Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Optimizing sparse tensor times matrix on GPUs

Yuchen Ma [a], Jiajia Li [b,*], Xiaolong Wu [c], Chenggang Yan [a], Jimeng Sun [b], Richard Vuduc [b]

[a] *Institute of Information and Control, Hangzhou Dianzi University, Hangzhou, China*
[b] *Computational Science & Engineering, Georgia Institute of Technology, Atlanta, GA 30332, United States*
[c] *Computer Science, Virginia Tech, Blacksburg, VA 24061, United States*

HIGHLIGHTS

- Designed an in-place SpTTM algorithm to avoid tensor-matrix data transformation.
- Optimized SpTTM and SspTTM on GPUs.
- Implement the Tucker decomposition on a single node GPU system.
- Get better performance than state of the arts.

ARTICLE INFO

Article history:
Received 22 April 2017
Received in revised form 26 June 2018
Accepted 21 July 2018
Available online xxxx

Keywords:
Sparse tensors
Irregular algorithms
Tensor decomposition
GPU

ABSTRACT

This work optimizes tensor-times-dense matrix multiply (TTM) for general sparse and semi-sparse tensors on CPU and NVIDIA GPU platforms. TTM is a computational kernel in tensor methods-based data analytics and data mining applications, such as the popular Tucker decomposition. We first design an in-place sequential SPTTM to avoid explicit data reorganizing between a tensor and a matrix in its conventional approach. We further optimize SPTTM on NVIDIA GPU platforms. Five approaches including employing fine thread granularity, arranging coalesced memory access, rank blocking, and using fast GPU shared memory are developed for GPU-SPTTM. We also optimize semi-sparse tensor-times-dense matrix multiply (SSPTTM) to take advantage of the inside dense sub-structures. The optimized SPTTM and SSPTTM are applied to Tucker decomposition to improve its overall performance.

Our sequential SPTTM is 3–120× faster than the SPTTM from Tensor Toolbox library. GPU-SPTTM obtains 6–19× speedup on NVIDIA K40c and 23–67× speedup on NVIDIA P100 over CPU-SPTTM respectively. Our GPU-SPTTM is 3.9× faster than the state-of-the-art GPU implementation. Our SSPTTM implementations outperform SPTTMS by up to 4.5×, which handles the input semi-sparse tensor in a general way. Tucker decomposition achieves up to 3.2× speedup after applying the optimized TTMS. The code will be publicly released in PARTI! library: https://github.com/hpcgarage/ParTI.

1. Introduction

This paper considers the problem of optimizing the tensor-times-dense matrix (TTM) operation for sparse tensors, which appears widely in tensor-based data analytics. Such applications arise in numerous domains, including neuroscience [13,33], healthcare analytics [21,22,59], natural language processing [25], signal processing [30], machine learning [1,2], and social network analytics [48]. Tensors, which are multi-way arrays, provide a natural way to represent multi-dimensional data; analysis of a tensor usually takes the form of factorizing or decomposing the tensor into interpretable components [23,28,46,49]. (This process is analogous to the use of matrix decompositions to analyze 2-way data; tensors generalize such analyses to the k-way case for $k > 2$.) The speed of some of the most popular tensor decompositions, including the so-called Tucker decomposition [28], depend critically on having a fast TTM, thereby motivating this study.[1]

Regarding this paper's scope, we consider thread parallelism and memory locality for single-node GPU platforms, and we are particularly interested in *sparse* input tensors. Sparsity refers to the tensor consisting mostly of zero entries, for which we wish to avoid explicit storage and computation. By contrast, several efficient methods exist for the case when a tensor is dense [45,50,51,54]. The sparse case is especially important to data analytics applications, since real-world data is often voluminous but sparse, and tensor factorizations as the initial data processing stage need to effectively and efficiently compress the data.

* Corresponding author.
E-mail address: jiajiali@gatech.edu (J. Li).

[1] Beyond TTM, other basic tensor operations that appear in other decompositions include tensor matricization (converting a tensor to an equivalent matrix), element-wise tensor operations, Kronecker products, Khatri–Rao products, and Matricized Tensor Times Khatri–Rao Product (MTTKRP).

https://doi.org/10.1016/j.jpdc.2018.07.018
0743-7315/© 2018 Elsevier Inc. All rights reserved.

In principle, a sparse TTM is similar to a sparse matrix-times-dense matrix. Conventional SPTTM implementations, such as those in Tensor Toolbox [4,5] and Cyclops Tensor Framework (CTF) [55], first transform a sparse tensor into an equivalent sparse matrix and then assume an optimized sparse matrix-times-dense matrix to invoke. This approach is reasonable by employing highly efficient state-of-the-art sparse matrix-times-dense matrix. However, this conversion step incurs non-trivial cost in time and space. Besides, the generated matrix could be extremely large in one dimension, explicitly indexing of which – for a many-way tensor – can quickly exceed the range of a 64-bit unsigned integer. Therefore, we are motivated primarily to avoid any such conversion, carrying out the sparse TTM "natively" on a given input tensor.

Tucker decomposition consists of two types of sparse TTMS, general sparse TTM (SPTTM) and semi-sparse TTM (SSPTTM) (details in Section 5). SPTTM, as we know, is a general sparse tensor times a dense matrix; while SSPTTM is a semi-sparse tensor times a dense matrix. A semi-sparse tensor is a dense-structured tensor that at least one mode is dense. For example, a tensor with its first mode dense means the fibers of this mode are either empty or fully dense. For a semi-sparse tensor, as observed by Baskaran et al. [7], its inside dense structures can be explored for better performance rather than being treated as in general. In this work, along with optimizing the performance of SPTTM on GPU platforms, we also accelerate SSPTTM on them.

Our proposed techniques make the following contributions:

First, we design an in-place sequential SPTTM algorithm to avoid tensor-matrix data transformation, which is based on a structured sparse tensor format and a certain special property. Additionally, an auxiliary array is used to avoid memory write conflict for the later-on parallel SPTTM algorithms. Our sequential SPTTM is 3–120$\times$ faster than the SPTTM from Tensor Toolbox library [5] (Sections 3 and 4.1).

Second, we parallelize SPTTM on single-node GPU systems. We propose several optimizing approaches for SPTTM on NVIDIA GPUs: employing fine thread granularity, arranging coalesced memory access, rank blocking, and using local (fast) memory ("shared memory" on GPUs). GPU-SPTTM obtains 6–19$\times$ speedup on NVIDIA K40c and 23–67$\times$ speedup on NVIDIA P100 over CPU-SPTTM respectively for real-world tensors. Our GPU-SPTTM is 3.9$\times$ faster than the state-of-the-art GPU implementation (Section 4.2).

Third, we implement SSPTTM on GPUs accordingly by better exploring the dense sub-structures. Our SSPTTM implementations outperform SPTTMS which handles the input semi-sparse tensor in a general way by 4.5$\times$ (Section 5).

Lastly, by applying SPTTM and SSPTTM to the Tucker decomposition, it outperforms CPU Tucker decomposition by 3.2$\times$ (Section 6).

Partial work has been published in our previous paper [35]. This paper is based on the previous work but extends it from the four aspects below. (1) This work further optimizes SPTTM on NVIDIA GPUs by employing five optimization approaches and providing a more in-depth analysis of their performance. (2) SSPTTM optimizations on these platforms are also explored, which further speed up TTM operations in Tucker decomposition. (3) We built a parallel Tucker decomposition for GPUs by applying the optimized SPTTM and SSPTTM. (4) Afterwards, these algorithms are tested on an extended sparse tensor set, including both third- and fourth-order real-world tensors.

2. Background

This section introduces some essential tensor notations. Some of its examples and definitions come from the overview by Kolda and Bader [28]. A list of symbols and notation in this paper is shown in Table 1.

Table 1
List of symbols and notations.

Symbols	Description
$\mathcal{X}, \mathcal{Y}, \mathcal{Z}$	Sparse and semi-sparse tensors
ind, val	Index and value arrays of $\mathcal{X}$ in COO format
$\mathbf{X}_{(n)}$	Matricized tensor $\mathcal{X}$ in mode-n
$\mathbf{A}, \mathbf{B}, \mathbf{C}, \tilde{\mathbf{A}}$	Dense matrices
$\mathbf{a_r}, \mathbf{b_r}, \mathbf{c_r}$	Dense vectors
–	Weight vector
N	Tensor order
I, J, K	Tensor mode sizes
nnz	#Nonzeros of a tensor
R	Approximate tensor rank (usually a small value)
n_{fibs}	#Mode-n fibers of $\mathcal{X}$
f_{ptr}	The beginnings of $\mathcal{X}$'s mode-n fibers, sized n_{fibs}
f_{len}	The average length of $\mathcal{X}$'s mode-n fibers, $flen = \frac{nnz}{n_{fibs}}$
$D_{\mathcal{Y}}$	Dense mode set of a semi-sparse tensor
$S_{\mathcal{Y}}$	Sparse mode set of a semi-sparse tensor
n_{chunks}	#$D_{\mathcal{Y}}$-chunks of $\mathcal{Y}$
c_{ptr}	The beginnings of $\mathcal{Y}$'s $D_{\mathcal{Y}}$-chunks, sized n_{chunks}
N_{TB}	Maximum #Threads per block
S_{SM}	Maximum shared memory size in words
N_{TR}	Maximum #Threads per block for a matrix row

2.1. Tensor representations

The order N of an Nth-order tensor is sometimes also referred to the number of *modes* or *dimensions*. A first-order ($N = 1$) tensor is a vector, denoted by a boldface lowercase letter, e.g., $\mathbf{v}$; A second-order ($N = 2$) tensor is a matrix, denoted by a boldface capital letter, e.g., $\mathbf{A}$. Higher-order tensors ($N \geq 3$) are denoted by bold capital calligraphic letters, e.g., $\mathcal{X}$. A scalar element at position (i, j, k) of a tensor $\mathcal{X}$ is x_{ijk}. We show an example of a sparse third-order tensor, $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$, in Fig. 1(a).

