A Heterogeneous Parallel Computing Approach Optimizing SpTTM on CPU-GPU via GCN

HAOTIAN WANG, WANGDONG YANG, RENQIU OUYANG, RONG HU, and KENLI LI, College of Computer Science and Electronic Engineering, Hunan University, China
KEQIN LI, Department of Computer Science, State University of New York, USA and College of Computer Science and Electronic Engineering, Hunan University, China

Sparse Tensor-Times-Matrix (SpTTM) is the core calculation in tensor analysis. The sparse distributions of different tensors vary greatly, which poses a big challenge to designing efficient and general SpTTM. In this paper, we describe SpTTM on CPU-GPU heterogeneous hybrid systems and give a parallel execution strategy for SpTTM in different sparse formats. We analyze the theoretical computer powers and estimate the number of tasks to achieve the load balancing between the CPU and the GPU of the heterogeneous systems. We discuss a method to describe tensor sparse structure by graph structure and design a new graph neural network SPT-GCN to select a suitable tensor sparse format. Furthermore, we perform extensive experiments using real datasets to demonstrate the advantages and efficiency of our proposed input-aware slice-wise SpTTM. The experimental results show that our input-aware slice-wise SpTTM can achieve an average speedup of 1.310× compared to ParTI! library on a CPU-GPU heterogeneous system.

CCS Concepts: • **Computing methodologies** → **Parallel algorithms**; Artificial intelligence; • **Mathematics of computing** → *Mathematical software performance;*

Additional Key Words and Phrases: CPU-GPU heterogeneous systems, format selection, GCN, parallel computing, SpTTM

ACM Reference format:
Haotian Wang, Wangdong Yang, Renqiu Ouyang, Rong Hu, Kenli Li, and Keqin Li. 2023. A Heterogeneous Parallel Computing Approach Optimizing SpTTM on CPU-GPU via GCN. *ACM Trans. Parallel Comput.* 10, 2, Article 9 (June 2023), 23 pages.
https://doi.org/10.1145/3584373

The research was partially funded by the National Key R&D Program of China (Grant No. 2020YFB2104000), the Key Program of National Natural Science Foundation of China (Grant No. U21A20461), the National Natural Science Foundation of China (Grant Nos. 61872127 and 61751204), the Research Innovation Project for Postgraduate Students of Hunan Province (No. CX20220412), and GHFUND A (No. ghfund202107013482).
Authors' addresses: H. Wang, W. Yang (corresponding author), R. Ouyang, R. Hu, and K. Li, College of Computer Science and Electronic Engineering, Hunan University, 116 Lu Shan South Road, Changsha, Hunan, 410082, China; emails: {wanghaotian, yangwangdong, rqouyang, upupwords, lkl}@hnu.edu.cn; K. Li, Department of Computer Science, State University of New York,1 Hawk Drive, New Paltz, New York, 12561, USA and College of Computer Science and Electronic Engineering, Hunan University,116 Lu Shan South Road, Changsha, Hunan, 410082, China; email: lik@newpaltz.edu.

2329-4949/2023/06-ART9 $15.00
https://doi.org/10.1145/3584373

1 INTRODUCTION

Sparse Tensor-Times-Matrix (SpTTM) is an indispensable core computation in various practical application scenarios of tensor analysis, e.g., healthcare analytics, signal processing, neuroscience, and machine learning [1, 16, 19, 20]. SpTTM is also a momentous computing bottleneck in numerous tensor algorithms, such as tensor decomposition [23, 30]. Therefore, it is urgent and crucial to design an efficient method for SpTTM algorithm. An efficient SpTTM algorithm can accelerate the common tensor decomposition algorithms, such as Tucker decomposition [17, 18, 39].

Different sparse tensors have different non-zero element distributions and sparsity structures in practical applications, which affect the performance of SpTTM. Figure 1 illustrates the performance of SpTTM using different sparse formats for two 3-*order* tensor datasets. It reveals that the optimal sparse formats are different for different tensors, and the optimal sparse formats are different for different orders of the same tensor. Therefore, identifying the sparse structures of the non-zero element distributions for different data sets becomes the key issue in improving the computing efficiency of different tensor data sets. Finding appropriate computation formats for different tensors can effectively improve the performance of tensor computation. Nonetheless, most current works directly and artificially select the storage formats for tensors, which may result in the storage formats are not suitable for tensors and affecting the performance of tensor computation.

The rapid development of artificial intelligence has opened up new ideas for the format selection of sparse tensors. Although there are additional costs associated with adopting machine learning and deep learning approaches, these overheads can be amortized by multiple iterations of SpTTM. The traditional machine learning method, e.g., **support vector machines (SVM)**, selects the sparse format of the tensor according to its statistical characteristics [6, 35]. Utilizing the deep learning methods, such as **Convolution Neural Networks (CNN)**, to select the sparse format of a tensor, requires firstly compressing the tensor into a fixed size matrix [43, 54]. The above methods all lead to the loss of some sparsely distributed features of the tensor, and the loss would become larger with the increase of the size of the tensor.

In recent years, certain computing tasks using the **graphics processing unit (GPU)** to accelerate research have become popular, e.g., using GPUs to accelerate computing tasks [2, 9, 31]. SpTTM is a computing task that can be accelerated using GPUs. Heterogeneous systems usually contain two or more interconnected processors, and each processor has a different computational speed. In heterogeneous computing systems equipped with both CPUs and GPUs, previous tensor computation libraries generally use only CPUs or GPUs, which will result in some computing resources being the idle and inadequate performance of machine hardware.

To solve these problems, in this paper, we

- describe SpTTM on CPU-GPU heterogeneous systems and give a parallel execution strategy for SpTTM in different sparse formats.
- design the task mapping strategy for load balancing by analyzing the peak computer power to estimate the theoretical amount of tasks for each of CPU and GPU.
- construct the matrix sparse format selection model SPT-GCN to select the appropriate sparse format for tensor execution of SpTTM using the graph convolution network method.
- perform extensive experiments using real datasets, and for choosing a sparse format for the tensor, SPT-GCN achieves an average accuracy of 0.851 on the CPU and 0.901 on the GPU. And our input-aware slice-wise SpTTM can achieve an average speedup of 1.310× compared to ParTI! library [22] on a CPU-GPU heterogeneous system.

The rest of the paper is organized as follows. Section 2 reviews the related work on optimization for tensor computing and sparse format selection. Section 3 presents the method of SpTTM

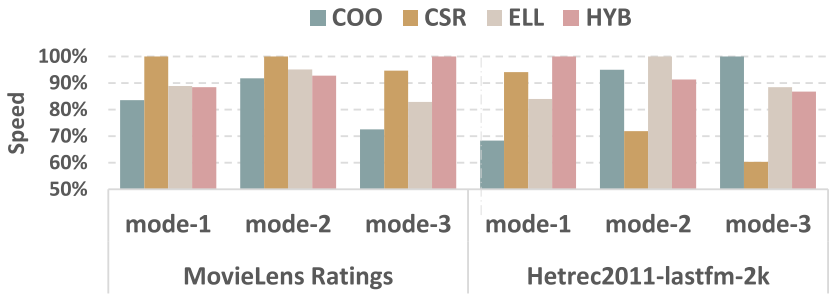


Fig. 1. SpTTM performance for different formats of tensors.

with the definition of various sparse tensor formats and introduces the CPU-GPU heterogeneous programming. Section 4 introduces the overall workflow of our slice-wise SpTTM and heterogeneous parallel pattern. Section 5 presents the task mapping strategy for load balancing between CPU and GPU. Section 6 introduces SPT-GCN, a graph neural network model for tensor selection in the best sparse format. Section 7 presents our experimental results and findings. Finally, Section 8 concludes the paper.

2 RELATED WORK

The development of deep learning brings new challenges to the study of high-performance tensor computation. In this section, we outline the research work related to (1) optimization for tensor computing and (2) sparse format selection.

2.1 Optimization for Tensor Computing

Designing efficient tensor computing is worth exploring as tensors are widely used. Li et al. [23] constructed a way to optimize SpTTM on multicore and multicore architectures. Ma et al. [30] designed an optimized way to compute SpTTM on GPU based on the characteristics of GPU. Liu et al. [27] implemented a unified optimization method for sparse tensor operations on GPUs by exploiting the fact that sparse tensor operations share similar computation patterns. Li et al. [21] demonstrated an adaptive dense TTM approach that can select a heuristic empirical model for the optimal configuration of the inputs to the TTM. Pawłowski et al. [37] suggested a new data structure in which the tensor is blocked and the blocks are stored in an order determined by Morton-ordered. Kaya and Uçar [17] contributed an efficient parallelization method for Tucker decomposition and provided intuitively shared memory parallelism for TTM and TRSVD steps. Bassoy [5] achieved a high-performance algorithm design for mode-q tensor-vector multiplication using an efficient implementation of **matrix-vector multiplication (GEMV)**. Smith and Karypis [40] proposed the **compressed sparse fiber (CSF)** and designed a method to tile over the sparse tensor to avoid locking during parallel computation. More, the latest tensor computation optimization studies are **Matricized Tensor Times Khatri-Rao Product (MTTKRP)** [4, 24, 29], **Tensor-Times-Vector (TTV)** [3, 10, 51], **Tensor Contraction (TC)** [11, 28, 42], and so on.

