BCB-SpTC: An Efficient Sparse High-Dimensional Tensor Contraction Employing Tensor Core Acceleration

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

***Abstract*—Sparse tensor contraction (SpTC) is an important operator in tensor networks, which tends to generate a large amount of sparse high-dimensional data, placing higher demands on the computational performance and storage bandwidth of the processor. Using GPUs with powerful arithmetic characteristics is a reliable choice for accelerating SpTC, however, the high dimensionality and sparsity of tensor makes GPU-accelerated SpTC operators suffer from the difficulties of low computational intensity and high memory consumption. The recent introduction of Tensor Core Units (TCUs) on GPUs brings even more powerful arithmetic, which exacerbates the memory wall problem. To cope with the challenges, this paper proposes a new BCB format that linearizes the indices of multidimensional blocks to reduce block index accesses and uses a bitmap to store the distribution of non-zero elements in a block to reduce the storage overhead. A parallel blocking algorithm of BCB-SpTC is designed to divide the binary linear indices into free and contracted indexes to improve the pairing overhead of computational tasks. Then based on the characteristic computation method of TCUs, the proprietary filling method of TCUs is designed to overcome the inefficiency of parallel computation of sparse data on TCUs. Finally, experimental results on the A100 dataset show that BCB-SpTC improves the acceleration ratio by $1.1\times$ to $21.3\times$ over the existing SpTC GPU method.**

***Index Terms*—Bit optimization, GPU, sparse tensor contraction (SpTC), sparse tensor format, tensor core.**

Received 5 September 2023; revised 6 September 2024; accepted 2 October 2024. Date of publication 10 October 2024; date of current version 21 October 2024. This work was supported in part by the National Key R&D Program of China under Grant 2021YFB0300800, in part by the Key Program of National Natural Science Foundation of China under Grant U21A20461, Grant 92055213, and Grant 62227808, in part by the Natural Science Foundation of Hunan Province, China, under Grant 2021JJ50158, in part by China Postdoctoral Science Foundation under Grant 2024M750879, in part by the China National Postdoctoral Program for Innovative Talents under Grant BX20240111, in part by the CCF-Huawei Populus Grove Fund under Grant CCF-HuaweiSY202407, and in part by the Major Projects of Xiangjiang Laboratory under Grant 22xj01011. Recommended for acceptance by A. Randles. (*Rong Hu and Haotian Wang contributed equally to this work.) (Corresponding author: Wangdong Yang.)*

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

Kenli Li is with the College of Computer Science and Electronic Engineering, Hunan University, Changsha 410082, China, also with the National Supercomputing Center in Changsha, 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.2024.3477746

I. Introduction

SPARSE tensor contraction (SpTC) has drawn a lot of interest from researchers in a variety of domains, such as quantum physics [1], [2], [3] and machine learning [4], [5]. For example, tensor contraction is a major operation in tensor networks and tensor decomposition algorithms [6]. The tensor contraction layer is a network structure with the ability to extract deep features in deep learning [7]. In these applications, tensors usually represent large datasets or high-dimensional physical models and need to be reshaped and reduced during computation. Tensor contraction can be traced back to matrix multiplication in linear algebra. Unlike matrix multiplication, tensor contraction can operate on tensors of any dimension. In tensor computation, the dimensions of tensors can sometimes be very large, so effectively reducing the dimensionality of tensors is crucial.

SpTC is confronted by a constellation of challenges, drawing parallels with the difficulties encountered in the realm of sparse matrices. These challenges encompass a range of issues, including the ambiguity of output dimensions, the need to anticipate non-zero patterns prior to computation, and the intricate puzzle of load imbalance [8], [9]. Moreover, SpTC is faced with the challenge of dealing with the complex subtleties that arise from the unique features associated with high-dimensional tensor contractions.

One of the primary issues faced by SpTC pertains to the complex field of multi-dimensional index matching. The presence of this intricate nature leads to the emergence of unpredictable memory access patterns, resulting in a significant negative effect on the overall performance landscape. In certain situations, the fundamental characteristics of SpTC require the utilization of various modes of tensor contraction, thereby demanding the coordination of matching for indices in many dimensions. Nevertheless, the inclusion of this orchestration element presents a possible burden on performance, hence introducing an extra level of complexity to the computing process.

Another significant challenge arises in the shape of a huge increase in memory usage. The overhead in question arises due to the inclusion of tensors and their intermediate products within the underlying structure of GPU. Empirical investigations have shown that sparse tensors can consume significant memory, typically several gigabytes. Previous studies report

usage ranging from 1.1 GB to 99.3 GB depending on tensor size and sparsity [10]. The memory usage is exacerbated by the dimensions of intermediary products and output tensors, thus compounding the footprint. The increasing memory requirements surpass the limitations of GPU capabilities, which becomes especially evident when the size of tensors continues to grow.

The third predicament of SpTC revolves around the phenomenon of diminished arithmetic intensity[11] during computational processes. Arithmetic intensity, a pivotal metric quantifying the ratio of arithmetic operations to the distinct data elements accessed, experiences a gradual decline as the dimensionality of sparse tensors increases. This reduction in arithmetic intensity is attributed to the inherent sparsity that emerges in higher-dimensional tensors. This challenge is particularly magnified in the context of SpTC, where higher-dimensional tensors are frequently encountered, ultimately resulting in a detrimental effect on the overall performance of the tensor contraction process.

Limited by the development of hardware platforms, optimizing the performance of tensor contraction is an important research direction [9], [12]. In particular, there is very little work about SpTC on GPU, because most of the current works are focused on dense tensor contraction [13], [14] and CUDA Cores [15]. While the processing capacity of GPU has greatly increased for dense tensor operations, it is significantly more difficult to employ Single Instruction Multiple Data (SIMD) for sparse tensor operations, such as sparse tensor-times-vector and SpTC, because of irregular data accesses, despite their intrinsic parallelisms matching SIMD computing power well. With the ever-increasing computing power demands from deep learning, more specialized hardware for matrix multiplications, such as Tensor Cores Units (TCUs), started to become available inside GPU. However, the majority of the existing sparse tensor storage formats are incompatible with such hardware accelerators because they are only designed to support small blocked dense matrices. TCUs are challenging to use in SpTC due to the high dimensional sparsity of the sparse tensor. TCUs are not effectively utilized in SpTC as sparse tensors usually exhibit high-dimensional sparse features and extremely sparse distribution of non-zero elements. Therefore, SpTC calculations cannot directly employ TCUs, and as the tensor dimension grows, this issue worsens.

To overcome the above difficulties, we

- propose a new sparse tensor storage format, Bit coordinate Bitmap (BCB), which intuitively reduces memory footprint and memory accesses, and is used to accelerate data index matching and memory accesses to alleviate the memory wall problem caused by high-speed computation in TCUs.
- design a parallel algorithm for SpTC based on BCB format (BCB-SpTC), which introduces a conflict resolution method to increase the occupancy of thread blocks by constructing task lists and removing dependencies between tasks to improve GPU utilization.
- implement a TCUs-based kernel function that adjusts the filling of TCUs according to different scaling tasks to overcome the inefficiency of parallel computation of sparse data on TCUs.
- conduct experiments with real-world datasets and the findings demonstrate that BCB format has a smaller memory footprint than current formats and the BCB-SpTC improves the acceleration ratio by $1.1\times$ to $21.3\times$ over the state-of-the-art method on the A100 GPU.

The rest of the paper is organized as follows. Section II presents the preliminaries of SpTC and TCUs. Section III describes the Bitmap Coordinates Format. Section IV gives the BCB-SpTC framework and details the optimization based on TCUs. Section V presents our experimental results and findings. Section VI reviews the related work on tensor operations, Bitmap format, and TCUs. Finally, Section VII concludes the paper.

II. Preliminaries

A. Sparse Tensor Contraction

A tensor is a multi-dimensional array. Each of its dimensions is called a mode, and the number of modes is its order. A vector is a first-order tensor, while a second-order tensor is a matrix. A tensor of order three or higher is called a higher-order tensor. By fixing all indices but two, a slice can be created. By fixing all indices except one, a fiber is defined.

Sparse tensor contraction, also known as sparse tensor-times-tensor [6], is an extension of sparse matrix-times-matrix. For sparse tensor $\mathcal{X} \in \mathbb{R}^{S_1^{\mathcal{X}} \times S_2^{\mathcal{X}} \times \ldots S_{d_{\mathcal{X}}}^{\mathcal{X}}}$, $\mathcal{Y} \in \mathbb{R}^{S_1^{\mathcal{Y}} \times S_2^{\mathcal{Y}} \times \ldots S_{d_{\mathcal{Y}}}^{\mathcal{Y}}}$ and $\mathcal{Z} \in \mathbb{R}^{S_1^{\mathcal{Z}} \times S_2^{\mathcal{Z}} \times \ldots S_{d_{\mathcal{Z}}}^{\mathcal{Z}}}$, whose dimensions are $d_{\mathcal{X}}$, $d_{\mathcal{Y}}$ and $d_{\mathcal{Z}}$, respectively. We can represent a sparse tensor contraction as

$$\mathcal{Z}_{\prod^{\mathcal{Z}}(I_m \cup I_n)} = \alpha \times \mathcal{X}_{\prod^{\mathcal{X}}(I_m \cup I_k)} \times \mathcal{Y}_{\prod^{\mathcal{Y}}(I_k \cup I_n)} + \beta \times \mathcal{Z}_{\prod^{\mathcal{Z}}(I_m \cup I_n)}, \tag{1}$$

where $\prod^{\mathcal{X}}$, $\prod^{\mathcal{Y}}$ and $\prod^{\mathcal{Z}}$ are permutations of the symbolic index sets $I_m := \{m_1, m_2, \ldots, m_\gamma\}$, $I_n := \{n_1, n_2, \ldots, n_\zeta\}$ and $I_k := \{k_1, k_2, \ldots, k_\xi\}$. The index sets I_m, I_n respectively are the free indices of $\mathcal{X}$ and $\mathcal{Y}$ (i.e., these indices appear in $\mathcal{X}$, $\mathcal{Y}$ and $\mathcal{Z}$), and I_k is the contracted indices of $\mathcal{X}$ and $\mathcal{Y}$ (i.e., those indices appear in both $\mathcal{X}$ and $\mathcal{Y}$, but not in $\mathcal{Z}$) [16]. The dimensions of $\mathcal{X}$, $\mathcal{Y}$ and $\mathcal{Z}$ are $d_{\mathcal{X}} = \gamma + \xi$, $d_{\mathcal{Y}} = \zeta + \xi$ and $d_{\mathcal{Z}} = \gamma + \zeta$.

