A Pipeline Computing Method of SpTV for Three-Order Tensors on CPU and GPU

WANGDONG YANG, Hunan University and Hunan City University
KENLI LI, Hunan University
KEQIN LI, Hunan University and State University of New York

Tensors have drawn a growing attention in many applications, such as physics, engineering science, social networks, recommended systems. Tensor decomposition is the key to explore the inherent intrinsic data relationship of tensor. There are many sparse tensor and vector multiplications (SpTV) in tensor decomposition. We analyze a variety of storage formats of sparse tensors and develop a piecewise compression strategy to improve the storage efficiency of large sparse tensors. This compression strategy can avoid storing a large number of empty slices and empty fibers in sparse tensors, and thus the storage space is significantly reduced. A parallel algorithm for the SpTV based on the high-order compressed format based on slices is designed to greatly improve its computing performance on graphics processing unit. Each tensor is cut into multiple slices to form a series of sparse matrix and vector multiplications, which form the pipelined parallelism. The transmission time of the slices can be hidden through pipelined parallel to further optimize the performance of the SpTV.

CCS Concepts: • **Theory of computation → Parallel algorithms**; • **Computing methodologies → Parallel algorithms**; • **Computer systems organization → Single instruction, multiple data**;

Additional Key Words and Phrases: Pipeline, slices, tensor, tensor and vector multiplication

ACM Reference format:
Wangdong Yang, Kenli Li, and Keqin Li. 2019. A Pipeline Computing Method of SpTV for Three-Order Tensors on CPU and GPU. *ACM Trans. Knowl. Discov. Data* 13, 6, Article 63 (October 2019), 27 pages.
https://doi.org/10.1145/3363575

1 INTRODUCTION

1.1 Motivation

Q1 These applications arise in numerous domains [Kolda and Bader 2009; Qi et al. 2007], including neurosciences [Osorio and Bhavaraju 2013], health care analytics [Wang et al. 2015], recommended

The research was partially funded by the National Key R&D Program of China (Grant nos. 2018YFB1003401), the National Natural Science Foundation of China (Grant nos. 61872127, 61572175, 61751204, 61472124), the National Outstanding Youth Science Program of National Natural Science Foundation of China (Grant no. 61625202), the Key Program of National Natural Science Foundation of China (Grant no. 61432005), and the Natural Science Foundation of Hunan Province, China (Grant no. 2018JJ2022).

Q2 Authors' addresses: W. Yang, Hunan University, College of Information Science and Engineering, Changsha, Hunan 410008, China; email: yangwangdong@163.com; K. Li and K. Li, Hunan University, College of Information Science and Engineering, Changsha, China; emails: lkl@hnu.edu.cn, lik@newpaltz.edu.

1556-4681/2019/10-ART63 $15.00
https://doi.org/10.1145/3363575

systems [Rendle et al. 2009], natural language processing [Bouchard et al. 2015], signal processing [Lathauwer and Moor 1998], machine learning [Sidiropoulos et al. 2017], and social network analytics [Nakatsuji et al. 2017]. 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. (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 most of the popular tensor decompositions, including the CANDECOMP/PARAFAC (CP) decomposition [Carroll and Chang 1970; Cattell 1944] and Tucker decomposition [Tucker 1966], depend critically on having a fast sparse tensor-vector multiplication (SpTV). The sizes of tensors that arise in the above applications grow with the expansion of application scales. Compared with the matrix, the computation of SpTV is more complex. Furthermore, the tensors have obvious sparsity in the multidimensional data space. The compressed storage of the sparse tensor is more complex than the sparse matrix. Therefore, it faces great challenges to improve the computing performance of the SpTV.

Using the powerful parallel computing ability of GPUs to improve the performance of tensor decomposition is a hot research topic. Especially for GPU- and CPU-based heterogeneous computing platforms, how to make full use of computing resources to maximize the parallel computing ability is the key to improve the performance of SpTV. Firstly, a reasonable partition of tensor for the characteristics of parallel computing is the basis of improving performance. Secondly, the concurrency of multiple blocks is very important to improve the parallel efficiency of CPUs and GPUs [Yang et al. 2015]. Reducing the synchronization between multiple processing steps can improve the concurrency of multiple blocks. Finally, an asynchronous mechanism can improve the whole parallel processing ability of the system.

1.2 Our Contributions

This article makes the following contributions to parallel pipeline computations of the SpTV on GPUs and CPUs.

- We analyze a variety of storage formats of tensors and extend the compression method of sparse matrices to design the corresponding compression schemes of sparse tensors, such as the high-order COO (HOCOO) format and the high-order CSR (HOCSR) format.
- A compressed fibers based (CFB) on buckets format is designed, and the CFB can avoid storing a large number of empty fibers in a sparse tensor to improve compression efficiency.
- We define a compressed data format, i.e., high-order compressed format based on slices (HOCFS) to store the sparse tensors according to a dimension, which can avoid storing a large number of empty slices and empty fibers in the sparse tensors.
- A parallel algorithm for SpTV based on the compression format (HOCFS) is designed to greatly improve its computing performance on GPUs. The tensor is cut into multiple slices to form a series of sparse matrix and vector multiplications (SpMV) operations, which form pipelined parallelism. The transmission time of the slices can be hidden through the pipelined parallel to further optimize the performance of SpTV.

The remainder of the article is organized as follows. In Section 2, we review the related research on tensor. In Section 3, we review the programming modeling of GPUs. In Section 4, we introduce notations and preliminaries of tensors. In Section 5, we analyze and develop the compressed strategies of tensors. In Section 6, we analyze the density of partitions to get the best density threshold. In Section 7, we describe the GPU and CPU parallel computing methods and pipeline optimization strategy for SpTV. In Section 8, we demonstrate our extensive experimental performance comparison results. In Section 9, we conclude the article.

2 RELATED WORK

Tensor decompositions originated with Hitchcock [1927], and the idea of a multiway model is attributed to Cattell [1944]. These concepts received scant attention until the work of Tucker [1966] and Carroll and Chang [1970] and Harshman [1970], all of which appeared in the psychometrics literature. Appellof and Davidson [1983] are generally credited as the first to use tensor decompositions (in 1981) in chemometrics, and tensors have since become extremely popular in that field [Bro 1997, 1998], even spawning a book in 2004 [Booth 2004]. In parallel to the development in psychometrics and chemometrics, there was a great deal of interest in the decompositions of bilinear forms in the field of algebraic complexity (see, e.g., Knuth [1973]). The most interesting example of this is the Strassen matrix multiplication, which is an application of a decomposition.

Several methods have been proposed to alleviate the need to store the entire tensor in high-performance memory. Biased random sampling was used in Papalexakis et al. [2012], and was shown to work well for sparse tensors, albeit without identifiability guarantees. In Sidiropoulos et al. [2014a], the big tensor was randomly compressed into a smaller tensor. If the big tensor admits a low-rank decomposition with sparse latent factors, the random sampling guarantees the identifiability of the low-rank decomposition of the big tensor from that of the smaller tensor. However, this guarantee may not hold if the latent factors are not sparse. In Sidiropoulos et al. [2014b], a method of randomly compressing the big tensor into multiple small tensors (PARACOMP) is proposed, where each small tensor is independently decomposed, and the decompositions are related through a master linear equation. The tensor data are accessed only once during the compression stage, and further operations are only performed on the smaller tensors.

One of the most widely used tensor implementations is the Tensor Toolbox. By carefully choosing which modes to compute with finer granularity, the intermediate data remain within the working memory. The Tensor Toolbox suffers from excessive data copies according to our experiments, which motivates our in-place approach. The Cyclops Tensor Framework (CTF) [Solomonik et al. 2013] provides another baseline implementation. CTF is a recent HPC implementation with two levels of parallelism (OpenMP and MPI), which focuses on communication reductions for symmetric tensor contractions that arise frequently in quantum chemistry. The TTM is a specific instance of tensor contraction [Li et al. 2017][Hirata 2003], a mature implementation that focuses on synthesizing code and dynamically maintains the load balance on distributed systems. The TCE (Tensor Contraction Engine) [Hirata 2003] also builds a model to choose the optimal data layout, while we choose from different matrix shapes. The Matricized Tensor Times Khatri-Rao Product (MTTKRP) is an essential step of CANDECOMP/PARAFAC Decomposition, and differs from general TTM in that the matrix is the result of the Khatri-Rao product of two matrices. Ravindran et al. created an in-place tensor-matrix product for MTTKRP [Smith et al. 2015], but their implementation operates on the "slice" representation of the tensor. Our work takes advantage of a more general sub-tensor representation, and in particular its opportunities for performance tuning. A number of sparse implementations have been proposed as well. The GigaTensor [Kang et al. 2012] restructures the MTTKRP as a series of Hadamard products in the MapReduce framework, which increases the parallelism at the cost of more work. DFacTo [Choi and Vishwanathan 2014] restructures the operation as a series of sparse distributed matrix-vector multiplies. Tew [2016] explored several sparse tensor formats while ultimately evaluating two implementations; one based on explicitly storing coordinates and one that compresses these coordinates. A sparse tensor operation method based on F-COO compressed storage format is provided in the [Liu et al. 2017]. But the literature does not provide open source codes. The CFB format described in our article is similar to the F-COO format. Alen [ATe n.d.] provides sparse tensor algorithms on CPU and GPU, but its Level 1,2 BLAS is implemented using the existing libraries of CPU and GPU, not directly providing

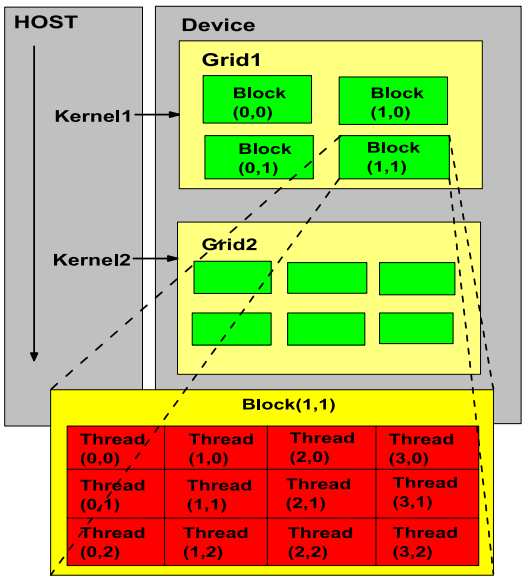


Fig. 1. The heterogeneous parallel computing system based on CUDA.

SpTV functions. The compression format of sparse tensor in Alen is similar to HOCSR format. The tensor and vector multiplication operations used in tensor operations are similar to the implementation of Algorithm 4 in our article. A Parallel Tensor Infrastructure (ParTI!) [Jiajia Li 2017] is to support fast essential sparse tensor operations and tensor decompositions on multicore CPU and GPU architectures. The sCOO compressed storage format of sparse tensor is provided in ParTI! is similar to The CFB format described in our article. Splatt [Smith et al. 2015] provides a CSF compression format for extended CSR compression algorithm, and provides a parallel computing method based on openMP. Tensor toolbox [Bader et al. 2015] and tensor lab [Sorber et al. 2016] are function libraries about tensors in matlab, which provide operation for sparse tensors, but tensor lab does not directly provide SpTV operation. Tensor Toolbox provides tensor operation functions for different levels, and uses sparse tensor compression format similar to HOCOO format in our article. Fastor [Poya et al. 2017], FTensor [FTe 2018], and ITensor [Matthew et al. 2017] provide basic function libraries for dense tensors, and are difficult to adapt to large scale sparse tensors.

