Optimizing Sparse Tensor Times Matrix on Multi-core and Many-core Architectures

Jiajia Li
Computational Science & Engineering
Georgia Institute of Technology
Atlanta, Georgia 30332
Email: jiajiali@gatech.edu

Yuchen Ma, Chenggang Yan
Institute of Information and Control
Hangzhou Dianzi University
Hangzhou, China

Richard Vuduc
Computational Science & Engineering
Georgia Institute of Technology
Atlanta, Georgia 30332

***Abstract*—This paper presents the optimized design and implementation of sparse tensor-times-dense matrix multiply (SpTTM) for CPU and GPU platforms. This primitive is a critical bottleneck in data analysis and mining applications based on tensor methods, such as the Tucker decomposition. We first design and implement sequential SpTTM to avoid explicit data transformations between a tensor and a matrix, which is the conventional approach. We further optimize SpTTM on multicore CPU and GPU systems by parallelizing, avoiding locks, and exploiting data locality. Our sequential SpTTM is up to 3.5× faster than the SpTTM from Tensor Toolbox and 1.5× over that from Cyclops Tensor Framework. Our parallel algorithms show 4.1× speedup on multicore Intel Core i7 and 18.8× speedup on NVIDIA K40c GPU over our sequential SpTTM respectively.**

I. Introduction

This paper considers the problem of optimizing the sparse tensor-times-dense matrix (SpTTM) operation, which appears widely in tensor-based data analysis applications. Such applications arise in numerous domains, including neuroscience [1, 2], healthcare analytics [3–5], natural language processing [6], signal processing [7], machine learning [8, 9], and social network analytics [10]. Tensors, which are multi-way arrays, provide a natural way to represent multidimensional data; a subsequent analysis of the tensor usually takes the form of factoring or decomposing the tensor into interpretable components [11–14]. (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 [13], depend critically on having a fast SpTTM, thereby motivating this study.[1]

Regarding our paper's scope, we consider parallelism and locality for single-node multicore CPU and 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, there are several efficient methods for the case when the tensor is dense [15–18]. The sparse case is especially important to data analysis applications, since real-world data is often voluminous but sparse.

In principle, an SpTTM is similar to sparse matrix-times-dense matrix (SpMM). Indeed, conventional SpTTM implementations, such as those in the Tensor Toolbox [19] and Cyclops Tensor Framework (CTF) [20], first transform a sparse tensor into an equivalent sparse matrix and then assume an optimized SpMM. This approach is reasonable and produces good results. However, this conversion step incurs nontrivial costs in time and space. Moreover, the generated matrix can be very large in one of its dimensions, explicit indexing of which—for a many-way tensor—can quickly exceed the range of a 64-bit unsigned integer. Thus, we are motivated primarily to avoid any such conversion, carrying out the SpTTM "natively" on the given input tensor.

With that as background, our proposed techniques make the following contributions:

- We design and implement sequential SpTTM to avoid data transformation between a tensor and a matrix.
- We optimize SpTTM on single-node multicore CPU and GPU systems by parallelizing, avoiding locks, and employing local (fast) memory (caches on CPUs and "shared memory" on GPUs).
- Our optimized SpTTM achieves up to 4.1× on multicore Intel Core i7 and 18.8× on NVIDIA GPU K40c over our sequential SpTTM baseline, which is maximally 3.5× faster than the SpTTM from Tensor Toolbox and 1.5× over that from Cyclops Tensor Framework.

II. Background

We first introduce the essential tensor notation for basic tensor operations. Several examples and definitions are drawn from the overview by Kolda and Bader [13].

A. Tensor Representation

A tensor is defined as a multi-way array, and the *order* of a tensor is the number of dimensions, also called *modes*. Vectors, or first-order tensors, are denoted by boldface lowercase letters, e.g., $\mathbf{v}$, and matrices, or second-order tensors, are denoted by boldface capital letters, e.g., $\mathbf{A}$. High-order tensors are denoted by bold capital calligraphic letters, e.g.,

[1] Beyond SpTTM, other basic tensor operations that can appear as bottlenecks in other decompositions include tensor matricization (converting a tensor into an equivalent matrix), elementwise tensor addition/subtraction/multiplication/division, Kronecker products, Khatri-Rao products, and Matricized Tensor Times Khatri-Rao Product (MTTKRP).

IA^3 2016; Salt Lake City, Utah, USA; November 2016
978-1-5090-3867-1/16/$31.00 ©2016 IEEE

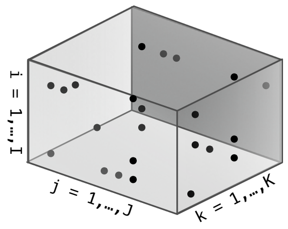


Fig. 1. A third-order sparse tensor $\mathcal{X}$.