Some tensor algorithms operate on the subsets of a tensor. One such subset is the mode-n *fiber*, shown in Fig. 1(b); it is a vector extracted by fixing the indices of all modes but mode n. For example, the mode-1 fiber of a tensor $\mathcal{X}$ is denoted by the vector $\mathbf{f}_{:jk} = \mathcal{X}(:, j, k)$, where the colon indicates all indices of mode 1. A *slice*, shown in Fig. 1(c), is a 2-dimensional cross-section (i.e., matrix) of a tensor, extracted by fixing the indices of all modes but two, e.g. $\mathbf{S}_{::k} = \mathcal{X}(:, :, k)$.

Some other tensor algorithms operate on the equivalent matrix by folding (or reshaping) a tensor. *Matricization* reshapes a tensor into an equivalent matrix by arranging all mode-n fibers to be the columns of a matrix. For example, mode-1 matricization of a tensor $\mathcal{X} \in \mathbb{R}^{3\times4\times5}$ results in a matrix $\mathbf{X}_{(1)} \in \mathbb{R}^{3\times20}$. (Readers may refer to Kolda and Bader's survey for more details [28].)

2.2. TTM

Tensor-Times-Matrix (TTM) in mode n, also known as the n-mode product, is the multiplication of a tensor $\mathcal{X} \in \mathbb{R}^{I_1\times\cdots\times I_n\times\cdots\times I_N}$ with a matrix $\mathbf{U} \in \mathbb{R}^{I_n\times R}$,[2] denoted by $\mathcal{Y} = \mathcal{X} \times_n \mathbf{U}$. This results in an $I_1 \times \cdots \times I_{n-1} \times R \times I_{n+1} \times \cdots \times I_N$ tensor, and its operation is defined as

$$y_{i_1\cdots i_{n-1} r i_{n+1}\cdots i_N} = \sum_{i_n=1}^{I_n} x_{i_1\cdots i_{n-1} i_n i_{n+1}\cdots i_N} u_{i_n r}. \tag{1}$$

Since the real application data is always sparse with a relatively small amount of non-zero entries, we consider sparse tensor-times-dense matrix in this paper. Sparse TTM is a critical kernel in tensor decomposition algorithms, such as the Tucker decomposition. The factor matrices in tensor decomposition are usually dense, and the number of columns of each matrix is called *rank*.

[2] Different from Kolda and Bader's definition [28], we use the transposed form of the matrix $\mathbf{U}$ for efficient TTM in row-majored storage pattern of C language.

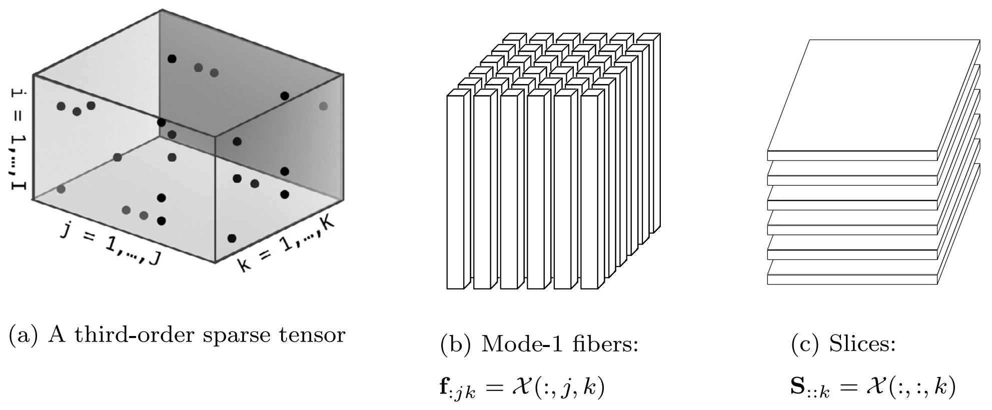
i = 1,...,I
j = 1,...,J
k = 1,...,K
(a) A third-order sparse tensor
(b) Mode-1 fibers: $\mathbf{f}_{:jk} = \mathcal{X}(:, j, k)$
(c) Slices: $\mathbf{S}_{::k} = \mathcal{X}(:, :, k)$

Fig. 1. Representations of a third-order tensor $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$, where a colon indicates all indices of a mode.

We focus on the popular low-rank tensor decompositions, which means the factor matrices all have a small number of columns, usually less than 100.

The traditional SpTtm algorithm in Tensor Toolbox [5] first matricizes the input sparse tensor to a sparse matrix, then does a sparse matrix-dense matrix multiplication, afterwards, transforms the output matrix back to a tensor. The explicit two-stage transformation consumes non-trivial execution time, which is about 17% for tensor *choa* using sparse Ttm in Tensor Toolbox.

2.3. NVIDIA GPU architecture

NVIDIA Graphic Processor Units (GPUs) are highly parallel, many-core processors with high memory bandwidth [14]. We particularly consider GPUs that support CUDA, a general-purpose parallel computing platform and programming model. A GPU consists of many CUDA cores, which are organized as groups of Streaming Multiprocessors (SMs). The P100 GPU card used in this work employs Pascal architecture, which integrates 56 SMs and each SM consists of 64 CUDA cores. All cores in an SM share 64 KB configurable on-chip memory, L1 cache and shared memory. Three configurations are available: 16 KB/48 KB, 32 KB/32 KB, and 48 KB/16 KB for L1 cache and shared memory respectively. Global memory is cached by both L1 and L2 caches depending on the data is writable or read-only. Writable global memory data can only be cached by the L2 cache, read-only global memory data also can be cached by the L1 cache by marking the data with both "`const`" and "`__restrict__`" keywords. It is necessary to carefully allocate memory access paths for different types of data. CUDA has three-level thread hierarchy, grids, thread blocks, and threads. The maximum of their sizes varies among different GPU architectures. A P100 GPU supports up to 1024 threads per thread block. Threads are executed simultaneously in warps, usually a group of 32 threads. It is critical to have enough blocks and threads, good thread behavior inside a warp, and efficient memory access to achieve a satisfiable performance of CUDA programs.

3. Sparse tensor formats and property

Two sparse tensor formats will be introduced along with a property which is critical to the fast implementations of both SpTtm and SspTtm.

3.1. Notations

We first introduce some names for tensor modes with specific properties, which will be used in the following content.

ind_1	ind_2	ind_3	val
1	1	1	1
1	1	2	2
2	1	1	3
2	1	2	4
3	3	1	5
3	3	2	6

(a) COO

ind_1	ind_2	val
1	1	1 2
2	1	3 4
3	3	5 6

(b) sCOO

Fig. 2. COO and sCOO formats of a semi-sparse $3 \times 3 \times 2$ tensor, with dense mode 3.

- *Product mode*: the mode in which a tensor times a matrix, e.g., mode n in Eq. (1).
- *Index mode*: the rest modes after excluding the product mode, e.g., modes $1, \ldots, n-1, n+1, \ldots, N$ in Eq. (1).
- *Dense mode*: the mode in which all non-empty fibers are fully dense.
- *Sparse mode*: the mode in which at least one non-empty fiber is sparse.

All the modes of a general sparse tensor are sparse modes, while at least one dense mode exists in a semi-sparse tensor.

3.2. Sparse tensor formats

COO format. Tensors generated from real-world applications are usually sparse with only a relatively small number of non-zero entries (as tensors will be shown in Table 3). Therefore, a sparse tensor is represented by a compressed format by storing only non-zero entries. The simplest yet popular format of a sparse tensor is coordinate (COO) format. *ind* and val represent the indices and values of a sparse tensor's non-zero entries respectively. val is an *nnz*-array of floating-point numbers, *ind* is an N-array of integer pointers, where every pointer stores the *nnz* indices of one mode. Fig. 2(a) shows a $3 \times 3 \times 2$ sparse tensor in the COO format. The indices of each mode are represented as ind_1, ind_2, and ind_3 for the third-order tensor. We notice that replicated indices exist in ind_1 (ind_2, or ind_3).

sCOO format. Since the indices of entries within a dense fiber inhere in their storage location, it is unnecessary to explicitly store the indices of dense modes using extra space for a semi-sparse tensor. We use a simple semi-COO (sCOO) format which only stores the indices of sparse modes and with all non-zeros in a particular order of dense modes. For example, the tensor in Fig. 2 is

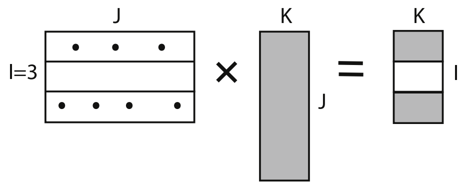

Fig. 3. Sparse matrix-dense matrix multiply.

actually a semi-sparse tensor with dense mode 3. Fig. 2(b) shows the sCOO format for it, where each index pair from ind_1 and ind_2 indexes a dense mode-3 fiber. The idea of distinguishing dense and sparse modes was first proposed by Baskaran et al. [7]. For this toy example tensor, sCOO format saves 50% storage compared to COO format. Assume integer and floating-point values have the same bit-length, sCOO format saves at least 25% storage for a third-order semi-sparse tensor with one dense mode and at least $\frac{k}{N+1}$ storage space for an Nth-order semi-sparse tensor with k dense modes. The storage saving is because (1) sCOO does not store all the indices of dense modes. (2) The rest index arrays of sCOO usually become shorter because of index compression, which saves more space.

3.3. Sparse tensor property

Property. SpTtm *outputs a semi-sparse tensor whose product mode is dense, while index modes remain unchanged.*

Proof. Assume an SpTtm takes a sparse tensor $\mathcal{X}$, a dense matrix $\mathbf{U}$ as inputs and a tensor $\mathcal{Y}$ as output. Mode-n fibers of $\mathcal{X}$ and $\mathcal{Y}$ tensors are defined as

$$\mathbf{f}_n^X = \mathcal{X}(i_1, \ldots, i_{n-1}, :, i_{n+1}, \ldots, i_N),$$

$$\mathbf{f}_n^Y = \mathcal{Y}(i_1, \ldots, i_{n-1}, :, i_{n+1}, \ldots, i_N),$$

where $i_1, \ldots, i_{n-1}, i_{n+1}, \ldots, i_N$ are given. $\mathbf{f}_n^X$ is a sparse fiber because of the sparsity of $\mathcal{X}$. Thus, Eq. (1) is equal to

$$f_n^Y(r) = \sum_{i_n=1}^{I_n} f_n^X(i_n) u_{i_n r}. \tag{2}$$

When r is fixed, element $\mathbf{f}_n^Y(r)$ is a dot-product of fiber $\mathbf{f}_n^X$ and $\mathbf{u}(:, r)$, a column of $\mathbf{U}$. Since $\mathbf{u}(:, r)$ is a dense vector, each $f_n^Y(r)$ is non-zero if at least one non-zero exists in fiber $\mathbf{f}_n^X$. That is, a non-empty fiber $\mathbf{f}_n^X$ generates a dense fiber $\mathbf{f}_n^Y$. For each pair of fixed indices $(i_1, \ldots, i_{n-1}, i_{n+1}, \ldots, i_N)$, we compute a mode-n fiber of $\mathcal{Y}$, which shows the indices $i_1, \ldots, i_{n-1}, i_{n+1}, \ldots, i_N$ are unchanged for the resulting tensor $\mathcal{Y}$. Fig. 3 shows the behavior of a second-order sparse tensor (sparse matrix) times a dense matrix. The product mode j is a dense mode in the resulting matrix, while its index mode i is the same with the input sparse matrix, except it indexes dense fibers of the output.