3 HETEROGENEOUS PROGRAMMING ARCHITECTURE

The modern 3D graphics processing unit (GPU) has evolved from a fixed-function graphics pipeline to a programmable parallel processor with computing power exceeding that of multicore CPUs. For the GPU architecture, CUDA (Compute Unified Device Architecture) was provided by NVIDIA to improve the efficiency of programming on GPUs. CUDA is a complete GPGPU solution that provides direct access to the hardware interface, rather than the traditional approach that must rely on the graphical interface API. The heterogeneous parallel computing system based on CPUs and GPUs can be built using CUDA, as shown Figure 1. Data transfer has some impacts on the performance of GPGPUs because of the bus bandwidth restrictions, which connect with the host. Therefore, the asynchronous data transmission mechanism was provided by CUDA to reduce the synchronization between transmission and computing. In order to facilitate concurrent executions between the host and the device, some function calls of CUDA are asynchronous. Control is returned to the host thread before the device has completed the requested task. Some GPUs

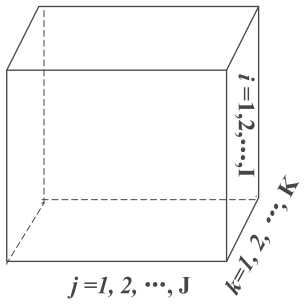


Fig. 2. A third-order tensor: $\mathcal{X}^{I \times J \times K}$.

of compute capability 1.1 and higher can perform copies between the page-locked host memory and the device memory concurrently with the kernel execution. Some GPUs of compute capability 2.x can execute multiple kernels concurrently and can perform a copy from the page-locked host memory to the device memory concurrently with a copy from the device memory to the page-locked host memory. Applications manage concurrency through streams in CUDA. A stream is a sequence of commands (possibly issued by different host threads) that execute in order. However, different streams may execute their commands out of order with respect to one another or concurrently. This behavior is not guaranteed and should therefore not be relied upon for correctness [NVIDIA 2013].

4 NOTATIONS AND PRELIMINARIES OF TENSORS

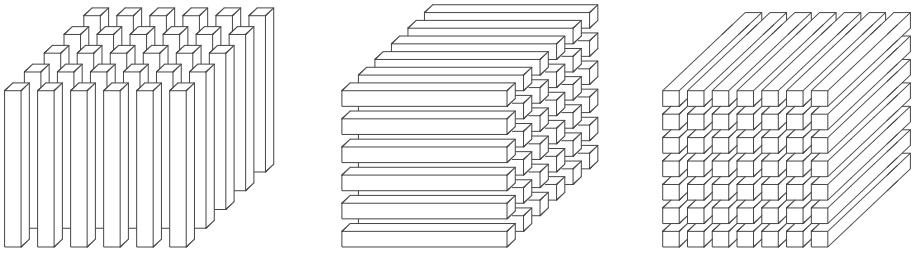
Tensors (i.e., multi-way arrays) are denoted by bold-faced Euler script letters (e.g., $\mathcal{X}$). The order of a tensor is the number of dimensions, also known as ways or modes, as shown in Figure 2. Matrices are denoted by bold-faced capital letters (e.g., $\mathbf{A}$), vectors are denoted by bold-faced lowercase letters (e.g., $\mathbf{a}$), and scalars are denoted by lowercase letters, e.g., a. The ith entry of vector $\mathbf{a}$ is denoted by a_i, element (i, j) of a matrix $\mathbf{A}$ by a_{ij}, and element (i, j, k) of a third-order tensor $\mathcal{X}$ by x_{ijk}. Indices typically range from 1 to their capital version (e.g., $i = 1, \ldots, I$). The nth element in a sequence is denoted by a superscript in parentheses. For example, $\mathbf{A}^{(n)}$ denotes the nth matrix in a sequence.

4.1 Vectorization of a Tensor

Fibers are the higher order analogues of matrix rows and columns. A fiber is defined by fixing every index but one. A matrix column is a mode-1 fiber and a matrix row is a mode-2 fiber. Third-order tensors have column, row, and tube fibers, denoted by $\mathbf{x}_{:jk}$, $\mathbf{x}_{i:k}$, and $\mathbf{x}_{ij:}$, respectively, as shown in Figure 3. Fibers are always assumed to be column vectors.

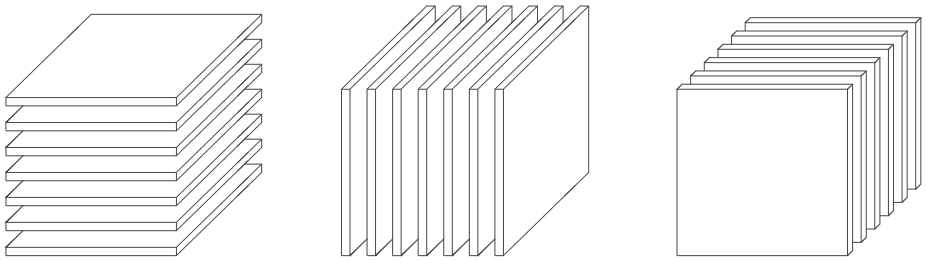
4.2 Matricization of a Tensor

Matricization, which is also known as unfolding or flattening, is the process of reordering the elements of an N-way array into a matrix, as shown in Figure 4. For an instance, a $2 \times 3 \times 4$ tensor can be arranged as a 6×4 matrix or a 2×12 matrix, and so on. In this article, we consider only a special case of a mode-n matricization because it is the only form relevant to our discussion. A more general treatment of matricization can be found in Kolda [Kolda and Bader 2009]. The mode-n matricization of a tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ is denoted by $\mathbf{X}(n)$ and arranges the mode-n fibers to be the columns of the matrix. Although conceptually simple, the formal notation is



(a) Mode-1 (column) fibers: $\mathbf{x}_{:jk}$ (b) Mode-2 (row) fibers: $\mathbf{x}_{i:k}$ (c) Mode-3 (tube) fibers: $\mathbf{x}_{ij:}$

Fig. 3. Vectorization of a third-order tensor.



(a) Horizontal slices: $\mathbf{X}_{i::}$ (b) Lateral slices: $\mathbf{X}_{:j:}$ (c) Frontal slices: $\mathbf{X}_{::k}$ (or $\mathbf{X}_k$)

Fig. 4. Matricization of a third-order tensor.

clunky. The tensor element $(i_1, i_2, \ldots, i_N)$ is mapped to the matrix element (i_n, j) where

$$j = 1 + \sum_{k=1, k\neq n}^{N} (i_n - 1)J_k,$$

with

$$J_k = \prod_{m-1, m\neq n}^{k-1} I_m.$$

4.3 The n-Mode Product of a Tensor

Tensors can be multiplied together, although the notation and symbols for this are obviously much more complex than for matrices. A full treatment of tensor multiplication is proposed by Kolda and Bader [2009]. Here, we consider only the tensor n-mode product (i.e., multiplying a tensor by a matrix (or a vector) in mode n). The n-mode (matrix) product of a tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ with a matrix $\mathbf{U} \in \mathbb{R}^{J \times I_n}$ is denoted by $\mathcal{X}_{\times \mathbf{n}} \mathbf{U}$ and is of size $I_1 \cdot \cdots \cdot I_{n-1} \cdot J \cdot I_{n+1} \cdot \cdots \cdot I_N$. Elementwise, we have

$$\mathcal{X}_{\times \mathbf{n}} \mathbf{U}_{\mathbf{i}_1 \cdots \mathbf{i}_{n-1} \mathbf{j} \mathbf{i}_{n+1} \mathbf{i}_N} = \sum_{\mathbf{i}_n - \mathbf{1}}^{\mathbf{I}_n} \mathbf{x}_{\mathbf{i}_1 \mathbf{i}_2 \cdots \mathbf{i}_N} \mathbf{u}_{\mathbf{j} \mathbf{i}_n}.$$

Each mode-n fiber is multiplied by the matrix $\mathbf{U}$. The idea can also be expressed in terms of unfolded tensors: $\mathcal{Y} = \mathcal{X}_{\times \mathbf{n}}\mathbf{U}, \mathbf{Y}_{(\mathbf{n})} = \mathbf{U}\mathbf{X}_{(\mathbf{n})}$. As an example, let the slices of $\mathcal{X}^{3\times4\times2}$ is

$$\mathbf{X}_1 = \begin{bmatrix} 1 & 4 & 7 & 0 \\ 2 & 5 & 8 & 11 \\ 3 & 6 & 9 & 12 \end{bmatrix},$$

$$\mathbf{X}_2 = \begin{bmatrix} 13 & 16 & 19 & 22 \\ 14 & 17 & 20 & 23 \\ 15 & 18 & 21 & 24 \end{bmatrix}.$$

Assume

$$\mathbf{U} = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}.$$

Then, the product $\mathcal{Y} = \mathcal{X}_{\times \mathbf{n}}\mathbf{U} \in \mathbb{R}^{2\times4\times2}$ is

$$\mathbf{Y}_1 = \begin{bmatrix} 22 & 49 & 76 & 103 \\ 28 & 64 & 100 & 136 \end{bmatrix},$$

$$\mathbf{Y}_2 = \begin{bmatrix} 130 & 157 & 184 & 211 \\ 172 & 208 & 244 & 280 \end{bmatrix}.$$

5 THE COMPRESSED STORAGE OF A TENSOR

For example, a sparse tensor $\mathcal{X}^{4\times5\times4}$ is split into four slices according to the first dimension, we have

$$\mathcal{X}_{(1::)} = \begin{bmatrix} 3 & 1 & 0 & 0 \\ 0 & 9 & 0 & 0 \\ 0 & 7 & 3 & 0 \\ 0 & 0 & 5 & 8 \\ 2 & 0 & 0 & 0 \end{bmatrix}, \mathcal{X}_{(2::)} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix},$$

$$\mathcal{X}_{(3::)} = \begin{bmatrix} 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 7 \\ 5 & 0 & 0 & 2 \\ 0 & 1 & 6 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, \mathcal{X}_{(4::)} = \begin{bmatrix} 0 & 7 & 0 & 8 \\ 3 & 0 & 0 & 0 \\ 0 & 0 & 2 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 6 & 5 \end{bmatrix}.$$

5.1 High-Order COO Format (HOCOO)

The coordinate (COO) format is a particularly simple storage scheme with a triplet (*row*, *column*, *value*) for a sparse matrix. The array's *rows*, *columns*, and *values* store the row indices, the column indices, and the values of the non-zero elements in a matrix respectively. The COO format can be extended to high-dimensional data, which uses four tuple (**Ti**, **Tj**, **Tk**, **Tv**) to store the three-order tensor. The arrays **Ti**, **Tj**, **Tk**, and **Tv** store the indices of the first, second, third dimensions, and the values of the non-zero elements in a tensor, respectively. For the example sparse tensor $\mathcal{X}^{4\times5\times4}$, we have

$$\mathbf{Ti} = (1, 1, 1, 1, 1, 1, 1, 1, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4, 4),$$

$$\mathbf{Tj} = (1, 1, 2, 3, 3, 4, 4, 5, 1, 2, 3, 3, 4, 4, 1, 1, 2, 3, 3, 5, 5),$$

$$\mathbf{Tk} = (1, 2, 2, 2, 3, 3, 4, 1, 3, 4, 1, 4, 2, 3, 2, 4, 1, 3, 4, 3, 4),$$

$$\mathbf{Tv} = (3, 1, 9, 7, 3, 5, 8, 2, 4, 7, 5, 2, 1, 6, 7, 8, 3, 2, 1, 6, 5).$$

The length of each array is the number of non-zeros (Abbreviation: *NNZ*) in the sparse tensor. For a sparse tensor $\mathcal{X}^{\mathbf{I}\times\mathbf{J}\times\mathbf{K}}$, there are *NNZ* non-zero elements. If the tensor is stored by HOCOO according to the first dimension, the sizes of **Ti**, **Tj**, **Tk**, and **Tv** are *NNZ*. The total size of storage using HOCOO is 4*NNZ*.

