Efficient, Out-of-Memory Sparse MTTKRP on Massively Parallel Architectures

Andy Nguyen
University of Oregon
andyn@uoregon.edu

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

Fabio Checconi
Intel Labs
fabio.checconi@intel.com

Jan Laukemann[*]
University of Erlangen-Nürnberg
jan.laukemann@fau.de

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

Yongseok Soh
University of Oregon
ysoh@uoregon.edu

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

Fabrizio Petrini
Intel Labs
fabrizio.petrini@intel.com

Jee W. Choi
University of Oregon
jeec@uoregon.edu

ABSTRACT

Tensor decomposition (TD) is an important method for extracting latent information from high-dimensional (multi-modal) *sparse* data. This study presents a novel framework for accelerating fundamental TD operations on massively parallel GPU architectures. In contrast to prior work, the proposed Blocked Linearized CoOrdinate (BLCO) format enables efficient *out-of-memory* computation of tensor algorithms using a *unified implementation* that works on a *single tensor copy*. Our adaptive blocking and linearization strategies not only meet the resource constraints of GPU devices, but also accelerate data indexing, eliminate control-flow and memory-access irregularities, and reduce kernel launching overhead. To address the substantial synchronization cost on GPUs, we introduce an opportunistic conflict resolution algorithm, in which threads collaborate instead of contending on memory access to discover and resolve their conflicting updates *on-the-fly*, without keeping any auxiliary information or storing non-zero elements in specific mode orientations. As a result, our framework delivers superior in-memory performance compared to prior state-of-the-art, and is the *only* framework capable of processing out-of-memory tensors. On the latest Intel and NVIDIA GPUs, BLCO achieves $2.12 - 2.6\times$ geometric-mean speedup (with up to $33.35\times$ speedup) over the state-of-the-art mixed-mode compressed sparse fiber (MM-CSF) on a range of real-world sparse tensors.

CCS CONCEPTS

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

[*]The author was employed by Intel when this research was conducted.

KEYWORDS

Tensor decomposition, MTTKRP, sparse tensors, sparse formats, parallel performance, GPU

ACM Reference Format:
Andy Nguyen, Ahmed E. Helal, Fabio Checconi, Jan Laukemann, Jesmin Jahan Tithi, Yongseok Soh, Teresa Ranadive, Fabrizio Petrini, and Jee W. Choi. 2022. Efficient, Out-of-Memory Sparse MTTKRP on Massively Parallel Architectures. In *2022 International Conference on Supercomputing (ICS '22), June 28–30, 2022, Virtual Event, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3524059.3532363

1 INTRODUCTION

Tensors are higher-order generalization of matrices, and they provide a natural abstraction for complex and inter-related data. Many critical applications, such as data mining [26, 39], social network analytics [14, 42], cybersecurity [9, 13], and healthcare [16, 18], generate massive amounts of multi-dimensional (multi-modal) data as *sparse tensors* that can be analyzed quickly and efficiently using *tensor decomposition* (TD). The most popular TD method is the canonical polyadic decomposition (CPD) model, which approximates a tensor as a sum of a finite number of rank-one tensors such that each rank-one tensor corresponds to a useful data property [5, 25]. Computing the CPD of a sparse tensor is typically dominated by the *matricized tensor times Khatri-Rao product* (MTTKRP) operation, which makes up approximately 90% of the total execution time [49].

TD algorithms for high-dimensional sparse data are challenging to execute on emerging parallel architectures due to their low arithmetic intensity, irregular memory access, workload imbalance, and synchronization overhead [10, 17]. To improve the performance of these memory-bound workloads, recent studies [12, 30, 35, 37, 40] exploit massively parallel architectures equipped with High Bandwidth Memory (HBM), namely GPUs, to accelerate the MTTKRP kernel. While such accelerators deliver memory bandwidth exceeding 2 TB/s [1], they suffer from limited memory capacity and high memory-access latency, which is in the order of hundreds of processor cycles [20, 33]. Moreover, the high memory latency together with the massive number of threads can substantially increase the synchronization overhead.

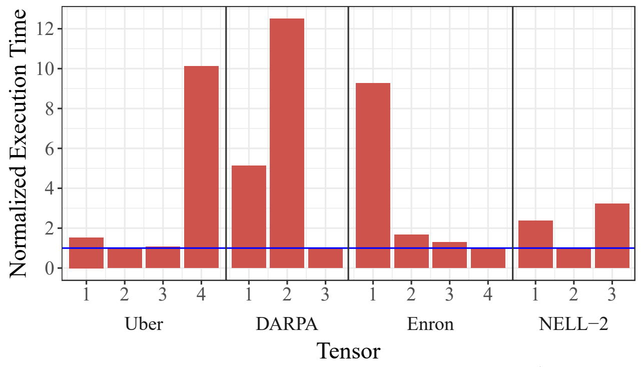

Figure 1: The MTTKRP execution time of MM-CSF[1] across all modes on the A100 GPU, normalized by the lowest execution time (denoted by the blue line) for each data set. The decomposition rank is 32. For NELL-2, modes 1 and 3 take 2–3× longer to execute than mode 2, while for Uber and Enron, one mode takes *more than* 9× *longer* to execute. For DARPA, mode 1 and mode 2 take 5× and 12× longer, respectively, than mode 3. Note that the number of FLOPs computed is *identical* across modes for each data set.

Therefore, the prior studies focus primarily on designing *sparse formats* that *compress* the tensor to decrease its memory footprint and/or *group* data-dependent non-zero elements to *reduce atomic operations*, which are particularly more expensive on massively parallel architectures. However, these strategies result in formats that are *mode-specific*, where non-zero elements are organized/accessed according to a specific mode (i.e., dimension) orientation. Mode-specific formats typically require keeping multiple tensor copies [12, 30, 37] and/or extra mapping and scheduling information (e.g., flags or arrays) about groups of data-dependent non-zero elements for *every* mode [12, 30, 40], which can significantly increase their overall memory footprint.

Furthermore, in such mode-specific tensor formats, high compression along one mode may result in poor performance along other modes [35]. For example, Figure 1 illustrates the impact of mode-specific compression on the performance of MTTKRP across all modes for the state-of-the-art mixed-mode compressed sparse fiber (MM-CSF) format [35], which currently has the best performance on GPUs, achieving 2−1.4× average speedup over prior GPU formats. This result demonstrates that depending on the data set, the execution time may vary by an *order of magnitude* across modes when the compression favors one particular mode over others.

Additionally, data formats based on compressed sparse fiber (CSF), e.g., balanced CSF (B-CSF) [37] and MM-CSF [35], require mode-specific implementations for tensor operations, such as MTTKRP, as their tree-like data structure necessitates different methods of tree traversal and intermediate result accumulation for each mode. This leads to poor code scalability and portability. Lastly, the mode-specific nature of these formats (i.e., tree-based structure and/or auxiliary copies/scheduling data) makes out-of-memory (OOM) tensors (i.e., large-scale tensors that do not fit in GPU memory) difficult to process. As a result, current GPU frameworks for MTTKRP are constrained to sparse tensors that can fit in the limited device memory (i.e., in-memory) and lack support for real-world OOM tensors with billions of non-zero elements.

[1]https://github.com/isratnisa/MM-CSF

To summarize, the state-of-the-art tensor decomposition approaches for massively parallel GPU architectures rely on *mode-specific* formats to reduce data movement via *compression* and to decrease the number of atomic operations by *reordering* of non-zero elements. However, these techniques can result in (i) drastic performance loss and variations along different tensor modes, (ii) significant memory overhead to keep extra tensor copies or mapping flags/arrays, (iii) complex algorithms and code implementations to handle tensor operations across different mode orientations, and (iv) limited support for real-world data sets due to inability to process OOM tensors on memory-constrained accelerators.

To address these limitations, we propose a novel *mode-agnostic* framework for large-scale sparse tensor decomposition on GPUs. We make four key contributions:

- We analyze prior state-of-the-art sparse tensor formats and parallel MTTKRP algorithms to determine the key performance bottlenecks and limitations (Section 3).
- We introduce Blocked Linearized CoOrdinate (BLCO), a new sparse tensor format that uses an adaptive blocking and linearization approach to generate coarse-grained tensor blocks, based on the resource constraints of target devices, while exposing the fine-grained parallelism within a block to efficiently utilize the accelerator hardware. Our BLCO format accelerates indexing, reduces data movement, decreases kernel launch overhead, supports out-of-memory tensors, and enables a unified tensor representation and MTTKRP implementation (Section 4).
- We present a novel massively parallel MTTKRP algorithm that eliminates irregularities in control-flow and memory-access, while efficiently discovering and resolving conflicting updates across threads *on-the-fly*. By employing cooperative thread teams and using low-latency registers/memories, our algorithm reduces synchronization cost *without* requiring any mode-specific information or storing the data in a particular mode order (Section 5).
- We demonstrate substantial performance improvement compared to prior state of the art, achieving 2.12−2.6× geometric-mean speedup (up to 33.35× speedup) across the latest Intel and NVIDIA GPUs for a representative set of real-world tensors. Furthermore, we show the utility of BLCO in processing out-of-memory tensors in contrast to existing GPU-based TD frameworks (Section 6).

2 BACKGROUND

In this section, we provide a brief overview of tensors, their decomposition, and related notations. For more details on tensor decomposition, we direct the reader towards the work by Kolda and Bader [5, 25].