2.2 Sparse Format Selection

The design and selection of sparse formats is a classical research direction in the field of high-performance parallel algorithms, and the wide application of deep learning brings new opportunities. Nisa et al. [34] developed a new storage-efficient representation for tensors called HB-CSF that enables high-performance, load-balanced execution of MTTKRP on GPUs and implemented sparse MTTKRP using the new sparse tensor representation. Nisa et al. [33] presented a mixed-mode

tensor representation that partitions the tensor's non-zero elements into disjoint sections, each of which is compressed to create fibers along a different mode. Xie et al. [50] developed a deep learning model for **SpGEMM (Sparse matrix-matrix multiplication)** called MatNet for automatically determining the optimal format and algorithm for arbitrary sparse matrices. Sun et al. [43, 44] designed SpTFS for MTTKRP, a framework for automatically predicting the optimal storage format for the input sparse tensor. Zhao et al. [54] systematically explored the prospects and special challenges of deep learning for sparse matrix format selection for **SpMV (Sparse matrix-vector multiplication)**. Tan et al. [45] developed an **automatic tuning system (SMAT)** that provided programmers with a uniform interface to **compressed sparse row (CSR)** sparse matrix formats by implicitly selecting the best format and fastest implementation of any input sparse matrix at runtime; SMAT uses machine learning models and a relocatable backend library to quickly predict the best combination. Benatia et al. [6] used a multiclass SVM classifier to select the best format for each input matrix on the GPU. Li et al. [26] proposed a flexible modal Tucker decomposition algorithm that implemented the switching of factor matrix and core tensor solvers, and used a machine learning adaptive solver selector to automatically handle changes in input data and hardware. Niu et al. [36] exploited the two-dimensional spatial structure of sparse matrices to optimize SpMV on GPU and devised a selection method to find the best format and SpMV implementation for each block. Sedaghati et al. [38] investigated the interrelationship between GPU architecture, sparse matrix representation, and sparse datasets and established a decision model using machine learning to automatically select the optimal sparse matrix representation on a given target system based on the characteristics of the sparse matrix. Dai et al. [12] proposed a data-aware GPU-based heuristic kernel, DA-SpMM, and implemented an input dynamic adaptive optimization code.

3 PRELIMINARIES

This section will introduce the knowledge of tensor computation with the example of SpTTM, including the background knowledge of the tensor operation, sparse format, the calculation process of SpTTM, and the CPU-GPU hybrid parallel programming. Table 1 shows the definitions of symbols used in this paper.

3.1 Tensor

A tensor can be seen as a multidimensional array. Each of its dimensions is called a *mode*, and the *order* of it is the number of dimensions, a.k.a. ways or modes. A vector as a first-*order* tensor, is denoted as boldface lowercase letter, e.g., x, and a matrix as a second-*order* tensor, is denoted by boldface capital letter, e.g., X. A tensor of *order* three or higher is called higher *order* tensor, is denoted as bold capital calligraphic letter, e.g., $\mathcal{X}$. We use $\mathcal{X}_{ijk}$ to represent the element of the third-*order* tensor $\mathcal{X}$ position (i, j, k).

Unfolding, a.k.a. matricization or flattening, reorders the elements of a tensor and converts this tensor to a matrix. For example, a third-*order* tensor $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$ can be converted to a matrix $X_{(1)} \in \mathbb{R}^{I \times JK}$, or a matrix $X_{(2)} \in \mathbb{R}^{J \times IK}$, or a matrix $X_{(3)} \in \mathbb{R}^{K \times IJ}$.

Given a tensor, we can get many sub-tensors which may be matrices and vectors. To be concrete, for a third-*order* tensor $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$, by fixing the indicators of mode J and mode K, a vector can be obtained, called a fiber, symbolized as $V_{:jk} = \mathcal{X}(:, j, k)$. By fixing the indicator of mode K in the third-order tensor, a matrix can be obtained, which is called a slice, symbolized as $X_{::k} = \mathcal{X}(:, :, k)$.

3.2 Sparse Tensor-Times-Matrix

Tensor-Times-Matrix (TTM) in mode n, also called the n-mode (matrix) product, is the product of a tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times \cdots \times I_n \times \cdots \times I_N}$ with a matrix $U \in \mathbb{R}^{R \times I_n}$ in mode n, denoted by $\mathcal{Y} = \mathcal{X} \times_n U$. This

Table 1. Table of Symbols

Symbol	Definition
x	A vector.
X, U, Y	A matrix.
$\mathcal{X}, \mathcal{Y}$	A third-*order* or higher-*order* tensor.
$X_{::k}$	The frontal slices of the third-*order* tensor $\mathcal{X}$.
$V_{:jk}$	The model-1 fiber of the third-*order* tensor $\mathcal{X}$.
$X_{(1)}$	The model-1 unfolding of the third-*order* tensor $\mathcal{X}$.
Num_c, Num_g	The number of cores in CPU or GPU.
F_c, F_g	The main frequency of CPU or GPU.
FMA_c, FMA_g	The number of floating point operations of CPU or GPU.
P_c, P_g	Peak computer power of CPU or GPU.
Q_c, Q_g	Amount of computing tasks assigned to CPU or GPU.
M	Amount of data to be computed or amount of accessed memory.
M_c, M_g	Amount of accessed memory for tasks assigned to CPU or GPU.
μ	The coefficient between computing tasks and accessed memory.
t_{cp}, t_{gp}	Calculation time on CPU or GPU.
t_{cm}, t_{gm}	Memory accessing time on CPU or GPU.
t_r	The time to copy data from the CPU to the GPU.
B	The bandwidth of the data from the CPU to the GPU.
λ	The percentage of peak computing in actual calculations.
G	A graph.
A	An adjacency matrix.
u, v	A node of the graph G.
e	An edge of the graph G.
d	Features of the nodes.
R, C, V	The set of nodes.
E	The set of edges.
$N(v)$	The neighborhood of node v.
$f(v)$	The feature vector of node v.
σ	Activation function.
$\mathbf{W}_1, \mathbf{W}_2$	Parameter matrices.

results in an $I_1 \times \cdots \times I_{n-1} \times R \times I_{n+1} \times \cdots \times I_N$ tensor, and its operation is defined as

$$\mathcal{Y}_{i_1 \cdots i_{n-1} r i_{n+1} \cdots i_N} = \sum_{i_n=1}^{I_n} \mathcal{X}_{i_1 \cdots i_{n-1} i_n i_{n+1} \cdots i_N} \times U_{r, i_n}. \tag{1}$$

When $\mathcal{X}$ is sparse and U is dense, the operation is called SpTTM. Since there are already many mature commercial libraries implementing high-performance matrix computation, such as MKL library [46] and CUSPARSE library [49], the common approach in practical applications is that the sparse tensor is first divided into multiple slices, and then each slice calls the matrix computation interface in the commercial libraries.

As an example, let the 1-mode unfoldings of the tensor $\mathcal{X} \in \mathbb{R}^{3\times4\times2}$ be

$$X_{(1)} = \left(\begin{array}{cccc|cccc} 1 & 4 & 7 & 10 & 13 & 16 & 19 & 22 \\ 2 & 5 & 8 & 11 & 14 & 17 & 20 & 23 \\ 3 & 6 & 9 & 12 & 15 & 18 & 21 & 24 \end{array}\right), \tag{2}$$

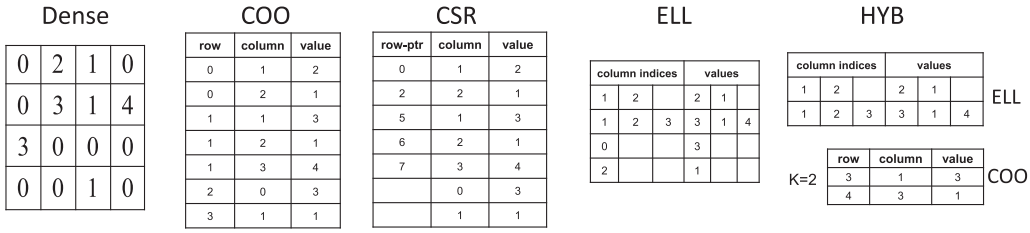


Fig. 2. Different sparse formats.

where $X_{(1)} \in \mathbb{R}^{3\times 8}$ is the result of tensor $\mathcal{X} \in \mathbb{R}^{3\times 4\times 2}$ 1-mode unfolding. And the frontal slices of $\mathcal{X} \in \mathbb{R}^{3\times 4\times 2}$ be

$$X_{::1} = \begin{pmatrix} 1 & 4 & 7 & 10 \\ 2 & 5 & 8 & 11 \\ 3 & 6 & 9 & 12 \end{pmatrix}, X_{::2} = \begin{pmatrix} 13 & 16 & 19 & 22 \\ 14 & 17 & 20 & 23 \\ 15 & 18 & 21 & 24 \end{pmatrix}. \tag{3}$$

The 1-mode product of the tensor $\mathcal{X} \in \mathbb{R}^{3\times 4\times 2}$ with the matrix $U \in \mathbb{R}^{3\times 5}$ can be equated to the matrix $U^T \in \mathbb{R}^{5\times 3}$ multiplied by two frontal slices $X_{::k} \in \mathbb{R}^{3\times 4}$, respectively, or the matrix $U^T \in \mathbb{R}^{5\times 3}$ multiplied by four lateral slices $X_{:j:} \in \mathbb{R}^{3\times 2}$, respectively.

3.3 Sparse Format

If the matrix has few zero elements, using dense formats and optimization methods such as chunking and merging access can effectively improve the parallel computation efficiency. Nevertheless, if the matrix has numerous zero elements, the dense format causes memory waste for storing zero values and loss of computer power for multiplying zero values, so sparse matrices need to use a suitable sparse format to help solve these problems.

The exploration of sparse formats for matrices has a long history of research, and thus many sparse formats have been proposed. Except for the dense format, we consider four popular formats COO, CSR, ELL, and HYB, as shown in Figure 2.

(1) COO (coordinate format) directly stores the index and value of non-zero elements and uses a list of (row, column, value) tuples to store the matrix.
(2) CSR (compressed sparse row format) uses row compression technology to be widely used. It implements parallel computing by dividing data by rows.
(3) ELL (ELLPACK) uses a list of (data, index) tuples to store matrices. Data arrays store the non-zero elements of each row. The integer array index stores the column index of each non-zero element.
(4) HYB (Hybrid ELL/COO) uses a hybrid ELL and COO format to classify dense and sparse rows according to a threshold K, and uses ELL format for dense rows and COO format for sparse rows.

