

ALTO: Adaptive Linearized Storage of Sparse Tensors

Ahmed E. Helal
Intel Labs
ahmed.helal@intel.com

Jan Laukemann
Intel Labs
jan.laukemann@intel.com

Fabio Checconi
Intel Labs
fabio.checconi@intel.com

Jesmin Jahan Tithi
Intel Labs
jesmin.jahan.tithi@intel.com

Teresa Ranadive
Laboratory for Physical Sciences
tranadive@lps.umd.edu

Fabrizio Petrini
Intel Labs
fabrizio.petrini@intel.com

Jeewhan Choi
University of Oregon
jeec@uoregon.edu

ABSTRACT

The analysis of high-dimensional sparse data is becoming increasingly popular in many important domains. However, real-world sparse tensors are challenging to process due to their irregular shapes and data distributions. We propose the Adaptive Linearized Tensor Order (ALTO) format, a novel mode-agnostic (general) representation that keeps neighboring nonzero elements in the multi-dimensional space close to each other in memory. To generate the indexing metadata, ALTO uses an adaptive bit encoding scheme that trades off index computations for lower memory usage and more effective use of memory bandwidth. Moreover, by decoupling its sparse representation from the irregular spatial distribution of nonzero elements, ALTO eliminates the workload imbalance and greatly reduces the synchronization overhead of tensor computations. As a result, the parallel performance of ALTO-based tensor operations becomes a function of their inherent data reuse. On a gamut of tensor datasets, ALTO outperforms an oracle that selects the best state-of-the-art format for each dataset, when used in key tensor decomposition operations. Specifically, ALTO achieves a geometric mean speedup of 8× over the best mode-agnostic (coordinate and hierarchical coordinate) formats, while delivering a geometric mean compression ratio of 4.3× relative to the best mode-specific (compressed sparse fiber) formats.

CCS CONCEPTS

• **Mathematics of computing** → **Mathematical software performance**; • **Computing methodologies** → **Parallel algorithms**.

KEYWORDS

Sparse tensors, tensor decomposition, MTTKRP, multi-core CPU

ACM Reference Format:

Ahmed E. Helal, Jan Laukemann, Fabio Checconi, Jesmin Jahan Tithi, Teresa Ranadive, Fabrizio Petrini, and Jeewhan Choi. 2021. ALTO: Adaptive Linearized Storage of Sparse Tensors. In *2021 International Conference on Supercomputing (ICS '21)*, June 14–17, 2021, Virtual Event, USA. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3447818.3461703>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICS '21, June 14–17, 2021, Virtual Event, USA

© 2021 Association for Computing Machinery.

ACM ISBN 978-1-4503-8335-6/21/06...\$15.00

<https://doi.org/10.1145/3447818.3461703>

1 INTRODUCTION

Many critical application domains such as healthcare [22, 59], cybersecurity [15, 25], data mining [27, 42], and machine learning [3, 46] produce and manipulate massive amounts of high-dimensional data. Such datasets can naturally be represented as sparse tensors, which store the values of the nonzero tensor elements along with indexing metadata that denote the position of each nonzero in the tensor. Therefore, a fundamental problem in sparse tensor computations involves determining how to store, group, and organize the nonzero elements to 1) reduce memory storage, 2) improve data locality, 3) increase parallelism, and 4) decrease workload imbalance and synchronization overhead. Since tensor algorithms perform computations along different mode (i.e., dimension) orientations, practical sparse tensor formats must be mode-agnostic to deliver acceptable performance and scalability across all modes. Because real-world sparse tensors are highly irregular in terms of their shape, dimensions, and distribution of nonzero elements, achieving these (oftentimes conflicting) goals is challenging.

To tackle this problem, researchers have proposed many sparse tensor formats [4, 5, 29–31, 40, 41, 45, 50, 52], which can be classified based on their encoding of the multi-dimensional coordinates into list-, block-, and tree-based formats [12]. List-based tensor representations, such as the simple coordinate (COO) format, explicitly store the nonzero elements along with their coordinates (i.e., the indices of all dimensions). Therefore, they are agnostic to the different mode orientations of tensor algorithms and, as a result, they remain the de facto sparse tensor storage [12] in many libraries (e.g., Tensor Toolbox [4], Tensorflow [1], and Tensorlab [57]). However, the list-based COO format does not impose any order on the multi-dimensional data and it suffers from a significant synchronization overhead to resolve the update/write conflicts across threads [31].

Prior block-based sparse tensor representations employ multi-dimensional tiling schemes to further compress the COO format [29, 30]. However, the efficacy of this hierarchical COO (HiCOO) storage completely depends on the characteristics of target tensors (such as their shape, density, and data distribution) as well as the format parameters (e.g., block size). In addition, the resulting parallel schedule of HiCOO blocks can suffer from limited parallelism and scalability due to conflicting updates across blocks.

Several proposals use tree-based data structures to extend traditional compressed matrix formats, such as the compressed sparse row (CSR) format, to higher-order tensors. The most popular example of these storage representations is the compressed sparse fiber (CSF) format [52], which uses multiple arrays of index pointers to compress the multi-dimensional indices of nonzero elements.

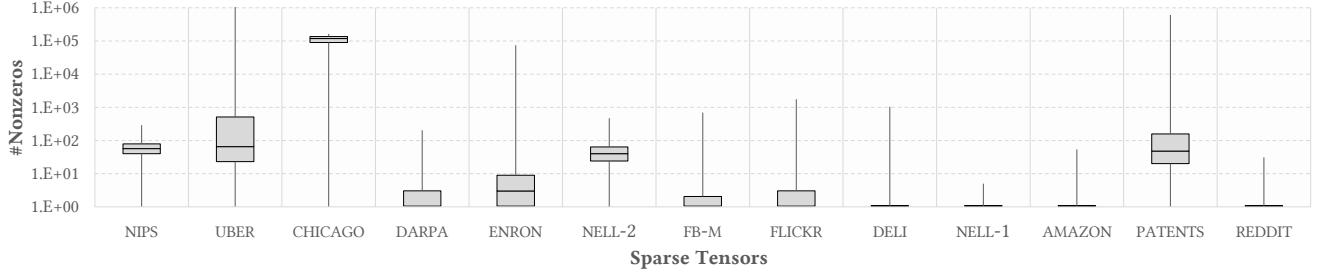


Figure 1: A box plot of the data (nonzero elements) distribution across the multi-dimensional blocks (subspaces) of the hierarchical coordinate storage [29]. The multi-dimensional subspace size is 128^N , where N is the number of dimensions (modes), as per prior work [55]. The sparse tensors are sorted in an increasing order of their size (number of nonzero elements).

While CSF-based formats can reduce memory traffic, they are mode-specific, i.e., they are oriented to favor a specific order of tensor modes. As the tensor order increases, these mode-specific formats require excessive memory to store multiple tensor copies [51] and/or different code implementations for computing along distinct modes [50]. Moreover, such a compressed, coarse-grained format can suffer from significant workload imbalance and limited scalability, especially in irregularly shaped tensors with short modes [29].

In summary, the above block- and tree-based approaches for sparse tensor storage rely on *clustering the nonzero elements based on their location in the multi-dimensional space* and then partitioning this space into *non-overlapping regions* to generate a compressed indexing metadata. Hence, they are constrained by the spatial distribution of nonzero elements. To this end, Figure 1 shows the distribution of nonzero elements in the multi-dimensional space for a set of 3D and 4D sparse tensors.¹ Specifically, it demonstrates that the number of nonzero elements per block fluctuates widely (note the use of logarithmic scale). As the sparsity of tensors increases (e.g., DELI, NELL-1, AMAZON, and REDDIT tensors), the location-based clustering fails to compress tensors and introduces a substantial memory overhead. As a result, the parallel performance and scalability of such location-based formats can be severely impacted by the irregular data distributions and unstructured sparsity patterns that typically emerge in higher-order sparse tensors.

1.1 Adaptive Linearized Tensor Order

We propose the *Adaptive Linearized Tensor Order* (ALTO) format, a mode-agnostic storage system for sparse tensors that addresses the irregularity in sparse tensor computations and the performance bottlenecks on modern parallel architectures. ALTO organizes and stores the nonzero elements of a given tensor in a one-dimensional data structure along with compact indexing metadata, such that neighboring nonzero elements in the tensor are close to each other in memory. Most importantly, it generates the indexing metadata using an adaptive bit encoding scheme based on the shape and dimensions of target tensors. Such an adaptive format trades off index computations (i.e., de-linearization) for lower memory usage and more efficient use of the effective memory bandwidth.

Unlike prior compressed [29, 52] and linearized [21] sparse tensor formats, ALTO not only improves data locality across all mode orientations but also *eliminates workload imbalance* and greatly

reduces synchronization cost, which have traditionally limited the performance and scalability of irregular sparse tensor computations. Moreover, it generates perfectly balanced partitions for effective parallel execution. Although each nonzero tensor element is strictly mapped to one partition, the subspace coordinates of ALTO partitions may overlap. To resolve update conflicts across these potentially overlapping partitions, ALTO locates their positions in the multi-dimensional space and automatically selects the appropriate synchronization mechanism based on the data reuse of target tensors. As a result, ALTO delivers superior performance over state-of-the-art sparse tensor formats. In what follows, we summarize the contributions of this work:

- We introduce ALTO, a novel sparse tensor format for high-performance and scalable execution of tensor operations. Unlike prior location-based clustering approaches for compressed sparse tensor storage, ALTO uses a single (mode-agnostic) tensor representation that improves data locality, eliminates workload imbalance, and greatly reduces memory usage and synchronization overhead, *regardless of the data distribution in the multi-dimensional space* (§3).
- We propose an adaptive synchronization mechanism to efficiently resolve conflicting updates (writes) across threads based on the average data reuse of target sparse tensors (§3).
- We present effective parallel algorithms to perform Matrix-Times Tensor Khatri-Rao Product (MTTKRP) operations, the main tensor decomposition performance bottleneck [5, 11, 24, 52], on sparse tensors stored via ALTO (§3).
- We demonstrate that ALTO outperforms the state-of-the-art sparse tensor formats via experimental evaluation over a variety of real-world datasets. ALTO achieves a geometric mean speedup of $8\times$ over the best mode-agnostic format and a geometric mean compression ratio of $4.3\times$ over the best mode-specific format (§4).

2 BACKGROUND

We begin with a brief overview of tensor decomposition methods and related notations. The work by Kolda and Bader [4, 26] provides a more in-depth discussion of tensor algorithms and applications.

2.1 Notations

Tensors are the higher-order generalization of matrices. An N -dimensional tensor is said to have N modes and is called a mode- N tensor. The following notations are used in this paper:

¹The implementation of HiCOO [29], a block-based COO format, only supports 3D and 4D sparse tensors and it does not auto-tune the blocking sizes. We include ten datasets from [29], and add three large-scale datasets with billions of elements.