4. Sparse tensor times matrix

4.1. SpTtm on CPUs

Based on COO and sCOO formats, we implement sequential SpTtm by directly operating on non-zero entries without explicit transformation between a tensor and a matrix.

Given a sparse tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ and a dense matrix $\mathbf{U} \in \mathbb{R}^{I_n \times R}$, we know the resulting tensor $\mathcal{Y}$ is a semi-sparse tensor from the above property (Section 3.3). The intuitive algorithm for SpTtm without explicit transformation is to loop all non-zeros of $\mathcal{X}$ by multiplying each with its corresponding row of $\mathbf{U}$. Then all rows having the same index pair $(i_1, \ldots, i_{n-1}, i_{n+1}, \ldots, i_N)$ are sum-reduced to get a fiber of $\mathcal{Y}$ ($\mathbf{f}_n^Y$).

This algorithm has two issues: First, in the sum-reduction stage, there is an implicit index comparing operation even if $\mathcal{X}$ is pre-sorted. The complexity of the index comparison is high especially for higher-order tensors. An extra comparison operation for an additional mode increases the SpTtm complexity by nnz^X, which is non-trivial especially for low-rank tensor decompositions with a small R. Second, the sum-reduction stage is hard to parallelize and may lead to severe memory contention.

To solve these problems, we design our sequential SpTtm algorithm (Algorithm 1) to avoid expensive index comparison and with inherent good parallelism. Each mode-n fiber of $\mathcal{Y}$ ($\mathbf{f}_n^Y$) is a sized-R dense vector, we record n_{fibs} as the number of $\mathbf{f}_n^Y$. Then the number of non-zeros of $\mathcal{Y}$: $nnz^Y = n_{fibs} \times R$. We use an extra array f_{ptr} to identify the beginning locations of every mode-n fiber of $\mathcal{X}$ ($\mathbf{f}_n^X$), then iterate all n_{fibs} fibers of $\mathcal{Y}$. Comparing to iterating $\mathcal{X}$ in the intuitive algorithm, we avoid the expensive index comparison in the sum-reduction stage.

Our SpTtm has two stages, preprocessing and computing. The preprocessing stage includes three steps: sorting $\mathcal{X}$, calculating $fptr$, and pre-allocating semi-sparse tensor $\mathcal{Y}$. $\mathcal{X}$ is first sorted according to the product mode n, then f_{ptr} is allocated and calculated to identify the beginning locations of n_{fibs} mode-n fibers of $\mathcal{X}$ ($\mathbf{f}_n^X$). From SpTTM's property (Section 3.3), the semi-sparse tensor's index modes remain unchanged, so the number of $\mathbf{f}_n^Y$ is also n_{fibs}. Based on the pre-sorted $\mathcal{X}$, we pre-allocate exact space for $\mathcal{Y}$ and only for its non-zero entries, because $\mathcal{X}$'s $(N-1)$ indices can be reused by $\mathcal{Y}$. During the computation stage of Algorithm 1, each $\mathbf{f}_n^Y = val^Y(i, :)$ locates the corresponding $\mathbf{f}_n^X = val^X(f_{ptr}(i), \ldots, f_{ptr}(i+1) - 1)$. Then $\mathbf{f}_n^Y$ is the sum of rows $\mathbf{u}(k, :)$ scaled by each non-zero of fiber $\mathbf{f}_n^X$.

Algorithm 1: CPU sequential SpTtm algorithm (SEQ-SpTtm).

```
Input: A sparse tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$, a dense matrix $\mathbf{U} \in \mathbb{R}^{I_n \times R}$, and
    an integer $n$;
Output: A semi-sparse tensor $\mathcal{Y} \in \mathbb{R}^{I_1 \times \cdots \times I_{n-1} \times R \times I_{n+1} \cdots \times I_N}$;
 1: $n_{fibs}$: the number of mode-n fibers of $\mathcal{Y}$
 2: $f_{ptr}$: the beginnings of each $\mathcal{X}$ mode-n fiber, size $n_{fibs}$.
 3: for $i = 0, \ldots, n_{fibs}$ do
 4:     for $j = f_{ptr}(i), \ldots, f_{ptr}(i+1) - 1$ do
 5:         $k = ind_n^X(j)$
 6:         for $r = 0, \ldots, R-1$ do
 7:             $val^Y(i, r) += val^X(j) \times u(k, r)$
 8:         end for
 9:     end for
10: end for
11: Return $\mathcal{Y}$;
```

The number of floating-point operations (flops) of sequential SpTtm (Algorithm 1) is

$$flops = 2 \cdot nnz^X \cdot R, \tag{3}$$

where nnz^X is the number of non-zeros of $\mathcal{X}$. Algorithm 1 eliminates the index comparison. For a third-order sparse tensor, our SpTtm only uses n_{fibs} extra space for $fptr$. Since $n_{fibs} \leq nnz^X$, the extra space is much smaller than the matricized tensor of $\mathcal{X}$ ($3 \cdot nnz^X$) in the traditional algorithms where tensor transformation is needed.

We parallelize SpTtm on the multicore CPU architecture using OpenMP. Since our sequential SpTtm iterates all independent fibers of $\mathcal{Y}$, we can easily parallelize this loop by assigning CPU threads, i.e., parallelize the outermost for loop. Each thread computes a size-R fiber $\mathbf{f}_n^Y$ independently and shares dense matrix $\mathbf{U}$. Because our SpTtm algorithm limits the sum-reduction dependency inside a thread, parallelized SpTtm naturally avoids locks and utilizes CPU caches well for writing $\mathcal{Y}$. We use CPU-SpTtm to represent parallel CPU SpTtm implementation in the following contents.

4.2. SpTtm on GPUs

To fully explain our optimizations on GPU algorithm, we first start by describing a naïve implementation, then propose four more parallelization and optimization approaches for SpTtm by incrementally considering GPU architecture characteristics. The five implementations are: naïve implementation, employing fine thread granularity, arranging coalesced memory access, rank blocking, and using fast shared memory.

In this work, we assume the sparse tensor $\mathcal{X}$ and the dense matrix $\mathbf{U}$ both reside in GPU memory. We assume all the division operations below are fully divisible.

4.2.1. Naïve implementation

As shown in Algorithm 2, we assign each CUDA thread to one mode-n fiber of $\mathcal{Y}$ ($\mathbf{f}_n^X$) and a fiber of $\mathcal{X}$ ($\mathbf{f}_n^X$). Each thread performs multiplication on every non-zero of fiber $\mathbf{f}_n^X$ with its corresponding rows of $\mathbf{u}(k,:)$. When launching the kernel, we set $dimGrid = N_{TB}$ and $dimBlock = n_{fibs}/N_{TB}$, where N_{TB} is the number of threads per block. We tune the value of N_{TB} for the best performance.

Algorithm 2: Naïve SpTTM algorithm on GPU (GPU-SpTtm-Naïve).

```
Input: A sparse tensor $\mathcal{X} \in \mathbb{R}^{I_1\times I_2\times\ldots\times I_N}$, a dense matrix $\mathbf{U} \in \mathbb{R}^{I_n\times R}$, an
    integer $n$, the beginnings of $n_{fibs}$ $\mathcal{X}$ mode-n fiber $f_{ptr}$, and GPU thread
    hierarchy $dimGrid = N_{TB}$ and $dimBlock = n_{fibs}/N_{TB}$;
Output: A semi-sparse tensor $\mathcal{Y} \in \mathbb{R}^{I_1\times\ldots\times I_{n-1}\times R\times I_{n+1}\ldots\times I_N}$;
1: $tidx = threadIdx.x$;
2: $i = blockIdx.x \times blockDim.x + tidx$;    ▷ $i$: global index of a mode-n $\mathcal{Y}$
   fiber.
                    ▷ $j$: global index of the non-zeros of mode-n $\mathcal{X}$ fiber.
3: for $j = f_{ptr}(i), \ldots, f_{ptr}(i+1)-1$ do
4:     $k = ind_n^X(j)$
5:     for $r = 0, \ldots, R-1$ do
6:         $val^Y(i,r) += val^X(j) \times u(k,r)$
7:     end for
8: end for
9: Return $\mathcal{Y}$;
```

Some inefficient spots are observed:

- Uncoalesced and redundant memory transfers. The memory transfers for matrix $\mathbf{U}$ are in strided-R pattern, and the data transferred are much larger than the size of $\mathbf{U}$ because of irregular memory access.
- Coarse-grained task granularity. Algorithm 2 assigns a fiber per CUDA thread to do f_{len} computations with sized-R rows of $\mathbf{U}$, where f_{len} is the average length of the fibers. For lightweight GPU threads, it might be better to allocate fine-grained tasks for each thread.
- Not utilizing fast memory. Algorithm 2 accesses data only from global memory without well utilized fast scratch-pad memory (shared memory or L1 cache) for writable data, since NVIDIA Kepler and later GPUs cannot use L1 cache for writable data automatically.

To deal with the inefficiency, we propose optimizations described below.