5.2 High-Order CSR Format (HOCSR)

The compressed sparse row (CSR) format is a popular and general-purpose sparse matrix representation scheme. The CSR explicitly stores the column indices and non-zero values in arrays **Aj** and **Av**. The third array **Ap** represents the starting position of each row in the array **Aj**. For an N-by-M matrix, **Ap** has length $N + 1$ and stores the offset of the ith row in $\mathbf{Ap}[i]$. The value of the last element is *NNZ*. For the example sparse matrix

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 2 & 0 \\ 0 & 0 & 7 & 0 \\ 0 & 5 & 0 & 8 \\ 0 & 1 & 0 & 3 \end{bmatrix},$$

we have

$$\begin{aligned} \mathbf{Ap} &= (0 \quad 2 \quad 3 \quad 5 \quad 7), \\ \mathbf{Aj} &= (0 \quad 2 \quad 2 \quad 1 \quad 3 \quad 1 \quad 3), \\ \mathbf{Av} &= (1 \quad 2 \quad 7 \quad 5 \quad 8 \quad 1 \quad 3). \end{aligned}$$

The CSR format for the sparse matrix can be extended to the three-order sparse tensor. A tensor is unfolded into a matrix $(\mathcal{X}_{1::}^{\mathbf{T}}\ \mathcal{X}_{2::}^{\mathbf{T}}\ \ldots\ \mathcal{X}_{\mathbf{I}::}^{\mathbf{T}})^{\mathbf{T}}$ according to the 1st dimension. The matrix has $T \cdot J$ rows. Define two arrays **Tk** and **Tv** to store the dimension K indices and the value of the non-zero elements respectively. Define an array **Tj** to represent the starting position of each row of the matrix in the array **Tk**. For the example sparse tensor $\mathcal{X}^{4\times5\times4}$, we have

$$\begin{aligned} \mathbf{Tj} &= (1, 3, 4, 6, 8, 9, 9, 9, 9, 9, 9, 10, 11, 13, 15, 15, 17, 18, 20, 20, 22), \\ \mathbf{Tk} &= (1, 2, 2, 2, 3, 3, 4, 1, 3, 4, 1, 4, 2, 3, 2, 4, 1, 3, 4, 3, 4), \\ \mathbf{Tv} &= (3, 1, 9, 7, 3, 5, 8, 2, 4, 7, 5, 2, 1, 6, 7, 8, 3, 2, 1, 6, 5). \end{aligned}$$

There are some redundant data in array **Tj** because of many empty rows in the matrix $(\mathcal{X}_{1::}^{\mathbf{T}}\ \mathcal{X}_{2::}^{\mathbf{T}}\ \ldots\ \mathcal{X}_{\mathbf{I}::}^{\mathbf{T}})^{\mathbf{T}}$. The size of **Tj** is more than the number of non-zero elements for the example sparse tensor. For a sparse tensor $\mathcal{X}^{\mathbf{I}\times\mathbf{J}\times\mathbf{K}}$, there are *NNZ* non-zero elements. If the tensor is stored by HOCSR according to the 1st dimension, the sizes of **Tj**, **Tk**, and **Tv** are $I \cdot J$, *NNZ*, and *NNZ*, respectively. The total size of the storage using HOCSR is $I \cdot J$+2*NNZ*.

5.3 Compressed Fibers Based on Buckets Format (CFB)

A tensor is transformed into a series of fibers $\mathcal{X}_{:\mathbf{jk}}$ according to the dimension I, and for a three-order tensor, the other two dimensions are constructed into a matrix $\mathcal{X}_{\mathbf{i}::}$, whose elements are the fibers. Define a series array's **fibers** as buckets to store the non-zero elements of the fibers is $\mathcal{X}_{:\mathbf{jk}}$. Furthermore, two arrays **Tb** and **Tn** are defined to store the indices and *NNZ* of the fibers $\mathcal{X}_{:\mathbf{jk}}$ in the $\mathcal{X}_{\mathbf{i}::}$. The index of the fibers $\mathcal{X}_{:\mathbf{jk}}$ is $j \cdot K + k$. The element of the array's **fibers** is a two-tuple (**idx**, **value**), where the **idx** and **value** store the index for the dimension I and the value of the non-zero

Table 1. The Storage Structure of $\mathcal{X}^{4\times5\times4}$ Based on the HOCFS Format

Slices	$format$	No	num	fibers	Ap	Aj	Av
1	1	1	5	1,2,3,4,5	0,2,3,5,7,8	1,2,2,2,3,3,4,1	3,1,9,7,3,5,8,2
2	1	3	4	1,2,3,4	0,1,2,4,6	3,4,1,4,2,3	4,7,5,2,1,6
3	1	4	4	1,2,3,5	0,2,3,5,7	2,4,1,3,4,3,4	7,8,3,2,1,6,5

element in the fiber $\mathcal{X}_{:\mathbf{jk}}$, respectively. For the example sparse tensor $\mathcal{X}^{4\times5\times4}$, we have

$$\mathbf{Tb} = (1, 2, 3, 5, 6, 8, 9, 10, 11, 12, 14, 15, 16, 17, 19, 20),$$
$$\mathbf{Tn} = (1, 2, 1, 1, 1, 1, 1, 1, 1, 2, 1, 2, 1, 1, 1, 1),$$
$$\begin{aligned}
\mathbf{fibers}[\mathbf{1}] &= (1,3), & \mathbf{fibers}[\mathbf{2}] &= (1,1),(4,7)\\
\mathbf{fibers}[\mathbf{3}] &= (3,4), & \mathbf{fibers}[\mathbf{4}] &= (4,3)\\
\mathbf{fibers}[\mathbf{5}] &= (1,9), & \mathbf{fibers}[\mathbf{6}] &= (3,7)\\
\mathbf{fibers}[\mathbf{7}] &= (3,5), & \mathbf{fibers}[\mathbf{8}] &= (1,7)\\
\mathbf{fibers}[\mathbf{9}] &= (1,3), & \mathbf{fibers}[\mathbf{10}] &= (3,2),(4,1)\\
\mathbf{fibers}[\mathbf{11}] &= (3,1), & \mathbf{fibers}[\mathbf{12}] &= (1,5),(3,6)\\
\mathbf{fibers}[\mathbf{13}] &= (1,8), & \mathbf{fibers}[\mathbf{14}] &= (1,2)\\
\mathbf{fibers}[\mathbf{15}] &= (4,6), & \mathbf{fibers}[\mathbf{16}] &= (4,5).
\end{aligned}$$

For a sparse tensor $\mathcal{X}^{\mathbf{I}\times\mathbf{J}\times\mathbf{K}}$, there are NNZ non-zero elements. If the tensor is stored by the CFB according to the 1st dimension, the size of the **fibers** is $2NNZ$. The sizes of **Tb** and **Tn** are the number of non-empty fibers. The total size of the storage using CFB is $2NNZF + 2NNZ$, where $NNZF$ is the number of non-empty fibers. Due to the sparsity of the dimension, $NNZF$ is less than $I \cdot J$, and the storage space of CFB is less than that of HOCSR.

5.4 High-Order Compressed Format Based on Slices (HOCFS)

A tensor is unfolded into a set of slices by matrixing according to a dimension, which can be stored by the compressed format of the matrix. There are many zero elements that may lead to empty slices without non-zero elements because of the sparsity of the tensor. Furthermore, most slices of the sparse tensor are sparse. Therefore, the sparse tensor could be compressed by the main dimension compression and the slice compression. The tensor is segmented into a set of slices according to one dimension, which is called the main dimension. The main dimension compression is that the empty slices of the set of slices are eliminated during storage to reduce the size of the main dimension. The sparse slices are the sparse matrices, which can be stored by some compressed formats, such as the CSR, ELL, and COO formats. We define a compressed data format HOCFS (see Figure 5) to store the sparse tensor according to a dimension.

The tensors that arise in many real-life applications are usually represented by multidimensional arrays, which are compressed to be stored by the HOCFS format as shown in Algorithm 1. There are some empty fibers without non-zero elements in the slice because of the sparsity of the slice. In many cases, the number of empty fibers is much greater than that of non-empty fibers. The **Ap** array contains a large amount of redundant data if the slice is stored by the traditional CSR or ELL format. The HOCFS format is adopted in Algorithm 1, in which only non-empty fibers are stored, and the indices of the non-empty fibers are stored by the fiber_num array to realize the corresponding consistency of the ordinal number in the operation process. The compressed storage of the tensor $\mathcal{X}^{4\times5\times4}$ is shown in Table 1 if it is compressed by the HOCFS format.

```
typedef struct {
    /*The type of compressed format for the slice 1:CSR; 2:ELL*/
    int format;
    /*Store the index of the slice in the main dimension.*/
    int No;
    /*Store the number of the rows with nonzero elements in the slice*/
    int num;.
    /*Store the indices of the rows with nonzero elements in the slice.*/
    int * fibers;
    /*For ELL format there is an elements in the Ap array, which stores
    the start position of the slice in the Aj array. For CSR format,
    the cumulative values of the NNZ's of the rows in the slice
     are stored in the Aj array.*/
     int * Ap;
    /*Store the indices of columns in the slice.*/
    int * Aj;
    /*Store the values of the elements in the slice.*/
    float * Av;
}CFS;
```

Fig. 5. The data structure of the tensor using the HOCFS format.

Q5 The second slice is not stored because it has no non-zero elements. The total size of storage space is the sum of the sizes of arrays **No**, **num**, **fibers**, **Ap**, **Aj**, and **Av**. The sizes of arrays **No** and **num** are the number of non-empty slices. The sizes of arrays **fibers** and **Ap** are the number of non-empty fibers. The sizes of arrays **Aj** and **Av** are the number of non-zero elements. The total size of storage space of tensor using HOCFS is $2NNZS+2NNZF+2NNZ$, where *NNZS*, *NNZF*, and *NNZ* are the number of non-empty slices, non-empty fibers, and non-zero elements, respectively. For the sparse tensor that arises from the recommender systems and social networking, there are many empty slices and fibers in the tensor, thus the *NNZS* and *NNZF* are less than I and $I \cdot J$ for $\mathcal{X}^{\mathbf{I}\times\mathbf{J}\times\mathbf{K}}$. For the sparse tensor, the storage space of HOCFS is less than that of HOCSR.

HOCFS can compress multiple dimensions for sparse tensors. Define a tensor $\mathcal{X} = (\mathbf{X_1}, \mathbf{X_2}, \mathbf{X_3}, \ldots, \mathbf{X_K})$, where $\mathbf{X_k}$, $k = 1, 2, 3, \ldots, K$, are low-dimensional tensors or matrices, or vectors, or scalars. $\mathbf{X_k}$ is uniformly defined as low dimensional elements. $\mathbf{X_k}$ is zero element if the content of $\mathbf{X_k}$ is empty or zero. The zero elements will not stored in HOCFS format. Algorithm 2 describes recursively how to compress high-dimensional sparse tensors by HOCFS. HOCFS can be extended to compress higher dimensional sparse tensors by Algorithm 2, and the sparse can be compressed for each dimension by dimensionality reduction.

6 PARALLEL ALGORITHM FOR SPTV

6.1 Parallel Implementation of SpTV on GPU

6.1.1 Parallel Implementation of SpTV Based on HOCOO. For a non-zero element (i, j, k, v) of the three-order tensor $\mathcal{X}$, the multiplying elements in a vector x are x_i, x_j, and x_k for SpTV using the 1-mode, 2-mode, and 3-mode, respectively. The calculation $v \times x_i$ is executed in the thread that is assigned the non-zero element (i, j, k, v).