- (1) *Scalars* are denoted by lower case letters (e.g., a).
- (2) *Vectors* are mode-1 tensors denoted by bold lower case letters (e.g., \mathbf{a}). The i^{th} element of a vector \mathbf{a} is denoted by a_i .
- (3) *Matrices* are mode-2 tensors denoted by bold capital letters (e.g., \mathbf{A}). If \mathbf{A} is a $I \times J$ matrix, it can also be denoted as $\mathbf{A} \in \mathbb{R}^{I \times J}$, and its element at index (i, j) is denoted as $a_{i,j}$.
- (4) *Higher-order tensors* are denoted by Euler script letters (e.g., \mathcal{X}). A mode- N tensor whose dimensions are $I_1 \times I_2 \times \dots \times I_N$ can be denoted as $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \dots \times I_N}$, and its element at index (i_1, i_2, \dots, i_N) is denoted as x_{i_1, i_2, \dots, i_N} .
- (5) *Fibers* are the higher-order analogue of matrix rows and columns. A mode- n fiber is defined by fixing every mode except the n^{th} mode. For example, a matrix column is defined by fixing the second mode, and is therefore a mode-1 fiber. A fiber is denoted by using a colon for the non-fixed mode (e.g., the j^{th} column of a matrix \mathbf{A} is denoted by $\mathbf{a}_{:,j}$).
- (6) *Slices* are the lower-order components of a tensor which result from fixing all but two modes. For example, the slices of a mode-3 tensor \mathcal{X} are matrices (such as $\mathcal{X}_{i,:}$ and $\mathcal{X}_{:,j,:}$).

2.2 Canonical Polyadic Decomposition

The Canonical Polyadic Decomposition (CPD) is a widely used type of tensor factorization, in which a mode- N tensor \mathcal{X} is approximated by the sum of R outer products of N vectors. Each outer product is called a *rank-1 component*, while the sum of the R components is said to be a *rank- R decomposition* of \mathcal{X} . The vectors forming the rank-1 outer products each correspond to a particular tensor mode. We may arrange the R vectors corresponding to each of the N modes into N different *factor matrices* so that the decomposition of \mathcal{X} is the outer product of these matrices. For example, if $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$, we may write a decomposition of \mathcal{X} in terms of factor matrices $\mathbf{A} \in \mathbb{R}^{I \times R}$, $\mathbf{B} \in \mathbb{R}^{J \times R}$, and $\mathbf{C} \in \mathbb{R}^{K \times R}$, where the columns of \mathbf{A} (resp. \mathbf{B} and \mathbf{C}) are the vectors used in forming the R outer products along mode-1 (resp. 2 and 3).

The CPD-Alternating Least Squares (CPD-ALS) method is a popular tensor decomposition algorithm. During each ALS iteration, one alternates between updating each of the individual factor matrices, i.e., updating a factor matrix to yield the best approximation of \mathcal{X} when all other factor matrices are fixed. The most expensive part of CPD-ALS, along with many other tensor algorithms, is the matrix-times Khatri-Rao product (MTTKRP) operation [52].

The MTTKRP operation involves two basic subroutines:

- (1) *Tensor matricization* – a process where a tensor is unfolded or flattened into a matrix. Moreover, the mode- n matricization of a tensor \mathcal{X} , denoted $\mathbf{X}_{(n)}$, is obtained by laying out the mode- n fibers of \mathcal{X} as the columns of $\mathbf{X}_{(n)}$. Hence, when $\mathbf{X}_{(n)}$ is multiplied by the Khatri-Rao product (see below), the tensor indices associated with the q -th column of $\mathbf{X}_{(n)}$ match those given by the rows of the factor matrices used to form the q -th row of the Khatri-Rao product.
- (2) *Khatri-Rao product* [33] – the “matching column-wise” Kronecker product between two matrices. That is, given matrices $\mathbf{B} \in \mathbb{R}^{J \times R}$ and $\mathbf{C} \in \mathbb{R}^{K \times R}$, their Khatri-Rao product \mathbf{K} , denoted $\mathbf{K} = \mathbf{B} \odot \mathbf{C}$, where \mathbf{K} is a $(J \cdot K) \times R$ matrix, is defined as: $\mathbf{B} \odot \mathbf{C} = [\mathbf{b}_1 \otimes \mathbf{c}_1 \ \mathbf{b}_2 \otimes \mathbf{c}_2 \ \dots \ \mathbf{b}_R \otimes \mathbf{c}_R]$.

For a mode-3 tensor \mathcal{X} , the mode-1 MTTKRP operation can be expressed as $\mathbf{X}_{(1)} (\mathbf{B} \odot \mathbf{C})$. Typically, MTTKRP operations along all modes are performed 10–100 times in one tensor decomposition calculation. Since these MTTKRP operations are similar, we only discuss mode-1 MTTKRP in this paper.

3 ALTO FORMAT

Real-world sparse tensors, which emerge in high-dimensional data analytics, are challenging to efficiently encode and represent as they suffer from highly irregular shapes and data distributions as well as unstructured sparsity patterns. For example, one mode of a tensor may represent a massive user database while another mode represents their demographic information, their interactions, or their (potentially incomplete) consumer preferences/ratings for a set of products [48].

Thus, the proposed ALTO format uses an adaptive (data-aware) recursive partitioning of the high-dimensional space that represents a given sparse tensor to generate a mode-agnostic linearized index, which maps a point (nonzero element) in this Cartesian space to a point on a compact line. Specifically, ALTO splits every mode into multiple regions based on the mode length, such that each distinct mode has a variable number of regions to adapt to the unequal cardinalities of different modes and to minimize the storage requirements. This adaptive linearization and recursive partitioning of the multi-dimensional space ensures that neighboring points in space are close to each other on the resulting compact line, thereby maintaining the inherent data locality of tensor algorithms. Moreover, the ALTO format is not only locality-friendly, but also parallelism-friendly as it decomposes the multi-dimensional space into perfectly balanced (in terms of workload) subspaces. Further, it intelligently arranges the modes in the derived subspaces based on their cardinality (dimension length) to further reduce the overhead of resolving the write conflicts that typically occur in parallel sparse tensor computations.

What follows is a detailed description and discussion of the ALTO format generation (§3.1) and the workload partitioning and scheduling methods (§3.2) using a concrete walk-through example. In addition, we present the ALTO-based sequential and parallel algorithms as well as adaptive conflict resolution mechanisms for efficient execution of illustrative sparse tensor operations on parallel shared-memory platforms (§3.3).

3.1 ALTO Tensor Generation

Formally, an ALTO tensor $\mathcal{X} = \{x_1, x_2, \dots, x_M\}$ is an ordered set of nonzero elements, where each element $x_i = \langle v_i, p_i \rangle$ is represented by a value v_i and a position p_i . The position p_i corresponds to a compact mode-agnostic encoding of the indexing metadata, which is used to quickly generate the tuple (i_1, i_2, \dots, i_N) that locates a nonzero element in the multi-dimensional Cartesian space.

The generation of an ALTO tensor is carried out in two stages: linearization and ordering. First, ALTO constructs the indexing metadata using a compressed encoding scheme, based on the cardinalities of tensor modes, to map each nonzero element to a position on a compact line. Second, it arranges the nonzero elements in linearized storage according to their line positions, i.e., the values of their ALTO index. Typically, the ordering stage dominates

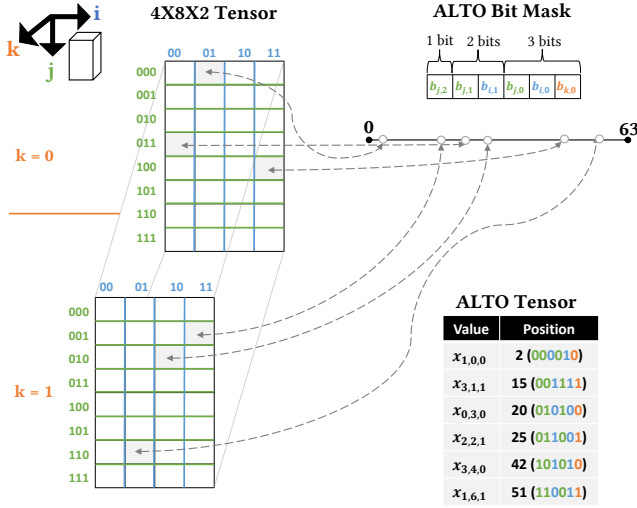


Figure 2: An example of the ALTO sparse encoding and representation for a three-dimensional tensor.

the format conversion/generation time. However, compared to the location-based sparse tensor formats [29, 30, 40, 41, 50, 52], ALTO requires a minimal generation time because ordering the linearized tensors incurs a fraction of the cost required to sort the multi-dimensional tensor formats (due to the reduction in the comparison operations, as detailed in §4).

Figure 2 provides an example of the ALTO format for a $4 \times 8 \times 2$ sparse tensor with six nonzeros (denoted $x_{i,j,k}$). The multi-dimensional indices (i , j , and k) are color coded and the r^{th} bit of their binary representation is denoted $b_{i/j/k,r}$. Specifically, ALTO keeps the value of a nonzero element along with a linearized index, where each bit of this index is selected to partition the multi-dimensional space into two hyperplanes. For example, the ALTO encoding in Figure 2 uses a compact line of length 64 (i.e., a 6-bit linearized index) to represent the target tensor of size $4 \times 8 \times 2$. This index consists of three groups of bits with variable sizes (resolutions) to efficiently handle high-order data of arbitrary dimensionality. Within each bit group, ALTO arranges the modes in increasing order of their length (i.e., the shortest mode first), which is equivalent to partitioning the multi-dimensional space along the longest mode first. Such an encoding aims to generate a balanced linearization of the irregular Cartesian space, where *the amount of information about the spatial position of a nonzero element decreases with every consecutive bit*, starting from the most significant bit. Therefore, the line segments encode subspaces with mode intervals of equivalent lengths, e.g., the line segments $[0 - 31]$, $[0 - 15]$, and $[0 - 7]$ encode subspaces of $4 \times 4 \times 2$, $4 \times 2 \times 2$, and $2 \times 2 \times 2$ dimensions, respectively.

Due to this adaptive partitioning of the multi-dimensional data, ALTO encodes the resulting linearized index in the minimum number of bits, and it improves data locality across all modes of a given sparse tensor. Hence, a mode- N tensor, whose dimensions are $I_1 \times I_2 \times \dots \times I_N$, can be efficiently represented using a single ALTO format with indexing metadata of size:

$$S_{\text{ALTO}} = M \times \left(\sum_{n=1}^N \log_2 I_n \right) \text{ bits}, \quad (1)$$

where M is the number of nonzero elements.

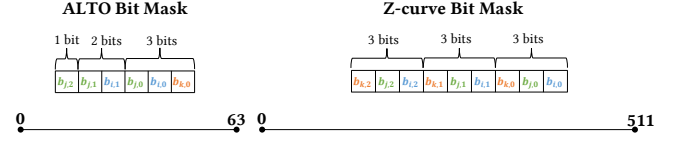


Figure 3: For the example in Figure 2, ALTO generates a non-fractal, yet more compact encoding compared to traditional space-filling curves, such as the Z-Morton order.

