note that for a third-order tensor, $\overline{c} = \mathcal{O}(1)$ (or $\mathcal{O}(I^2)$) does not necessarily mean $\overline{c_b} = \mathcal{O}(1)$ (or $\mathcal{O}(B^2)$) because of potential nonuniform local and global nonzero distributions.

The HICOO format uses

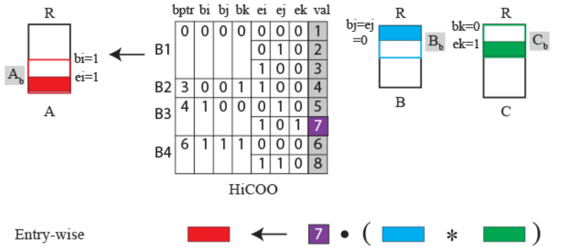
$$\begin{aligned} S_{\text{hicoo}} &= \underbrace{(n_b+1)\cdot\beta_{\text{long}}}_{\text{bptr}} + \underbrace{N\cdot n_b\cdot\beta_{\text{int}}}_{\text{block indices}} + \underbrace{N\cdot M\cdot\beta_{\text{byte}}}_{\text{element indices}} \\ &\approx M[\alpha_b\cdot\beta_{\text{long}} + \alpha_b N\cdot\beta_{\text{int}} + N\cdot\beta_{\text{byte}}] \end{aligned} \tag{13}$$

bits of storage. Given a sparse tensor, its storage size of a HICOO representation only depends on α_b. The index space shown in Table II takes two extreme values 0 and 1 of α_b to get its lower and upper bounds. This amount will be smaller than COO when $S_{\text{hicoo}} < S_{\text{coo}}$, or

$$\alpha_b < \frac{\beta_{\text{int}} - \beta_{\text{byte}}}{\beta_{\text{int}} + \frac{1}{N}\beta_{\text{long}}}. \tag{14}$$

For a third-order tensor ($N = 3$) with $\beta_{\text{int}} = 32$ bits, $\beta_{\text{long}} = 64$ bits, $\beta_{\text{byte}} = 8$ bits, then $\alpha_b < 0.45$. This means if there are more than 2 nonzeros per block on average, theoretically, HICOO consumes less space than COO. The threshold of α_b grows with tensor order N, i.e., the sparsity restriction of HICOO becomes looser with increasing dimensionality.

Overall, HICOO is compact, exposes data locality in every tensor mode, and is mode-generic. HICOO, as a general sparse tensor format, is able to support diverse types of tensor operations and various computing platforms.



	bptr	bi	bj	bk	ei	ej	ek	val
B1	0	0	0	0	0	0	0	1
					0	1	0	2
					1	0	0	3
B2	3	0	0	1	1	0	0	4
B3	4	1	0	0	0	1	0	5
					1	0	1	7
B4	6	1	1	1	0	0	0	6
					1	1	0	8

Fig. 4. A HICOO-MTTKRP example showing the operations on one nonzero tensor entry 7. Its corresponding matrix blocks $\mathbf{A}_b, \mathbf{B}_b, \mathbf{C}_b$ are shown as bounded blank boxes, and its corresponding matrix rows are plotted as solid boxes inside.

V. HICOO-MTTKRP ALGORITHMS

We use the HICOO format for the MTTKRP operation and introduce our optimization methods for sequential and multicore parallel algorithms.

A. *Sequential Algorithm*

Figure 4 depicts HICOO-MTTKRP algorithm on the example $4\times4\times3$ HICOO tensor from Figure 3. Taking the nonzero entry with a value 7 in block $B3$ as an example, its block indices are $(1, 0, 0)$ and element indices are $(1, 0, 1)$. Block indices $(\text{bi}, \text{bj}, \text{bk})$ identify the beginning locations of blocked matrices $\mathbf{A}_b, \mathbf{B}_b, \mathbf{C}_b$ (marked as blank boxes) by offsetting $\text{bi}\cdot B, \text{bj}\cdot B, \text{bk}\cdot B$ rows from $\mathbf{A}, \mathbf{B}, \mathbf{C}$. Every tensor block only accesses blocked matrices $\mathbf{A}_b, \mathbf{B}_b, \mathbf{C}_b \in \mathbb{R}^{B\times R}$. Then the nonzeros within a block can be indexed only by element indices $(\text{ei}, \text{ej}, \text{ek})$. The matrix rows needed by this nonzero element are plotted as solid boxes. At the bottom of Figure 4, the two rows of $\mathbf{C}$ and $\mathbf{B}$ first do element-wise product, then their resulting vector is scaled by the nonzero value 7 to update one row of $\mathbf{A}$. We use SIMD directives to accelerate the vector operations of each nonzero entry. Sequential HICOO-MTTKRP algorithm for a third-order tensor is shown in Algorithm 2 in Appendix A.

If block size B is configured to keep $\mathbf{A}_b, \mathbf{B}_b, \mathbf{C}_b$ cached in local memory, they can be re-used multiple times without memory transfers. Assume the number of nonzeros of a tensor block is $\overline{M_b}$, and every slice in the block has an equal size $\overline{c_b}$, thus the memory traffic of one blocked matrix is $NR\min\{B, \overline{M_b}\}\beta_{\text{float}}$ bits. When $\overline{M_b} > B$, or equally $\overline{c_b} > 1$, at most NBR values are transferred from memory for this blocked matrix because all its values are already cached after the first transfers; otherwise, if $\overline{c_b} < 1$, all $NR\overline{M_b}$ values will be transfered from memory as COO-MTTKRP and no reuse for them. Tensor blocking exploits data locality in memory transfers for the matrix operands, and also benefits the parallel algorithms that follow.

Analysis. Sequential HICOO-MTTKRP algorithm has

$$\text{Flops}_{\text{hicoo}} = N\cdot M\cdot R \tag{15}$$

flops, identical to COO-MTTKRP algorithm in Equation (5). Since HICOO-MTTKRP exploits matrix reuse, its memory traffic is

$$\begin{aligned} \text{Bytes}_{\text{hicoo}} =& \frac{n_b}{8}[\underbrace{\beta_{\text{long}}}_{\text{one bptr value}} + \underbrace{2N\cdot\beta_{\text{int}}}_{\text{block indices and blocked matrices pointers}} \\ &+ \underbrace{\overline{M_b}(N\cdot\beta_{\text{byte}} + \beta_{\text{float}})}_{\text{element indices and values per block}} + \underbrace{NRmin\{B, \overline{M_b}\}\cdot\beta_{\text{float}}}_{\text{blocked matrix values}}] \\ &\approx \frac{M}{8}[2\alpha_b\cdot\beta_{\text{int}} + \beta_{\text{byte}} + R\min\{\frac{1}{\overline{c_b}}, 1\}\cdot\beta_{\text{float}}]N \end{aligned} \tag{16}$$

bytes, where $n_b \times \overline{M_b} \approx M$, $\alpha_b = \frac{n_b}{M}$. Given a sparse tensor, the memory traffic depends on α_b, $\overline{c_b}$, and R. α_b determines the traffic of loading the tensor, while $\overline{c_b}$ and R influences that of loading factor matrices which generally dominates the whole memory traffic of HICOO-MTTKRP. Therefore, its arithmetic intensity (AI) is about $\max\{\frac{1}{4}, \frac{\overline{c_b}}{4}\}$ when $\beta_{\text{int}}, \beta_{\text{float}} = 32$ bits, $\beta_{\text{byte}} = 8$ bits, which is listed in Table II. When $\overline{c_b} > 1$, theoretically HICOO-MTTKRP has higher arithmetic intensity than all the state-of-the-art MTTKRP algorithms.