ALGORITHM 1: The compressed algorithm for a tensor based on the HOCFS format.

Input: The three-order tensor $\mathcal{X}^{\mathbf{I}\times\mathbf{J}\times\mathbf{K}}$; Three-dimensional arrays $\mathcal{X}_{(:::)}$;The compression format of the slices *type*.
Output: The slices stored by CFS format $S_1, S_2, \ldots, S_m$.

```
idx ← 0;
for each i in [1, I] do
    if X_(i::) contains the non-zero elements then
        idx ← idx + 1;
        S_idx.No ← i;
        rid ← 0;
        for each j in [1, J] do
            if X_(ij:) contains the non-zero elements then
                rid ← rid + 1;
                S_idx.fibers[rid] ← j;
                if type == 1 then
                    S_idx.Aj, S_idx.Av, S_idx.Ap ← Compress_CSR(X_(ij:));
                else
                    if type == 2 then
                        S_idx.Aj, S_idx.Av, S_idx.Ap ← Compress_ELL(X_(ij:));
                    end
                end
            end
        end
        S_idx.num ← rid;
    end
end
```

ALGORITHM 2: The recursive compressed algorithm for a high-dimensional sparse tensors by HOCFS.

Input: The high-dimensional sparse tensor $\mathcal{X} \in \mathbb{R}^{\mathbf{I}_1\times\mathbf{I}_2\times\cdots\times\mathbf{I}_\mathbf{K}}$.
Output: The sub-tensors $\mathbf{X}_1, \mathbf{X}_2, \ldots, \mathbf{X}_\mathbf{T}$, $\mathbf{X}_\mathbf{i} \in \mathbb{R}^{\mathbf{I}_1\times\mathbf{I}_2\times\cdots\times\mathbf{I}_{\mathbf{K}-1}}$, $i = 1, 2, \ldots, T$, $T \leq K$.

```
for each i in [1, K] do
    if X_i is not the zero element. X_i ∈ R^(I_1×I_2×···×I_(K−1)) then
        Compress_HOCFS(X_i);
    end
    else
        delete X_i;
    end
end
```

The SpTV based on the HOCOO format is described in Algorithm 3. The four arrays of HOCOO can be read by only one element each time in a thread since each non-zero is assigned to a thread when the SpTV is computed on a GPU. The data accessed in each thread contains two parts: one element from each of the four arrays of HOCOO and the multiplying element in the vector.

The results of the above computation need to be summarized for each fiber by the function **segmented_reduction**, which realizes the summation of the calculated results to get the result matrix **M**. The results of one fiber are summed by one warp. The parallel reduction algorithm is used when the summation of a fiber is calculated in Algorithm 3. This approach implicitly relies on

ALGORITHM 3: The SpTV based on the HOCOO format.

Input: The three-order tensor $\mathcal{X}^{\mathbf{I}\times\mathbf{J}\times\mathbf{K}}$ using HOCOO format **Ti**, **Tj**, **Tk**, **Tv**; The number of non-zero elements, *nnz*; The vector v; The mode n.
Output: The result matrix of $\mathcal{X} \times_{\mathbf{n}} \mathbf{v}$ **M**.

```
for each idx in [1, nnz] do
    if n == 1 then
        Tv[idx] ← Tv[Ti[idx]] · v[Ti[idx]];
    else
        if n == 2 then
            Tv[idx] ← Tv[Tj[idx]] · v[Tj[idx]];
        else
            if n == 3 then
                Tv[idx] ← Tv[Tk[idx]] · v[Tk[idx]];
            end
        end
    end
end
segmented_reduction(Ti, Tj, Tk, Tv, n, M);
```

ALGORITHM 4: The SpTV based on the HOCSR format.

Input: The three-order tensor $\mathcal{X}^{\mathbf{I}\times\mathbf{J}\times\mathbf{K}}$ based on HOCSR format using mode-1 **Tj**, **Tk**, **Tv**; The vector v.
Output: The result matrix of $\mathcal{X} \times_{\mathbf{n}} \mathbf{v}$ **M**.

```
for each idx in [0, J · K − 1] do
    j ← idx / K;
    k ← idx mod K;
    for each m in [Tj[idx], Tj[idx + 1]] do
        M[j, k] ← M[j, k] + Tv[m] · v[Tk[m]];
    end
end
```

the fact that each warp processes only one fiber of the tensor. In contrast, a segmented reduction allows warps to span multiple fibers.

6.1.2 Parallel Implementation of SpTV Based on HOCSR. The implementation method of SpTV is similar to SpMV since the tensor is unfolded into a matrix using the HOCSR. Lines 10–11 can be processed in the same thread since the dot product of each row and right-vector yields a value of the resulting matrix. Different j in line 9 can be processed in different threads, and thus, the SpTV based on the HOCSR is relatively easy to be implemented in parallel computing.

6.1.3 Parallel Implementation of SpTV Based on CFB. A sparse tensor $\mathcal{X}$ is compressed by the CFB format for mode-n. The SpTV $\mathcal{X} \times_{\mathbf{n}} \mathbf{v}$ based on the CFB is described in Algorithm 4. The SpTV based on the CFB is relatively easy to be implemented in parallel computing, which is similar to that of the HOCSR. Lines 10–11 can be processed in the same thread to compute **fiber** $\times\, v$. Define the length of the array **Tb** as S, and at most S threads can be started to compute the SpTV simultaneously.

6.1.4 Parallel Implementation of SpTV Based on HOCFS. A sparse tensor is compressed into CFS arrays $S_1, S_2, \ldots, S_m$ using the HOCFS format. For HOCFS, the SpTV is composed of a series of SpMVs, which is $S_i \times v$. Each slice is multiplied by the vector and can be computed independently.

ALGORITHM 5: The SpTV based on the CFB format.

Input: The three-order tensor $\mathcal{X}^{\mathbf{I}\times\mathbf{J}\times\mathbf{K}}$ using CFB format **Tb**, **fibers**;
The vector v.
Output: The result matrix of $\mathcal{X} \times_{\mathbf{n}} \mathbf{v}$ **M**.
for *each* $Tb[b]$ *in* Tb **do**
  $j \leftarrow Tb[b] / K$;
  $k \leftarrow Tb[b] \bmod K$;
  for *each* $fiber[idx]$ *in* $fibers[b]$ **do**
    $M[j,k] \leftarrow M[j,k] + fiber[idx].v \cdot v[fiber[idx].i]$;
  end
end

ALGORITHM 6: The SpTV based on the HOCFS format.

Input: The three-order tensor $\mathcal{X}^{\mathbf{I}\times\mathbf{J}\times\mathbf{K}}$ using CFS format $S_1, S_2, \ldots, S_m$, **fibers**; The vector v.
Output: The result matrix of $\mathcal{X} \times_{\mathbf{n}} \mathbf{v}$ **M**.
for *each* i *in* $[1, m]$ **do**
  $j \leftarrow S_i.No$;
  for *each* k *in* $[1, S_i.num]$ **do**
    $k \leftarrow S_i.fiber\_num[k]$;
    if $S_i.format == 1$ **then**
      //The slice is stored by CSR format.
      for *each* idx *in* $[S_i.fiber\_cou[k-1], S_i.fiber\_cou[k]]$ **do**
        $M[j,k] \leftarrow M[j,k] + S_i.fiber\_Av[idx] \cdot v[S_i.fiber\_Aj[idx]]$;
      end
    else
      if $S_i.format == 2$ **then**
        //The slice is stored by ELL format.
        for *each* idx *in* $[1, S_i.fiber\_cou[k]]$ **do**
          $M[j,k] \leftarrow M[j,k] + S_i.fiber\_Av[idx] \cdot v[S_i.fiber\_Aj[idx]]$;
        end
      end
    end
  end
end

Furthermore, it can be computed in parallel. Each row in the slice is multiplied by the vector and also can be computed independently [Yang et al. 2018]. Thus, two level parallel computing can be used for the SpTV based on the HOCFS format. The first level parallel computing is $S_i \times v$, and the second level is the dot product of each row in S_i and the vector v.

The data grid is built from the tensor by the HOCFS using the slices as the X dimension and the rows in slices as the Y dimension. The data grid is assigned into the CUDA thread's grid as shown in Figure 6. To facilitate the construction of the data grid, a Slice_Ap array is defined to store the start position of each slice in all fiber sequences of the HOCFS. If the slices are stored in the CSR format, the storage structure of the HOCFS is shown in Figure 7, and if the slices are stored by the ELL format, the storage structure of the HOCFS is shown in Figure 8. The kernel function of the SpTV using the HOCFS with the CSR format is shown Figure 9, and the kernel function of the SpTV using the HOCFS with the ELL format is shown in Figure 10. The result of the SpTV is a

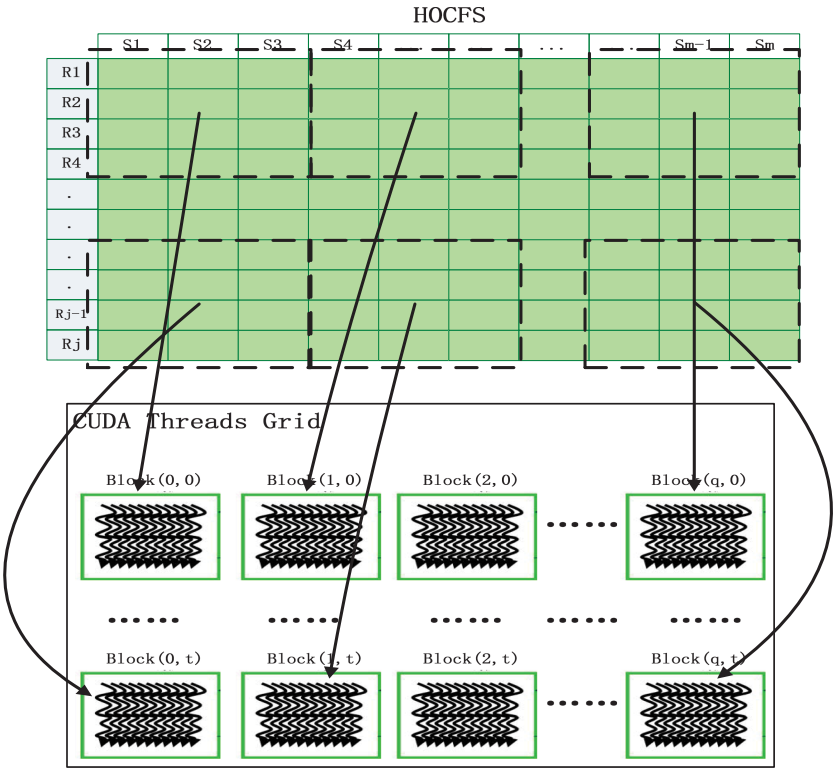


Fig. 6. The parallel computing model of the SpTV on the GPU based on the HOCFS format.

matrix, which is stored by the COO format (r, c, v), and the r, c, v arrays store the indices of the rows, columns, and the values of the elements, respectively. The number of slices is stored in the parameter CFS_num.

We adopt data parallelism for SpTV. HOCFS format divides sparse tensor into a series of slices, and then different slices are allocated to different thread blocks to perform calculations in GPU. There is no data correlation between slices, and there is no data communication between different thread blocks. The execution time of parallel programs can be calculated by $T = T_1 + T_2 + T_3$, where T_1, T_2, and T_3 are computing time, scheduling time, and communication time, respectively. GPU uses hardware to thread scheduling, and its scheduling time is almost negligible. The parallel computing performance of SpTV is mainly related to computing time if T_3 is zero. Increasing the core number of GPU can provide more thread blocks that can be used for computing simultaneously, and more slices can be computed at the same time, thus reducing the overall computation time of the SpTV.

