A Unified Optimization Approach for Sparse Tensor Operations on GPUs

Bangtian Liu*, Chengyao Wen*, Anand D. Sarwate*, Maryam Mehri Dehnavi*
*Rutgers, The State University of New Jersey
{bangtian.liu, chengyao.wen, anand.sarwate, maryam.mehri}@rutgers.edu

***Abstract*—Sparse tensors appear in many large-scale applications with multidimensional and sparse data. While multidimensional sparse data often need to be processed on manycore processors, attempts to develop highly-optimized GPU-based implementations of sparse tensor operations are rare. The irregular computation patterns and sparsity structures as well as the large memory footprints of sparse tensor operations make such implementations challenging. We leverage the fact that sparse tensor operations share similar computation patterns to propose a unified tensor representation called F-COO. Combined with GPU-specific optimizations, F-COO provides highly-optimized implementations of sparse tensor computations on GPUs. The performance of the proposed unified approach is demonstrated for tensor-based kernels such as the Sparse Matricized Tensor-Times-Khatri-Rao Product (SpMTTKRP) and the Sparse Tensor-Times-Matrix Multiply (SpTTM) that are used in tensor decomposition algorithms. Compared to state-of-the-art work we improve the performance of SpTTM and SpMTTKRP up to 3.7 and 30.6 times respectively on NVIDIA Titan-X GPUs. We implement the CANDECOMP/PARAFAC (CP) decomposition and achieve up to 14.9 times speedup using the unified method over state-of-the-art libraries on NVIDIA Titan-X GPUs.**

I. Introduction

Tensors, defined as multi-dimensional or N-way arrays, represent multidimensional data naturally. Tensor-based computations or *multilinear algebraic* methods such as tensor decomposition(s) appear widely in a variety of fields including machine learning [1], [2], data mining [3], [4], computer vision [5], [6], recommender systems [7], and quantum chemistry [8]. A number of industry-initiated frameworks for deep learning such as TensorFlow [9] and Torch [10] also use tensor representations. Tensor operations are essential building blocks and tend to be the determinant operations for the performance of tensor algorithms and applications. In many applications, the tensors are sparse, that is, most of their elements are zeros. Thus, developing parallel algorithms and libraries that accelerate sparse tensor computations on modern architecture is essential.

Previous work has optimized sparse tensor operations on different hardware platforms including shared memory systems [11], [12], [13], distributed systems with MapReduce [3], [14] and on distributed memory with MPI [15], [16], [17]. Due to their embarrassingly parallel execution model, GPUs are good candidates to accelerate sparse tensor computations; however, using GPUs is challenging because of the inherently irregular computation patterns in sparse tensor algebra. To our knowledge, Parallel Tensor Infrastructure (ParTI [18]) is the only work that accelerates tensor operations on GPUs. The optimizations in ParTI are not memory efficient, lead to load imbalance, and are sensitive to mode changes and increase with the tensor computation rank.

Previous work on tensor computations, optimize tensor operations independently and thus use different approaches to accelerate sparse tensor operations. For example, the work in [13], [18], [16] optimizes the sparse tensor-times-dense matrix (SpTTM) operation while others [11], [12], [15], [17], [18], [3] mainly focus on the sparse Matricized Tensor Times Khatri-Rao Product (SpMTTKRP). The type of optimizations and the order in which they are applied are often shared between different sparse tensor operations. By investigating the underlying computation patterns and computation orders in sparse tensor operations, we propose an approach to generalize sparse tensor representations. Our unified storage format and parallel algorithms can be used across many sparse tensor operations and can be extended to high-order tensor computations.

Numerous challenges exist in optimizing sparse tensor operations on GPUs, such as *i)* finding a good parallelization granularity, *ii)* reducing storage costs and irregularities in memory accesses, and *iii)* dealing with atomic updates. Prior work has used fiber- or slice-level computations as the granularity for parallelization. However, such approaches lead to noticeable load imbalance between threads on the GPU because of the sparse nature of the tensors. Also, the optimizations do not deliver consistent performance across different modes and ranks. The large intermediate data created during the sparse tensor computations is also very expensive to store on GPUs. Finally many of the sparse tensor operations require atomic updates that are expensive to perform on GPUs. We propose a unified optimization method for sparse tensor operations to address these challenges on GPUs. Our major contributions are as follows:

1) **F-COO: A unified storage format for sparse tensors**. We propose a new storage format that is based on the tensor modes for sparse tensor computations. F-COO is memory efficient compared to other sparse tensor storage formats and can be used as a unified storage format across different sparse tensor operations.
2) **Unified parallel algorithms for sparse tensor operations**. F-COO is used to implement parallel algorithms

2168-9253/17 $31.00 © 2017 IEEE
DOI 10.1109/CLUSTER.2017.75

and optimizations for sparse tensor operations on GPUs. We demonstrate how optimizations of sparse tensor operations such as SpMTTKRP and SpTTM that have been treated differently in previous tensor literature are inherently the same. Our unified parallel algorithms are used across different tensor operations, are not sensitive to mode changes, and scale well with increases in the tensor computation rank.

3) **GPU-specific optimizations**. By using the flag arrays in F-COO, we enable the application of efficient algorithms commonly used in sparse matrix literature such as the segmented scan method without unfolding the tensor. Other optimizations such as kernel fusion, warp shuffle, and data reuse are also enabled in our unified optimizations.
4) **Significant speedups on real datasets for SpTTM, SpMTTKRP, and CP decomposition.** The proposed unified approach leads to 3.7× speedup for SpTTM and 30.6× speedup for SpMTTKRP for tested benchmarks over state-of-art work on GPU platforms. The CP decomposition is accelerated up to 14.9× times compared to state-of-the-art libraries.

II. Background

Tensor notation: A tensor is a multi-way array. The *order* of a tensor refers to the number of dimensions, also called *modes*. Vectors (first-order tensors) and matrices (second-order tensors) are presented by boldface lowercase and boldface capital letters respectively. We generally use calligraphic letters for higher-order tensors (e.g., $\mathcal{X}$). The scalar element at position (i, j, k) of a third-order tensor $\mathcal{X}$ is shown as $\mathcal{X}(i, j, k)$. We also use the colon notation from MATLAB (as does SPLATT [11]), in which a colon in the place of an index represents all members of that mode. For example, $\mathbf{A}(m, :)$ is the m-th row of the matrix $\mathbf{A}$.

A *fiber* is a one-dimensional segment of a tensor along one of the modes. A third-order tensor $\mathcal{X}$ has three kinds of fibers on three different modes represented by $\mathcal{X}(:, j, k)$, $\mathcal{X}(i, :, k)$, and $\mathcal{X}(i, j, :)$. *Slices* are two dimensional segments of a tensor, obtained by fixing all indices except for two. A third-order tensor $\mathcal{X}$ also has three kinds of slices, written as $\mathcal{X}(i, :, :)$, $\mathcal{X}(:, j, :)$, and $\mathcal{X}(:, :, k)$.

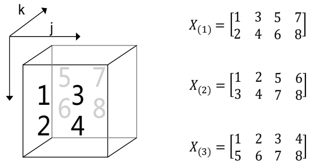

k
j
i
1
2
3
4
5
6
7
8
$X_{(1)} = \begin{bmatrix} 1 & 3 & 5 & 7 \\ 2 & 4 & 6 & 8 \end{bmatrix}$
$X_{(2)} = \begin{bmatrix} 1 & 2 & 5 & 6 \\ 3 & 4 & 7 & 8 \end{bmatrix}$
$X_{(3)} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{bmatrix}$

Figure 1: The matricization of a $(2 \times 2 \times 2)$ tensor.