Since $n_b \times \overline{M_b} \approx M$, $\alpha_b \times \overline{c_b} \approx \frac{1}{B}$. $\overline{c_b}$ is inversely proportional to α_b. However, there are cases that a reordered tensor has the same α_b but different $\overline{c_b}$ values which lead to MTTKRP performance change. This is because we use the geometric mean for $\overline{c_b}$, $\alpha_b\times\overline{c_b}$ is not always strictly equal to $\frac{1}{B}$, but this $\overline{c_b}$ can better reflect performance. The two parameters are both needed to guide users in Section V-C.

B. Parallel Algorithm

As mentioned in Section IV, our small tensor blocks are good for data locality but have parallel issues on multicore CPUs. Parallelizing small tensor blocks of sequential HiCOO-MTTKRP (Line 1 in Algorithm 2) occurs a huge number of small workloads per thread, which is not efficient for heavyweight CPU threads. Besides, parallelizing either the first (line 1) or the second loop (line 4) leads to write conflicts, multiple threads may write the same blocked matrix $\tilde{\mathbf{A}}_b$ or even the same row of it. Because the rank R is very small, parallelizing the R-loop (line 6) is not efficient and causes false-sharing for $\tilde{\mathbf{A}}_b$. A simple solution is to use expensive locks or atomic operations to protect $\tilde{\mathbf{A}}_b$. Our work completely removes write conflicts by taking advantage of pre-knowledge from HiCOO construction process.

Logical Superblocks. We increase the workload granularity of scheduling and employ two-level blocking that groups small blocks into larger ones which we call *superblocks*. A superblock is essentially a "logical" subtensor that can potentially consist of many small blocks. During HiCOO format conversion, we first extract $L \times \cdots \times L$ nonzero superblocks then treat each superblock as an independent tensor to convert it to the physical HiCOO format with $B \times \cdots \times B$ blocks ($L \geq B$). An additional array lptr is included to store the beginning pointers of nonzero superblocks, with its size n_l, the number of superblocks.

Superblock Scheduling. Since HiCOO systematically partitions a Z-order sorted tensor into superblocks, data race information can be recorded and then used to guide parallel MTTKRP execution. A superblock scheduling table lschr is constructed to indicate write-conflict information between them. Generally, the overhead of storing lptr and lschr are negligible compared to the other components of HiCOO, e.g., einds, binds, because of the small number of superblocks. Different superblock sizes will generate different scheduling tables.

Figure 5 shows a lschr table for mode-1 MTTKRP. In the y-direction, superblocks (e.g., 0, 1, 2, 3) have no write conflicts because they are updating different regions of $\tilde{\mathbf{A}}$. These superblocks can be independently parallelized, therefore, a reasonable large number of superblocks in the y-direction is preferable. In the x-direction, superblocks (e.g., 0, 4, 8) have potential write conflicts (shown as arrows) that are, therefore, avoidable if they are executed sequentially. In this figure, three iterations are needed to serialize all the 10 superblocks. The larger the number of iterations, the stronger dependence shown in this MTTKRP. With this superblock scheduler, we iterate superblocks in two loops: one for independent superblocks, the other for iterations; and choose a strategy to parallelize one of them.

Parallel Strategies. The easy case is that we have reasonably large independent superblocks to directly parallelize. However, the number of independent superblocks depends on the product mode size (i.e., I_n in mode n) and the superblock size L. For mode-n MTTKRP, the number of independent superblocks is roughly $\frac{I_n}{L}$. If it is small (e.g., 2 for `choa` in mode 3), most superblocks cannot run simultaneously.

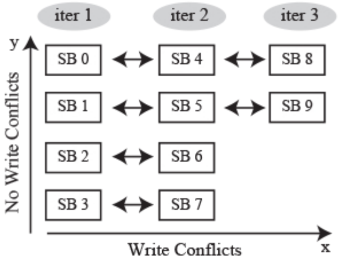


Fig. 5. Superblock scheduling table for mode-1 MTTKRP.

We introduce a widely used privatization parallel strategy [28, 38] to parallelize iterations, especially for irregularly-shaped tensors. Thread-local buffers are created locally to keep the updates of the superblocks from distinct iterations, then they can be parallelized independently. Afterwards, an extra parallel reduction stage is used to get the final results. The privatization strategy is only used for MTTKRP in short tensor modes, i.e., small corresponding factor matrices, due to their disadvantage of direct parallelization. Besides, it consumes trivial extra memory. To better fit the dynamic workload of superblocks, we use dynamic OpenMP strategy for parallelism.

C. Parameter Guidance

Several of the parameters in Table I can affect the performance of HiCOO-MTTKRP and HiCOO's storage size. We illustrate five critical parameters from our experiments and provide an analysis here that can guide users on how to tune them. (Our implementation of this analysis can suggest to users if a particular parameter value might not be a good choice.) The first three are the parameters of the HiCOO format, while the last two are choices among COO, CSF, and HiCOO formats. Since our analyses in Section III, IV, V are for general tensors, these parameters could describe the features of both cubical and irregularly-shaped tensors which frequently occur in real applications.

Block Size. Block size B should be set to keep all the blocked matrices in fast cache, i.e., $NBR\beta_{\text{float}} \leq S_{cache}$. Users could compute the largest B for HiCOO. This explains almost all HiCOO-MTTKRPs obtain their best performance with $B = 128$ in our experiments, when $R = 16$.

Parallel Strategy. If the number of independent superblocks is too small (e.g., $< 4P$, P is the number of threads) to be parallelized efficiently and the iterations are much more than independent superblocks (e.g., 20× larger in our experiments), we use privatization strategy to parallelize superblocks between iterations. This case generally occurs on irregularly-shaped tensors. For example, mode 3 of tensor `choa` is much smaller than the other two modes. Given a superblock size $L = 512$, the number of independent superblocks is only 2 in mode 3, while the number of iterations is as large as 22475. In this case, privatization strategy is chosen for better performance.

