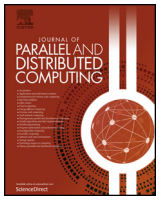
Journal of Parallel and Distributed Computing 181 (2023) 104741

Contents lists available at ScienceDirect

Journal of Parallel and Distributed Computing

journal homepage: www.elsevier.com/locate/jpdc



IAP-SpTV: An input-aware adaptive pipeline SpTV via GCN on CPU-GPU ☆

Haotian Wang [a,b], Wangdong Yang [a,b,*], Rong Hu [a,b], Renqiu Ouyang [a,b], Kenli Li [a,b], Keqin Li [a,b,c]

[a] College of Computer Science and Electronic Engineering, Hunan University, Changsha, Hunan 410082, China
[b] The National Supercomputing Center in Changsha, Changsha, Hunan 410082, China
[c] Department of Computer Science, State University of New York, New Paltz, NY 12561, USA

ARTICLE INFO

Article history:
Received 24 June 2022
Received in revised form 26 June 2023
Accepted 8 July 2023
Available online 17 July 2023

Keywords:
Format selection
GCN
Hybrid format
Pipeline model
SpTV

ABSTRACT

Sparse tensor-times-vector (SpTV) is the core computation of tensor decomposition. Optimizing the computational performance of SpTV on CPU-GPU becomes a challenge due to the complexity of the non-zero element sparse distribution of the tensor. To solve this problem, we propose IAP-SpTV, an input-aware adaptive pipeline SpTV via Graph Convolutional Network (GCN) on CPU-GPU. We first design the hybrid tensor format (HTF) and explore the challenges of the HTF-based Pipeline SpTV algorithm. Second, we construct Slice-GCN to overcome the challenge of selecting a suitable format for each slice of HTF. Third, we construct an IAP-SpTV performance model for pipelining to achieve the maximum overlap between transfer and computation time during pipelining. Finally, we conduct experiments on two CPU-GPU platforms of different architectures to verify the correctness, effectiveness, and portability of IAP-SpTV. Overall, IAP-SpTV provides a significant performance improvement of about 24.85% to 58.42% compared to the state-of-the-art method.

1. Introduction

Tensors as a common form of high-dimensional representation are widely used in Big Data [16]. Analyzing and utilizing these tensor data is a fundamental problem in data analysis. Tensor decomposition is one of the most commonly used data analysis techniques, whose main performance bottleneck is SpTV.

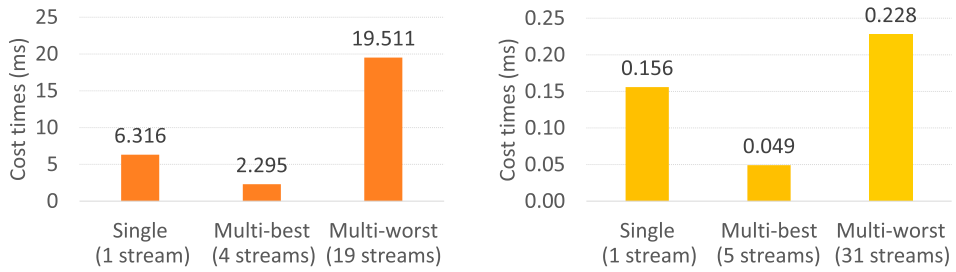
Previous SpTV approaches adopt one of three strategies for utilization. (1) The traditional way is to unfold a tensor into an equivalent matrix and to make use of sparse matrix-times-vector (SpMV) in Tensor Toolbox [14]. This approach takes full advantage of existing matrix optimization methods but requires enormous time-consuming conversion costs. (2) Element-wise operation uses non-zero elements as computational granularity, avoiding other additional overheads. The necessary atomic update operations [21] in ParTI! [18] lead to unnecessary wait times. (3) The stream-based and hybrid format computing model in HOCFS [39] sets the number of streams by experience and uses a hybrid format consisting of two formats. However, this method requires numerous experiments to determine the optimal number of streams, and the hybrid format has limited applicability. In contrast to the above work, we consider introducing a new complex slice-based hybrid format for SpTV to enhance the adaptability for sparse structures. We overlap the execution time of data transfer and computing via the pipeline technique. Naturally, deciding on the best format for each slice and developing a pipeline SpTV algorithm are vital concerns. Various sparse formats are appropriate for data with various sparse distributions. There will be a lack of data locality and performance degradation if a tensor is selected in an inappropriate sparse format. However, selecting a suitable format for various types of sparse data still works.