6.2 A Pipeline Computing Method for SpTV on GPU

The pipelined computing model is illustrated in Figure 11. First, a large task is input to the CPU. Second, the large task is partitioned into some sub-tasks $(C1, C2, \ldots, Ck)$. Third, the sub-tasks $(C1, C2, \ldots, Ck)$ are executed in the GPU. They are transported into the GPU by streams, and the computed results are also obtained by the streams. Fourth, these results are merged in the CPU. Finally, the results of the large task are output. To improve the computational efficiency, the process is split into three sub-steps that input data into the GPU (step 3.1 in Figure 12), execute computations on the GPU (step 3.2 in Figure 12), and get results from the GPU (step 3.3 in Figure 12). For a

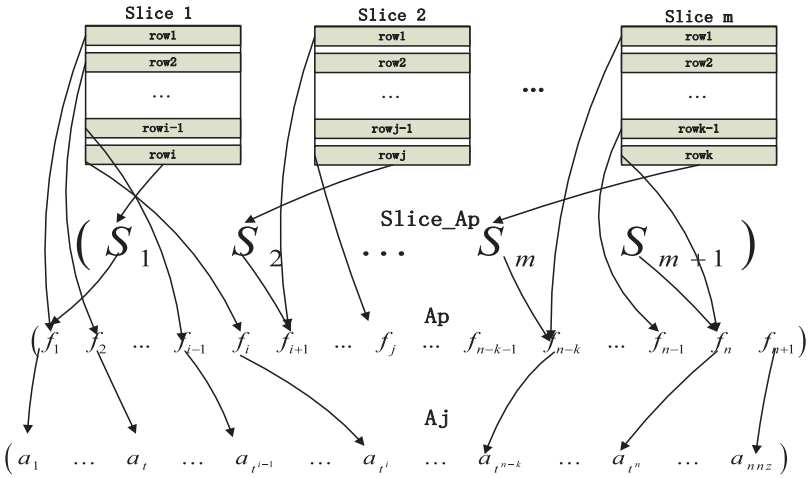


Fig. 7. The storage structure of the HOCFS based on CSR.

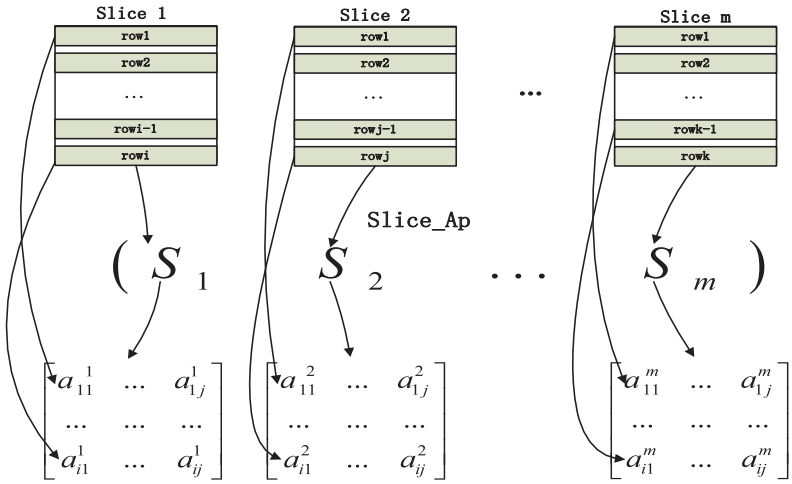


Fig. 8. The storage structure of the HOCFS based on ELL.

stream, these steps are processed sequentially. For different streams, steps 3.1, 3.2, and 3.3 can be executed concurrently to compose a pipeline in which steps 3.1 and 3.3 are asynchronous transfer processes based on the stream model, while step 3.2 is processed in the GPU.

For the SpTV using the HOCOO format, the results of different streams may be cumulative since there is no guarantee that data from the same row are assigned to the same stream. Therefore, synchronization may be needed between different streams, which results in reduced pipelining efficiency.

For the SpTV using the HOCSR format, different slices are assigned to different streams, which is similar to the SpMV. How many rows are allocated in one stream is an important factor that affects the performance of the pipeline. Too little data in one stream can reduce the transmission efficiency of the PCIe, and too much data in a stream will make the pipelining parallelism be less obvious.

For the SpTV using the CFB format, different fiber buckets are assigned to different streams, which is similar to the HOCSR since a fiber bucket stores the non-zero elements of a slice.

```
__global__ void HOCFS_CSR_SpTVKernel
(int CFS_num,int * Slice_Ap, int * fibers,int * Ap,int * Aj, float * Av,
 float * b, int * r,int * c,float * v)
{
  //Get the index of the slice
  int CFS_i= blockIdx.y*blockDim.y+ threadIdx.x;
  /*Get the position of the row which is processed in the thread.*/
  int fiber_i=Slice_Ap[CFS_i]+blockIdx.x*blockDim.x+threadIdx.x;

  if(CFS_i<CFS_num && fiber_i<Slice_Ap[CFS_i+1])
  {
        /*Get the start position of the row in the Aj array by the Ap array.*/
        int start_i = Ap[fiber_i];
        //Get the index of the row.
        int fiber_inx=fibers[fiber_i];
        /*Get the end position of the row in the Aj array by the Ap array.*/
        int end_i = Ap[fiber_i+1];
        float temp = 0;
        for( int j=start_i;j< end_i;j++)
        {
                //execute SpTV
                float val=Av[j];
                int idx=Aj[j];
                temp+=val*b[idx];
        }
        r[fiber_i]=CFS_i;
        c[fiber_i]=fiber_inx;
        v[fiber_i]=temp;
  }
}
```

Fig. 9. The kernel function of the SpTV using the HOCFS based on CSR on the GPU.

For the SpTV using the HOCFS format, different slices are assigned to different streams. The SpTV is equivalent to an independent SpMV operation for each slice.

The pipelined computational process is illustrated in Figure 13. Assume that the computational time of a tensor set using the synchronization mode is T_s and that using the pipelined mode is T_p. If each step in the pipeline does not wait for the data, the transmission time of the SpTV can be hidden. For the ith collection of the slices or buckets, assume that the times of step 3.1, step 3.2, and step 3.3 in Figure 12 are $\Delta t1_i$, $\Delta t2_i$, and $\Delta t3_i$, respectively. Firstly, the first collection is transferred into the GPU. Then, the collection is computed in the GPU and the second collection is transferred concurrently. Next, the second collection is computed when the above two steps are completed. Thus, the time is $\Delta t1_1 + \max\{\Delta t1_2, \Delta t2_1\}$ before the pipeline completely overlaps. The time of the ith cycle is $\max\{\Delta t1_{i+2}, \Delta t2_{i+1}, \Delta t3_i\}$ since the three steps are processed in parallel. The time is $\max\{\Delta t3_{k-1}, \Delta t2_k\} + \Delta t3_k$ when the pipeline exits from the complete overlap as shown in Figure 13. For k collections, T_s and T_p are calculated by the following two equations, respectively [King et al. 1990]:

$$T_s = \sum_{i=1}^{k} (\Delta t1_i + \Delta t2_i + \Delta t3_i),$$

```
__global__ void HOCFS_ELL_SpTVKernel
(int CFS_num,int * Slice_Ap, int * Ap, int * fibers, int * int * Aj,
 float * Av, float * b,  int * r,int * c,float * v)
{
  //Get the index of the slice
  int CFS_i= blockIdx.y*blockDim.y+threadIdx.x;
  /*Get the position of the row which is processed in the thread.*/
  int fiber_i=Slice_Ap[CFS_i]+blockIdx.x*blockDim.x+threadIdx.x;
  /*Get the start position of the slice in the Slice_Ap array.*/
  int CFS_start=Slice_Ap[CFS_i];
  /*Get the number of nonzero elements of the rows with the most
  nonzero elements in the slice.*/
  int row_max =(Ap[CFS_i+1]-Ap[CFS_i])/(Slice_Ap[CFS_i+1]-CFS_start);
  if(CFS_i<CFS_num && fiber_i<Slice_Ap[CFS_i+1])
  {
    /*Get the start position of the row in the Aj array by the Ap array.*/
    int start_i=Ap[CFS_i]+fiber_i*row_max;
    /*Get the end position of the row in the Aj array by the Ap array.*/
    int end_i=Ap[CFS_i]+(fiber_i+1)*row_max;
    fiber_i=CFS_start+fiber_i;
    //Get the index of the row.
    int fiber_inx=fibers[fiber_i];
    float temp=0;
    int j=start_i;
    while(j< end_i && Aj[j]>=0)
    {
        //execute SpTV
        float val=Av[j];
        int idx=Aj[j];
        temp+=val*b[idx];
        j++;
    }
    r[fiber_i]=CFS_i;
    c[fiber_i]=fiber_inx;
    v[fiber_i]=temp;
  }
}
```

Fig. 10. The kernel function of the SpTV using the HOCFS based on ELL on the GPU.

and

$$T_p = \Delta t1_1 + \max\{\Delta t1_2, \Delta t2_1\} + \sum_{i=1}^{k-2} \max\{\Delta t1_{i+2}, \Delta t2_{i+1}, \Delta t3_i\} + \max\{\Delta t3_{k-1}, \Delta t2_k\} + \Delta t3_k.$$

7 EXPERIMENTAL EVALUATION

7.1 Experiment Settings

The following test environment has been used for all benchmarks. The test computer is equipped with an Intel i7-6700 CPU with hyper-threading technology and a NVIDIA GTX1070 GPU with

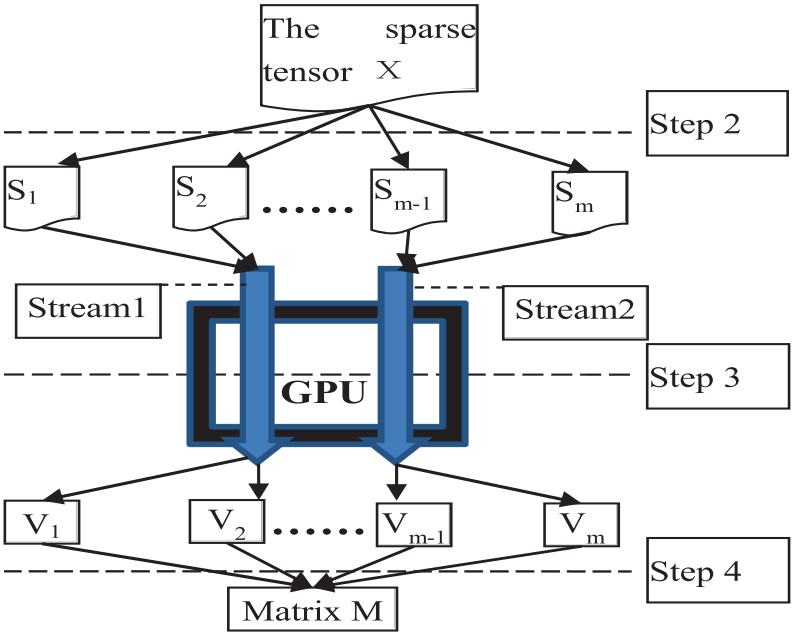


Fig. 11. The pipelined computing model on the CPU and GPU.