As a result, compared to the de facto COO format, ALTO reduces the storage requirement regardless of the tensor's characteristics. That is, the metadata compression ratio of the ALTO format relative to COO is always ≥ 1 . On a hardware architecture with a word-level memory addressing mode, this compression ratio is given by:

$$\frac{S_{\text{COO}}}{S_{\text{ALTO}}} = \frac{\sum_{n=1}^N \left\lceil \frac{\log_2 I_n}{W_b} \right\rceil}{\left\lceil \frac{\sum_{n=1}^N \log_2 I_n}{W_b} \right\rceil}, \quad (2)$$

where W_b is the word size in bits. For example, on an architecture with a byte-level addressing mode (i.e., $W_b = 8$ bits), the sparse tensor in Figure 2 requires three bytes to store the mode indices of a nonzero element in the *list-based* COO format and a single byte to store the linearized index of the same element in the ALTO format: the metadata compression ratio of ALTO, compared to the list-based formats, is three.

Most importantly, the ALTO format not only reduces the overall memory traffic of sparse tensor computations, but also decreases the number of memory transactions required to access the indexing metadata of a sparse tensor, as reading the linearized index requires fewer memory transactions compared to reading several multi-dimensional indices. In addition, this natural coalescing of the multi-dimensional indices into a single linearized index increases the memory transaction size to make more efficient use of the main memory bandwidth.

It is important to note that ALTO uses a non-fractal² encoding scheme, unlike the traditional space-filling curves (SFCs) [43]. In contrast, such SFCs (e.g., Z-Morton order [38]) are based on continuous self-similar (fractal) functions that can be extremely inefficient to encode the irregularly shaped multi-dimensional spaces that emerge in higher-order sparse tensor algorithms, as they require indexing metadata of size:

$$S_{\text{SFC}} = M \times N \times \max_{n=1}^N (\log_2 I_n) \text{ bits}. \quad (3)$$

Therefore, the application of SFCs in sparse tensor computations has been *limited to reordering the nonzero elements to improve their data locality* rather than compressing the indexing metadata [29]. Figure 3 shows the compact encoding generated by ALTO compared to the fractal encoding scheme based on the Z-Morton curve. In this example, the non-fractal encoding scheme used by ALTO reduces the length of the encoding line by a factor of eight, which not only decreases the overall size of the indexing metadata, but also reduces the linearization/de-linearization time required to map the multi-dimensional space to/from the compact encoding line.

²A fractal pattern is a hierarchically self-similar pattern that looks the same at increasingly smaller scales.

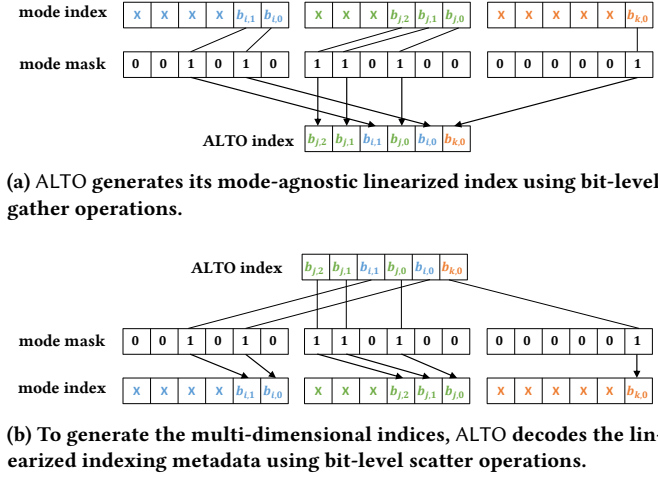


Figure 4: The ALTO-based bit encoding and decoding mechanisms for the example in Figure 2.

To allow fast indexing of the linearized tensors during sparse tensor operations, the ALTO encoding is implemented using a set of simple N bit masks, where N is the number of modes, on top of common data processing primitives. Figure 4 shows the linearization and de-linearization mechanisms, which are used during the ALTO format generation and the sparse tensor computations, respectively. The linearization is implemented as a bit-level gather, while the de-linearization is performed as a bit-level scatter. Thus, while the compressed representation of the proposed ALTO format comes at the cost of a de-linearization (decompression) overhead, such a computational overhead is minimal and can be effectively overlapped with the memory accesses of the memory-intensive sparse tensor operations, as shown in §4.

3.2 Workload Partitioning and Scheduling

The prior compressed sparse tensor formats, such as block- and CSF-based approaches, seek to reduce the size of the indexing metadata by clustering the nonzero elements into coarse-grained structures (e.g., tensor blocks, slices, and/or fibers) that divide the multi-dimensional space of a given tensor into non-overlapping regions. However, due to the irregular shapes and distributions of higher-order data, such coarse-grained approaches can suffer from severe workload imbalance, in terms of nonzero elements, which in turn leads to limited parallel performance and scalability.

Thus, the proposed ALTO representation works at the finest granularity level (i.e., a single nonzero element), which exposes the maximum fine-grained parallelism and allows scalable parallel execution. While a non-overlapping space partitioning of a tensor can be obtained from the ALTO encoding scheme, using a subset of the index bits, the workload balance of such a partitioning still depends on the sparsity patterns of the tensor.

To decouple the performance of sparse tensor computations from the distribution of nonzero elements, ALTO eliminates the workload imbalance and generates perfectly balanced partitions. Figure 5 depicts an example of ALTO's workload decomposition when applied to the sparse tensor in Figure 2. Moreover, ALTO divides the line segment containing the nonzero elements of the

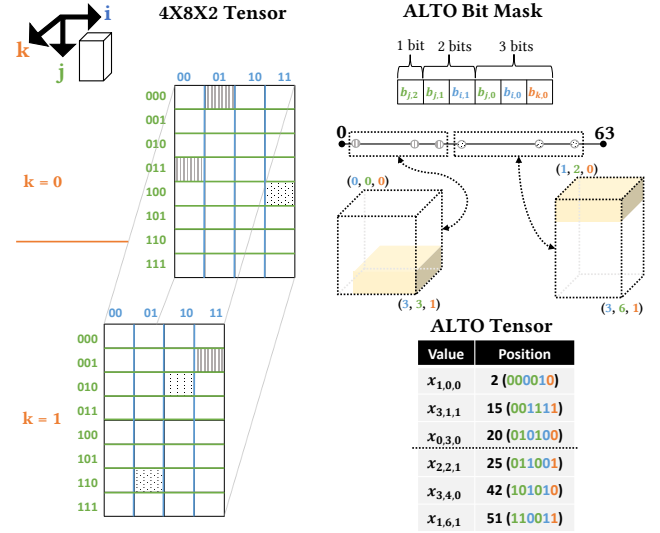


Figure 5: ALTO partitioning of the example in Figure 2, which generates balanced partitions in terms of workload (nonzero elements) for efficient parallel execution.

target tensor into smaller line segments, all of which have the same number of nonzeros, thus perfectly splitting the workload. Therefore, in Figure 5, ALTO partitions the linearized tensor into two line segments: [2 – 20] and [25 – 51]. Although the resulting line segments have different lengths (i.e., 18 and 26), they have the same number of nonzero elements.

Once the linearized sparse tensor is divided into multiple line segments, ALTO identifies the basis mode intervals (coordinate ranges) of the multi-dimensional subspaces that correspond to these segments. For example, the line segments [2 – 20] and [25 – 51] correspond to three-dimensional subspaces bounded by the mode intervals $\{[0 - 3], [0 - 3], [0 - 1]\}$ and $\{[1 - 3], [2 - 6], [0 - 1]\}$, respectively. While the derived multi-dimensional subspaces of the line segments may overlap, as highlighted in yellow in Figure 5, each nonzero element is assigned to exactly one line segment. That is, ALTO imposes a partitioning on a given linearized tensor that generates a disjoint set of *non-overlapping and balanced line segments*, yet it does not guarantee that such a partitioning will decompose the multi-dimensional space of the tensor into non-overlapping subspaces. In contrast, the prior sparse tensor formats decompose the multi-dimensional space into non-overlapping (yet highly imbalanced) regions, namely, tensor slices and fibers in CSF-based formats and multi-dimensional spatial blocks in block-based formats (e.g., HiCOO).

More formally, a set of L line segments partitions a linearized ALTO tensor X , which encodes a mode- N sparse tensor, such that $X = X_1 \cup X_2 \cdots \cup X_L$ and $X_i \cap X_j = \emptyset \forall i$ and j , where $i \neq j$. Each line segment X_i is an ordered set of nonzero elements that are bounded in an N -dimensional space by a set of N closed mode intervals $T_i = \{[T_{i,1}^s, T_{i,1}^e], [T_{i,2}^s, T_{i,2}^e], \dots, [T_{i,N}^s, T_{i,N}^e]\}$, where each mode interval $T_{i,j}$ is delineated by a start $T_{i,j}^s$ and an end $T_{i,j}^e$. The intersection of two sets of mode intervals represents the subspace overlap between their corresponding line segments (partitions).

3.3 Adaptive Tensor Operations

Since high-dimensional data analytics is becoming increasingly popular in rapidly evolving areas [22, 25, 42, 46], a fundamental goal of the proposed ALTO format is to deliver superior performance without compromising the productivity of end users to allow fast development of tensor algorithms and operations. Algorithm 1 illustrates the popular MTTKRP tensor operation using the ALTO format. First, unlike CSF-based formats, ALTO enables end users to perform tensor operations using *a unified code implementation that works on a single copy of the sparse tensor*, regardless of the different mode orientations of such operations. Second, by decoupling the representation of a sparse tensor from the distribution of its nonzero elements, ALTO does not require *manual* tuning to select the optimal *format parameters* for this tensor, in contrast to block-based storage approaches such as HiCOO. Instead, the ALTO format is automatically generated based on the shape and dimensions of the target sparse tensor (as explained in §3.1).

Algorithm 1 Mode-1 sequential MTTKRP-ALTO algorithm.