4.2.2. Fine thread granularity

Instead of one-dimensional thread blocks, we assign two-dimensional thread blocks, so both rows and columns of $\mathbf{U}$ are parallelized in Algorithm 3. However, if the matrix row size R is a little large, say 128, the number of threads in x-dimension can be only up to 8, which leads to imbalanced parallelism for x and y dimensions. To prevent this issue, we set a threshold N_{TR} to limit the number of threads allocated to each matrix column, and compute a segment of a column in one iteration.

When launching Algorithm 3, we set $dimBlock = (N_{TB}/N_{TR}, N_{TR})$ and $dimGrid = n_{fibs}/N_{TB}$.

Algorithm 3: SpTtm algorithm with fine thread granularity on GPU (GPU-SpTtm-FG).

```
Input: A sparse tensor $\mathcal{X} \in \mathbb{R}^{I_1\times I_2\times\ldots\times I_N}$, a dense matrix $\mathbf{U} \in \mathbb{R}^{I_n\times R}$, an
    integer $n$, the beginnings of $n_{fibs}$ $\mathcal{X}$ mode-n fiber $f_{ptr}$, and GPU thread
    hierarchy $dimGrid = n_{fibs}/N_{TB}$ and $dimBlock = (N_{TB}/N_{TR}, N_{TB})$;
Output: A semi-sparse tensor $\mathcal{Y} \in \mathbb{R}^{I_1\times\ldots\times I_{n-1}\times R\times I_{n+1}\ldots\times I_N}$;
1: $n_{loops}^r = \frac{R}{blockDim.y}$                    ▷ #Iterations for large $R$.
2: $tidx = threadIdx.x$;
3: $tidy = threadIdx.y$;
4: $i = blockIdx.x \times blockDim.x + tidx$;    ▷ $i$: global index of a mode-n $\mathcal{Y}$
   fiber.
                    ▷ $j$: global index of the nonzeros of mode-n $\mathcal{X}$ fiber.
5: for $j = f_{ptr}(i), \ldots, f_{ptr}(i+1)-1$ do
6:     $k = ind_n^X(j)$
7:     for $l_r = 0, \ldots, n_{loops}^r$ do
8:         $r = tidy + l_r \times blockDim.y$
9:         $val^Y(i,r) += val^X(j) \times u(k,r)$
10:    end for
11: end for
12: Return $\mathcal{Y}$;
```

4.2.3. Coalesced memory access

From Algorithm 3, we observe a memory access pattern where the memory access of tensor indices and values are contiguous and coalesced but that of matrix $\mathbf{U}$ is not. In Algorithm 3, threads in a warp, $(0,0), (1,0), \ldots, (32,0)$ (x-dimension dominates), fetch elements from random addresses because of different k indices, which leads to an uncoalesced memory access. To solve this problem, we swap thread dimensions x and y, thus x-dimension points to ranks (columns of $\mathbf{U}$) and y-dimension operates on fibers of $\mathcal{X}$ and $\mathcal{Y}$. The algorithm is named GPU-SpTtm-MC.

When launching this algorithm, we set $dimBlock = (N_{TR}, N_{TB}/N_{TR})$ and $dimGrid = n_{fibs}/N_{TB}$. Therefore, one warp fetches coalesced global memory addresses of tensor indices, values, and matrix elements.

4.2.4. Rank blocking

The above algorithms consider the spatial data locality within a fiber, but not the temporal locality between fibers. In Algorithm 3, different fibers may have the same k index, which means accessing the same row of $\mathbf{U}$. Rank blocking strategy is proposed to increase the reuse of row data. Since the computation in-between matrix columns are independent, we exchange the loop order to ensure the loop over ranks (matrix columns) goes before the loop over fiber elements. This strategy enlarges the chance of short rows staying in caches.

Its algorithm (named GPU-SpTtm-RB) can be obtained by swapping $tidx$ and $tidy$ usage and the two loops in Algorithm 3. However, for each batch of short size-N_{TR} matrix rows, the index ks and values of the tensors need to be reloaded from global memory. The benefit of rank blocking depends on the non-zero distribution of the input tensor, it is not easy to determine whether rank blocking is beneficial compared with the above memory coalesced algorithm (GPU-SpTtm-MC).

4.2.5. Using shared memory

Obviously, the output val^Y is reused R times for each fiber of $\mathcal{X}$. As we mentioned, the writable val^Y of $\mathcal{Y}$ cannot be cached in the L1 cache for Kepler and later NVIDIA GPU architectures,

Algorithm 4: SpTTM algorithm using GPU shared memory (GPU-SpTTM-SM).

```
Input: A sparse tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \dots \times I_N}$, a dense matrix $\mathbf{U} \in \mathbb{R}^{I_n \times R}$, an
    integer $n$, the beginnings of each $\mathcal{X}$ mode-n fiber $f_{ptr}$, sized $n_{fibs}$, and
    GPU thread hierarchy $dimGrid = N_{TB}$ and $dimBlock = (N_{TR}, N_{TB}/N_{TR})$;
Output: A semi-sparse tensor $\mathcal{Y} \in \mathbb{R}^{I_1 \times \dots \times I_{n-1} \times R \times I_{n+1} \dots \times I_N}$;
 1: $\mathbf{y_{shr}}$: shared memory space for $\mathcal{Y}$.
 2: $n^r_{loops} = \frac{R}{blockDim.x}$
 3: $tidx = threadIdx.x$;
 4: $tidy = threadIdx.y$;
 5: $i = blockIdx.x \times blockDim.y + tidy$;    ▷ $i$: global index of a $\mathcal{Y}$ mode-n
    fiber.
 6: for $l_r = 0, \dots, n^r_{loops}$ do
 7:     $r = tidx + l_r \times blockDim.x$
 8:     $y_{shr}(tidy, tidx) = 0$;
 9:     $\_\_sync()$;
                ▷ $j$: global index of the nonzeros of $\mathcal{X}$ mode-n fiber.
10:     for $j = f_{ptr}(i), \dots, f_{ptr}(i+1) - 1$ do
11:         $k = ind^X_n(j)$
12:         $y_{shr}(tidy, tidx) += val^X(j) \times u(k, r)$
13:     end for
14:     $\_\_sync()$;

15:     $val^Y(i, r) = y_{shr}(tidy, tidx)$;
16:     $\_\_sync()$;
17: end for
18: Return $\mathcal{Y}$;
```

but can only reside in the slower L2 cache. Therefore, we store the output in shared memory first and then write them back to global memory all at once. In this way, we reduce global memory transfers. Its algorithm is shown in Algorithm 4. , When launching this algorithm, we set $dimBlock = (N_{TR}, N_{TB}/N_{TR})$, $dimGrid = n_{fibs}/N_{TB}$ and set shared memory size $S_{SM} = N_{TB}$ words because the size of $yshr$ is $dimBlock.y \times dimBlock.x$. Since $nT_B \leq 1024$ for the GPUs we used, the needed shared memory space is smaller than 8KB. That means only the configuration of 48 KB L1 cache/16 KB shared memory is used for Algorithm 4.

5. Semi-Sparse tensor times matrix

SsᴘTᴛᴍ is defined as the Tᴛᴍ product of a semi-sparse tensor and a dense matrix, the former being the result of an SᴘTᴛᴍ. Although it is possible to convert the semi-sparse tensor back to fully sparse so that SsᴘTᴛᴍ can be performed with the above SᴘTᴛᴍ algorithm, we propose a tailored SᴘTᴛᴍ algorithm optimized specially for semi-sparse tensors.

Given a semi-sparse tensor $\mathcal{Y} \in \mathbb{R}^{I_1 \times I_2 \times \dots \times I_N}$ with $D_{\mathcal{Y}} \subset \{1, 2, \dots, N\}$ being a set of dense modes of $\mathcal{Y}$ and a dense matrix $\mathbf{U} \in \mathbb{R}^{I_n \times R}$, requiring $n \notin D_{\mathcal{Y}}$, the SsᴘTᴛᴍ algorithm also computes $\mathcal{Z} = \mathcal{Y} \times_n \mathbf{U}$. We use $S_{\mathcal{Y}}$ to represent the sparse modes of $\mathcal{Y}$, $S_{\mathcal{Y}} = \{1, 2, \dots, N\} - D_{\mathcal{Y}}$.

Analogous to SᴘTᴛᴍ, our SsᴘTᴛᴍ also has two stages, preprocessing and computing. To begin with, we sort values and indices of $\mathcal{Y}$ in a specific mode order s.t. $S_{\mathcal{Y}} \prec n \prec D_{\mathcal{Y}}$. For example, if modes 1, 3, 5 are sparse, mode 2,4 is dense, $1 \prec 5 \prec 3 \prec 2 \prec 4$ would be a valid sorting order for an SsᴘTᴛᴍ in mode 3. The purpose of this sorting is to gather elements sharing identical sparse mode indices together in a dense "chunk". In the example above, a chunk of $\mathcal{Y}$ is a dense array of size $I_2 \times I_4$, which is a generalization of fibers in SᴘTᴛᴍ algorithms. Based on the sparse tensor property (Section 3.3), $\mathcal{Z}$ will have the shape of $I_3 \times I_2 \times I_4$ for all chunks by converting mode 3 to a dense mode and enlarging its chunks.

However, different from SᴘTᴛᴍ, SsᴘTᴛᴍ in Tucker decomposition does not need the expensive sorting step of preprocessing stage. In the Tucker decomposition, an $(N-1)$-Tᴛᴍ-chain (a sequence of Tᴛᴍs, refer to Section 6) can be ordered the same with the sorting order of the input sparse tensor $\mathcal{X}$, and an SᴘTᴛᴍ or SsᴘTᴛᴍ keeps the same order for their sparse modes. Then, only one sorting per Tᴛᴍ-chain for a sparse tensor $\mathcal{X}$ is needed for the first SᴘTᴛᴍ. The input semi-sparse tensors for the following SsᴘTᴛᴍs remain sorted. For an N-mode Tucker decomposition, there are N different sorting orders throughout the entire algorithm, which can be reused among iterations. We cache the N preprocessed tensors in CPU memory and transfer them to GPU memory when necessary to speed up the preprocessing process and save the precious GPU memory.