Superblock Size. Superblock size L is important to parallel MTTKRP performance. It can be tuned to ensure reasonable amount of tasks for CPU threads. After fixing a parallel strategy, we can compute the number of parallel tasks (either independent superblocks or iterations) for different Ls. When this number is reasonable, e.g., between $[4P, 100P]$, this L is acceptable; otherwise, a smaller or larger L will be suggested.

Storage space. As explained in Section IV, α_b determines the storage size of HiCOO, smaller is better. This effect will also be shown from Table IV in Appendix C. The threshold of HiCOO to achieve compressed storage than COO can be calculated from Equation (14). We suggest to first compare the value to its threshold (0.45 for 3D tensors), if α_b is larger than it, a HiCOO representation is highly possible to use more storage than one COO and also CSF representation.

Performance. $\overline{c_b}$ is critical to HiCOO-MTTKRP's memory traffic and thus its performance from Table II, larger is better. From our experiments in Figure 10, unless $\overline{c_b} < 0.01$ (excluding `deli` and `nell1` for 3D tensors), HiCOO-MTTKRP is faster than CSF-MTTKRP by taking advantage of our parallel strategy. Besides, α_b is also taken into account because it affects the memory traffic of loading the input tensor, especially when the tensor is much larger than factor matrices. The effects of α_b and $\overline{c_b}$ on MTTKRP performance and last-level cache behavior will be shown in Figure 10 of Appendix C. If α_b is larger and $\overline{c_b}$ is smaller than their thresholds, HiCOO-MTTKRP may not behave well. Therefore, CSF-MTTKRP is favorable.

Note that these thresholds are obtained from empirical tests on our platform. A user may need to experiment with other thresholds.

VI. Evaluation and Analysis

A. Experimental Setup

Platform The experiments are ran on a Linux server with Intel Xeon E7-4850 v3 multicore platform consisting 56 physical cores with 2.2 GHz frequency on four sockets. It is a Haswell platform with 32 KiB L1 data cache and 1970 GiB memory. Code is written in C language with OpenMP directives, and is compiled by icc 18.0.2.

Dataset We use the sparse tensors, derived from real-world applications, that appear in Table III, 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 (Refer to the details in [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 [40], and `choa` is built from electronic health records (EHRs) of pediatric patients at Children's Healthcare of Atlanta (CHOA) [41].

Implementations We compare the performance of MTTKRP algorithms on multicore CPUs using COO and CSF formats[5]. COO-MTTKRP from ParTI! library [42] is a C implementation of the MTTKRP that appears in Tensor Toolbox [43]. We use OpenMP with privatization method to obtain its highest possible performance at a cost of maybe large extra space to save the local copies. CSF-MTTKRP is from Splatt v1.1.1 [34] which is regarded as the state-of-the-art MTTKRP and CPD library [28]. We configured Splatt for its best performance, meaning the `ALLMODE` setting (storing all N CSF trees) and enabling the tiling option. All parallel programs are configured using "numactl" to interleave allocated memory system-wide, i.e., on all sockets.

TABLE III
Description of sparse tensors.

Tensors	Order	Dimensions	#Nonzeros	Density
`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}
`deli`	3	$533K \times 17M \times 2.5M$	140M	6.1×10^{-12}
`nell1`	3	$3M \times 2M \times 25M$	144M	9.1×10^{-13}
`crime`	4	$6K \times 24 \times 77 \times 32$	5M	1.5×10^{-2}
`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}
`flickr`	4	$320K \times 28M \times 2M \times 731$	113M	1.1×10^{-14}
`deli4d`	4	$533K \times 17M \times 2M \times 1K$	140M	4.3×10^{-15}

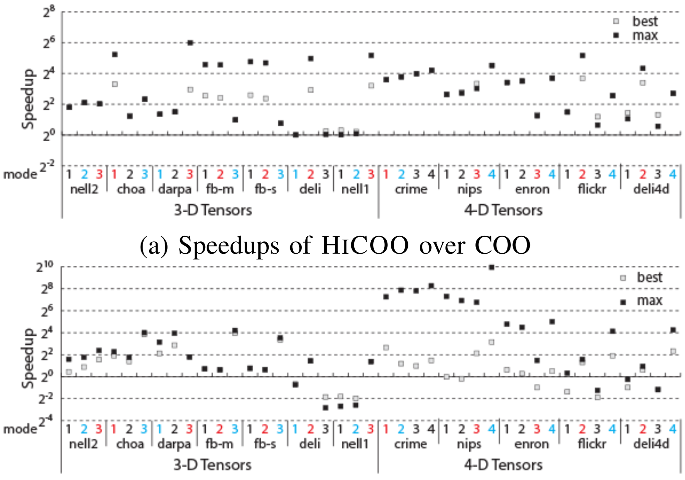
B. Overall Performance

The tensor rank R is set to 16 in our experiments. For all experiments, we use 32-bit unsigned integers for indices and pointers, which are enough for these tensors, and 32-bit single-precision floating-point values. Most experiments find $B = 128$ achieve the best performance, even though we can use up to 16 bits for larger B, while L varies much among these tensors. Generally, for a long tensor mode, direct parallelism is used; while for a very short mode, privatization method is chosen. All execution times are averaged over five iterations.

Figure 6 (a) and (b) show the speedup of parallel MTTKRP in a single mode based on HiCOO over COO and CSF (`ALLMODE` setting) formats, respectively. We test up to 56 threads and show the performance obtained by the most threads ("max") and under the thread configuration with the highest performance ("best") separately. The y-axis shows the speedup and x-axis gives all the cases of MTTKRPs, i.e., MTTKRPs in every mode on all tensors. A single parallel HiCOO-MTTKRP achieves up to 63.5× (12.6× on average) speedup over COO format and up to 991.4× (62.0× on average) speedup over CSF format when using 56 threads. The extraordinary high speedup over CSF is achieved on tensors `crime` and `nips`, because CSF-MTTKRP scales poorly in their very short modes. Under the "best" thread configuration for all implementations, a single parallel HiCOO-MTTKRP achieves up to 23.0× (6.8× on average) speedup over COO and up to 15.6× (3.1× on average) speedup over CSF. The highest performance of CSF-MTTKRP on tensors `crime` and `nips` is mostly obtained on 2 and 4 threads. In some tensor modes, only a black square can be recognized since the highest speedup is achieved by using all 56 threads. Mostly the "max" configuration gets higher or equal speedup to the "best" configuration because HiCOO-MTTKRP is the most scalable in the three implementations (see Figure 8 below). However,

[5] F-COO is implemented only for GPUs.

in some cases when HiCOO-MTTKRP is less well-scaled, the "best" configuration may get better speedup. In the following content, we only refer to the speedup achieves under the "best" thread configuration.