$\mathcal{X}$. Figure 1 shows a third-order tensor $\mathcal{X} \in \mathbb{R}^{I\times J\times K}$. The scalar element at position (i, j, k) of tensor $\mathcal{X}$ is denoted by x_{ijk}.

High-order tensors have a close relation with matrices and vectors. A *slice* (figure 2(a)) is a 2-dimensional cross-section of a tensor, achieved by fixing all indices but two, e.g $\mathbf{S}_{::k} = \mathcal{X}(:, :, k)$ in MATLAB notation. A *fiber* (figure 2(b)) is a vector extracted from a tensor along a specified mode, selected by fixing all indices but one, e.g $\mathbf{f}_{:jk} = \mathcal{X}(:, j, k)$. A tensor consists of a set of slices or fibers. Besides, a tensor can be transformed into an equivalent matrix by recalculating indices, this is called *matricization*, or *unfolding*. For example, mode-1 matricization of a $3 \times 4 \times 5$ tensor $\mathcal{X}$ would result in a 3×20 matrix $\mathbf{X}_{(1)}$. We recommend readers to see more details in [13]. In the following tensor operations, different tensor representations will be used for the best explanation.

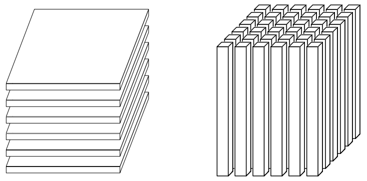


Fig. 2. Slices and fibers of a third-order tensor $\mathcal{X} \in \mathbb{R}^{I\times J\times K}$. (a) Slices $\mathbf{S}_{::k} = \mathcal{X}(:, :, k)$; (b) Fibers $\mathbf{f}_{:jk} = \mathcal{X}(:, j, k)$.

B. Tensor-Times-Matrix

Tensor-Times-Matrix (TTM) on mode n, also known as the n-mode product, is the multiplication of a tensor $\mathcal{X} \in \mathbb{R}^{I_1\times I_2\times\cdots\times I_n\times\cdots\times I_N}$ 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 a $I_1 \times I_2 \times \cdots \times I_{n-1} \times R \times I_{n+1} \times \cdots \times I_N$ 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 nonzero entries, we focus on sparse tensor-times-dense matrix (SpTTM) in this paper. SpTTM 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

[2]Different from [13], We use the transposed form of the matrix $\mathbf{U}$ for efficient TTM in row-majored storage pattern of C language.

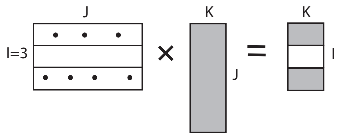


Fig. 3. Sparse matrix-dense matrix multiply.

of each matrix is called *rank*. We focus on the popular low-rank tensor decompositions, which means the factor matrices all have small number of columns.

III. SpTTM Property and sCOO format

We first explain an important property of SpTTM to its implementation, then we introduce sCOO format for a semi-sparse tensor, which is a special sparse tensor.

A. Notation

We introduce some notation to be used in the following paper.

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

B. SpTTM Property

SpTTM outputs a semi-sparse tensor whose product mode is dense and index modes are unchanged.

Mode-n fibers of $\mathcal{X}$ and $\mathcal{Y}$ tensors are defined as

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

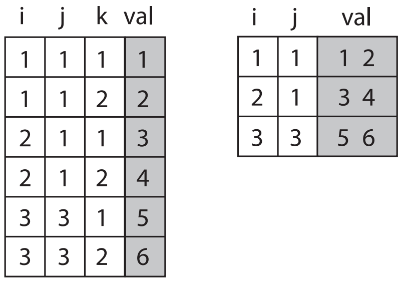
where $i_1, \ldots, i_{n-1}, i_{n+1}, \ldots, i_N$ are fixed. $\mathbf{f}_n^X$ is a sparse fiber because of the sparsity of $\mathcal{X}$. Equation 1 is equal to

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

when r is fixed. Element $\mathbf{f}_n^Y(r)$ is the dot-product of fiber $\mathbf{f}_n^X$ and $\mathbf{u}(:, r)$, a column of U. Since $\mathbf{u}(:, r)$ is a dense vector, each $f_n^Y(r)$ is non-zero only if there exist at least one nonzero 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 $\mathcal{Y}$, so the indices $i_1, \ldots, i_{n-1}, i_{n+1}, \ldots, i_N$ are unchanged for the resulting tensor $\mathcal{Y}$. We call a sparse tensor with one or more dense modes as a *semi-sparse* tensor. Figure 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 index i is indexing dense fibers now.