Input: A third-order ALTO sparse tensor $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$ with M nonzero elements, dense factor matrices $\mathbf{A} \in \mathbb{R}^{I \times R}$, $\mathbf{B} \in \mathbb{R}^{J \times R}$, and $\mathbf{C} \in \mathbb{R}^{K \times R}$

Output: Updated dense factor matrix $\tilde{\mathbf{A}} \in \mathbb{R}^{I \times R}$

```

1: for  $x = 1, \dots, M$  do
2:    $i = \text{EXTRACT}(\text{pos}(x), \text{MASK}(1))$   $\triangleright$  De-linearization.
3:    $j = \text{EXTRACT}(\text{pos}(x), \text{MASK}(2))$ 
4:    $k = \text{EXTRACT}(\text{pos}(x), \text{MASK}(3))$ 
5:   for  $r = 1, \dots, R$  do
6:      $\tilde{\mathbf{A}}(i, r) += \text{val}(x) \times \mathbf{B}(j, r) \times \mathbf{C}(k, r)$ 
7:   end for
8: end for
9: return  $\tilde{\mathbf{A}}$ 

```

Because processing the nonzero elements in parallel (line 1 in Algorithm 1) can result in write conflicts across threads (line 6 in Algorithm 1), we devise an effective parallel execution and synchronization algorithm that handles these conflicts by exploiting the inherent data reuse of target tensors. Algorithm 2 describes the proposed workload distribution and scheduling scheme using a representative parallel MTTKRP operation that works on a sparse tensor stored in the ALTO format. After ALTO imposes a partitioning on a given sparse tensor, as detailed in §3.2, each partition can be assigned to a different thread. To resolve the update/write conflicts that may happen during parallel sparse tensor computations, ALTO uses an *adaptive conflict resolution* approach that *automatically* selects the appropriate global synchronization technique (highlighted by the different gray backgrounds) across threads based on the reuse of the target fibers. This metric represents the average number of nonzero elements per fiber (the generalization of a matrix row/column) and it is estimated by simply dividing the total number of nonzero elements by the number of fibers along the target mode. When a sparse tensor operation suffers from limited fiber reuse, ALTO resolves the conflicting updates across its line segments (partitions) using direct atomic operations (line 8). Otherwise, it uses a limited amount of temporary (local) storage to keep the local

Algorithm 2 Adaptive parallel execution of mode-1 MTTKRP-ALTO kernel. ALTO automatically uses `local storage` or `atomics`, based on the reuse of output fibers, to resolve the update conflicts.

Input: A third-order ALTO sparse tensor $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$ with M nonzero elements, dense factor matrices $\mathbf{A} \in \mathbb{R}^{I \times R}$, $\mathbf{B} \in \mathbb{R}^{J \times R}$, and $\mathbf{C} \in \mathbb{R}^{K \times R}$

Output: Updated dense factor matrix $\tilde{\mathbf{A}} \in \mathbb{R}^{I \times R}$

```

1: for  $l = 1, \dots, L$  in parallel do  $\triangleright$  ALTO line segments.
2:   for  $\forall x \in \mathcal{X}_l$  do
3:      $i = \text{EXTRACT}(\text{pos}(x), \text{MASK}(1))$   $\triangleright$  De-linearization.
4:      $j = \text{EXTRACT}(\text{pos}(x), \text{MASK}(2))$ 
5:      $k = \text{EXTRACT}(\text{pos}(x), \text{MASK}(3))$ 
6:     for  $r = 1, \dots, R$  do
7:        $\text{Temp}_l(i - T_{l,1}^s, r) += \text{val}(x) \times \mathbf{B}(j, r) \times \mathbf{C}(k, r)$ 
8:        $\text{ATOMIC}(\tilde{\mathbf{A}}(i, r) += \text{val}(x) \times \mathbf{B}(j, r) \times \mathbf{C}(k, r))$ 
9:     end for
10:   end for
11: end for
12: for  $b = 1, \dots, I$  in parallel do  $\triangleright$  Pull-based accumulation.
13:   for  $\forall l$  where  $b \in [T_{l,1}^s, T_{l,1}^e]$  do
14:     for  $r = 1, \dots, R$  do
15:        $\tilde{\mathbf{A}}(b, r) += \text{Temp}_l(b - T_{l,1}^s, r)$ 
16:     end for
17:   end for
18: end for
19: return  $\tilde{\mathbf{A}}$ 

```

updates of different partitions (line 7) and then merges the conflicting global updates (lines 12–18) using an efficient pull-based parallel reduction, where the final results are computed by pulling the partial results from the ALTO partitions.

ALTO considers the fiber reuse large enough to use local staging memory for conflict resolution, if the average number of nonzero elements per fiber is *more than* the maximum cost of using this two-stage (buffered) accumulation process, which consists of initialization (omitted for brevity), local accumulation (line 7 in Algorithm 2), and global accumulation (lines 12–18). Specifically, the buffered accumulation cost is four memory operations (two read and two write operations) at most, i.e., in the worst (no reuse) case. As explained in §3.2, each line segment \mathcal{X}_l is bounded in an N -dimensional space by a set of N closed mode intervals T_l , which is computed during the partitioning of an ALTO tensor; thus, the size of the temporary storage accessed during the accumulation of \mathcal{X}_l 's updates along a mode n is directly determined by the mode interval $[T_{l,n}^s, T_{l,n}^e]$ (see lines 7 and 13).

Finally, ALTO allows automated generation and use of helper flags to further reduce the conflict resolution overhead in the sparse tensors that suffer from limited fiber reuse. That is, it exploits the unused (do-not-care) bits in the linearized index to encode if a nonzero element is a boundary or internal (exclusive) element along a mode with limited fiber reuse. Based on this information, ALTO determines whether to execute the global update (line 8 in Algorithm 2) as an atomic operation (for boundary elements) or a normal write (for internal elements).

4 EVALUATION

We evaluate ALTO against the state-of-the-art sparse tensor representations in terms of tensor storage, parallel performance and execution time, and format generation cost. We conduct a thorough study of key tensor decomposition operations (§2.2) and demonstrate the performance characteristics of ALTO not only compared to the prior formats, but also relative to an oracle that selects the *best* mode-agnostic and mode-specific format for each tensor dataset.

4.1 Implementation

We implemented ALTO as a C++ library and used OpenMP [13] for multi-threaded execution. The implementation utilizes automatic vectorization and loop unrolling optimizations [6], and it uses templates [56] for generalized support of tensors with arbitrary sizes. Specifically, we use a generic type to represent our mode-agnostic indexing metadata, which allows the same code to support linearized indices of different widths. This template-based approach avoids code duplication and reduces the time and effort required to port ALTO to other hardware platforms.

To improve the performance of sparse tensor kernels, the state-of-the-art tensor libraries specialize these kernels for different tensor orders (e.g., 3D and 4D tensors). In our library, a canonical tensor operation, such as MTTKRP, has an entry point that acts as a dispatcher, which invokes the generic implementation or more specialized versions of this implementation when available. This malleable approach leverages the compiler to transparently generate optimized code for common tensor orders and/or for typical decomposition ranks (called rank specialization). However, for a fair comparison with existing tensor libraries, we report the performance of ALTO *without rank specialization* and discuss the potential performance improvement of such an optimization.

We also incorporated ALTO into the popular SPLATT library [52] (i.e., replaced the CSF-based MTTKRP operation with the ALTO-based MTTKRP) to validate its usability in the CPD-ALS algorithm. Given the same initial values, our implementation calculates identical factor matrices as the original SPLATT implementation and shows the same convergence properties (i.e., same number of iterations to convergence and fit to the original tensor). Our implementation also shows similar fit compared to the Tensor Toolbox [4] CPD-ALS implementation.

4.2 Experimental Setup

4.2.1 Platform. All experiments were conducted on an Intel Xeon Platinum 8280 CPU with Cascade Lake-X (CLX) microarchitecture. It consists of two sockets, each with 28 physical cores running at a fixed frequency of 1.8 GHz for accurate measurements. The server has 384 GiB of memory and it runs CentOS 7.7 Linux distribution. The code is built using Intel C/C++ compiler (version 19.1.3) with the optimization flags `-O3 -xCORE-AVX512 -qopt-zmm-usage=high` to fully utilize vector units. Unless otherwise stated, the parallel experiments use all hardware threads (112) on the target platform. We report the performance numbers as an average over 100 iterations/runs, after a warmup iteration as per prior tensor libraries. For performance counter measurements and thread pinning, we use the LIKWID tool suite v5.1.0 [19].

Table 1: Characteristics of the target sparse tensors.

Tensor	Dimensions	#NNZs	Density	Fib. reuse
LBNL	$1.6K \times 4.2K \times 1.6K \times 4.2K \times 868.1K$	1.7M	4.2×10^{-14}	Limited
NIPS	$2.5K \times 2.9K \times 14K \times 17$	3.1M	1.8×10^{-06}	High
UBER	$183 \times 24 \times 1.1K \times 1.7K$	3.3M	3.8×10^{-04}	High
CHICAGO	$6.2K \times 24 \times 77 \times 32$	5.3M	1.5×10^{-02}	High
VAST	$165.4K \times 11.4K \times 2 \times 100 \times 89$	26M	7.8×10^{-07}	High
DARPA	$22.5K \times 22.5K \times 23.8M$	28.4M	2.4×10^{-09}	Limited
ENRON	$6K \times 5.7K \times 244.3K \times 1.2K$	54.2M	5.5×10^{-09}	High
NELL-2	$12.1K \times 9.2K \times 28.8K$	76.9M	2.4×10^{-05}	High
FB-M	$23.3M \times 23.3M \times 166$	99.6M	1.1×10^{-09}	Limited
FLICKR	$319.7K \times 28.2M \times 1.6M \times 731$	112.9M	1.1×10^{-14}	Limited
DELI	$532.9K \times 17.3M \times 2.5M \times 1.4K$	140.1M	4.3×10^{-15}	Medium
NELL-1	$2.9M \times 2.1M \times 25.5M$	143.6M	9.1×10^{-13}	Medium
AMAZON	$4.8M \times 1.8M \times 1.8M$	1.7B	1.1×10^{-10}	High
PATENTS	$46 \times 239.2K \times 239.2K$	3.6B	1.4×10^{-03}	High
REDDIT	$8.2M \times 177K \times 8.1M$	4.7B	4.0×10^{-10}	High

4.2.2 Datasets. For a comprehensive evaluation, the experiments consider a gamut of real-world tensor datasets with various characteristics. These tensors are often used in related works and they are publicly available in the FROSTT [48] and HaTen2 [23] repositories. Table 1 shows the detailed features of the target datasets, ordered by size, in terms of dimensions, number of nonzero elements (#NNZs), and density. To make the results clear and interpretable, the tensors are classified based on the average reuse of their fibers into high, medium, or limited reuse. We consider the fibers along a given mode to have high reuse, if they are reused more than eight times on average; when the fibers are reused between five to eight times, they have medium reuse; otherwise, the fibers suffer from limited reuse. Since the target tensor operations access fibers along all modes, a tensor with one or more modes of limited/medium reuse is considered to have an overall limited/medium fiber reuse.