2.1 Notation

Tensors are multi-modal arrays that generalize the concepts of vectors and matrices. An N-order tensor is an array with N modes. We use the following notation in this paper:

(1) *Scalars* are written with lowercase letters (e.g., a).

(2) *Vectors* (first-order tensors) are written with bold lowercase letters (e.g., $\mathbf{a} \in \mathbb{R}^I$). The i^{th} entry of $\mathbf{a} \in \mathbb{R}^I$ is denoted a_i.
(3) *Matrices* (second-order tensors) are written with bold capital letters (e.g., $\mathbf{A} \in \mathbb{R}^{I \times J}$). The $(i,j)^{th}$ entry of $\mathbf{A} \in \mathbb{R}^{I \times J}$ is denoted $a_{i,j}$.
(4) *Higher-order tensors* are written with Euler script letters (e.g., $\mathcal{X} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$). The $(i_1, \ldots, i_N)^{th}$ entry of the N-order tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ is denoted $x_{i_1,\ldots,i_N}$.
(5) *Fibers* are the analogue of matrix rows/columns for higher-order tensors. A mode-n fiber of a tensor $\mathcal{X}$ is any vector formed by fixing all indices of $\mathcal{X}$, *except* the n^{th} index (e.g., a matrix column is defined by fixing the second index, and is therefore a mode-1 fiber).
(6) *Hadamard* product is an element-wise product between two vectors or matrices, and is denoted by the symbol "$*$".
(7) *Kronecker* product between two matrices $\mathbf{A} \in \mathbb{R}^{I \times J}$ and $\mathbf{B} \in \mathbb{R}^{K \times L}$ produces the matrix $\mathbf{C} \in \mathbb{R}^{IJ \times KL}$, where

$$\mathbf{C} = \begin{bmatrix} a_{1,1}\mathbf{B} & a_{1,2}\mathbf{B} & \cdots & a_{1,J}\mathbf{B} \\ a_{2,1}\mathbf{B} & a_{2,2}\mathbf{B} & \cdots & a_{2,J}\mathbf{B} \\ \vdots & \vdots & \ddots & \vdots \\ a_{I,1}\mathbf{B} & a_{I,2}\mathbf{B} & \cdots & a_{I,J}\mathbf{B} \end{bmatrix}$$

and is denoted $\mathbf{A} \otimes \mathbf{B}$.

2.2 Canonical Polyadic Decomposition

The canonical polyadic decomposition (CPD) is a widely used tensor factorization model in data analysis. CPD approximates an N-order tensor $\mathcal{X}$ by the sum of R outer products of N appropriately sized vectors, for some *a priori* chosen R. Each of the N vectors in a given outer product corresponds to a particular tensor mode. The outer products are called the rank-1 tensors of the decomposition and the quantity R is called the decomposition *rank*. By arranging the R vectors corresponding to a particular mode as the columns of a matrix, we obtain the *factor matrix* associated with that mode. The decomposition of $\mathcal{X}$ can then be written in terms of its factor matrices. For example, if $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$ is a third-order tensor, the CPD of $\mathcal{X}$ may be written in terms of three factor matrices $\mathbf{A}^{(1)} \in \mathbb{R}^{I \times R}$, $\mathbf{A}^{(2)} \in \mathbb{R}^{J \times R}$, and $\mathbf{A}^{(3)} \in \mathbb{R}^{K \times R}$, as shown in Figure 2.

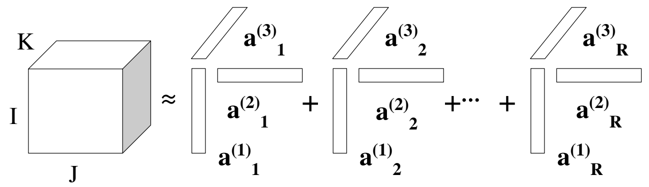

Figure 2: Rank-R CPD of a third-order tensor. The factor matrix $\mathbf{A}^{(1)}$ consists of vectors $\mathbf{a}_1^{(1)}, \mathbf{a}_2^{(1)}, \cdots, \mathbf{a}_R^{(1)}$.

CANDECOMP/PARAFAC alternating least squares (CP-ALS) algorithm is a popular method for calculating CPD. During each CP-ALS iteration, we update the factor matrix corresponding to a given tensor mode by solving a linear least squares problem, while fixing the remaining factor matrices; this is done exactly once for each factor matrix per iteration. The most expensive operation of CP-ALS, as well as many other tensor algorithms, is the the matricized tensor times Khatri-Rao product (MTTKRP) [49]. The CP-ALS algorithm for an N-order tensor is detailed in Algorithm 1. Line 4 shows the MTTKRP computations.

Algorithm 1 The CP-ALS algorithm.

```
Input: An $N$-order sparse tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$, randomly initialized dense factor matrices $\mathbf{A}^{(1)} \in \mathbb{R}^{I_1 \times R}$, $\mathbf{A}^{(2)} \in \mathbb{R}^{I_2 \times R}$, $\cdots$, $\mathbf{A}^{(N)} \in \mathbb{R}^{I_N \times R}$.
Output: Updated factor matrices that approximate $\mathcal{X}$.
1: repeat
2:     for $n = 1, \ldots, N$ do
3:         $\mathbf{V} \leftarrow \mathbf{A}^{(1)T}\mathbf{A}^{(1)} * \cdots * \mathbf{A}^{(n-1)T}\mathbf{A}^{(n-1)} * \mathbf{A}^{(n+1)T}\mathbf{A}^{(n+1)} * \cdots * \mathbf{A}^{(N)T}\mathbf{A}^{(N)}$
4:         $\mathbf{M} \leftarrow \mathbf{X}_{(n)}(\mathbf{A}^{(N)} \odot \cdots \odot \mathbf{A}^{(n+1)} \odot \mathbf{A}^{(n-1)} \odot \cdots \odot \mathbf{A}^{(1)})$
5:         $\mathbf{A}^{(n)} \leftarrow \mathbf{M}\,\mathbf{V}^{\dagger}$        ▷ $\dagger$ denotes the pseudo-inverse
6:     end for
7: until fit ceases to improve or maximum # of iterations reached
8: return $\mathbf{A}^{(1)}, \cdots, \mathbf{A}^{(N)}$
```

2.3 Sparse MTTKRP

The matricized tensor times Khatri-Rao product (MTTKRP) kernel involves two basic operations:

(1) *Tensor matricization* is the process in which a tensor is unfolded into a matrix. The mode-n matricization of a tensor $\mathcal{X}$, denoted by $\mathbf{X}_{(n)}$, is obtained by laying out the mode-n fibers of $\mathcal{X}$ as the columns of $\mathbf{X}_{(n)}$.
(2) The *Khatri-Rao product* [32] is the "matching column-wise" Kronecker product of two matrices. Given $\mathbf{A} \in \mathbb{R}^{I \times R}$ and $\mathbf{B} \in \mathbb{R}^{J \times R}$, their Khatri-Rao product is $\mathbf{K} = \mathbf{A} \odot \mathbf{B}$, where $\mathbf{A} \odot \mathbf{B} = [\mathbf{a}_1 \otimes \mathbf{b}_1\ \mathbf{a}_2 \otimes \mathbf{b}_2 \ldots \mathbf{a}_R \otimes \mathbf{b}_R] \in \mathbb{R}^{I \cdot J \times R}$.

For an N-order tensor $\mathcal{X}$ and factor matrices $\mathbf{A}^{(1)}, \mathbf{A}^{(2)}, \cdots, \mathbf{A}^{(N)}$, the mode-$n$ MTTKRP is given by

$$\mathbf{M} = \mathbf{X}_{(n)}(\mathbf{A}^{(N)} \odot \cdots \odot \mathbf{A}^{(n+1)} \odot \mathbf{A}^{(n-1)} \odot \cdots \odot \mathbf{A}^{(1)}). \quad (1)$$

$\mathbf{M}$ has the same number of rows and columns as $\mathbf{A}^{(n)}$ and it is utilized to update $\mathbf{A}^{(n)}$ in CP-ALS (Line 5 in Algorithm 1). Note that for *sparse* tensors, explicitly forming and multiplying $\mathbf{X}_{(n)}$ and the corresponding Khatri-Rao product (cf. (1)) is expensive and unnecessary. In practice [5, 17, 28, 49], MTTKRP is calculated using the factor matrix rows corresponding to the non-zero elements of the tensor only as needed (c.f. Figure 3).

3 SPARSE TENSOR FORMATS FOR GPUS

The state-of-the-art tensor formats for massively parallel GPUs use list-based [12, 30, 40] or tree-based [35, 37] data structures to store high-dimensional sparse data in a *mode-specific* form. In this section, we provide an overview of the flagged coordinate (F-COO) and MM-CSF formats, which are representative of the two main format categories for GPU architectures. In contrast to sparse linear algebra, higher-order tensor algorithms typically perform tensor operations on *every mode orientation*. Using MTTKRP as a case study, we illustrate the challenges in optimizing and efficiently executing sparse tensor operations on GPUs.