(a) Speedups of HiCOO over COO

(b) Speedups of HiCOO over CSF in ALLMODE setting

Fig. 6. MTTKRP performance comparison with the "max" and "best" thread configurations. The longest modes and shortest modes of every tensor are marked in red and blue respectively.

As we mentioned, irregularly-shaped tensors have distinct performance behavior in different modes. We summarize the speedup separately for the longest (marked in red in Figure 6) and the shortest (marked in blue) modes of every tensor. HiCOO-MTTKRP in the shortest mode achieves an average 6.7× speedup over COO and 5.7× speedup over CSF; while in the longest mode, HiCOO-MTTKRP obtains an average 8.1× speedup over COO and 2.8× over CSF. HiCOO is more advantageous in short modes compared to CSF, because of the less shape-sensitivity and privatization parallel strategy of HiCOO. Compared to COO, HiCOO shows a marginal win in long modes, because HiCOO avoids the relatively expensive reduction stage of privatization method used in COO.

HiCOO-MTTKRP is almost always faster than COO-MTTKRP due to its better data locality and smaller memory footprint. Compared to CSF-MTTKRP, the speedup of HiCOO-MTTKRP is not stable because CSF-MTTKRP distinctly behaves on long and short modes, where long modes are more favorable. Some tensors, e.g., nell1 and deli, have almost all the blocks containing just a few nonzero entries, making them hypersparse ($\overline{c_b} \ll 1$). Thus, HiCOO cannot reduce much memory traffic of MTTKRP (see Table II). Since our method flexibly chooses the two parallelization strategies, tensors fb-m and fb-s need only 249KB and 760KB extra space for the local matrix copies respectively, trading that for much higher performance.

C. Optimization Breakdown

We show the advantages of HiCOO by evaluating three factors separately, which are sorting, compression, and SIMD. The baseline is COO-MTTKRP with the COO representation sorted in a lexicographic mode order, then we use Z-order sorting to rearrange nonzeros but still run COO-MTTKRP, where we get 18% speedup on average. We convert tensors to HiCOO by compressing indices, an extra 20% improvement is shown. Lastly, SIMD directives are applied to vectorize matrix rows, which gives an average of 22% speedup. With all these optimizations, HiCOO-MTTKRP doubles the performance of COO-MTTKRP for a single thread.

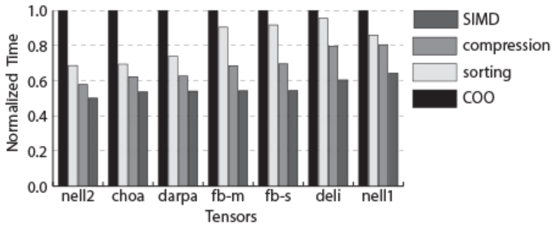
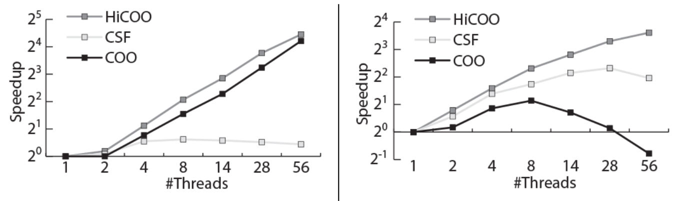


Fig. 7. HiCOO optimization breakdown on 3D tensors.

D. Thread Scalability

Figure 8 compares the thread scalability of COO, CSF, and HiCOO MTTKRPs using tensors fb-s and choa representing the behavior in the shortest and longest modes of a tensor. Generally, CSF-MTTKRP scales poorly in short modes: 9 out of all 12 tensors achieve the best performance in their shortest modes with ≤ 8 threads; while only 2 tensors achieve the best performance in their longest modes with ≤ 8 threads. COO-MTTKRP does not scale well in long modes: 7 out of 12 tensors achieve the best performance in their longest mode with ≤ 8 threads. The x-axis shows the number of threads and the y-axis shows the speedup over sequential MTTKRP. HiCOO-MTTKRP scales better than COO and CSF in most cases despite the extra lptr and lschr structures needed to support superblock scheduling. CSF-MTTKRP has a potentially low parallel degree for short modes and may uses locks, while COO-MTTKRP always employs privatization with an expensive reduction stage for long modes, which affects its scalability. These observations verify the results in Section VI-B.



(a) tensor fb-s in mode 3 (shortest mode)

(b) tensor choa in mode 1 (longest mode)

Fig. 8. Thread scalability of parallel COO, CSF, and HiCOO MTTKRPs on two representative cases.

E. Application

Figure 9 depicts a CANDECOMP/PARAFAC decomposition using alternating least squares algorithm (CP-ALS) [25] in all the three formats, where MTTKRP is the most expensive computational kernel. The speedup and compression rates are relative to CSF in ALLMODE setting, and "CSF-1" refers to CSF in ONEMODE setting. HiCOO achieves the best speedup on most tensors with comparable compression rate to CSF ONEMODE. COO does not gain any advantages either from performance or storage aspect. HiCOO-CPD outperforms 6.2× over COO-CPD and 2.1× over CSF-CPD on average.

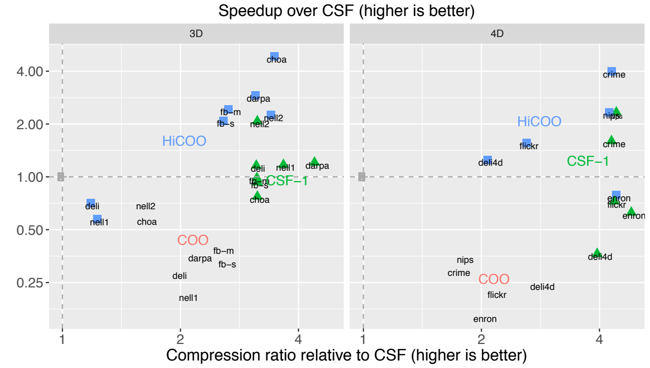


Fig. 9. CPD time and tensor storage comparison relative to CSF in ALLMODE setting.

