






A Novel Parallel Algorithm for Sparse Tensor Matrix Chain Multiplication via TCU-Acceleration

Haotian Wang , Wangdong Yang , Rong Hu , Renqiu Ouyang , Kenli Li , *Senior Member, IEEE*,
and Keqin Li , *Fellow, IEEE*

Abstract—Analysis of multi-dimensional data, especially tensor decomposition, which extracts latent information, is becoming considerably popular. Although multi-dimensional sparse data is typically processed on multi-core processors, developing highly optimized GPU-based Sparse Tensor Matrix Chain Multiplication (SpTMCM) is challenging. The purpose of this paper is to investigate a novel approach named SpTMCM and to explore the discovery of SpTMCM coupled with the emerging computing core, Tensor Core Unit (TCU). In contrast to prior work, the proposed novel approach enables a uniform storage format and optimization approach for SpTMCM. We design a hybrid tensor format based on multi-dimensional tiling that divides the tensor depending on the tile threshold to address the inefficient memory accesses caused by the irregular nonzero distribution of the sparse tensor. Further, we develop a TCU-based tensor parallel algorithm with our novel approach to increase the memory bandwidth. Compared to state-of-the-art works, our method achieves $1.16 \sim 24.12\times$ speedup for SpMTTKRP and $5.07 \sim 7.15\times$ speedup for SpTTMChain across NVIDIA A100 GPU on a range of real-world sparse tensors.

Index Terms—GPU, hybrid format, parallel performance, SpMTTKRP, SpTTMChain, sparse tensor, tensor core.

I. INTRODUCTION

TENSOR is commonly used to represent multi-dimensional data and is typically defined as multi-dimensional arrays or N -directional arrays. Popular applications of tensor decomposition include data mining [1], recommendation systems [2], anomaly detection [3], and so on [4]. Sparse Tensor Matrix Chain Multiplication (SpTMCM) is a fundamental kernel of

tensor algorithms and applications, such as CP decomposition [5] and Tucker decomposition [6]. Most tensors in practical applications are sparse, meaning that most of their elements are zero. Consequently, it is essential to design and develop parallel algorithms to optimize SpTMCM on contemporary architectures.

In the past, numerous studies have been conducted on SpTMCM, and the optimization of SpTMCM on various hardware platforms has been an important research topic. Shared-memory and distributed-memory systems [7], [8] are limited by a cumbersome parallel execution model for SpTMCM [9], so optimizing SpTMCM on GPU has become a prudent move. Moreover, GPUs are undergoing a rapid evolution and Tensor Core Units (TCUs) available on the latest GPUs offer new opportunities for optimizing SpTMCM. TCUs provide high performance and low power consumption, but its sole function is Matrix Multiplication (MM) [10], [11]. GPU is inherently scalable from domain-specific to general-purpose computing, making the study of extending MM-specific TCUs to SpTMCM interesting and significant.

Previous studies of optimization SpTMCM such as Sparse Matricized Tensor Times Khatri-Rao Product (SpMTTKRP) and Sparse Tensor Times Matrix Chain (SpTTMChain) are independent and typically require a series of tedious pre-settings to optimize the performance of individual SpTMCM, and these complex processes often cause a lot of troubles for users [12]. Fortunately, there are similarities in data patterns and types of optimization among these various SpTMCM, and there have been multiple attempts by researchers to explore a novel approach for various SpTMCM [9], [13], [14], [15], [16], [17]. The focus of these studies has been mainly on the generality of different multiplications at nonzero granularity, which is detrimental to the design of parallel algorithms based on TCU that focus on MM. In light of this, we propose a novel MM-based approach for representing SpMTTKRP and SpTTMChain by studying the computational model of SpTMCM.

SpTMCM is inherently memory-constrained kernel due to the irregular nonzero distribution of the sparse tensor, which leads to frequent memory accesses [18], [19]. GPU with High-Bandwidth Memory (HBM) can speed up SpTMCM in memory-constrained kernel, making this a viable option. Despite the fact that the high-bandwidth memory of GPU can surpass the memory limit of SpTMCM, there are still limitations such as limited memory capacity and high memory access latency. Designing a tensor format suitable for GPU is a feasible way to solve above

Manuscript received 12 October 2022; revised 23 February 2023; accepted 19 June 2023. Date of publication 22 June 2023; date of current version 5 July 2023. This work was supported in part by the Key-Area Research and Development Program of Guangdong Province under Grant 2021B0101190004, in part by the National Key R&D Program of China under Grant 2021YFB0300800, in part by the Key Program of the National Natural Science Foundation of China under Grants U21A20461 and 92055213, in part by the Research Innovation Project for Postgraduate Students of Hunan Province under Grant CX20220412, and in part by the National Natural Science Foundation of China under Grant 61872127. Recommended for acceptance by D. Tiwari. (*Corresponding author: Wangdong Yang.*)

Haotian Wang, Wangdong Yang, Rong Hu, Renqiu Ouyang, and Kenli Li are with the College of Computer Science and Electronic Engineering, Hunan University, Changsha, Hunan 410082, China, and also with the National Supercomputing Center in Changsha, Changsha, Hunan 410082, China (e-mail: wanghaotian@hnu.edu.cn; yangwangdong@hnu.edu.cn; upupwords@hnu.edu.cn; rqouyang@hnu.edu.cn; lkl@hnu.edu.cn).

Keqin Li is with the College of Computer Science and Electronic Engineering, Hunan University, Changsha 410082, China, and also with the Department of Computer Science, State University of New York, New Paltz, NY 12561 USA (e-mail: lik@newpaltz.edu).

Digital Object Identifier 10.1109/TPDS.2023.3288520

problems. In previous studies, the compressed tensor format can reduce the memory footprint and group dependency but is limited by the mode-specificity [20], [21]. The tensor tiling format, which stores tensors in tiles and exploits the localization of data to reduce memory access overhead, has little utility for tensors with irregular nonzero distribution [22], [23], [24]. Consequently, we construct a hybrid tensor format based on tensor tiling, in which the nonzeros of a tensor with good data locality are aggregated into a tile and saved, while the discrete distribution of the tensor is still recorded as a single nonzero. This makes effective use of the localization of tensor data and reduces the additional burden of tile aggregation.

TCU, the latest computational component on GPU, is bound to be a big trend to use to accelerate general-purpose computations [25], [26], [27]. TCU is designed to accommodate only small dense matrices, hence conventional tensor storage formats are incompatible with such hardware gas pedals. The process of tensor tiling divides the tensor into smaller pieces, which corresponds to the design of TCU. Based on this, we investigate the acceleration of TCU for SpTMCM, using the fast MM of TCU to accelerate the computation of tiles and maximize the utilization of Multiplicative Accumulation (MAC) operation on TCU.

In general, to address the above issues, in this paper, we

- propose a novel approach that coarsens the multiplication pattern of SpTMCM from nonzero granularity to slice granularity, which facilitates the design of the TCU-based parallel algorithm.
- construct a hybrid tensor format based on multi-dimensional tiling to provide higher memory efficiency for the TCU-based parallel algorithm.
- develop a TCU-based SpTMCM parallel algorithm to maximize the performance benefits of new-generation GPU through efficient memory access patterns and full utilization of the MAC manipulation of TCU.
- test the performance of our proposed method on two recent GPUs, and the experimental results show that our method has a significant performance advantage over the baseline method.

The rest of the paper is organized as follows: Section II outlines the basics of tensor, Section III summarizes recent related research on SpTMCM, Section IV describes the MM-based form of SpTMCM, Section V introduces the hybrid tensor format, Section VI demonstrates a detailed implementation of TCU-based SpTMCM, Section VII reveals our experimental results, and Section VIII summarizes the works and contributions of this paper.

II. BACKGROUND

A. Tensor Notation

We follow the definitions and symbols used by Kolda and Sun in [4]. A tensor is a multi-dimensional array. Each of its dimensions is called a mode, and the *order* of a tensor refers to the number of dimensions or modes. A *N*-th-order tensor has *N* dimensions or modes. For example, a vector is a first-order tensor, which is denoted by lowercase letter, e.g., v , and it is also a row or column of matrix, by $U(i, :)$ or $U(:, j)$, and

TABLE I
TABLE OF SYMBOLS

Symbol	Definition
v, a, b, c	A vector or first-order tensor.
U, Y, A, B, C	A matrix or second-order tensor.
\mathcal{X}, \mathcal{Y}	A third-order tensor.
I, J, K	The mode size of third-order tensor \mathcal{X} .
$U(i, :)$	The row vector of matrix.
$U(:, j)$	The column vector of matrix.
$\mathcal{X}(i, j, :)$	The mode-1 fiber of the third-order tensor \mathcal{X} .
$\mathcal{X}(i, :, :)$	The frontal slice of the third-order tensor \mathcal{X} .
$X_{(1)}$	The mode-1 unfolding of the third-order tensor \mathcal{X} .
S_I, S_J, S_K	The size of tiles in each dimension.
T	The threshold of sparse tile.
M_S^i	The bit space size of linear indices of HLS-S format.
M_D^i	The bit space size of linear indices of HLS-D format.
M_S^v	The bit space size of values of HLS-S format.
M_D^v	The bit space size of values of HLS-D format.
M_{bm}	The bit space size of bitmap.
M_H	The bit space size of HLS format.

it can be represented as a fiber which is defined by fixing all tensor indices but one, by $\mathcal{X}(i, j, :)$. A matrix, a second-order tensor, is denoted by capital letter, e.g., U , and it can also be regarded as a slice which by fixing every tensor indices but two, by $\mathcal{X}(i, :, :)$. A tensor of *order* 3 or higher is called *high-order* tensor, is denoted by bold capital calligraphic letter, e.g., \mathcal{X} . An element at position (i, j, k) of tensor \mathcal{X} is denoted by $\mathcal{X}(i, j, k)$. An element at position (i, j) of matrix U is denoted by $U(i, j)$. An element at position i of vector v is denoted by v_i . For the rest of paper, unless otherwise noted, a third-order tensor \mathcal{X} is of dimensions $I \times J \times K$. The symbols commonly used in our paper are summarized in Table I.