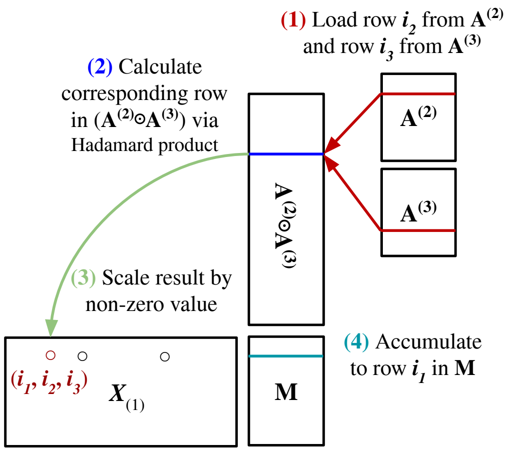
(1) Load row i_2 from $\mathbf{A}^{(2)}$ and row i_3 from $\mathbf{A}^{(3)}$
(2) Calculate corresponding row in $(\mathbf{A}^{(2)} \odot \mathbf{A}^{(3)})$ via Hadamard product
$\mathbf{A}^{(2)}$
$\mathbf{A}^{(3)}$
$\mathbf{A}^{(2)} \odot \mathbf{A}^{(3)}$
(3) Scale result by non-zero value
(4) Accumulate to row i_1 in $\mathbf{M}$
(i_1, i_2, i_3)
$\mathcal{X}_{(1)}$
$\mathbf{M}$

Figure 3: Mode-1 MTTKRP operation for a third-order sparse tensor. For each non-zero element with index (i_1, i_2, i_3), rows i_2 and i_3 from factor matrices $\mathbf{A}^{(2)}$ and $\mathbf{A}^{(3)}$ are fetched (1) and their Hadamard product (element-wise product) is calculated (2). The result is scaled by the non-zero element's value (3), and then accumulated to matrix row i_1 from matrix M (4), which is later used to calculate $\mathbf{A}^{(1)}$.

i_1	i_2	i_3	v
1	1	1	1.0
1	1	2	2.0
1	3	3	3.0
2	1	2	4.0
2	1	3	5.0
3	1	2	6.0
3	4	4	7.0
4	2	1	8.0
4	2	2	9.0
4	3	3	10.0
4	3	4	11.0
4	4	4	12.0

(a) COO.

	bf	i_2	i_3	v
sf=1	1	1	1	1.0
	1	1	2	2.0
	0	3	3	3.0
	1	1	2	4.0
sf=1	0	1	3	5.0
	1	1	2	6.0
	0	4	4	7.0
	1	2	1	8.0
sf=0	1	2	2	9.0
	1	3	3	10.0
	1	3	4	11.0
	0	4	4	12.0

(b) F-COO for mode-1 MTTKRP.

Figure 4: Comparison between COO (4a) and F-COO (4b). In F-COO, the target mode index (i_1) is replaced by bf (bit flag) that goes from 1 to 0 when the index changes. The sf (start flag) stores a bit for each non-zero group, where 1 indicates that a new target index has been encountered by the group.

3.1 F-COO Format

F-COO is an example of list-based formats that explicitly store non-zero elements with their coordinate indices. The simplest form of this category is the coordinate (COO) format, which keeps N+1 lists for an N-order tensor—N lists for the indices, and one list for the non-zero values. While COO is a *mode-agnostic* format, it suffers from substantial synchronization overhead due to update conflicts across threads.

For example, Figure 3 depicts the mode-1 MTTKRP operation, where every thread that is processing non-zero elements with mode-1 index of i_1 (red and black circles) accumulates its partial result to row i_1 of matrix **M** (step (4)). This results in chains of read-after-write (RAW) data hazards, which require expensive locks or atomic operations to resolve; on GPUs, this can quickly become a severe performance bottleneck because of the large number of concurrent threads and the high-latency memory system.

To address this issue, F-COO [30] stores sparse tensors in a sorted *mode-specific* form to *group* non-zero elements with the same i_1 index together (same i_1 for mode-1 MTTKRP, same i_2 for mode-2 MTTKRP, etc.), so that the accumulation of partial results can be performed locally using segmented scan [44, 54] and globally using atomic operations when group boundaries are crossed. Additionally, F-COO keeps extra mapping/scheduling information (flags) for synchronization to delineate the end of a non-zero group. Thus, the F-COO format needs to keep N tensor copies in the GPU memory for an N-order tensor. As a result, while F-COO reduces the number of global atomic operations, it has high memory usage due to extra data. Figure 4 shows an example sparse tensor in the COO (4a) and F-COO (4b) representations.

3.2 MM-CSF Format

MM-CSF represents a class of compressed formats that use tree-based structures to encode higher-order tensors. The original CSF format [47, 49] extends traditional compressed matrix formats, such as the compressed sparse row (CSR) format, by storing a tensor as a collection of index sub-trees with mode-specific ordering. Given a CSF representation with a mode ordering of 1-2-3, where 1 is the root mode and 3 is the leaf mode, the root node of each sub-tree represents the factor matrix row that will be updated during *mode*-1 MTTKRP, and the leaf nodes represent the non-zero elements that contribute to that update.

Thus, to calculate MTTKRP for all modes, multiple CSF copies with different root modes are needed, which increases the memory footprint by a factor of N, where N is the number of modes. Alternatively, the index sub-trees can be traversed both bottom-up and top-down, meeting at the tree level with the target mode for MTTKRP. While the second approach allows computing MTTKRP for all modes with one copy of the CSF format, it requires expensive synchronization to avoid update conflicts as well as separate tree traversal implementations. Moreover, regardless of the strategy used, CSF suffers from *workload imbalance* due to the variable size of each sub-tree.

Balanced CSF (B-CSF) [38] creates sub-trees that are more balanced, but still requires N copies of the tensor. The state-of-the-art MM-CSF [36] improves upon B-CSF by using a single copy of the tensor. This is achieved by analyzing fiber density (i.e., number of non-zero elements in a fiber) and creating sub-trees with fibers that are as dense as possible. However, the root of these sub-trees can come from any mode; hence, different traversal methods and parallel algorithms are required to perform MTTKRP, based on the mode that is being calculated, leading to drastic performance variations across different modes as shown in Figure 1. Furthermore, the complexity of the tree-based structure requires different implementation for each tensor order (i.e., number of modes) and it also restricts MM-CSF to tensors that can fit in the limited GPU memory. As a result, the current MM-CSF implementation only supports 3- and 4-dimensional tensors and cannot handle OOM tensors. Figure 5 shows an example of the MM-CSF format for the tensor from Figure 4a.

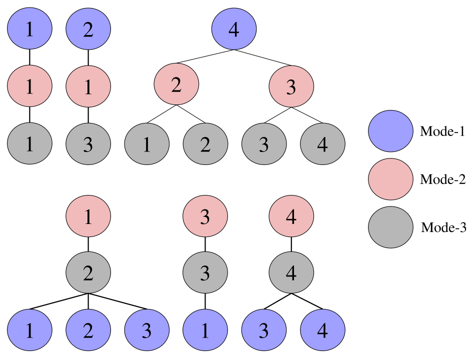

Figure 5: MM-CSF format for the sparse tensor from Figure 4a. Unlike CSF, where every sub-tree has the same mode orientation, MM-CSF identifies fibers with the highest number of non-zero elements, and then constructs sub-trees with different mode orientations.

4 THE BLCO FORMAT

To address the limitations of prior GPU-based formats, we propose the Blocked Linearized CoOrdinate (BLCO) format, a new sparse tensor representation devised for massively parallel architectures. BLCO linearizes and aggregates non-zero elements into coarse-grained blocks that meet the resource constraints of target GPUs to (i) minimize data movement, (ii) accelerate indexing, (iii) enable a unified tensor representation and algorithmic implementation, and (iv) support out-of-memory tensor computation. Figure 6 depicts an example of the BLCO format for the sparse tensor in Figure 4a. The generation of BLCO tensors consists of two stages: tensor linearization (Section 4.1) and adaptive blocking (Section 4.2).

4.1 Tensor Linearization

The BLCO format leverages index linearization [15, 17] to map multi-dimensional space onto one-dimensional space, such that a point in N-D space, represented by N coordinates, can be mapped to a point on an encoding line, represented by a *single* index. The length of the encoding line determines the number of bits required to represent linearized indices. During computation (e.g., MTTKRP), the linear indices are *de-linearized* to recover the original coordinates. Thus, fast de-linearization is important for high-performance execution of tensor algorithms using linearized formats.

To efficiently encode multi-dimensional spaces with irregular shapes (which typically arise in higher-order sparse data) in a mode-agnostic way, prior state-of-the-art linear format (ALTO [17]) recursively partitions the multi-dimensional space and "traverses" this space linearly using a compact space-filling curve. When the multi-dimensional space is regular, i.e., all modes have the same length, the resulting space-filling curve is similar to Morton-Z ordering [34]. This linearization outperforms state-of-the-art tree-, list-, and block-based formats [17] on CPU-based platforms, and therefore, we adopt its ordering for BLCO.