3.4 CPU–GPU Hybrid Parallel Programming

With the mushroom growth of multi-core technology, the number of cores in CPU has been increasing. CPU with 8 cores, 32 cores or even higher, enter the domain of general computing, which greatly improves the parallel computing power of CPU [47]. Although the FLOPS of GPU is much higher than CPU, it does not have process control power in CUDA, which is controlled by CPU [25]. Utilizing CPU and GPU to build a heterogeneous programming environment can effectively make up for the shortcomings of CPU and GPU. In this model, the data is transferred from CPU to GPU via PCIe bus, and then the CPU schedules the calculation process of GPU by calling kernel

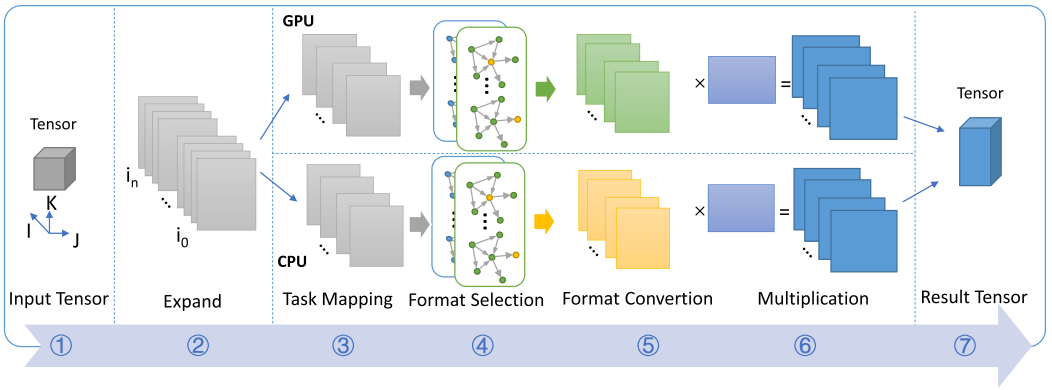


Fig. 3. The workflow of the input-aware slice-wise SpTTM.

functions. Using OpenMP to realize parallel processes on CPU, which splits one thread to control the GPU and other threads are used to share the workload between the remaining CPU cores. The data need is divided into two parts and allocated to CPU and GPU. Then, two groups of threads in OpenMP are created, where one thread controls the GPU and other threads take on the workload on CPU.

4 INPUT-AWARE SLICE-WISE SPTTM ON CPU-GPU

To efficiently utilize the computational power of CPU-GPU heterogeneous systems to enhance the performance of SpTTM, we design and implement input-aware slice-wise SpTTM. In the rest of this section, we describe its overall workflow and the heterogeneous parallel computing pattern.

4.1 Overall Workflow

In this subsection, we introduce the details of our proposed input-aware slice-wise SpTTM. The overall workflow consists of seven steps, as illustrated in Figure 3.

— **Step 1: Input Tensor.** The input data is organized into tensor.
— **Step 2: Expand.** The tensor is expanded into a series of slices by mode I.
— **Step 3: Task Mapping.** SpTTM computational tasks are load-balancedly mapped on heterogeneous devices.
— **Step 4: Format Selection.** SPT-GCN is used to select the appropriate sparse format for the tensor on the CPU and GPU.
— **Step 5: Format Conversion.** The tensor is converted to the format selected by SPT-GCN.
— **Step 6: Multiplication.** The input-aware slice-wise SpTTM is executed on a CPU-GPU heterogeneous system.
— **Step 7: Result Tensor.** The results on CPU and GPU are reorganized into tensor by modal I.

4.2 CPU-GPU Heterogeneous Parallel Computing Pattern

In this subsection, we outline the computing flow of the input-aware slice-wise SpTTM and describe the parallel implementation of SpTTM with different sparse formats.

4.2.1 Parallel SpTTM Algorithm. In the CPU-GPU heterogeneous parallel method, the most common general model is to invoke the calculation process of GPU by calling the kernel function. In this case, CPU waits until the kernel finishes, which means that the parallelism is lying idle. For increasing CPU utilization, we employ OpenMP to combine with CUDA in our heterogeneous

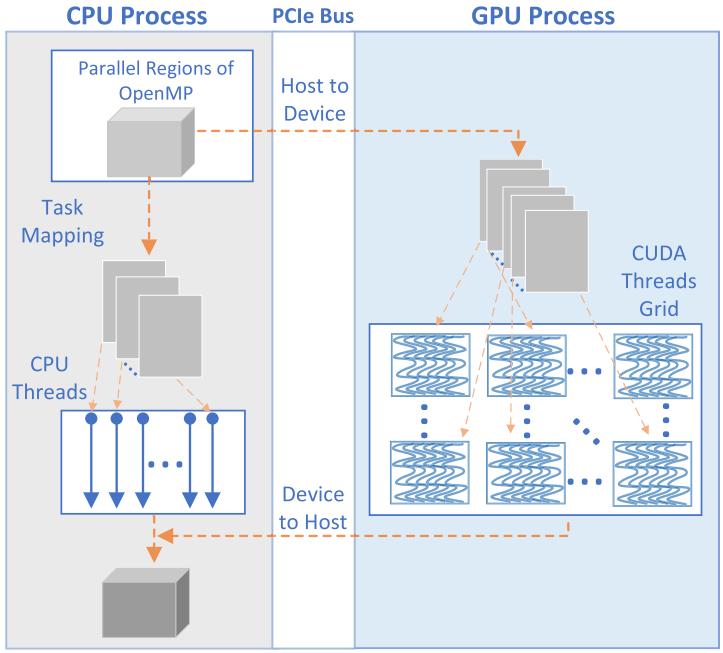


Fig. 4. The CPU–GPU heterogeneous parallel computing pattern.

parallel model. One thread in CPU is assigned to control the GPU, and others execute programs in parallel. As shown in Figure 4, the tensor initially is split into two slices set by task mapping which will be detailed in Section 5. The larger one is allocated to the GPU because of the greater parallelism, and another set is executed in parallel at the CPU. Then, two groups of threads are created on the CPU, one of which controls the kernel function in the CUDA threads Grid of the GPU, while the rest of the threads take on the CPU parallel workload. Each of the two systems, CPU and GPU, selects its own most appropriate storage format based by SPT-GCN and we describe in detail how each sparse format is parallelized in SpTTM on the CPU and GPU in Section 4.2.2. When these calculations are completed, the result slices are all gathered to the CPU.

The implementation for SpMV uses OpenMP to achieve parallelism on a CPU–GPU heterogeneous computing system, as shown in Algorithm 1. The execution process on CPU-GPU for slice-wise SpTTM includes four steps.

(1) The sparse tensor is split into two slice sets: $S_{CPU\_\mathcal{X}}$ and $S_{GPU\_\mathcal{X}}$.
(2) SPT-GCN is utilized to select the optimal formats, F_{CPU} and F_{GPU}, for each of $S_{CPU\_\mathcal{X}}$ and $S_{GPU\_\mathcal{X}}$. F_{CPU} and F_{GPU} must belong to COO, CSR, ELL, HYB. After determining the optimal format, two slice sets are converted.
(3) The $S_{GPU\_\mathcal{X}}$ is stored by F_{GPU} format and is executed by SpTTM_GPU_F_{GPU} with CUDA to obtain $S_{GPU\_\mathcal{Y}}$. The $S_{CPU\_\mathcal{X}}$ is stored by F_{CPU} format and is executed by SpTTM_CPU_F_{CPU} with OpenMP to obtain $S_{CPU\_\mathcal{Y}}$.
(4) $S_{GPU\_\mathcal{Y}}$ and $S_{CPU\_\mathcal{Y}}$ are gathered and merged into the result tensor $\mathcal{Y}$.

4.2.2 Detail Parallel Format Implementation on CPU-GPU. The parallel execution method of slice-wise SpTTM computation is different for different sparse formats, and we design the parallel execution for the four selected sparse formats as follows in support to heterogeneous invocation. Each format is designed with the appropriate amount of threads and the proper thread execution manner.

ALGORITHM 1: The process of Slice-wise SpTTM on CPU-GPU heterogeneous computing system.

Input:
A tensor $\mathcal{X}$;
A matrix U;
Mode J;
Output:
A tensor $\mathcal{Y}$;
Divide $\mathcal{X}$ into two slices set $S_{CPU\_\mathcal{X}}$ and $S_{GPU\_\mathcal{X}}$ by task mapping strategy;
Select one optimal format, F_{CPU} and F_{GPU}, for each of $S_{CPU\_\mathcal{X}}$ and $S_{GPU\_\mathcal{X}}$ by SPT-GCN and convert these slices;
# pragma omp parallel
{
    # pragma omp sections nowait
    {
        // Build the parallel sections for GPU
        # pragma omp section
        {
            $S_{GPU\_\mathcal{Y}} \leftarrow$ Call the F_{GPU} kernel function SpTTM_GPU_F_{GPU} ($S_{GPU\_\mathcal{X}}$);
        }
        // Build the parallel sections for CPU
        # pragma omp section
        {
            $S_{CPU\_\mathcal{Y}} \leftarrow$ Call the F_{CPU} kernel function SpTTM_CPU_F_{CPU} ($S_{CPU\_\mathcal{X}}$);
        }
    }
}
$\mathcal{Y} \leftarrow$Gather and Merge $S_{GPU\_\mathcal{Y}}$ and $S_{CPU\_\mathcal{Y}}$;
return $\mathcal{Y}$;

For COO. For slices stored in COO format. Due to the independence of elements in COO, parallel multiplication operations can be performed directly between elements. Then a segmented reduction is delivered to the final accumulation of intermediate products. For the parallel approach on the CPU, non-zero elements are assigned to threads in a segmented manner performing element multiplication. Each thread executes a segmented reduction to sum the intermediate products. For GPU, successive threads in one warp obtain adjacent elements and eventually perform interleaved reduction to get the final result.

For CSR. For slices stored in CSR format, all the non-zero elements are compressed by a continuous way. We naturally associate this with parallel operations in row units. Each row of the input slice is independent with others, which is assigned to a thread in CPU. Each thread independently performs vector-matrix multiplication to obtain one row of the final result slice. For GPU, we allocate one warp for each row to achieve better coalesced memory access and store the data in the order of the main columns. This memory layout makes sure that the rows of one slice are separate and the intermediate products are not mixed up. The rows in a slice are repeated over and over again whose times are the number of columns of the input matrix. As such, shared memory for less latency is suitable to store the non-zero elements in reused rows in the input tensor which reduces the total number of global memory access by $row\_num \times (column\_num - 1)$.