Mode-*n* unfolding, also known as matricization, is a common operation of SpTMCM. It is the process of transforming a tensor into a matrix along mode-*n* and reordering nonzeros of the tensor. For instance, a third-order tensor $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$, can be unfolded into a matrix according mode-1, which is denoted as $X_{(1)} \in \mathbb{R}^{I \times JK}$.

Some multiplication operations between vectors and matrices are also fundamental operations of SpTMCM [16], [21], [28], including Hadamard product, Kronecker product, and Khatri-Rao product [29].

Hadamard product (also known as element-wise product or Schur product) is a binary operation that takes two matrices (vectors) of the same dimension and produces another matrix (vector) of the same dimension as the operands. The Hadamard product of two vectors $a, b \in \mathbb{R}^I$ is a vector $c = a * b, c \in \mathbb{R}^I$, with elements $c_i = a_i \cdot b_i$.

Kronecker product is an operation on two matrices (vectors) of arbitrary size resulting in a matrix (vector). It is a generalization of the outer product (which is denoted by the same symbol) from vectors to matrices. The Kronecker product of two vectors $a \in \mathbb{R}^I$ and $b \in \mathbb{R}^J$ results in a vector $c = a \otimes b, c \in \mathbb{R}^{IJ}$, defined as

$$c = a \otimes b = \begin{bmatrix} a_1 b \\ a_2 b \\ \vdots \\ a_I b \end{bmatrix}. \quad (1)$$

For two matrices $A \in \mathbb{R}^{I \times K}$ and $B \in \mathbb{R}^{J \times K}$, their Khatri-Rao product is $C = A \odot B$ which corresponds to Kronecker product of their corresponding column vectors, defined as

$$C = A \odot B = [a_1 \otimes b_1, a_2 \otimes b_2, \dots, a_K \otimes b_K], \quad (2)$$

where $C \in \mathbb{R}^{IJ \times K}$.

B. TCU Programming Model

TCU programming model plays an important role in designing and implementing rectangle computation on GPU, which has tremendously aided the advancement of machine learning, deep learning, and high performance computing. Compare with CUDA Core, which calculates a floating-point operation per clock cycle, TCU performs one matrix-matrix operation ($D = A \times B + C$) within each clock cycle, where A, B, C and D are matrices of size 16×16 .

For ease of programming and using TCU, NVIDIA provides Warp-level Matrix Multiply and Accumulate (WMMA) API. The process of employing TCU can be divided into 3 stages. First, the WMMA fragments `Afrag`, `Bfrag`, and `Cfrag` are being declared. Second, `Cfrag`, the accumulator fragment used to store the result of tile multiplication, is being set to zero, and then the input tiles (A and B) are being loaded into the fragments `Afrag` and `Bfrag` by executing the command `wmma::load_matrix_sync()`. Third, tile multiplication is performed by calling the command `wmma::mma_sync()`, and then result is transferred from `Cfrag` to D in the GPU global memory.

III. RELATED WORK

Optimizing SpTTMChain and SpMTTKRP operations has been the key area in tensor composition, which proposes various parallel strategies to achieve load balance and data reuse. Our survey of previous work is based on (i) storage optimization for sparse tensors; and (ii) parallel algorithms on SpTMCM.

Storage optimization for sparse tensors [30]: Hybrid formats [21], [31] are made available to handle nonzeros without favorable localization. For a tensor with N modes, SpTMCM usually requires a sequence of N computations which is unacceptable in practice. Nisa et al. [31] proposed a mixed-mode tensor representation, where heavy fibers and slices are stored at their most suitable modes. By storing just a portion of fibers, the memory overhead associated with storing multiple modes can be significantly reduced. Another hybrid scheme is to split the entire tensor into dense and sparse parts according to a given threshold condition. The parallel approach varies based on different densities. Nisa et al. [21] constructed a Hybrid Balanced-CSF format by splitting fibers to achieve inter-warp load balance and partitioning tensor slices into three groups.

Bit-operation, as a common method of data compression, is designed to efficiently process data that is frequently accessed. Li et al. [24] proposed HiCOO format with a sparse-blocked pattern to store a sparse tensor. It divided a third-order tensor into sub-tensors of size $2 \times 2 \times 2$ and represented every sub-tensor with fewer bits. Nguyen et al. [22] designed BLCO format which linearized the indices of one nonzero to an encoding line

represented by a single index. Each nonzero of the indices only needs to be accessed once rather than as many times as modes has the tensor.

Parallel algorithms for SpTMCM [32]: The most challenging aspect of the sparse tensor is its irregular and imbalanced computation pattern. For sparse data, routine parallel strategies cause severe load imbalance, with very large differences in the data processed by each parallel unit. To solve the problem of load imbalance, Li et al. [33] proposed a novel tensor memorization algorithm and focused on a sequence of SpMTTKRP operations to search the way to achieve the balance of computational overheads among modes. Nisa et al. [21] split fibers and slices to address the load imbalance among thread blocks on GPU. Heavy fibers are split into fiber-segments, which enables near-equal workloads for all the warps in the thread block. Liu et al. [9] designed unified parallel algorithms for both SpMTTKRP and SpTMChain which were not sensitive to mode changes. They captured changes in the computation pattern and used two flag arrays to represent any changes in the index modes.

Fiber-, slice- or block-level computation as the granularity are the basic tensor tiling methods. With tiling, index reuse diminishes the reading times of nonzero locations. Fiber-level partition represents the 1D tensor partition scheme. This formulation of SpTMCM is based on the fiber-wise and column-wise partitioning of input tensor and factor matrices, respectively. Hu et al. [34] utilized vector inner-product to formulate SpMTTKRP and designed a fiber-level partitioning scheme. Hayashi et al. [35] designed a two-step SpMTTKRP to use BLAS subroutines. The 2-step algorithm first performed a partial SpMTTKRP for a slice in a tensor followed by a tensor-times-vector operation between a tensor and vectorization of a Khatri-Rao product.

Application of TCUs: The development of TCUs has been fueled by deep learning. How to boost performance across a range of applications has emerged as a research priority because TCUs only provide matrix multiplication on small dense matrices. Huang et al. [27] utilized TCUs to optimize hierarchical Tucker tensor learning primitives and employed the optimized primitives to optimize hierarchical Tucker tensor decomposition algorithms for extensive data analysis. Dakkak et al. [36] expressed both reduction and scan by matrix multiplication operations on TCUs and demonstrated a novel, straightforward, and effective mapping of the reduction and scan primitives on TCUs. TCUs were also commonly used in tensor analysis. Wang et al. [37] proposed a graph neural network (GNN) acceleration framework based on TCUs, which reconciles the sparse GNN computation with TCUs. Hu et al. [38] demonstrated a TCU-accelerated query engine, which expedited several query operations, such as group-by aggregates and natural joins. Chen et al. [39] expressed a column-vector-sparse-encoding and designed SpMM and SDDMM kernels on TCUs based on this.

IV. A NOVEL APPROACH OF SP TMCM

TCU is a novel type of functional unit included in the latest NVIDIA GPU that significantly accelerates matrix multiplication operations relative to standard general-purpose processor cores. It is difficult to use TCU to directly speed SpTMCM

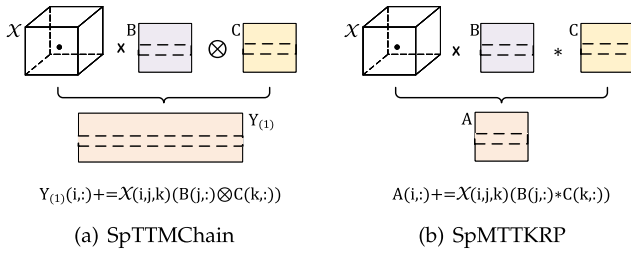


Fig. 1. Graphical representation of the Common Approach.

because of the enormous dimensionality of the tensor. The exploration of a novel approach for SpTMCM such as SpTTMChain and SpMTTKRP is currently an essential research topic due to the complexity of the computational process of SpTMCM [9], [13], [14], [15], [17]. In this section, we detail the novel approach of SpTMCM we have designed to take advantage of TCU.

A. A Common Approach

SpTTMChain and SpMTTKRP are the key operations in tensor decomposition, and these operations can be described using some modal notations. The derivation of the modal notation allows us to propose a unified optimization method for SpTMCM [9], [15]. Taking the third-order tensor as an example, SpTTMChain and SpMTTKRP can be expressed uniformly as an element-wise computational form as follows:

$$Y_{(1)}(i, :) += \mathcal{X}(i, j, k) (B(j, :) \otimes C(k, :)), \quad (3)$$

$$A(i, :) += \mathcal{X}(i, j, k) (B(j, :) * C(k, :)). \quad (4)$$

Equation (3) describes the element-wise computational form of SpTTMChain, where $\mathcal{X}(i, j, k)$ is a nonzero of the tensor \mathcal{X} , $B(j, :)$ and $C(k, :)$ are row vectors of the factor matrix, and $Y_{(1)}(i, :)$ is a row vector of a mode-1 unfolded tensor \mathcal{Y} .

Similar to (3), (4) describes the element-wise computational form of SpMTTKRP. The difference with (3) is that the result of (4) is the row vector of the factor matrix A . From the perspective of the computational process, (4) is Hadamard product between the row vectors of the factor matrices B and C , while (3) is Kronecker product.

The element-wise computational form of (3) and (4) as shown in Fig. 1 uses the nonzeros of the tensor as the traversal objective, which presents an opportunity to design efficient SpTMCM. This has inspired us to design a novel approach of MM-based calculation to better utilize TCU and fully exploit the performance advantages of TCU in rectangular computing.

B. A Novel MM-Based Approach

To clearly demonstrate the approach we have designed for SpTMCM, the formula for each element of the SpTTMChain result is first given

$$\mathcal{Y}(i, r_1, r_2) = \sum_{j=1}^J \left(\sum_{k=1}^K \mathcal{X}(i, j, k) C(k, r_2) \right) B(j, r_1), \quad (5)$$

where $B(j, r_1)$ and $C(k, r_2)$ are scalar values of the corresponding factor matrices, respectively, and $\mathcal{Y}(i, r_1, r_2)$ and $\mathcal{X}(i, j, k)$ are nonzeros of the corresponding tensors, respectively.