The CPU parallel algorithm of SsᴘTᴛᴍ is Algorithm 5, where the outmost loop-i is parallelized to launch multiple OpenMP threads. For SsᴘTᴛᴍ GPU implementations, we map a CUDA thread block into a chunk, because the access pattern of the values within a $\mathcal{Z}$ chunk only depends on the continuously stored $\mathcal{Y}$ chunks from the previous sorting stage, so inter-chunk calculations can be safely parallelized without data race. In a Tucker decomposition application, where a typical $R = 16$, the number of threads per block can vary from 16 to 65 536. The first Tᴛᴍ in the Tucker decomposition is an SᴘTᴛᴍ, the rest SsᴘTᴛᴍs handle CUDA block sizes in a typical range of 256–65 536, large enough to make full use of the Streaming Multiprocessors. We adjust the block size according to CUDA's restriction of 1024 threads per block, by calculating chunks in batch. The amount of values per block is small enough to fit into L1/L2 caches so we do not apply shared memory to this algorithm.

Algorithm 5: CPU parallel SsᴘTᴛᴍ algorithm (CPU-SsᴘTᴛᴍ).

```
Input: A semi-sparse tensor $\mathcal{Y} \in \mathbb{R}^{I_1 \times I_2 \times \dots \times I_N}$ with dense modes $D_{\mathcal{Y}}$, a
    dense matrix $\mathbf{U} \in \mathbb{R}^{I_n \times R}$, and an integer $n$;
Output: A semi-sparse tensor $\mathcal{Z} \in \mathbb{R}^{I_1 \times \dots \times I_{n-1} \times R \times I_{n+1} \dots \times I_N}$;
 1: $n^Z_{chunks}$: the number of chunks in $\mathcal{Z}$
 2: $s^Y_{chunk}$: the size of a chunk in $\mathcal{Y}$
 3: $s^Z_{chunk}$: the size of a chunk in $\mathcal{Z}$
 4: $c_{ptr}$: the beginnings of each chunk of $\mathcal{Y}$, size $n^Z_{chunks}$.
 5: parfor $i = 0, \dots, n^Z_{chunks} - 1$ do
 6:     for $j = c_{ptr}(i), \dots, c_{ptr}(i+1) - 1$ do
 7:         $r = ind^Y_n(j)$;
 8:         for $c = 0, \dots, R - 1$ do
 9:             for $k = 0, \dots, s^Y_{chunk} - 1$ do
10:                 $val^Z(i \times s^Z_{chunk} + r \times s^Y_{chunk} + k) += val^Y(j \times s^Y_{chunk} + k) \times$
    $u(r, c)$;
11:             end for
12:         end for
13:     end for
14: end parfor
15: Return $\mathcal{Z}$;
```

6. Tucker decomposition

Based on our optimizations of SᴘTᴛᴍ and SsᴘTᴛᴍ, we design Tucker decomposition for a heterogeneous CPU–GPU platform.

6.1. Tucker-ALS

Tucker decomposition approximates a tensor as a product of a small core tensor and a set of factor matrices: $\mathcal{X} \approx \mathcal{G} \times_1 \mathbf{U}_1 \times \mathbf{U}_2 \times \dots \times \mathbf{U}_N$, where $\mathcal{X} \in \mathbb{R}^{I_1 \times \dots \times I_N}$, $\mathcal{G} \in \mathbb{R}^{R_1 \times \dots \times R_N}$ is the core tensor and $\mathbf{U}_i \in \mathbb{R}^{I_i \times R_i}, i = 1, \dots, N$ are factor matrices. The number of columns of each factor matrix is also called a rank of Tucker decomposition, say rank-$(R_1, \dots, R_N)$ tensor decomposition. $R_i, i = 1, \dots, N$ is usually small for low-rank decomposition, and they can be different from each other.

Table 2
Experimental platforms configuration.

Parameters	Intel	NVIDIA	
	Xeon CPU E5-2650 v4	Tesla K40c	Tesla P100
Microarchitecture	Broadwell-EP	Kepler	Pascal
Frequency	2.2 GHz	0.75 GHz	0.72 GHz
#Physical cores	12	2880	3584
Peak SP performance	845 GFlop/s	4290 GFlop/s	9300 GFlop/s
Last-level cache	30 MB	1.6 MB	4 MB
Memory size	792 GB	12 GB	16 GB
Memory bandwidth	77 GB/s	288 GB/s	732 GB/s
Compiler	gcc 5.4.1	nvcc 9.0	nvcc 9.0

Higher-Order Orthogonal Iteration (HOOI) algorithm [18], as a popular Tucker decomposition algorithm, is an iteration algorithm with each iteration updating all factor matrices once at a time. In this paper, we study HOOI based on the alternating least squares paradigm (Algorithm 6). HOOI algorithm takes random factor matrices or a decomposition produced by Higher-Order SVD (HOSVD) [17] as the initial settings and iterates until a pre-defined convergence rate satisfied or exhausting the maximum number of iterations. For an iteration, each factor matrix is updated by a TTM-chain and an SVD operation in its corresponding mode. After all factor matrices are acceptable, the core tensor $\mathcal{G}$ is computed. Therefore, TTM-chain and SVD are the dominant computations of the Tucker decomposition.

Algorithm 6: Tucker decomposition algorithm (HOOI) with ALS.

```
Input: A sparse tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \dots \times I_N}$, and a set of ranks $R_1, \dots, R_N$;
Output: Factor matrices $\mathbf{U}_1, \dots, \mathbf{U}_N$ and a core tensor $\mathcal{G}$;
1: Initialize $\mathbf{U}_n \in \mathbb{R}^{I_n \times R}$, $n = 1, \dots, N$ randomly or using HOSVD
2: do
3:     for $n = 1, \dots, N$ do
4:         $\mathcal{Y} \leftarrow \mathcal{X} \times_1 \mathbf{U}_1 \dots \times_{(n-1)} \mathbf{U}_{(i-1)} \times_{(n+1)} \mathbf{U}_{(n+1)} \dots \times_N \mathbf{U}_N$
5:         $\mathbf{U}_n \leftarrow R_n$ leading left singular vectors of $\mathcal{Y}_{(n)}$
6:     end for
7: while fit ceases to improve or maximum iterations exhausted
8: $\mathcal{G} \leftarrow \mathcal{X} \times_1 \mathbf{U}_1 \dots \times_N \mathbf{U}_N$
9: Return $\mathcal{G}, \mathbf{U}_1, \dots, \mathbf{U}_N$;
```

6.2. TTM-Chain

The TTM-chain for a sparse input tensor consists of two types of TTMs—SPTTM and SSPTTM. The first TTM in Line 4 of Algorithm 6 is an SPTTM, and the following TTMs are all SSPTTMs. We use our fastest CUDA implementations of them to accelerate the TTM-chains. The output tensor of every TTM (SPTTM or SSPTTM) resides in GPU memory, thus no memory transfer is needed inside a TTM-chain.

6.3. SVD

According to the property in Section 3.3, the output $\mathcal{Y}$ of TTM-chain is dense in all except mode n. We treat it as a dense tensor, then $\mathbf{Y}_{(n)}$ is a dense matrix. For the SVD operation, we employ svd function ("`sgesvd`") from OpenBLAS library [47] on CPUs, while use SVD function ("`cusolverDnSgesvd`") from cuSOLVER [15]. However, from our experiments, the SVD function of cuSOLVER is not as efficient as OpenBLAS version, mainly because of the irregular shape of the matricized $\mathbf{Y}_{(n)}$.

7. Experiments

We test our algorithms on three platforms, one Intel multicore CPU platform and two GPU platforms. Our SPTTM performance is compared with state-of-the-art Tensor Toolbox [28] library and a recent sparse TTM implementation [43] on GPUs. We also analyze our GPU optimization methods incrementally and the performance behavior by varying product modes and rank sizes.

Table 3
Description of sparse tensors.

Tensors	Order	Mode sizes	NNZ	Density
`choa`	3	$712K \times 10K \times 767$	27M	5.0×10^{-06}
`nell2`	3	$12K \times 9K \times 29K$	77M	1.3×10^{-05}
`fb-m`	3	$23M \times 23M \times 166$	100M	1.1×10^{-09}
`fb-s`	3	$39 \times 39 \times 532$	140M	1.7×10^{-10}
`deli`	3	$533K \times 17M \times 2M$	140M	6.1×10^{-12}
`nell1`	3	$3M \times 2M \times 26M$	144M	9.1×10^{-13}
`nips`	4	$2K \times 3K \times 14K \times 17$	3M	1.8×10^{-06}
`uber`	4	$183 \times 24 \times 1140 \times 1717$	3M	3.8×10^{-04}
`enron`	4	$6K \times 6K \times 244K \times 1K$	54M	5.5×10^{-09}
`flickr`	4	$320K \times 28M \times 2M \times 731$	113M	1.1×10^{-14}
`deli4d`	4	$533K \times 17M \times 2M \times 1K$	140M	4.3×10^{-15}

7.1. Platforms and dataset

We use Intel Xeon CPU E5-2650 v4 and NVIDIA Tesla K40c and P100 platforms, their configurations are shown in Table 2. NVIDIA Tesla K40c and P100 have much higher peak floating-point performance and memory bandwidth than the Intel Xeon CPU. All experiments perform single-precision floating-point values and the performance numbers are averaged over five runs. Without further specification, R is set to 16.

We use third-order and fourth-order sparse tensors from real applications, collected in FROSTT [52] and HaTen2 [24]. Tensors consist of medical data from Children's Healthcare of Atlanta project (`choa` with patient-medication-diagnosis), Never Ending Language Learning (NELL) project [11] (`nell1` and `nell2` with noun-verb-noun), Freebase RDF data [24] "freebase-music" and "freebase-sampled" (`fb-m` and `fb-s` with entity-entity-relation), data crawled from tagging systems [20] (`deli` with user-item-tag and `deli4d` with user-item-tag-date), Enron emails (`enron` with sender-receiver-word-date), Uber pickup data in 2016 (`uber` with date-hour-latitude-longitude), NIPS papers published from 1987 to 2003 (`nips` with id-author-word-year), and tags from Flickr (`flickr` with user-item-tag-date). The tensor property is shown in Table 3. Please refer to the original datasets [24,52] for further information.