Matricization, also called *unfolding* or *flattening*, transforms a tensor into a matrix. The result of mode-n matricization $\mathbf{X}_{(n)}$ is a matrix which it's columns are mode-n fibers of the tensor $\mathcal{X}$ meaning that mode-n fibers in tensor $\mathcal{X}$ become the columns of the resulting matrix. Given a tensor $\mathcal{X}$ of size $I \times J \times K$, $\mathbf{X}_{(1)}$ is of size $I \times JK$. Figure 1 illustrates how to unfold a $(2{\times}2{\times}2)$ tensor along each of the three modes. The *Kronecker product* of matrices $\mathbf{A} \in \mathbb{R}^{I \times J}$ and $\mathbf{B} \in \mathbb{R}^{K \times L}$ is represented by $\mathbf{A} \otimes \mathbf{B}$, which generates a matrix of size $IK \times JL$. The Kronecker product is defined as

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{11}\mathbf{B} & a_{12}\mathbf{B} & \dots & a_{1J}\mathbf{B} \\ a_{21}\mathbf{B} & a_{22}\mathbf{B} & \dots & a_{2J}\mathbf{B} \\ \vdots & \vdots & \ddots & \vdots \\ a_{I1}\mathbf{B} & a_{I2}\mathbf{B} & \dots & a_{IJ}\mathbf{B} \end{bmatrix}. \quad (1)$$

The *Khatri-Rao product* of $\mathbf{A}$ and $\mathbf{B}$, also known as the *column-wise Kronecker product*, is written as $\mathbf{A} \odot \mathbf{B}$. If $\mathbf{A} \in \mathbb{R}^{I \times K}$ and $\mathbf{B} \in \mathbb{R}^{J \times K}$, the resulting matrix of size $IJ \times K$ is defined by

$$\mathbf{A} \odot \mathbf{B} = \begin{bmatrix} a_1 \otimes b_1 & a_2 \otimes b_2 & \dots & a_k \otimes b_k \end{bmatrix}. \quad (2)$$

Tensor-Times-Matrix: Tensor-Times-Matrix (TTM) on mode n, also called the n-mode product, is a product of multiplying the tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \ldots \times I_n \times \ldots \times I_N}$ by the matrix $\mathbf{U} \in \mathbb{R}^{I_n \times R}$ along the n-th dimension; represented by $\mathcal{Y} = \mathcal{X} \times_n U$. For third-order tensors, TTM on mode-3 is represented by:

$$\mathcal{Y}(i, j, :) = \sum_{k=1}^{K} \mathcal{X}(i, j, k)\mathbf{U}(k, :). \quad (3)$$

When $\mathcal{X}$ is sparse, $\mathbf{U}$ is dense and the operation is called SpTTM. In this case, the resulting tensor $\mathcal{Y}$ is only *semi-sparse* since each fiber at index (i, j) becomes dense and the length of the fiber will be equal to the number of columns of the dense matrix $\mathbf{U}$. SpTTM can be seen as a high dimensional generalization of the sparse matrix-vector multiply (SpMV) operation. SpTTM to tensor-based computation is what SpMV is to matrix-based computation. Similarly, SpTTM is usually the building block and bottleneck operation in tensor-based computations.

As demonstrated in [16], [19], the key operation in the ALS-based Tucker decomposition algorithm, namely Higher Order Orthogonal Iteration (HOOI), is called the tensor times matrix-chain (TTMc) product. For an N-order tensor, TTMc on mode-n indicates tensor times matrix (TTM) products with $N - 1$ different matrices along the corresponding modes other than mode-n. For example, a typical mode-1 TTMc in Tucker decomposition for a 3-order tensor is $\mathcal{Y} = \mathcal{X} \times_2 \mathbf{U}_2 \times_3 \mathbf{U}_3$. Previous work on Tucker decomposition provides a high-performance parallel algorithm and implementation of TTMc [20], [16]. Equation (4) shows TTMc based on the coordinate storage format:

$$\mathbf{Y}_{(1)} = \sum_{\mathcal{X}(i,j,k) \in \mathcal{X}} \mathcal{X}(i, j, k)(\mathbf{U}_2(j, :) \otimes \mathbf{U}_3(k, :)) \quad (4)$$

More information about TTMc can be found Kaya et al. [16].

Matricized Tensor Times Khatri-Rao Product: MTTKRP is an important sparse tensor operation and is the main computation bottleneck in the CP decomposition algorithm. Parallel implementations of CP mainly focus on accelerating the execution of MTTKRP. Equation (5) represents MTTKRP operations along the first tensor mode. It unfolds the tensor along the first mode and then multiplies it with the Khatri-Rao product of the corresponding matrices $\mathbf{B}$ and $\mathbf{C}$:

$$M = \mathbf{X}_{(1)}(\mathbf{C} \odot \mathbf{B}). \tag{5}$$

For large-scale sparse tensors, $\mathbf{C} \odot \mathbf{B}$ cannot be explicitly computed since the resulting matrix is dense and has a size of $JK \times R$ which can consume more memory than the sparse tensor $\mathcal{X}$ itself in most cases. Therefore, previous research on SpMTTKRP focuses on how to map SpMTTKRP to other less-costly operations based on the sparsity pattern of the tensor to avoid computing the Khatri-Rao product explicitly [3], [11]. The SpMTTKRP for the first mode can be written as follows:

$$\begin{aligned}
\mathbf{M}(i,r) &= \sum_{z=1}^{JK} \mathbf{X}_{(1)}(i,z)(\mathbf{B}(z\%J,r)\mathbf{C}(z/J,r)) \\
\mathbf{M}(i,:) &= \sum_{z=1}^{JK} \mathbf{X}_{(1)}(i,z)(\mathbf{B}(z\%J,:) * \mathbf{C}(z/J,:)) \\
&= \sum_{k=1}^{K}\sum_{j=1}^{J} \mathcal{X}(i,j,k)(\mathbf{B}(j,:) * \mathbf{C}(k,:)),
\end{aligned} \tag{6}$$

where $\%$ is the modulus operation. While $\mathcal{X}$ is sparse, the result matrix $\mathcal{M}$ is a dense matrix where two product modes are replaced with the column dimensions of the dense matrices $\mathbf{B}$ and $\mathbf{C}$. The index mode i is also dense due to the fact that a sparse tensor can not have empty slices in the i-dimension.

Tensor Decomposition: The CANDECOMP/PARAFAC (CP) Decomposition [19] factorizes a tensor into a sum of component rank-one tensors. The most popular algorithm for fitting the CP decomposition is based on the Alternating Least Squares (ALS) method. The ALS method iterates through the modes of the tensor, updating a factor matrix for each mode while holding the other factors constant. The algorithm for a 3-way tensor is given in Algorithm 1.

Algorithm 1 CP-ALS for a 3-way tensor

```
Input: $\mathcal{X}$: A 3rd order tensor R: The rank of approximation
Output: CP decomposition $[\boldsymbol{\lambda}; \mathbf{A}, \mathbf{B}, \mathbf{C}]$
repeat
    $\mathbf{A} \leftarrow \mathbf{X}_{(1)}(\mathbf{C} \odot \mathbf{B})(\mathbf{B}^\top\mathbf{B} * \mathbf{C}^\top\mathbf{C})^\dagger$
    Normalize columns of $\mathbf{A}$
    $\mathbf{B} \leftarrow \mathbf{X}_{(2)}(\mathbf{C} \odot \mathbf{A})(\mathbf{A}^\top\mathbf{A} * \mathbf{C}^\top\mathbf{C})^\dagger$
    Normalize columns of $\mathbf{B}$
    $\mathbf{C} \leftarrow \mathbf{X}_{(3)}(\mathbf{B} \odot \mathbf{A})(\mathbf{A}^\top\mathbf{A} * \mathbf{B}^\top\mathbf{B})^\dagger$
    Normalize columns of $\mathbf{C}$ and store the norms as $\boldsymbol{\lambda}$
until no improvement or maximum iterations reached
```

III. Related Work

Sparse tensor operations such as SpTTM [13], [16] and SpMTTKRP [11], [12], [15] have been implemented as standalone routines to improve the performance of tensor algorithms and applications. Most of these work first propose or choose a tensor format and then propose parallel algorithms that operate on these formats efficiently. Therefore, our survey of previous work is based on storage formats for sparse tensors and parallel algorithms on storage formats.