Second, the computational form is adjusted to converge to the inner product of vectors

$$\begin{aligned} \mathcal{Y}(i, r_1, r_2) &= \sum_{j=1}^J (\mathcal{X}(i, j, :) C(:, r_2)) B(j, r_1) \\ &= B(:, r_1)^\top (\mathcal{X}(i, :, :) C(:, r_2)), \end{aligned} \quad (6)$$

where $B(:, r_1) \in \mathbb{R}^{J \times 1}$ and $C(:, r_2) \in \mathbb{R}^{K \times 1}$ are columns of factor matrices, $\mathcal{X}(i, j, :)$ is a fiber, and $\mathcal{X}(i, :, :)$ is a slice of the tensor \mathcal{X} .

Third, the computational form is modified to yield the slice-wise computational form of SpTTMChain.

$$\mathcal{Y}(i, :, :) = B^\top (\mathcal{X}(i, :, :) C), \quad (7)$$

where $B \in \mathbb{R}^{J \times R_1}$ and $C \in \mathbb{R}^{K \times R_2}$ are factor matrices, $\mathcal{Y}(i, :, :) \in \mathbb{R}^{R_1 \times R_2}$ is a slice of the tensor \mathcal{Y} .

Similarly, the formula for each element in the result of SpMTTKRP can be derived as follows:

$$A(i, r) = \sum_{j=1}^J \left(\sum_{k=1}^K \mathcal{X}(i, j, k) C(k, r) \right) B(j, r), \quad (8)$$

where $A(i, r)$, $B(j, r)$, and $C(k, r)$ are scalar values of the factor matrix. After adjusting the computation form it can be expressed as

$$\begin{aligned} A(i, r) &= \sum_{j=1}^J (\mathcal{X}(i, j, :) C(:, r)) B(j, r) \\ &= B(:, r)^\top (\mathcal{X}(i, :, :) C(:, r)), \end{aligned} \quad (9)$$

where $B(:, r)$ and $C(:, r)$ are associated, both being the r -th column of the corresponding factor matrix, which is precisely the distinction from (6).

Further, since the result of SpMTTKRP is a factor matrix A whose number of columns is R , just like the columns of factor matrices B and C . The slice-wise computational form of SpMTTKRP is expressed as

$$A(i, :) = \text{diag}(B^\top (\mathcal{X}(i, :, :) C)), \quad (10)$$

where the *diag* function serves to take the diagonal of the matrix, which is a vector $A(i, :) \in \mathbb{R}^{R \times 1}$. Examining (7) and (10) side by side, it can be observed that taking the diagonal of the result matrix of (7) yields the result vector of (10), which is the novel approach we desire.

In summary, (7) and (10) demonstrate a novel approach as shown in Fig. 2, and the principal pieces of this way are all matrix multiplications, which is a motivating design for constructing TCU-based parallel algorithms.

V. HYBRID LINEAR STORAGE FORMAT

Typically, a single sparse storage format is used to store sparse tensor. Due to the absence of zero entries, the sparse storage format considerably conserves storage space when compared to

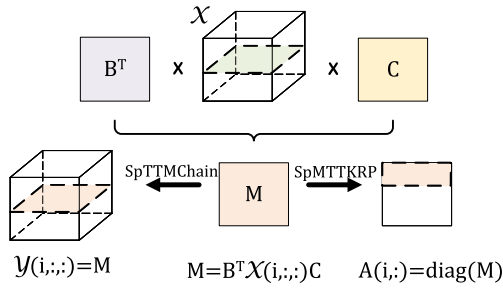


Fig. 2. Graphical representation of the MM-based Approach.

the dense storage format. However, because of the discontinuous storage of nonzeros, sparse formats will cause high memory access delay and performance reduction. To solve the above issues, we propose the Hybrid Linear Storage (HLS) format consisting of Dense Part (HLS-D) and Sparse Part (HLS-S). In this section, we describe HLS format comprehensively.

A. Format Conversion

To overcome the aforementioned problems, the sparse tensor is split into the dense part and sparse part, which will improve the data locality of the former and facilitate the use of TCU to accelerate SpTMCM. Furthermore, the integer index is replaced by a binary linear index in HLS to mitigate the problem of high memory access delay and accelerate data indexing.

Sparse tensor is usually stored in Coordinate (COO) format. The conversion of a sparse tensor $\mathcal{X} \in \mathbb{R}^{6 \times 4 \times 4}$ from COO to HLS format is shown as Fig. 3. The whole process can be divided into 4 stages. The first stage is *Tiling*. The sparse tensor \mathcal{X} is divided into a lot of third-order tiles of size $S_I \times S_J \times S_K$ (this size in Fig. 3 is $2 \times 2 \times 2$). In Fig. 3, t_i, t_j, t_k indicate the index of tiles. Since the irregular nonzero distribution of sparse tensor, *Tiling* will result in non-empty and empty tiles. However, only non-empty tiles are recorded, as empty tiles are irrelevant to the result of SpTMCM.

Classify: After *Tiling*, some non-empty tiles are dense, and others are sparse due to the irregular nonzero distribution of sparse tensors. These tiles are classified into dense and sparse tiles according to threshold T (assuming that a tile is dense if the number of nonzeros of it is great than or equal to T). If sparse tiles are stored as dense, it will cause lots of storage waste. Furthermore, the performance of sparse tiles on TCU is even worse than that on CUDA Core on SpTMCM. Hence, only nonzeros in sparse tiles are recorded, and sparse tiles are released. The tiles (in the rest of this paper, tile refers to dense tiles unless otherwise noted) and discrete nonzeros will be stored as dense and sparse parts, which is described in detail in Section V-B.

Linearize: The multi-dimensional integer index of tiles (nonzeros) are mapped to binary linear index (t_i, t_j , and t_k convert to Binary in Fig. 3), which reduces the complexity of multi-dimensional matching. Note that the index of entries within each tile begins at 0. The process of this stage is described in detail in Section V-B.

Bitmap Represent: Although the use of binary linear index can reduce storage usage, the storage of tiles remain still significant. To further reduce it, the bitmap is used. A set of binary values represents each tile value called a bitmap. It is used to track which items of tile have nonzero values. The corresponding bit in the bitmap is set to 1 if the value is nonzero and 0 otherwise (i, j , and k convert to *bitmap* in Fig. 3).

B. Bit Optimization Techniques

As shown in Fig. 3. After *classify*, the tiles and nonzeros are stored as HLS-D and HLS-S, respectively. Following is a detailed description of HLS format.

HLS-S preserves only the information about nonzeros. Their values of them are preserved in array `nnz_vals`. The number of nonzeros is stored as variable `nnz`. And the number of bits in each mode of nonzero's integer index mapped to the binary linear index is stored in array `nnz_linear_bits`.

For HLS-S, the indices of nonzero in each mode are compressed into a single element (*Binary* in Fig. 3), which is stored in `nnz_inds`. Bit represent of a nonzero is shown as Fig. 4(a). In this example, the mode size of the sparse tensors are 6, 4, and 4, so `nnz_linear_bits` is set as $\{3, 2, 2\}$. The integer indices of this nonzero (1, 3, and 3) convert to binary linear indices are $\{001, 11, 11\}$. So the bit represents 0011111. The bit representation of HLS-D is similar to HLS-S.

For the dense part, HLS-D stores it as a whole tile. The tile size in each mode is kept in array `b_shapes`. And the number of tiles is variable `nmb`. Array `b_inds` is employed to hold the binary linear index of tiles. Array `b_vals_ptr` represents the offset of each tile relative to the first tile. The starting position of the first element of each tile is stored in array `b_vals`. Variable `b_threshold` is tile threshold. And the representation of array `b_linear_bits` is similar to `nnz_linear_bits`.

For bitmap storage of tiles, HLS-D format employs array `b_bmp` to store the bitmap of tiles. Each element of `b_bmp` corresponds to a bitmap of each tile. As shown in Fig. 4(b), this is a $2 \times 2 \times 2$ tile with 3 nonzeros. The relative position of each nonzero in the tile can be calculated by index. Therefore, `b_bmp` is set as $\{10010001\}$.

C. Spatial Complexity Analysis

Consider a sparse tensor $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$ with N nonzeros. It has N_s nonzeros in sparse part. Hence, the space size M_S^i required to store the binary linear index of the sparse part is

$$M_S^i = (\lceil \log_2 I \rceil + \lceil \log_2 J \rceil + \lceil \log_2 K \rceil) \times N_s. \quad (11)$$

In the tensor decomposition application scenario, half is adequate to satisfy the accuracy criteria, so assuming that the data type of the nonzero values is half (16 bits). The space size M_S^v required for nonzero values is

$$M_S^v = 16 \times N_s. \quad (12)$$

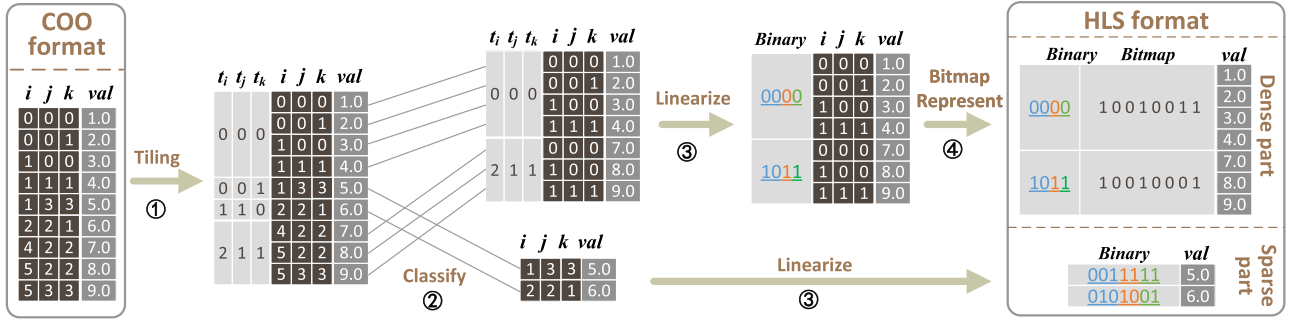


Fig. 3. Example of a tensor converted from COO to HLS format.