C. sCOO format

Tensors generated from real-world applications are usually hyper-sparse with only a small number of nonzero entries. Thus, a sparse tensor is represented by a compressed format by storing only the nonzero entries. The most straight-forward and popular format of a sparse tensor is coordinate (COO) format. COO format uses one array to store the nonzero entries, and N arrays to store all the indices for N modes. Figure 4(a) shows a $3 \times 3 \times 2$ semi-sparse tensor with mode-3 (indexing by k) as the dense mode in the COO format.



i	j	k	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

i	j	val
1	1	1 2
2	1	3 4
3	3	5 6

(b) sCOO

Fig. 4. COO and sCOO formats of a semi-sparse $3 \times 3 \times 2$ tensor.

Since the indices of a dense mode can be computed from the location of nonzero entries, it is unnecessary to store dense modes' indices by extra arrays in a semi-sparse tensor. Besides, due to the highly sparse property of the input tensor of SpTTM, the output is still a sparse tensor but with a dense mode. We use a simple semi-COO (sCOO) format which only stores the indices of sparse modes and all the nonzeros. The idea of distinguishing dense and sparse modes was first proposed by Baskaran et al in [21]. Figure 4(b) shows the sCOO format for the example semi-sparse tensor. The sCOO format saves 50% space compared to the COO format to store a third-order semi-sparse tensor with one dense mode, when integer and floating-point values occupy the same number of bits. The space benefit comes from two aspects: First, sCOO eliminates the index array of the dense mode; Second, each index array of sCOO is shorter because of index compression. If a Nth-order semi-sparse tensor has k dense modes, then sCOO will save at least $\frac{k}{N+1}$ storage space.

IV. Sequential SpTTM

Based on COO and sCOO formats, we implement sequential SpTTM by directly operating on nonzero entries without explicit transformation.

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 Equation 1, the ideal time complexity of SpTTM is

$$T_{ideal} = 2 \times Xnnz \times R, \tag{3}$$

where $Xnnz$ is the number of nonzeros of $\mathcal{X}$. The intuitive algorithm for SpTTM without explicit transformation is to loop all nonzeros of $\mathcal{X}$ by timing each one with its corresponding row of $\mathbf{U}$. Then all rows with the same indices $(i_1, \ldots, i_{n-1}, i_{n+1}, \ldots, i_N)$ are sum-reduced to get a fiber of $\mathcal{Y}$ ($\mathbf{f}_n^Y$). This algorithm has two problems: First, in the sum-reduction stage there is an implicit index comparing operation even if $\mathcal{X}$ is pre-sorted. The complexity of comparing indices $(i_1, \ldots, i_{n-1}, i_{n+1}, \ldots, i_N)$ is high especially for high-order sparse tensors. One more comparison for an extra mode increases the SpTTM complexity by $Xnnz$, which is non-trivial compared to Equation 3 especially for low-rank tensor decomposition with a small R. Second, the sum-reduction stage is hard to parallelize and may lead to memory write conflicts.

To solve these problems, we design our SpTTM algorithm (Algorithm 1) to avoid expensive index comparison and with easy-to-implement parallelization. Each mode-n fiber of $\mathcal{Y}$ ($\mathbf{f}_n^Y$) is a sized-R dense vector, we record $nfibs$ as the number of $\mathbf{f}_n^Y$. Thus the number of nonzeros of $\mathcal{Y}$: $Ynnz = nfibs * R$. We use an extra array $fptr$ to identify the beginning locations of every mode-n fiber of $\mathcal{X}$ ($\mathbf{f}_n^X$), then transverse all $nfibs$ fibers of $\mathcal{Y}$. Instead of transversing $\mathcal{X}$, we avoid comparing the indices.

Our SpTTM has two steps, pre-allocating semi-sparse tensor $\mathcal{Y}$ and computing. From SpTTM's property in Section III, the semi-sparse tensor keeps the index modes unchanged, so the $nfibs$ of $\mathcal{Y}$ is the same with the number of $\mathbf{f}_n^X$. By pre-sorting on $\mathcal{X}$, we can allocate the accurate space for $\mathcal{Y}$. During the computation step, each $\mathbf{f}_n^Y = yval(i,:)$ locates the corresponding $\mathbf{f}_n^X = xval(fptr(i), \ldots, fptr(i+1)-1)$. Then $\mathbf{f}_n^Y$ is the sum of rows $\mathbf{u}(mind,:)$ scaled by each nonzero of fiber $\mathbf{f}_n^X$. Algorithm 1 achieves the ideal time complexity (Equation 3) by eliminating the index comparison. For a third-order sparse tensor, our SpTTM only uses $O(4 \times Xnnz)$ space which is much smaller than the traditional $O(7 \times Xnnz)$ counting the transformed matrix in COO format. Thus, our SpTTM saves at least 42% space for a third-order tensor.

Algorithm 1 Sequential SpTTM algorithm.