Utilizing the index sets I_m, I_n, and I_k makes it easy to represent an arbitrary dimension SpTC. In the following, we assume that I_m, I_n, and I_k are not empty in SpTC.

Example: For three-order tensor $\mathcal{X} \in \mathbb{R}^{S_1^{\mathcal{X}} \times S_2^{\mathcal{X}} \times S_3^{\mathcal{X}}}$ and $\mathcal{Y} \in \mathbb{R}^{S_1^{\mathcal{Y}} \times S_2^{\mathcal{Y}} \times S_3^{\mathcal{Y}}}$. The contracted indices of $\mathcal{X}$ and $\mathcal{Y}$, i.e. I_m, is $\{k_1\}$. The free indices of $\mathcal{X}$ and $\mathcal{Y}$ are $\{m_1, m_2\}$ and $\{n_1, n_2\}$, respectively. This example operation is denoted as

$$\mathcal{Z}_{\prod^{\mathcal{Z}}(m_1,m_2)\cup(n_1,n_2))} = \beta \times \mathcal{Z}_{\prod^{\mathcal{Z}}(m_1,m_2)\cup(n_1,n_2))} + \alpha \times \mathcal{X}_{\prod^{\mathcal{X}}((m_1,m_2)\cup(k_1))} \times \mathcal{Y}_{\prod^{\mathcal{Y}}((k_1)\cup(n_1,n_2))}, \tag{2}$$

where the dimension of $\mathcal{X}$, $\mathcal{Y}$ and $\mathcal{Z}$ are $d_{\mathcal{X}} = 3$, $d_{\mathcal{Y}} = 3$ and $d_{\mathcal{Z}} = 4$.

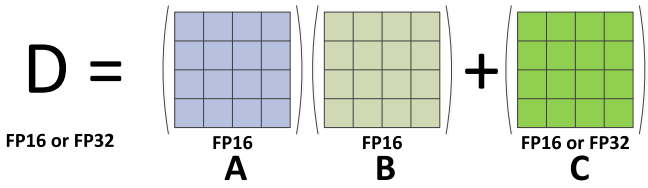


Fig. 1. Tensor Cores Units 4x4x4 matrix multiply and accumulate.

B. Tensor Cores Units

Different forms implement a floating-point computation on CUDA Cores per clock cycle. TCUs perform a matrix-matrix computation within each clock cycle [17]. To be concrete, TCUs provide a $16 \times 16 \times 16$ matrix processing array which performs the operation $D = A \times B + C$, where A, B, C and D are 16×16 matrices as Fig. 1 shows. The matrix multiply inputs A and B are FP16 matrices, while the accumulation matrices C and D may be FP16 or FP32 matrices. Before calling TCUs, all registers in a warp need to collaboratively store these matrices into a fragment (GPU register memory for storing the input matrices), which allows data sharing across registers [18]. Significantly, TCUs can only operate data from registers.

NVIDIA provides Warp-level Matrix Multiply and Accumulate (WMMA) API to program TCUs. NVIDIA introduced the WMMA API in CUDA 9.0 for developers to use TCUs. Through the WMMA API, the $D = A \times B + C$ (A, B, C, and D could be tiles of the larger matrix) can be regarded as the warp-level operation and all threads of warp can cooperate to multiply and add matrices on these tiles. A CUDA kernel that performs a matrix multiplication of two matrices with one CUDA Wrap (WMMA allows one computation with 256 elements in a wrap) can be divided into five stages. First, declaring the WMMA fragments `Afrag`, `Bfrag`, and `Cfrag`. Second, setting `Cfrag`, the accumulator fragment, for storing the result of the matrix multiply, to zero. Third, loading the input matrices into the fragments `Afrag`, `Bfrag` using `wmma::load_matrix_sync()`. Fourth, the multiplication is performed by calling the `wmma::mma_sync()`. Finally, moving the results from the fragment `Cfrag` to D in the GPU global memory.

III. BCB Format

In this section, we introduce the BCB format for storing sparse tensors. First, the BCB format is introduced in detail by describing the process of COO to BCB format. Then, the storage optimization techniques in it are given. Finally, its theoretical storage size is analyzed.

A. Format Conversion

Observing (1), SpTC essentially involves searching for the intersection of tensor spaces $\mathcal{X}, \mathcal{Y}$ with the same contracted indices I_k. We only need to search for all non-zero elements, but the search space is the entire tensor, which is too large. Therefore, we divide the entire tensor set into several subsets, i.e., blocking. The BCB structure stores the tensor in blocks, and this multi-dimensional blocking method splits the blocks by treating the sparse tensor as if it were dense. Because the high-dimensional tensor is sparse, this results in a large number of empty blocks with no non-zero elements. These empty blocks do not contribute to the SpTC results, so we only record blocks with non-zero elements.

By introducing indices of non-zero elements, the COO format avoids the memory traffic of sparse tensors with a significant number of zero elements, but at the cost of low arithmetic intensity. To improve the arithmetic intensity, further elimination of redundancy in non-zero element indices is considered. The primary goal of the BCB format is to replace the integer type with a bit type in order to reduce the amount of redundant data contained in index data. In addition to reducing memory storage, bit operations also facilitate set operations (intersection). BCB format stores non-zero elements of each block in dense form. That is, each block element can be either zero or a non-zero value. We use a bitmap, a set of binary numbers corresponding to each value in the block, to keep track of which elements have non-zero values.

Fig. 2 illustrates the conversion procedure from COO to BCB format. First, multi-dimensional blocking techniques are combined to improve the localization of data. Second, the non-zero elements in each block are stored in a bitmap to facilitate TCUs acceleration. Third, the multi-dimensional index of each block is mapped to a binary linear space to reduce the complexity of multi-dimensional matching and the number of memory instruction accesses during computation.

B. Multi-Dimensional Blocking

As shown in the first step of Fig. 2, multi-dimensional blocking splits blocks by treating a sparse tensor as a dense tensor. Multi-dimensional blocking is a popular technique used in sparse tensors to improve the efficiency of the computation process. This technique involves dividing the tensor set into several subsets of blocks, which are then separately processed. Multi-dimensional blocking is used in the BCB format to split the blocks by treating the sparse tensor as if it were dense. The high-dimensional tensor is often sparse, resulting in a large number of empty blocks with no non-zero elements. These empty blocks do not contribute to the SpTC results, so it is crucial to only record the blocks that contain non-zero elements. By dividing the tensor into blocks, each block can be processed separately, improving the efficiency of the contraction process.

Tensor contraction involves searching for non-zero elements and finding matching elements between two tensors. Multi-dimensional blocking helps to reduce the size of the search space and increase the search step, as the data is localized and can be processed more efficiently. By reducing the search space, the efficiency of the contraction process can be significantly improved.

C. Improvement of Blocking and Bit Encoding

While the concept of blocking and bit encoding has been explored in HiCOO [19], ALTO [20], and BLCO [21], our BCB

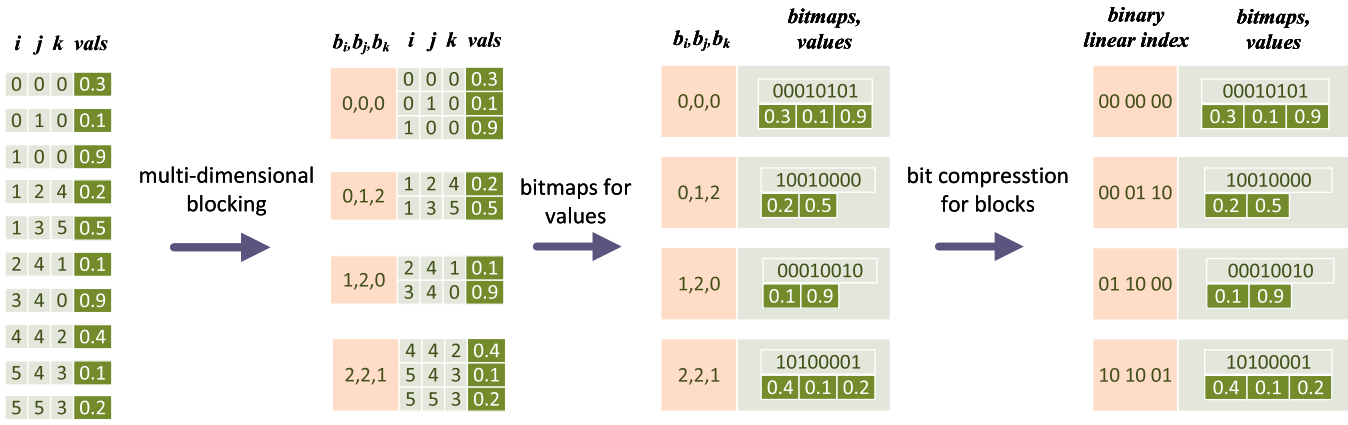


Fig. 2. Conversion process from COO to BCB format. The tensor block size is set to $2 \times 2 \times 2$, as an example indicated above. (b_i, b_j, b_k) represents the index of a block within the tensor space. After blocking, the indices of all elements within a block are mapped into an 8-bit bitmap. Additionally, (b_i, b_j, b_k) is compressed into a 6-bit binary linear index. .

format introduces key distinctions that optimize both storage and computation.

- Storage: HiCOO, ALTO, and BLCO primarily compress indices by focusing on individual non-zero elements. HiCOO uses hierarchical storage to compress relative indices within each block, while ALTO and BLCO linearize the absolute indices across the entire tensor. These approaches reduce memory usage by minimizing the storage required for each element's location. In contrast, BCB compresses indices at the level of tensor blocks rather than individual elements, utilizing bitmaps to encode the positions of non-zero elements within each block. This compact representation captures the structure of a block in a single, efficient data structure. By operating on blocks, BCB significantly reduces the overhead of storing and accessing index data, making it well-suited for handling large, sparse tensors while maintaining low memory consumption.
- Computation: The computational strategies of HiCOO, ALTO, and BLCO are element-centric, with HiCOO navigating hierarchical indices and ALTO/BLCO iterating over linearized indices to process each element. BCB, however, leverages block-level storage to perform computations in batches. Instead of sequentially processing individual elements, BCB processes entire tensor blocks at once. This batch processing is particularly efficient when using TCUs, which are optimized for parallel matrix operations. By performing block-wise computations, BCB fully exploits the parallelism of TCUs, resulting in faster tensor contractions.