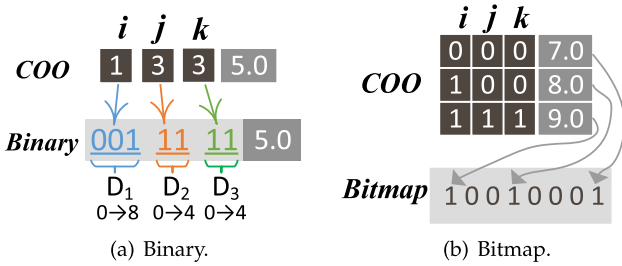


Fig. 4. Examples of two bit optimization techniques.

Assuming that \mathcal{X} has T tiles ($S_I \times S_J \times S_K$). Therefore, the binary linear index of tiles occupies the space size M_D^i is

$$M_D^i = \left(\left\lceil \log_2 \frac{I}{S_I} \right\rceil + \left\lceil \log_2 \frac{J}{S_J} \right\rceil + \left\lceil \log_2 \frac{K}{S_K} \right\rceil \right) \times T. \quad (13)$$

Further, the space size M_{bm} required for all tiles to store indexes is

$$M_{bm} = (S_I \times S_J \times S_K) \times T. \quad (14)$$

And the space size M_D^v required to store the nonzero values in dense part is

$$M_D^v = 16 \times (N - N_s). \quad (15)$$

So total space size M_H required for \mathcal{X} stored in HLS is

$$\begin{aligned} M_H &= M_S^i + M_S^v + M_D^i + M_{bm} + M_D^v \\ &= (\lceil \log_2 I \rceil + \lceil \log_2 J \rceil + \lceil \log_2 K \rceil) \times N_s \\ &\quad + \left(\left\lceil \log_2 \frac{I}{S_I} \right\rceil + \left\lceil \log_2 \frac{J}{S_J} \right\rceil + \left\lceil \log_2 \frac{K}{S_K} \right\rceil \right) \times M \\ &\quad + (S_I \times S_J \times S_K) \times M + 16 \times N. \end{aligned} \quad (16)$$

Dense format stores all element values of a sparse tensor with float, whether or not that value is zero, where the float type is 32 bits. Thus, the aforementioned derivation can lead to the conclusion that HLS takes up substantially less space than Dense format. COO format stores only nonzeros, and for third-order tensor, uses three integer arrays to store indices and a float array to store nonzero values, where the integer type is 32 bits. For HLS-S, it takes up significantly less storage compared to COO

because of the use of the binary linear index. For HLS-D, because bitmap is used, the storage occupied by it does not increase with the increase of nonzeros of tile, and storage occupied is constant. However, for COO, as the number of nonzeros rises, so does the storage that it occupies. Consequently, HLS-D requires less storage when the number of nonzeros of COO exceeds the tile threshold, that is, HLS takes up less space.

VI. HLS-BASED SPTMCM PARALLEL ALGORITHM ON GPU

In this section, the implementation of HLS-based SpTMCM on GPU is demonstrated. First, the parallel algorithm for accelerating SpTMCM using TCU is given. And then, the tile threshold setting and task load scheme regarding this algorithm are discussed.

A. Tcu-Accelerated Sptmcm

The parallel algorithm of HLS-based SpTMCM mainly consists of two kernel functions, dense and sparse. For the dense kernel function, tiles are stored in HLS-D format and are carefully designed to be loaded onto TCU. For the sparse kernel function, tiles are stored in HLS-S format and are loaded onto CUDA Core. Next, we characterize these two kernel functions in detail.

For the nonzeros stored in HLS-D format, one result slice is easily obtained by two multiplications of the corresponding input slice and two factor matrices, as shown in SpTTMChain by (7). SpMTTKRP just adds the operation of taking the diagonal to this in (10). We perform SpTMCM under tiles, as indicated in Fig. 5. The workflow of SpTMCM is broken down into four steps. First, the input tensor \mathcal{X} is partitioned into several tiles. These tiles and two factor matrices are transferred into global memory across PCIe. Second, all the mode-1 slices in each tile are traversed to execute SpTMCM. Once factor matrices are loaded, reusability is built into the following loop. Third, the final product of SpTMCM needs to be processed further. In SpTTMChain, this product is accumulated to the corresponding slice in the result tensor \mathcal{Y} . In SpMTTKRP, this product has only diagonal elements contributing to the result matrix A , so we accumulate these diagonal elements to the corresponding rows in the result matrix A . Finally, the result is transferred to CPU across PCIe.

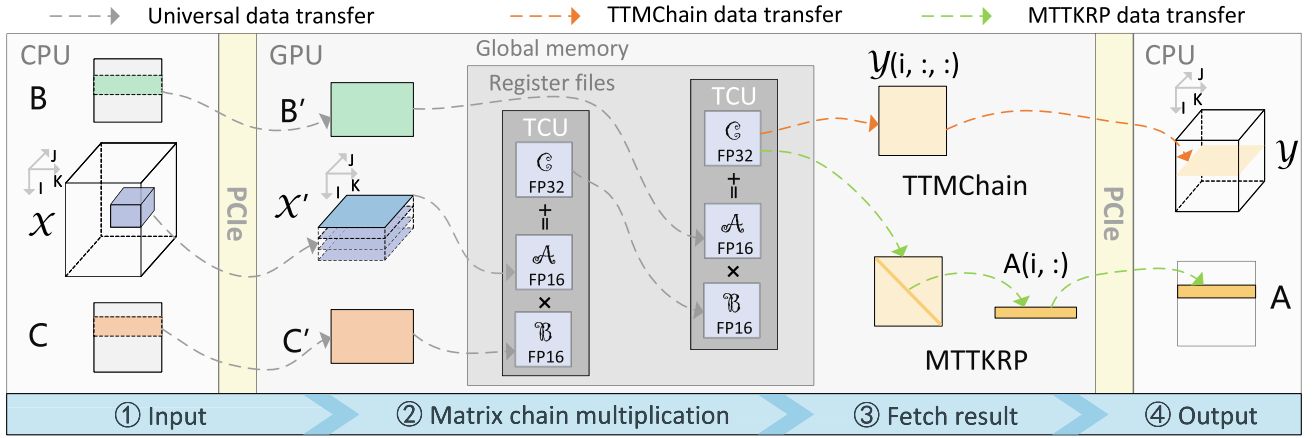


Fig. 5. Flow diagram of TCU-accelerated SpTMCM.

The pseudocode of how to employ TCU for (7) with our HLS-D format is shown in Algorithm 1. The input tensor is divided into tiles in terms of $b_threshold$, whose size is recorded by the array b_shape . The variable nmb denotes the number of tiles. A 1D grid of thread blocks is used, and a thread block is exploited to work on a single tile. Threads are organized in the form of warps and lanes. The 32 threads form a warp and the index of lanes indicates the relative index of the thread within the warp. A thread block can be further divided into warps sized of N_w , where these warps are employed to update a slice of a tensor or a row of a matrix. To make better use of TCU, b_shape is set as $\{16, 16, 16\}$, which means 16 slices, each the sized of 16×16 . The fact that all modes in the tile are the same helps to reduce the conversion overhead of SpTMCM under different modes.

In Fig. 4(a), the indices of tiles are mapped to a binary linear index in HLS format. To get the corresponding matrix location for the current tile, the binary linear index has to be uncompressed to the coordinates. Algorithm 2 represents the conversion function $Binary2COO$ from binary linear index to integer index of COO format, which is called in Line 9 of Algorithm 1. It is obvious that all slices with one fixed mode share the same factor matrices in (7). After factor matrices B and C have been loaded for the first time by a warp, they will always reside in the registers among iterations (Line 12 in Algorithm 1). This yields data reuse for the slices with the same fixed mode.

The loop of each mode-1 slice in the tile is parallelized by warps. Each warp handles $16/N_w$ slices (Line 19 in Algorithm 1). Then, the intermediate product $\mathcal{X}(i, :, :)C$ is obtained by multiplying $\mathcal{X}(i, :, :)$ with C . For matrix multiplication in WMMA, multipliers, multiplicands, and products have their own structs, which are not to be assigned directly to each other. In Line 21 in Algorithm 1, the i -th slice is filled into the multiplier struct in the manner depicted in Fig. 4(b). The intermediate product ($\mathcal{X}(i, :, :)C$) in the product struct should be transferred to the for the second matrix multiplication (Lines 24 and 25 in Algorithm 1). The i -th mode-1 slice is produced by multiplying B^T in the multiplier struct and $\mathcal{X}(i, :, :)C$ in the multiplicand struct for the second matrix multiplication. Finally,

all contributions to the i -th mode-1 slice are accumulated to complete the i -th mode-1 slice in the result tensor. Observed that an additional step is required in SpMTTKRP, which takes the diagonal elements of the last product, the i -th row of the factor matrix A is composed of diagonal elements. While the traditional work followed the element-wise computation, our method is more all-purpose for SpTTMChain and SpMTTKRP. In one tile, factor matrices involved in all slices are invariable the same, which reside in the registers instead of being exchanged. The transformation formula of SpTMCM not only reduces the data transfer but also boosts the arithmetic intensity.

For nonzero stored in HLS-S format, we use an element-wise way to compute the result corresponding to each nonzero element. Each thread block computes an equal number of nonzero elements, and each nonzero element is assigned to a thread. First, each nonzero is assigned to a thread based on its thread number. Second, the binary linear index is converted to an integer index by Algorithm 2. Third, the corresponding element in the result matrix Y is updated by each thread according to (4) in SpMTTKRP, and the result tensor \mathcal{Y} is updated according to (3) in SpTTMChain.

The SpTMCM with HLS format not only expresses the optimization based on TCU but also leads to two key issues: (1) the tile threshold setting of HLS format, which determines the optimal computing budget allocation to minimize the total time consuming; and (2) the task load scheme, which impacts the occupancy on GPU and further reacts to the bandwidth. To address these key issues, we will elaborate on our solutions in the next sections.