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

V. Optimized SpTTM

Based on our sequential SpTTM, we optimize SpTTM on multicore CPU and GPU platforms by parallelizing, avoiding locks, and using local (fast) memory (caches on CPUs and "shared memory" on GPUs).

A. Multi-threading SpTTM

We first parallelize SpTTM on the multicore CPU architecture using OpenMP. Since our sequential SpTTM transverses all independent fibers of $\mathcal{Y}$, we can easily parallelize this loop by assigning different CPU threads. In Algorithm 2, each thread computes a length R fiber $\mathbf{f}_n^Y$ independently and shares matrix $\mathbf{U}$. Because our SpTTM algorithm limits the sum-reduction dependency inside a thread, OMP SpTTM naturally avoids locks and well utilizes CPU caches for reading $\mathcal{X}$ and its indices and writing $\mathcal{Y}$.

Algorithm 2 Multi-threading SpTTM algorithm.

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

B. GPU SpTTM

We also parallelize SpTTM on the GPU architecture using CUDA programing interface and optimize it using GPU shared memory.

1) Parallelization strategy: We assign two-dimensional thread blocks to compute each nonzero entry of $\mathcal{Y}$ with the size $ntx \times nty$, where nty is set to R, the number of columns of $\mathbf{U}$ and ntx could be $16, 32, 64, 128$ according to different nty values. Since our SpTTM algorithm is a main kernel for low-rank decomposition, the rank R is usually a number smaller than 100 [13]. Thus, only one-dimensional grids are needed, each of which consists of up to $nb = nfibs/ntx$ blocks. A real sparse tensor may have more $nfibs$ than a grid's capability, we further divide SpTTM into ng device kernels, with each kernel launches the maximum number of blocks of a grid. To summarize,

$$ng = \frac{nfibs}{maxGridSize \times ntx} \tag{4}$$

$$nb = \begin{cases} \frac{nfibs}{ntx}, ng = 1 \\ \frac{maxGridSize}{ntx}, ng > 1 \end{cases} \tag{5}$$

Because of GPU hardware limitation, the maximum number of threads per block $maxBlockSize = 1024$. So,

$$ntx < \frac{maxBlockSize}{nty} \tag{6}$$

According to different $nty = R$, we choose the maximal possible value under this limitation. Overall, our GPU SpTTM has ng kernels and each kernel has nb blocks of $ntx \times nty$ threads. Because our SpTTM algorithm doesn't need communication between threads, each thread can execute independently.

2) Using GPU shared memory: We first analyze the GPU SpTTM behavior to find data locality. The only data reuses are matrix $\mathbf{U}$ and tensor $\mathcal{Y}$. $u(mind, tidy)$ can be reused by ntx threads with the same $tidy$ when $mind$ happens to be the same. Large sparse tensors usually have low reuse degree because of the hyper-sparse nonzero distribution. For example the density of 'nell2' tensor from Never Ending Language Learning (NELL) project [22] is 2.40e-05. Thus, the reuse of $\mathbf{U}$ can be reasonably ignored when no pre-processing of $\mathcal{X}$ is used for better locality. Therefore, we only reuse $\mathcal{Y}$, each $yval(i, tidy)$ is reused by $fptr(i-1) - fptr(i)$ times.

In Algorithm 3, we show GPU SpTTM algorithm by using GPU shared memory for tensor $\mathcal{Y}$. The needed shared memory size is $ntr \times R \times sizeof(double) < 8K$ according to Equation 6, which is smaller than shared memory size (16K or 48K). Although data locality of $\mathcal{Y}$ could also be explored by GPU caches, long fibers and their indices of large tensor $\mathcal{X}$ may potentially generate more cache misses for writing $\mathcal{Y}$, especially for relatively small GPU cache (Table I). Thus, we secure tensor $\mathcal{Y}$ using GPU shared memory.

Algorithm 3 GPU SpTTM algorithm with GPU shared memory utilization.

```
Input: A sparse tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$, a dense matrix
 1: $\mathbf{U} \in \mathbb{R}^{I_n \times R}$, an integer $n$, and GPU thread hierarchy $dimGrid$
    and $dimBlock$;
Output: A semi-sparse tensor $\mathcal{Y} \in \mathbb{R}^{I_1 \times \cdots \times I_{n-1} \times R \times I_{n+1} \cdots \times I_N}$;
 2: $nfibs$: the number of mode-n fibers of $\mathcal{Y}$
 3: $fptr$: the beginnings of each $\mathcal{X}$ mode-n fiber, size $nfibs$.
 4: Yshr: shared memory space for $\mathcal{Y}$.
 5: $tidx = threadIdx.x$;
 6: $tidy = threadIdx.y$;
                                   ▷ $i$: global index of a $\mathcal{Y}$ mode-n fiber.
 7: $i = blockIdx.x * blockDim.x + tidx$;
 8: $yshr(tidx, tidy) = 0$;
                         ▷ $j$: global index of the nonzeros of $\mathcal{X}$ mode-n fiber.
 9: for $j = fptr(i), \ldots, fptr(i+1) - 1$ do
10:     $mind = xinds(n, j)$
11:     $yshr(tidx, tidy) += xval(j) * u(mind, tidy)$
12: end for
13: $yval(i, tidy) = yshr(tidx, tidy)$;
14: Return $\mathcal{Y}$;
```