VII. Related Work

Study on improving the performance of sparse problems has a long history, for example, some optimization methods for sparse matrices are enlightening and applicable to tensors [2–16]. Optimizing CPD and the MTTKRP operation has been well-studied recently [25, 44, 45]. Tensor Toolbox [43] and Tensorlab [23] implemented MTTKRP in the most popular COO format through multiple sparse tensor-vector products using MATLAB. SPLATT [19, 28] algorithmically improved MTTKRP by factoring out inner multiplications and proposed a more compressed CSF format. According to their experiments [28], SPLATT outperforms Tensor Toolbox, GigaTensor, and DFacTo (introduced below), becoming the one achieving the highest performance on multicore CPUs by far. GigaTensor [31] reformulated MTTKRP as a series of Hadamard products to utilize the massive parallelism of MapReduce. However, this algorithm is not suitable for multicore CPUs because of its high computational complexity. DFacTo [30] considered MTTKRP as a series of sparse matrix-vector multiplications for distributed systems, however, it requires explicit matricization which takes non-negligible time. HyperTensor [29] investigated fine- and also coarse-grained parallel algorithms for distributed systems, while its MTTKRP implementation is based on COO format. A more recent work [22] proposed Flagged-COOrdinate (F-COO) format. As we state in Section I and III, CSF and F-COO formats can achieve compactness but not, simultaneously, mode-generic orientation. Some tensor formats are proposed for structural sparse tensors or specific tensor operations, such as "mode-generic and mode-specific" [20] and semi-COOrdinate (sCOO) [21, 46] formats for the ones with dense modes, Extended Karnaugh Map Representation (EKMR) for some other tensor operations.

Our work proposed a new compressed HiCOO format for general sparse tensors, which does not favor one mode over the others and preserves the mode-generic orientation. HiCOO is proposed to explore an alternative approach of sparse tensor formats. Compared to the state-of-the-art CSF from SPLATT [28, 34], HiCOO has the following advantages. First, HiCOO could exploit data locality for all tensor dimensions while CSF's tree structure limits this mostly to the leaf level. Second, HiCOO has more potential of its storage compression because of the avoidance of larger and indeterminate-length pointers compared to indices. We are pursuing more aggressive compression techniques, as proposed for sparse matrices [47], as part of our future work for HiCOO. Third, from our parallel strategies and the experiments, HiCOO-MTTKRP is less sensitive to irregularly-shaped tensors which frequently appear in real applications. Besides, our blocking strategy has similarities with cache tiling used in SPLATT, but is fine-grained and totally removes write conflicts. SPLATT uses coarse-grained parallelism which is quite similar to row parallelism in SpMV. This work is orthogonal to some optimization work for an MTTKRP sequence as a whole, such as [32, 48], we will consider integrating them as our future work.

VIII. Conclusion

"Flexible format
Of hierarchical sparse blocks
Small, and often fast"
– Haiku for HiCOO

HiCOO is a flexible, compact, and mode-generic format for general sparse tensors. It derives from but also improves upon COO by compressing the indices in units of sparse tensor blocks, which compresses the storage while promoting data locality. Our multicore-parallel HiCOO MTTKRP achieves remarkable speedups over COO and another state-of-the-art format, compressed sparse fiber (CSF) formats. HiCOO uses up to 2.5× less storage than COO format and comparable storage to one CSF representation. When used within CPD, we also observe speedups against COO- and CSF-based implementations.

Other platforms such as GPUs and distributed memory systems can also benefit from the data locality of HiCOO format and the two-level blocking to accelerate tensor algorithms. Our future work will explore the behavior of the performance critical parameters of HiCOO and predict their optimal choice by sampling nonzero distribution of an input tensor. We also plan to develop HiCOO variants of other widely used kernels, such as tensor-times-matrix multiplication (TTM). Our code is publicly available.[6]

Acknowledgments

We thank Tamara Kolda, Bora Uçar, and Shaden Smith for their constructive feedback. This research is supported by the U.S. National Science Foundation (NSF) Award Number 1533768, 2017–2018 IBM Ph.D. Fellowship Award, and the Laboratory Directed Research and Development program at Sandia National Laboratories, a multi-mission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

[6] https://github.com/hpcgarage/ParTI

References

[1] Y. Saad, "SPARSKIT: A basic tool kit for sparse matrix computation," CSRD, University of Illinois Urbana-Champagne, Urbana, IL, USA, Tech. Rep. CSRD TR 1029, 1990. [Online]. Available: http://www-users.cs.umn.edu/~saad/software/SPARSKIT/

[2] A. Azad, G. Ballard, A. Buluç, J. Demmel, L. Grigori, O. Schwartz, S. Toledo, and S. Williams, "Exploiting multiple levels of parallelism in sparse matrix-matrix multiplication," *SIAM Journal on Scientific Computing*, vol. 38, no. 6, pp. C624–C651, 2016.

[3] A. Buluç and J. R. Gilbert, "Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments," *SIAM Journal on Scientific Computing*, vol. 34, no. 4, pp. C170–C191, 2012.

[4] ——, "On the representation and multiplication of hypersparse matrices," in *2008 IEEE International Symposium on Parallel and Distributed Processing*, April 2008, pp. 1–11.

[5] P. Koanantakool, A. Azad, A. Buluç, D. Morozov, S. Y. Oh, L. Oliker, and K. Yelick, "Communication-avoiding parallel sparse-dense matrix-matrix multiplication," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2016, pp. 842–853.

[6] A. Azad and A. Buluç, "A work-efficient parallel sparse matrix-sparse vector multiplication algorithm," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2017, pp. 688–697.

[7] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of sparse matrix–vector multiplication on emerging multicore platforms," *Parallel Computing*, vol. 35, no. 3, pp. 178 – 194, 2009, revolutionary Technologies for Acceleration of Emerging Petascale Applications.

[8] J. W. Choi, A. Singh, and R. W. Vuduc, "Model-driven autotuning of sparse matrix-vector multiply on GPUs," in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '10, 2010, pp. 115–126.

[9] R. W. Vuduc, "Automatic performance tuning of sparse matrix kernels," Ph.D. dissertation, University of California, Berkeley, CA, USA, January 2004.

[10] R. Vuduc, J. W. Demmel, and K. A. Yelick, "OSKI: A library of automatically tuned sparse matrix kernels," *Journal of Physics: Conference Series*, vol. 16, no. 1, p. 521, 2005.

[11] D. Merrill and M. Garland, "Merge-based parallel sparse matrix-vector multiplication," in *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2016, pp. 678–689.

[12] W. Liu and B. Vinter, "CSR5: An efficient storage format for cross-platform sparse matrix-vector multiplication," in *Proceedings of the 29th ACM International Conference on Supercomputing*, ser. ICS '15. ACM, 2015, pp. 339–350. [Online]. Available: http://doi.acm.org/10.1145/2751205.2751209

[13] Y. Nagasaka, A. Nukada, and S. Matsuoka, "High-performance and memory-saving sparse general matrix-matrix multiplication for NVIDIA Pascal GPU," in *2017 46th International Conference on Parallel Processing (ICPP)*, Aug 2017, pp. 101–110.