B. Tile Threshold of HLS

Different tile thresholds affect the computation of HLS-D and HLS-S formats. If too much or too little computation for each of the two formats, the algorithm will not perform optimally. So it is necessary to find an optimal tile threshold. Experience is commonly used to set tile threshold, that is, use the overall sparsity of sparse tensor to approximate the optimal tile threshold.

Algorithm 1: Kernel Function of SpTTMChain on Mode-1 for Third-Order Tensors With HLS-D Format.

Input:

$b_inds[nnb]$, $b_bmp[64 \times nnb]$, $b_vals[nnz_D]$,
 $b_val_ptr[nnb + 1]$, $b_linear_bits[3]$, $B[J][R_1]$,
 $C[K][R_2]$;

Output:

Result tensor: $\mathcal{Y} \in \mathbb{R}^{I \times R_1 \times R_2}$;

```

1: using namespace nvcuda::wmma;
2:  $id\_w \leftarrow threadIdx.x/32$ ;
3:  $\_\_shared\_\_ uint pre\_shm[64]$ ;
4:  $\_\_shared\_\_ half xcs[N\_w \times 256]$ ;
5: fragment  $x\_frag, c\_frag, m\_frag, bT\_frag,$ 
 $xc\_frag, y\_frag$ ;
6: if  $blockIdx.x < nnb$  then
7: // Get the tile coordinate index.
8:  $b\_index \leftarrow b\_inds[blockIdx.x]$ ;
9:  $b\_COO \leftarrow binary2COO(b\_linear\_bits, b\_index)$ ;
10: // Get the corresponding matrix location.
11:  $B\_t \leftarrow B + b\_COO[1] \times 16 \times R_1$ ,
 $C\_t \leftarrow C + b\_COO[2] \times 16 \times R_2$ ;
12: load_matrix_sync( $bT\_frag, B\_t, R_1$ ),
load_matrix_sync( $C\_frag, C\_t, R_2$ );
13: // Get the bitmap location for the current tile.
14:  $bmp\_t \leftarrow b\_bmp + 64 \times blockIdx.x$ ;
15:  $vals\_t \leftarrow b\_vals + b\_val\_ptr[blockIdx.x]$ ;
16: // Get the prefix sum of the number of bits in the
bitmaps that are set to 1.
17: Get_Prefix_Sum( $pre\_shm, bmp\_t$ );
18:  $off\_w = id\_w \times 16/N\_w$ ;
19: for  $i \leftarrow off\_w$  to  $(id\_w + 1) \times 16/N\_w$  do
20: fill_fragment( $xc\_frag, 0.0f$ );
fill_fragment( $x\_frag, 0.0f$ );
21: Fill2Frag( $x\_frag, vals\_t, pre\_shm, bmp\_t$ );
22: mma_sync( $xc\_frag, x\_frag, c\_frag, xc\_frag$ );
23: store_matrix_sync( $xcs + id\_w \times 256, xc\_frag, 16,$ 
 $mem\_row\_major$ );
24: load_matrix_sync( $m\_frag, xcs + id\_w \times 256, 16$ );
25: mma_sync( $y\_frag, bT\_frag, m\_frag, y\_frag$ );
26:  $\mathcal{Y}(i, :, :) += y\_frag$ ;
27: end for
28: end if
29: return  $\mathcal{Y}$ ;

```

However, there is a certain error between the optimal threshold by this method and the actual optimal threshold. According to the introduction of the previous article, the computing budget allocation of HLS-D format in a tile is fixed, regardless of the number of nonzeros of the tile. However, the computing budget allocation of HLS-S format changes linearly with the number of discrete nonzeros. Therefore, an optimal threshold is such that the computing budget allocation of HLS-S and HLS-D formats in the size of a tile is the same. To find this optimal threshold, a large number of experiments are conducted, which is experimentally verified as described in Section VII-A3.

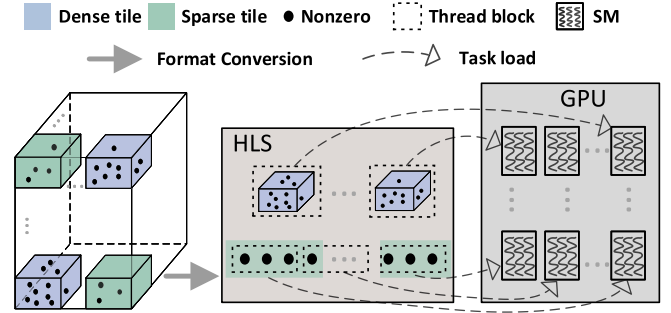


Fig. 6. Diagram of the process by which a tensor in HLS format loads computational tasks onto the SM of the GPU.

Algorithm 2: Binary2COO Function.

Input:

$b_linear_bits[3]$, b_index ;

Output:

$b_COO[3]$;

```

1:  $mask \leftarrow 0xFFFFFFFFFFFFFFFF$ ;
2:  $b0 \leftarrow b\_linear\_bits[0]$ ;
3:  $b1 \leftarrow b\_linear\_bits[1]$ ;
4:  $b2 \leftarrow b\_linear\_bits[2]$ ;
5:  $b\_COO[0] \leftarrow b\_index \gg (b1 + b2)$ ,
 $b\_COO[1] \leftarrow (b\_index \gg b1) \& \sim (mask \ll b1)$ ,
 $b\_COO[2] \leftarrow b\_index \& \sim (mask \ll b2)$ ;
6: return  $b\_COO[3]$ ;

```

C. Task Load Scheme

A suitable task scheduling strategy is beneficial for the performance improvement of operators [36]. The two commonly used task load strategies for hybrid formats on GPU are the sequential load scheme and fusion load scheme. The sequential load scheme is to launch each part of the computational task in the hybrid format as a grid to the GPU in turn, such as CUSP [37], whose hybrid format is implemented by sequentially launching the kernel functions of the dense and sparse parts in turn. The advantage of this scheme is that it is easy to implement. And the disadvantage is that if the number of thread blocks in either part of the computational grid of the hybrid format is too little, then some Streaming Multiprocessors (SMs) are left idle, i.e., low GPU occupancy.

Instead, the fusion load scheme schedules the task load uniformly for hybrid formats, such as TileSpGEMM [38]. If a block of threads from any part of the hybrid format cannot occupy the entire GPU, the fusion load scheme schedules a block of threads from other parts of the hybrid format to enter the GPU early for execution. Also, the advantage of this scheme is high GPU utilization but is more difficult to implement than the sequential load scheme.

To fully utilize the GPU, a fusion load scheme based on HLS is designed. Fig. 6 illustrates a tensor in HLS format loading computational tasks onto the SMs of the GPU. The tensor in HLS format is comprised of dense tiles and discrete nonzeros that are

TABLE II
EXPERIMENTAL PLATFORMS CONFIGURATION

Parameters	blueA100 Platform		blueTITAN Platform	
	Intel(R) Xeon(R) Gold 5120 CPU	NVIDIA A100 GPU	Intel(R) Xeon(R) Silver 4110 CPU	NVIDIA TITAN RTX GPU
Micro architecture	Skylake	Ampere	Skylake	Turing
CUDA Cores	None	6912	None	4608
Tensor Cores	None	432	None	576
Frequency	2.20 GHz	1.41 GHz	2.10 GHz	1.77GHz
Memory size	64 G	40 G	132 G	24 G
Last-level-cache	19.25 M	80 M	11 M	6 M
Compiler	GCC 7.5.0	NVCC 11.2	GCC 5.4.0	NVCC 10.1

loaded onto the SMs in distinctive ways. The computational tasks of each tile are organized into a thread block and executed in a novel MM-based way, described in detail in Section VI-A. It is refined to nonzero granularity since discrete nonzeros are inhospitable to MM. Each thread block is designed to handle a batch of an equal number of nonzeros to ensure load balancing among each thread block.

VII. EXPERIMENTS

In this section, a large number of experiments are used to fully evaluate our proposed method to **Hybrid Linear Storage Format Based Tensor Matrix Chain Multiplication Operation** (abbreviated HLSTO). First, the parameter settings of HLSTO are given, and the effectiveness of the convergent load optimization scheme is verified. Second, the storage space and computational bandwidth of HLS format are analyzed, and the advantages and disadvantages of the HLS format are explained. Third, the current state-of-the-art SpTMCM implementations are compared to demonstrate the superiority and advancement of the overall performance of our method. In our experiments, the values shown are the average of the results of 100 repetitions due to the relatively small standard deviation rates.

A. Setup

1) *Platforms*: We conduct experiments on two different architectures of GPUs: NVIDIA A100 GPU (abbreviated A100) with Ampere architecture, which is mounted on an Intel(R) Xeon(R) Gold 5120 CPU, and NVIDIA TITAN RTX GPU (abbreviated TITAN) with Volta architecture, which is mounted on an Intel(R) Xeon(R) Silver 4110 CPU. The full configuration is shown in Table II.

2) *Datasets*: We evaluate our method on six third-order sparse tensors that are obtained from real-world datasets with varying characteristics. These six real-world tensor datasets are from GroupLens Research¹ and Didi Chuxing GAIA Initiative.² Table III is a summary of the six datasets.

The first three datasets exhibited in Table III are all published by GroupLens Research. The movie recommendation service MovieLens members' anonymous movie ratings are described in

TABLE III
SUMMARY OF DATASETS

Datasets	I	J	K	nonzeros	sparsity
Movies	1,238	5,665	1,078	11,440,996	0.152%
Art	669	379	251	63,066	0.991%
Book	364	104	227	93,866	1.092%
G1	480	875	749	10,685,223	3.397%
G2	480	875	749	10,806,566	3.435%
G3	480	875	749	10,638,476	3.381%

the dataset *Movies*.³ The *Art* dataset⁴ includes social networking, tagging, and musician listening data from a group of 2000 users of the internet music service Last.fm. The *Book* dataset includes social networking, bookmarking, and tagging information from a group of 2000 users of Delicious social bookmarking system. The last three datasets *G1*, *G2*, and *G3* come from the vehicle trajectory data under different dates in the same scenario, and they reflect the similarity of the tensor in the actual usage scenario.