A. Storage Formats for Sparse Tensors

While some parallel algorithms for sparse tensor operations are directly based on the coordinate format (COO) for SpMTTKRP [17] and SpTTM [16], a number of novel storage formats have been proposed to reduce floating point computations and exploit more parallelism; the *compressed data fiber* (CSF) format [12] is an example used in SpMTTKRP. These storage formats are proposed for distributed memory or shared memory systems. Dfacto [15] and SPLATT [11] unfold a tensor along one mode to reduce floating point operations at the cost of increased memory usage. As pointed in [17], unfolding tensors requires column index values up to $\prod_{k\neq i}^{N} I_k$, which easily exceeds integer value limits when the input tensor is large in each mode. CSF is a tree-based data structure used in SPLATT that enables the extension of an efficient implementation of SpMTTKRP to higher dimensions. For GPU implementations of SpTTM on GPUs [13], Li *et al.* proposes the semi-COO (sCOO) format which stores a semi-sparse tensor, which is the output of SpTTM; sCOO does not store indices for the dense modes in the semi-sparse tensor.

The objective of sparse tensor storage formats is to facilitate parallel implementations of tensor computations on shared and distributed memory systems. However, they often can not directly be used on GPU platforms since they will lead to significant overheads. For example, the *compressed data fiber* (CSF) [12], extends the compressed row storage format (CSR) to sparse tensors and is a fiber-centric and tree-based storage format. The recursive algorithms used in CSF-based optimization methods are not a good fit for GPU architectures. Also, the general storage format COO causes too many atomic operations when non-zeros processed by different threads share identical indices. The sCOO format stores the output of SpTTM which is a semi-sparse tensor [13]. Since sCOO is designed to store the output of a tensor operation, it can not be used to control the type of algorithms and optimizations used to implement the tensor operation.

B. Parallel Sparse Tensor Algorithms and Implementations

Sparse tensor operations have been parallelized on different types of processing platforms such as shared-memory, distributed-memory, and GPUs.

Shared memory systems: The Tensor Toolbox [21], [4] and N-way Toolbox [22] are two widely used MATLAB toolboxes for tensor operations on share memory systems. The Cyclops Tensor Framework (CTF) [23] is a C++ library which provides automatic parallelization of sparse tensor operations.

CTF transforms sparse tensors to matrices via unfolding and can only store the SpTTM output as a dense tensor. This restriction significantly reduces its efficiency. SPLATT [11] is a library used for parallelizing the CP decomposition on shared-memory systems. It proposes a compressed and fiber-centric data structure for sparse tensors called *compressed data fiber* (CSF). Based on the CSF data structure, hypergraph models and multi-partite graphs are used to partition non-zeros into semi-sparse regions and improve data locality for SpMTTKRP.

Distributed memory systems: Many parallel algorithms have been proposed for large-scale tensor operations on distributed-memory systems. Gigatensor [3] handles tera-scale tensors using the MapReduce framework. Gigatensor is the first work that minimizes the intermediate data sizes in SpMTTKRP and the number of floating operations for large-scale tensor operations. Dfacto [15] also provides a distributed tensor decomposition implementation. However, the performance of Dfacto is limited by high memory footprints and data communication overhead since it needs to transform a tensor $\mathcal{X}$ into three matrices $\mathbf{X}_{(1)}$, $\mathbf{X}_{(2)}$, $\mathbf{X}_{(3)}$ along three modes before operating on its data. Hypertensor [17] is a sparse tensor library for SpMTTKRP on distributed-memory environments. Hypergraphs are used to partition the non-zero elements in a tensor and thus improve load balance and reduce data communication in sparse tensor operations on distributed memory environments [17], [16].

GPU: Li et al. [13] propose a parallel algorithm and implementation of SpTTM on GPUs via parallelizing the algorithm on fibers. Since fibers in a sparse tensor may have different sizes their proposed implementation suffers from load imbalance and leads to warp divergence on GPU platforms for some sparse tensors. They also implement the SpMTTKRP algorithm on GPUs in ParTI [18] where data partitions are created based on the non-zeros of a tensor. The performance of their algorithm is limited by the overhead of atomic operations when updating divided slices via multiple threads.

IV. A unified optimization method for sparse tensor Operations on GPUs

We propose a unified approach for the storage and optimization of tensor operations. We generalize some of the mode notations used in previous literature to categorize the computation patterns and structures in sparse tensor operations. The modes are then encoded into a novel sparse storage format that we call as F-COO (flagged-coordinate). F-COO can be used as a unified format across different tensor operations. We show how the mode encoding in F-COO allows our proposed parallel algorithms to operate on tensor non-zeros directly, eliminating the need to store intermediate data. This shows how a unified approach enables a one-shot approach to computing tensor operations such as SpMTTKRP. F-COO also enables the application of the segmented scan algorithm, a highly efficient algorithm used in sparse matrix computations, without the need to unfold the tensor into a matrix. In this section we describe our generalization of tensor modes, introduce F-COO, and show how this unified approach applies to parallel algorithms.

A. Unified Form of Sparse Tensor Modes

The operations and computations in tensor methods can be characterized using a number of mode notations originally introduced by Baskran *et al.* [24] and Li *et al.* [13] for SpTTM. In the following we extend these notations to other sparse tensor operations such as SpMTTKRP which enables us to propose a unified storage format and optimization method for sparse tensor operations:

- *Product modes*: are defined as the modes in which a tensor gets multiplied by a matrix. Mode-3 in Equation (3) for TTM and mode-(2,3) in Equation (5) for MTTKRP are the product modes.
- *Index modes*: are all modes except for the product mode such as mode-(1,2) in Equation (3) for TTM and mode-1 in Equation (5) for MTTKRP.
- *Sparse mode*: is when at least one non-empty fiber in this mode is sparse. For example, if at least one of the fibers in $\mathcal{X}(i,j,:)$ is sparse mode-3 will be a sparse mode.
- *Dense mode*: is when all fibers in the mode are dense vectors. For example, if the fibers in $\mathcal{X}(i,j,:)$ are all dense then mode-3 is in a dense mode.

B. The F-COO Storage Format

This section discusses our proposed F-COO storage format which: *i)* encodes changes in tensor modes and thus can be extended to support different sparse tensor operations; *ii)* eliminates the need for tensor unfolding while enabling the application of efficient sparse matrix algorithms such as segmented scan; *iii)* requires less storage compared to formats used in previous tensor literature. F-COO follows a similar storage approach to the COO format where all non-zeros of the tensor are stored with their corresponding indices and values. However, to enable unified computations for sparse tensor operations, the tensor modes discussed in the previous subsection are encoded into F-COO. As a result of this encoding, F-COO captures changes in the computation pattern during the sparse tensor operations such as switching to a new fiber/slice or changing from a dense operation to a sparse mode.