VI. Experiments

In this section, we test our algorithms on two platforms and also compare the performance with state-of-the-art Tensor Toolbox library [13]. Then we analyze the performance behavior by varying product modes and ranks.

A. Platforms and Dataset

We use Intel Core i7-4770K and NVIDIA Tesla K40c testing platforms (Table I). NVIDIA Tesla K40c has much higher peak performance and bandwidth than multicore Intel Core i7. All experiments are performed in double-precision

TABLE I
EXPERIMENTAL PLATFORMS CONFIGURATION

Parameters	Intel Core i7-4770K	NVIDIA Tesla K40c
Microarchitecture	Haswell	Kepler
Frequency	3.5 GHz	0.75 GHz
#Physical cores	4	2880
Peak DP Performance	56 Gflops	1430 Gflops
Last-level cache	8 MB	1.5 MB
Memory size	32 GiB	12 GB
Memory bandwidth	25.6 GB/s	288 GB/s
Compiler	gcc 5.4.0	nvcc 7.5

and the performance numbers are the average of five running tests.

We use sparse tensors from real applications including functional Magnetic Resonance Imaging (fMRI) measurements of brain activity [23] ("brainq" with *noun-voxel-human*), Never Ending Language Learning (NELL) project [22] ("nell1" and "nell2" with *noun-verb-noun*), and data crawled from tagging systems [24] ("deli" with *user-item-tag*). The details of the data are shown in Table II.

TABLE II
DESCRIPTION OF SPARSE TENSORS.

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	1.3e-05
nell1	3	$2.9M \times 2.1M \times 25.5M$	144M	3.1e-13
deli	3	$0.5M \times 17.3M \times 2.5M$	140M	6.1e-12

B. Performance Speedup

We test our algorithms on CPU and GPU platforms and show their speedups over our sequential SpTTM on real tensors in Figure 5. The number of columns of the dense matrix is set to 16, to reflect tensor decomposition's low-rank property. All speedup numbers are SpTTM on mode-3 and averaged over five iterations. Because of the large tensor sizes, GPU memory cannot hold all the data of some SpTTM algorithms. The OMP SpTTM speedups over our sequential SpTTM are obtained by employing the maximum eight threads using hyper-threading technique. The GPU SpTTM speedups over our sequential SpTTM are tested by setting $ntx = 32$. Compared to our sequential SpTTM, OMP SpTTM achieves up to $4.1\times$ speedup and GPU SpTTM achieves up to $18.8\times$ speedup on the four real sparse tensors. GPU SpTTM's speedup over OMP SpTTM is up to $6.0\times$, showing the advantage of using GPU to accelerate SpTTM.

We also compare our results with MATLAB Tensor Toolbox's implementation from Sandia National Laboratories [19] and Cyclops Tensor Framework [20] in C++ language. We compare with Tensor Toolbox version 2.6 and CTF-1.4.1, while Tensor Toolbox is build in MATLAB R2014b, CTF is compiled with Intel icc-17.0.0 linked with Intel Math Kernel Library [25]. We record their sequential execution time and get

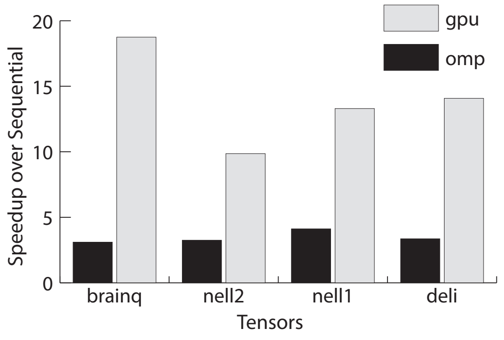


Fig. 5. Parallel SpTTM's speedup over sequential SpTTM.