7.2. Overall performance

We first show the speedups of our GPU-SPTTM algorithms over the OpenMP parallelized CPU-SPTTM in Fig. 4 and compare their performance to FCOO-SPTTM performance [43] on the GPU P100 platform.[3] The speedup numbers of each tensor are averaged among all its modes. SPTTM GPU performance is shown using the best one of all the five GPU implementations in Section 4.2. GPU-SPTTM achieves up to 67× speedups over CPU-SPTTM. Compared to the state-of-the-art work [43], our GPU-SPTTM implementations overperform FCOO-SPTTM by up to 3.9×. From our experiments, the best speedup of GPU-SPTTM is mostly obtained by GPU-SPTTM-SM (Algorithm 4), which verifies our optimizations. The detailed analysis on the five approaches will be given afterward. CPU-SPTTM's performance is obtained by using 12 threads.

An interesting phenomenon is the speedup difference between tensor `deli` and `deli4d`. From Table 3, `deli4d` has the same number of non-zeros with `deli`, but with an extra "date" mode, whereas GPU-SPTTM achieves higher speedup on the fourth-order `deli4d`. One main reason is that of the load imbalance between CUDA threads incurred by different fiber lengths of the input tensor

[3] We do not compare with Tensor Toolbox [5] and CTF [55] because they lack GPU parallel implementations for sparse tensors.

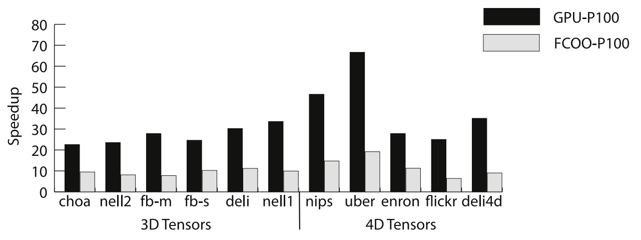

Fig. 4. Our GPU-SpTTM and FCOO-SpTTM [43] speedups over CPU-SpTTM.

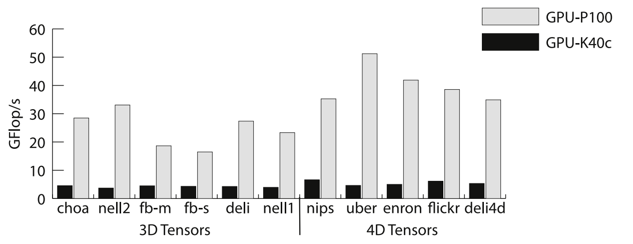

Fig. 5. GPU-SpTTM performance in GFlop/s.

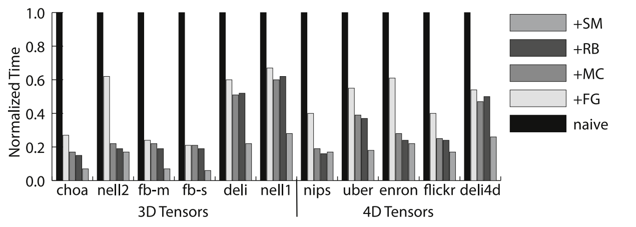

Fig. 6. GPU optimization methods comparison on GPU P100.

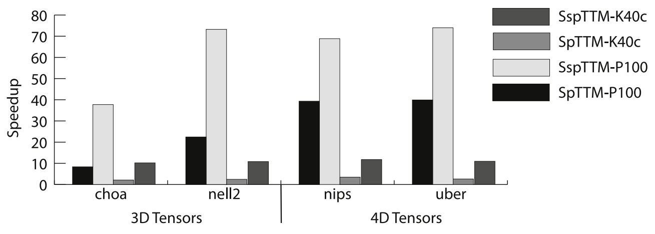

Fig. 7. GPU-SpTTM and GPU-SspTTM speedups over corresponding OpenMP parallelized CPU implementations on two NVIDIA GPU platforms.

Table 4
Sequential SpTTM performance comparison.

Time (s)	Tensor Toolbox	SEQ-SpTTM
`choa`	7.39	2.78
`nell2`	86.50	7.33
`nips`	120.07	0.58

$\mathcal{X}$. We calculate the standard deviation of fiber lengths for each tensor mode. `deli4d` has a standard deviation 2.54×10^4 about a half magnitude less than `deli`'s 1.21×10^5.

Fig. 5 gives the actual SpTTM floating-point operations per second (flop/s) of "GPU-P100" and "GPU-K40c". "GPU-P100" achieves better performance, 16–51 GFlop/s, while "GPU-K40c" obtains less than 10 GFlop/s. The performance numbers obtained in this work on P100 are better than the FCOO-SpTTM [43] and sparse matrix-dense matrix multiplication [40]. However, compared to either the peak machine performance or the attainable performance limited by memory bandwidth, our performance is far below these bounds. There is space for further performance tuning.

7.3. Analysis

This work is analyzed from different aspects: incremental GPU optimization effects, sequential algorithm comparison to show the advantage of our SpTTM algorithm, SspTTM and SpTTM comparison for semi-sparse tensors, mode behavior, and rank behavior. After all, we apply our optimized algorithms to Tucker decomposition.

7.3.1. GPU optimization comparison

We analyze the five GPU SpTTM approaches in Section 4.2: naïve implementation, fine thread granularity, coalesced memory access, rank blocking, and using shared memory, in Fig. 6 on a GPU P100. The times are normalized to the naïve GPU implementation and are averaged over all modes for every tensor. As the optimization methods incrementally applied to SpTTM, we tend to get shorter execution time. Naïve SpTTM is the simplest and slowest GPU implementation. SpTTM performance is incrementally improved: 53% by integrating fine thread granularity, 29% by arranging coalesced memory access, 7% by rank blocking, and 40% by using shared memory for writable data, on average of all tensors. Therefore, fine thread granularity and using shared memory optimizations are the two most effective optimizations for SpTTM, arranging coalesced memory access also improves some performance, while the effect of rank blocking depends on the given sparse tensors. We also test these approaches by setting R to 32. SpTTM performance is incrementally improved: 74% (+FG), 37% (+MC), −20% (+RB), and 43% (+SM), on average of all tensors. This shows rank blocking is more beneficial for small ranks, because of the re-loads of index ks and values, which coincides with our analysis in Section 4.2.

7.3.2. Sequential SpTTM comparison

We test the SpTTM from Tensor Toolbox [5] on only three small tensors without exceeding memory in Table 4.[4] Our SEQ-SpTTM achieves 3–120× speedup over Tensor Toolbox. One reason is Tensor Toolbox is built on MATLAB environment, this may generate some extra overhead. For the fourth-order tensor `nips`, though it only has 3 million nonzeros, Tensor Toolbox runs much slower than we expect. This shows Tensor Toolbox may be not friendly enough for higher-order tensors.

Table 5 shows the storage of Tensor Toolbox compared to our SpTTM in mode 1. Tensor Toolbox consumes about twice storage space than SpTTM.

7.3.3. SspTTM v.s. SpTTM for Semi-Sparse Tensors

We compare the performance of GPU-SspTTM with GPU-SpTTM for semi-sparse tensors in Fig. 7. We use the second TTM operation in the TTM-chain of Tucker decomposition (Algorithm 6) to ensure the input tensor is semi-sparse. SpTTM handles the semi-sparse input tensor as a general sparse tensor. Since the input semi-sparse tensors could be easily as large as the tensors in Table 3 because of the introduced dense mode. We can only run SpTTM for this operation on small tensors. GPU-SspTTM achieves up to 74× and 12× speedups on GPU P100 and K40c platforms respectively. Compared to SpTTM, which treats semi-sparse tensors as general sparse tensors, SspTTM achieves up to 4.5× speedup. This figure shows the usefulness of the SspTTM algorithm for semi-sparse tensors.

7.3.4. Mode behavior

We compare the "GPU-P100" SpTTM behavior in different modes, using the best execution time of the five approaches. Fig. 8

[4] Since Cyclops Tensor Framework (CTF) [55] indexes a sparse tensor using a one-dimensional long index to help distribute nonzero elements in a certain way in distributed platforms, all tensors we tested cannot be represented after vectorization, so they cannot run using CTF except in distributed memory environment.

Table 5
Total storage (GB) of sparse tensors.

Tensors	choa	nell2	fb-m	fb-s	deli	nell1	nips	uber	enron	flickr	deli4d
TTBox	2.72	2.30	30.35	44.90	20.69	11.80	0.40	0.09	1.77	13.81	46.07
SpTtm	1.55	1.32	17.34[a]	25.65[a]	11.82	6.74	0.23	0.05	1.01	7.89	19.75[a]

[a]Since an Ttm only needs one index instead of all, these tensors can fit into GPU memory.

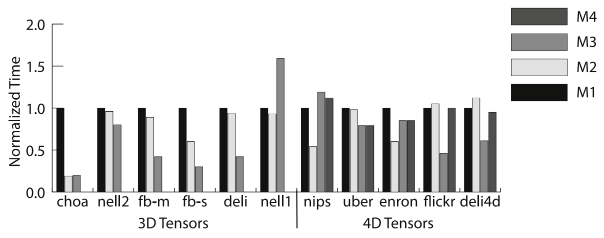

Fig. 8. Relative SpTtm time in different modes.

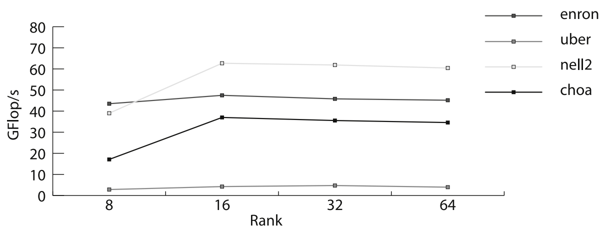

Fig. 9. Execution time of small tensors in different rank sizes.

Table 6
Tucker decomposition performance.

	CPU (s)	GPU (s)
choa	49.3	15.6
nell2	48.6	16.7
enron	389.6	290.3

takes mode-1 SpTtm as the baseline, then computes the SpTtms in other modes normalized to it. Some tensors have very diverse SpTtm performance in different modes. From our observations, the modes in which fibers' length has a small standard deviation tend to be faster compared to other modes. For example, tensor choa has small fiber sizes in mode 2 and 3, so their SpTtm performance is much better than that in mode 1.

7.3.5. Rank behavior

Fig. 9 shows the relative performance of mode-1 "GPU-P100" SpTtm on tensors choa, nell2, uber, and enron by increasing the rank-size. We only test on small tensors because when rank-size is 32 or 64, some tensors are too large to reside in GPU memory. As the rank-size increases, the performance increases for R from 8 to 16 but stops increasing after 16.