Table I shows our classification of tensor modes for different operations and Figure 2 demonstrates how this classification is used to store the tensor for the SpTTM and SpMTTKRP operations. The val vector stores the non-zero values of the tensor. Except for the flag arrays, all other vectors such as i, j, k are used to store the indices corresponding to the product mode. The indices in F-COO that correspond to the product mode are used to guide the Kronecker or Hadamard product operations. F-COO also uses two flag arrays, i.e. the bit-flag (bf) and the start-flag (sf). The bf array is used to represent any changes in the index modes which consequently shows the computation has switched to another fiber (in SpTTM) or to another slice (in SpMTTKRP). F-COO also comes with a start-flag (sf) that is used to indicate whether a new fiber or slice

Operations	Equation	Product mode	Index mode	Sparse mode of result	Dense mode of result
SpTTM on mode-3	$\mathcal{Y}(i,j,:) \mathrel{+}= \mathcal{X}(i,j,k)\mathbf{U}(k,:)$	mode-3	mode-(1,2)	mode-(1,2)	mode-3
SpMTTKRP on mode-1	$\mathbf{M}(i,:) \mathrel{+}= \mathcal{X}(i,j,k)(\mathbf{B}(j,:) * \mathbf{C}(k,:))$	mode-(2,3)	mode -1	mode-1	mode-(2,3)⇒mode-2
SpTTMc on mode-1	$\mathbf{Y}_{(1)}(i,:) \mathrel{+}= \mathcal{X}(i,j,k)(\mathbf{U}_2(j,:) \otimes \mathbf{U}_3(k,:))$	mode-(2,3)	mode-1	mode-1	mode-(2,3)

Table I: Mode definitions for sparse tensor operations (mode-1: i, mode-2: j, mode-3: k). The symbol $\Rightarrow$ indicates the mode change from input tensor to output. Based on mode classification, we can provide a unified view for sparse tensor operations.

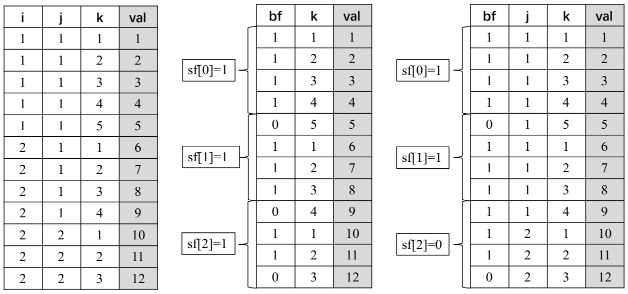
starts inside the current partition. Section IV-D demonstrates how the flag arrays are used to implement segmented scan to remove atomic updates and increase parallelism in tensor computations. Table I and the F-COO storage format can be extended to support other tensor operations and higher-order tensors.

As demonstrated in Figure 2, F-COO is used as a unified storage format for different tensor operations reducing tensor storage costs and enabling the application of unified parallel algorithms across tensor operations. Existing methods optimize tensor operations in isolation, requiring a different storage format and optimization strategy for each tensor operation [18]. For example, ParTI parallelizes SpTTM on the tensor fibers where the input is stored in a compressed fiber-order. For the SpMTTKRP operation, ParTI uses the COO storage format to enable operating on the non-zeros of the tensor. Our unified algorithm and storage format capture the similarity between these two operations.

Like COO, F-COO stores non-zero tensor elements. However, it does not suffer from load imbalance and can maintain maximum parallelism when operating on sparse tensors on different modes. Also, similar to COO, F-COO is insensitive to the irregularities of the underlying sparse tensor structures; this is why COO is useful in sparse matrix computations [25]. One of the major drawbacks of using COO in tensor computations is that COO has a high memory footprint because all the product and index mode indices have to be explicitly stored and accessed. Compared to COO, F-COO is more memory-efficient because it only keeps the indices on the product mode; the index modes are not stored and only a change in their values are stored in a considerably smaller bit-flag array. The number of non-zeros processed per thread depends on the data type selected for the bf array shown in Figure 2. For example, if we use uint8_t or unsigned char to bf, the number of non-zeros processed per thread will be 8. For the sf array, we use unsigned int to compress 32 bits for values accessed by threads in one warp concurrently to save bandwidth. Table II compares the storage costs of COO and F-COO for the SpTTM and SpMTTKRP operations; these can be extended to other sparse tensor operations.

C. One-Shot Sparse Tensor Computations

In the proposed unified approach, the F-COO storage format is used to compute tensor operations in *one-shot*. The one-shot strategy eliminates the need to create large intermediate data and avoids transformations between different F-COO representations. As shown in Figure 3a, when a sparse tensor operation such as SpMTTKRP is transformed into a series of sparse computations it will generate intermediate tensors which lead to extra storage costs. Figure 3a shows that operating on the sparse tensor $\mathcal{X}$ of size $I \times J \times K$ will generate an intermediate tensor $\mathcal{Y}$ of size $I \times J \times R$, which has larger storage costs compared to $\mathcal{X}$ because mode-3 is dense. Also, operations on intermediate tensors require mode change in F-COO which is an expensive operation.



i	j	k	val
1	1	1	1
1	1	2	2
1	1	3	3
1	1	4	4
1	1	5	5
2	1	1	6
2	1	2	7
2	1	3	8
2	1	4	9
2	2	1	10
2	2	2	11
2	2	3	12

(a) COO

bf	k	val
1	1	1
1	2	2
1	3	3
1	4	4
0	5	5
1	1	6
1	2	7
1	3	8
0	4	9
1	1	10
1	2	11
0	3	12

(b) F-COO for mode-3 SpTTM

bf	j	k	val
1	1	1	1
1	1	2	2
1	1	3	3
1	1	4	4
0	1	5	5
1	1	1	6
1	1	2	7
1	1	3	8
1	1	4	9
1	2	1	10
1	2	2	11
0	2	3	12

(c) F-COO for mode-1 SpTTMRP

Figure 2: F-COO for a 3-order tensor computing SpTTM on mode-3 and SpMTTKRP on mode-1. As shown in Table I for SpTTM, the index modes are *i* and *j*. bf (bit-flag) will change from 1 to 0 when a change in the *i* or *j* value occurs. For SpMMTKRP, bf change from 1 to 0 when the index mode *i* changes. If each partition holds 4 tensor non-zeros, sf (start-flag) indicates whether partitions processed by the current thread start new indices on index mode over previous thread. sf for thread 0 is always 1 since it always starts new indices.

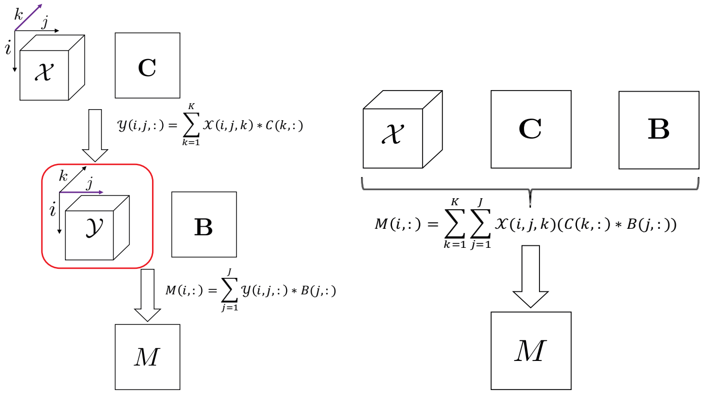
As shown in Figure 3b, the overhead in both storage cost and conversion between different F-COO representations is eliminated when performing sparse tensor operations in one-shot. The one-shot algorithm uses the product mode indices in F-COO to obtain rows from the dense factor matrices, C or B, and to perform a Hammard or Kronecker product. This intermediate result is then scaled using the corresponding non-zero value in the sparse tensor and is accumulated to the correct location using the indices from the index mode. The results are accumulated using segmented scan to reduce atomic operations in the parallel implementation.