For ELL. For slices stored in ELL format, elements are stored continuously and the original matrix is compressed to the left. All non-zero elements are packaged into two matrix arrays of the same size facilitating the distribution of rows which store the value and column index of each non-zero element. Each row of the slice in ELL format has the same number of elements by zero filling. The number of rows in a slice with ELL format is the same as that of the slice but the number of columns depends on the maximum non-zero element of all rows in the slice. For OpenMP in CPU, each thread is assigned to a row in the resulting slice to access these two arrays at the same time to gain the value and column index. The parallel method significantly tasks a row as the execution unit which avoids the write-read conflicts among threads. In GPU, each row of the input matrix is accessed by a warp instead of accessing the input slice which magnifies the coalesced access. In that fashion, reusability was built into each non-zero element by a warp and threads within the same warp access slices consecutively.

For HYB. For slices stored in HYB format, the whole slice is divided into two parts, dense and sparse, by a threshold, which employs two sparse formats, ELL and COO. To cut down the additional memory for padding, all the rows that stand out much more than others are clipped separately to ELL and COO formats. The denser part is stored in the ELL format and the remaining elements are in the COO format. By the way, the threshold is often max (4096, $row\_number/3$). The structure of HYB calculations is almost identical to ELL, except for detaching some threads to perform COO operations.

4.2.3 Key Problems. For input-aware slice-wise SpTTM on CPU-GPU, there are two key problems. The first problem is how to design a load-balanced mapping strategy due to the different arithmetic intensities of CPU and GPU. The second problem is how to choose the appropriate sparse format for the tensor due to the different sparse structures of the tensor. In the next two sections, we will describe in detail how we solve these two problems.

5 LOAD BALANCE FOR TASK MAPPING ON CPU-GPU

The efficient execution of an application on a heterogeneous system requires the cooperation of multiple specialized devices. In a heterogeneous system equipped with both CPU and GPU, if the program is only executed on the CPU or GPU, it is a waste of computing resources. Efficient SpTTM calculation requires the CPU and GPU to calculate together.

In the input-aware slice-wise SpTTM algorithm, SpTTM is divided into multiple calculation tasks of matrix multiplication. How to design a task mapping scheme so that multiple tasks are efficiently loaded by different devices is the primary problem that needs to be solved. When designing a task mapping scheme, the load balancing of computing tasks on different devices needs to consider the difference in the amount of calculation between tasks, the different computer power of devices in heterogeneous systems, the difference in memory accessed amount, and the different costs of moving data between devices. Therefore, we can give an evaluation method for load balancing task mapping schemes.

Given a heterogeneous system, the peak computer power (the number of FMA instructions per second) of the CPU is P_c, which is denoted by

$$P_c = Num_c \times F_c \times FMA_c, \tag{4}$$

where Num_c is the number of cores on the CPU, F_c is the main frequency of CPU, and FMA is the number of floating point operations (multiplication and addition) performed by CPU in a single cycle.

The peak computer power of the GPU is P_g, which can be denoted as

$$P_g = Num_g \times F_g \times FMA_g, \tag{5}$$

where Num_g is the number of CUDA Cores on the GPU, F_g is the main frequency of GPU and FMA is the number of floating point operations (multiplication and addition) performed by GPU in a single cycle.

Assuming that the amount of data to be computed is equal to the total amount of accessed memory, denoted as M, the amount of accessed main memory allocated to the CPU is M_c, and the amount of accessed global memory allocated to the GPU is M_g, then

$$M = M_c + M_g. \tag{6}$$

The number of tasks allocated to the CPU is Q_c, and the number of tasks allocated to the GPU is Q_g. It is reasonable to assume a linear relationship between the number of tasks and the amount of data to be computed. Given the coefficient μ, then

$$Q_c = \mu M_c, \quad Q_g = \mu M_g. \tag{7}$$

The calculation times on the CPU and GPU are denoted as t_{cp} and t_{gp}, and the percentage of peak computing in actual calculations is λ, then

$$t_{cp} = \frac{Q_c}{\lambda P_c}, \quad t_{gp} = \frac{Q_g}{\lambda P_g}. \tag{8}$$

The memory accessing time on the CPU and GPU are denoted as t_{cm} and t_{gm}, then

$$t_{cm} = \frac{M_c}{B_c}, \quad t_{gm} = \frac{M_g}{B_g}, \tag{9}$$

where B_c is the bandwidth of the main memory in CPU and B_g is the bandwidth of the global memory in GPU. The bandwidth of the data from the CPU to the GPU is B. Especially, the time to copy data from the CPU to the GPU is recorded as t_r, then

$$t_r = \frac{M_g}{B}. \tag{10}$$

The time for the result data from GPU to CPU is not considered because the outgoing data is small compared to t_r. Inspired by Yu et al. [53], we divided the execution time into computation time and memory access time. The load balancing strategy needs to make the execution time on the CPU equivalent to the execution time on the GPU, so it can be expressed as

$$t_{cp} + t_{cm} = t_{gp} + t_{gm} + t_r. \tag{11}$$

Substituting Equations (8), (9), (10) into (11),

$$\frac{Q_c}{\lambda P_c} + \frac{M_c}{B_c} = \frac{Q_g}{\lambda P_g} + \frac{M_g}{B_g} + \frac{M_g}{B}, \tag{12}$$

then combining Equations (6), (7), and (12) to get the number of tasks allocated to the CPU and the number of tasks allocated to the GPU. Equation (7) shows that Q and M have a linear relationship, so we use Mg as an example to illustrate the assignment relationship between CPU and GPU.

Equation (12) is deformed to

$$\frac{\mu(M - M_g)}{\lambda P_c} + \frac{M - M_g}{B_c} = \frac{\mu M_g}{\lambda P_g} + \frac{M_g}{B_g} + \frac{M_g}{B}, \tag{13}$$

then M_g is solved,

$$M_g = \frac{P_g B_g B(\mu B_c + \lambda P_c)}{P_g B_g B(\mu B_c + \lambda P_c) + P_c B_c(\mu B_g B + \lambda P_g B + \lambda P_g B_g)} M. \tag{14}$$

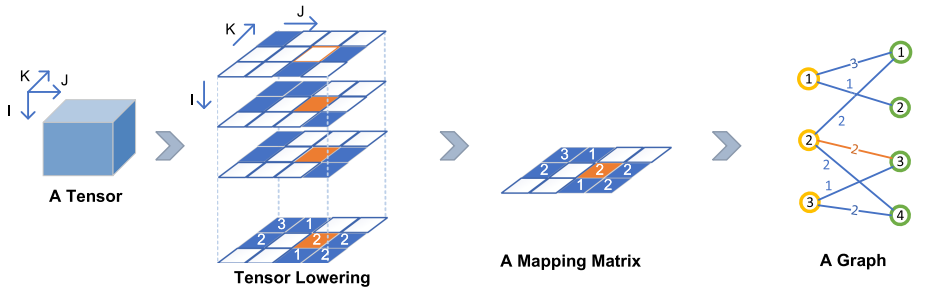


Fig. 5. A diagram of converting a tensor into a bipartite graph.

Further, according to Equations (6), (7), and (14), we can finally obtain the relationship between the number of tasks and the amount of data,

$$Q_g = \mu M_g = \frac{\mu P_g B_g B(\mu B_c + \lambda P_c)}{P_g B_g B(\mu B_c + \lambda P_c) + P_c B_c(\mu B_g B + \lambda P_g B + \lambda P_g B_g)} M, \tag{15}$$

$$Q_c = \mu(M - M_g) = \frac{\mu P_c B_c(\mu B_g B + \lambda P_g B + \lambda P_g B_g)}{P_g B_g B(\mu B_c + \lambda P_c) + P_c B_c(\mu B_g B + \lambda P_g B + \lambda P_g B_g)} M. \tag{16}$$

In the input-aware slice-wise SpTTM algorithm, the tasks calculated by CPU and GPU are closely related to the non-zero element of each slice. We first count the non-zero elements of each slice and assume that the total number of non-zero elements is equal to the total amount of access memory M. Secondly, we sort these slices from large to small according to the counting result. Thirdly, distributing slices to GPU and CPU alternately from large to small to ensure that their task volume meets the Q_c and Q_g above. Finally, a load-balanced task mapping scheme is obtained by the above process.

6 FORMAT SELECTION

In this section, first, a bipartite graph representation is given for describing the sparse structure of the tensor. Second, basic concepts of graph neural networks are introduced, and third, we describe the network structure and hyperparameter settings of SPT-GCN, a model for sparse tensor format selection for SpTTM.

6.1 Graph Representation

Considering the complexity of the tensor high-dimensional space, we perform tensor lowering, that is, use a mapping technique to reduce the tensor into a matrix, which we call the mapping matrix. The mapping matrix is the projection of all slices of the tensor, so the shape of the mapping matrix is as same as the shape of a slice. For each element in the mapping matrix, it is the number of non-zero elements that exist at the corresponding positions of all slices of the tensor, as shown in Figure 5. By tensor lowering, we compress the sparse information of the tensor onto a mapping matrix. Considering bipartite graphs as a standard model for studying the division of rows and columns of matrices [15, 41, 48], we further represent the mapping matrix as an entitled bipartite graph.

As shown in Figure 5, the bipartite graph converted from a tensor is a weighted undirected graph. The graph contains two vertex sets, representing the rows and columns of the matrix, and the non-zero elements of the matrix are represented as the weights of the edges connecting the two sets of vertices. We use graph $G = (R, C, E, W)$ to represent matrix X. If matrix X has a non-zero

element 3 in the 2nd row and 3rd column, there must be an edge $e_{2,3} = (r_2, c_3, w_{2,3})$ in graph G from node r_2 in the vertex set R to node c_3 in the vertex set C, where $w_{2,3}$ is 3, the value of the non-zero element.