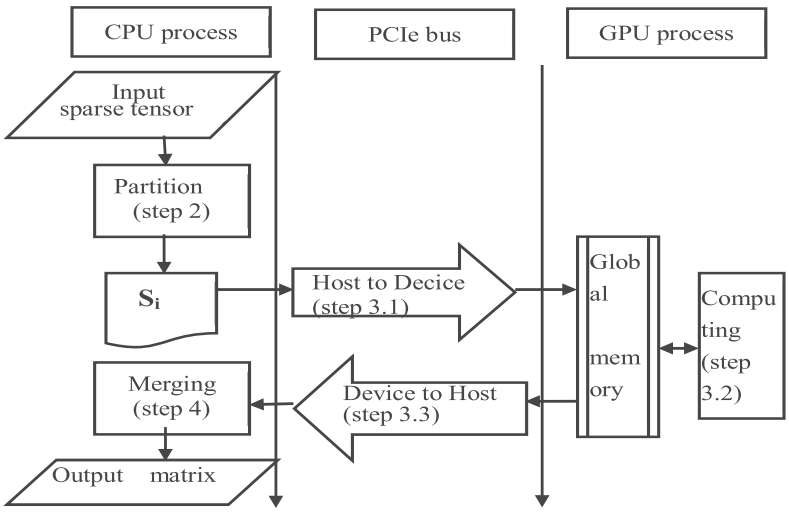


Fig. 12. The pipelined executing process of the SpTV on the GPU based on the HOCFS format.

the Pascal architecture. CUDA compute capacity of GTX1070 is 6.1, but the bus of the tested GPU only supports PCIe 3.0. The specific parameters of CPU and GPU are shown in Table 2.

We evaluate the performance of the algorithms in the article by comparing to two state-of-the-art tensor libraries, namely ParTI [Jiajia Li 2017] and SPLATT [Smith et al. 2015]. ParTI can accelerate sparse tensor operations on multicore CPU and GPU architectures. SPLATT provides high-performance implementations of sparse tensor operations on shared-memory systems. SPLATT cannot support sparse tensor operations on GPUs.

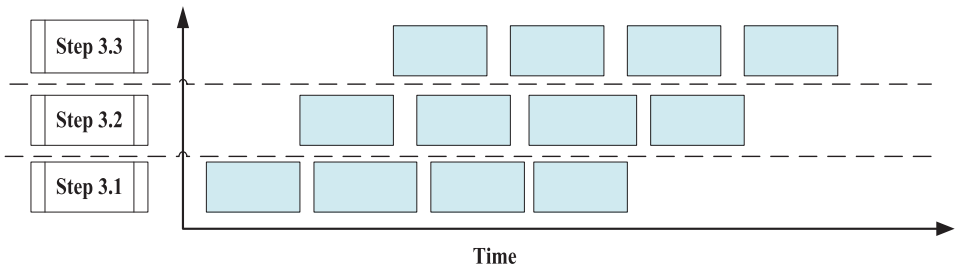


Fig. 13. The computing process by the pipeline.

Table 2. Parameters for Processors of the Test Computer

Parameters Description	CPU (i7 6700)	GPU (GTX1070)
Core number of processors	4	1,092
Working frequency of processors	3.40GHz	1.683GHz
Memory of processors	32G(DDR4)	8 GB(DDR5)
Bandwidth	136GB/s	256GB/s
Bus width of memory	256 bit (4 channels)	256 bit
Working frequency of memory	2,133MHz	2,002MHz

Table 3. General Information of the Tensors Used in the Evaluation

No.	Tensor	I_1	I_2	I_3	Non-zeros
1	user_ratedmovies	71,534	65,133	60	855,598
2	user_taggedmovies	71,534	65,133	31	47,957
3	user_taggedbookmarks	108,035	107,253	31	437,593
4	user_taggedartists	2,100	18,745	60	186,479
5	ratings-m1	6,040	3,952	31	1,000,209
6	ratings-m20	71,567	130,642	31	1,048,575
7	user_shopping-x10	100,208	10,147	10,000	9,999,998
8	user_shopping-x22	100,208	10,392	2,000	19,999,898
9	user_shopping-x40	100,208	10,504	1,000	29,999,529

We experimented with nine tensors that we formed from real world data, and the tensors properties are shown in Table 3. All of the tested tensors have three dimensions, that are extracted from the datasets, which were published by the GroupLeans research group. The datasets (user_ratedmovies.dat and user_taggedmovies.dat) have been maintained by users with both rating and tagging information, which link the movies from the MovieLens dataset with their corresponding web pages at the Internet Movie Database (IMDb) and Rotten Tomatoes movie review systems [Cantador et al. 2011]. This dataset (user_taggedbookmarks.dat) contains social networking, bookmarking, and tagging information from a set of 2K users from the Delicious social bookmarking system [Cantador et al. 2011]. This dataset (user_taggedartists.dat) contains social networking, tagging, and music artist listening information from a set of 2K users from the Last.fm online music system [Cantador et al. 2011]. This dataset (ratings-m20.dat) describes 5-star ratings and free-text tagging activity from MovieLens, a movie recommendation service [Harper and Konstan 2016]. It contains 10,000,054 ratings and 95,580 tags applied to 10,681 movies by 71,567 users. The dataset (ratings-m1.dat) contains 1,000,209 anonymous ratings of approximately 3,900 movies

Table 4. Storage Spaces of the Test Tensors for HOCOO, HOCSR, CFB, and HOCFS Formats Based on Mode-1

No.	HOCOO	$Slice_{num}$	$Fiber_{num}$	HOCSR	CFB	HOCFS
1	3,422,392	10,109	165,687	5,619,176	2,042,570	2,062,788
2	191,828	5,908	25,086	2,115,037	146,086	157,902
3	1,750,372	69,223	100,484	4,200,029	1,076,154	1,214,600
4	745,916	12,523	13,788	1,497,658	400,534	425,580
5	4,000,836	3,706	89,427	2,122,930	2,179,272	2,186,684
6	4,194,300	14,026	174,067	6,147,052	2,445,284	2,473,336
7	39,999,992	9,999	9,516,446	121,469,996	39,032,888	39,052,886
8	79,999,592	9,999	12,642,118	60,783,796	65,284,032	65,304,030
9	119,998,116	9,999	9,502,956	70,503,058	79,004,970	79,024,968

Table 5. Storage Spaces of the Test Tensors for HOCOO, HOCSR, CFB, and HOCFS formats Based on Mode-2

No.	HOCOO	$Slice_{num}$	$Fiber_{num}$	HOCSR	CFB	HOCFS
1	3,422,392	60	27,567	6,003,236	1,766,330	1,766,450
2	191,828	31	4,688	2,313,468	105,290	105,352
3	1,750,372	31	34,551	4,224,271	944,288	944,350
4	745,916	60	2,223	498,958	377,404	377,524
5	4,000,836	31	17,723	2,187,658	2,035,864	2,035,926
6	4,194,300	31	26,464	4,315,727	2,150,078	2,150,140
7	39,999,992	9,999	9,950,117	1,022,079,996	39,900,230	39,920,228
8	79,999,592	1,999	19,033,609	240,415,796	78,067,014	78,071,012
9	119,998,116	999	25,917,967	160,207,058	111,834,992	111,836,990

made by 6,040 MovieLens users who joined MovieLens in 2000[Harper and Konstan 2016]. These datasets (user_shopping-x10, user_shopping-x22, and user_shopping-x40) describe user shopping information of e-commerce websites. They include ordering customers, ordering products, and categorization information.

7.2 Storage Spaces Analysis

The storage spaces of the test tensors are analyzed for HOCOO, HOCSR, CFB, and HOCFS formats, which are illustrated in Tables 4–6. The HOCOO, HOCSR, CFB, and HOCFS in Tables 4–6 represent the number of the storage units of tensor using HOCOO, HOCSR, CFB, and HOCFS formats, and $Slice_{num}$ and $Fiber_{num}$ represent the number of non-empty slices and fibers, respectively. The storage units represent the number of all data that need to be stored for a tensor, including the values of non-zero elements, the indices of the elements, the pointers of the slices and the fibers, and so on. The storage units of HOCOO is the same for different modes of the tensor, because only the non-zero elements are stored for HOCOO format. The storage units of HOCSR varies greatly for different modes of the tensor, and generally larger than that of the other formats, because many empty slices and fibers are stored for HOCSR format. Compared with other formats, CFB format takes up the least storage units. The storage units of HOCFS is very close to that of CFB, and less than that of the other formats. The test tensor is not split into slices and fibers for HOCOO format, and directly is divided non-zero elements into different computing tasks for parallel computation. Parallel efficiency and scale are difficult to improve because the granularity of parallelism is too

Table 6. Storage Spaces of the Test Tensors for HOCOO, HOCSR, CFB, and HOCFS formats Based on Mode-3

No.	HOCOO	$Slice_{num}$	$Fiber_{num}$	HOCSR	CFB	HOCFS
1	3,422,392	2,113	855,598	4,660,935,218	3,422,392	3,426,618
2	191,828	2,113	27,712	4,659,319,936	151,338	155,564
3	1,750,372	1,867	104,799	11,587,953,041	1,084,784	1,088,518
4	745,916	1,892	71,064	39,737,458	515,086	518,870
5	4,000,836	6,039	1,000,209	25,870,498	4,000,836	4,012,914
6	4,194,300	7,119	1,048,575	9,351,753,164	4,194,300	4,208,538
7	39,999,992	99,999	9,950,296	1,036,810,572	39,900,588	40,100,586
8	79,999,592	99,999	19,801,191	1,081,361,332	79,602,178	79,802,176
9	119,998,116	99,999	29,555,291	1,112,583,890	119,109,640	119,309,638

Table 7. Computation Time of the SpTV on the CPU (unit: milliseconds)

No.	HOCOO	HOCSR	CFB	HOCFS_CSR	HOCFS_ELL	TTB	ParTI	splatt
1	8.69	8.40	9.13	7.48	7.93	333.00	13.81	7.82
2	6.57	2.80	5.69	1.12	1.16	13.00	1.21	1.10
3	10.78	3.42	6.37	3.39	3.29	163.00	10.19	3.38
4	3.75	1.88	4.38	1.32	1.02	69.00	4.31	1.08
5	4.58	2.96	6.16	1.64	1.99	378.00	15.62	3.20
6	14.74	8.48	9.61	8.51	8.42	435.00	17.18	7.69
7	97.19	94.32	97.56	31.27	30.75	4,260.00	86.23	32.16
8	198.21	187.63	186.75	67.09	60.68	8,571.00	173.69	68.64
9	304.16	298.72	294.57	85.57	82.28	12,978.00	288.12	99.98

small for HOCOO format. CFB and HOCFS formats have better robustness against different modes of tensor. CFB has a good compression effect, but the tensor stored by CFB format cannot be parallel computed based on slices, resulting in parallel efficiency of CFB less than that of HOCFS. Although the tensor stored by HOCSR can be parallel computed based on slices and fibers, the computation efficiency is reduced due to too many empty slices and fibers.

7.3 SpTV Test and Performance Evaluation

We have performed the following two experiments for the comparative performance evaluation.

(1) The SpTV is tested using Algorithms 6 and is compared to that using Algorithms 3, 4, and 5 on the CPU. Furthermore, Algorithms 6 is compared to Tensor toolbox [Bader et al. 2015], ParTI [Jiajia Li 2017] lib, and SPLATT [Smith et al. 2015] lib on the CPU.

(2) The SpTV is tested using the pipeline mode and the synchronization mode using Algorithms 6 on the GPU, and the performance is compared to that of Algorithms 3, 4, and 5 on the GPU. Furthermore, Algorithms 6 is compared to ParTI [Jiajia Li 2017] lib on the GPU.