D. Parallel Algorithms for Sparse Tensor Operations on GPUs

We propose unified parallel algorithms for sparse tensor operations on GPUs based on the F-COO storage format. This section discusses how our unified algorithm uses the segmented scan primitive to improve parallelization and how it reduces atomic updates in sparse tensor computations. We will also discuss our parallelization strategy which operates on the non-zeros of the sparse tensor to maintain load balance and

storage format	SpTTM on mode-3	SpMTTKRP on mode-1
COO	$16 \times nnz$	$16 \times nnz$
F-COO	$(8 + 1/8 + 1/(8 * threadlen)) \times nnz$	$(12 + 1/8 + 1/(8 * threadlen)) \times nnz$

Table II: Storage cost of a 3-order tensor for COO vs. F-COO. The storage cost of COO for a 3-order sparse tensor is $16 \times nnz$ bytes when integer and single-precision floating-point are used to store indices and non-zeros respectively. For SpTTM, indices in the index mode are replaced by bits, therefore, F-COO for SpTTM only takes $(8 + 1/8 + 1/(8 * threadlen)) \times nnz$ bytes (*threadlen* indicates the number of non-zeros processed per thread), where $8 \times nnz$ bytes are the storage cost for indices in the product mode (one integer per non-zero) and data values (one float per non-zero), $(1/8)nnz$ bytes is the memory cost from the bit-flag array and $1/(8 * threadlen)nnz$ is the storage cost for the start-flag array, which is a flag array for each thread. The storage cost of SpMTTKRP is obtained analogously.


k
j
i
$\mathcal{X}$
C
$\mathcal{Y}(i,j,:) = \sum_{k=1}^{K} \mathcal{X}(i,j,k) * C(k,:)$
k
j
i
$\mathcal{Y}$
B
$M(i,:) = \sum_{j=1}^{J} \mathcal{Y}(i,j,:) * B(j,:)$
M
$\mathcal{X}$
C
B
$M(i,:) = \sum_{k=1}^{K} \sum_{j=1}^{J} \mathcal{X}(i,j,k)(C(k,:) * B(j,:))$
M

(a) Previous method for SpMTTKRP

(b) One-shot method for SpMTTKRP

Figure 3: Figure (a) shows that previous fiber-centric SpMTTKRP implementations first multiply along mode-k with matrix C, then multiply along mode-j with matrix B. The drawback is that an intermediate tensor $\mathcal{Y}$ is generated and tensor operations will have to switch between different modes. Figure (b) illustrates our proposed one-shot method for SpMTTKRP. Our method directly performs computations on the non-zeros of the sparse tensor in one-shot.

provides consistently good performance for different ranks in sparse tensor computations. GPU-specific optimization techniques are also discussed. Finally, we will demonstrate how unified can be used to implement complete tensor algorithms such as CP decomposition.

Enabling segmented scan: Using the F-COO storage format, the non-zeros in the tensor are accessed to apply the computations in parallel and reduce the results using the product indices. The F-COO format has two flags, the bit-flag and the start-flag, both of which are used to implement the segmented scan algorithm to parallelize and reduce product results in the sparse tensor computation. The bit-flag is toggled when a new fiber or slice starts and the start-flag is used to indicate the start of a fiber or slice inside a partition. Partitions are allocated to different threads and their sizes are tuned for best performance. Details of the segmented scan algorithm can be found in [26], [27] and are not repeated here.

Parallelization strategy: The unified approach partitions the data and parallelizes computations based the non-zeros of the sparse tensor and the columns of the dense factor matrices as shown in Figure 4. As a result, our approach delivers consistently good performance for larger factor matrices where the rank of the tensor operation increases. The number of columns in dense matrices represent the rank of the tensor decomposition. Previous work on sparse tensor optimizations on GPUs [13] uses two-dimensional thread blocks in their implementation where the shape of thread blocks varies with the rank of the tensor operation. For example, when the number of threads is 512 in a two-dimensional thread block and rank is 32, the shape of the two-dimensional thread block will be $(16, 32)$. Since the threads inside a warp are in charge of computing the product of two dense columns in the implementations proposed in [13], the shape of the thread block can lead to thread divergence inside a warp and cause strided memory accesses. As a result, the performance of the code from [13] can vary for different ranks of the tensor operation.

To resolve this issue, we launch two-dimensional thread grids with one-dimensional thread blocks. One-dimensional thread blocks operate on their allocated partitions of the sparse tensor and columns of the dense matrices indicated by thread block index (bIdx, bIdy) as shown in Figure 4. Since the thread block dimensions in our approach do not vary with rank, the rank of sparse tensor operations will not affect parallelism and the memory access patterns in the proposed unified approach.

GPU-specific optimizations: A number of GPU-specific optimization techniques are used to further improve the performance of our unified algorithm on GPUs. Since sparse tensor operations on GPUs are memory-bound, all of the techniques are memory-oriented optimizations to efficiently use the GPU

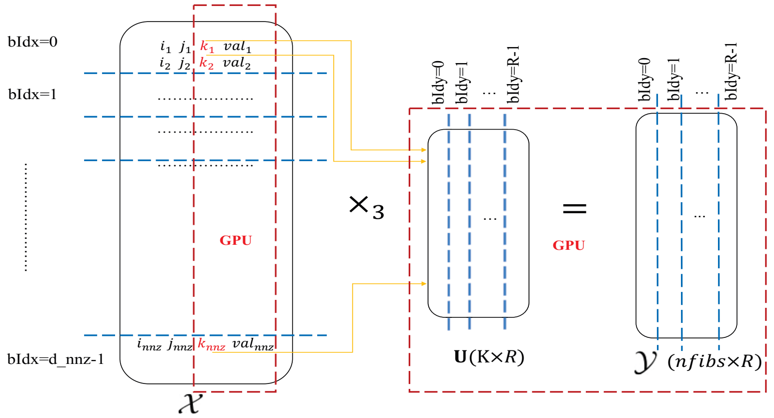


Figure 4: The parallelization and partitioning strategy used in unified for SpTTM on mode-3 (($\mathcal{Y} = \mathcal{X} \times_3 U$)). R is the number of columns in the dense factor matrix. A thread block is shown by a two-dimensional index (bIdx, bIdy) in the figure. d_nnz represents the dimension of the thread grid along the *x* dimension: non-zeros of sparse tensor, R represents the dimension of thread grid along the *y* dimension: column dimension of the dense matrix.

memory hierarchy. Our optimizations include using the read-only data cache, fusing kernels, and applying warp shuffle. Since in a single SpTTM and SpMMTKRP operation the dense factor matrices are read-only, they are cached in the Read-Only Data-Cache to further reduce global memory loads. Adjacent synchronization [27] is used to perform inter-block communication and to fuse the kernels in the sparse tensor implementation. Kernels such as the product kernel, segmented scan, and the accumulation kernels are fused to increase data reuse and keep intermediate data in shared memory. For the segment scan implementation, warp shuffle is used to increase data sharing inside a warp. Warp shuffle enables register to register data exchange and thus reduces the shared memory footprint and avoids overusing shared memory.

Complete tensor-based algorithms: Sparse tensor operations such as SpMTTKRP and SpTTM are used inside *complete tensor-based algorithms* such as the Tucker and CP decomposition algorithms. To our knowledge there are currently no implementations of CP or Tucker for sparse tensors on GPUs. We implement the CP decomposition algorithm to show our unified approach is insensitive to the mode being operated on. As a result, the MTTKRP operations in lines 2, 4, and 6 in Algorithm 1 will have very similar and well-balanced execution times. To eliminate the need for format conversations or CPU-GPU data transfers inside a CP iteration, F-COO is preprocessed for different modes on the host and will only be transferred once in the beginning to the GPU. For very large tensors, multiple-GPUs can be used. A similar approach can be used to implement Tucker using unified.