6.2 Graph Convolution Network

Notations. In this paper, we use $G = (V, E)$ to denote a graph, where $V = \{v_1, \ldots, v_n\}$ is the set of nodes and E is the set of edges. Let $v_i \in V$ represent a node in the graph, then $e_{i,j} = (v_i, v_j) \in E$ is represented as an edge from node i to j. We define the neighborhood of node v as $N(v) = \{u \in V | (v, u) \in E\}$. Then the adjacency matrix of the graph is represented by $A \in \mathbb{R}^{n \times n}$, and if $e_{ij} \in E$, $A_{ij} = 1$, otherwise $A_{ij} = 0$. If each node of the graph has d features, then the attribute vector of the node v_i is denoted as f_{v_i}, and the attribute matrix of the graph is represented as $F \in \mathbb{R}^{n \times d}$.

A basic GCN model is composed of a set of neural network layers. Each layer gathers local neighborhood information around each node and then transfers the gathered information to the next layer [8]. Assuming that the center node is v, the feature vector is denoted as $f(v)$, and the set of neighbor nodes is denoted as $N(v)$, so we calculate the new feature of the center node v in the t-th layer,

$$f^{(t)}(v) = \sigma\left(f^{(t-1)}(v) \cdot \mathbf{W}_1^{(t)} + \sum_{u \in N(v)} f^{(t-1)}(u) \cdot \mathbf{W}_2^{(t)}\right), \tag{17}$$

where $\mathbf{W}_1$ and $\mathbf{W}_2$ are parameter matrices, and σ is the activation function. We replace the above equation with a permutation invariant, differentiable function and compute the new identity $f^{(t)}(v)$ as

$$f^{(t)}(v) = f_{merge}^{(\mathbf{W}_1)}\left(f^{(t-1)}(v), f_{aggr}^{(\mathbf{W}_2)}\left(f^{(t-1)}(u)\right)\right), \tag{18}$$

where $u \in N(v)$ is a neighbor node of v, $f_{aggr}^{(\mathbf{W}_2)}$ is the operation of merging the neighborhood features of node v, and $f_{merge}^{(\mathbf{W}_1)}$ denotes the operation of aggregating the current features of node v and its neighborhood features [14].

Therefore, the vector representation $f_{GCN}(G)$ of the entire graph can be obtained by summing the vector representations of all nodes,

$$f_{GCN}(G) = \sum_{v \in V(G)} f^{(T)}(v), \tag{19}$$

where $T > 0$ denotes the last layer of the graph neural network [52].

6.3 SPT-GCN

As shown in Figure 6, SPT-GCN combines a GCN and a **multilayer perceptron (MLP)** to predict the optimal storage format for the tensor [7]. The input of SPT-GCN is a bipartite graph converted from a tensor. Specifically, an input node of GraphConv represents a row or a column of the mapping matrix. To reflect the spatial distribution of tensor sparse nonzero elements, the following five attributes are considered as characteristics of the node: the **total number** of nonzero elements in that row/column, the **maximum value**, the **minimum value**, the **mean value**, and the **variance** of the column/row index of all nonzero elements in that row/column.

Considering that higher-*order* structures play an important role in the representation of matrix graphs, we choose k-dimensional Graph Neural Networks [32] to aggregate high-dimensional neighborhood information. For each layer $t > 0$, the computing process of the new features can

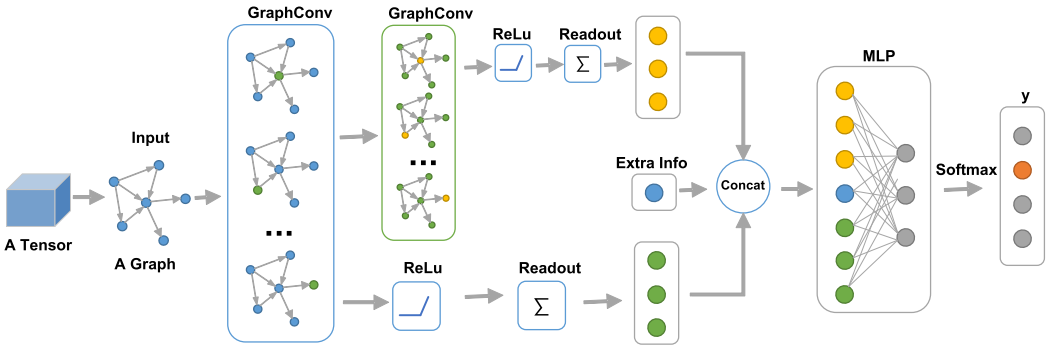


Fig. 6. SPT-GCN model structure diagram.

be expressed as

$$f^{(t)}(v) = \sigma\left(f^{(t-1)}(v)\mathbf{W}_1^{(t)} + \sum_{u \in N_L \cup N_G} f^{(t-1)}(u)\mathbf{W}_2^{(t)}\right), \tag{20}$$

where N_L denotes the node set of local neighborhoods for node v and N_G denotes the node set of global neighborhoods for node v. In k-dimensional Graph Neural Networks, the local neighborhood means all neighboring nodes in the k-th dimension, and the global neighborhood refers to all neighboring nodes except the local neighborhood [32].

Next, we present the working process of the SPT-GCN model in two stages.

Graph convolution is Stage 1, where we obtain knowledge from graph data. First, do a graph convolution operation on the input graph to get the first-*order* graph convolution result. Second, perform another graph convolution operation on the first-*order* graph convolution result to obtain the second-*order* graph convolution result. Each graph convolution operation includes three operators GraphConv, ReLU, and Readout. GraphConv is a graph convolution operation, used to aggregate information from neighboring nodes. ReLU is a pooling operation that aims to reduce the size of parameters and produce smaller representations by downsampling nodes, thus the problems that overfitting, substitution invariance, and computing complexity need to be avoided. The Readout operation is mainly used to generate graph-level representations based on node representations.

Multilayer perceptron is Stage 2, the purpose of this stage is to choose the suitable sparse format for the pair of matrices. The output of Stage 1 is connected using a multilayer neural network, and finally a softmax layer is connected. The output of this stage is the probability of achieving optimal performance for each tensor format, and the tensor format with the highest probability is predicted to be the most suitable for a given tensor.

The larger the k of the k-dimensional Graph Neural Network, the wider the perceptual field of the nodes. However, when $k > 1$, due to the neighboring nodes, it will propagate features to each other many times, and it will generate redundancy. And as k increases, there will be more redundant entries, which is disadvantageous to the final prediction results. Therefore, SPT-GCN uses a 2-dimensional graph neural network to aggregate the 2-hop neighbor information of nodes after several trials. A k-dimensional graph neural network has two graph convolution processes. After the first graph convolution, we output the result as a vector of length 128. After the first graph convolution, we output the result as a vector of length 128. The second graph convolution is performed on the result of the first graph convolution, and the output is again a vector of length 128. Splicing the two vectors gives the final output of the k-dimensional graph neural network, which is a vector of length 256.

Table 2. Experimental Platforms Configuration

Parameters	Intel(R) Xeon(R) Gold 6248 CPU	NVIDIA V100 GPU
Microarchitecture	Cascade Lake	Volta
Frequency	2.50 GHz	1.455 GHz
#Physical cores	20	5120
Last-level cache	27.5 M	16 M
Memory size	132 G	32 G
Memory bandwidth	17.6 GB/s	900 GB/s
Compiler	gcc 5.5.0	nvcc 10.1

The input of MLP contains a feature vector (ExtraInfo) besides splicing the vector output from the two GCNs. ExtraInfo is a vector of length 9, which records some overall information about the tensor: The number of modes of the tensor, the overall sparsity of the tensor, the maximum number of dimensions of the tensor, and the minimum number of dimensions of the tensor. Thus, the input layer of the MLP has 521 = 512 + 4 neurons, and the number of neurons in the two hidden layers is set to 128 and 32. Finally, the number of neurons in the output layer of the MLP is set to the number of categories of the matrix.

Hyperparameter Settings. In addition, when training the model, we try to set the batch size to *[1, 2, 4, 8, 16, 32]* and the learning rate to *[0.001, 0.0005, 0.0001, 0.00001, 0.000001]*. After several attempts, we use the parameter combination set that can achieve better training results, with batch size set to 4 and learning rate set to 0.00001. To speed up the operation and improve data stability, we use a combination of LogSoftmax and NLLLoss for our loss function.

7 EXPERIMENTATION

In this section, we perform a number of experiments to evaluate the performance of the input-aware slice-wise SpTTM, and SPT-GCN prediction accuracy and the effectiveness of load balancing scheme. All performance comparison experiments perform single-precision floating-point values, and the results are averaged after 10 runs.

7.1 Platforms and Datasets

7.1.1 Hardware and Software Systems. Our tensor computation performance comparison experiments are conducted on a heterogeneous system equipped with both Intel(R) Xeon(R) Gold 6248 CPU and NVIDIA V100 GPU. The full hardware configuration is shown in Table 2. Moreover, our design SPT-GCN implemented using Python 3, PyTorch 1.4, and PyTorch Geometric (PyG) 1.6 [13].

7.1.2 Datasets. To better train SPT-GCN, we prepared more tensors as the training set. We generated 4,426 third-*order* tensors based on 2,016 matrices selected from the SuiteSparse Matrix Collection.[1] The number of non-zero elements ranged from 42 to 2,652,858. For SpTTM, we evaluated sparse tensor storage formats, including COO, CSR, ELL, and HYB. The y-label of each tensor in the training set was the format with the lowest average time overhead for 10 iterations.

The datasets used in the experiments are publicly available and we chose to use the following datasets from five different application domains. Besides the training dataset, the datasets used for our tests are *Uber Pickups*, *Hetrec2011 Lastfm 2k*, *Chicago Crime*, and *NeurIPS Publications*. These are publicly available and collected in *FROSTT*[2] for real-world applications.

[1] http://sparse.tamu.edu/.
[2] frostt.io/tensors/

Table 3. Details of the Tensor Datasets

Tensor Dataset	Description	I	J	K	Non-zero Elements
Uber Pickups	(Time, Latitude, Longitude)	4,392	1,140	1,717	3,309,490
Hetrec2011-lastfm-2k	(User, Artist, ListeningCount)	2,100	18,744	12,647	186,479
Chicago Crime	(Time, Area , Type)	148,464	77	32	5,330,673
NeurIPS Publications	(PaperID, Author, Word)	42,194	2,862	14,036	3,101,609

Table 3 is a summary of these datasets.