[14] W. Liu and B. Vinter, "A framework for general sparse matrix-matrix multiplication on GPUs and heterogeneous processors," *Journal of Parallel and Distributed Computing*, vol. 85, no. C, pp. 47–61, nov 2015. [Online]. Available: http://dx.doi.org/10.1016/j.jpdc.2015.06.010

[15] C. Yang, A. Buluç, and J. Owens, "Design principles for sparse matrix multiplication on the GPU," *24th International European Conference on Parallel and Distributed Computing*, 2018.

[16] J. Li, G. Tan, M. Chen, and N. Sun, "SMAT: An input adaptive auto-tuner for sparse matrix-vector multiplication," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13. New York, NY, USA: ACM, 2013, pp. 117–126. [Online]. Available: http://doi.acm.org/10.1145/2491956.2462181

[17] Y. Zhao, J. Li, C. Liao, and X. Shen, "Bridging the gap between deep learning and sparse matrix format selection," in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '18. New York, NY, USA: ACM, 2018, pp. 94–108. [Online]. Available: http://doi.acm.org/10.1145/3178487.3178495

[18] B. W. Bader and T. G. Kolda, "Efficient MATLAB computations with sparse and factored tensors," *SIAM Journal on Scientific Computing*, vol. 30, no. 1, pp. 205–231, December 2007.

[19] S. Smith and G. Karypis, "Tensor-matrix products with a compressed sparse tensor," in *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*. ACM, 2015, p. 7.

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

[21] J. Li, Y. Ma, C. Yan, and R. Vuduc, "Optimizing sparse tensor times matrix on multi-core and many-core architectures," in *Proceedings of the Sixth Workshop on Irregular Applications: Architectures and Algorithms*, ser. IAˆ3 '16. Piscataway, NJ, USA: IEEE Press, 2016, pp. 26–33. [Online]. Available: https://doi.org/10.1109/IA3.2016.10

[22] B. Liu, C. Wen, A. D. Sarwate, and M. M. Dehnavi, "A unified optimization approach for sparse tensor operations on GPUs," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, Sept 2017, pp. 47–57.

[23] L. Sorber, M. Van Barel, and L. De Lathauwer, "Tensorlab (Version v3.0)," Available online, 2014.

[24] I. Jeon, E. E. Papalexakis, U. Kang, and C. Faloutsos, "HaTen2: Billion-scale tensor decompositions," in *IEEE International Conference on Data Engineering (ICDE)*, 2015.

[25] T. G. Kolda and B. W. Bader, "Tensor decompositions and applications," *SIAM Review*, vol. 51, no. 3, pp. 455–500, 2009.

[26] L. Grasedyck, D. Kressner, and C. Tobler, "A literature survey of low-rank tensor approximation techniques," *GAMM-Mitteilungen*, vol. 36, no. 1, pp. 53–78, 2013.

[27] A. Cichocki, N. Lee, I. V. Oseledets, A. Phan, Q. Zhao, and D. Mandic, "Low-rank tensor networks for dimensionality reduction and large-scale optimization problems: Perspectives and challenges part 1," *ArXiv e-prints*, Sep. 2016.

[28] S. Smith, N. Ravindran, N. Sidiropoulos, and G. Karypis, "SPLATT: Efficient and parallel sparse tensor-matrix multiplication," in *Proceedings of the 29th IEEE International Parallel & Distributed Processing Symposium*, ser. IPDPS, 2015.

[29] O. Kaya and B. Uçar, "Scalable sparse tensor decompositions in distributed memory systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. New York, NY, USA: ACM, 2015, pp. 77:1–77:11.

[30] J. H. Choi and S. Vishwanathan, "DFacTo: Distributed factorization of tensors," in *Advances in Neural Information Processing Systems 27*, Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Weinberger, Eds. Curran Associates, Inc., 2014, pp. 1296–1304.

[31] U. Kang, E. Papalexakis, A. Harpale, and C. Faloutsos, "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*, ser. KDD '12. New York, NY, USA: ACM, 2012, pp. 316–324.

[32] J. Li, J. Choi, I. Perros, J. Sun, and R. Vuduc, "Model-driven sparse CP decomposition for higher-order tensors," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2017, pp. 1048–1057.

[33] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, "Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks," in *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '09. New York, NY, USA: ACM, 2009, pp. 233–244. [Online]. Available: http://doi.acm.org/10.1145/1583991.1584053

[34] S. Smith and G. Karypis, "SPLATT: The Surprisingly ParalleL spArse Tensor Toolkit (Version 1.1.1)," http://cs.umn.edu/~splatt/, 2016.

[35] ——, "Accelerating the Tucker decomposition with compressed sparse tensors," in *European Conference on Parallel Processing*. Springer, 2017.

[36] H. Wang, W. Liu, K. Hou, and W.-c. Feng, "Parallel transposition of sparse data structures," in *Proceedings of the 2016 International Conference on Supercomputing*, ser. ICS '16. ACM, 2016, pp. 33:1–33:13. [Online]. Available: http://doi.acm.org/10.1145/2925426.2926291

[37] G. M. Morton, "A computer oriented geodetic data base; and a new technique in file sequencing," Ottawa, Canada: IBM Ltd, Tech. Rep., 1966.

[38] S. Smith, J. Park, and G. Karypis, "Sparse tensor factorization on many-core processors with high-bandwidth memory," *31st IEEE International Parallel & Distributed Processing Symposium (IPDPS'17)*, 2017.

[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. [Online]. Available: http://frostt.io/

[40] I. Jeon, E. E. Papalexakis, and C. F. U Kang, "HaTen2: Billion-scale tensor decompositions (Version 1.0)," Available from http://datalab.snu.ac.kr/haten2/, 2015.

[41] I. Perros, E. E. Papalexakis, F. Wang, R. Vuduc, E. Searles, M. Thompson, and J. Sun, "SPARTan: Scalable PARAFAC2 for large & sparse data," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '17. New York, NY, USA: ACM, 2017, pp. 375–384. [Online]. Available: http://doi.acm.org/10.1145/3097983.3098014

[42] J. Li, Y. Ma, and R. Vuduc, "ParTI!: A parallel tensor infrastructure for multicore CPU and GPUs (Version 0.1.0)," https://github.com/hpcgarage/ParTI, 2016.

[43] B. W. Bader, T. G. Kolda *et al.*, "MATLAB Tensor Toolbox (Version 2.6)," Available online, February 2015.

[44] A. Cichocki, "Era of big data processing: A new approach via tensor networks and tensor decompositions," *CoRR*, vol. abs/1403.2048, 2014. [Online]. Available: http://arxiv.org/abs/1403.2048

[45] N. D. Sidiropoulos, L. De Lathauwer, X. Fu, K. Huang, E. E. Papalexakis, and C. Faloutsos, "Tensor decomposition for signal processing and machine learning," *IEEE Transactions on Signal Processing*, vol. 65, no. 13, pp. 3551–3582, July 2017.