V. Experiments

We evaluate the performance of the unified approach by comparing to two state-of-the-art tensor libraries, namely ParTI [18] and SPLATT [11]. ParTI accelerates sparse tensor operations on multicore CPU and GPU architectures. SPLATT provides high-performance implementations of SpMTTKRP on shared-memory systems. SPLATT doesn't support sparse tensor operations on GPUs. All the experiments on the CPU platform for ParTI-omp and SPLATT are executed with 12 threads. For fair comparison, we follow the execution instructions provided by the authors of SPLATT and ParTI libraries.

Parameters	Intel Core i7-5820K	NVIDIA GeForce GTX Titan X
Microarchitecture	Haswell	Maxwell
Frequency	3.3GHz	1.0 GHz
Physical cores	6	3072
Peak SP Performance	56.72 Gflops	6144Gflops
Last-level cache	15MB	3MB
Memory size	64GB	12GB
Memory bandwith	68 GB/S	336GB/S
compiler	gcc 5.4.0	nvcc 8.0

Table III: Experimental platform configuration.

Dataset	order	Mode sizes	nnz	density
brainq	3	$60 \times 70K \times 9$	11M	$2.9e-01$
nell2	3	$12K \times 9K \times 29K$	77M	$2.5e-05$
dellicious	3	$0.5M \times 17.3M \times 2.5M$	140M	$6.1e-12$
nell1	3	$2.9M \times 2.1M \times 25.5M$	144M	$9.3e-13$

Table IV: Description of sparse tensor datasets.

Our experiments are performed on an Intel Core i7-5820K CPU and the NVIDIA GeForce GTX Titan X platforms. Hardware configurations are shown in Table III. We use a number of datasets from real applications provided by FROSTT [28]. Nell1 and nell2 come from Never Ending Language Learning (NELL) project [29] and represent *noun-verb-noun* triplets. The brainq dataset is generated from functional Magnetic Imaging (fMRI) measurements of brain activity [30], which represents *noun-voxel-human* subjects. Delicious is a *user-item-tag* tensor crawled from tagging systems [31]. Table IV provides more details on the datasets.

Because both the sparsity pattern of the tensor and its partitioning scheme will impact the memory footprint of sparse tensor operations on GPUs, we tune the *threadlen* and *BLOCK_SIZE* parameters to find their best configuration. The parameter *threadlen* indicates the number of non-zeros processed by each thread and *BLOCK_SIZE* shows the number of threads inside a thread block. As shown in Figure 5, the best parameter configurations for nell1 and brainq for SpMTTKRP on mode-1 are (*BLOCK_SIZE*=128 and *threadlen*=64) and (*BLOCK_SIZE*=32, *threadlen*=16) respectively. The best parameter configuration for each dataset and sparse tensor operation can be found in Table V. These parameters are used to obtain the results for unified in the upcoming sections.

Operations	nell-1	delicious	nell-2	brainq
SpTTM	(32,8)	(512,8)	(256,64)	(1024,32)
SpMTTKRP	(32,16)	(32,8)	(1024,64)	(128,64)

Table V: The best parameters for SpTTM on mode-3 and SpMTTKRP on mode-1.

A. Performance Results and Analysis

This section compares the performance of the unified method with ParTI and SPLATT. ParTI-omp and ParTI-GPU are parallel implementations of ParTI on multi-core and GPU architectures respectively. The number of columns of dense matrices in tensor computations is set to 16, which is also the rank of the tensor decomposition. Since SPLATT only supports SpMTTKRP, the performance of unified method for the SpTTM operation is only compared to ParTI. As shown in Figure 6a, compared to ParTI-omp unified achieves $5.3\times$ (nell1) to $215.7\times$ (brainq) speedup. Compared to ParTI-GPU, unified achieves $1.1\times$ (nell1) to $3.7\times$ (brainq) speedup. For the SpMTTKRP operation ParTI-GPU runs out of memory for larger tensors such as nell1 and delicious. As shown in Figure 6b, compared to ParTI-omp, our proposed method achieves from $8.1\times$ (nell1) to $102.5\times$ (brainq) speedup. Compared to ParTI-GPU, we achieve $23.7\times$ speedup on nell2 and $30.6\times$ speedup on brainq. Unified achieves a speedup of $1.4\times$ for nell2 and $12.5\times$ for brainq compared to SPLATT.

The experiments show the unified method achieves best performance for the brainq dataset and does not perform well for the nell1 dataset. nell1 is extremely sparse with a density of 9.1e-13 (nell1) while brainq is the densest tensor in our dataset. The performance of tensor operations on GPUs tend not to be good for very sparse tensors because the non-zero elements processed by one warp will need access to columns of the dense factor matrices that are located far apart based on the indices of the product mode. To resolve this issue, we use a read-only data cache to cache accesses to the dense matrices. However, if the indices in the product mode vary to a large extent, cache hit rates will decrease. Thus, while unified does not perform well for nell1 it performs better for brainq (density: 2.9e-01), nell-2 (density: 2.5e-05), and delicious (density: 6.1e-12).

B. Mode Behavior

The experiments in this section demonstrate that the performance of the unified method does not depend on the mode being operated on and performs relatively the same mainly because it is based on the F-COO format. Since brainq is one of the "oddly" shaped tensors in our dataset with $60\times 70K\times 9$ dimensions, it is used to examine the performance of unified on different modes. As shown in Figure 7, for both SpTTM or SpMTTKRP, unlike ParTI-GPU and SPLATT, the running time of unified method remains relatively the same for different modes.

Unified performs well for all modes because it uses the F-COO format to partition and parallelize based on the tensor non-zeros. However, because ParTI-GPU operates on fibers of the sparse tensor, its performance changes for each mode. For example, when computing SpTTM on mode-2 of brainq, ParTI only launches up to 540 threads to perform computations on fibers in parallel and thus does not efficiently use the GPU resources. SPLATT organizes the sparse tensor as a tree. Thus, parallelizing computations for different modes requires operating on different levels of the tree which changes the level of parallelization and the memory access patterns.

C. Rank Behavior

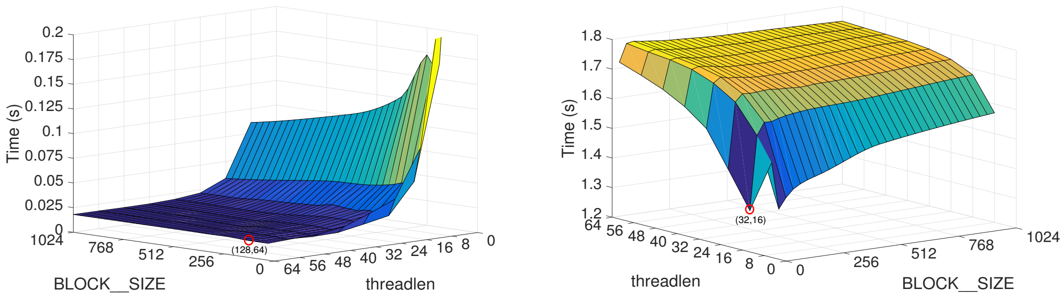
As discussed in Section IV-D, the performance of unified scales well when the number of columns in the dense factor matrices is increased, i.e. increasing the rank of the tensor decomposition. We ran unified for different ranks $(8, 16, 32, 64)$. As demonstrated in [13], when the rank increases the resulting tensor or matrix from the sparse tensor operations becomes larger, thus, we only test rank behavior for the two smallest tensors brainq and nell2. As shown in the Figure 8, when the rank varies from 8 to 64 the execution time of ParTI increases at a faster rate compared to unified. The speedup of unified over ParTI-GPU for brainq varies from $3.7\times$ to $4.3\times$ and the speedup of unified over ParTI-GPU for nell1 varies from $2.1\times$ to $2.4\times$.