4.2.3 Configurations. We evaluate the proposed ALTO format compared to the mode-agnostic COO and HiCOO formats [29] as well as the mode-specific CSF formats [50, 52]. Specifically, we use the latest code of the state-of-the-art sparse tensor libraries for CPUs, namely, ParTI³ and SPLATT.⁴ We report the best-achieved performance across the different configurations of the COO format; that is, with or without thread privatization (which keeps local copies of the output factor matrix). For the HiCOO format, its performance and storage are highly sensitive to the block and superblock (SB) sizes, which benefit from tuning. Since the current HiCOO implementation does not auto-tune the format parameters or consider the required tuning time in the format generation cost, we use a block size of 128 (2^7) and two superblock sizes of 2^{10} and 2^{14} according to prior work [55], and we report the performance of each format variant (“HiCOO-SB10” and “HiCOO-SB14”). We evaluate two variants of the mode-specific formats: CSF and CSF with tensor tiling

³Available at: <https://github.com/hpcgarage/ParTI>

⁴Available at: <https://github.com/ShadenSmith/splatt>

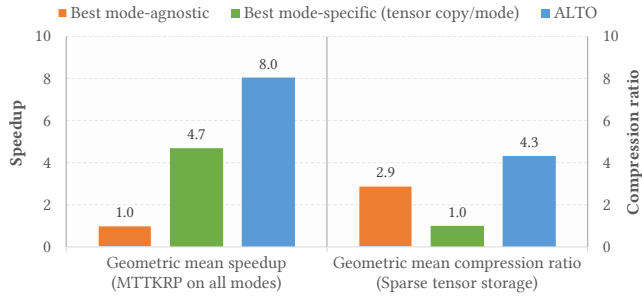


Figure 6: The performance metrics of the ALTO format (higher is better) in comparison with an oracle that selects the best mode-agnostic or mode-specific format variant for tensor datasets.

(“CSF-tile”), both of which use N representations (“SPLATT-ALL”) for an order- N sparse tensor to achieve the best performance.⁵ Similar to previous studies [10, 50, 52], the experiments use double-precision arithmetic and 64-bit (native word) integers. While the target datasets require a linearized index of size between 32 and 80 bits, we configured ALTO to select the size of its linearized index to be multiples of the native word size (i.e., 64 and 128 bits) for simplicity. We use a decomposition rank $R = 16$ for all experiments.

4.3 Summary of Performance Metrics

Figure 6 compares the performance characteristics of the proposed ALTO format to an oracle that selects the best format for target tensors. The oracle considers two distinct types of sparse tensor formats: 1) mode-agnostic or general formats (COO, HiCOO-SB10, and HiCOO-SB14), which use a single tensor representation, and 2) mode-specific formats (CSF and CSF-tile), which keep multiple tensor copies (one per mode) for best performance. The results show that ALTO outperforms the best mode-agnostic as well as mode-specific formats in terms of the speedup of tensor operations (MTTKRP on all modes) and the tensor storage. Specifically, ALTO achieves a geometric mean speedup of $8\times$ compared to the best general formats, while delivering a geometric mean compression ratio of $4.3\times$ relative to the best mode-specific formats. What follows is a detailed analysis and discussion of the performance results.

4.4 Performance Results and Analysis

To evaluate the parallel performance and scalability of the ALTO format, Figure 7 shows the speedup of the different parallel implementations of MTTKRP, which dominates the execution time of tensor decomposition methods. Unlike prior formats, the parallel performance of ALTO depends on the inherent data reuse of sparse tensors rather than the spatial distribution of their nonzero elements. As explained in Algorithm 2, the parallel MTTKRP-ALTO algorithm consists of two stages: 1) computing the output fibers of the target mode using the input fibers along all other modes, and 2) merging the conflicting updates of output fibers across threads. ALTO demonstrates linear scaling for the sparse tensors with high

reuse (on the left of the figure). Specifically, it achieves a geometric mean speedup of $47\times$ on 56 cores (i.e., more than 80% parallel efficiency) by exploiting the data locality of input fibers and by locally computing the partial updates of output fibers in higher levels of the memory system hierarchy. This way, the overhead of merging these partial updates is amortized over the large number of output fiber reuse. The analysis of performance counters, as detailed below, shows that most of the input/output fiber data is accessed in the cache, which strongly reduces the pressure on the main memory. For the AMAZON and REDDIT tensors, their extreme sparsity impacts the parallel scalability compared to other high-reuse cases, due to the reduction of useful work performed over the application’s memory footprint. In addition, when sparse tensors have limited/medium data reuse, as shown on the right of Figure 7, ALTO has a sub-linear scaling (a geometric mean speedup of $16\times$ on 56 cores) due to the large number of accesses to main memory. On our test platform, the STREAM Triad [36] bandwidth is 10 GB/s and 210 GB/s for a single core and a full node of 56 cores, respectively. Hence, ALTO obtains around 76% of the maximum realizable speedup ($21\times$) in such memory-bound cases.

On the contrary, the performance of prior sparse tensor formats varies widely due to their sensitivity to the irregular shapes and data distributions of higher-order tensors. As a result, the location-based formats (HiCOO-SB10/SB14, CSF, and CSF-tile) suffer from workload imbalance and ineffective compression depending on the distribution of the nonzero elements of each dataset. Additionally, based on the update conflicts across blocks/superblocks, the parallel schedule of block-based formats (HiCOO-SB10/SB14) can encounter serialization and/or substantial parallelization overhead. However, in a few cases, namely, NIPS, AMAZON, and NELL-1 tensors, the mode-specific CSF format shows slightly better performance ($1.2\times$ geometric-mean speedup) than the mode-agnostic ALTO format by keeping multiple sparse tensor representations along different mode orientations, which in turn significantly increases the storage of CSF (by $2.5\times$ – $4.5\times$ compared to ALTO, as shown in Figure 11).

4.4.1 Performance across Modes. Figure 8 demonstrates the run-time variations of the different formats while performing MTTKRP across tensor modes. The selected sparse tensors exhibit different characteristics in terms of shape, size, density, and data reuse. Compared to the other formats, ALTO has a relatively consistent performance regardless of the mode orientation of tensor operations. In the DARPA tensor, the execution time of ALTO along mode-3 is higher than mode-1 and mode-2 due to the limited fiber reuse of mode-3. While fibers along all modes are read during MTTKRP execution, only the output fibers along the target mode are updated. Since the memory read bandwidth is typically higher than the write bandwidth, the limited fiber reuse of mode-3 has more impact on mode-3 MTTKRP compared to other modes. In contrast, the other sparse formats suffer from significant performance variations across modes (note the use of logarithmic scale). While the execution time of the mode-specific CSF formats is expected to change based on the characteristics of the target mode, as they use a distinct tensor representation for each mode orientation, the results show that the parallel performance of block-based formats varies widely due to the different update conflicts and parallelism degrees of tensor blocks/superblocks across modes.

⁵In REDDIT dataset, SPLATT runs out of memory. While keeping fewer tensor copies is possible, it significantly degrades the performance on non-root modes due to using different recursion- and lock-based algorithmic variants.

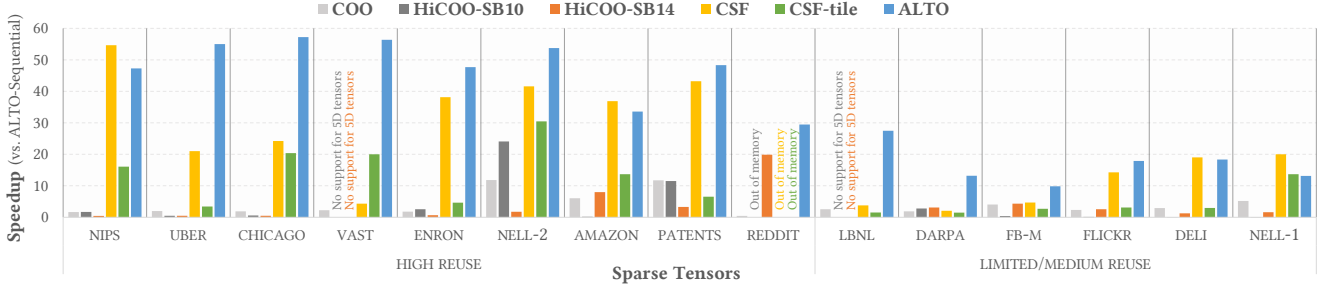


Figure 7: The overall parallel performance/speedup of MTTKRP (all modes) using the different sparse tensor formats. The speedup is reported compared to the optimized sequential MTTKRP-ALTO version to show the parallel scalability. The sparse tensors are categorized and then sorted in increasing order of their size (number of nonzero elements).

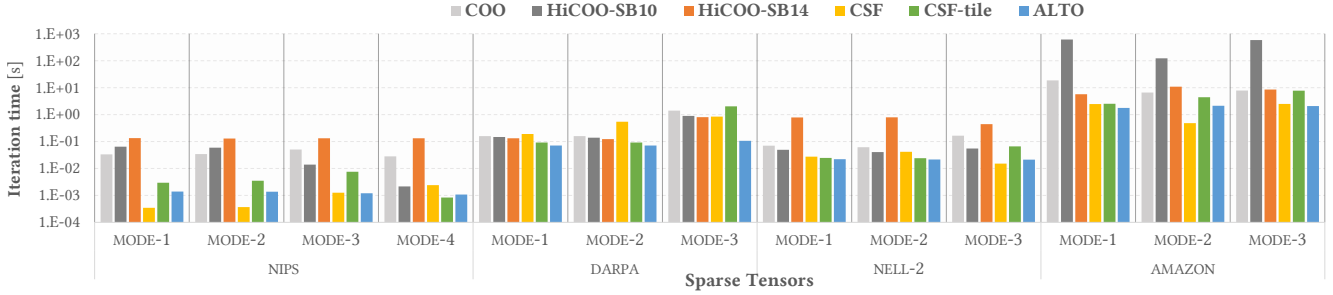


Figure 8: The execution time of parallel MTTKRP implementations across different modes for select tensors with various characteristics.

4.4.2 Roofline Analysis. To better understand the performance characteristics of MTTKRP using the ALTO format, we created a Roofline model [60] for the test platform and collected performance counters across a set of representative parallel runs. For the Roofline model, an upper performance limit Π is given by $\Pi = \min(\Pi_{\text{peak}}, B_{\text{peak}} \times OI)$, where Π_{peak} is the theoretical peak performance, B_{peak} is the peak memory bandwidth, and OI is the *operational intensity* (i.e., the ratio of executed floating-point operations per byte). In addition, we enhance our Roofline model to consider the bandwidth of the different cache levels. While the L2 cache, L3 cache, and main memory bandwidth in our model are phenomenological, i.e., measured using likwid-bench, L1 bandwidth measurements are error-prone. Therefore, we use the theoretical L1 load bandwidth of two cache lines per cycle per core for this particular roofline. The peak performance, Π_{peak} , is calculated based on the ability of the cores to execute two fused multiply-add (FMA) instructions on eight-element double precision vector registers (due to the availability of AVX-512) per cycle.