Such a *mode-agnostic* linearized encoding results in an *interleaving* of bits from different mode indices, as shown in Figure 6a. To quickly process indices encoded in this manner, efficient support for bit-level *scatter* and *gather* operations are needed. However, GPUs lack native support for advanced bit manipulation, and emulating

l		v
0	$(000000)_2$	1.0
4	$(000100)_2$	2.0
5	$(000101)_2$	4.0
10	$(001010)_2$	8.0
12	$(001100)_2$	6.0
15	$(001111)_2$	9.0
33	$(100001)_2$	5.0
48	$(110000)_2$	3.0
57	$(111001)_2$	10.0
61	$(111101)_2$	11.0
62	$(111110)_2$	7.0
63	$(111111)_2$	12.0

(a) Initial linearization.

b	l		v
0	0	$(00000)_2$	1.0
	16	$(10000)_2$	2.0
	17	$(10001)_2$	4.0
	6	$(00110)_2$	8.0
	18	$(10010)_2$	6.0
	23	$(10111)_2$	9.0
1	1	$(00001)_2$	5.0
	8	$(01000)_2$	3.0
	11	$(01011)_2$	10.0
	27	$(11011)_2$	11.0
	30	$(11110)_2$	7.0
	31	$(11111)_2$	12.0

(b) BLCO tensor.

Figure 6: BLCO format for the sparse tensor in Figure 4a, where the bits of linearized indices are color-coded to indicate different modes. First, BLCO linearizes the tensor (6a) based on a recursive partitioning of the multi-dimensional space [17]. Next, it aggregates non-zero elements into blocks (6b) according to accelerators' resource constraints, namely, on-device memory, length of encoding line, and support for bit manipulation. For simplicity, we assume that the maximum length of the encoding line is 32 (2^5) and each block has no more than 6 non-zero elements. Note that BLCO re-encodes the linearized index (l) to allow the use of bitwise mask/shift (which are efficiently supported on GPUs) for de-linearization instead of bit-level gather/scatter.

these instructions can be prohibitive,[2] leading to inefficient tensor processing operations.

To address this issue, the BLCO format arranges the non-zero elements using ALTO ordering, but *re-encodes* the linearized indices to allow the use of bitwise *shift* and *mask* (AND) instructions, which are natively supported on accelerators, to de-linearize the indices. As illustrated in Figure 6b, BLCO achieves this goal by rearranging the encoding bits of linearized indices (l) into contiguous mode sets that can be quickly extracted on GPUs during tensor computations. This re-encoding resembles the index concatenation of the LCO [15] format, but with bitwise shift and mask used for de-linearization instead of arithmetic division and modulo.

4.2 Adaptive Blocking

To map a tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ into a linearized form, an encoding line of length $I_1 \times \cdots \times I_N$ is required. As the number of modes and their lengths increase, the encoding line and the range of linear index values can significantly expand. Hence, large-scale tensors may require more than 64 bits to encode a linear index. Since GPUs do not provide native support for large integer (more than 64 bits) operations, a custom implementation[3] of large integer arithmetic is needed, which are typically not as efficient as native instructions. In addition, many tensors can require a slightly higher bit resolution (only a *few* additional bits) than 64 bits, so using large integers for linearized indices may lead to wasted memory.

[2]For a third-order tensor, we estimate that a naïve emulation would require 276 bitwise operations to de-linearize each non-zero element.
[3]https://github.com/curtisseizert/CUDA-uint128

Most importantly, the state-of-the-art libraries for sparse MTTKRP require the *entire* tensor to be in the *limited* GPU memory, due to their compressed and/or mode-specific tensor formats. In particular, it is challenging to stream the data in small *chunks* that can be partially processed using tree- (e.g., B-CSF [38] and MM-CSF [36]) and block-based (e.g., HiCOO [28]) formats, where the granularity of tensor chunks (e.g., sub-trees and HiCOO blocks) are difficult to control. List-based formats (e.g., GenTen [40] and F-COO [31]) are more amenable to streaming, but will require additional information for synchronization across these data chunks.

To address these issues, we propose *adaptive blocking* to aggregate non-zero elements into coarse-grained blocks by partitioning the tensor into smaller sub-tensors. Blocking the tensor to exploit density structures and to compress the data has been previously explored by the HiCOO [28] format. However, such a compression comes at the cost of severe workload imbalance across blocks, due to the irregular spatial distributions of sparse data [17, 29], and it requires expensive tuning to find the best block size. In contrast, the proposed blocking technique aims to meet the requirements of memory-constrained GPUs, while at the same time generate the largest possible blocks that can efficiently utilize these throughput-oriented accelerators with massive parallelism.

As such, our BLCO format first uses the *uppermost bits* from every mode of linearized indices that exceeds the target integer size to form the initial blocks, which allows the sub-spaces spanned by these blocks to naturally adapt to the underlying tensor space. For example, if a tensor requires 72 bits for linearized indices and the size of target integers is 64 bits, a total of 8 (72 − 64) uppermost bits across all the modes are stripped from the linear indices and used as a *key* to group the non-zero elements into blocks; then, these bits are stored as metadata along with each block (b), as depicted in Figure 6b. This strategy does *not* require expensive tuning to create the blocks and, compared to using longer encoding lines, it reduces the memory footprint and leverages more efficient native integer (64-bit) instructions. Next, BLCO further splits the initial blocks, if needed, based on the available on-device memory to ensure that each block has no more than the maximum number of non-zero elements that can fit in the target device.

While the resulting BLCO format may still suffer from variance in the number of non-zero elements across blocks, it exposes the fine-grained parallelism within a block, due to its linearized list-based form. Specifically, it allows workload to be partitioned at the granularity of a non-zero element rather than a compressed block. Hence, a GPU hardware scheduler can automatically hide the execution and memory-access latency as well as balance the workload across threads, as long as there are enough non-zero elements to be processed in the GPU memory. In addition, our massively parallel MTTKRP algorithm (Section 5), with opportunistic conflict resolution, allows BLCO blocks to be processed independently. Therefore, once the GPU has processed a block, it will fetch the next available block to automatically occupy any available GPU resources. These properties enable our BLCO format and MTTKRP algorithm to seamlessly handle OOM tensors, in contrast to prior work.

Our implementation uses device queues (SYCL queues or CUDA streams) to launch BLCO blocks, allowing the computation of active blocks to be done asynchronously with the transmission of pending blocks. Each device queue has reserved memory, corresponding

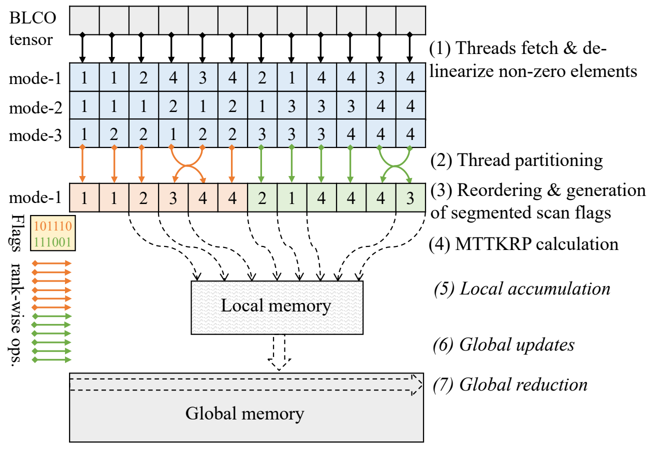

Figure 7: Steps (1) – (7) illustrates mode-1 MTTKRP with hierarchical conflict resolution for the BLCO tensor from Figure 6b. For register-based conflict resolution, step (6) is the last one, where the results are written directly, bypassing the local accumulation in step (5), to the factor matrix in global memory using atomic updates, as opposed to writing to temporary factor matrix copies. In all conflict resolution mechanisms, steps (1) – (4) are common and writing the updates happens at segment boundaries.

to the number of non-zero elements that can be processed at one time, which is reused across BLCO blocks assigned to the same queue. In our experiments, we use up to 8 device queues and set the maximum number of non-zero elements per block to 2^{27} to fill the GPU; these parameters allow for enough concurrency and further tuning them has negligible performance impact.

Hypersparse tensors may generate several BLCO blocks that can fit in the same memory reservation of a single device queue. Launching these blocks across multiple queues can potentially incur kernel launch overhead on some GPU architectures. To address this, we batch all BLCO blocks that can be processed by one device queue into a GPU kernel and explicitly store block mappings and element offsets at the work-group (thread block) boundaries. This additional batching information is calculated during format construction and thus creates minimal overhead during tensor computations.

5 BLCO-BASED SPARSE MTTKRP

On massively parallel systems such as GPUs, atomic updates to global memory can be expensive because of the sheer number of concurrently executing threads that could conflict and the long data access latency. Therefore, prior studies focused on reducing atomic operations by storing the data in a mode-specific form [12, 30, 36, 38] and/or keeping mode-specific mapping/scheduling information [12, 30, 40]. However, these strategies lead to drastic performance variation across modes, substantial memory overhead for extra tensor copies, and/or complex algorithms and implementations (c.f. Section 3).

Here, we describe a novel massively parallel MTTKRP algorithm that eliminates control-flow and memory-access irregularities while resolving update conflicts (RAW hazards) using an *opportunistic on-the-fly* update mechanism, which reduces atomic operations without requiring extra tensor copies or mode-specific information.

5.1 Hierarchical Conflict Resolution

Figure 7 illustrates our massively parallel algorithm for mode-1 MTTKRP kernel on the third-order sparse tensor from Figure 4a. Note that the non-zero elements are linearized and grouped according to the BLCO format (Figure 6b). For simplicity, we assume a work-group (thread-block) size of 12 and tile size of 6 work-items (threads) in the figure.