For the 9 test cases, the computation time of the SpTV are shown in Table 7, where the HOCOO, HOCSR, CFB, TTB, ParTI, splatt represent the computation time of the SpTV using Algorithms 3, 4, 5, Tensor toolbox lib, ParTI lib, and SPLATT lib on the CPU respectively, and the HOCFS_CSR and HOCFS_ELL represent the computation time of the SpTV using Algorithm 6 based on the CSR and ELL, respectively. It is observed from Table 7 that the performances of HOCFS_CSR and HOCFS_ELL are similar and that the average performance improvement as a percentage for HOCFS_CSR compared with HOCSR is 38.75% on the CPU.

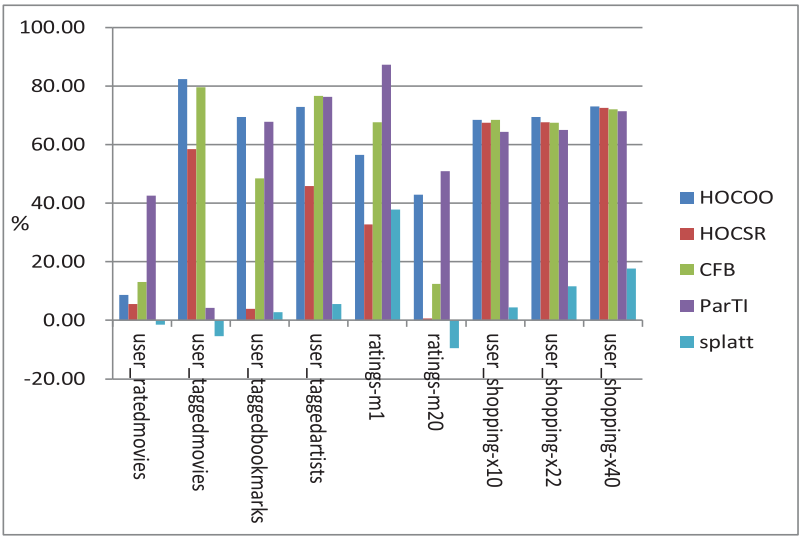


Fig. 14. Performance improvements of the SpTV using Algorithm 6 over Algorithms 3, 4, 5, ParTI, and splatt on the CPU (unit: %).

For the nine test cases, the performance improvements as a percentage of computation time using the HOCFS based on the ELL over the HOCOO, HOCSR, CFB, ParTI, and SPLATT on the CPU are shown in Figure 14. The performance improvement as a percentage is calculated by $(t_1 - t_2)/t_1 \times 100$, where t_2 is the computing time of the SpTV using the HOCFS based on the ELL and t_1 is that of the HOCOO, HOCSR, CFB, ParTI, or SPLATT. For all test cases, it is observed from Figure 14 that the average performance improvements as percentages are 60.39%, 39.42%, 56.22%, 58.88%, and 7.06% using the HOCFS compared with the HOCOO, HOCSR, and CFB, ParTI, and SPLATT, respectively. The performance of HOCFS has obvious advantages compared with Tensor toolbox in matlab, because Tensor toolbox uses serial computing mode.

The process of the SpTV includes three steps, which are the transmission, pretreatment, and computation. A sparse tensor $\mathcal{X}$ and a vector x must be loaded into the global memory of the GPU by the PCIe bus before the SpTV is executed on the GPU, and then a result matrix $\mathbf{A}$ is returned from the GPU after the SpTV has been executed.

For the nine test cases, the computation time of SpTV are shown in Table 8, where HOCOO, HOCSR, CFB, and ParTI represent the computation time of the SpTV using Algorithms 3, 4, 5, and ParTI lib on the GPU, and HOCFS_CSR and HOCFS_ELL represent the computation time of the SpTV on the GPU using Algorithm 6 based on CSR and ELL, respectively. CFB-p represents the computation time of the CFB format using the pipeline mode and HOCFS-p represents the computation time of the HOCFS based on the ELL using the pipeline mode on the CPU-GPU. It is observed from Figure 16 that the average performance improvement as a percentage of HOCFS_CSR compared with HOCSR is 50.82% on the GPU. The HOCFS format based on ELL using the pipeline mode is used to test because the performance of HOCFS_ELL is better than that of HOCFS_CSR. It is observed from Figure 16 that the average performance improvement as a percentage of HOCFS_ELL compared with HOCFS_CSR is 10.22% on the GPU. It can align the access to reduce data read and write times and improve the whole computing performance on the GPU since the length of fibers in the same slice is the same for the HOCFS format using ELL [Li et al. 2015].

Table 8. Computation Time of the SpTV Using Algorithms 3, 4, 5, 6, and ParTI on the GPU (unit: milliseconds)

No.	HOCOO	HOCSR	CFB	CFB-p	HOCFS_CSR	HOCFS_ELL	HOCFS-p	ParTI
1	9.31	7.69	8.80	8.13	3.07	2.37	2.30	5.27
2	4.82	2.57	3.67	2.79	1.43	1.42	1.36	1.39
3	5.42	2.78	3.27	2.67	2.44	1.97	1.79	2.63
4	2.42	1.75	2.47	2.18	0.98	0.89	0.88	1.11
5	2.56	2.58	3.71	3.36	1.88	1.67	1.59	5.10
6	7.74	7.74	8.03	7.41	3.46	2.86	2.53	7.16
7	92.34	87.05	92.37	86.18	25.15	23.43	21.36	61.90
8	176.93	168.51	165.87	151.04	51.29	49.81	44.23	128.24
9	287.59	280.95	281.23	254.56	74.49	72.43	63.74	187.32

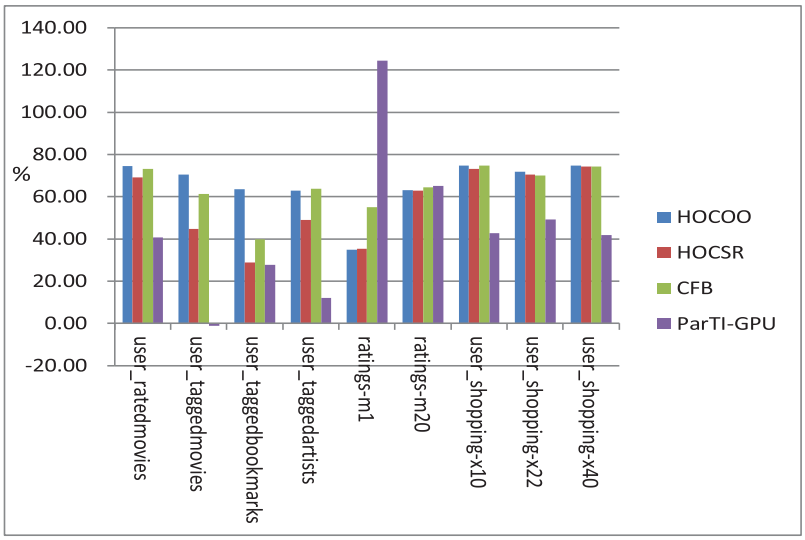


Fig. 15. Performance improvements of the SpTV using Algorithm 6 over Algorithms 3, 4, 5, and ParTI on the GPU (unit: %).

Due to the irregularity of slices of the sparse tensor, the utilization of memory bandwidth is low for SpTV. Maintaining the continuity of data addresses accessed by multiple threads can improve the bandwidth utilization of memory for parallel algorithms. The tensor is divided into slices by HOCFS format and stored in CSR or ELL compression format. For CSR format, each thread usually calculates a row of data independently. Because the data length of different rows may not be the same, data accessed simultaneously by multiple threads may not be in a continuous address space. The data length of each row of slices stored in ELL format is the same, and the data accessed by multiple threads will basically be in a relatively continuous address space. Therefore, the bandwidth utilization of ELL format is generally better than that of CSR format. It is observed from the experimental results that the performance of SpTV based on HOCFS_ELL format is better than that of HOCFS_CSR format.

For the nine test cases, the performance improvements as a percentage of computation time using HOCFS over HOCOO, HOCSR, and CFB on the GPU are shown in Figure 15. The performance improvement as a percentage is calculated by $(t_1 - t_2)/t_1 \times 100$, where t_2 is the computation time

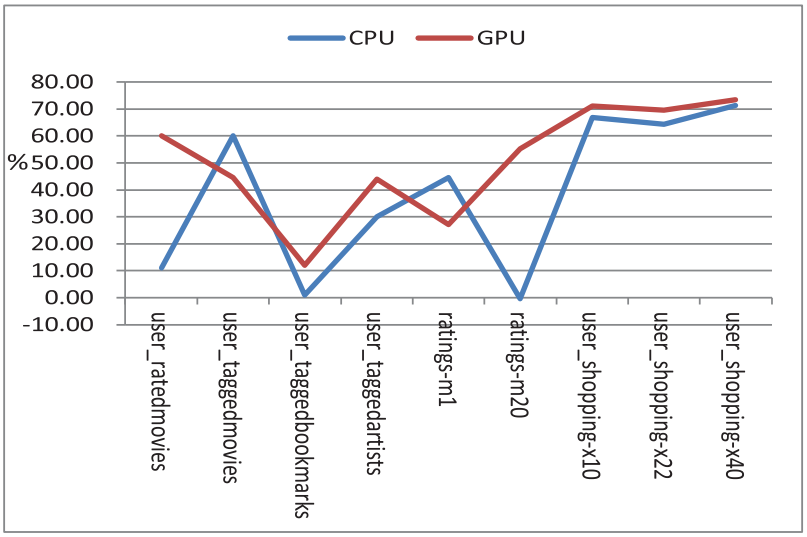


Fig. 16. Performance improvements of the SpTV using the HOCFS base on the CSR over the HOCSR on the CPU and GPU.

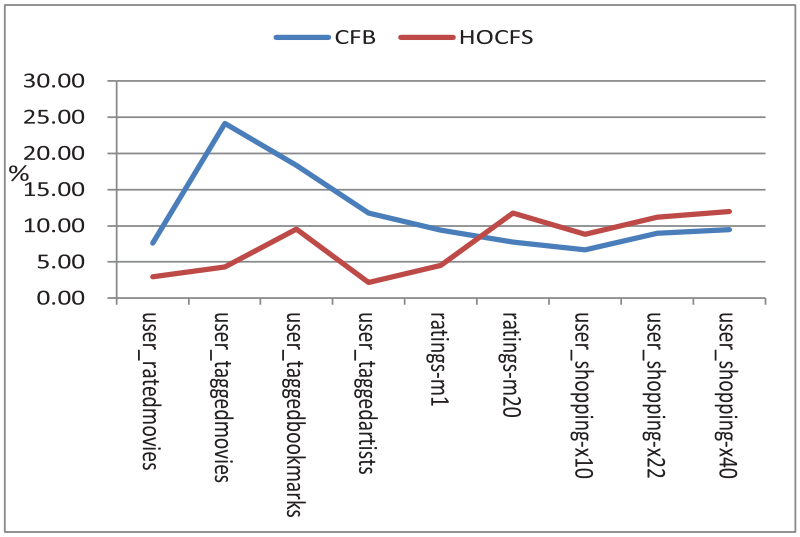


Fig. 17. Performance improvements of the CFB and HOCFS format using the pipeline mode.

of the SpTV using the HOCFS and t_1 is that of the HOCOO, HOCSR, CFB, or ParTI. For all test cases, it is observed from Figure 15 that the average performance improvements as a percentage are 65.65%, 56.43%, 64.01%, and 44.72% using the HOCFS compared with the HOCOO, HOCSR, CFB, and ParTI, respectively.