3) *Tile Threshold Setting*: In the multi-dimensional tiling approach, the size of tiles is set as $16 \times 16 \times 16$. To determine the tile threshold, lots of tensors with different tile densities are generated as test datasets. When these datasets are generated, the sparsity of each mode is set to control the overall sparsity of the sparse tensor, and the density within tiles is set to control the test of tile thresholds within tiles. The optimal threshold is found by keeping the sparsity of the sparse tensor the same in each mode and adjusting the threshold for each experiment. The experiment results are illustrated in Fig. 7. It clearly shows that the optimal density of tiles in SpMTTKRP and SpTTMChain is 1.9% and 2.0% on A100, 1.5% and 0.9% on TITAN, i.e., the optimal thresholds of tiles are 78 and 61 on A100, 82 and 41 on TITAN. In the following experiments, if the threshold values of HLS are not specified, the above threshold values are used by default.

4) *Validity of Fusion Load Strategy*: To better verify the validity of the fusion load scheme, several experiments are carried out. Since the fusion scheme is only effective in the case of low GPU occupancy, which means that if a tensor in HLS format with dense tiles and or discrete nonzeros cannot occupy the entire GPU, our fusion load scheme will have a better performance improvement. In the datasets of our experiments, this is the case only for *Art* and *Book*, so we focus on showing their experimental results.

Fig. 8 shows the time overhead comparison before and after performing the fusion load scheme of the implemented HLSTO. As can be seen from the figure, performing the fusion load scheme achieves better performance improvement. On A100, it achieves a speedup of $8.30\times$ and $7.00\times$ on the *Art* and *Book* tensors of the implemented SpMTTKRP respectively, and $5.00\times$ and $1.38\times$ of the implemented SpTTMChain. Note that SpTTMChain has a relatively low execution time on *Book* dataset. It has two reasons. First, compared to the SpMTTKRP and

¹<https://grouplens.org>

²<https://gaia.didichuxing.com>

³<https://grouplens.org/datasets/movielens/>

⁴<https://grouplens.org/datasets/hetrec-2011/>

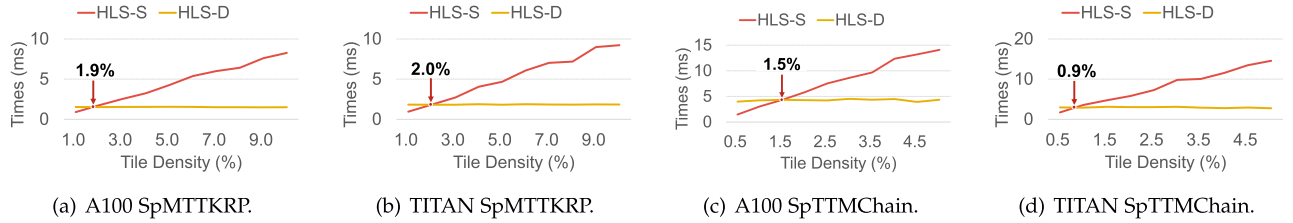


Fig. 7. Time overhead of tensor with various tile density.

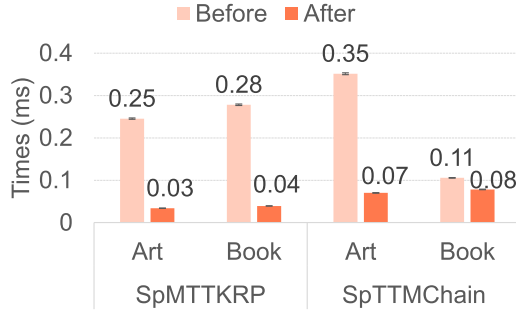


Fig. 8. Comparison of the time overhead before and after performing fusion task load scheme on A100.

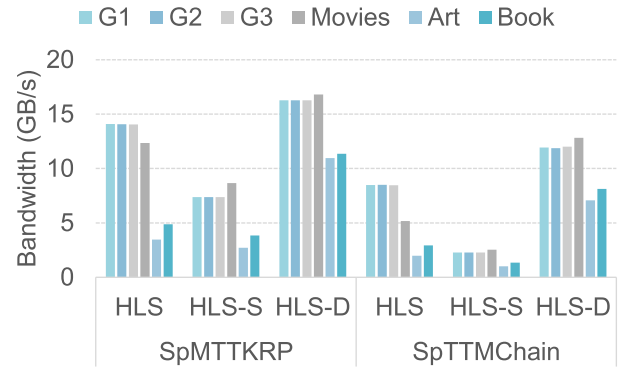


Fig. 9. Bandwidth of SpTMCM on A100.

TABLE IV
MEMORY USAGE (MB) OF DATASETS

Tensor	blueHLS		HLS-S	HLS-D	COO	Dense		
	blueSpTTMChain							
	A100	blueTITAN						
Movies	49.88	blue49.63	51.58	blue55.48	65.47	175.25	174.58	28840.28
Art	0.25	blue0.25	0.27	blue0.29	0.30	0.87	0.96	242.77
Book	0.39	blue0.39	0.41	blue0.45	0.45	1.04	1.43	32.78
G1	38.11	blue37.95	38.98	blue40.45	50.95	48.23	163.04	1200.03
G2	38.26	blue38.09	39.17	blue40.61	50.95	48.35	164.90	1200.03
G3	38.00	blue37.83	38.87	blue40.40	50.73	48.23	162.33	1200.03

SpTTMChain over *Book*, SpMTTKRP has more computation steps and higher arithmetic intensity than SpTTMChain (see Step 3 in Fig. 5), which makes it easier for SpMTTKRP to make the GPU fully load. Second, compared to the SpTTMChain over *Art* and *Book*, *Book* is denser than *Art* (see the sparsity in Table III). So *Book* is more suitable for computation using our algorithm. To sum up, SpTTMChain has a relatively low execution time on the *Book* tensor.

B. Effectiveness of HLS

1) *Memory Usage Analysis*: Table IV details the relative storage of the sparse tensors in HLS format on A100. The indices of each format are stored using integers, and the values of each format are stored using floating-point numbers, but different formats use different floating-point precision. For instance, the precision of the HLS (HLS-D and HLS-S) format is 16 bits, compared to the 32 bits of the Dense and COO formats. Note that SpMTTKRP and SpTTMChain have different memory usage for the same datasets in HLS format, whose optimal thresholds are distinct in Section VII-A3. The HLS (SpMTTKRP) and HLS (SpTTMChain) in the table indicate the storage space required

for HLS under the two threshold conditions, and HLS-D, HLS-S, COO, and Dense indicate the storage space required to store the entire tensor in that format. It can be seen that the storage space overhead for storing the sparse tensor in Dense format is the largest, followed by COO format. Furthermore, the storage space of HLS is smaller than that of HLS-D and HLS-S for both the threshold conditions of SpMTTKRP and SpTTMChain. For *G1*, *G2*, and *G3* tensors, the storage space of HLS-D is less than that of HLS-S because the trajectory data has the property of well block sparsity, which means that the nonzeros in the tensor are always distributed in a minority of tiles.

Overall, the experimental results show that HLS reduces storage space by 19.95% on average compared to HLS-S and 43.83% on average compared to HLS-D. Therefore, the hybrid format is more beneficial to save storage space. More, HLS reduces storage space by an average of 74.17% over COO format and 97.85% over Dense format. Therefore, HLS format demands less storage space compared to other formats.

2) *Bandwidth Analysis*: Fig. 9 depicts the memory bandwidth of SpMTTKRP and SpTTMChain average on three modes. Statistics are given for storage in fully sparse (HLS-S), fully dense (HLS-D) and hybrid (HLS) formats on A100. Since these different formats only compress the index data, we only count the memory space of the indices. SpTMCM with HLS results in a higher bandwidth than that with HLS-S format. The bandwidth of *G1*, *G2*, and *G3* datasets can reach 14 GB/S using HLS format, and only 13 GB/S with HLS-S format. However, the bandwidth under HLS-D format is higher, which reaches about 16 GB/s. The overall throughput with HLS is lower compared to

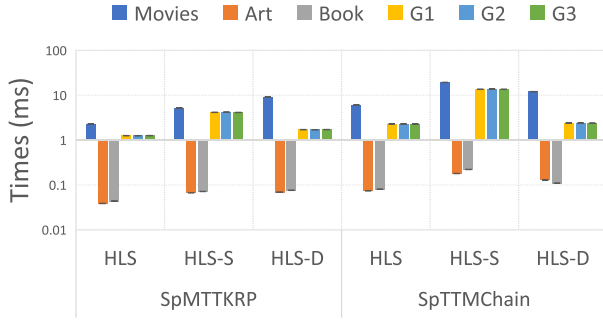


Fig. 10. Overhead of SpTMCM on A100.

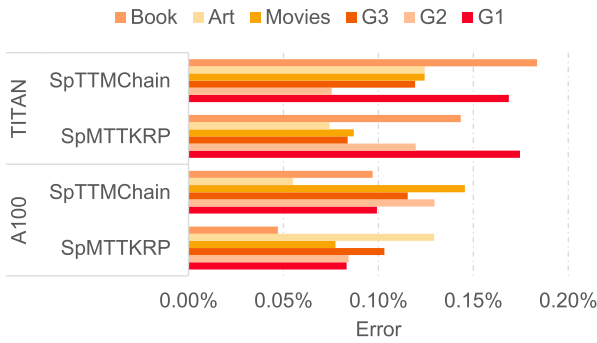


Fig. 11. Error of SpTMCM.

that with HLS-D due to the dense storage approach. The memory overheads incurred by HLS-D are much higher than those of HLS. The magnitude of the memory improvement takes place at a lower rate than the rate of time improvement. The memory analysis shows that HLS achieves greater memory overheads for all tensors compared to HLS-S and HLS-D, especially in *Movies* tensor and *Art* tensor, which gets about 5 times the importance. This result demonstrates that our hybrid format (HLS) decreases the memory overheads on GPU effectively and increases the bandwidth of sparse tensors significantly.