To eliminate control-flow and memory-access irregularities, the proposed algorithm has two phases: processing and computing. In the processing phase, steps (1) – (3), each thread is assigned to a non-zero element and threads collaborate to perform *on-the-fly* de-linearization and reordering of non-zero elements as well as generation of segmented scan flags. In the computing phase, steps (4) – (7), threads are reassigned to perform the rank-wise MTTKRP computations and merge conflicting updates at the register, local memory, and global memory levels. In each phase, the threads and their global memory accesses are coalesced.

5.1.1 Processing Phase. In step (1), a group of threads (work-group or thread-block) is assigned to a certain number of non-zero elements and they first collaborate so that each thread loads a linearized non-zero element (i.e., a linear index and its corresponding value) from the global memory in a coalesced manner. Each thread then proceeds to de-linearize the linear index and recover the multi-dimensional coordinates, where each coordinate can be calculated independently to expose more instruction-level parallelism. In step (2), the threads are partitioned into multiple tiles, where a tile size is no more than the sub-group (warp) size. Next, in step (3), the threads within a tile collaborate to reorder the non-zero elements into groups (segments) according to their target mode indices (mode-1 in the example) and then generate segmented scan flags, where 1 indicates the start of a new segment. To minimize thread divergence and data access latency, we implement the reordering of non-zero elements via parallel histogram and prefix sum, using sub-group (warp-level) data exchange primitives (e.g., *shuffle* operations), and broadcast/store the segmented scan flags across threads in a low-latency register. After each thread completes the on-the-fly processing phase, the non-zero elements are stored, according to their new order, in local memory for later access.

5.1.2 Computing Phase. Once the data has been processed, we reassign the threads to calculate the rank-wise MTTKRP computations in step (4), where each thread is responsible for one or more elements along the decomposition rank (i.e., threads process the same non-zero element and perform rank-wise operations in parallel). The algorithm iterates over the non-zero elements using the segmented scan flags, so that each thread accumulates its partial results to a register as long as the target mode index remains the same. At the end of each segment, i.e., when the index changes, each thread writes the accumulated result to the stash (a software-controlled cache) in local memory (shown as "step (5)" in the figure). In step (6), when all the non-zero elements have been processed, the results are copied from the stash (local memory) to one of the multiple copies of the factor matrix in global memory. By using multiple copies of the factor matrix, we can minimize the probability of conflict when multiple thread blocks are copying their results back to global memory at the same time. Finally, in step (7), the multiple factor matrix copies are merged in global memory to produce the final result.

5.2 Register-based Conflict Resolution

Our register-based conflict resolution is a subset of the hierarchical algorithm, discussed above. It bypasses the local memory entirely and writes the accumulated result in registers to global memory, without the need for a final global reduction. Specifically, after each thread calculates and accumulates the result in its register and reaches a segment boundary, it bypasses step (5) and completes execution at step (6) after writing the updates to the final factor matrix, as opposed to one of its copies, using atomic operations.

5.3 Adaptation Heuristic

The proposed synchronization mechanisms use different number of atomic operations to perform MTTKRP. Register-based conflict resolution requires more atomic operations, as updates at each segment boundary need atomic add operations to the factor matrix in global memory. Hierarchical conflict resolution uses fewer atomic operations, as updates at each segment boundary are copied first to the local-memory stash, and then atomic operations are used only at the end of the work-group (thread-block) execution to write the accumulated result in the stash to the factor matrix in global memory. Furthermore, multiple copies of the factor matrix can be used to reduce the probability of an update conflict, at the cost of a final global reduction.

We propose a simple heuristic for selecting the best conflict resolution mechanism based on the characteristics of target modes and GPU devices. In modern GPUs [1, 8], the execution units (EUs) are aggregated into subslices or streaming multi-processors (SMs). Multiple subslices are grouped into a GPU slice or a graphics processing cluster (GPC). Our heuristic selects the hierarchical conflict resolution, when the target mode length is less than the number of subslices (SMs). At this mode length, the contention from atomic operations to global memory is severe, and using a local-memory stash and factor matrix copies (one copy for each slice or GPC) alleviates this contention. For all other cases, we use the register-based conflict resolution.

6 EXPERIMENTS

We evaluate the proposed sparse MTTKRP and BLCO format[4] using a representative set of in-memory and out-of-memory tensors with different characteristics. We compare our performance to the state-of-the-art sparse tensor frameworks for massively parallel GPUs, namely, MM-CSF [36], GenTen [40], and F-COO [30]. While the other frameworks only support tensors that can fit in the device memory, BLCO can process both in- and out-of-memory tensors and delivers superior performance across all in-memory tensors.

6.1 Evaluation Setup

6.1.1 Test Platform. We conduct the experiments on the latest Intel discrete GPU with a single-tile (denoted *Intel Device1*) and two NVIDIA GPUs from the most recent micro-architecture generations: A100 (*Ampere*) and V100 (*Volta*). For the format generation, we use

[4]Available at https://github.com/jeewhanchoi/blocked-linearized-coordinate

Table 1: Hardware and software setup.

	CPU	GPU	
Model	AMD EPYC 7662	NVIDIA A100	NVIDIA V100
μ-arch	Zen 2	Ampere	Volta
Frequency	3.3 GHz	1.41 GHz	1.38 GHz
Cores	64 (×2 sockets)	108 (SM[1]) 6912 (CC[2])	80 (SM) 5120 (CC)
Caches	2MB L1, 32MB L2, 256MB L3	15MB L1D, 40MB L2	10MB L1D, 6MB L2
DRAM (Bandwidth)	256 GB	40 GB (1555 GB/s)	32 GB (900 GB/s)
Interconnect	PCI-e Gen 4	NVLink 3.0	NVLink 2.0
OS/Driver (version)	RHEL (8.3)	Driver (470.42.01)	Driver (440.33.01)
Compiler	gcc 9.3.0	nvcc 11.4	nvcc 11.0

[1] Streaming Multiprocessor [2] CUDA Cores

Table 2: The sparse tensor data sets used for evaluation, ordered by the number of non-zero elements.

Tensor	Dimensions	NNZs	Density
NIPS	$2.5K \times 2.9K \times 14K \times 17$	$3.1M$	1.8×10^{-06}
Uber	$183 \times 24 \times 1.1K \times 1.7K$	$3.3M$	3.8×10^{-04}
Chicago	$6.2K \times 24 \times 77 \times 32$	$5.3M$	1.5×10^{-02}
Vast-2015	$165.4K \times 11.4K \times 2$	$26M$	7.8×10^{-07}
DARPA	$22.5K \times 22.5K \times 23.8M$	$28.4M$	2.4×10^{-09}
Enron	$6K \times 5.7K \times 244.3K \times 1.2K$	$54.2M$	5.5×10^{-09}
NELL-2	$12.1K \times 9.2K \times 28.8K$	$76.9M$	2.4×10^{-05}
FB-M	$23.3M \times 23.3M \times 166$	$99.6M$	1.1×10^{-09}
Flickr	$319.7K \times 28.2M \times 1.6M \times 731$	$112.9M$	1.1×10^{-14}
Delicious	$532.9K \times 17.3M \times 2.5M \times 1.4K$	$140.1M$	4.3×10^{-15}
NELL-1	$2.9M \times 2.1M \times 25.5M$	$143.6M$	9.1×10^{-13}
Amazon	$4.8M \times 1.8M \times 1.8M$	$1.7B$	1.1×10^{-10}
Patents	$46 \times 239.2K \times 239.2K$	$3.6B$	1.4×10^{-03}
Reddit	$8.2M \times 177K \times 8.1M$	$4.7B$	4.0×10^{-10}

a dual-socket AMD Epyc 7662 CPU system and employ 128 threads. Our prototype is implemented in Data Parallel C++ (DPC++) [41] as well as CUDA [23]. Since the current sparse tensor frameworks lack portability across GPU architectures, we ported the state-of-the-art MM-CSF to DPC++ using the Intel DPC++ Compatibility Tool [2]. Due to confidentiality requirements, Table 1 summarizes the publicly available specifications of the target hardware and software environment.

6.1.2 Data Sets. We consider 14 *real-world* tensor data sets from the FROSTT [46] and HaTen2 [19] open-source repositories that cover a wide range of tensor properties and sparsity structures—number of modes, mode lengths, number of non-zero elements, and density. Table 2 lists the sparse tensor data sets used for evaluation, ordered by increasing number of non-zero elements (NNZs). The large-scale tensors with billions of non-zero elements (namely, Amazon, Patents, and Reddit) are considered out-of-memory as they fail to execute on current tensor decomposition frameworks, which need to keep the entire tensor and factor matrices in device memory, producing memory allocation errors on target GPUs.

6.1.3 Configurations. The experiments use a decomposition rank of 32 for MTTKRP, as per prior work [36]. Using double-precision values and 64-bit integers, we report the performance as an average over 25 iterations. We tune each sparse tensor framework to the best of our abilities. For the state-of-the-art MM-CSF, we exhaustively tune the number of warps per fiber and the thread-block size and report the best execution time. While BLCO can benefit from tuning, we use our adaptation heuristic (Section 5.3) and set the thread-coarsening factor (NNZs per thread) for the hierarchical conflict resolution mechanism (Section 5.1) to 4 and 2 on Intel and NVIDIA GPUs, respectively.