D. Optimized Storage of Blocks

As shown in the second step of Fig. 2, the BCB format stores non-zero elements of each block in dense form. That is, each block element can be either zero or a non-zero value. We use a bitmap, a set of binary numbers corresponding to each value in the block, to keep track of which elements are non-zero values. If the value is greater than zero, the corresponding bit in the bitmap is set to 1, otherwise it is set to 0. This way, we can easily determine whether an element is non-zero or not by checking the corresponding bit in the bitmap. By using bitmaps to reduce the index redundancy of non-zero elements within a block, bitwise storage helps improve the arithmetic intensity of SpTC.

As shown in the third step of Fig. 2, BCB format uses bit compression to reduce the redundancy of non-empty block indices. The purpose of bit compression is to map the multi-dimensional indices of each block into a binary linear space to reduce the complexity and memory traffic of multi-dimensional matching. The bit compression method adjusts the number of bits based on the size of dimensions rather than using the same size shaping to store index values for different dimensions of the tensor. Thus, the bit-compression method converts all non-zero element indices into binary linear indices and maintains them in the smallest amount of storage space for tensors of various shapes.

E. Storage Space Analysis

For a sparse tensor $\mathcal{X} \in \mathbb{R}^{S_1 \times S_2 \times \cdots \times S_K}$ containing N non-zero elements, assume that the block $\mathcal{B}$ dimension is $H_1 \times H_2 \times \cdots \times H_K$. Therefore, the binary linear index of each block after block bit compression occupies the bit space size W_{bc} is

$$W_{bc} = \sum_{k=1}^{K} \left\lceil \log_2 \frac{S_k}{H_k} \right\rceil \quad bits. \tag{3}$$

Further, assuming that $\mathcal{X}$ has M non-empty blocks, the bit space size D_{bc} required to store all the block indices is

$$D_{bc} = W_{bc} \times M \quad bits. \tag{4}$$

Moreover, the elements within the blocks are stored in bitmap format, and the bit space size W_{bm} occupied by storing the bitmap index of each block is

$$W_{bm} = \prod_{k=1}^{K} H_k \quad bits, \tag{5}$$

Algorithm 1: BCB-SpTC.

Input:
A tensor $\mathcal{X} \in \mathbb{R}^{S_1^{\mathcal{X}} \times S_2^{\mathcal{X}} \times \cdots \times S_{d_{\mathcal{X}}}^{\mathcal{X}}}$;
A tensor $\mathcal{Y} \in \mathbb{R}^{S_1^{\mathcal{Y}} \times S_2^{\mathcal{Y}} \times \cdots \times S_{d_{\mathcal{Y}}}^{\mathcal{Y}}}$;
Output:
A tensor $\mathcal{Z} \in \mathbb{R}^{S_1^{\mathcal{Z}} \times S_2^{\mathcal{Z}} \times \cdots \times S_{d_{\mathcal{Z}}}^{\mathcal{Z}}}$;
1: /*Store two input tensors with BCB format based on multi-dimensional blocking*/
2: $\text{BCB}_{\mathcal{X}} \leftarrow \text{COO2BCB}(\mathcal{X})$;
3: $\text{BCB}_{\mathcal{Y}} \leftarrow \text{COO2BCB}(\mathcal{Y})$;
4: /*Generate the task list between input tensors for the matching relationship between blocks*/
5: $T \leftarrow \text{Task\_List}(\text{BCB}_{\mathcal{X}}, \text{BCB}_{\mathcal{Y}})$;
6: **for** each item $(\hat{\mathcal{X}}, \hat{\mathcal{Y}})$ in T **do**
7: /*Carry out the block-based tensor contraction between $\hat{\mathcal{X}}, \hat{\mathcal{Y}}$*/
8: $\hat{\mathcal{Z}} \leftarrow \text{TC\_TCUs}(\hat{\mathcal{X}}, \hat{\mathcal{Y}})$ in Algorithm 2;
9: **end for**
10: /*Remove zero values after contraction and compact the result tensor*/
11: $\text{Compact\_Zero}(\mathcal{Z})$;
12: **return** $\mathcal{Z}$;

then the bit space size D_{bm} needed to store the bitmap index of all non-empty blocks is

$$D_{bm} = W_{bm} \times M \quad bits. \tag{6}$$

Assuming that the data type of the non-zero element values is float and that each non-zero element occupies 32 bits of space, the bit space size required to store the non-zero element values is

$$D_{val} = 32 \times N \quad bits. \tag{7}$$

Summing up the above analysis, the total storage space size D_B required for tensor $\mathcal{X}$ stored in BCB format is

$$\begin{aligned} D_B &= D_{bc} + D_{bm} + D_{val} \\ &= \left(\sum_{k=1}^{K} \left\lceil \log_2 \frac{S_k}{H_k} \right\rceil + \prod_{k=1}^{K} H_k \right) \times M \\ &\quad + 32 \times N \quad bits. \end{aligned} \tag{8}$$

For COO format with integer index and float value type, the total storage space size D_{COO} required for tensor $\mathcal{X}$ with N non-zero elements is

$$D_{COO} = 32 \times K \times N + 32 \times N \quad bits \tag{9}$$

Hence, when

$$D_B < D_{COO}$$

$$\left(\sum_{k=1}^{K} \left\lceil \log_2 \frac{S_k}{H_k} \right\rceil + \prod_{k=1}^{K} H_k \right) \times M < 32 \times K \times N \tag{10}$$

holds, our BCB format will be better than COO format in storage size.

IV. BCB-SpTC Parallel Algorithm

When applying SpTC, researchers must address certain challenges, such as the multi-dimensional matching problem and the high memory overhead caused by dimensions and sparsity. In particular, SpTC may struggle to locate non-zero elements with identical indices in high-dimensional spaces due to its multi-dimensional nature.

A. Framework Design

To enhance notation convenience, we refer to our approach as BCB-SpTC, which introduces the sparse tensor contraction based on our newly proposed BCB format. Fig. 3 presents an overview of the framework, which is divided into four parts as outlined in Algorithm 1. First, we use multi-dimensional blocking to split the tensor. The BCB storage method (Section III) stores non-empty, multidimensional blocks whose index maps to a binary linear index, and non-zero elements in each block are kept in a bitmap. Next, we generate a task list to capture the matching relationship between blocks without block computation. The search space is reduced thanks to the elimination of empty blocks. After obtaining the matching block pairs in the task list, we perform block-based contraction using TCUs. Bitmaps are used to load non-zero elements in the corresponding locations of TCUs based on free and contracted indices. Finally, since tensor contraction is performed on a block-by-block basis, some resulting blocks may have the same free indices, requiring accumulation. Additionally, we allocate GPU warps to compute a pair of blocks using WMMA, with the warp scheduler scheduling another active warp for the calculation to achieve higher throughput and address load imbalance.

To contract two tensors, it is crucial to first identify the tensor blocks that can be contracted, meaning those with the same contracted indices. Inspired by the Expansion-Sorting-Compression (ESC) strategy introduced by [22] and further developed in CUSP [23], we divide the task list generation process into three stages: pairing, sorting, and compression.

In the *pairing* stage, represented by the $\text{Task\_List}(\cdot, \cdot)$ function, we identify and pair tensor blocks that share the same contracted indices. Specifically, for each output tensor block, which is represented by a binary linear index composed of two free indices, $F_{\mathcal{X}}$ and $F_{\mathcal{Y}}$, from the input tensors, we match the corresponding contracted indices, $C_{\mathcal{X}}$ and $C_{\mathcal{Y}}$. The *sorting* stage, also part of the $\text{Task\_List}(\cdot, \cdot)$ function, involves arranging the paired tensor blocks based on the memory required for contraction. This step ensures that multi-dimensional blocks from the input tensors, identified by the pairs $(F_{\mathcal{X}}, C_{\mathcal{X}})$ and $(F_{\mathcal{Y}}, C_{\mathcal{Y}})$, where $C_{\mathcal{X}}$ equals $C_{\mathcal{Y}}$, are placed adjacently. This adjacency is crucial for optimizing memory access patterns during contraction. The *compression* stage compresses the sorted tuples by their free indices $(F_{\mathcal{X}}, F_{\mathcal{Y}})$ after finishing tensor contraction by $\text{TC\_TCUs}(\cdot, \cdot)$ (Algorithm 2), as represented in $\text{Compact\_Zero}(\cdot)$ function. Due to the adjacency achieved by the sorting stage. We only need to sum up all the contracted blocks with the same free indices.

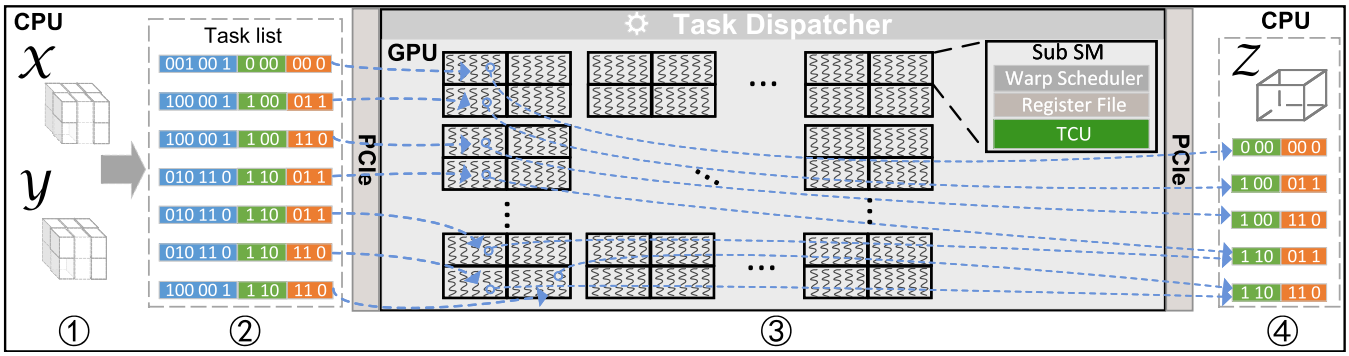


Fig. 3. Overview of our proposed BCB-SpTC parallel computing framework. The BCB-SpTC framework consists of four stages: ① Multi-dimensional blocking, ② Generation of the task list, ③ Contraction between blocks, and ④ Write-back of the result tensor.