Fig. 10 exhibits the time overhead of SpMTTKRP and SpTTMChain for HLS, HLS-S, and HLS-D on A100. For SpMTTKRP, HLS has slightly lower performance than HLS-S and HLS-D for the datasets with few nonzeros, such as *Art* and *Book*. For most datasets with lots of nonzeros, such as *Movies*, *G1*, *G2*, and *G3*, HLS significantly has better performance than HLS-S and HLS-D. Thus, for SpMTTKRP, HLS achieves a speed-up of $1.35\times$ to $3.98\times$. For SpTTMChain, the performance of HLS is better than HLS-S and HLS-D on all datasets. HLS achieves a speed-up of $1.04\times$ to $6.00\times$.

Overall, HLS improves the computational bandwidth by 136.00% on average compared to HLS-S. Compared to HLS-D, HLS reduces the computational bandwidth by only 36.41% on average, even if it stores a lot more space (43.83%) for invalid indices.

3) *Error Analysis*: TCUs perform SpTMCM with 16-bit precision. Fig. 11 shows the Symmetric Mean Absolute Percentage

Error in comparison to a full 32-bit approach on CPU ((17))

$$Error = \frac{100\%}{n} \sum_{i=1}^n \frac{|x_i - \hat{x}_i|}{|x_i| + |\hat{x}_i|}. \quad (17)$$

Since the two SpTMCM return different results, the way they calculate the error is slightly different. SpTTMChain considers the error between two tensors, and SpMTTKRP considers the error between two factor matrices. From the experimental results, we can see that their errors are below 0.17% on both A100 and TITAN GPUs. The lowest relative error reaches 0.04% on A100. This means that HLSTO has low relative errors on different GPU architectures, and therefore HLSTO is scalable.

C. Comparison to Other Implementations

1) *Runtime Overhead*: To better demonstrate the progress of our HLSTO method, we chose other recent research results (Hi-COO [24], BLCO [22], MM-CSF [31], F-COO [9], FWC [34], B-CSF [21]) as the baseline method. The above latest research work is presented in Section III. In summary, Hi-COO, BLCO, F-COO, and our proposed HLSTO are all extensions of COO format, whereas MM-CSF, FWC, and B-CSF are all extensions of Compressed Sparse Row (CSR) storage format.

Fig. 12 shows the performance of SpMTTKRP implemented by all methods used in the experiments. The Cut-off line is given as an upper bound for the demonstrated time overhead because the experimental overhead is too large for some of the experiments. And mode-1 is abbreviated as m-1, and it can be seen that HLSTO has more stable performance on mode-1, mode-2, and mode-3. Specifically, for tensors with relatively few nonzero, such as *Art*, *Book*, HLSTO has less time overhead than any other methods. For most datasets with a large number of nonzeros, such as *G1*, *G2*, and *G3*, HLSTO has better performance relative to all other comparison methods. Only for tensors with high sparsity such as *Movies*, HLSTO will have more time overhead than methods such as BLCO, B-CSF, etc., but still has a large advantage over methods such as MM-CSF, HiCOO, etc. Overall, for SpMTTKRP, our HLSTO method achieves a speedup ratio of $1.16\times$ to $24.12\times$.

Since some of the comparison methods do not implement SpTTMChain, we choose the methods in which SpTTMChain exists as the baseline. Fig. 13 shows the time overhead of SpTTMChain implemented by Hi-COO, MM-CSF, and HLSTO. It can be seen that our methods all have well performance improvement with respect to the baseline methods. Overall, for SpTTMChain, our HLSTO method achieves a speedup ratio of $5.07\times$ to $7.15\times$.

2) *Format Construction Overhead*: The conversion from the standard sparse format COO to other formats is preprocessed before the actual computation. That time overhead is usually much larger than the single runtime overhead of tensor matrix chain multiplication. In practice, SpTMCM usually needs to be iterated hundreds or thousands of times, and these iterations will amortize the additional overhead from preprocessing. Unlike other methods, our approach needs to generate a bitmap for each block to utilize TCU, which leads to many atomic operations.

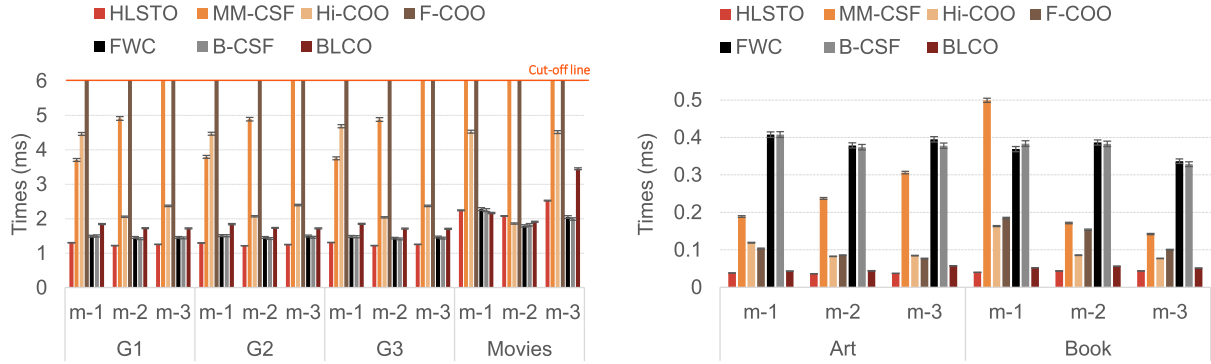


Fig. 12. SpMTTKRP performance of different methods on A100.

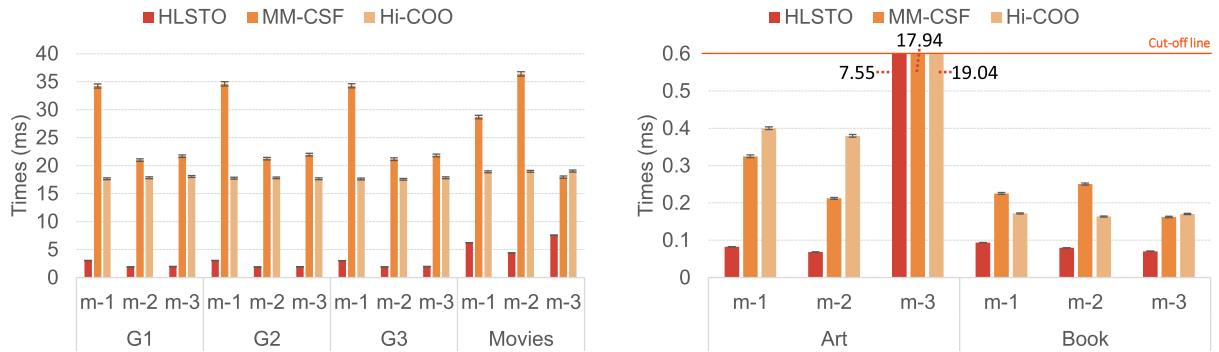


Fig. 13. SpTTMChain performance of different methods on A100.

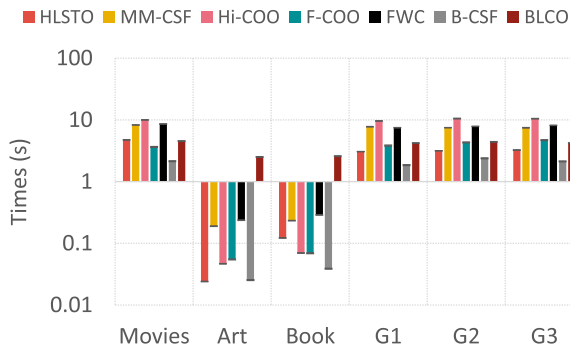


Fig. 14. Conversion times from the standard sparse format COO to other formats.

Hence, we design fine-grained data structures to avoid atomic operations.

Fig. 14 shows the conversion time from the standard sparse format COO to other formats, and it can be seen that the conversion time required by all advanced formats is measured in seconds. Compared to other methods, our method requires less conversion time than most, which means that our method is highly competitive.

VIII. CONCLUSION

In this paper, we have addressed the efficient implementation of SpTMCM on GPU with a novel approach. SpMTTKRP and SpTTMChain are treated as the SpTMCM, which facilitates bandwidth enhancement with the assistance of TCU. Several techniques were used to further improve the performance of SpTMCM. A hybrid format called HLS was designed to divide the tensor into multiple rectangular tiles to achieve better data locality. Thresholds for HLS were obtained through extensive experimentation. Fusion load strategies were employed to facilitate GPU utilization. The experiments showed that our method outperformed state-of-the-art tensor techniques on GPU.

Our method applies to higher dimensional tensors above three-dimensional ones, but the bitmap mapping needs to be redesigned. The reason is that bitmaps are linear, but blocks are multidimensional, so the mapping relationship between bitmaps and tensor blocks changes as the dimensionality increases. We will extend our method from a three-dimensional to a higher-dimensional tensor in future work.

REFERENCES

- [1] J.-G. Jang and U. Kang, "Fast and memory-efficient tucker decomposition for answering diverse time range queries," in *Proc. 27th ACM SIGKDD Conf. Knowl. Discov. Data Mining*, 2021, pp. 725–735.