Figure 9 shows the performance of the parallel MTTKRP-ALTO computations (lines 1-11 in Algorithm 2) for several representative cases. While the memory-intensive MTTKRP operation suffers from low operational intensity, ALTO can still exceed the peak main memory bandwidth by exploiting the inherent data reuse and by efficiently resolving the update conflicts. The NIPS tensor ① is relatively small and it has high reuse of output fibers, which can be held locally in L1/L2 caches. The performance counters show a superior cache hit rate while computing output fibers, which explains the high effective bandwidth and floating-point throughput. Mode-3 of NELL-2 ② has a smaller amount of fiber reuse and it accesses a larger number of local output fibers compared to NIPS ①. Nonetheless,

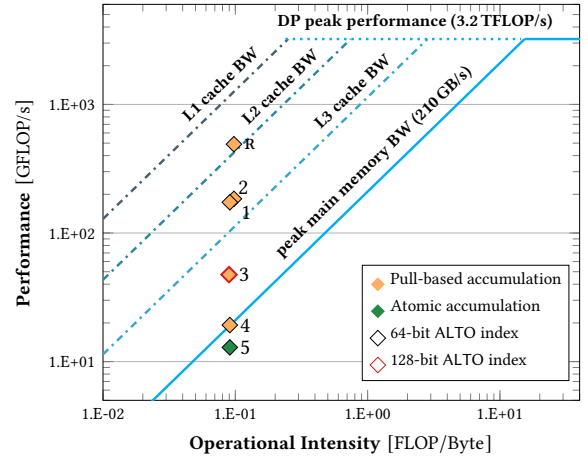


Figure 9: The parallel performance of ALTO across MTTKRP runs: NIPS mode-4 without ① and with ③ rank specialization, NELL-2 mode-3 ②, AMAZON mode-1 ③, DARPA mode-1 ④ and mode-3 ⑤.

ALTO efficiently utilizes all threads to do useful work and amortizes the parallelization overheads due to the larger number of nonzero elements, and we observe a slightly higher performance than ①. While the AMAZON tensor ③ has high fiber reuse, its extreme sparsity and large size increase the memory pressure. In addition, although the tensor can be encoded using a 72-bit linearized index, our current ALTO implementation uses a 128-bit linearized index (twice the native word size) for simplicity. Together these factors increase the data volume and the de-linearization overhead, which



Figure 10: The speedup of ALTO using rank specialization relative to the default ALTO parallel implementation. The sparse tensors are categorized and then sorted in an increasing order of their size.

in turn reduces the effective floating-point throughput compared to other tensors with high data reuse ① and ②.

In contrast, the DARPA tensor has limited fiber reuse (along mode-3) which restricts the performance compared to the previous cases. During the computation of MTTKRP on mode-3 ⑤, the limited reuse of output fibers does not justify the use of local/temporary memory to resolve the conflicting updates. As a result, ALTO automatically chooses to use atomic updates for conflict resolution. While DARPA mode-1 ④ has higher reuse of output fibers, comparable to ②, reading the input fibers along mode-3 dominates the execution time and results in more data traffic across the memory system levels because of the limited temporal locality; thus, we can observe that its performance is bounded by the memory bandwidth.

4.5 Impact of Rank Specialization

The generic (template-based) implementation of ALTO (which is discussed in §4.1) not only allows effortless specialization of sparse tensor kernels for common tensor orders (e.g., 3D and 4D tensors), but also makes it possible to specialize these kernels for typical ranks — an optimization that we call rank specialization. For a fair comparison with the prior sparse tensor implementations, all other performance results in this paper using ALTO *do not leverage rank specialization*. Here, we show the potential of this feature to further improve the performance by allowing the compiler to gain more insight into the length of rank-wise update operations. This way, the compiler can generate more optimized code, which leads to a better overall performance, as shown for NIPS mode-4 ⑧. The analysis of the generated assembly files with OSACA [28] shows that rank specialization results in better control structures with fewer branches compared to the standard implementation. Additionally, the compiler can reduce the pressure on the load units and, thus, decreases the execution time by more than 2× in NIPS ⑧.

Figure 10 shows a comprehensive comparison between the rank specialized ALTO version and the default parallel version used in prior experiments. For smaller tensors with high reuse, the compiler manages to optimize the computations and to improve the performance by more than 2×. In larger tensors as well as tensors with limited reuse, we can still observe a performance benefit of more than 10%, leading to an overall geometric mean speedup of 1.4× across the different sparse tensors.

4.6 Memory Storage

Figure 11 details the relative storage of the sparse tensors using the proposed ALTO format as well as the block-based (HiCOO-SB10/SB14) and tree-based (CSF and CSF-tile) formats. Compared to the de facto COO format, ALTO always requires less storage space due to its efficient linearization of the multi-dimensional space, as explained in §3.1. The mode-specific (CSF and CSF-tile) formats consume more storage space than COO because they keep multiple sparse representations for the different mode orientations. Overall, imposing a tiling over the tensors (CSF-tile) ends up increasing the storage requirements of CSF. Conversely, the memory consumption of the block-based formats is highly variable as it depends on the block/superblock sizes and the distribution of the nonzero elements in the multi-dimensional space. Thus, HiCOO can reduce the storage compared to COO when the number of nonzero elements per block is relatively high. However, as the sparsity of the tensors increases, the block-based formats consume more space, such as in the DELI, NELL-1, AMAZON, and REDDIT tensors.

4.7 Format Generation Cost

Figure 12 compares the cost of constructing the different sparse tensor formats from a sparse tensor in the COO format. Since ALTO works on a linearized rather than a multi-dimensional representation of tensors, the cost of sorting the nonzero elements in its linearized storage (which is the most expensive step in the format generation) is substantially reduced because of the decrease in the number of comparison operations. In addition to sorting the nonzero elements, the block-based HiCOO formats require costly clustering of these elements based on their multi-dimensional coordinates as well as scheduling of the resulting blocks/superblocks to avoid conflicts. As a result, ALTO has more than an order-of-magnitude speedup relative to HiCOO in terms of the format conversion time. Although the tree-based CSF formats are generated from *presorted* COO tensors, ALTO still requires less time for format construction on average. In large-scale tensors, such as AMAZON and PATENTS, ALTO has comparable conversion cost to the tree-based formats as the tensor sorting time increases with the number of nonzero elements. Currently, the format generation of ALTO leverages the C++ Standard Template Library (STL) and a further reduction of this construction cost is possible.

5 RELATED WORK AND DISCUSSION

Due to the popularity of high-dimensional data analytics, a rich set of work has been recently developed to efficiently store sparse tensors and to optimize various tensor computations. Here, we discuss the related sparse tensor formats and operations as well as the conventional recursive data layouts.

5.1 Sparse Tensor Formats

Our study was motivated by the linearized coordinate (LCO) format [21], which also flattens sparse tensors but along a given orientation of tensor modes. Hence, LCO requires either multiple tensor copies or on-the-fly permutation of tensors for efficient computation. In addition, the authors limit their focus to sequential algorithms, and it is not clear how LCO can be used to efficiently parallelize sparse tensor computations.

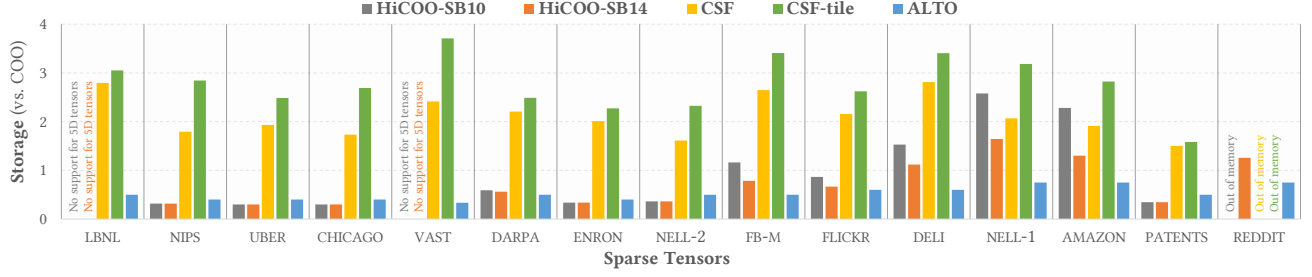


Figure 11: The sparse tensor storage using the different formats compared to COO. The tensors are sorted in an increasing order of their size.

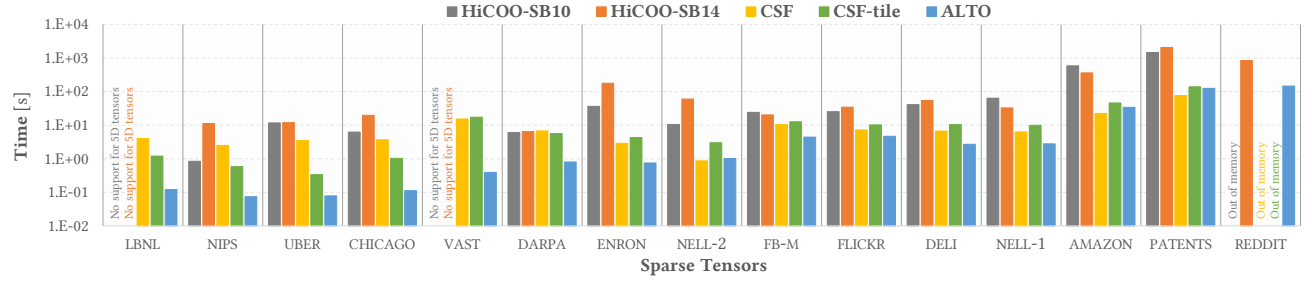


Figure 12: The format construction cost in seconds. The sparse tensors are sorted in an increasing order of their size.

Recent extensions [31, 45] to the list-based COO format reduce the synchronization overhead across threads using *mode-specific* scheduling/permutation arrays. However, their output-oriented traversal of sparse tensors adversely affects the input data locality and/or result in random access of the nonzero elements [45]. Moreover, storing this fine-grained scheduling information for all tensor modes can more than double the memory consumption, compared to the COO format [45].

The block-based formats, such as HiCOO [29], are highly sensitive to the characteristics of sparse tensors as well as the block size. When the blocking ratio (i.e., the number of blocks to the number of nonzero elements) is high, HiCOO can use more storage space than the simple COO format [29]. Due to the nonuniform (skewed) data distributions in the multi-dimensional space, the number of nonzero elements per block varies widely across HiCOO blocks, even after expensive mode-specific tensor permutations which in practice further increase workload imbalance [30]. Moreover, such block-based formats, which use small data types for element indices, can end up under-utilizing the compute units and memory bandwidth because 1) mainstream architectures are word-oriented, which leads to limited efficiency of sub-word (narrow-width) operations [2, 37], and 2) applications need to generate large memory transactions to efficiently utilize the bandwidth of commodity DRAM chips [18].

The mode-specific CSF format [50, 52] clusters the nonzero tensor elements into coarse-grained tensor slices and fibers, which limits its scalability on massively parallel architectures. To improve the performance on GPUs, recent CSF-based formats [40, 41] expose more balanced and fine-grained parallelism but at the expense of substantial synchronization overheads and expensive preprocessing and format generation costs. The sparse tensor format selection (SpTFS) framework [55] leverages deep learning models [61, 64] to predict the best of COO, HiCOO, and CSF formats to perform the MTTKRP operation on a given sparse tensor.