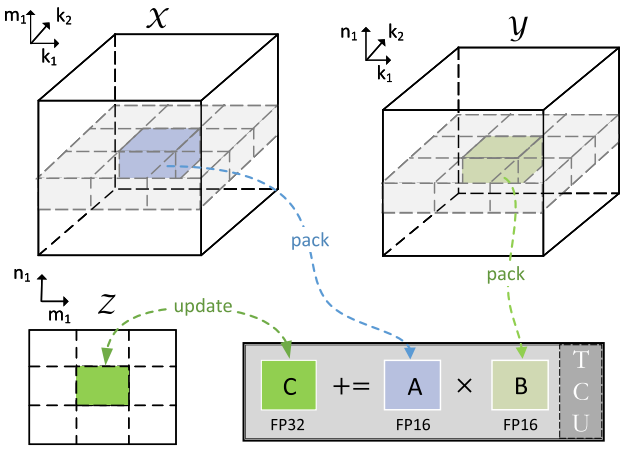


Fig. 4. Multi-dimensional blocks contraction with TCUs. Each block is packed into a TCU fragment. After computation, the result will update the contracted block.

Algorithm 2: TC_TCUs: WMMA Operations for the 1-Mode Contraction in GPU Kernel Function.

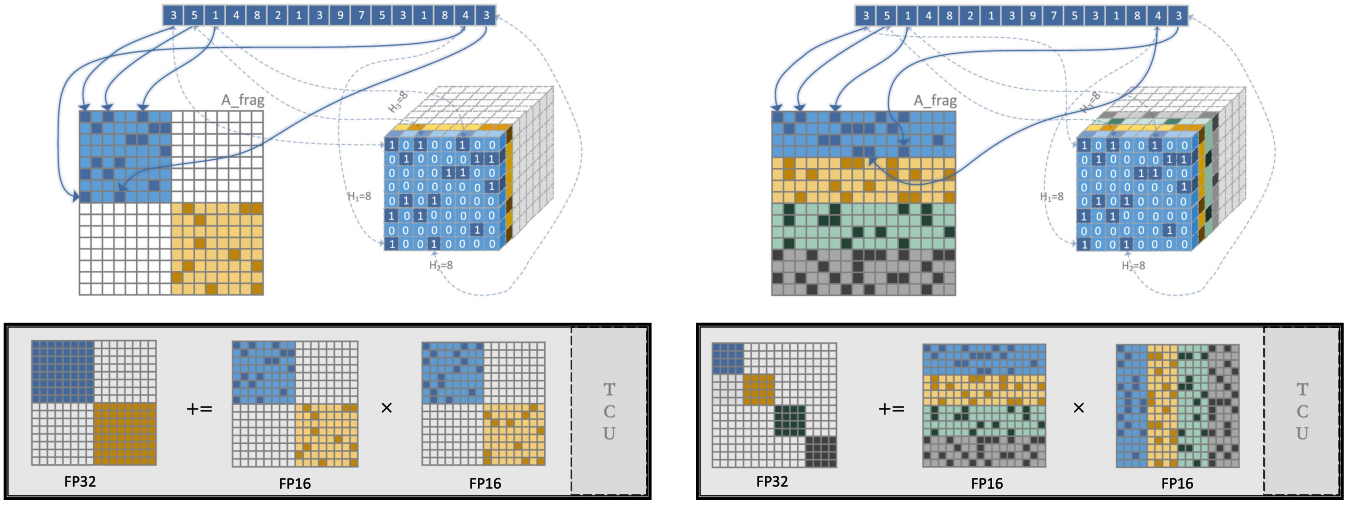
Input:
Element array: $\mathcal{X}\_ele, \mathcal{Y}\_ele$;
Task list arrays: $task\_list\_ptr, task\_list$;
Output:
Result array: $\mathcal{Z}\_ele$;
Bitmap index array: bmp_3
int $WID \leftarrow threadIdx.x/32$;
wmma::fragment A_frag, B_frag, C_frag;
int $t_1 \leftarrow task\_list\_ptr[WID]$;
int $t_2 \leftarrow task\_list\_ptr[WID+1]$;
$bmp_3 \leftarrow 0$;
for $i = t_1$ **to** t_2 **do**
  $(bmp_1, bmp_2) \leftarrow task\_list(i)$;
  $slice\_num \leftarrow 2$;
  for $s = 0$ **to** 8 **do**
    Load2Frag($s, slice\_num, bmp_1, \mathcal{X}\_ele, A\_frag$);
    Load2Frag($s, slice\_num, bmp_2, \mathcal{Y}\_ele, B\_frag$);
    wmma::mma_sync($C\_frag, A\_frag, B\_frag, C\_frag$);
    Load2Bmp($C\_frag, bmp_3, \mathcal{Z}\_ele$);
    $s+ = slice\_num$;
  end for
end for
return $bmp_3, \mathcal{Z}\_ele$;

B. Loading Scheme for TCUs

TCUs are components on GPUs used for matrix computations $D = A \times B + C$, and how to utilize them for tensor contraction is a matter that requires careful consideration. We design our method to take advantage of their parallelism for block-wise sparse tensor contraction. By mapping sparse data into block structures, we are able to exploit the computational power of TCUs, which significantly reduces the time required for contraction operations. The blockwise tensor contraction is illustrated in Fig. 4. After the TCU computation is completed, a new task is retrieved from the task list to execute tensor contraction, and this process is repeated until all items in the task list have been traversed.

The loading scheme for tensor contraction on TCUs depends on free indices and contracted indices. The free index of tensor $\mathcal{X}$ is treated as the row index of matrix A, and the free index of tensor $\mathcal{Y}$ is treated as the column index of matrix B. Bitmaps are employed to find the loading position for non-zero elements in TCUs.

The APIs (WMMA) provided by NVIDIA only provide one calculation with 256 elements in a warp. We choose the unsigned long long integer for bitmaps, which is equal to 64 elements being computed (8×8) at once, resulting in not exploiting the full performance of matrix multiplication of TCUs at once. Regardless, WMMA directly loads elements from global memory into registers of each thread containing a fragment of the matrix, where the partial matrix is loaded and stored. Fig. 5 illustrates the loading method for a tensor block (size $8 \times 8 \times 8$) in `Afrag` of TCUs under 1-mode and 2-mode contraction. Light colors represent bits set to 0in the bitmap, while dark colors represent bits set to 1. For `Bfrag` of TCUs, the loading method is similar. To provide a clear illustration, we mainly depict the first slice of the tensor block and the positions of non-zero element



(a) The contraction of one mode. (b) The contraction of two modes.

Fig. 5. Loading methods for blocks. The white spaces in TCUs represent the initial state of the fragment, where all positions are zeroed out and not yet loaded with data. The light-colored blocks correspond to elements where the bitmap's bit is set to 0, meaning these are structural zeros in the tensor. The dark-colored blocks indicate non-zero elements, where the bitmap's bit is set to 1, representing the active data that will be used in the tensor contraction process.

Algorithm 3: Load2Frag: Load Two 8×8 Slices Into a 16×16 Fragment Matrix Using 64-Bit Bitmaps.

Input: $slice\_idx, slice\_num, bmp, ele\_ptr$
Output: $frag$
Initialize $frag$ as a 16×16 matrix filled with zeros
for $i = 0$ **to** 7 **do**
  for $j = 0$ **to** 7 **do**
    /*Calculate the bit position in the bitmap*/
    $bit\_pos \leftarrow i \times 8 + j$
    if $(bmp[s]\&(1 \ll bit\_pos)) \neq 0$ **then**
      $frag[i][j] \leftarrow *ele\_ptr\ ++$
    end if
  end for
end for
for $i = 8$ **to** 15 **do**
  for $j = 8$ **to** 15 **do**
    /*Calculate the bit position in the bitmap*/
    $bit\_pos \leftarrow (i - 8) \times 8 + (j - 8)$
    if $(bmp[s+1]\&(1 \ll bit\_pos)) \neq 0$ **then**
      $frag[i][j] \leftarrow *ele\_ptr\ ++$
    end if
  end for
end for
return $frag$

Algorithm 4: Load2Bmp: Store Bitmaps From the non-Zero Elements of Two 8×8 Diagonal Submatrices Within a 16×16 Fragment.

Input: $frag, bmp, ele\_ptr$
Output: bmp, ele_ptr
$bit\_idx \leftarrow 0$
$bmp_0 \leftarrow 0, bmp_1 \leftarrow 0$
for $i = 0$ **to** 7 **do**
  for $j = 0$ **to** 7 **do**
    /*Set the corresponding bit in bmp to 1*/
    if $frag[i][j] \neq 0$ **then**
      $bit\_idx \leftarrow i \times 8 + j$
      $bmp_0 \leftarrow bmp_0 | (1 \ll bit\_idx)$
      $*ele\_ptr\ ++ \leftarrow frag[i][j]$
    end if
  end for
end for
for $i = 8$ **to** 15 **do**
  for $j = 8$ **to** 15 **do**
    /*Set the corresponding bit in bmp to 1*/
    if $frag[i][j] \neq 0$ **then**
      $bit\_idx \leftarrow (i - 8) \times 8 + (j - 8)$
      $bmp_1 \leftarrow bmp_1 | (1 \ll bit\_idx)$
      $*ele\_ptr\ ++ \leftarrow frag[i][j]$
    end if
  end for
end for
bmp.push(bmp_0, bmp_1)
return bmp

loadings at the beginning and end of the slice and execute, and we perform the sparse tensor self-contraction operation for a tensor block unit $\mathcal{B} \in \mathbb{R}^{8\times8\times8}$, namely $\mathcal{Z}_{\prod^{\mathcal{Z}}(I_m \cup I_m)} = \mathcal{B}_{\prod^{\mathcal{B}}(I_m \cup I_k)} \times \mathcal{B}_{\prod^{\mathcal{B}}(I_k \cup I_m)}$.