The burgeoning popularity of deep learning gives new chances to the format selection of sparse tensors [36]. SpTV iterations amortize the additional costs associated with the implementation of machine learning and deep learning. And repeated execution of SpTV is common in large-scale linear systems and iterative processes of tensor decomposition. The selection of the sparse format

☆ The research was partially funded by the Key-Area Research and Development Program of Guangdong Province (Grant no. 2021B0101190004), the National Key R&D Program of China (Grant no. 2021YFB0300800), the Key Program of National Natural Science Foundation of China (Grant nos. U21A20461, 92055213), the Research Innovation Project for Postgraduate Students of Hunan Province (Grant no. CX20220412), and the National Natural Science Foundation of China (Grant no. 61872127).

* Corresponding author at: College of Computer Science and Electronic Engineering, Hunan University, Changsha, Hunan 410082, China.
E-mail address: yangwangdong@hnu.edu.cn (W. Yang).

https://doi.org/10.1016/j.jpdc.2023.104741
0743-7315/



(a) CSR. (b) ELL.

Fig. 1. Performance comparison of SpTV with different stream settings, where the maximum number of streams is limited to 32.

of a tensor using deep learning methods, such as convolutional neural networks (CNN), as the recent research, requires first scaling the tensor into a fixed size matrix [29]. For large scale tensors, the preceding scaling method results in the loss of a significant number of sparsely distributed features, resulting in an unsuitable format selection. Graph Convolutional Network (GCN) delivers strong classification results for structured data [20,23,37] and is a potential method for selecting the optimal sparse format for slices.

The pipelining technique is a skillful method for some computation on the CPU-GPU [3]. This method covers communication overhead by overlapping the computation time and data transfer time on the CPU-GPU. The implementation of the pipeline computing paradigm depends on CUDA streams, and the optimal number of streams is a topic worthy of investigation. Few streams cannot completely overlap data transfer and computation time, whereas a large number of streams will introduce a significant amount of unnecessary overhead for launching streams.

Fig. 1 shows the performance comparison of SpTV with different stream settings. It can be seen that using multiple streams with just the suitable number of streams (Mulit-best) improves performance compared to using only a single stream (Single) while using too many streams (Mulit-worse) reduces performance. The number of streams in the commonly used pipeline computation paradigm is set empirically, while the high-dimensional and intricate sparse structure of the tensor brings a new challenge to the method of setting the number of streams empirically.

To address these issues while operating on three-mode tensors, we present IAP-SpTV, an adaptive pipeline SpTV algorithm on CPU-GPU. In this paper, we

- design a new sparse format called HTF and develop an adaptive pipeline SpTV algorithm. The algorithm uses different types of formats for slices and adaptively set the optimal number of streams for the slice set of each format.
- construct Slice-GCN, a GCN which selects suitable sparse formats for slices, and the model efficiently utilizes tensor fine-grained sparse features to accelerate parallel computing.
- develop an IAP-SpTV performance model for pipelining. The model is used to determine the ideal CUDA stream number setting to maximize the overlap of transfer and computation times during pipelining.
- perform experiments using real datasets, and the results show that IAP-SpTV boosts performance on NVIDIA TITAN RTX GPU by an average of 58.42% over the ParTI! library and 24.85% over HOCFS. Likewise, IAP-SpTV boosts performance on NVIDIA Tesla V100 GPU by an average of 32.23% over the ParTI! library and 24.87% over HOCFS.

The rest of the paper is organized as follows: Section 2 presents the relevant definitions of tensors. Section 3 describes the hybrid format for tensors and a pipeline SpTV algorithm. Section 4 outlines our format selection method. Section 5 constructs a performance analysis model for IAP-SpTV. Section 6 presents our experimental results and findings. Section 7 reviews the related work on tensor operations and tensor sparse format selection. Finally, Section 8 concludes the paper.

Table 1
Table of symbols.

Symbol	Definition
v	A vectors or first-order tensors.
X, U, Y	A matrices or second-order tensors.
$\mathcal{X}$, $\mathcal{Y}$	A third-order tensor.
I, J, K	Tensor mode sizes.
$X_{::k}$	The frontal slices of third-order tensor $\mathcal{X}$.
sp	The sparsity threshold of a tensor.
$\mathbb{G}$	A graph.
h, c	A node of the graph $\mathbb{G}$.
e	An edge of the graph $\mathbb{G}$.
$\mathbb{H}, \mathbb{C}$	The set of nodes.
$\mathbb{E}$	The set of edges.
ξ	The random variable.
Υ_z	The number of rows containing z nonzeros.
β_i	The size of an integer.
β_f	The size of a single-precision floating value.
H	The threshold of HYB format.
H_e	The maximum value of nonzeros of fibers.
F_n	The number of fibers.
E	The length of the fiber.
O	The non-zero numbers of F_n fibers.
B	The bandwidth of PCIe.
G_v, G_s	The storage size.
M	The size of memory transaction.
R	The clock rate of global memory.
N, n	The number of concurrent streams.
N_{sp}	The number of stream processors.
N_w	The warp size.
R_{fma}	The rate of FMA instruction on SM.
T	The total execution time for SpTV.
TT, TT_{h2d}, TT_{d2h}	PCIe transfer time.
CT	Kernel computation time.
MA	Memory accessing time.
CC	Core computation time.
A_i, A_v	The index amount and the value amount.