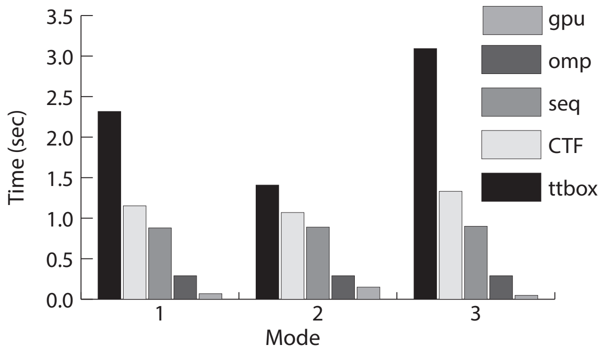


Fig. 6. Execution time comparison of Tensor Toolbox, Cyclops Tensor Framework, and our SpTTMs on the "brainq" tensor.

the average over five iterations. SpTTMs in Tensor Toolbox and CTF first transform a sparse tensor into a sparse matrix, and then do a sparse matrix and a dense matrix multiplication, finally transform the sparse matrix back to a sparse or dense tensor. Our SpTTM algorithm is directly implemented on the input sparse tensor without transformation, which saves time and space. Figure 6 shows the execution time of our SpTTM algorithms comparing with Tensor Toolbox (TTBox) and Cyclops Tensor Framework (CTF) on the "brainq" tensor, which is the only tensor they can run without exceeding CPU memory. [3] Our SpTTM algorithms ("seq", "omp", and "gpu") are much faster than TTBox and CTF. One reason is that we avoid transformation time, and the other reason is our optimized sequential SpTTM improves the performance. This figure also proves comparing OMP and GPU SpTTM with our own written sequential SpTTM is fair enough.

C. Analysis

We first analyze the tensor transformation time percentage of Tensor Toolbox, then analyze SpTTM performance behavior by varying product modes and ranks. We also measure the benefit of using GPU shared memory.

1) Tensor transformation: We use "ttbox" and "ttbox w/o transform" to show the performance of Tensor Toolbox and the pure multiply operation in it (without data transformation)

[3]CTF stores its SpTTM's output in a dense tensor, while TTBox uses a sparse or dense format depending on the output tensor's sparsity.

in Figure 7. The data transformation, including the conversion time of matricization and tensorization before and after the multiply, takes 14-31% execution time. Tensor Toolbox with and without data transformation are both slower than our sequential SpTTM in all cases, which shows our transformation avoiding is beneficial and our SpTTM algorithm (Algorithm 1) outperforms Tensor Toolbox's SpTTM.

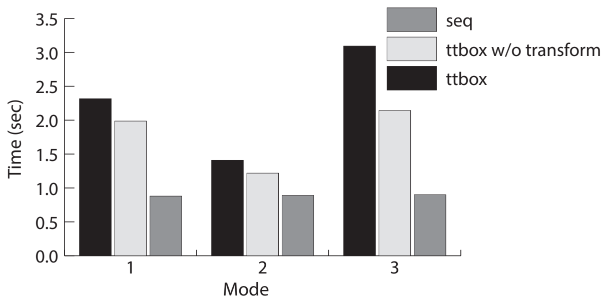


Fig. 7. The execution time of Tensor Toolbox w/ and w/o tensor transformation on the "brainq" tensor.

2) Mode behavior: From Figure 6, the running time of GPU SpTTM on the three modes varies more than OMP and sequential SpTTM. GPU SpTTM gets the longest running time on mode-2. Considering the dimension size of "brainq" is $60 \times 70K \times 9$ from Table II, SpTTM on mode-2 can only have up to 540 thread blocks, which is quite small for the GPU architecture. Thus, GPU SpTTM runs even slower than OMP SpTTM on mode-2. This figure shows SpTTM on different modes may have different performance behavior according to the dimension sizes.

3) Rank behavior: We also test SpTTM on different ranks $8, 16, 32, 64$. From our experiments, keeping 512 threads per block achieves the highest performance, so we set block sizes as $(64, 8), (32, 16), (16, 32), (8, 64)$ to test GPU SpTTM. Note that in this experiment, only the block shape varies, the number of blocks and threads are kept the same. When the rank size grows, the output tensor $\mathcal{Y}$ becomes larger, thus we can do a full test only on the two relative small tensors "brainq" and "nell2". We use relative performance by mapping the GPU SpTTM performance with rank size 8 to 1. Figure 8 shows the performance grows when increasing the rank size, because a longer dense vector is processed by each thread with regular memory accesses. The actual performance numbers of "brainq" and "nell2"are $2-7$ Gflops and $3-12$ Gflops respectively, which are both much lower than the GPU's peak performance in Table I because of the irregularity of sparse tensors.

4) GPU shared memory: Figure 9 compares the performance of the GPU SpTTM with or without using GPU shared memory using the speedups over sequential SpTTM. This experiment uses the same parameter settings as in Section VI-B. Our GPU SpTTM with GPU shared memory optimization doesn't benefit much from this method on the testing tensors, the improvement is 5%-27%. Further optimization on GPU SpTTM is our next research step.