In 1-mode contraction, for each element in the result tensor $\mathcal{Z}$, we have

$$\mathcal{Z}(h_1, h_3, h_3, h_1) = \mathcal{B}(h_1, h_3, :) \times \mathcal{B}(:, h_3, h_1), \tag{11}$$

where $I_m = \{H_1, H_3\}$ and $I_k = \{H_2\}$. Then, we extend it to the slice form,

$$\mathcal{Z}(h_1, :, :, h_1) = \mathcal{B}(h_1, :, :) \times \mathcal{B}(:, :, h_1). \tag{12}$$

As depicted in Fig. 11(a), achieving 1-mode contraction on H_2, merely requires placing each slice into the diagonal matrices of the fragment. Remarkably, the contraction of two slices can be accomplished by executing a single WMMA operation. For an $8 \times 8 \times 8$ block, four WMMA operations are required. We extract solely the diagonal matrices from the resulting matrix to produce the 1-mode contraction results.

In Fig. 11(b), we adopt a similar approach to slicing and fragment placement. For 2-mode contraction, we have

$$\mathcal{Z}(h_3, h_3) = \sum_{h_1=1}^{8} \sum_{h_2=1}^{8} \mathcal{B}(h_3, :, :) \times \mathcal{B}(:, :, h_3). \tag{13}$$

where $I_m = \{H_1, H_2\}$ and $I_k = \{H_3\}$.

However, for 2-mode contractions, we upscale our original 8×8 slice into a 4×16 one. One WMMA operation suffices for the contraction of four slices, and hence, for an $8 \times 8 \times 8$ block, two WMMA operations are necessary. It is noteworthy that after computing the 2-mode contraction with TCUs, we must sum up the four small matrices located only on the diagonal of the resulting matrix separately. Consequently, we obtain just four contraction results from one WMMA operation.

Algorithm 2 illustrates the execution of GPU threads for 1-mode tensor block contraction using WMMA operations. Within Algorithm 2, the variable *WID* denotes the warp to which each thread belongs. Each warp iterates through matched pairs contributing to a single result block based on the information from array $task\_list\_ptr$ in lines 6 to 16. The variable bmp_3 represents the result block in BCB format. A matched pair (bmp_1, bmp_2) is obtained in line 7. During computation, each warp reads non-zero elements from $slice\_num$ slices and stores them in registers, specifically A_frag and B_frag, utilizing the index information outlined in lines 10 and 11, as detailed in Algorithm 3. Algorithm 3 aims at fill the values in $frag$ with the corresponding position by the bit-AND operation. Subsequently, C_frag is produced by performing matrix-matrix multiplication between A_frag and B_frag in line 12 with the wmma built-in function. Two diagonal submatrices in C_frag are stored in line 13, as detailed in Algorithm 4. Algorithm 4 store two diagonal submatrices of the wmma result matrix which are transformed into bitmaps with bit-OR operation. After processing all matched pairs assigned to this warp, the result block is transferred to the result tensor $\mathcal{Z}$.

The procedure for 2-mode contraction exhibits similarities to that of 1-mode contraction. However, there are two distinctions in the functions Load2Frag and Load2Bmp. Fig. 11(b) illustrates the process of 2-mode contraction, where both A_frag and B_frag are simultaneously loaded with 4 slices to participate in TCUs (lines 10-11). Here, the variable $slice\_num$ is assigned a value of 4. Additionally, the extraction of four diagonal matrices is performed, followed by the summation of all elements within each matrix. Consequently, the sum of elements in each diagonal submatrix is treated as an individual element within the resulting tensor in line 13. Therefore, for the contraction of a four-dimensional tensor and so forth, the process can be extrapolated by simply modifying the loading method.

V. Evaluation

In this section, we conduct a number of experiments on GPU to evaluate the performance of our proposed approach. First, we analyze the storage space size of the BCB format and evaluate its effectiveness in improving the performance of SpTC computation. Second, we verify the efficient utilization of TCUs under different downsizing modes by our proposed TCU kernel loading method. Third, by comparing the current SpTC implementations, we demonstrate the superiority and advancement of BCB-SpTC in terms of overall performance.

A. Experimental Setup

1) Platforms: The GPU used for the experiments is the NVIDIA A100 GPU (A100 for short) of the Ampere architecture, which is installed on an Intel(R) Xeon(R) Gold 5120 CPU. The CPU memory size used for the experiments is 128 GB, and the GPU global memory size is 40 GB, which has 6912 cuda cores and 432 tensor cores. We use GCC 7.5.0 and NVCC 11.2.

2) Datasets: We evaluate our method on four sparse tensors that are obtained from the real-world datasets with varying characteristics. These four tensor datasets are from GroupLens Research,[1] Chengdu Taxi data[2] and vertical excitation energies in uracil [24]. The details are as follows.

- *Ratings & Art:* Published by GroupLens Research. They described the anonymous movie and art ratings from members of the movie recommendation service MovieLens. The dimension size of *Ratings* is $6040 \times 3952 \times 5$ and the number of its records is 1000209. The dimension size of *Art* is $669 \times 379 \times 251$ and the number of its records is 63064.
- *Taxi:* The traffic flow data collected frome Chengdu City area in August 2014. It obtained over 1.4 billion GPS trajectory data of more than 14000 taxis. The dimension size of *Ratings* is $7471 \times 1080 \times 28$ and the number of its records is 688758.
- *Uracil:* The chemistry data came from Vertical excitation energies in uracil in the gas phase and in the water solution. It joined in the perturbative triples correction (T) in Coupled Cluster (CC) methods. The dimension size of *Uracil* is $90 \times 90 \times 174 \times 174$ and the number of its records is 1034755.

3) Parameter Settings: Tensor contraction is a computationally challenging process due to the huge dimension of tensors. To facilitate the observation of the performance of BCB-SpTC framework, we perform a tensor contraction of $\mathcal{Z}_{\prod^z(I_m \cup I_m)} = \mathcal{X}_{\prod^x(I_m \cup I_k)} \times \mathcal{X}_{\prod^x(I_k \cup I_m)}$ to generate two tensors with the same sparse structure. For the 3-dimensional tensor, we consider the computational task of the contraction of one mode and the computational task of the contraction of two modes. For the 4-dimensional tensor, we also consider the computational task of the contraction of three modes. Limited by the device memory, we always contract the longer indices. To better demonstrate the progress of our method, we chose other recent research

[1] https://grouplens.org

[2] https://challenge.datacastle.cn/v3/cmptDetail.html?id=175

TABLE I
MEMORY USAGE (MB) OF DATASETS

Tensors	BCB-SpTC	tSparse	GSpTC	DBCSR
Ratings	5.14	9.13	11.45	7.89
Art	0.12	0.21	0.74	0.83
Taxi	2.90	5.16	7.88	5.59
Uracil	0.82	1.46	142.11	81.72

results (tSparse [25], DBCSR [26], GSpTC [15]) as the baseline methods. These methods are both executed on GPU and only tSparse utilizes TCUs. It should be noted that we have extended the tSparse, originally used for computing sparse matrix multiplication, to handle sparse tensor contractions. Both BCB-SpTC and tSparse utilize bitmap-based storage for non-zero elements and employ block partitioning strategies with TCUs. However, a pivotal divergence arises concerning the structure of block units and the approach to indexing. BCB-SpTC adopts tensors as block units and employs a binary linear indexing scheme to store block information. In contrast, tSparse, while extended to tensor operations, retains matrices as block units and records block indices using the COO format. Furthermore, we continue to utilize tSparse for representation.

B. Efficiency of BCB Format

1) Storage Usage Analysis: The memory usage data for four datasets (Ratings, Art, Taxi, and Uracil) under four methods are summarized in Table I. As shown in Table I, the BCB-SpTC method consistently outperforms the other three methods in terms of storage usage. Across all datasets, BCB-SpTC exhibits lower memory consumption by an average of 1.75 MB, 33.80 MB, and 21.76 MB compared to tSparse, GSpTC, and DBCSR, respectively. These results demonstrate the superior storage efficiency of the BCB-SpTC method for SpTC on GPU. Although the relative improvement in storage usage with BCB-SpTC is more modest for the Ratings, Art, and Taxi datasets (maximum 6.31 MB, 0.71 MB, and 4.98 MB reductions, respectively), a significant advantage is observed for the Uracil dataset. BCB-SpTC achieves substantial memory savings, reducing storage consumption by 141.29 MB and 80.90 MB compared to GSpTC and DBCSR, respectively. This outcome underscores the particular strength of the BCB-SpTC method for handling large-scale datasets, offering potential benefits for memory-constrained computing environments. In BCB format, the utilization of block partitioning reduces memory overhead and facilitates processing of large sparse tensors by breaking them into manageable blocks. And the bitmap format for index storage efficiently represents non-zero elements, resulting in compact memory usage. Moreover, the adoption of binary linear indexing simplifies data access within each tensor block, further optimizing memory usage.

Overall, the results indicate that BCB-SpTC outperforms the other methods in terms of storage efficiency, making it a promising approach for SpTC on GPU, particularly when handling large-scale datasets.

2) Time Analysis by Stage: The operation steps of the various SpTC algorithms are different, but the execution time can be uniformly divided into three parts according to the computation phase. They are the pre-computation phase (task processing), the computation phase (computation), and the post-computation phase (output processing). In BCB-SPTC, we refer to the generation of the task list as task processing, contraction between blocks as computation, and write-back as output processing.