7.3.6. Tucker decomposition

Table 6 shows the execution time of the Tucker decomposition by applying our optimized SpTtm and SspTtm algorithms. Due to space limitation, large tensors cannot run on GPU platforms due to the non-trivial intermediate results. The optimized Tucker decomposition achieves up to 3.2× speedup. From Tucker decompositions, we observed that SVD and Ttm-chain performance are both important to Tucker performance. We test SVD by using OpenBLAS and cuSOLVER. However, SVD of cuSOLVER cannot achieve as good performance as OpenBLAS, especially for truncated SVD. Therefore, we use OpenBLAS to solve SVD and include the memory transfer time into our GPU Tucker implementation.

8. Related work

Several other libraries also implemented fundamental sparse tensor operations, such as Tensor Toolbox [5] and Cyclops Tensor Framework (CTF) [55]. Tensor Toolbox is implemented in MATLAB environment, and CTF is implemented in C++ language. From algorithm aspect, Tensor Toolbox implements SpTtm by converting a sparse tensor into a sparse matrix and converting the matrix back to a sparse/dense tensor after a sparse matrix and a dense matrix multiplication. The conversion consumes non-trivial time. CTF is a parallel framework of tensor contraction for distributed CPU systems, and it indexes a sparse tensor using a one-dimensional long index. Even for a reasonable large sparse tensor, CTF may exceed the range of 64-bit long unsigned integer. Our in-place SpTtm algorithms directly operate on the coordinate format of a sparse tensor, which avoids the indexing problem, and the resulting tensor is stored in a hybrid format (sCOO) to save space.

Some work proposed efficient storage formats for a sparse tensor, such as the Compressed Sparse Fiber (CSF) format [53] and the "mode-generic sparse storage format" [7]. Our sCOO format is a simple version of the "mode-generic sparse storage format" proposed by Baskaran et al., with dense modes fixed on the last several modes. CSF is a hierarchical, fiber-centric format by extending the popular CSR format to sparse tensors, and it is memory-efficient and shows high speedups on the Mttkrp and Ttm operations over the COO format. However, when operating on a non-root mode, the recursive algorithm based on CSF format may be not suitable for GPU architecture. Another recent work [43] proposed F-COO format and implemented Ttm operations based on it on GPUs. Our optimized SpTtm uses the most popular COO format and achieves better performance. Besides, it is the first try by implementing Tucker decomposition on GPUs.

Some work on sparse matrix formats [8,12,19,29,31,32,36,37,42,56,57,60–62,64] and sparse matrix–matrix multiplication [3,6,9,10,16,27,38–41,44,58,63] is also related to our work. However, our focus is to avoid unnecessary transformation, which is a problem only for tensors. The optimization methods of sparse matrix–matrix multiplication can be referred for our future optimization.

9. Conclusion

This paper presents an optimized design and implementation of tensor-times-dense matrix multiply (Ttm) for sparse and semi-sparse tensors on CPU and GPU platforms. This primitive is a critical bottleneck in tensor decompositions, such as the Tucker decomposition. We design and implement sequential SpTtm and SspTtm to avoid data transformation, and further optimize them on multicore CPU and GPU systems. Five approaches are developed for SpTtm on GPUs, including employing fine thread granularity, arranging coalesced memory access, rank blocking, and using local (fast) memory ("shared memory" on GPUs). Our sequential SpTtm is 3–120× faster than the SpTtm in Tensor Toolbox. Our GPU parallel algorithms achieve good speedups on NVIDIA P100 over our OpenMP-parallelized CPU-SpTtm and also outperform the state-of-the-art GPU implementation by 3.9×. From our analysis, different input sparse tensors, ranks, and operating on different modes all influence SpTtm performance. Adaptive parameters of the SpTtm algorithms will be helpful to achieve fairly good performance for a particular input sparse tensor.

In the future, we will further improve the Tucker decomposition by overlapping some preprocessing stages and memory transfer with the Ttm computation time. We also intend to do further optimization on SpTtm to better handle the load-balance issue. We will extend our algorithms to multi-GPU platforms to support larger sparse tensors. Some tensor work [26,34] is orthogonal to our work, which can be applied to Tucker decomposition.

Acknowledgments

This material is based upon work supported by the U.S. National Science Foundation (NSF) Award Number 1533768, Zhejiang Province Nature Science Foundation of China LR17F030006, National Nature Science Foundation of China (61671196, 61327902), IBM Ph.D. Fellowship Award, and the Laboratory Directed Research and Development program at Sandia National Laboratories, a multi-mission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF, NSFC, IBM, or Sandia National Laboratories.

References

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G.S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, X. Zheng, TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems, 2015. Software available from www.tensorflow.org.

[2] A. Anandkumar, R. Ge, D. Hsu, S.M. Kakade, M. Telgarsky, Tensor decompositions for learning latent variable models, J. Mach. Learn. Res. 15 (1) (2014) 2773–2832.

[3] A. Azad, G. Ballard, A. Buluç, J. Demmel, L. Grigori, O. Schwartz, S. Toledo, S. Williams, Exploiting Multiple Levels of Parallelism in Sparse Matrix-Matrix Multiplication, Tech. Rep. 1510.00844, arXiv, October 2015.

[4] B.W. Bader, T.G. Kolda, Efficient MATLAB computations with sparse and factored tensors, SIAM J. Sci. Comput. 30 (1) (2007) 205–231, http://dx.doi.org/10.1137/060676489.

[5] B.W. Bader, T.G. Kolda, et al., MATLAB Tensor Toolbox (Version 2.6), Available online. February 2015. URL http://www.sandia.gov/~tgkolda/TensorToolbox/.

[6] G. Ballard, A. Druinsky, N. Knight, O. Schwartz, Hypergraph Partitioning for Sparse Matrix-Matrix Multiplication, Tech. Rep. 1603.05627, arXiv 2016.

[7] M. Baskaran, B. Meister, N. Vasilache, R. Lethin, Efficient and scalable computations with sparse tensors, in: High Performance Extreme Computing (HPEC), 2012 IEEE Conference on, 2012, pp. 1–6, http://dx.doi.org/10.1109/HPEC.2012.6408676.

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

[9] A. Buluç, J.R. Gilbert, On the representation and multiplication of hypersparse matrices, in: 2008 IEEE International Symposium on Parallel and Distributed Processing, 2008, pp. 1–11, http://dx.doi.org/10.1109/IPDPS.2008.4536313.

[10] A. Buluç, J.R. Gilbert, Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments, SIAM J. Sci. Comput. 34 (4) (2012) C170–C191, http://dx.doi.org/10.1137/110848244.

[11] A. Carlson, J. Betteridge, B. Kisiel, B. Settles, E. Hruschka, T. Mitchell, Toward an architecture for never-ending language learning, 2010.

[12] J.W. Choi, A. Singh, R.W. Vuduc, Model-driven autotuning of sparse matrix-vector multiply on GPUs, in: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '10, ACM, New York, NY, USA, 2010, pp. 115–126, http://dx.doi.org/10.1145/1693453.1693471, URL http://doi.acm.org/10.1145/1693453.1693471.

[13] A. Cichocki, Tensor decompositions: A new concept in brain data analysis? 2013. arXiv preprint arXiv:1305.0395.

[14] CUDA C Best Practices Guide, 2017. URL http://http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/.

[15] cuSOLVER v9.0, 2018. URL http://docs.nvidia.com/cuda/cusolver/index.html.

[16] S. Dalton, L. Olson, N. Bell, Optimizing sparse matrix–matrix multiplication for the GPU, ACM Trans. Math. Software 41 (4) (2015) 25:1–25:20, http://dx.doi.org/10.1145/2699470.

[17] L. De Lathauwer, B. De Moor, J. Vandewalle, A multilinear singular value decomposition, SIAM J. Matrix Anal. Appl 21 (2000) 1253–1278.

[18] L. De Lathauwer, B. De Moor, J. Vandewalle, On the best Rank-1 and Rank-(R1, R2, ..., RN) approximation of higher-order tensors, SIAM J. Matrix Anal. Appl. 21 (4) (2000) 1324–1342, http://dx.doi.org/10.1137/S0895479898346995.

[19] J.P. Ecker, R. Berrendorf, F. Mannuss, New efficient general sparse matrix formats for parallel SpMV operations, in: F.F. Rivera, T.F. Pena, J.C. Cabaleiro (Eds.), Euro-Par 2017: Parallel Processing, Springer International Publishing, Cham, 2017, pp. 523–537.

[20] O. Görlitz, S. Sizov, S. Staab, PINTS: Peer-to-peer infrastructure for tagging systems, in: Proceedings of the 7th International Conference on Peer-to-peer Systems, IPTPS'08, USENIX Association, Berkeley, CA, USA, 2008, pp. 19–19.

[21] J.C. Ho, J. Ghosh, S.R. Steinhubl, W.F. Stewart, J.C. Denny, B.A. Malin, J. Sun, Limestone: High-throughput candidate phenotype generation via tensor factorization, J. Biomed. Inform. 52 (2014) 199–211.

[22] J.C. Ho, J. Ghosh, J. Sun, Marble: High-throughput phenotyping from electronic health records via sparse nonnegative tensor factorization, in: Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '14, ACM, New York, NY, USA, 2014, pp. 115–124, http://dx.doi.org/10.1145/2623330.2623658.

[23] F. Huang, N. Niranjan U., I. Perros, R. Chen, J. Sun, A. Anandkumar, Scalable latent tree model and its application to health analytics, 2014. ArXiv e-prints, arXiv:1406.4566.

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

[25] U. Kang, E. Papalexakis, A. Harpale, 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, KDD '12, ACM, New York, NY, USA, 2012, pp. 316–324, http://dx.doi.org/10.1145/2339530.2339583.

[26] O. Kaya, B. Uçar, High-performance Parallel Algorithms for the Tucker Decomposition of Higher Order Sparse Tensors, Tech. rep., Inria - Research Centre Grenoble – Rhone-Alpes, 2015.