D. Memory Consumption on GPUs

SpTTM does not generate intermediate data, thus, the memory consumption of unified and ParTI is nearly the same. However, ParTI does generate intermediate data for SpMTTKRP. Since ParTI runs out of memory when operating on the larger tensor datasets we computed the memory consumption by hand for nell1 and delicious based on ParTI's open source code. The memory consumption for the other two datasets is measured by executing the code. As shown in Figure 9, compared to ParTI-GPU, our method reduces the memory consumption by 68.6% for nell1 and 88.6% for brainq because of the one-shot computations.

E. CANDECOMP/PARAFAC (CP) Decomposition on GPUs

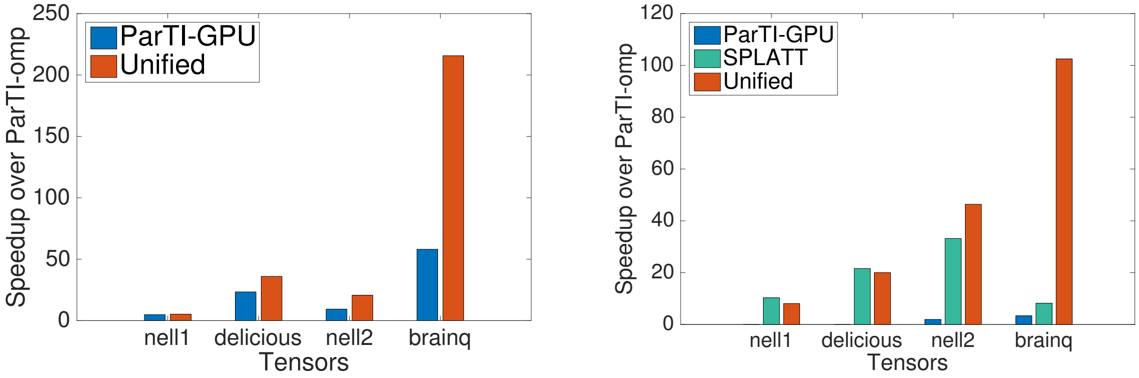
To the best of our knowledge, this work provides the first implementation for the CP decomposition on GPUs. Our implementation creates two GPU streams where one stream performs SpMTTKRP on different modes and the other



(a) brainq

(b) nell1

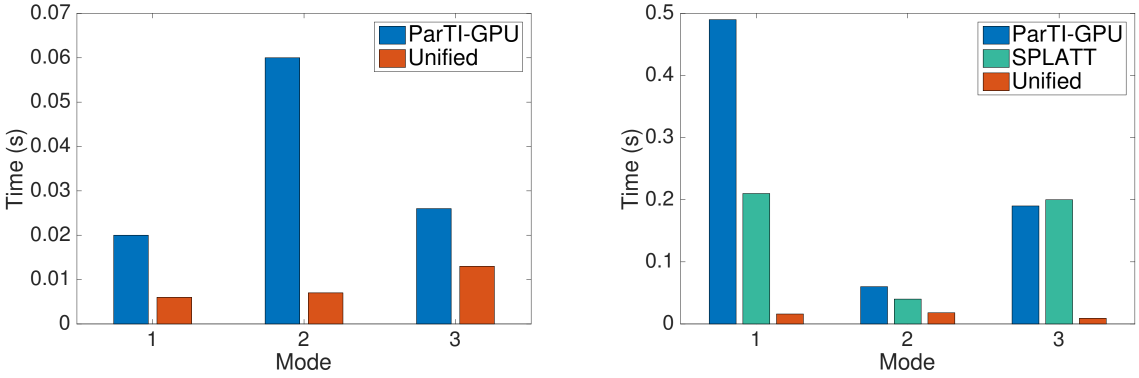
Figure 5: Tuning the best parameters *threadlen* and *BLOCK_SIZE* for SpMTTKRP on mode-1.



(a) SpTTM on mode-3

(b) SpMTTKRP on mode-1

Figure 6: Unified's (GPU) speedup over ParTI (CPU, GPU) and SPLATT (CPU) (rank=16); higher is better.



(a) SpTTM

(b) SpMTTKRP

Figure 7: Mode behavior of different implementations on the GPU (SPLATT is executed on the CPU) (rank=16) on the ”brainq” datasets; lower is better.

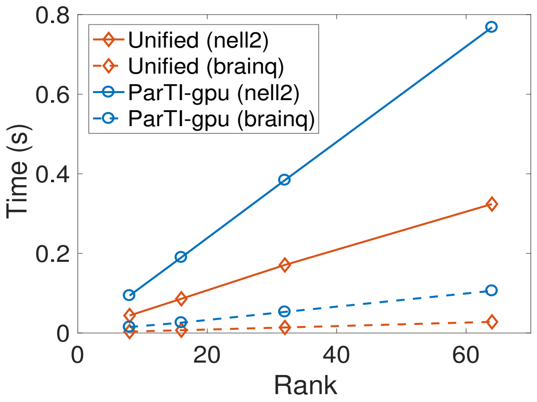


Figure 8: The execution time of unified (GPU) and ParTI (CPU) for SpTTM with different rank sizes for the "brainq" and "nell2" datasets; lower is better.

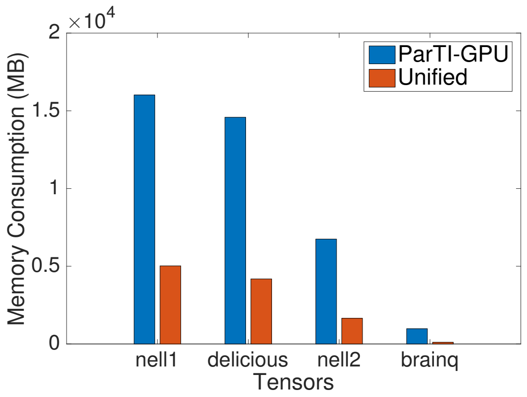


Figure 9: The GPU global memory consumption of unified vs. ParTI for SpMTTKRP on mode-1; lower is better.

performs matrix operations including matrix multiplication and matrix inversion using the CUBLAS library [32]. As a result, the computations performed by separate streams would be overlapped automatically when possible. Because a single-GPU memory can not store all the tensor data for the CP decomposition we provide results for brainq and nell2. For larger datasets, multiple GPU cards can be used. As shown in Figure 10, most of execution time for CP decomposition is spent on the SpMTTKRP operation and unlike SPLATT, in unified computations are well-balanced between operations for both tensors.

We compare the performance of SPLATT with our unified method for CP decomposition. ParTI doesn't support CP on GPUs. The rank of the tensor decomposition is fixed to 8 to represent the low-rank property of the tensor decomposition. Another important reason is that one of the dimensions in brainq $(60 \times 70k \times 9)$ is 9 and ranks larger than 9 will create a deficient matrix in the tensor decomposition algorithm. Figure 10 shows per iteration runtimes for CP decomposition; converting the storage format to F-COO is done only once before iterations and thus is not reported and is considered as per-processing overhead. From Figure 10, the unified method achieves $14.9\times$ and $2.9\times$ speedup over SPLATT for brainq and nell2 respectively. As shown in Figure 10, most of time in our implementation is spent on performing SpMTTKRP operations.

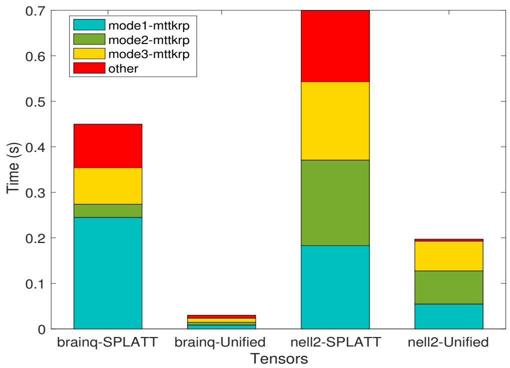


Figure 10: Running time per iteration of the unified method (on GPU) and SPLATT on CPU performing CP decomposition on "brainq" and "nell2" datasets; lower is better.