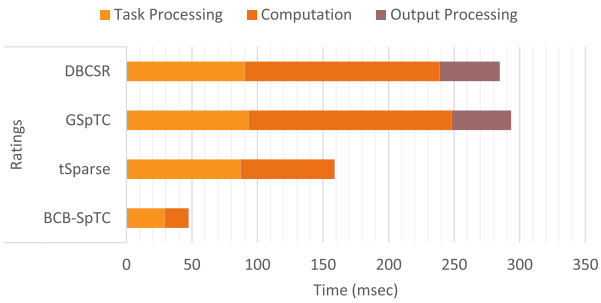
Fig. 6 shows the execution time of 2-mode contraction for four methods on the four datasets. The execution time is presented in three parts. On the Ratings and Taxi datasets, the execution time of BCB-SPTC is much less than the other methods. BCB-SPTC is up to 6.19 times faster than the other methods and 5.18 times faster on average on Ratings. Moreover, BCB-SPTC is up to 2.54 times faster than the other methods and 2.37 times faster on average on Taxi. On the Uracil dataset, BCB-SPTC is significantly faster than DBCSR and GSpTC, and the execution time of BCB-SpTC is only half of tSparse. On Art, since the dataset is relatively sparse and has many sparse non-zero elements, BCB-SPTC does not have a large advantage over other methods on the execution time. However, it is still better than other methods. Overall, the results reveal that BCB-SPTC outperforms the other methods in terms of execution time.

For each part of the execution time, the BCB-SPTC has a short time of computation than other methods over all datasets. Furthermore, the task processing time of BCB-SPTC is less than other methods on Ratings, Taxi, and Uracil. BCB-SPTC has a longer task processing time since the Art dataset is relatively sparse, and there are a lot of sparse non-zero elements in it. But the time of this part of BCB-SPTC is still less than tSparse. For the time of output processing, BCB-SPTC outperforms DBCSR and GSpTC over all datasets. The time of this part for BCB-SPTC is pretty much the same as tSparse. It is worth mentioning that the time of output for BCB-SPTC is slightly faster than that of tSparse. Overall, except for the time of task processing on the Art dataset, BCB-SPTC outperforms the other methods in time at each stage across all datasets.

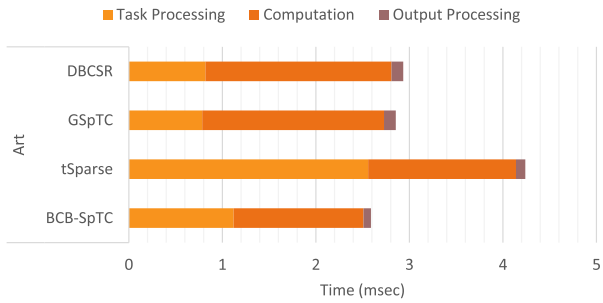
In conclusion, the experience results indicate that BCB-SPTC outperforms the other methods in terms of execution time, and the time for each part of BCB-SPTC is better than the other methods in most cases on all datasets, particularly when handling large-scale datasets and relatively dense datasets.

C. Utilization of Tensor Cores

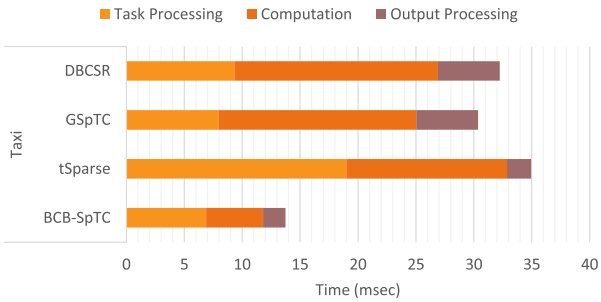
1) Computation Time Analysis: To highlight the efficiency of BCB-SpTC in utilizing TCUs, we compare its computation time with tSparse, as both methods leverage TCUs for sparse tensor computations. TCUs use half-precision (FP16) calculations, which can introduce numerical errors compared to the single-precision (FP32) computations typically used by CUDA cores. This makes a comparison with tSparse crucial for evaluating both performance and accuracy in the TCU context. Since tSparse is the only baseline method designed to utilize TCUs, other methods relying on CUDA cores would not provide a meaningful comparison. The results are shown in Fig. 7. For all 1-mode contraction, it can be seen that the difference between the computation time of BCB-SpTC and tSparse is almost negligible, which is due to the fact that BCB-SpTC,



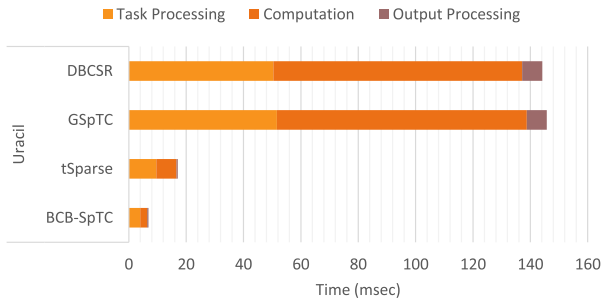
(a) 2-mode contraction on Ratings dataset.



(b) 2-mode contraction on Art dataset.



(c) 2-mode contraction on Taxi dataset.



(d) 2-mode contraction on Uracil dataset.

Fig. 6. Execution time of each part on A100.

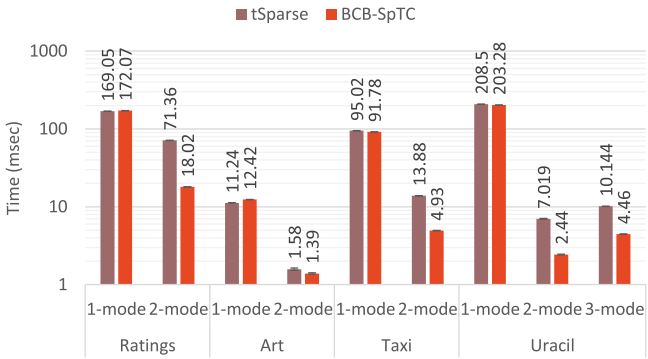


Fig. 7. Computation Time Comparison with TCUs.

which performs 1-mode contraction, and tSparse have the same method of data populating the TCUs, i.e., they both compute two slice at a time. For 2-mode and 3-mode contraction, it can be seen that BCB-SpTC has a larger performance improvement relative to tSparse. This is because tSparse still uses two slice computation at a time to fill the TCUs, while BCB-SpTC uses four slice computation at a time to fill the TCUs in response to the characteristics of the 2-mode and 3-mode computations. Specifically, BCB-SpTC achieves an average speedup ratio of 2.70× relative to tSparse when performing 2-mode contraction. In the case of performing a 3-mode contraction, the average speedup ratio is 2.27×. The reason for the slightly lower performance gain of 3-mode reduction over 2-mode reduction is that the workload for 3-mode loading is higher, which slightly reduces the utilization of the TCUs.

2) Error Analysis: Since the acceleration of computation using TCUs is to use FP16 as the type of input data and FP32 as the type of output data, this can lead to numerical errors in the computation process. Therefore, in order to verify the effectiveness of BCB-SpTC using TCU, we compare the error of the computation results of BCB-SpTC and tSparse with the results of the computation using FP32 accuracy on CPU, where the error R is calculated as follows:

$$R = \sqrt{\frac{1}{|\Omega|}\sum_{i\in\Omega}\left(\mathcal{Z}_{FP32}(i) - \mathcal{Z}_{FP16}(i)\right)^2}$$

$$|\Omega| = \prod_{j=1}^{d_{\mathcal{Z}}} S_j^{\mathcal{Z}} \tag{14}$$

where $\mathcal{Z}_{FP32}$ is the result of use FP32, and $\mathcal{Z}_{FP16}$ is the result with FP16. Fig. 8 illustrates the numerical errors in the calculations of BCB-SpTC and tSparse. 1-mode contraction of BCB-SpTC and tSparse have the same method of data populating the TCUs, making their errors similar. For the 2-mode reduction, the errors are larger than 1-mode and smaller than 3-mode, due to the fact that as the number of contracted modes increases, so does the error generated by the computation using TCU. Overall, the numerical errors are below 1.5% for both tSparse and BCB-SpTC.

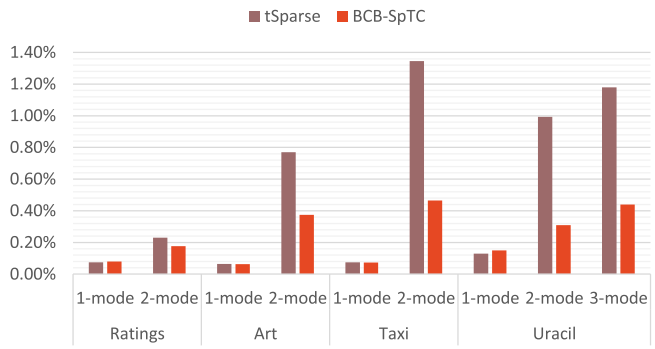


Fig. 8. Numerical Error Comparison with TCUs.

D. Comparison to Other Implementations

1) Performance on Matrix Groups: To evaluate the performance of BCB-SpTC on synthetic data, we created tensors by combining several matrices from the SuiteSparse matrix collection.[3] Specifically, we generated two tensors: one from the oscil_dcop matrix series with dimensions $(430 \times 430 \times 57)$, containing 88,008 non-zero elements, and another from the fpga_dcop matrix series with dimensions $(1220 \times 1220 \times 51)$, containing 300,359 non-zero elements. We performed tensor contractions in both mode-1 and mode-2 operations on an A100 GPU. The analysis covered execution time and the distribution of time across different computational phases, highlighting the efficiency of BCB-SpTC in handling these synthetic tensors. Fig. 10 shows the performance of BCB-SpTC compared to other methods for both mode-1 and mode-2 tensor contractions on the synthetic data. BCB-SpTC demonstrates competitive execution times, largely due to the spatial locality present in the synthetic tensors. The organization of non-zero elements in these tensors, derived from grouped matrices, aligns well with BCB-SpTC's multi-dimensional blocking strategy, enabling efficient handling of localized data. Fig. 11 show the time distribution for different phases–task processing, computation, and output processing–during mode-1 and mode-2 contractions. BCB-SpTC shows a balanced distribution of time across all phases, particularly excelling during the computation phase due to its block-wise strategy, which is well-suited to the synthetic data's structure.

The strong performance of BCB-SpTC on synthetic data is likely due to the spatial locality of non-zero elements within the tensors derived from grouped matrices. This structure allows BCB-SpTC to optimize data access and minimize redundant computations, leading to faster execution times and more efficient utilization of GPU resources.