7.1.3 Baseline Methods. The most used methods to recognize sparse matrix structures by machines are SVM [6] and CNN [54]. Thus, we choose CNN and SVM as our proposed SPT-GCN comparison methods. The SVM method uses the statistical information of the matrix as the features of the matrix and classifies the matrix according to the features of the statistical information. The CNN method first forms the matrix into a histogram or dense graph form, and then uses Convolution neural networks for feature extraction and classification. In our experiments, the CNN method uses the source code provided by the authors to compare the experimental results, while the SVM and the native GCN are implemented based on algorithmic ideas to compare the experimental results.

We compare our method with two state-of-the-art tensor libraries, ParTI! and SPLATT,[3] to evaluate the performance of input-aware slice-wise SpTTM. To distinguish between ParTI! library implementations on CPU and GPU, the parallel implementation of ParTI! library on CPU multicore is noted as ParTI!-OMP and on GPU architecture is noted as ParTI!-GPU. We perform SpTTM calculations on five tensor data sets in the test set, and compare the performance of our proposed input-aware slice-wise SpTTM method with SPLATT, ParTI!-OMP and ParTI!-GPU.

In particular, we propose the input-aware slice-wise SpTTM method on the CPU-GPU heterogeneous system, and we name the input-aware slice-wise SpTTM using the SVM method as AdaptSVM, the one using the CNN method as AdaptCNN, and the one using the SPT-GCN method as AdaptSPT-GCN.

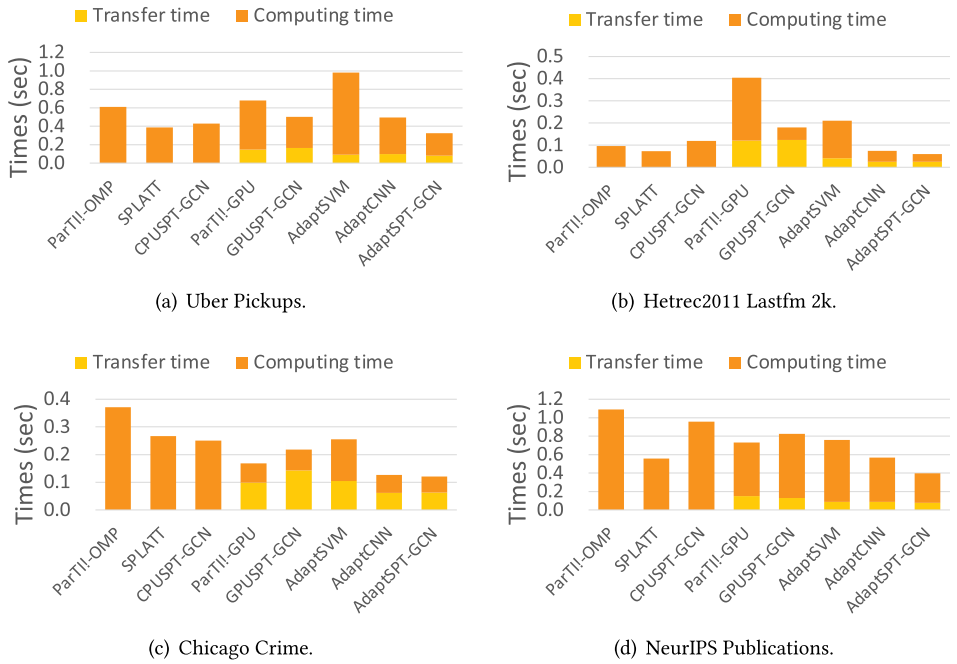
More, to demonstrate the performance benefits of heterogeneous computing more concretely, we compare CPUSPT-GCN, an approach that uses SPT-GCN only on the CPU, and GPUSPT-GCN, an approach that uses SPT-GCN only on the GPU.

7.2 Performance

7.2.1 Experiment Setting. To evaluate the performance of input-aware slice-wise SpTTM, we recorded the time overhead of executing SpTTM on different tensor datasets. In practical applications such as tensor decomposition, SpTTM usually needs to be executed on all modes of the tensor. Therefore, the experiment is repeated 10 times for each tensor dataset, and the average of the sum of the time overheads of executing SpTTM once for each mode is recorded and presented.

7.2.2 Time Overhead. The experiments are executed on the Intel(R) Xeon(R) Gold 6248 CPU and NVIDIA TITAN RTX GPU heterogeneous system, and the results are shown in Figures 7(a), (b), (c), (d). Comparing AdaptSPT-GCN and AdaptCNN, and AdaptSVM, it can be seen that SPT-GCN has a better acceleration effect than CNN and SVM for optimizing the performance of SpTTM computation. Figure 8 specifically shows the prediction accuracy and performance of different methods, where SVM (GPU-62.77% and CPU-63.76%) in the legend indicates that the prediction accuracy of SVM is 62.77% on GPU and 63.76% on CPU, and the representation of CNN and SPT-GCN is

[3] https://github.com/ShadenSmith/splatt/.



(a) Uber Pickups. (b) Hetrec2011 Lastfm 2k.

(c) Chicago Crime. (d) NeurIPS Publications.

Fig. 7. SpTTM performance of different methods.

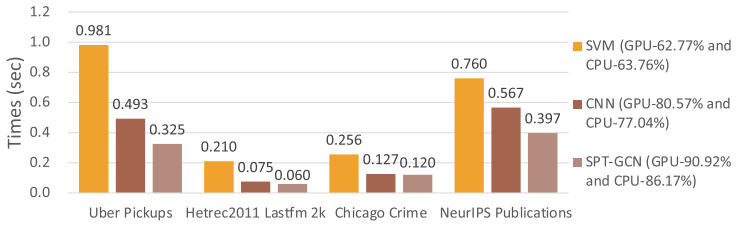


Fig. 8. SpTTM performance of different methods with various prediction accuracy.

similar. Observing the results, it can be further argued that for the sparse format selection model of tensor, the higher the prediction accuracy, the more significant the performance improvement of SpTTM.

For the implementation library on the CPU, we measure the computing time for its execution of SpTTM. For the implementation library on GPU, we measure the computation time to execute SpTTM and the transfer time between CPU and GPU. Figure 9 shows the results of our measured time comparison for format conversion between our implementation and the SPLATT and ParTI! libraries. As can be seen, the time for format conversion is a heavy overhead for all the implemented libraries. Fortunately, SpTTM has iterated dozens or even hundreds of times in tensor decomposition applications, and the time overhead of format conversion can be evenly spread by these iterations.

Comparing AdaptSPT-GCN with CPUSPT-GCN, and GPUSPT-GCN, it can be seen that the heterogeneous computing approach has better performance improvement compared to using only CPU or GPU. Specifically, the CPU-GPU heterogeneous method improves performance by 45.98% over the CPU method and 49.62% over the GPU method.

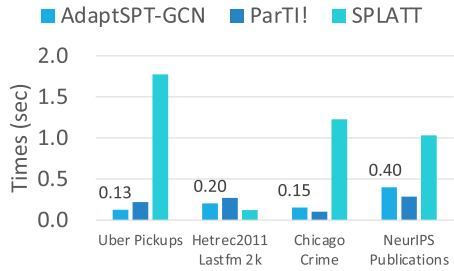
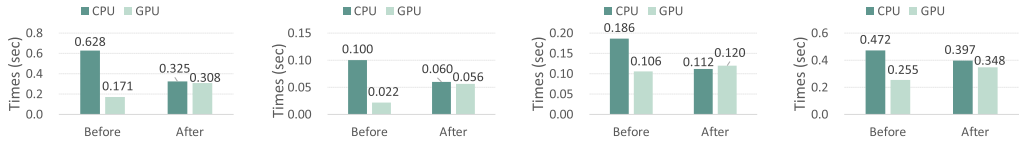


Fig. 9. Time overhead for format conversion of different methods.



(a) Uber Pickups. (b) Hetrec2011 Lastfm 2k. (c) Chicago Crime. (d) NeurIPS Publications.

Fig. 10. Comparison of the time overhead before and after performing load balancing.

More, comparing AdaptSPT-GCN with SPLATT, ParTI!-OMP, and ParTI!-GPU, our proposed AdaptSPT-GCN method achieves SPLATT of 1.504×, ParTI!-OMP of 2.636× and ParTI!-GPU acceleration of 1.310×.

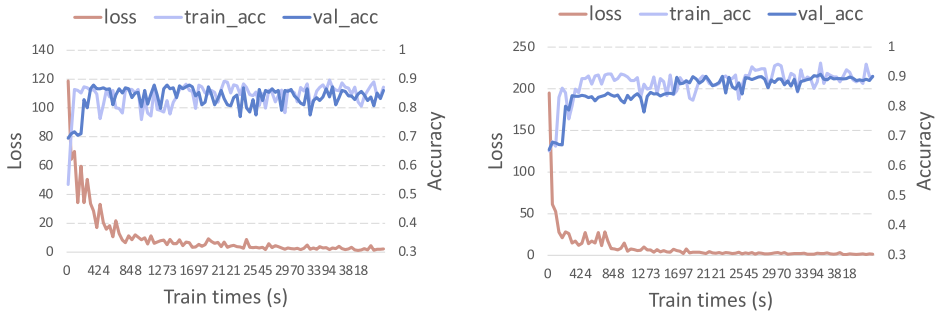
7.3 Load Balance

7.3.1 Experiment Setting. To demonstrate the effectiveness of our designed load balancing scheme, we compare the time overhead before and after load balancing. The task allocation scheme without load balancing is a random allocation of slices on the CPU and GPU, noted as Before. And the scheme after load balancing is denoted as After.