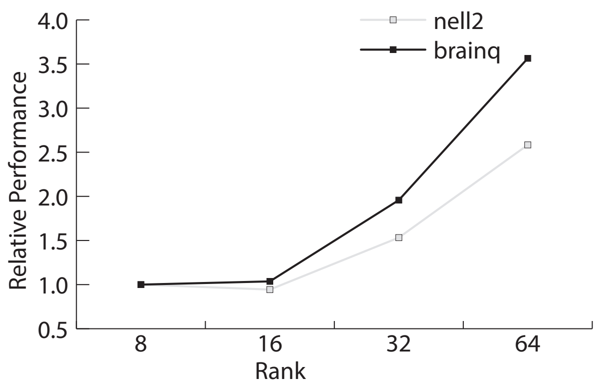


Fig. 8. Relative performance of GPU SpTTM with different rank sizes on "brainq" and "nell2" tensors.

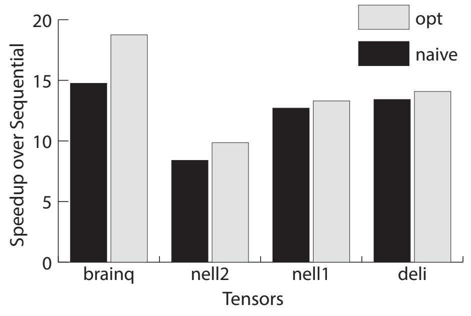


Fig. 9. GPU SpTTM Speedup over sequential SpTTM with ("opt") or without ("naive") GPU shared memory.

VII. Related Work

Several other libraries also realized basic sparse tensor operations, such as Tensor Toolbox [19] and Cyclops Tensor Framework (CTF) [20, 26]. Tensor Toolbox is implemented in MATLAB environment, and CTF is implemented in C++ language. From algorithm aspect, Tensor Toolbox implement 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-trial time. CTF is a parallel framework of tensor contraction for distributed CPU systems, and it only supports storing the output of sparse tensor-times-dense matrix as a dense tensor, which is not space-efficient for hyper-sparse tensors. Our SpTTM algorithms directly operate on the coordinate format of a sparse tensor, and the resulting tensor is stored in a hybrid format (sCOO) to save space. Besides, our work is the first bringing sparse tensor operations to GPU, to utilize GPUs powerful computing capability. [4]

Some work proposed efficient storage formats for a sparse tensor, such as the Compressed Sparse Fiber (CSF) format [28] and the "mode-generic sparse storage format" [21]. Our sCOO format is a simple version of the "mode-generic sparse storage format" proposed by Baskaran et al, with dense modes fixed

[4]CTF supports some GPU functions now but not SpTTM, although given the CTF infrastructure and a CUDA sparse matrix library (e.g. cuSPARSE [27]), it would be easy to support SpTTM in the future.

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 operation 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.

Some work on sparse matrix and dense matrix multiplication [29–36] 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 and dense matrix multiplication can be referred for our future optimization.

VIII. Conclusion

This paper presents an optimized design and implementation of sparse tensor-times-dense matrix multiply (SpTTM) for CPU and GPU platforms. This primitive is a critical bottleneck in tensor decompositions, such as Tucker decomposition. We design and implement sequential SpTTM to avoid data transformation, and further optimize SpTTM on multicore CPU and GPU systems by parallelizing, avoiding locks, and exploring data locality. Our sequential SpTTM is maximally $3.5\times$ and $1.5\times$ faster than the SpTTMs in Tensor Toolbox and Cyclops Tensor Framework respectively. Our parallel algorithms show $4.1\times$ speedup on multicore Intel Core i7 and $18.8\times$ speedup on NVIDIA K40c GPU over our sequential SpTTM respectively. 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 intend to do further optimization on SpTTM to increase the dense matrix reuse and better handle the load-balance issue. From our experiments and GPU memory limitation, we will extend our algorithms to multi-GPU platforms to support larger sparse tensors. Besides, more tensor operations will be integrated into our Sparse Tensor Operation Library (SpTOL), such as Matriced Tensor Times Khatri-Rao Product (MTTKRP) and general tensor contraction.

Acknowledgment

Thanks for Prof. Edgar Solomonik's help with CTF experiments. 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, and National Nature Science Foundation of China (61671196,61327902).

References

[1] C.-F. V. Latchoumane, F.-B. Vialatte, J. Solé-Casals, M. Maurice, S. R. Wimalaratna, N. Hudson, J. Jeong, and A. Cichocki, "Multiway array decomposition analysis of eegs in alzheimer's disease," *Journal of neuroscience methods*, vol. 207, no. 1, pp. 41–50, 2012.