2) Performance on Real Tensors: We provide a comparison between BCB-SpTC and three alternative methods. To further evaluate the scalability and robustness of our BCB-SpTC method, we extended our experiments to include two significantly larger and more sparse tensors: the Nips and Uber tensors [27]. The Nips tensor has dimensions of $2482 \times 2862 \times 14036 \times 17$ with 3101609 non-zero elements, while the Uber tensor has dimensions of $183 \times 24 \times 1140 \times 1717$ with 3309490 non-zero elements. Fig. 9 illustrates the performance of sparse tensor contraction using six methods on A100 GPU. Each of the comparative methods focuses on achieving sparse tensor contraction on GPU across six diverse datasets, encompassing 1-mode, 2-mode, and 3-mode tensor contraction scenarios. It is noteworthy to mention that tSparse, initially designed for sparse matrix multiplication, is extended to accommodate tensor contractions by us. The results indicate that the time required for tensor contraction increases with the dimensionality and size of the tensor. Despite this increase, BCB-SpTC maintains its competitive edge in performance, demonstrating effective scaling across these more challenging datasets. In 1-mode sparse contraction, BCB-SpTC consistently outperforms tSparse, yielding speedup ratios ranging from approximately $1.4\times$ to $2.2\times$. In 2-mode contraction, BCB-SpTC maintains its competitive edge, with speedup ratios between $1.6\times$ and $3.3\times$, further emphasizing its advantage over tSparse. These findings underscore BCB-SpTC's efficiency in the context of tensor contraction. And this performance discrepancy can be attributed to BCB-SpTC's efficient TCUs utilization and optimized indexing strategy.

When comparing BCB-SpTC to GSpTC and DBCSR, a clear trend emerges in favor of BCB-SpTC's execution times. GSpTC leverages COO format for sparse tensor contractions' parallel optimization, while DBCSR employs a similar parallel optimization strategy for sparse tensor contractions using the DBCSR framework. In contrast, BCB-SpTC excels in memory utilization and computational speed by utilizing tensor-based block units and binary linear indexing. The empirical results in Fig. 9 consistently portray BCB-SpTC's superior execution time. As an example, in the 1-mode contraction of the Taxi dataset, BCB-SpTC takes approximately 5.8 $\times$ less time than GSpTC (827.6 milliseconds) and around 5.6 $\times$ less time than DBCSR (806.9 milliseconds) to complete the task. This performance advantage is rooted in BCB-SpTC's effective strategy of BCB format and TCUs, resulting in improved data access and computation efficiency. For instance, with the addition of the Uber and Nips tensors, the increased computational demand due to their larger dimensions and lower sparsity further highlights BCB-SpTC's capability to handle large-scale tensor contractions efficiently.

In conclusion, our comparison analysis establishes BCB-SpTC as a powerful and memory-efficient solution for executing sparse tensor contractions on GPU. Its unique approach of utilizing TCUs and BCB format grants BCB-SpTC a distinct edge in terms of computational speed and memory efficiency. Overall, the results in Fig. 9 indicate that BCB-SpTC has a significant advantage over tSparse, GSpTC, and DBCSR in terms of computational speed and efficiency, with an average speedup ratio of $4.6\times$ and speedup ratios ranging from $1.1\times$ to $21.3\times$.

VI. Related Work

In the realm of scientific and engineering applications, emerges as a computational task that poses significant challenges. To tackle these challenges and optimize the performance of algorithms, researchers have delved into various methods.

[3]https://sparse.tamu.edu/

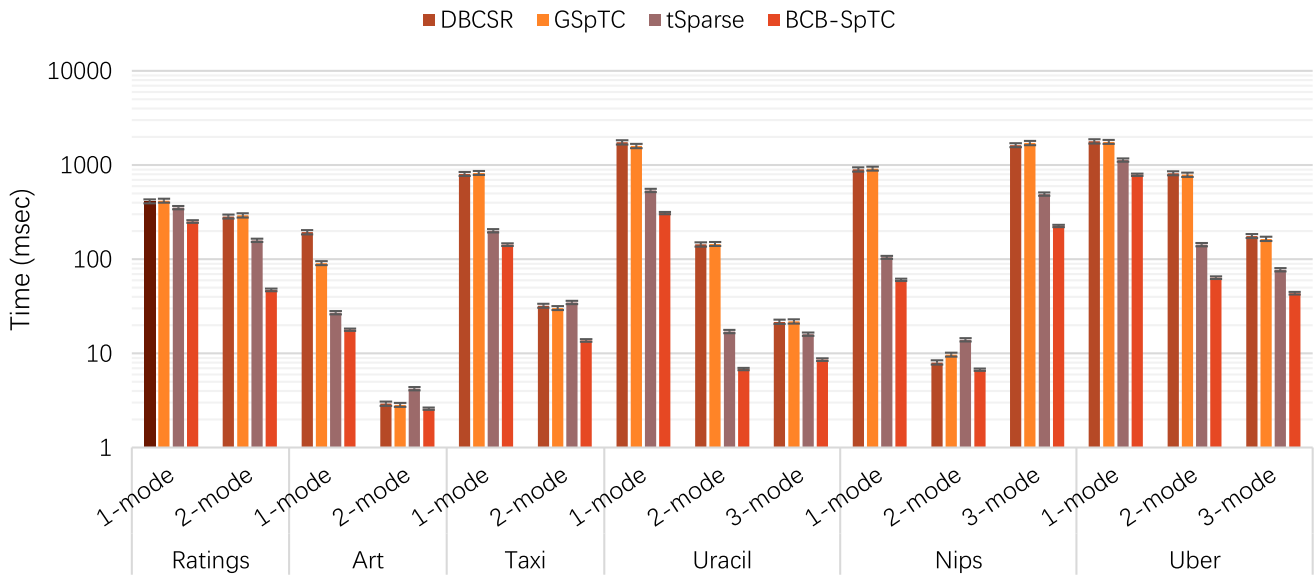


Fig. 9. SpTC performance of different methods on A100.

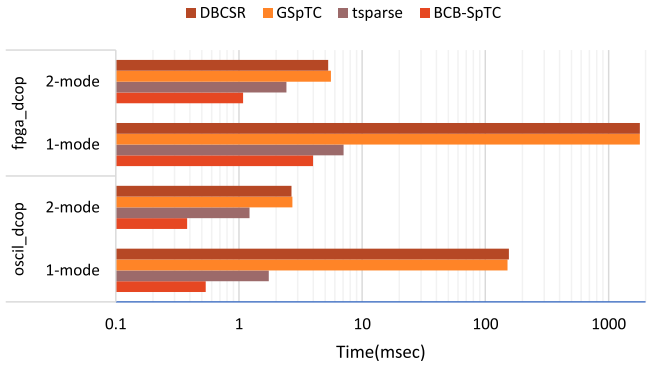
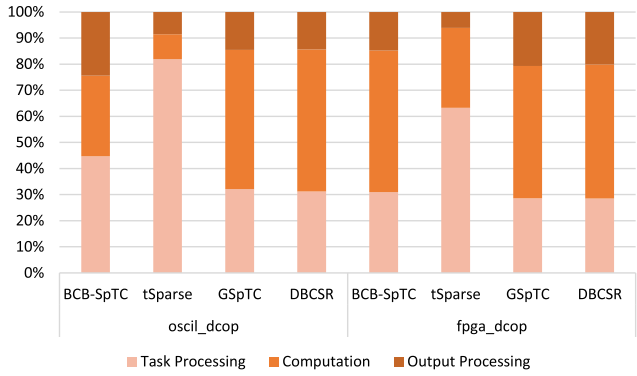
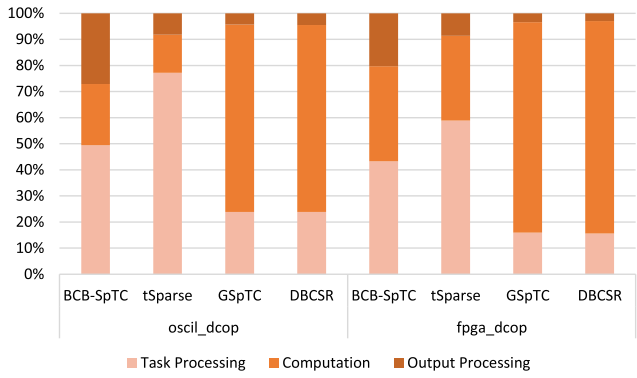


Fig. 10. SpTC performance of different methods about synthetic tensors on A100.



(a) The contraction of one mode.



(b) The contraction of two modes.

Fig. 11. Time Distribution for each part about synthetic tensors on A100.

The primary goal of optimizing SpTC is to leverage the sparsity property. This objective has led to numerous approaches proposed in recent years. Ozog et al. [28] investigated scheduling algorithms for block-sparse tensor contractions within the NWChem computational chemistry code, considering different degrees of sparsity. Liu et al. [9] developed Sparta, which employed novel data representations and structures to address specific challenges facing the optimization of SpTC. Hérault et al. [29] implemented tensor contraction efficiently and scalably using the task-focused (PaRSEC) runtime. Xiao et al. [15] presented GSpTC, an efficient element-wise SpTC framework on CPU-GPU heterogeneous system, which addressed the problem of index matching and accumulation. However, these works lacked the exploration of SpTC acceleration using TCU which is new generation hardware.

Bitmap-based formats have emerged as a promising solution for improving the optimization of performance. Various researchers have proposed these formats to overcome the challenge of leveraging sparsity effectively. Wang et al. [30] implemented Sparse Tensor-Times-Matrix (SpTTM) on CPU-GPU heterogeneous hybrid systems, which gave a parallel execution strategy for SpTTM in different sparse formats and designed a new graph neural network SPT-GCN to select a

suitable tensor sparse format. Zhang et al. [31] developed data parallel algorithms to pair up bitmap-indexed sparse matrix blocks for SpGEMM utilizing data parallel primitives. Kannan et al. [32] proposed a bitmapped sparse matrix format that stores entries as blocks without incurring load overheads. Chan et al. [33] presented a comprehensive framework for studying the design space of bitmap indices for selection queries. Wang et al. [34] proposed IAP-SpTV, an input-aware adaptive pipeline SpTV via Graph Convolutional Network (GCN) on CPU-GPU. They designed the hybrid tensor format (HTF) and constructed Slice-GCN to select a suitable format for each slice of HTF. A lot of research has focused on monolithic compression formats to fine-grained blocked compression formats. The tiled and blocked compression formats have become the research hotspot.