6.2 Comparison Against TD Frameworks

We first compare our BLCO-based MTTKRP against the other popular sparse tensor decomposition frameworks. Figure 8 shows the MTTKRP execution time for all modes, across the GPU frameworks, normalized by the execution time of the state-of-the-art MM-CSF. The results demonstrate that BLCO consistently outperforms all other frameworks, achieving a geometric mean speedup between 2.12× and 2.6× over MM-CSF across the different GPU devices. For the other CUDA frameworks, GenTen has comparable performance to MM-CSF, outperforming MM-CSF on six out of 11 data sets, whereas F-COO has lower performance on average compared to MM-CSF, especially on the V100 GPU. The missing data points for F-COO is due to its limited support for higher-order tensors (i.e., 3-D only) and "segfault" errors.

Our performance analysis indicates that the performance of MM-CSF (as well as GenTen) can substantially decrease with higher synchronization cost, which leads to lower average performance than BLCO on the GPU devices with more expensive synchronization. Since MM-CSF reorders non-zero elements to increase tensor compression, its performance is sensitive to the number of non-zero elements per fiber. On large-scale data sets with low fiber density, such as DARPA, Enron, and FB-M, MM-CSF has lower compression, leading to significant performance degradation compared to BLCO.

6.3 Comparison Against State of the Art

For a comprehensive evaluation against the state-of-the-art MM-CSF, Figure 9 demonstrates the speedup achieved by our BLCO-based MTTKRP for every mode of the tensors that can fit in the device memory of target GPUs. The results demonstrate that our BLCO format achieves better or comparable performance to MM-CSF for every mode (up to 33.35× speedup) across all data sets, except Uber and NIPS. These data sets are not only small, but they also have exceptionally short modes, allowing the data to fit in cache; thus, the higher compression achieved by MM-CSF, due to its mode-specific nature (c.f. Table 3), translates to better performance. Yet, such a mode-specific compression leads to substantial performance variations across different modes, as shown in Figure 1, and as a result, BLCO still outperforms MM-CSF for all-mode MTTKRP (c.f. Figure 8).

6.4 Memory Traffic Analysis

Since sparse tensor decomposition is a memory-bound workload, its performance is largely limited by the data volume and the effective memory throughput. Hence, we provide detailed analysis of these memory metrics across both in- and out-of-memory tensors.

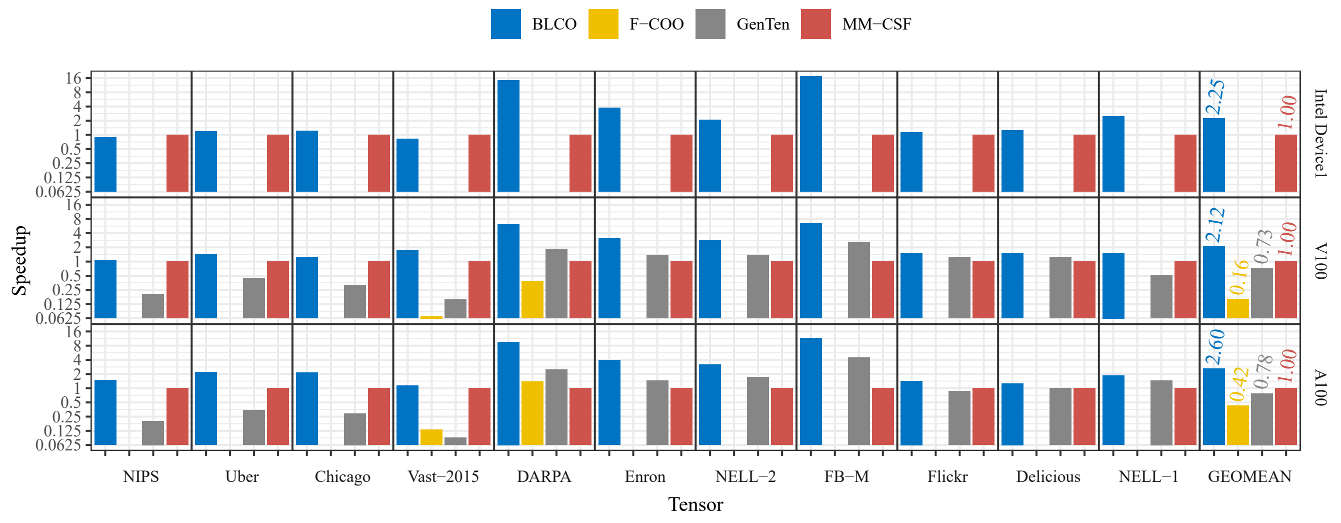

Figure 8: Comparison of BLCO-based MTTKRP against popular GPU frameworks on the data sets that fit in GPU memory. Each bar represents the speedup obtained against MM-CSF for computing MTTKRP on all modes. The right-most group of bars show the geometric mean speedup.

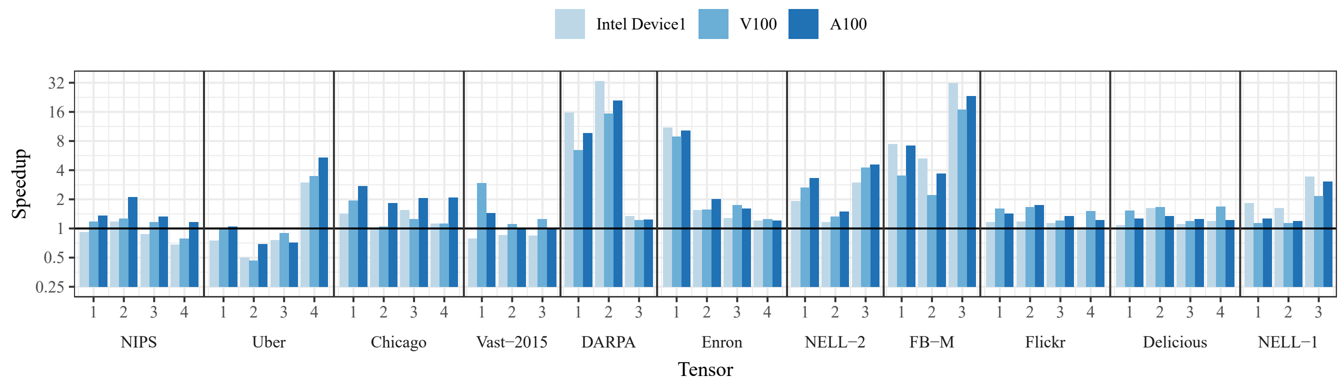

Figure 9: The per-mode speedup of BLCO-based MTTKRP against MM-CSF across *every* tensor mode for the data sets that fit in GPU memory.

6.4.1 In-Memory Tensors. Table 3 details the memory metrics of BLCO-based MTTKRP compared to MM-CSF on the A100 GPU. The metrics are collected using the Nsight Compute profiler [3]. The memory analysis shows that MM-CSF achieves higher compression than BLCO thanks to its tree-like data structure, and the total volume of data fetched from memory (as shown in column "Vol") is lower in most cases. However, due to the irregular memory access and expensive synchronization associated with traversing a tree-like tensor representation, MM-CSF under-utilizes the memory system compared to BLCO and has lower memory throughput (as shown in column "TP"). In addition, both the memory volume and throughput of MM-CSF vary significantly across modes because of its mode-specific traversal and processing of tensors, which leads to substantial performance variations (c.f. Figure 1). In contrast, while BLCO requires more data volume because of its mode-agnostic form, it achieves higher memory throughput by eliminating memory-access irregularities, exploiting data locality, and merging conflicting updates across threads in low-latency registers and memories. Thereby, BLCO fetches/writes more data from/to higher levels of the memory hierarchy in a coalesced way, leading to improved performance by up to an order of magnitude compared to MM-CSF.

6.4.2 Out-of-Memory Tensors. Figure 10 demonstrates the memory throughput of BLCO-based MTTKRP for out-of-memory tensors (Amazon, Patents, and Reddit) on the A100 GPU. Since no other GPU framework supports these tensors, a direct comparison is not possible; instead, we report the overall throughput as well as the throughput without host-device data exchange (in-memory throughput) of our BLCO-based MTTKRP, as measured using the Nsight Systems profiler [4]. The overall throughput is measured based on the total execution time (both MTTKRP computations and host-device data transfers), while the in-memory throughput only includes MTTKRP computations. Without the overhead of host-device communication resulting from the limited GPU memory, the in-memory throughput is on par with the performance observed

Table 3: Comparison of the memory related metrics between BLCO and MM-CSF for MTTKRP on the A100 GPU.