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

[3] J. C. Ho, J. Ghosh, S. R. Steinhubl, W. F. Stewart, J. C. Denny, B. A. Malin, and J. Sun, "Limestone: High-throughput candidate phenotype generation via tensor factorization," *Journal of biomedical informatics*, vol. 52, pp. 199–211, 2014.

[4] J. C. Ho, J. Ghosh, and 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*, ser. KDD '14. New York, NY, USA: ACM, 2014, pp. 115–124.

[5] Y. Wang, R. Chen, J. Ghosh, J. C. Denny, A. Kho, Y. Chen, B. A. Malin, and 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*, ser. KDD '15. New York, NY, USA: ACM, 2015, pp. 1265–1274.

[6] U. Kang, E. Papalexakis, A. Harpale, and C. Faloutsos, "Gigatensor: Scaling tensor analysis up by 100 times - algorithms and discoveries," in *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '12. New York, NY, USA: ACM, 2012, pp. 316–324.

[7] D. Lahat, T. Adal, and C. Jutten, "Challenges in multimodal data fusion," in *Signal Processing Conference (EUSIPCO), 2014 Proceedings of the 22nd European*, Sept 2014, pp. 101–105.

[8] 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, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org.

[9] A. Anandkumar, R. Ge, D. Hsu, S. M. Kakade, and M. Telgarsky, "Tensor decompositions for learning latent variable models," *J. Mach. Learn. Res.*, vol. 15, no. 1, pp. 2773–2832, Jan. 2014.

[10] E. E. Papalexakis, C. Faloutsos, and 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*, ser. ECML PKDD'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 521–536.

[11] A. Novikov, D. Podoprikhin, A. Osokin, and D. Vetrov, "Tensorizing neural networks," *CoRR*, vol. abs/1509.06569, 2015.

[12] I. Perros, R. Chen, R. Vuduc, and J. Sun, "Sparse hierar-

chical tucker factorization and its application to healthcare," *IEEE International Conference on Data Mining (ICDM)*, 2015.

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

[14] F. Huang, N. Niranjan U., I. Perros, R. Chen, J. Sun, and A. Anandkumar, "Scalable Latent Tree Model and its Application to Health Analytics," *ArXiv e-prints*, Jun. 2014.

[15] E. D. Napoli, D. Fabregat-Traver, G. Quintana-Ort, and P. Bientinesi, "Towards an efficient use of the BLAS library for multilinear tensor contractions," *Applied Mathematics and Computation*, vol. 235, pp. 454 – 468, 2014.

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

[17] E. Solomonik, J. Demmel, and T. Hoefler, "Communication lower bounds for tensor contraction algorithms," ETH Zürich, Tech. Rep., 2015.

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

[19] B. W. Bader, T. G. Kolda *et al.*, "Matlab tensor toolbox (Version 2.6)," Available online, February 2015. [Online]. Available: http://www.sandia.gov/~tgkolda/TensorToolbox/

[20] E. Solomonik, D. Matthews, J. Hammond, and J. Demmel, "Cyclops Tensor Framework: reducing communication and eliminating load imbalance in massively parallel contractions," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2012-210, Nov 2012.

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

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

[23] 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.

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

[25] Intel, "Math kernel library," http://developer.intel.com/software/products/mkl/.

[26] E. Solomonik and T. Hoefler, "Sparse Tensor Algebra as a Parallel Programming Model," *ArXiv e-prints*, Nov. 2015.

[27] NVIDIA, "cuSPARSE library," *DU-06709-001_v8.0*, 2016.

[28] 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.

[29] R. Yuster and U. Zwick, "Fast sparse matrix multiplication," *ACM Trans. Algorithms*, vol. 1, no. 1, pp. 2–13, Jul. 2005.

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

[31] M. McCourt, B. Smith, and H. Zhang, "Sparse matrix-matrix products executed through coloring," *SIAM Journal on Matrix Analysis and Applications*, vol. 36, no. 1, pp. 90–109, 2015.

[32] S. Dalton, L. Olson, and N. Bell, "Optimizing sparse matrix&mdash;matrix multiplication for the gpu," *ACM Trans. Math. Softw.*, vol. 41, no. 4, pp. 25:1–25:20, Oct. 2015.

[33] W. Liu and B. Vinter, "An efficient gpu general sparse matrix-matrix multiplication for irregular data," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, May 2014, pp. 370–381.

[34] A. Azad, G. Ballard, A. Buluc, J. Demmel, L. Grigori, O. Schwartz, S. Toledo, and S. Williams, "Exploiting multiple levels of parallelism in sparse matrix-matrix multiplication," arXiv, Tech. Rep. 1510.00844, October 2015.

[35] G. Ballard, A. Druinsky, N. Knight, and O. Schwartz, "Hypergraph partitioning for sparse matrix-matrix multiplication," arXiv, Tech. Rep. 1603.05627, 2016.

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