[46] Y. Ma, J. Li, X. Wu, C. Yan, J. Sun, and R. Vuduc, "Optimizing sparse tensor times matrix on GPUs," *Journal of Parallel and Distributed Computing*, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731518305161

[47] J. Willcock and A. Lumsdaine, "Accelerating sparse matrix computations via data compression," in *International Conference on Supercomputing*, June 2006, pp. 307–316.

[48] O. Kaya and B. Uçar, "Parallel CP decomposition of sparse tensors using dimension trees." *Inria - Research Centre Grenoble – Rhone-Alpes*, RR-8976, 2016.

Artifact Description Appendix: HiCOO: Hierarchical Storage of Sparse Tensors

A. Abstract

This appendix details how to construct the HiCOO format and run both serial and multicore parallel matricized tensor-times-Khatri-Rao product (MTTKRPs) and CANDECOMP/PARAFAC decomposition using HiCOO and COO formats on real-world tensors.

B. Description

1) Check-list (artifact meta information):

- **Algorithm: MTTKRP, CPD**
- **Program: C program with OpenMP**
- **Compilation: Compile with CMake**
- **Data set: the Formidable Repository of Open Sparse Tensors and Tools (FROSTT) and HaTen2 dataset**
- **Hardware: General multicore CPUs**
- **Execution: See details below**
- **Output: updated matrices and execution times.**
- **Publicly available?: Yes**

2) How software can be obtained: The experiments in this paper can be reproduced from the Github repo located at https://github.com/hpcgarage/ParTI.

3) Hardware dependencies: Multicore CPU platforms

4) Software dependencies: C Compiler, CMake, OpenBLAS library

5) Datasets: the Formidable Repository of Open Sparse Tensors and Tools (FROSTT) http://frostt.io and HaTen2 dataset https://datalab.snu.ac.kr/haten2/.

C. Installation

- Create a file by *touch build.config* to define "OpenBLAS_DIR"
- Type *./build.sh*
- Check *build/* for resulting library
- Check *build/tests/* for resulting tests

D. Experiment workflow

Run the following tests.

- Construct HiCOO: ./build/tests/convert_hicoo –help will give detailed options.
- COO-MTTKRP: *./build/tests/mttkrp –help* will give detailed options.
- HiCOO-MTTKRP: *./build/tests/mttkrp_hicoo_matrixtiling –help* will give detailed options.
- COO-CPD: *./build/tests/cpd –help* will give detailed options.
- HiCOO-CPD: *./build/tests/cpd_hicoo –help* will give detailed options.

All the running programs support both serial and parallel implementations by setting dev_id = -2 for serial code, dev_id = -1 for parallel code.

E. Evaluation and expected result

- HiCOO has smaller storage than the COO format.
- Parallel HiCOO-MTTKRP achieves better performance than COO-MTTKRP and most CSF-MTTKRP from Splatt package (http://cs.umn.edu/~splatt/) on the above dataset.
- Parallel HiCOO-CPD achieves better performance than COO-CPD and most CSF-CPD from Splatt on the above dataset.

APPENDIX

We use this appendix to show pseudocode, explain more about the improvement of HICOO over CSB format for sparse matrices, then show additional analysis of HICOO parameters and its behavior on another manycore architecture.

A. Algorithm pseudocode

The MTTKRP algorithms are illustrated in Algorithm 1 and 2.

Algorithm 1 Sequential COO-MTTKRP algorithm ([18]).

```
Input: A third-order sparse tensor $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$, dense matrices $\mathbf{B} \in \mathbb{R}^{J \times R}$, $\mathbf{C} \in \mathbb{R}^{K \times R}$;
Output: Updated dense matrix $\tilde{\mathbf{A}} \in \mathbb{R}^{I \times R}$;
                                        ▷ $\tilde{\mathbf{A}} \leftarrow \mathcal{X}_{(1)}(\mathbf{C} \odot \mathbf{B})$
for $x = 1, \dots, M$ do
    $i = \text{inds}(x, 1)$, $j = \text{inds}(x, 2)$, $k = \text{inds}(x, 3)$;
    for $r = 1, \dots, R$ do
        $\tilde{A}(i, r) += \text{val}(x) C(k, r) B(j, r)$
return $\tilde{\mathbf{A}}$;
```

Algorithm 2 Sequential HICOO-MTTKRP algorithm.

```
Input: A third-order HICOO sparse tensor $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$, dense matrices $\mathbf{B} \in \mathbb{R}^{J \times R}$, $\mathbf{C} \in \mathbb{R}^{K \times R}$, block size $B$;
Output: Updated dense matrix $\tilde{\mathbf{A}} \in \mathbb{R}^{I \times R}$;
                                        ▷ $\tilde{\mathbf{A}} \leftarrow \mathcal{X}_{(1)}(\mathbf{C} \odot \mathbf{B})$
for $b = 1, \dots, n_b$ do                                          ▷ block $b$
    $bi = \text{binds}(b, 1)$, $bj = \text{binds}(b, 2)$, $bk = \text{binds}(b, 3)$;
    $\mathbf{A}_b = \mathbf{A} + bi \cdot B \cdot R$; $\mathbf{B}_b = \mathbf{B} + bj \cdot B \cdot R$; $\mathbf{C}_b = \mathbf{C} + bk \cdot B \cdot R$;
    for $x = bptr[b], \dots, bptr[b+1] - 1$ do                     ▷ entry $x$
        $ei = \text{einds}(x, 1)$, $ej = \text{einds}(x, 2)$, $ek = \text{einds}(x, 3)$
        for $r = 1, \dots, R$ do
            $\tilde{A}_b(ei, r) += \text{val}(x) C_b(ek, r) B_b(ej, r)$
return $\tilde{\mathbf{A}}$;
```

B. Comparison with CSB

The Compressed Sparse Blocks (CSB) format is proposed by Buluç et al. [33] for sparse matrices. Two critical features of CSB inspired us: 1) it targets large matrices with hyper-sparse *matrix blocks*; 2) CSB is mode-generic and allows efficient computation of both Sparse Matrix-Vector Multiplication (SpMV) and Sparse Matrix-Transpose-Vector Multiplication (SpMTV) using a single CSB representation.