The nine tensors are tested using the CFB and HOCFS format on the pipelined mode and the synchronization mode, respectively. The tasks between different streams are executed by out-of-order in the GPU. The computed results must be synchronized, thus leading to an overall performance deterioration for a tensor if the slices or buckets partitioned from the tensor are assigned into

different streams to be processed. If a stream is used only in the pipeline, the efficiency of the pipeline is not high since the sizes of the slices or buckets may have big deviation, thus leading to performance degradation according to the equation for T_p. However, the polling is costly for the streams if there are many streams in CUDA. We find that the test using 3–5 streams has better performance for the tested cases. We find that the transmission is the bottleneck for the SpTV on the GPU. Therefore, the transmission time can be hidden to improve the performance when a tensor is partitioned and processed by pipeline.

The performance improvements of the pipelined mode over the synchronization mode on the CPU-GPU are shown in Figure 17, where CFB and HOCFS represent the performance improvements of the CFB and HOCFS formats, respectively. The performance improvements are calculated by $(t_1 - t_2)/t_1 \times 100$, where t_2 is the computation time of the SpTV using the pipelined mode and t_1 is that of the synchronization mode. For all test cases, it is observed from Figure 17 that the average performance improvements in percentages are 11.56% and 7.48% using the pipelined mode compared with the synchronization mode for the CFB and HOCFS formats, respectively.

8 CONCLUSION

In this article, we analyze and design a compressed storage strategy for large-scale sparse tensors and propose a layered compression format (HOCFS) based on slices, which can greatly improve the compression ratio of sparse tensors. A pipeline parallel algorithm for the SpTV based on the compression format (HOCFS) is designed to greatly improve its computing performance on the GPU. In future work, we will address tensor decomposition based on the current SpTV method.

REFERENCES

[n.d.]. ATen: A TENsor library for C++11. Retrieved from https://github.com/zdevito/ATen.

2018. FTensor. Retrieved from https://bitbucket.org/wlandry/ftensor.

C. J. Appellof and E. R. Davidson. 1983. Three-dimensional rank annihilation for multi-component determinations. *Analytica Chimica Acta* 146, FEB (1983), 9–14.

Brett W. Bader, Tamara G. Kolda, et al. 2015. MATLAB Tensor Toolbox Version 2.6. Retrieved from http://www.sandia.gov/tgkolda/TensorToolbox/.

David E. Booth. 2004. Multi-way analysis: Applications in the chemical sciences. *Technometrics* 47, 4 (2004), 518–519.

Guillaume Bouchard, Jason Naradowsky, Sebastian Riedel, Tim Rocktäschel, and Andreas Vlachos. 2015. Matrix and tensor factorization methods for natural language processing. In *Tutorials*. 16–18.

Rasmus Bro. 1997. PARAFAC. Tutorial and applications. *Chemometrics and Intelligent Laboratory Systems* 38, 2 (1997), 149–171.

Rasmus Bro. 1998. Multi-way analysis in the food industry. Models, algorithms, and applications. *Ethical Theory and Moral Practice* 6, 2 (1998), 231–235.

Iván Cantador, Peter Brusilovsky, and Tsvi Kuflik. 2011. Second workshop on information heterogeneity and fusion in recommender systems (HetRec 2011). In *5th ACM Conference on Recommender Systems (RecSys'11)*. ACM, New York, NY.

J. Douglas Carroll and Jih Jie Chang. 1970. Analysis of individual differences in multidimensional scaling via an n-way generalization of Eckart-Young decomposition. *Psychometrika* 35, 3 (1970), 283–319.

Raymond B. Cattell. 1944. Parallel proportional profiles and other principles for determining the choice of factors by rotation. *Psychometrika* 9, 4 (1944), 267–283.

Joon Hee Choi and S. V. N. Vishwanathan. 2014. DFacTo: Distributed factorization of tensors. In *Advances in Neural Information Processing Systems*. 1296–1304.

F. Maxwell Harper and Joseph A. Konstan. 2016. The MovieLens datasets. *ACM Transactions on Interactive Intelligent Systems* 5, 4 (2016), 1–19.

R. A. Harshman. 1970. Foundations of the PARAFAC procedure: Model and conditions for an 'exploratory' multimodal factor analysis. In *UCLA Working Papers in Phonetics*.

So Hirata. 2003. Tensor contraction engine: Abstraction and automated parallel implementation of configuration-interaction, coupled-cluster, and many-body perturbation theories. *Journal of Physical Chemistry A* 107, 46 (2003), 9887–9897.

Frank L. Hitchcock. 1927. The expression of a tensor or a polyadic as a sum of products. *Studies in Applied Mathematics* 6, 1–4 (1927), 164–189.

Richard Vuduc Jiajia Li, Yuchen Ma. 2017. ParTI! : A Parallel Tensor Infrastructure. Retrieved from https://github.com/hpcgarage/ParTI.

U. Kang, Evangelos Papalexakis, Abhay Harpale, and Christos Faloutsos. 2012. GigaTensor: Scaling tensor analysis up by 100 times - algorithms and discoveries. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 316–324.

C. T. King, W. H. Chou, and L. M. Ni. 1990. Pipelined data parallel algorithms-I: Concept and modeling. *IEEE Transactions on Parallel and Distributed Systems* 1, 4 (1990), 470–485.

Donald E. Knuth. 1973. *The Art of Computer Programming: Seminumerical Algorithms*. Pearson Schweiz AG.

Tamara G. Kolda and Brett W. Bader. 2009. Tensor decompositions and applications. *SIAM Review* 51, 3 (2009), 455–500.

L. De Lathauwer and B. De Moor. 1998. From matrix to tensor: Multilinear algebra and signal processing. In *4th IMA International Conference on Mathematics in Signal Processing*. 1–15.

Jiajia Li, Yuchen Ma, Chenggang Yan, and Richard Vuduc. 2017. Optimizing sparse tensor times matrix on multi-core and many-core architectures. In *Irregular Applications: Architecture and Algorithms*. 26–33.

Kenli Li, Wangdong Yang, and Keqin Li. 2015. Performance analysis and optimization for SpMV on GPU using probabilistic modeling. *IEEE Transactions on Parallel and Distributed Systems* 26, 1 (2015), 196–205.

Bangtian Liu, Chengyao Wen, Anand D. Sarwate, and Maryam Mehri Dehnavi. 2017. A unified optimization approach for sparse tensor operations on GPUs. In *IEEE International Conference on Cluster Computing (CLUSTER'17)*. 47–57.

Fishman Matthew, E. Miles Stoudenmire, and Steven R. White. 2017. ITensor. Retrieved from http://itensor.org/index.html.

Makoto Nakatsuji, Qingpeng Zhang, Xiaohui Lu, Bassem Makni, and James A. Hendler. 2017. Semantic social network analysis by cross-domain tensor factorization. *IEEE Transactions on Computational Social Systems* 4, 4 (2017), 207–217.

Q10 NVIDIA. 2013. *NVIDIA CUDA C Programming Guide*. Technical Report. NVIDIA.

Q11 Ivan Osorio and Naresh C. Bhavaraju. 2013. Stimulation methodologies and apparatus for control of brain states.

Evangelos E. Papalexakis, Christos Faloutsos, and Nicholas D. Sidiropoulos. 2012. ParCube: Sparse parallelizable tensor decompositions. *ACM Transactions on Knowledge Discovery from Data* 10, 1 (2012), 521–536.

Roman Poya, Antonio J. Gil, and Rogelio Ortigosa. 2017. A high performance data parallel tensor contraction framework: Application to coupled electro-mechanics. *Computer Physics Communications* (2017). DOI : https://doi.org/10.1016/j.cpc.2017.02.016 Q12

Liqun Qi, Wenyu Sun, and Yiju Wang. 2007. Numerical multilinear algebra and its applications. *Frontiers of Mathematics in China* 2, 4 (2007), 501–526.

Steffen Rendle, Leandro Balby Marinho, Alexandros Nanopoulos, and Lars Schmidt-Thieme. 2009. Learning optimal ranking with tensor factorization for tag recommendation. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 727–736.

Nicholas D. Sidiropoulos, Lieven De Lathauwer, Xiao Fu, Kejun Huang, Evangelos E. Papalexakis, and Christos Faloutsos. 2017. Tensor decomposition for signal processing and machine learning. *IEEE Transactions on Signal Processing* 65, 13 (2017), 3551–3582.

N. D. Sidiropoulos, E. E. Papalexakis, and C. Faloutsos. 2014b. A parallel algorithm for big tensor decomposition using randomly compressed cubes (PARACOMP). In *IEEE International Conference on Acoustics, Speech and Signal Processing*. 1–5.

N. D. Sidiropoulos, E. E. Papalexakis, and C. Faloutsos. 2014a. Parallel randomly compressed cubes: A scalable distributed architecture for big tensor decomposition. *IEEE Signal Processing Magazine* 31, 5 (2014), 57–70.

Shaden Smith, Niranjay Ravindran, Nicholas D. Sidiropoulos, and George Karypis. 2015. SPLATT: Efficient and parallel sparse tensor-matrix multiplication. In *IEEE International Parallel and Distributed Processing Symposium*. 61–70.

Edgar Solomonik, Devin Matthews, Jeff Hammond, and James Demmel. 2013. Cyclops tensor framework: Reducing communication and eliminating load imbalance in massively parallel contractions. In *IEEE International Symposium on Parallel and Distributed Processing*. 813–824.

P. A. Tew. 2016. An Investigation of Sparse Tensor Formats for Tensor Libraries. Retrieved from http://groups.csail.mit.edu/commit/papers/2016/parker-thesis.pdf.

Ledyard R. Tucker. 1966. Some mathematical notes on three-mode factor analysis. *Psychometrika* 31, 3 (1966), 279–311.

Sorber L. Van Barel M. Vervliet N., Debals O. and De Lathauwer L. 2016. MATLAB Tensorlab Version 3.0. Retrieved from https://www.tensorlab.net/.

Yichen Wang, Robert Chen, Joydeep Ghosh, Joshua C. Denny, Abel Kho, You Chen, Bradley A. Malin, and Jimeng Sun. 2015. Rubik: Knowledge guided tensor factorization and completion for health data analytics. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 1265–1274.

Wangdong Yang, Kenli Li, and Keqin Li. 2018. A parallel computing method using blocked format with optimal partitioning for SpMV on GPU. *Journal of Computer and System Sciences* 92, 1 (2018), 152–170.
Wangdong Yang, Kenli Li, Zeyao Mo, and Keqin Li. 2015. Performance optimization using partitioned SpMV on GPUs and multicore CPUs. *IEEE Transactions on Computers* 64, 9 (2015), 2623–2636.

Received May 2018; revised July 2019; accepted September 2019

Author Queries

Q1: AU: Author affiliation line should include all affiliations the author had while completing this article, and the author address line should only include the authors' current addresses. Please review for correctness.

Q2: AU: Please include complete current mailing addresses for all authors.

Q3: AU: Please verify the expansion of "SpMV" as "sparse matrix and vector multiplications" for correctness.

Q4: AU: Reference citation "Vervliet N. and L (2016)" has been changed to "Sober et al. (2016)" as per the reference list. Please validate.

Q5: AU: Please check whether the edit made in sentence "The second slice is . . . elements" retain your intended meaning.

Q6: AU: Please provide the author group and year of publication in reference "[n.d.]. ATen: A TENsor library for C++11."

Q7: AU: Please provide the author group in reference "2018. FTensor."

Q8: AU: Please list the name of all authors in place of et al. in reference "Bader et al. (2015)."

Q9: AU: Please check whether reference "Harshman (1970)" is okay as set.

Q10: AU: Please check whether reference "NVIDIA (2013)" is okay as set.

Q11: AU: Please provide the complete bibliographic details in reference "Osorio and Bhavaraju (2013)."

Q12: AU: Please provide the volume number, issue number and page range in reference "Poya et al. (2017)."