7.3.2 Time Overhead. Figure 10 shows the time overhead on the CPU and the GPU before and after executing the load balancing scheme on the four tensor datasets. It can be seen that without load balancing, that is, before executing the load balancing scheme, there is a large difference between the time overhead on CPU and GPU. Since the time overhead of heterogeneous computation depends on the maximum value of time overhead on CPU and GPU, the large difference in time overhead on CPU and GPU slows down the overall performance. In contrast, after implementing the load balancing scheme, the difference in time overhead on CPU and GPU is relatively small. Specifically, the average performance improvement after load balancing is 34.91% relative to the before. Thus, it can be seen that our scheme enables input-aware slice-wise SpTTM load balancing on CPU-GPU.

7.4 Prediction Accuracy of Format Selection

7.4.1 Experiment Setting. Cross-validation is a common evaluation method in statistical learning. Therefore, we use 5-fold cross-validation to separate the validation data from the training data. The top 20% of the dataset is randomly removed to form a validation set and the rest is used for training. This process was then repeated five times, each time selecting a different subset of the dataset as the validation set. To evaluate the accuracy of SPT-GCN predictions, we measure the average prediction accuracy. We denote the number of tensors with the same category as y-label obtained by SPT-GCN on the validation set as m and the total number of all tensors in the



(a) CPU. (b) GPU.

Fig. 11. The training process of SPT-GCN.

Table 4. The Proportion of Tensors of Different Sizes of the Train Dataset

The number of non-zeros of the tensor	0–100	100–1,000	1,000–10,000	10,000–100,000	>100,000
The number of tensors	252	1360	1304	1130	380
Proportion	5.69%	30.73%	29.46%	25.53%	8.59%

validation set as n, then the accuracy rate is expressed as $\frac{m}{n}$, and the average accuracy of 5-fold cross-validation is denoted as

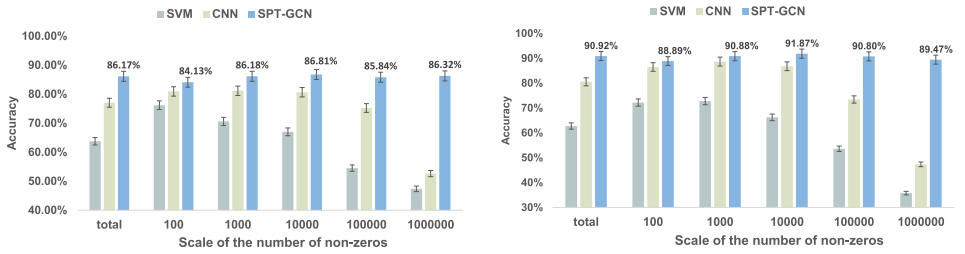
$$acc = \frac{\sum_{f=1}^{F} \frac{m_f}{n_f}}{F}, \tag{21}$$

where the value of F is 5, and m_f and n_f denote the number of tensors with the same category as y-label and the total number of all tensors in the validation set obtained in each 5-fold cross-validation.

7.4.2 Prediction Accuracy. Figures 11(a) and (b) show the training process of SPT-GCN. The loss indicates the loss value of SPT-GCN, which gradually decreases with the increase in train times. And the loss value eventually stabilizes, which means the SPT-GCN is converged. The train_acc and val_acc indicate the prediction accuracy of SPT-GCN on the training and validation sets, which is incrementally stabilized. On the CPU, the accuracy of the training set stabilizes to 0.878 and the accuracy of the validation set stabilizes to 0.851. On the GPU, the accuracy of the training set stabilizes to 0.924, and the accuracy of the validation set stabilizes to 0.901. Therefore, it can be seen that SPT-GCN can choose a suitable sparse format for the tensor, whether on CPU or GPU.

To deeply analyze the difference between our method and other methods, we divide the whole dataset into five different sub-datasets according to the number of non-zero elements and analyze the results on the sub-datasets.

The statistical results of different sub-datasets are shown in Table 4. It can be seen that 85.72% of the tensors in the training dataset we used have a number of non-zero elements in the interval from 100 to 100,000, while only 5.69% is less than 100 and only 8.59% is greater than 100,000. Figures 12(a) and (b) give the experimental comparison results of the prediction accuracy of SPT-GCN with other machine learning methods. It can be seen that SPT-GCN outperforms other methods both on CPU and GPU, and the advantage becomes more obvious as the size of tensor non-zero elements increases. The reason is that for large matrices, CNN loses too much structural information in the process of scaling the matrix into a fixed-size matrix. Similarly, SVM generates large errors when classifying in large-scale matrices using only the overall features of the matrix. In contrast, the use of SPT-GCN is not affected by the size of the matrix in terms of accuracy.



(a) CPU. (b) GPU.

Fig. 12. Accuracy of optimal sparse format prediction.

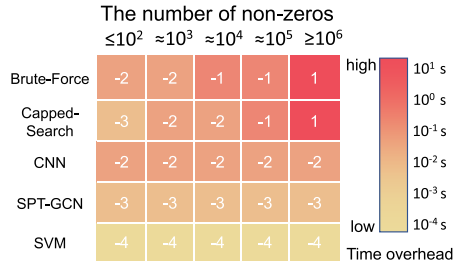


Fig. 13. Time overhead of searching for the best sparse format.

7.4.3 Overhead Analysis. We also discuss the time overhead of using a deep learning selection method compared to an iterative search approach. For selection approaches, we still use three selection methods, SVM, CNN, and SPT-GCN. For traversal approaches, we choose the brute force traversal method and the greedy traversal method. The brute force method is the most common traversal method, in which all the methods are run through and the method with the shortest running time is selected as the best method. The greedy traversal method is based on brute-force traversal, which takes the time of the fastest method among the currently known methods as the upper limit, and does not continue to compute as long as the running time of subsequent methods exceeds this limit. Thus, the time overhead of the greedy traversal method is lower than that of the brute-force traversal method.

Figure 13 shows the time overhead of various methods to identify the best format of the matrix, the higher the time overhead of the method and the redder the background color. Since the values of the different methods vary widely, we use a logarithmic scale to describe the time overhead, e.g., -1 shows the time overhead at the 10^{-1}s level.

The experimental results show that the time overhead required by the traversal method is at least one order level higher than that of the recognition method in most cases. CNN is 10^{-2}s level of overhead and SPT-GCN is 10^{-3}s level of overhead, numerically; SPT-GCN has a lower time overhead than CNN, which means SPT-GCN is faster than CNN. SVM is 10^{-4}s level of overhead; it is faster than SPT-GCN, but it has lower prediction accuracy. Overall, the time overhead required by our proposed SPT-GCN method is only one order of magnitude larger than that of SVM because SVM only uses the statistical features of the matrix and the computing effort is small. And SPT-GCN is about 1 level less than CNN because the time overhead of convolution computation is larger than that of graph convolution.

8 CONCLUSION

This paper introduces a method to develop heterogeneous parallel slice-wise SpTTM on a hybrid CPU-GPU system. Our SpTTM employs a parallel strategy in various sparse formats and designs

the task mapping scheme to facilitate the computer power of both CPU and GPU. The theoretical analyses exploit the peak performance in different processors. Then, we describe the tensor nonzero element distribution by a graph structure, and design a graph neural network model SPT-GCN, which is able to select a suitable sparse format for the tensor and verify the effectiveness of SPT-GCN through experiments. To sum up, the input-aware slice-wise SpTTM on a CPU-GPU heterogeneous system is implemented to improve the overall performance of SpTTM through task partitioning and parallel optimization. Finally, we experimentally verify that our proposed method has a better speedup ratio compared with the ParTI! library and SPLATT library. In future work, we will further extend the optimization method to more tensor computing operators such as MTTKRP to significantly improve the performance of tensor applications.

REFERENCES

[1] Animashree Anandkumar, Rong Ge, Daniel Hsu, Sham M. Kakade, and Matus Telgarsky. 2014. Tensor decompositions for learning latent variable models. *Journal of Machine Learning Research* (2014).

[2] Hartwig Anzt, Terry Cojean, Chen Yen-Chen, Jack J. Dongarra, Goran Flegar, Pratik Nayak, Stanimire Tomov, Yuhsiang M. Tsai, and Weichung Wang. 2020. Load-balancing sparse matrix vector product kernels on GPUs. *ACM Transactions on Parallel Computing (TOPC)* 7 (2020), 1–26.

[3] Brett W. Bader and Tamara G. Kolda. 2007. Efficient MATLAB computations with sparse and factored tensors. *SIAM J. Sci. Comput.* 30 (2007), 205–231.

[4] Grey Ballard and Kathryn Rouse. 2020. General memory-independent lower bound for MTTKRP. In *PPSC*.

[5] Cem Savas Bassoy. 2019. Design of a high-performance tensor-vector multiplication with BLAS. In *ICCS*.

[6] Akrem Benatia, Weixing Ji, Yizhuo Wang, and Feng Shi. 2016. Sparse matrix format selection with multiclass SVM for SpMV on GPU. *2016 45th International Conference on Parallel Processing (ICPP)* (2016), 496–505.

[7] Hong Chang. 2021. Multilayer perceptron. *Machine Learning – A Journey to Deep Learning* (2021).

[8] Cen Chen, Kenli Li, Sin Gee Teo, Xiaofeng Zou, Kuan-Ching Li, and Zeng Zeng. 2020. Citywide traffic flow prediction based on multiple gated spatio-temporal convolutional neural networks. *ACM Transactions on Knowledge Discovery from Data (TKDD)* 14 (2020), 1–23.

[9] Yuedan Chen, Guoqing Xiao, Kenli Li, Francesco Piccialli, and Albert Y. Zomaya. 2022. fgSpMSpV: A fine-grained parallel SpMSpV framework on HPC platforms. *ACM Transactions on Parallel Computing (TOPC)* 9 (2022), 1–29.

[10] Yuedan Chen, Guoqing Xiao, M. Tamer Özsu, Chubo Liu, Albert Y. Zomaya, and Tao Li. 2020. aeSpTV: An adaptive and efficient framework for sparse tensor-vector product kernel on a high-performance computing platform. *IEEE Transactions on Parallel and Distributed Systems* 31 (2020), 2329–2345.

[11] Andrzej Cichocki. 2014. Era of big data processing: A new approach via tensor networks and tensor decompositions. *ArXiv* abs/1403.2048 (2014).