We find that small blocks are more suitable for sparse tensors. However, small blocks pose two issues of a straightforward extension of CSB. 1) First, CSB is not storage-efficient for small blocks as stated in [33]. Small blocks certainly lead to more compressed nonzero indices by being capable of using fewer bits, however, the storage of block indices, which is saved contiguously as a dense array, increases much faster. Therefore, the overall storage of CSB is not beneficial from small blocks. 2) Secondly, small blocks imply a relatively fine-grained parallelism. On our target multicore platforms, heavyweight CPU threads are not efficient to schedule a huge number of threads with a small workload of each. HICOO improves from CSB idea by block index compression and an extra superblock level for efficient CPU multithreading.

C. More experimental results

1) Storage Space: Table IV compares the storage space of COO, CSF, F-COO, and HICOO formats, along with HICOO's compression rates over COO and its α_b values. We show two settings for CSF format, ALLMODE (using all CSF trees) and ONEMODE (using only one CSF tree). For F-COO, all representations must be stored. HICOO uses less space than ALLMODE CSF and F-COO formats on all tensors, up to 2.5× less storage than COO format, and comparable storage with ONEMODE CSF. HICOO consumes more space than COO on tensors deli, deli4d and nell1. As discussed in Section V-C, the value of the block ratio should satisfy $\alpha_b < 0.45$ (3D) or 0.5 (4D) for a tensor to take less space in HICOO than in COO. The α_b values of these three tensors are much larger than them, so it is not surprising that their HICOO representations are larger. These observations suggest that a simple test, perhaps based on sampling, could be used to determine when to use COO versus HICOO (see also Section V-C).

TABLE IV
SPARSE TENSOR SPACE COMPARISON IN DIFFERENT FORMATS.

Tensors	COO (MiB)	CSF (MiB) ALL	CSF (MiB) ONE	F-COO (MiB)	HICOO (MiB)	Compress Rate	α_b Values
choa	411	666	212	935	**192**	2.14	0.02
darpa	434	958	**218**	986	308	1.41	0.22
nell2	1150	1850	589	2667	**543**	2.12	0.02
fb-m	1480	3760	**1200**	3453	1420	1.04	0.42
fb-s	2080	5410	**1720**	4854	2100	0.99	0.46
deli	2090	4120	**1320**	4861	3490	0.60	0.99
nell1	2140	4430	**1210**	4981	3610	0.59	1.00
crime	102	176	**41**	328	**41**	2.49	0.00
nips	59	106	**24**	191	25	2.36	0.02
enron	1010	2030	**421**	3334	460	2.20	0.04
flickr	2100	4540	**1040**	6944	1740	1.21	0.36
deli4d	2610	7340	**1860**	8619	3540	0.74	0.80

2) Performance parameters: Figure 10 illustrates the trends of parameters α_b and c_b with performance and last-level cache behavior on third-order tensors. We use the speedup of HICOO-MTTKRP over CSF in ALLMODE setting and the last-level cache hit rates respectively. This figure verifies that smaller α_b and larger c_b are good for HICOO performance and its MTTKRP data locality.

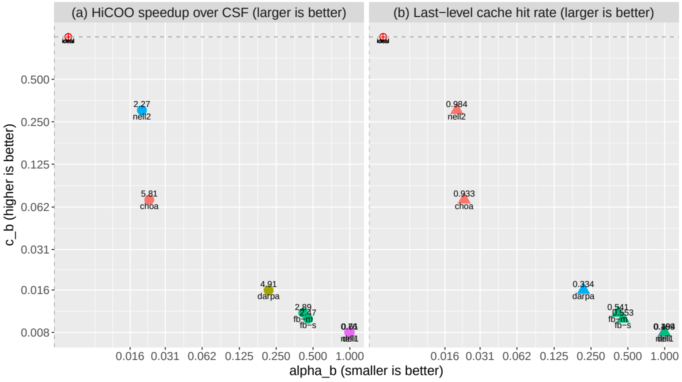
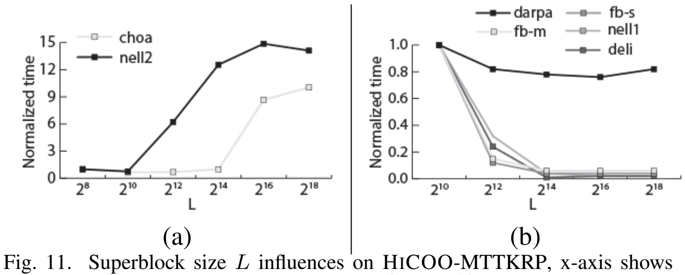


Fig. 10. α_b and c_b relations with (a) the speedup of HICOO over CSF in ALLMODE setting and (b) last-level cache hit rates.

3) Superblock Size L: The choice of superblock size L affects the number of independent tasks available in HICOO-MTTKRP as discussed in Section V-C. Figure 11 shows how the execution time of a HICOO-MTTKRP sequence varies with L on 3D tensors. The x-axis shows L values, while the y-axis shows the execution times of parallel HiCOO-MTTKRP normalized to that of $L = 2^8$ or 2^{10}. Tensors `nell2` and `choa` show their best performance at $L = 2^{10}$ in Figure 11(a). By contrast, MTTKRP times on the other tensors in Figure 11(b) tend to decrease asymptotically with L, with some small variability beyond a certain point ($L = 2^{14}$). The main difference between these two groups is nonzero density, which is relatively high for tensors in Figure 11(a) and low in Figure 11(b). The guidance in Section V-C helps users to fast identify the optimal L.



(a) (b)

Fig. 11. Superblock size L influences on HiCOO-MTTKRP, x-axis shows L values, times are normalized to $L = 2^8$ are 2^{10} for these two figures respectively. Lower is better.

4) Experiments on KNL: We also test HiCOO-MTTKRP on a Intel Xeon Phi Processor 7250 (“KNL”) platform using the cache mode for multi-channel dynamic random access memory (MCDRAM). Figure 12 shows the speedup of HiCOO over COO format on KNL using 68 threads. A HiCOO-MTTKRP sequence achieves $1.0 - 97.4\times$ speedup and comparable performance with CSF-MTTKRP on the two tensors from Smith et al.’s recent work [38]. HiCOO-MTTKRP also achieves $0.25-3.27\times$ speedup on KNL over on Haswell multicore platform using 56 threads. This experiment gives a proof-of-concept that other accelerators, e.g., GPUs, can also benefit from HiCOO format with its good thread scalability.

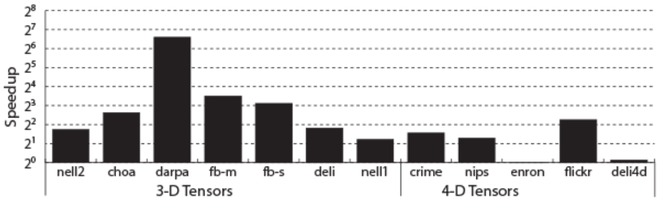


Fig. 12. HiCOO-MTTKRP speedup of HiCOO over COO on a KNL.