5.2 Sparse Tensor Operations

In addition to the widely used sparse tensor decomposition, which we extensively discussed in previous sections, other important tensor algebra operations can benefit from the proposed ALTO format. Examples of these operations include sparse tensor transposition and contraction as well as streaming tensor analysis.

Sparse tensor transpositions are prevalent in high-dimensional data processing and analysis algorithms [39]. The state-of-the-art approaches [39, 58] reduce these important operations to sorting the nonzero elements based on their multi-dimensional coordinates. ALTO accelerates the tensor sorting process due to its linearized representation and recursive ordering of nonzero elements, which makes the sparse tensors stored in the ALTO format amenable to partial radix-based sorting.

Sparse tensor contractions emerge in many critical scientific domains [54], e.g., computational physics and quantum chemistry. These operations need random access into tensors, which can be supported by hash-based COO representations, such as Sparta [32]. ALTO can further improve performance by using the target subset of encoding bits that represent the contract modes to quickly build and access a hash-based representation of linearized tensors.

To analyze infinite data sources, streaming tensor decomposition assembles batches of incoming data into a sequence of sparse sub-tensors and incrementally computes factor matrices [49, 53]. As shown in §4.7, ALTO uses a fraction of the format generation time required by prior compressed formats, which makes it more suitable for accelerating such streaming algorithms.

5.3 Recursive Data Layouts

Previous studies explored cache-oblivious [17, 20] data layouts for optimizing memory-intensive operations, especially in the context of linear algebra. These data layouts exploit the underlying memory hierarchy without knowing its specific structure or configuration.

Recursive algorithms and storage schemes [9, 14, 16, 44] are popular in Basic Linear Algebra Subprogram (BLAS) kernels, due to the regularity of such dense computations.

Yzelman et al. [62, 63] proposed recursive data layouts for sparse matrices, based on hypergraph partitioning, to improve the performance of sparse-matrix, dense-vector (SpMV) multiplication. However, these methods require permuting the rows and columns of sparse matrices to generate more structured sparsity patterns. Martone et al. [34, 35] introduced the Recursive Sparse Block (RSB) storage scheme for efficient execution of symmetric SpMV operations. On multi-core CPUs, this hierarchical representation demonstrates comparable performance to tiled (block-based) formats, such as Compressed Sparse Blocks (CSB) [8]. The Distributed Block Compressed Sparse Row (DBCSP) library [7] adopts a hybrid approach, in which a sparse matrix is recursively divided until it has a pre-defined number of blocks. Moreover, the DBCSP library provides a tensor interface [47] to execute tensor contractions using the optimized sparse matrix multiplication kernels.

6 CONCLUSION

Motivated by the vulnerability of existing compressed formats to the irregular characteristics of higher-order sparse data, this work proposed the ALTO format to efficiently encode sparse tensors of arbitrary dimensionality and spatial distributions using a single mode-agnostic representation. An implementation of the ALTO-based tensor decomposition operations (all-modes MTTRP) proved to outperform an oracle that selects the best state-of-the-art format, in terms of parallel performance and tensor storage. Specifically, ALTO delivers $8\times$ geometric-mean speedup and $4.3\times$ geometric-mean compression ratio over the best general and mode-specific formats, respectively, due to its superior workload balance, adaptive synchronization, and compact encoding. Furthermore, the experiments showed the potential of our template-based implementation to bring extra performance improvement ($1.4\times$ geometric-mean speedup) by specializing the ALTO-based tensor kernels for target application features.

Our future work will investigate additional massively parallel and distributed-memory platforms that can benefit from the superior performance characteristics of ALTO. We also plan to explore the use of the proposed format to accelerate other common sparse tensor algorithms, besides tensor decomposition.

ACKNOWLEDGMENTS

The authors would like to thank Dr. Tamara G. Kolda for setting us on the direction of tensor linearization and for providing valuable feedback.

REFERENCES

- [1] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) (OSDI'16). USENIX Association, USA, 265–283. <https://dl.acm.org/doi/10.5555/3026877.3026899>
- [2] Andreas Abel and Jan Reineke. 2019. uops.info: Characterizing Latency, Throughput, and Port Usage of Instructions on Intel Microarchitectures. In *ASPLOS (Providence, RI, USA) (ASPLOS '19)*. ACM, New York, NY, USA, 673–686. <https://doi.org/10.1145/3297858.3304062>
- [3] Animashree Anandkumar, Rong Ge, Daniel Hsu, Sham M. Kakade, and Matus Telgarsky. 2014. Tensor Decompositions for Learning Latent Variable Models. *J. Mach. Learn. Res.* 15, 1 (Jan. 2014), 2773–2832. <https://dl.acm.org/doi/10.5555/2627435.2697055>
- [4] Brett W. Bader and Tamara G. Kolda. 2007. Efficient MATLAB computations with sparse and factored tensors. *SIAM JOURNAL ON SCIENTIFIC COMPUTING* 30, 1 (2007), 205–231. <https://doi.org/10.1137/060676489>
- [5] M. Baskaran, B. Meister, N. Vasilache, and R. Lethin. 2012. Efficient and Scalable Computations with Sparse Tensors. In *2012 IEEE Conference on High Performance Extreme Computing*. 1–6. <https://doi.org/10.1109/HPEC.2012.6408676>
- [6] Aart J. C. Bik, Milind Girkar, Paul M. Grey, and Xinmin Tian. 2002. Automatic Intra-Register Vectorization for the Intel Architecture. *Int. J. Parallel Program.* 30, 2 (April 2002), 65–98. <https://dl.acm.org/doi/10.5555/586554.586555>
- [7] Urban Borštnik, Joost VandeVondele, Valéry Weber, and Jürg Hutter. 2014. Sparse matrix multiplication: The distributed block-compressed sparse row library. *Parallel Comput.* 40, 5–6 (2014), 47–58. <https://doi.org/10.1016/j.parco.2014.03.012>
- [8] Aydin Buluç, Jeremy T. Fineman, Matteo Frigo, John R. Gilbert, and Charles E. Leiserson. 2009. Parallel Sparse Matrix-Vector and Matrix-Transpose-Vector Multiplication Using Compressed Sparse Blocks (SPAA '09). Association for Computing Machinery, New York, NY, USA, 233–244. <https://doi.org/10.1145/1583991.1584053>
- [9] S. Chatterjee, A. R. Lebeck, P. K. Patnala, and M. Thottethodi. 2002. Recursive Array Layouts and Fast Matrix Multiplication. *IEEE Transactions on Parallel and Distributed Systems* 13, 11 (2002), 1105–1123. <https://doi.org/10.1109/TPDS.2002.1058095>
- [10] 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. <https://doi.org/10.1109/IPDPS.2018.00066>
- [11] Joon Hee Choi and S. V. N. Vishwanathan. 2014. DFacTo: Distributed Factorization of Tensors. In *Proceedings of the 27th International Conference on Neural Information Processing Systems* (Montreal, Canada) (NIPS'14). MIT Press, Cambridge, MA, USA, 1296–1304. <https://dl.acm.org/doi/10.5555/2968826.2968971>
- [12] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. 2018. Format Abstraction for Sparse Tensor Algebra Compilers. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 123 (Oct. 2018), 30 pages. <https://doi.org/10.1145/3276493>
- [13] L. Dagum and R. Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering* 5, 1 (1998), 46–55. <https://doi.org/10.1109/99.660313>
- [14] Erik Elmroth, Fred Gustavson, Isak Jonsson, and Bo Kågström. 2004. Recursive Blocked Algorithms and Hybrid Data Structures for Dense Matrix Library Software. *SIAM Rev.* 46, 1 (2004), 3–45. <https://doi.org/10.1137/S0036144503428693>
- [15] Hadi Fanaee-T and João Gama. 2016. Tensor-based anomaly detection: An interdisciplinary survey. *Knowledge-Based Systems* 98 (2016), 130–147. <https://doi.org/10.1016/j.kmosys.2016.01.027>
- [16] Jeremy D. Frens and David S. Wise. 1997. Auto-Blocking Matrix-Multiplication or Tracking BLAS3 Performance from Source Code. *SIGPLAN Not.* 32, 7 (June 1997), 206–216. <https://doi.org/10.1145/263767.263789>
- [17] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. 1999. Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science (Cat. No. 99CB37039)*. 285–297. <https://doi.org/10.1109/SFSCS.1999.814600>
- [18] Saugata Ghose, Tianshi Li, Nastaran Hajinazar, Damla Senol Cali, and Onur Mutlu. 2019. Demystifying Complex Workload-DRAM Interactions: An Experimental Study. 3, 3, Article 60 (Dec. 2019), 50 pages. <https://doi.org/10.1145/3366708>
- [19] Thomas Gruber, Jan Eitzinger, Georg Hager, and Gerhard Wellein. 2020. RRZE-HPC/likwid: likwid-5.1.0. <https://doi.org/10.5281/zenodo.4282696>
- [20] F. G. Gustavson. 1997. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. *IBM Journal of Research and Development* 41, 6 (1997), 737–755. <https://doi.org/10.1147/rd.416.0737>
- [21] A. P. Harrison and D. Joseph. 2018. High Performance Rearrangement and Multiplication Routines for Sparse Tensor Arithmetic. *SIAM Journal on Scientific Computing* 40, 2 (2018), C258–C281. <https://doi.org/10.1137/17M1115873>
- [22] Joyce C. Ho, Joydeep Ghosh, and Jimeng 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* (New York, New York, USA) (KDD '14). Association for Computing Machinery, New York, NY, USA, 115–124. <https://doi.org/10.1145/2623330.2623658>
- [23] I. Jeon, E. E. Papalexakis, U. Kang, and C. Faloutsos. 2015. HaTen2: Billion-scale tensor decompositions. In *2015 IEEE 31st International Conference on Data Engineering*. 1047–1058. <https://doi.org/10.1109/ICDE.2015.7113355>
- [24] U. Kang, Evangelos Papalexakis, Abhay Harpale, and Christos 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*. Association for Computing Machinery, New York, NY, USA, 316–324. <https://doi.org/10.1145/2339530.2339583>
- [25] Teruyoshi Kobayashi, Anna Sapienza, and Emilio Ferrara. 2018. Extracting the multi-timescale activity patterns of online financial markets. *Scientific Reports* 8,