[12] Guohao Dai, Guyue Huang, Shang Yang, Zhongming Yu, Hengrui Zhang, Yufei Ding, Yuan Xie, Huazhong Yang, and Yu Wang. 2022. Heuristic adaptability to input dynamics for SpMM on GPUs. *ArXiv* abs/2202.08556 (2022).

[13] Matthias Fey and Jan E. Lenssen. 2019. Fast graph representation learning with PyTorch geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*.

[14] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. 2017. Neural message passing for quantum chemistry. In *Proceedings of the 34th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Doina Precup and Yee Whye Teh (Eds.), Vol. 70. PMLR, 1263–1272.

[15] Bruce Hendrickson and Tamara G. Kolda. 2000. Partitioning rectangular and structurally unsymmetric sparse matrices for parallel processing. *SIAM J. Sci. Comput.* 21 (2000), 2048–2072.

[16] Joyce C. Ho, Joydeep Ghosh, Steven R. Steinhubl, Walter F. Stewart, Joshua C. Denny, Bradley A. Malin, and Jimeng Sun. 2014. Limestone: High-throughput candidate phenotype generation via tensor factorization. *Journal of Biomedical Informatics* (2014).

[17] Oguz Kaya and Bora Uçar. 2016. High performance parallel algorithms for the Tucker decomposition of sparse tensors. *2016 45th International Conference on Parallel Processing (ICPP)* (2016), 103–112.

[18] T. Kolda and B. Bader. 2009. Tensor decompositions and applications. *SIAM Rev.* 51 (2009), 455–500.

[19] Dana Lahat, Tulay Adali, and Christian Jutten. 2014. Challenges in multimodal data fusion. *European Signal Processing Conference* (2014).

[20] Charles-Francois Vincent Latchoumane, Francois-Benois Vialatte, Jordi Solé-Casals, Monique Maurice, S. Wimalaratna, N. R. Hudson, Jaeseung Jeong, and Andrzej Cichocki. 2012. Multiway array decomposition analysis of EEGs in Alzheimer's disease. *Journal of Neuroscience Methods* (2012).

[21] Jiajia Li, Casey Battaglino, I. Perros, Jimeng Sun, and R. Vuduc. 2015. An input-adaptive and in-place approach to dense tensor-times-matrix multiply. *SC15: International Conference for High Performance Computing, Networking, Storage and Analysis* (2015), 1–12.

[22] Jiajia Li, Yuchen Ma, and Richard Vuduc. 2018. ParTI! : A Parallel Tensor Infrastructure for multicore CPUs and GPUs. (Oct. 2018). http://parti-project.org. Last updated: Jan. 2020.

[23] Jiajia Li, Y. Ma, C. Yan, and R. Vuduc. 2016. Optimizing sparse tensor times matrix on multi-core and many-core architectures. *2016 6th Workshop on Irregular Applications: Architecture and Algorithms (IA3)* (2016), 26–33.

[24] Jiajia Li, Bora Uçar, Ümit V. Çatalyürek, Jimeng Sun, Kevin J. Barker, and Richard W. Vuduc. 2019. Efficient and effective sparse tensor reordering. *Proceedings of the ACM International Conference on Supercomputing* (2019).

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

[26] Min Li, Chuanfu Xiao, and Chao Yang. 2020. a-Tucker: Input-adaptive and matricization-free Tucker decomposition for dense tensors on CPUs and GPUs. *ArXiv* abs/2010.10131 (2020).

[27] B. Liu, Chengyao Wen, A. Sarwate, and M. Dehnavi. 2017. A unified optimization approach for sparse tensor operations on GPUs. *2017 IEEE International Conference on Cluster Computing (CLUSTER)* (2017), 47–57.

[28] Jiawen Liu, Jie Ren, Roberto Gioiosa, Dong Li, and Jiajia Li. 2021. Sparta: High-performance, element-wise sparse tensor contraction on heterogeneous memory. *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2021).

[29] Linjian Ma and Edgar Solomonik. 2021. Efficient parallel CP decomposition with pairwise perturbation and multi-sweep dimension tree. *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2021), 412–421.

[30] Y. Ma, Jiajia Li, X. Wu, C. Yan, Jimeng Sun, and R. Vuduc. 2019. Optimizing sparse tensor times matrix on GPUs. *J. Parallel Distributed Comput.* 129 (2019), 99–109.

[31] Duane Merrill, Michael Garland, and Andrew S. Grimshaw. 2015. High-performance and scalable GPU graph traversal. *ACM Transactions on Parallel Computing (TOPC)* 1 (2015), 1–30.

[32] Christopher Morris, Martin Ritzert, M. Fey, William L. Hamilton, J. E. Lenssen, Gaurav Rattan, and Martin Grohe. 2019. Weisfeiler and Leman go neural: Higher-order graph neural networks. In *AAAI*.

[33] Israt Nisa, Jiajia Li, Aravind Sukumaran-Rajam, Prashant Singh Rawat, Sriram Krishnamoorthy, and P. Sadayappan. 2019. An efficient mixed-mode representation of sparse tensors. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2019).

[34] Israt Nisa, Jiajia Li, Aravind Sukumaran-Rajam, Richard W. Vuduc, and P. Sadayappan. 2019. Load-balanced sparse MTTKRP on GPUs. *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2019), 123–133.

[35] Israt Nisa, Charles Siegel, Aravind Sukumaran Rajam, Abhinav Vishnu, and P. Sadayappan. 2018. Effective machine learning based format selection and performance modeling for SpMV on GPUs. *International Parallel and Distributed Processing Symposium* (2018).

[36] Yuyao Niu, Zhengyang Lu, Meichen Dong, Zhou Jin, Weifeng Liu, and Guangming Tan. 2021. TileSpMV: A tiled algorithm for sparse matrix-vector multiplication on GPUs. *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2021), 68–78.

[37] Filip Pawłowski, Bora Uçar, and Albert-Jan Yzelman. 2019. A multi-dimensional Morton-ordered block storage for mode-oblivious tensor computations. *J. Comput. Sci.* 33 (2019), 34–44.

[38] Naser Sedaghati, Te Mu, Louis-Noël Pouchet, Srinivasan Parthasarathy, and P. Sadayappan. 2015. Automatic selection of sparse matrix representation on GPUs. *Proceedings of the 29th ACM on International Conference on Supercomputing* (2015).

[39] Shruti Shivakumar, Jiajia Li, Ramakrishnan Kannan, and Srinivas Aluru. 2021. Efficient parallel sparse symmetric Tucker decomposition for high-order tensors. *ACDA* (2021).

[40] Shaden Smith and George Karypis. 2015. Tensor-matrix products with a compressed sparse tensor. *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms* (2015).

[41] Shaden Smith, Niranjay Ravindran, N. Sidiropoulos, and George Karypis. 2015. SPLATT: Efficient and parallel sparse tensor-matrix multiplication. *2015 IEEE International Parallel and Distributed Processing Symposium* (2015), 61–70.

[42] Edgar Solomonik, Devin Matthews, Jeff R. Hammond, John F. Stanton, and James Demmel. 2014. A massively parallel tensor contraction framework for coupled-cluster computations. *J. Parallel and Distrib. Comput.* 74, 12 (2014), 3176–3190.

[43] Qingxiao Sun, Yi Liu, Ming Dun, Hailong Yang, Zhongzhi Luan, Lin Gan, Guangwen Yang, and Depei Qian. 2020. SpTFS: Sparse tensor format selection for MTTKRP via deep learning. *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis* (2020), 1–14.

[44] Qingxiao Sun, Yi Liu, Hailong Yang, Ming Dun, Zhongzhi Luan, Lin Gan, Guangwen Yang, and Depei Qian. 2022. Input-aware sparse tensor storage format selection for optimizing MTTKRP. *IEEE Trans. Comput.* 71 (2022), 1968–1981.

[45] Guangming Tan, Junhong Liu, and Jiajia Li. 2018. Design and implementation of adaptive SpMV library for multicore and many-core architecture. *ACM Transactions on Mathematical Software (TOMS)* 44 (2018), 1–25.
[46] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. 2014. Intel Math Kernel Library.
[47] Tongfeng Weng, Xu Zhou, Kenli Li, Peng Peng, and Kuan-Ching Li. 2022. Efficient distributed approaches to core maintenance on large dynamic graphs. *IEEE Transactions on Parallel and Distributed Systems* 33 (2022), 129–143.
[48] Tongfeng Weng, Xu Zhou, Kenli Li, Kian-Lee Tan, and Kuan-Ching Li. 2023. Distributed approaches to butterfly analysis on large dynamic bipartite graphs. *IEEE Transactions on Parallel and Distributed Systems* 34 (2023), 431–445.
[49] Martin Winter, Daniel Mlakar, Rhaleb Zayer, Hans-Peter Seidel, and Markus Steinberger. 2019. Adaptive sparse matrix-matrix multiplication on the GPU. *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2019).
[50] Zhen Xie, Guangming Tan, Weifeng Liu, and Ninghui Sun. 2019. IA-SpGEMM: An input-aware auto-tuning framework for parallel sparse matrix-matrix multiplication. *Proceedings of the ACM International Conference on Supercomputing* (2019).
[51] Wangdong Yang, Kenli Li, and Keqin Li. 2019. A pipeline computing method of SpTV for three-order tensors on CPU and GPU. *ACM Transactions on Knowledge Discovery from Data (TKDD)* 13 (2019), 1–27.
[52] Rex Ying, Jiaxuan You, Christopher Morris, Xiang Ren, William L. Hamilton, and J. Leskovec. 2018. Hierarchical graph representation learning with differentiable pooling. In *NeurIPS*.
[53] Yuanhang Yu, Dong Wen, Ying Zhang, Xiaoyang Wang, Wenjie Zhang, and Xuemin Lin. 2021. Efficient matrix factorization on heterogeneous CPU-GPU systems. *2021 IEEE 37th International Conference on Data Engineering (ICDE)* (2021), 1871–1876.
[54] Yue Zhao, Jiajia Li, Chunhua Liao, and X. Shen. 2018. Bridging the gap between deep learning and sparse matrix format selection. *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2018).

Received 17 May 2022; accepted 13 February 2023