VI. Conclusion

This paper proposes a unified sparse storage format and parallel algorithms for sparse tensor operations on GPUs. The tensor modes for different operations are encoded into unified to deliver highly-efficient implementations of sparse tensor operations. Unified is used across different tensor operations and also accelerates the execution of complete tensor-based algorithms. Unified's performance is not sensitive to mode changes in tensor methods, scales well with rank updates, and reduces memory footprints and storage costs on GPUs. Several techniques are used to further improve the performance of unified on GPUs such as the segmented scan method, kernel fusion, warp shuffle, and data reuse. The experiments show that our unified solution significantly outperforms state-of-art implementations of sparse tensor operations on multicore CPUs and manycore GPUs.

VII. Acknowledgement

This work is supported by the U.S. National Science Foundation (NSF) Award Numbers 1657175 and 1525276 and NVIDIA. We would like to thank Jiajia Li and Shaden Smith for their assistance with the ParTI and SPLATT experiments.

References

[1] A. Anandkumar, R. Ge, D. Hsu, S. M. Kakade, and M. Telgarsky, "Tensor decompositions for learning latent variable models." *Journal of Machine Learning Research*, vol. 15, no. 1, pp. 2773–2832, 2014.

[2] F. Huang, U. Niranjan, M. U. Hakeem, and A. Anandkumar, "Online tensor methods for learning latent variable models," *Journal of Machine Learning Research*, vol. 16, pp. 2797–2835, 2015.

[3] U. Kang, E. Papalexakis, A. Harpale, and C. Faloutsos, "Gigatensor: scaling tensor analysis up by 100 times-algorithms and discoveries," in *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2012, pp. 316–324.

[4] T. G. Kolda and J. Sun, “Scalable tensor decompositions for multi-aspect data mining,” in *Data Mining, 2008. ICDM’08. Eighth IEEE International Conference on*. IEEE, 2008, pp. 363–372.

[5] M. A. O. Vasilescu and D. Terzopoulos, “Multilinear analysis of image ensembles: Tensorfaces,” in *European Conference on Computer Vision*. Springer, 2002, pp. 447–460.

[6] M. A. O. Vasilescu, “Multilinear projection for face recognition via canonical decomposition,” in *Automatic Face & Gesture Recognition and Workshops (FG 2011), 2011 IEEE International Conference on*. IEEE, 2011, pp. 476–483.

[7] Y. Shi, A. Karatzoglou, L. Baltrunas, M. Larson, A. Hanjalic, and N. Oliver, “Tfmap: optimizing map for top-n context-aware recommendation,” in *Proceedings of the 35th international ACM SIGIR conference on Research and development in information retrieval*. ACM, 2012, pp. 155–164.

[8] V. Khoromskaia and B. N. Khoromskij, “Tensor numerical methods in quantum chemistry: from hartree–fock to excitation energies,” *Physical Chemistry Chemical Physics*, vol. 17, no. 47, pp. 31 491–31 509, 2015.

[9] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin *et al.*, “Tensorflow: Large-scale machine learning on heterogeneous distributed systems,” *arXiv preprint arXiv:1603.04467*, 2016.

[10] R. Collobert, K. Kavukcuoglu, and C. Farabet, “Torch7: A matlab-like environment for machine learning,” in *BigLearn, NIPS Workshop*, no. EPFL-CONF-192376, 2011.

[11] S. Smith, N. Ravindran, N. D. Sidiropoulos, and G. Karypis, “Splatt: Efficient and parallel sparse tensor-matrix multiplication,” in *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*. IEEE, 2015, pp. 61–70.

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

[13] J. Li, Y. Ma, C. Yan, and R. Vuduc, “Optimizing sparse tensor times matrix on multi-core and many-core architectures,” in *Proceedings of the Sixth Workshop on Irregular Applications: Architectures and Algorithms*. IEEE Press, 2016, pp. 26–33.

[14] I. Jeon, E. E. Papalexakis, U. Kang, and C. Faloutsos, “Haten2: Billion-scale tensor decompositions,” in *2015 IEEE 31st International Conference on Data Engineering*. IEEE, 2015, pp. 1047–1058.

[15] J. H. Choi and S. Vishwanathan, “Dfacto: Distributed factorization of tensors,” in *Advances in Neural Information Processing Systems*, 2014, pp. 1296–1304.

[16] O. Kaya and B. Uçar, “High performance parallel algorithms for the tucker decomposition of sparse tensors,” in *Parallel Processing (ICPP), 2016 45th International Conference on*. IEEE, 2016, pp. 103–112.

[17] ——, “Scalable sparse tensor decompositions in distributed memory systems,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2015, p. 77.

[18] J. Li, Y. Ma, C. Yan, J. Sun, and R. Vuduc, “Parti: a parallel tensor infrastructure for data analysis,” in *NIPS, Tensor-Learn Workshop*, 2016.

[19] T. G. Kolda and B. W. Bader, “Tensor decompositions and applications,” *SIAM review*, vol. 51, no. 3, pp. 455–500, 2009.

[20] W. Austin, G. Ballard, and T. G. Kolda, “Parallel tensor compression for large-scale scientific data,” *arXiv preprint arXiv:1510.06689*, 2015.

[21] B. W. Bader, T. G. Kolda *et al.*, “Matlab tensor toolbox version 2.6, available online, february 2015,” 2015.

[22] C. A. Andersson and R. Bro, “The n-way toolbox for matlab,” *Chemometrics and intelligent laboratory systems*, vol. 52, no. 1, pp. 1–4, 2000.

[23] E. Solomonik, D. Matthews, J. Hammond, and J. Demmel, “Cyclops tensor framework: Reducing communication and eliminating load imbalance in massively parallel contractions,” in *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*. IEEE, 2013, pp. 813–824.

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

[25] N. Bell and M. Garland, “Implementing sparse matrix-vector multiplication on throughput-oriented processors,” in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM, 2009, p. 18.

[26] S. Sengupta, M. Harris, and M. Garland, “Efficient parallel scan algorithms for gpus,” *NVIDIA, Santa Clara, CA, Tech. Rep. NVR-2008-003*, no. 1, pp. 1–17, 2008.

[27] S. Yan, G. Long, and Y. Zhang, “Streamscan: fast scan algorithms for gpus without global barrier synchronization,” in *ACM SIGPLAN Notices*, vol. 48, no. 8. ACM, 2013, pp. 229–238.

[28] S. Smith, J. W. Choi, J. Li, R. Vuduc, J. Park, and G. Karypis. (2017) FROSTT: The formidable repository of open sparse tensors and tools. [Online]. Available: http://frostt.io/

[29] A. Carlson, J. Betteridge, B. Kisiel, B. Settles, E. R. Hruschka Jr, and T. M. Mitchell, “Toward an architecture for never-ending language learning.” in *AAAI*, vol. 5, 2010, p. 3.

[30] T. M. Mitchell, S. V. Shinkareva, A. Carlson, K.-M. Chang, V. L. Malave, R. A. Mason, and M. A. Just, “Predicting human brain activity associated with the meanings of nouns,” *science*, vol. 320, no. 5880, pp. 1191–1195, 2008.

[31] O. Gorlitz, S. Sizov, and S. Staab, “Pints: peer-to-peer infrastructure for tagging systems.” in *IPTPS*. Citeseer, 2008, p. 19.

[32] C. Nvidia, “Cublas library,” *NVIDIA Corporation, Santa Clara, California*, vol. 15, p. 27, 2008.