- 1 (2018), 1–11. <https://doi.org/10.1038/s41598-018-29537-w>
- [26] Tamara G. Kolda and Brett W. Bader. 2009. Tensor Decompositions and Applications. *SIAM Rev.* 51, 3 (September 2009), 455–500. <https://doi.org/10.1137/07070111X>
- [27] T. G. Kolda and J. Sun. 2008. Scalable Tensor Decompositions for Multi-aspect Data Mining. In *2008 Eighth IEEE International Conference on Data Mining*. 363–372. <https://doi.org/10.1109/ICDM.2008.89>
- [28] J. Laukemann, J. Hammer, G. Hager, and G. Wellein. 2019. Automatic Throughput and Critical Path Analysis of x86 and ARM Assembly Kernels. In *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. 1–6. <https://doi.org/10.1109/PMBS49563.2019.00006>
- [29] J. Li, J. Sun, and R. Vuduc. 2018. HiCOO: Hierarchical Storage of Sparse Tensors. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. 238–252. <https://doi.org/10.1109/SC.2018.00022>
- [30] Jiajia Li, Bora Ucar, Ümit V. Çatalyürek, Jimeng Sun, Kevin Barker, and Richard Vuduc. 2019. Efficient and Effective Sparse Tensor Reordering. In *Proceedings of the ACM International Conference on Supercomputing (Phoenix, Arizona) (ICS '19)*. Association for Computing Machinery, New York, NY, USA, 227–237. <https://doi.org/10.1145/3330345.3330366>
- [31] 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>
- [32] Jiawen Liu, Jie Ren, Roberto Gioiosa, Dong Li, and Jiajia Li. 2021. Sparta: High-Performance, Element-Wise Sparse Tensor Contraction on Heterogeneous Memory. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (Virtual Event, Republic of Korea) (PPoPP '21)*. Association for Computing Machinery, New York, NY, USA, 318–333. <https://doi.org/10.1145/3437801.3441581>
- [33] Shangzhi Liu and Götz Trenkler. 2008. Hadamard, Khatri-Rao, Kronecker, and Other Matrix Products. *International Journal of Information and Systems Sciences* 4, 1 (2008), 160–177. <https://doi.org/10.1155/2016/8301709>
- [34] Michele Martone. 2014. Efficient multithreaded untransposed, transposed or symmetric sparse matrix–vector multiplication with the Recursive Sparse Blocks format. *Parallel Comput.* 40, 7 (2014), 251–270. <https://doi.org/10.1016/j.parco.2014.03.008> 7th Workshop on Parallel Matrix Algorithms and Applications.
- [35] M. Martone, S. Filippone, M. Paprzycki, and S. Tucci. 2010. On BLAS Operations with Recursively Stored Sparse Matrices. In *2010 12th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*. 49–56. <https://doi.org/10.1109/SYNASC.2010.72>
- [36] John D. McCalpin. 1995. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* (Dec. 1995), 19–25. http://tab.computer.org/tcca/NEWS/DEC95/dec95_mccalpin.ps
- [37] Sparsh Mittal. 2017. A Survey of Techniques for Designing and Managing CPU Register File. *Concurrency and Computation: Practice and Experience* 29, 4 (2017), e3906. <https://doi.org/10.1002/cpe.3906>
- [38] Guy M. Morton. 1966. *A computer Oriented Geodetic Data Base; and a New Technique in File Sequencing*. Technical Report. IBM Ltd., 150 Laurier Ave., Ottawa, Ontario, Canada. <https://dominoweb.draco.res.ibm.com/reports/Morton1966.pdf>
- [39] Suzanne Mueller, Peter Ahrens, Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. 2020. Sparse Tensor Transpositions. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures (Virtual Event, USA) (SPAA '20)*. Association for Computing Machinery, New York, NY, USA, 559–561. <https://doi.org/10.1145/3350755.3400245>
- [40] Israt Nisa, Jiajia Li, Aravind Sukumaran-Rajam, Prasant Singh Rawat, Sriram Krishnamoorthy, and P. Sadayappan. 2019. An Efficient Mixed-Mode Representation of Sparse Tensors. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (Denver, Colorado) (SC '19)*. Association for Computing Machinery, New York, NY, USA, Article 49, 25 pages. <https://doi.org/10.1145/3295500.3356216>
- [41] I. Nisa, J. Li, A. Sukumaran-Rajam, R. Vuduc, and P. Sadayappan. 2019. Load-Balanced Sparse MTTKRP on GPUs. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 123–133. <https://doi.org/10.1109/IPDPS.2019.00023>
- [42] Evangelos E. Papalexakis, Christos Faloutsos, and Nicholas D. Sidiropoulos. 2016. Tensors for Data Mining and Data Fusion: Models, Applications, and Scalable Algorithms. *ACM Trans. Intell. Syst. Technol.* 8, 2, Article 16 (Oct. 2016), 44 pages. <https://doi.org/10.1145/2915921>
- [43] Giuseppe Peano. 1890. Sur une courbe, qui remplit toute une aire plane. *Math. Ann.* 36, 1 (1890), 157–160. <https://doi.org/10.1007/BF01199438>
- [44] Elmar Peise and Paolo Bientinesi. 2017. Algorithm 979: Recursive Algorithms for Dense Linear Algebra—The ReLAPACK Collection. *ACM Trans. Math. Softw.* 44, 2, Article 16 (Sept. 2017), 19 pages. <https://doi.org/10.1145/3061664>
- [45] Eric T. Phipps and Tamara G. Kolda. 2019. Software for Sparse Tensor Decomposition on Emerging Computing Architectures. *SIAM Journal on Scientific Computing* 41, 3 (2019), C269–C290. <https://doi.org/10.1137/18M1210691>
- [46] 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 (2017), 3551–3582. <https://doi.org/10.1109/TSP.2017.2690524>
- [47] Ilia Sivkov, Patrick Seewald, Alfio Lazzaro, and Jürg Hutter. 2019. DBCSR: A Blocked Sparse Tensor Algebra Library. *CoRR* abs/1910.13555 (2019). arXiv:1910.13555 <http://arxiv.org/abs/1910.13555>
- [48] Shaden Smith, Jee W. Choi, Jiajia Li, Richard Vuduc, Jongsoo Park, Xing Liu, and George Karypis. 2017. FROSTT: The Formidable Repository of Open Sparse Tensors and Tools. <http://frostt.io/>
- [49] Shaden Smith, Kejun Huang, Nicholas D. Sidiropoulos, and George Karypis. 2018. Streaming Tensor Factorization for Infinite Data Sources. In *Proceedings of the 2018 SIAM International Conference on Data Mining (SDM)*. SIAM, 81–89. <https://doi.org/10.1137/1.9781611975321.10>
- [50] Shaden Smith and George Karypis. 2015. Tensor-Matrix Products with a Compressed Sparse Tensor. In *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms (Austin, Texas) (IA³ '15)*. Association for Computing Machinery, New York, NY, USA, Article 5, 7 pages. <https://doi.org/10.1145/2833179.2833183>
- [51] Shaden Smith and George Karypis. 2017. Accelerating the Tucker Decomposition with Compressed Sparse Tensors. In *Euro-Par 2017: Parallel Processing*, Francisco F. Rivera, Tomás F. Pena, and José C. Cabaleiro (Eds.). Springer International Publishing, Cham, 653–668. https://doi.org/10.1007/978-3-319-64203-1_47
- [52] S. Smith, N. Ravindran, N. D. Sidiropoulos, and G. Karypis. 2015. SPLATT: Efficient and Parallel Sparse Tensor-Matrix Multiplication. (2015), 61–70. <https://doi.org/10.1109/IPDPS.2015.27>
- [53] Yongseok Soh, Patrick Flick, Xing Liu, Shaden Smith, Fabio Checconi, Fabrizio Petrini, and Jee Choi. 2021. High Performance Streaming Tensor Decomposition. In *35th IEEE International Parallel and Distributed Processing Symposium (IPDPS'21)*.
- [54] Edgar Solomonik, Devin Matthews, Jeff R. Hammond, John F. Stanton, and James Demmel. 2014. A Massively Parallel Tensor Contraction Framework for Coupled-Cluster Computations. *J. Parallel and Distrib. Comput.* 74, 12 (2014), 3176–3190. <https://doi.org/10.1016/j.jpdc.2014.06.002> Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.
- [55] Qingxiao Sun, Yi Liu, Ming Dun, Hailong Yang, Zhongzhi Luan, Lin Gan, Guangwen Yang, and Depei Qian. 2020. SpTFS: Sparse Tensor Format Selection for MTTKRP via Deep Learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (Atlanta, Georgia) (SC '20)*. IEEE Press, Article 18, 14 pages. <https://dl.acm.org/doi/abs/10.5555/3433701.3433724>
- [56] David Vandevoorde and Nicolai M Josuttis. 2002. *C++ Templates: The Complete Guide, Portable Documents*. Addison-Wesley Professional. https://cds.cern.ch/record/781664/files/0201734842_TOC.pdf
- [57] N. Vervliet, O. Debals, and L. De Lathauwer. 2016. Tensorlab 3.0 — Numerical optimization strategies for large-scale constrained and coupled matrix/tensor factorization. In *2016 50th Asilomar Conference on Signals, Systems and Computers*. 1733–1738. <https://doi.org/10.1109/ACSSC.2016.7869679>
- [58] Hao Wang, Weifeng Liu, Kaixi Hou, and Wu-chun Feng. 2016. Parallel Transposition of Sparse Data Structures. In *Proceedings of the 2016 International Conference on Supercomputing (Istanbul, Turkey) (ICS '16)*. Association for Computing Machinery, New York, NY, USA, Article 33, 13 pages. <https://doi.org/10.1145/2925426.2926291>
- [59] Yichen Wang, Robert Chen, Joydeep Ghosh, Joshua C. Denny, Abel Kho, You Chen, Bradley A. Malin, and Jimeng Sun. 2015. Rubik: Knowledge Guided Tensor Factorization and Completion for Health Data Analytics (KDD '15). Association for Computing Machinery, New York, NY, USA, 1265–1274. <https://doi.org/10.1145/2783258.2783395>
- [60] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM* 52, 4 (2009), 65–76. <https://doi.org/10.1145/1498765.1498785>
- [61] Zhen Xie, Guangming Tan, Weifeng Liu, and Ninghui Sun. 2019. IA-SpGEMM: An Input-Aware Auto-Tuning Framework for Parallel Sparse Matrix-Matrix Multiplication. In *Proceedings of the ACM International Conference on Supercomputing (Phoenix, Arizona) (ICS '19)*. Association for Computing Machinery, New York, NY, USA, 94–105. <https://doi.org/10.1145/3330345.3330354>
- [62] A.N. Yzelman and Rob H. Bisseling. 2011. Two-Dimensional Cache-Oblivious Sparse Matrix–Vector Multiplication. *Parallel Comput.* 37, 12 (2011), 806–819. <https://doi.org/10.1016/j.parco.2011.08.004> 6th International Workshop on Parallel Matrix Algorithms and Applications (PMAA'10).
- [63] A. N. Yzelman and Rob H. Bisseling. 2009. Cache-Oblivious Sparse Matrix–Vector Multiplication by Using Sparse Matrix Partitioning Methods. *SIAM Journal on Scientific Computing* 31, 4 (2009), 3128–3154. <https://doi.org/10.1137/080733243>
- [64] Yue Zhao, Jiajia Li, Chunhua Liao, and Xipeng Shen. 2018. 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 (Vienna, Austria) (PPoPP '18)*. Association for Computing Machinery, New York, NY, USA, 94–108. <https://doi.org/10.1145/3178487.3178495>