Data Set	Format	*n*	Vol[1]	TP[2]	Data Set	Format	*n*	Vol[1]	TP[2]
Uber	BLCO	1	2.78	3.60	Enron	BLCO	1	44.82	4.11
		2	2.75	3.61			2	46.23	4.62
		3	2.75	3.53			3	47.88	4.92
		4	2.73	2.77			4	47.22	4.70
	MM-CSF	1	1.68	1.68		MM-CSF	1	41.39	0.31
		2	1.33	2.03			2	62.83	3.16
		3	1.33	1.93			3	37.15	2.29
		4	2.12	0.32			4	37.05	3.01
Vast-2015	BLCO	1	16.91	3.92	NELL-1	BLCO	1	107.5	2.44
		2	16.73	3.77			2	104.5	2.32
		3	13.92	2.90			3	110.7	2.39
	MM-CSF	1	9.19	1.19		MM-CSF	1	123.1	2.21
		2	8.36	1.57			2	118.5	2.19
		3	8.36	1.45			3	122.1	0.86

[1] Memory volume in GB, measured by *l1tex__t_bytes.sum* in Nsight Compute [3]
[2] Memory throughput in TB/s, calculated by (Vol / total execution time)

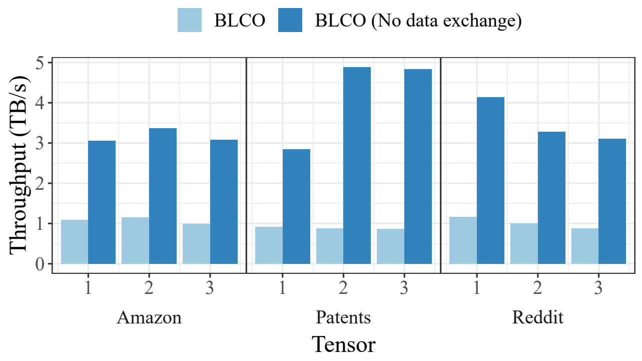

Figure 10: The memory throughput of BLCO-based MTTKRP (with and without host-device data exchange) for out-of-memory tensors across every mode on the A100 GPU.

for the tensors that can fit in the GPU (c.f. Table 3). However, the overall throughput is lower due to the limited bandwidth of the host-device interconnect compared to the device memory bandwidth. While BLCO achieves perfect overlap between host-device transfers and MTTKRP computations, the communication overhead still dominates the execution time, leading to lower overall throughput (57%–75% of the memory bandwidth). This result demonstrates that our framework handles out-of-memory tensors as efficiently as in-memory tensors, but the overall performance for out-of-memory tensors is limited by the bandwidth of the host-device interconnect.

6.5 Format Construction

Figure 11 shows the generation time of the GPU sparse formats, namely, BLCO, GenTen, and MM-CSF, as well as the CPU-based ALTO format. Furthermore, Figure 12 details the run-time distribution across the different format generation stages of BLCO. The tensor formats are generated from raw data, in the COO representation, on the host CPU listed in Table 1. By adopting a mode-agnostic linearized form rather than a multi-dimensional representation, BLCO significantly decreases the format generation cost, which is typically dominated by sorting and clustering non-zero elements. Additionally, BLCO does not require extra mode-specific mapping or scheduling information for reducing synchronization. As a result, BLCO is several times (up to 13.6×) cheaper to generate than the state-of-the-art MM-CSF format. On the A100 GPU, BLCO needs approximately 12 full (all-mode) MTTKRP iterations on average to amortize its format generation time, while the other GPU formats require up to an order of magnitude more iterations to amortize their construction cost.

Compared to ALTO, BLCO enables efficient execution of tensor operations on massively parallel architectures by (i) re-encoding linearized indices for fast decoding on GPUs and (ii) blocking the tensor into smaller chunks that fit in limited device memory and require at most 64 bits for indices. However, these additional stages typically consume less than 25% of the overall format construction cost, as shown in Figure 12.

7 RELATED WORK

Optimizing sparse tensor decomposition and MTTKRP operations has been the subject of several prior studies, which propose various sparse tensor formats along with parallel algorithms to process and analyze the multi-modal data on CPU- and GPU-based hardware architectures. List-based formats, such as F-COO [31], GenTen [40], and TB-COO [12], explicitly store the multi-dimensional coordinates of each non-zero element. To reduce atomic operations, these formats keep multiple mode-specific copies of the tensor and/or extra scheduling information, which substantially increases their memory footprint. Tree-based formats, including CSF [47, 49], B-CSF [38], and MM-CSF [35], extend the compressed sparse row (CSR) matrix format to higher-order tensors. While these mode-specific formats can compress the sparse data, they have load imbalance issues and significant performance variation across various modes of execution.

Block-based formats (e.g., HiCOO [28]) cluster non-zero elements to compress the sparse tensor and to exploit data locality. However, as the number of tensor modes and sparsity increase, the majority of HiCOO blocks consist of a few non-zero elements, leading to more memory usage than COO [17, 28]. In addition, the irregular spatial distributions of sparse data result in severe load imbalance and synchronization issues across HiCOO blocks [17, 28], and as a result, it has no GPU implementation [27]. In contrast, BLCO addresses these limitations by generating coarse-grained blocks that fit in the GPU memory, while efficiently utilizing the GPU resources by encoding the non-zero elements within each block in a fine-grained linearized form, amenable to caching and parallel execution.

CPU-based tensor linearization approaches, such as the LCO [15] and ALTO [17] formats, compress the tensor by mapping the multi-dimensional coordinates of a non-zero element into a single index. In particular, ALTO enables high-performance tensor operations by (i) leveraging the efficient bit-level scatter/gather instructions and large integers on CPUs, and (ii) using adaptive atomic- and reduction-based conflict resolution algorithms, tailored for architectures with coarse-grained cores/threads. However, massively parallel GPUs lack native support for the bit manipulation instructions and large integer arithmetic that are needed for efficient processing of prior linearized formats. Additionally, GPUs suffer from limited device memory and require sophisticated conflict resolution, due to their massive fine-grained parallelism and substantial synchronization overhead. BLCO directly addresses these issues to allow efficient execution of large-scale tensors on accelerators.

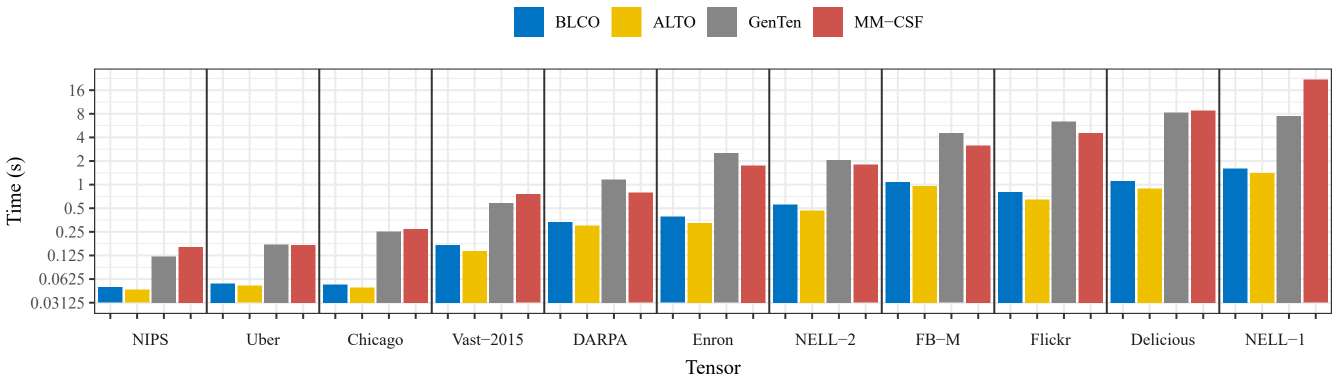

Figure 11: Comparison of the BLCO construction/generation cost against popular GPU formats as well as the CPU-based ALTO format on the data sets that fit in GPU memory.

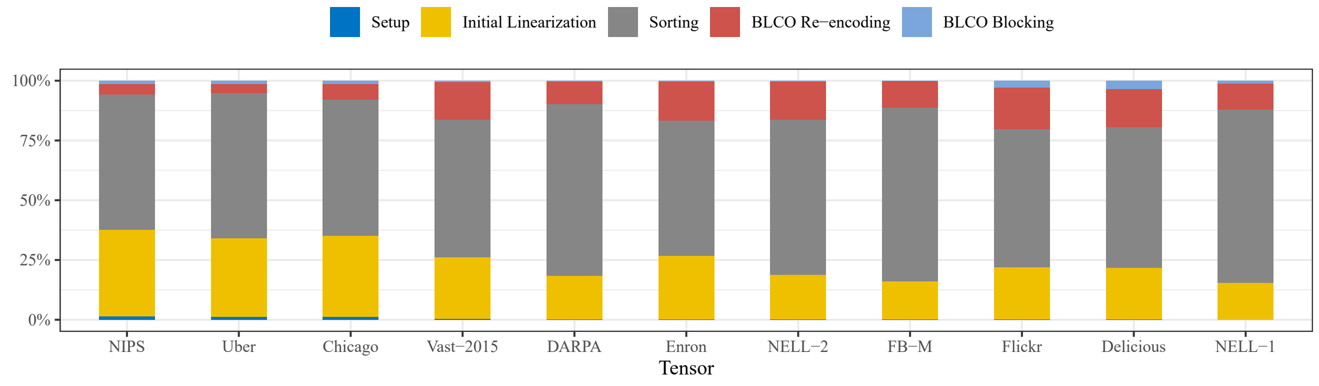

Figure 12: Breakdown of the BLCO construction cost for the data sets that fit in GPU memory. Compared to ALTO, BLCO requires additional blocking and re-encoding to enable efficient execution on GPUs.

Segmented scan and reduction [44, 54] have been used to reduce the synchronization cost of sparse workloads [6, 12, 30, 53, 55] on parallel architectures. Prior studies apply these primitives to *mode-specific* formats with delineated and/or sorted groups of non-zero elements according to the target mode. In contrast, we devise an opportunistic algorithm that leverages the *mode-agnostic* BLCO format to reduce synchronization by discovering conflicts and performing segmented scan on-the-fly and without needing sorted non-zero elements or keeping extra scheduling information. In addition, our novel conflict resolution algorithm eliminates control-flow and memory-access irregularities by specializing threads to perform different operations at each execution phase.