Recently, researchers have started exploring the potential of TCUs to enhance the optimization of. TCUs are specialized hardware components that accelerate tensor operations on NVIDIA GPUs. Zachariadi et al. [25] designed tSparse, which employs a bitmap-based format to store the matrix, partitions the input matrices into tiles, and multiplies tiles using the mixed precision mode of TCUs to perform SpGEMM. Feng et al. [35] developed Emulated GEMM on Tensor Cores (EGEMM-TC) to extend the usage of Tensor Cores to accelerate scientific computing applications without compromising the precision requirements. Wang et al. [36] proposed a novel approach to GPU-based Sparse Tensor Matrix Chain Multiplication (SpTMCM) and explored the discovery of SpTMCM coupled with the emerging computing core, TCUs. It developed a TCU-based tensor parallel algorithm with a novel approach to increase the memory bandwidth. The current exploration of TCUs accelerated calculations mainly focuses on matrix computation which is a low-dimensional computation, and lacks exploration of high-dimensional data contraction computation.

VII. Conclusion

We propose BCB-SpTC, a parallel computing framework based on bit compression and bitmaps on GPU. BCB-SpTC blocks the entire tensor in multiple dimensions and stores the sparse distribution in a bitmap to minimize index storage. Multiple indices of blocks are replaced with binary indices to achieve multi-dimensional index matching for block matching. For block computation, the element indices are mapped into the TCUs for tensor contraction using bitmaps. Experiments on real-world tensor datasets demonstrated that BCB-SpTC outperformed conventional tensor contraction methods.

For future work, we will continue to improve the computational efficiency of bit operations and focus on the design pattern of the TCUs from PTX-level optimization. For extremely sparse tensors, the bitmaps will contain more non-zero elements by appropriate data transformation and transposition.

References

[1] S. Hirata, "Tensor contraction engine: Abstraction and automated parallel implementation of configuration-interaction, coupled-cluster, and many-body perturbation theories," *J. Phys. Chem. A*, vol. 107, pp. 9887–9897, 2003.

[2] J. Gray and S. Kourtis, "Hyper-optimized tensor network contraction," *Quantum*, vol. 5, 2021, Art. no. 410.

[3] J. Kim et al., "Optimizing tensor contractions in CCSD(T) for efficient execution on GPUs," in *Proc. Int. Conf. Supercomputing*, 2018, pp. 96–106.

[4] C. Roberts et al., "TensorNetwork: A library for physics and machine learning," 2019, *arXiv: 1905.01330*.

[5] J. Kossaifi, A. Khanna, Z. C. Lipton, T. Furlanello, and A. Anandkumar, "Tensor contraction layers for parsimonious deep nets," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. Workshops*, 2017, pp. 1940–1946.

[6] A. Cichocki, "Era of Big Data processing: A new approach via tensor networks and tensor decompositions," 2014, *arXiv:1403.2048*.

[7] J. Kossaifi, Z. C. Lipton, A. Kolbeinsson, A. Khanna, T. Furlanello, and A. Anandkumar, "Tensor regression networks," *J. Mach. Learn. Res.*, vol. 21, no. 1, pp. 4862–4882, 2020.

[8] R. Kunchum, A. Chaudhry, A. Sukumaran-Rajam, Q. Niu, I. Nisa, and P. Sadayappan, "On improving performance of sparse matrix-matrix multiplication on GPUs," in *Proc. Int. Conf. Supercomputing*, 2017, pp. 14:1–14:11.

[9] J. Liu, J. Ren, R. Gioiosa, D. Li, and J. Li, "Sparta: High-performance, element-wise sparse tensor contraction on heterogeneous memory," in *Proc. ACM Sigplan Symp. Princ. Pract. Parallel Program.*, 2021, pp. 318–333.

[10] S. Smith and G. Karypis, "Tensor-matrix products with a compressed sparse tensor," in *Proc. 5th Workshop Irregular Appl.: Architectures Algorithms*, 2015, pp. 1–7.

[11] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[12] M. T. Fishman, S. R. White, and E. M. Stoudenmire, "The ITensor software library for tensor network calculations," 2020, *arXiv: 2007.14822*.

[13] D. A. Matthews, "High-performance tensor contraction without transposition," *SIAM J. Sci. Comput.*, vol. 40, 2018, Art. no. C1–C24.

[14] P. J. Martínez-Ferrer, A. N. Yzelman, and V. Beltran, "A native tensor–vector multiplication algorithm for high performance computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 12, pp. 3363–3374, Dec. 2022.

[15] G. Xiao, C. Yin, Y. Chen, M. Duan, and K. Li, "GSpTC: High-performance sparse tensor contraction on CPU-GPU heterogeneous systems," in *Proc. IEEE 24th Int. Conf. High Perform. Comput. Commun.; 8th Int. Conf. Data Sci. Syst.; 20th Int. Conf. Smart City; 8th Int. Conf. Dependability Sensor, Cloud Big Data Syst. Appl.*, 2022, pp. 380–387.

[16] P. Springer and P. Bientinesi, "Design of a high-performance GEMM-like tensor–tensor multiplication," *ACM Trans. Math. Softw.*, vol. 44, no. 3, pp. 1–29, 2018.

[17] S. Markidis, S. W. D. Chien, E. Laure, I. B. Peng, and J. S. Vetter, "Nvidia tensor core programmability, performance & precision," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops*, 2018, pp. 522–531.

[18] Z. Jia, M. Maggioni, J. K. Smith, and D. P. Scarpazza, "Dissecting the NVidia turing T4 GPU via microbenchmarking," 2019, *arXiv: 1903.07486*.

[19] J. Li, J. Sun, and R. Vuduc, "HiCOO: Hierarchical storage of sparse tensors," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2018, pp. 238–252.

[20] A. E. Helal et al., "ALTO: Adaptive linearized storage of sparse tensors," in *Proc. ACM Int. Conf. Supercomputing*, 2021, pp. 404–416.

[21] A. Nguyen et al., "Efficient, out-of-memory sparse MTTKRP on massively parallel architectures," in *Proc. 36th ACM Int. Conf. Supercomputing*, 2022, pp. 1–13.

[22] N. Bell, S. Dalton, and L. N. Olson, "Exposing fine-grained parallelism in algebraic multigrid methods," *SIAM J. Sci. Comput.*, vol. 34, pp. C123–C152, 2012.

[23] S. Dalton, N. Bell, L. Olson, and M. Garland, "CUSP: Generic parallel algorithms for sparse matrix and graph computations," 2014. [Online]. Available: http://cusplibrary.github.io/

[24] E. Epifanovsky, K. Kowalski, P.-D. Fan, M. Valiev, S. Matsika, and A. I. Krylov, "On the electronically excited states of uracil," *J. Phys. Chemistry. A*, vol. 112, no. 40, pp. 9983–9992, 2008.

[25] O. Zachariadis, N. Satpute, J. Gómez-Luna, and J. Olivares, "Accelerating sparse matrix-matrix multiplication with gpu tensor cores," *Comput. Elect. Eng.*, vol. 88, 2020, Art. no. 106848.

[26] I. Sivkov, P. Seewald, A. Lazzaro, and J. Hutter, "DBCSR: A blocked sparse tensor algebra library," in *Proc. Int. Conf. Parallel Comput.*, 2019, pp. 331–340. [Online]. Available: https://api.semanticscholar.org/CorpusID:204972719

[27] S. Smith et al., FROSTT: The formidable repository of open sparse tensors and tools, 2017. [Online]. Available: http://frostt.io/
[28] D. Ozog, J. R. Hammond, J. Dinan, P. Balaji, S. Shende, and A. D. Malony, "Inspector-executor load balancing algorithms for block-sparse tensor contractions," in *Proc. 42nd Int. Conf. Parallel Process.*, 2013, pp. 30–39.
[29] T. Hérault et al., "Distributed-memory multi-GPU block-sparse tensor contraction for electronic structure," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2021, pp. 537–546.
[30] H. Wang, W. Yang, R. Ouyang, R. Hu, K. Li, and K. Li, "A heterogeneous parallel computing approach optimizing SpTTM on CPU-GPU via GCN," *ACM Trans. Parallel Comput.*, vol. 10, pp. 1–23, 2023.
[31] J. Zhang and L. Gruenwald, "Regularizing irregularity: Bitmap-based and portable sparse matrix multiplication for graph data on GPUs," in *Proc. Int. Conf. Manage. Data*, 2018, pp. 1–8.
[32] R. Kannan, "Efficient sparse matrix multiple-vector multiplication using a bitmapped format," in *Proc. IEEE Int. Conf. High Perform. Comput., Data, Analytics*, 2013, pp. 286–294.
[33] C. Y. Chan and Y. E. Ioannidis, "Bitmap index design and evaluation," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 1998, pp. 355–366.
[34] H. Wang, W. Yang, R. Hu, R. Ouyang, K. Li, and K.-C. Li, "IAP-SpTV: An input-aware adaptive pipeline SpTV via GCN on CPU-GPU," *J. Parallel Distrib. Comput.*, vol. 181, 2023, Art. no. 104741.
[35] B. Feng, Y. Wang, G. Chen, W. Zhang, Y. Xie, and Y. Ding, "EGEMM-TC: Accelerating scientific computing on tensor cores with extended precision," in *Proc. 26th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2021, pp. 278–291.
[36] H. Wang, W. Yang, R. Hu, R. Ouyang, K. Li, and K.-C. Li, "A novel parallel algorithm for sparse tensor matrix chain multiplication via TCU-acceleration," *IEEE Trans. Parallel Distrib. Syst.*, vol. 34, no. 8, pp. 2419–2432, Aug. 2023.

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

Haotian Wang received the PhD degree in computer science from Hunan University, China, in 2023. He is currently working as a postdoctoral fellow with Hunan University, China. He previously completed a one year joint PhD program from Nanyang Technological University, and he is an ACM member. His research interests include parallel computing, tensor compilation, and artificial intelligence.

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.

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.

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 five 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).

Kenli Li (Senior Member, IEEE) received the PhD degree in computer science from the Huazhong University of Science and Technology, China, in 2003., and the MS degree in mathematics from Central South University, China, in 2000. 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. The main research fields are 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 serving on the editorial board of IEEE TRANSACTIONS ON COMPUTERS. He is an outstanding member of CCF.