- [2] R. Warlop, J. Mary, and M. Gartrell, "Tensorized determinantal point processes for recommendation," in *Proc. 25th ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2019, pp. 1605–1615.
- [3] K. Xie et al., "Fast tensor factorization for accurate internet anomaly detection," *IEEE/ACM Trans. Netw.*, vol. 25, no. 6, pp. 3794–3807, Dec. 2017.
- [4] T. Kolda and B. Bader, "Tensor decompositions and applications," *SIAM Rev.*, vol. 51, pp. 455–500, 2009.
- [5] Q. Sun et al., "Input-aware sparse tensor storage format selection for optimizing MTTKRP," *IEEE Trans. Comput.*, vol. 71, no. 8, pp. 1968–1981, Aug. 2022.
- [6] S. Smith and G. Karypis, "Accelerating the tucker decomposition with compressed sparse tensors," in *Proc. Eur. Conf. Parallel Process.*, 2017, pp. 653–668.
- [7] T. Weng, X. Zhou, K. Li, P. Peng, and K.-C. Li, "Efficient distributed approaches to core maintenance on large dynamic graphs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 1, pp. 129–143, Jan. 2022.
- [8] T. Weng, X. Zhou, K. Li, K.-L. Tan, and K.-C. Li, "Distributed approaches to butterfly analysis on large dynamic bipartite graphs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 34, no. 2, pp. 431–445, Feb. 2023.
- [9] B. Liu, C. Wen, A. D. Sarwate, and M. M. Dehnavi, "A unified optimization approach for sparse tensor operations on GPUs," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2017, pp. 47–57.
- [10] F. Silvestri and F. Vella, "A computational model for tensor core units," in *Proc. 32nd ACM Symp. Parallelism Algorithms Architectures*, 2020, pp. 519–521.
- [11] A. Li and S. Su, "Accelerating binarized neural networks via bit-tensor-cores in turing GPUs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 7, pp. 1878–1891, Jul. 2021.
- [12] C. Psarras, L. Karlsson, and P. Bientinesi, "The landscape of software for tensor computations," 2021, *arXiv:2103.13756*.
- [13] Y. Ma, J. Li, X. Wu, C. C. Yan, J. Sun, and R. W. Vuduc, "Optimizing sparse tensor times matrix on GPUs," *J. Parallel Distrib. Comput.*, vol. 129, pp. 99–109, 2019.
- [14] T. Zhang, X.-Y. Liu, X. Wang, and A. Walid, "cuTensor-tubal: Efficient primitives for tubal-rank tensor learning operations on GPUs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 3, pp. 595–610, Mar. 2020.
- [15] N. Srivastava, H. Jin, S. Smith, H. Rong, D. H. Albonese, and Z. Zhang, "Tensaurus: A versatile accelerator for mixed sparse-dense tensor computations," in *Proc. IEEE Int. Symp. High Perform. Comput. Architecture*, 2020, pp. 689–702.
- [16] O. Kaya and B. Uçar, "High performance parallel algorithms for the tucker decomposition of sparse tensors," in *Proc. 45th Int. Conf. Parallel Process.*, 2016, pp. 103–112.
- [17] Q. Xiao, S. Zheng, B. Wu, P. Xu, X. Qian, and Y. Liang, "Towards agile hardware and software co-design for tensor computation," in *Proc. ACM/IEEE 48th Annu. Int. Symp. Comput. Architecture*, 2021, pp. 1055–1068.
- [18] O. Kaya, "High performance parallel algorithms for tensor decompositions," Ph.D. dissertation, Univ. Lyon, Lyon, France, 2017.
- [19] S. Oh, N. Park, J.-G. Jang, L. Sael, and U. Kang, "High-performance tucker factorization on heterogeneous platforms," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 10, pp. 2237–2248, Oct. 2019.
- [20] N. Abubaker, S. Acer, and C. Aykanat, "True load balancing for matricized tensor times Khatri-Rao product," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 8, pp. 1974–1986, Aug. 2021.
- [21] I. Nisa, J. Li, A. Sukumaran-Rajam, R. W. Vuduc, and P. Sadayappan, "Load-balanced sparse MTTKRP on GPUs," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2019, pp. 123–133.
- [22] A. K. Nguyen et al., "Efficient, out-of-memory sparse MTTKRP on massively parallel architectures," in *Proc. 36th ACM Int. Conf. Supercomputing*, 2022, pp. 26:1–26:13.
- [23] J. W. Choi, X. Liu, S. Smith, and T. A. Simon, "Blocking optimization techniques for sparse tensor computation," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2018, pp. 568–577.
- [24] J. Li, J. Sun, and R. W. Vuduc, "HiCOO: Hierarchical storage of sparse tensors," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2018, pp. 238–252.
- [25] J. Choquette, W. Gandhi, O. Giroux, N. Stam, and R. Krashinsky, "NVIDIA A100 tensor core GPU: Performance and innovation," *IEEE Micro*, vol. 41, no. 2, pp. 29–35, Mar./Apr. 2021.
- [26] C. A. Navarro, R. Carrasco, R. J. Barrientos, J. A. Riquelme, and R. Vega, "GPU tensor cores for fast arithmetic reductions," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 1, pp. 72–84, Jan. 2021.
- [27] H. Huang, X.-Y. Liu, W. Tong, T. Zhang, A. Walid, and X. Wang, "High performance hierarchical tucker tensor learning using GPU tensor cores," *IEEE Trans. Comput.*, vol. 72, no. 2, pp. 452–465, Feb. 2022.
- [28] O. Kaya and B. Uçar, "Scalable sparse tensor decompositions in distributed memory systems," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2015, pp. 1–11.
- [29] S. Liu et al., "Hadamard, Khatri-Rao, kronecker and other matrix products," *Int. J. Inf. Syst. Sci.*, vol. 4, no. 1, pp. 160–177, 2008.
- [30] J. Li, B. Uçar, Ü. V. Çatalyürek, J. Sun, K. J. Barker, and R. W. Vuduc, "Efficient and effective sparse tensor reordering," in *Proc. ACM Int. Conf. Supercomputing*, 2019, pp. 227–237.
- [31] I. Nisa, J. Li, A. Sukumaran-Rajam, P. S. Rawat, S. Krishnamoorthy, and P. Sadayappan, "An efficient mixed-mode representation of sparse tensors," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2019, pp. 49:1–49:25.
- [32] X.-Y. Liu, H. Lu, and T. Zhang, "cuTensor-CP: High performance third-order CP tensor decomposition on GPUs," in *Proc. IJCAI Workshop Tensor Netw. Representations Mach. Learn.*, 2020.
- [33] J. Li, J. W. Choi, I. Perros, J. Sun, and R. W. Vuduc, "Model-driven sparse CP decomposition for higher-order tensors," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2017, pp. 1048–1057.
- [34] R. Hu, W. Yang, X. Zhou, K. Li, and K. Li, "Performance analysis and optimization for MTTKRP of sparse tensor on CPU and GPU," in *Proc. IEEE 22nd Int. Conf. High Perform. Comput. Commun.; IEEE 18th Int. Conf. Smart City; IEEE 6th Int. Conf. Data Sci. Syst.*, 2020, pp. 545–550.
- [35] K. Hayashi, G. Ballard, Y. Jiang, and M. J. Tobia, "Shared-memory parallelization of MTTKRP for dense tensors," in *Proc. 23rd ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2018, pp. 393–394.
- [36] K. Li, X. Tang, B. Veeravalli, and K.-C. Li, "Scheduling precedence constrained stochastic tasks on heterogeneous cluster systems," *IEEE Trans. Comput.*, vol. 64, no. 1, pp. 191–204, Jan. 2015.
- [37] Y. Niu, Z. Lu, H. Ji, S. Song, Z. Jin, and W. Liu, "TileSpGEMM: A tiled algorithm for parallel sparse general matrix-matrix multiplication on GPUs," in *Proc. 27th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2022, pp. 90–106.
- [38] S. Dalton, N. Bell, L. Olson, and M. Garland, "CUSP: Generic parallel algorithms for sparse matrix and graph computations," 2014, version 0.5.0. [Online]. Available: <http://cusplibrary.github.io/>



Haotian Wang received the BS degree from the College of Information Engineering, Nanchang University, China, in 2018. He is currently working toward the PhD degree with the College of Information Science and Engineering, Hunan University, China. His research interests include parallel computing, artificial intelligence, and data mining.



Wangdong Yang received the PhD degree in computer science from Hunan University, China. He is a professor of computer science and technology with Hunan University, China. His research interests include modeling and programming for heterogeneous computing systems, parallel and distributed computing, and numerical computation. He has published more than 60 papers in International conferences and journals.



Rong Hu received the BS degree from Chang'an University, China, and the MS degree from Hunan University, China, where she is currently working toward the PhD degree. Her research interests include parallel and scientific computing, with focus on sparse tensor decomposition.



Renqiu Ouyang received the BS degree from the Hunan University of Technology, China. He is currently working toward the PhD degree. His research interests include parallel and scientific computing, with focus on sparse tensor decomposition.



Kenli Li (Senior Member, IEEE) received the MS degree in mathematics from Central South University, China, in 2000, and the PhD degree in computer science from the Huazhong University of Science and Technology, China, in 2003. He was a visiting scholar with the University of Illinois at Urbana-Champaign from 2004 to 2005. He is a full professor of computer science and technology with Hunan University. His research interests include parallel and distributed processing, supercomputing and cloud computing, high-performance computing for Big Data and artificial intelligence, etc. He has published more than 300 papers in international conferences and journals. He is currently served on the editorial boards of *IEEE Transactions on Computers*. He is an outstanding member of CCF.



Keqin Li (Fellow, IEEE) is a SUNY distinguished professor of computer science with the State University of New York. He is also a national distinguished professor with Hunan University, China. His current research interests include cloud computing, fog computing and mobile edge computing, energy-efficient computing and communication, embedded systems and cyber-physical systems, heterogeneous computing systems, Big Data computing, high-performance computing, CPU-GPU hybrid and cooperative computing, computer architectures and systems, computer networking, machine learning, intelligent and soft computing. He has authored or coauthored more than 870 journal articles, book chapters, and refereed conference papers, and has received several best paper awards. He holds nearly 70 patents announced or authorized by the Chinese National Intellectual Property Administration. He is among the world's top 5 most influential scientists in parallel and distributed computing in terms of both single-year impact and career-long impact based on a composite indicator of Scopus citation database. He has chaired many international conferences. He is currently an associate editor of the *ACM Computing Surveys* and the *CCF Transactions on High Performance Computing*. He has served on the editorial boards of the *IEEE Transactions on Parallel and Distributed Systems*, the *IEEE Transactions on Computers*, the *IEEE Transactions on Cloud Computing*, the *IEEE Transactions on Services Computing*, and the *IEEE Transactions on Sustainable Computing*. He is an AAIA fellow. He is also a member of Academia Europaea (Academician of the Academy of Europe).