The TACO compiler [24] automatically generates various sparse matrix and tensor algebra kernels, including sparse MTTKRP. However, prior work showed that hand-optimized implementations of CSF-based formats (namely, B-CSF) still outperform the auto-generated TACO code on GPU architectures, even with extensive auto-scheduling and optimization [43].

A wealth of work perform MTTKRP computations on distributed-memory platforms using MPI [11, 22, 45, 48, 50], or the MapReduce [7, 21] framework. Other studies [51, 52, 56] explore format selection based on machine learning models to efficiently leverage existing sparse formats.

8 CONCLUSION AND FUTURE WORK

To enable high-performance sparse tensor decomposition on massively parallel GPU architectures, this work proposes the BLCO format. In contrast to prior approaches, which have been restricted to in-memory tensors, BLCO allows efficient processing of both in-memory and out-of-memory tensors. By discovering and merging conflicting updates on-the-fly, without any mode-specific information or ordering of non-zero elements, our BLCO-based MTTKRP demonstrated substantial speedup (up to 33.35×) over the state-of-the-art MM-CSF. Our future work will explore heterogeneous distributed-memory systems as well as other tensor algorithms.

ACKNOWLEDGMENTS

The authors would like to thank Intel colleagues, John Pennycook and Nitin Dhamankar, for their help and valuable feedback.

REFERENCES

[1] 2021. *NVIDIA A100 80GB PCIe GPU.* Technical Report. Nvidia Corp.
[2] 2022. Intel DPC++ Compatibility Tool. https://software.intel.com/en-us/get-started-with-intel-dpcpp-compatibility-tool. Online; accessed 14 May 2022.
[3] 2022. Nsight Compute Command Line Interface. https://docs.nvidia.com/nsight-compute/pdf/NsightComputeCli.pdf. Online; accessed 14 May 2022.
[4] 2022. Nsight Systems User Guide. https://docs.nvidia.com/nsight-systems/pdf/UserGuide.pdf. Online; accessed 14 May 2022.

[5] 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

[6] Nathan Bell and Michael Garland. 2009. Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors. In *Proceedings of the conference on high performance computing networking, storage and analysis*. 1–11.

[7] Zachary Blanco, Bangtian Liu, and Maryam Mehri Dehnavi. 2018. CSTF: Large-Scale Sparse Tensor Factorizations on Distributed Platforms. In *Proceedings of the 47th International Conference on Parallel Processing*. ACM, Eugene OR USA, 1–10. https://doi.org/10.1145/3225058.3225133

[8] David Blythe. 2020. The Xe GPU Architecture. In *2020 IEEE Hot Chips 32 Symposium (HCS)*. IEEE Computer Society, 1–27.

[9] David Bruns-Smith, Muthu M. Baskaran, James Ezick, Tom Henretty, and Richard Lethin. 2016. Cyber Security through Multidimensional Data Decompositions. In *2016 Cybersecurity Symposium (CYBERSEC)*. 59–67. https://doi.org/10.1109/CYBERSEC.2016.017

[10] Jee Choi, Xing Liu, Shaden Smith, and Tyler 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] Ming Dun, Yunchun Li, Hailong Yang, Qingxiao Sun, Zhongzhi Luan, and Depei Qian. 2021. An Optimized Tensor Completion Library for Multiple GPUs *(ICS '21)*. Association for Computing Machinery, New York, NY, USA, 417–430. https://doi.org/10.1145/3447818.3460692

[13] Hadi Fanaee-T and João Gama. 2016. Tensor-based anomaly detection: An interdisciplinary survey. *Knowl. Based Syst.* 98 (2016), 130–147. https://doi.org/10.1016/j.knosys.2016.01.027

[14] Sofia Fernandes, Hadi Fanaee-T, and João Gama. 2019. Evolving Social Networks Analysis via Tensor Decompositions: From Global Event Detection Towards Local Pattern Discovery and Specification. In *Discovery Science*, Petra Kralj Novak, Tomislav Šmuc, and Sašo Džeroski (Eds.). Springer International Publishing, Cham, 385–395.

[15] 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

[16] Huan He, Jette Henderson, and Joyce C Ho. 2019. Distributed Tensor Decomposition for Large Scale Health Analytics. In *The World Wide Web Conference* (San Francisco, CA, USA) *(WWW '19)*. Association for Computing Machinery, New York, NY, USA, 659–669. https://doi.org/10.1145/3308558.3313548

[17] 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 *Proceedings of the ACM International Conference on Supercomputing* (Virtual Event, USA) *(ICS '21)*. Association for Computing Machinery, New York, NY, USA, 404–416. https://doi.org/10.1145/3447818.3461703

[18] 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

[19] 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

[20] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P Scarpazza. 2018. Dissecting the NVIDIA volta GPU architecture via microbenchmarking. *arXiv preprint arXiv:1804.06826* (2018).

[21] 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

[22] Oguz Kaya and Bora Uçar. 2015. Scalable Sparse Tensor Decompositions in Distributed Memory Systems. In *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–11. https://doi.org/10.1145/2807591.2807624

[23] David B Kirk. 2006. NVIDIA CUDA Software and GPU Parallel Computing Architecture. (2006).

[24] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The Tensor Algebra Compiler. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–29.

[25] 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

[26] Tamara G. Kolda and Jimeng 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

[27] Jiajia Li, Yuchen Ma, and Richard Vuduc. 2018. ParTI! : A Parallel Tensor Infrastructure for Multicore CPUs and GPUs. http://parti-project.org Last updated: Jan 2020.

[28] 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

[29] Jiajia Li, Bora Uçar, 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

[30] Bangtian Liu, Chengyao Wen, Anand D. Sarwate, and Maryam Mehri 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

[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] 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

[33] Xinxin Mei and Xiaowen Chu. 2016. Dissecting GPU Memory Hierarchy through Microbenchmarking. *IEEE Transactions on Parallel and Distributed Systems* 28, 1 (2016), 72–86.

[34] 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

[35] 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

[36] 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

[37] Israt Nisa, Jiajia Li, Aravind Sukumaran-Rajam, Richard 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

[38] 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

[39] Evangelos E. Papalexakis, Christos Faloutsos, and Nicholas D. Sidiropoulos. 2016. Tensors for Data Mining and Data Fusion: Models, Applications, and Scalable Algorithms. 8, 2, Article 16 (Oct. 2016), 44 pages. https://doi.org/10.1145/2915921

[40] 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

[41] James Reinders, Ben Ashbaugh, James Brodman, Michael Kinsner, John Pennycook, and Xinmin Tian. 2021. *Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL*. Springer Nature.

[42] Achim Rettinger, Hendrik Wermser, Yi Huang, and Volker Tresp. 2012. Context-Aware Tensor Decomposition for Relation Prediction in Social Networks. *Social Network Analysis and Mining* 2, 4 (2012), 373–385. https://doi.org/10.1007/s13278-012-0069-5

[43] Ryan Senanayake, Changwan Hong, Ziheng Wang, Amalee Wilson, Stephen Chou, Shoaib Kamil, Saman Amarasinghe, and Fredrik Kjolstad. 2020. A Sparse Iteration Space Transformation Framework for Sparse Tensor Algebra. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–30.

[44] Shubhabrata Sengupta, Mark Harris, Michael Garland, et al. 2008. Efficient Parallel Scan Algorithms for GPUs. *NVIDIA, Santa Clara, CA, Tech. Rep. NVR-2008-003* 1, 1 (2008), 1–17.

[45] Kijung Shin and U Kang. 2014. Distributed Methods for High-Dimensional and Large-Scale Tensor Factorization. In *2014 IEEE International Conference on Data Mining*. IEEE, 989–994.

[46] 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/

[47] 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) *(IA^3 '15)*. Association for Computing Machinery, New York, NY, USA, Article 5, 7 pages. https://doi.org/10.1145/2833179.2833183

[48] Shaden Smith and George Karypis. 2016. A Medium-Grained Algorithm for Sparse Tensor Factorization. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 902–911. https://doi.org/10.1109/IPDPS.2016.113

[49] Shaden Smith, Niranjay Ravindran, Nicholas D. Sidiropoulos, and George Karypis. 2015. SPLATT: Efficient and Parallel Sparse Tensor-Matrix Multiplication. In *2015 IEEE International Parallel and Distributed Processing Symposium*. 61–70. https://doi.org/10.1109/IPDPS.2015.27

[50] Edgar Solomonik, Devin Matthews, Jeff Hammond, and James Demmel. 2013. Cyclops Tensor Framework: Reducing Communication and Eliminating Load Imbalance in Massively Parallel Contractions. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE, 813–824.

[51] 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

[52] 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

[53] Shengen Yan, Chao Li, Yunquan Zhang, and Huiyang Zhou. 2014. YaSpMV: Yet Another SpMV Framework on GPUs. In *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 107–118.

[54] Shengen Yan, Guoping Long, and Yunquan Zhang. 2013. StreamScan: Fast Scan Algorithms for GPUs Without Global Barrier Synchronization. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 229–238.

[55] Yao Zhang, Jonathan Cohen, and John D Owens. 2010. Fast Tridiagonal Solvers on the GPU. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 127–136.

[56] 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