[27] P. Koanantakool, A. Azad, A. Buluç, D. Morozov, S.Y. Oh, L. Oliker, K. Yelick, Communication-avoiding parallel sparse-dense matrix-matrix multiplication, in: 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2016, pp. 842–853, http://dx.doi.org/10.1109/IPDPS.2016.117.

[28] T.G. Kolda, B.W. Bader, Tensor decompositions and applications, SIAM Rev. 51 (3) (2009) 455–500.

[29] K. Kourtis, V. Karakasis, G. Goumas, N. Koziris, CSX: An extended compression format for Spmv on shared memory systems, in: Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming, PPoPP '11, ACM, New York, NY, USA, 2011, pp. 247–256, http://dx.doi.org/10.1145/1941553.1941587, URL http://doi.acm.org/10.1145/1941553.1941587.

[30] D. Lahat, T. Adalý, C. Jutten, Challenges in multimodal data fusion, in: Signal Processing Conference (EUSIPCO), 2014 Proceedings of the 22nd European, 2014, pp. 101–105.

[31] D. Langr, I. Šimeček, P. Tvrdík, Storing sparse matrices to files in the adaptive-blocking hierarchical storage format, in: 2013 Federated Conference on Computer Science and Information Systems, 2013, pp. 479–486.

[32] D. Langr, I. Šimeček, P. Tvrdík, T. Dytrych, J. Draayer, Adaptive-blocking hierarchical storage format for sparse matrices, in: 2012 Federated Conference on Computer Science and Information Systems, FedCSIS, 2012, pp. 545–551.

[33] C.-F.V. Latchoumane, F.B. Vialatte, J. Solé-Casals, M. Maurice, S.R. Wimalaratna, N. Hudson, J. Jeong, A. Cichocki, Multiway array decomposition analysis of EEGs in Alzheimer's disease, J. Neurosci. Methods 207 (1) (2012) 41–50.

[34] J. Li, J. Choi, I. Perros, J. Sun, R. Vuduc, Model-Driven sparse cp decomposition for higher-order tensors, in: 2017 IEEE International Parallel and Distributed Processing Symposium, IPDPS, 2017, pp. 1048–1057. http://dx.doi.org/10.1109/IPDPS.2017.80.

[35] J. Li, Y. Ma, C. Yan, 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, IA^3 '16, IEEE Press, Piscataway, NJ, USA, 2016, pp. 26–33, http://dx.doi.org/10.1109/IA3.2016.10.

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

[37] Y. Liang, W.T. Tang, R. Zhao, M. Lu, H.P. Huynh, R.S.M. Goh, Scale-free sparse matrix-vector multiplication on many-core architectures, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 36 (12) (2017) 2106–2119, http://dx.doi.org/10.1109/TCAD.2017.2681072.

[38] W. Liu, Parallel and scalable sparse basic linear algebra subprograms, (Ph.D. thesis), University of Copenhagen, 2015.

[39] J. Liu, X. He, W. Liu, G. Tan, Register-based implementation of the sparse general matrix-matrix multiplication on GPUs, in: Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '18, ACM, New York, NY, USA, 2018, pp. 407–408, http://dx.doi.org/10.1145/3178487.3178529, URL http://doi.acm.org/10.1145/3178487.3178529.

[40] W. Liu, B. Vinter, An efficient GPU general sparse matrix-matrix multiplication for irregular data, in: Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium, IPDPS '14, IEEE Computer Society, Washington, DC, USA, 2014, pp. 370–381, http://dx.doi.org/10.1109/IPDPS.2014.47.

[41] W. Liu, B. Vinter, A framework for general sparse matrix-matrix multiplication on GPUs and heterogeneous processors, J. Parallel Distrib. Comput. 85 (C) (2015) 47–61, http://dx.doi.org/10.1016/j.jpdc.2015.06.010.

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

[43] B. Liu, C. Wen, A.D. Sarwate, M.M. Dehnavi, A unified optimization approach for sparse tensor operations on GPUs, in: 2017 IEEE International Conference on Cluster Computing (CLUSTER), 2017, pp. 47–57. http://dx.doi.org/10.1109/CLUSTER.2017.75.

[44] M. McCourt, B. Smith, H. Zhang, Sparse matrix-matrix products executed through coloring, SIAM J. Matrix Anal. Appl. 36 (1) (2015) 90–109, http://dx.doi.org/10.1137/13093426X, arXiv:https://doi.org/10.1137/13093426X.

[45] E.D. Napoli, D. Fabregat-Traver, G.Q. -Ortí, P. Bientinesi, Towards an efficient use of the BLAS library for multilinear tensor contractions, Appl. Math. Comput. 235 (2014) 454–468, http://dx.doi.org/10.1016/j.amc.2014.02.051.

[46] A. Novikov, D. Podoprikhin, A. Osokin, D. Vetrov, Tensorizing neural networks, 2015. CoRR, abs/1509.06569.

[47] OpenBLAS : An optimized BLAS library, 2017. URL https://github.com/xianyi/OpenBLAS.

[48] E.E. Papalexakis, C. Faloutsos, N.D. Sidiropoulos, ParCube: Sparse parallelizable tensor decompositions, in: Proceedings of the 2012 European Conference on Machine Learning and Knowledge Discovery in Databases - Volume Part I, ECML PKDD'12, Springer-Verlag, Berlin, Heidelberg, 2012, pp. 521–536, http://dx.doi.org/10.1007/978-3-642-33460-3_39.

[49] I. Perros, R. Chen, R. Vuduc, J. Sun, Sparse hierarchical Tucker factorization and its application to healthcare, in: IEEE International Conference on Data Mining (ICDM), 2015.

[50] S. Rajbhandari, A. Nikam, P.-W. Lai, K. Stock, S. Krishnamoorthy, P. Sadayappan, A communication-optimal framework for contracting distributed tensors, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '14, IEEE Press, Piscataway, NJ, USA, 2014, pp. 375–386, http://dx.doi.org/10.1109/SC.2014.36.

[51] Y. Shi, U.N. Niranjan, A. Anandkumar, C. Cecka, Tensor contractions with extended BLAS kernels on CPU and GPU, 2016. CoRR, abs/1606.05696.

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

[53] S. Smith, 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.

[54] E. Solomonik, J. Demmel, T. Hoefler, Communication Lower Bounds for Tensor Contraction Algorithms, Tech. rep., 36, ETH Zürich, 2015.

[55] E. Solomonik, T. Hoefler, Sparse tensor algebra as a parallel programming model, 2015. arXiv preprint arXiv:1512.00066.

[56] G. Tan, J. Liu, J. Li, Design and implementation of adaptive SpMV library for multicore and manycore architecture, ACM Trans. Math. Software (2018).

[57] R. Vuduc, Automatic performance tuning of sparse matrix kernels, (Ph.D. thesis), University of California, Berkeley, 2003.

[58] R. Vuduc, J. W.D.emmel, K. A.Y.elick, OSKI: A library of automatically tuned sparse matrix kernels, J. Phys. Conf. Ser. 16 (1) (2005) 521, URL http://stacks.iop.org/1742-6596/16/i=1/a=071.

[59] Y. Wang, R. Chen, J. Ghosh, J.C. Denny, A. Kho, Y. Chen, B.A. Malin, J. Sun, Rubik: Knowledge guided tensor factorization and completion for health data analytics, in: Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '15, ACM, New York, NY, USA, 2015, pp. 1265–1274, http://dx.doi.org/10.1145/2783258.2783395.

[60] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, J. Demmel, Optimization of sparse matrix–vector multiplication on emerging multicore platforms, in: Revolutionary Technologies for Acceleration of Emerging Petascale Applications, Parallel Comput. 35 (3) (2009) 178–194, http://dx.doi.org/10.1016/j.parco.2008.12.006, URL http://www.sciencedirect.com/science/article/pii/S0167819108001403.

[61] B. Xie, J. Zhan, X. Liu, W. Gao, Z. Jia, X. He, L. Zhang, CVR: Efficient vectorization of SpMV on x86 processors, in: Proceedings of the 2018 International Symposium on Code Generation and Optimization, CGO 2018, ACM, New York, NY, USA, 2018, pp. 149–162, http://dx.doi.org/10.1145/3168818, URL http://doi.acm.org/10.1145/3168818.

[62] S. Yan, C. Li, Y. Zhang, H. Zhou, yaSpMV: Yet another SpMV framework on GPUs, in: Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, ACM, New York, NY, USA, 2014, pp. 107–118, http://dx.doi.org/10.1145/2555243.2555255, URL http://doi.acm.org/10.1145/2555243.2555255.

[63] R. Yuster, U. Zwick, Fast sparse matrix multiplication, ACM Trans. Algorithms 1 (1) (2005) 2–13, http://dx.doi.org/10.1145/1077464.1077466, URL http://doi.acm.org/10.1145/1077464.1077466.

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

Yuchen Ma is an under-graduate student at Zhuoyue Honors College, Hangzhou Dianzi University, China.

Jiajia Li is a Ph.D. student in the School of Computational Science and Engineering, Georgia Tech. Her research interest includes high performance computing, parallel algorithms, computational data mining.

Xiaolong Wu is a Ph.D. student in the School of Computer Science, Virginia Tech. His research interest includes cloud computing, operating system, and computer architecture.

Chenggang Clarence Yan is a Professor of Department of Automation, Hangzhou Dianzi University, China. He received his Ph.D. from the Institute of Computing Technology, Chinese Academy of Sciences, China in 2013. He was a post-doctoral research fellow with the Department of Automation, Tsinghua University, China. His research interests include parallel computing, video coding, computational photography, computer vision, and multimedia communication.

Jimeng Sun is an Associate Professor of School of Computational Science and Engineering at College of Computing at Georgia Institute of Technology. His research interests are on data mining and machine learning for health applications especially analysis of electronic health records (EHR): computational phenotyping from electronic health records, deep learning for healthcare, tensor analysis.

Rich Vuduc is an associate professor in the School of CSE, Georgia Tech. His research lab, the HPC Garage (hpcgarage.org), is interested in all-things-high-performance-computing, with an emphasis on parallel algorithms, performance analysis, and performance tuning. He is a member of the DARPA Computer Science Study Panel, received the NSF CAREER Award, and co-recipient of the Gordon Bell Prize (2010). His lab's work has received a number of best paper nominations and awards, including most recently the 2012 Best Paper Award from the SIAM Conference on Data Mining.