2. Notations and preliminaries

This section will introduce the basics of tensors and the computation process of SpTV. Table 1 shows the definitions of symbols used in this paper.

2.1. Background

A tensor is a multi-dimensional array. N-way or Nth-order refers to the dimensions of the tensor. The vector is a first-order tensor, denoted by italicized lowercase letters, e.g., v, and the matrix is a second-order tensor, denoted by italicized capital letters, e.g., U. The tensor of third-order or higher is a high-order tensor, denoted by italicized capital calligraphic letters, e.g., $\mathcal{X}$. And

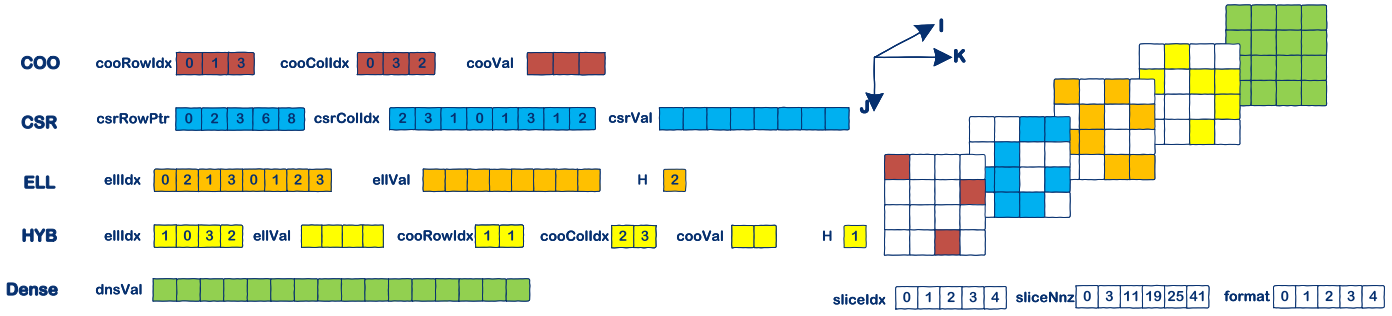


Fig. 2. The tensor $\mathcal{X}$ is of size $5 \times 4 \times 4$ with five slices stored as HTF.

we use $\mathcal{X}_{ijk}$ to represent the element of the third-order tensor $\mathcal{X}$ position (i, j, k).

For a third-order tensor $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$, holding all but one of the indices constant, such as holding all but one of the indices of mode J constant, yields a vector that is called fiber. Holding all but two of the indices constant, such as holding all but two of the indices of mode J and K constant, yields a matrix that is called slice [13].

2.2. The n-mode (vector) product

The n-mode (vector) product is a tensor times vector in mode n. If a tensor is sparse, it is called Sparse tensor-times-vector (SpTV). Taking a tensor as an example, it is a product of multiplying the tensor $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$ by the vector $v \in \mathbb{R}^{J}$ in mode J, denoted by $Y = \mathcal{X} \times_J v$. This results in $Y \in \mathbb{R}^{I \times K}$, where Y is two-order tensor, and its operation is defined as

$$Y_{ik} += \sum_{j=1}^{J} \mathcal{X}_{ijk} v_j. \tag{1}$$

As an example, let the frontal slices of $\mathcal{X} \in \mathbb{R}^{3 \times 4 \times 2}$ be

$$\mathcal{X}_{::1} = \begin{pmatrix} 1 & 7 & 13 & 19 \\ 3 & 9 & 15 & 21 \\ 5 & 11 & 17 & 23 \end{pmatrix}, \mathcal{X}_{::2} = \begin{pmatrix} 0 & 6 & 12 & 18 \\ 2 & 8 & 14 & 20 \\ 4 & 10 & 16 & 22 \end{pmatrix}. \tag{2}$$

Assume a vector $v = [1 \ \ 2 \ \ 3 \ \ 4]^{\mathrm{T}}$, that the result of the 2-mode product of $\mathcal{X}$ and v is Y, and $Y \in \mathbb{R}^{3 \times 2}$ is

$$Y = \begin{pmatrix} 130 & 120 \\ 150 & 140 \\ 170 & 160 \end{pmatrix}. \tag{3}$$

2.3. Sparse format

A common parallel implementation of SpMV is to partition the matrix into multiple segments in matrix operations. Typically, each segment is independent of the others, which makes better reuse of fetched matrix elements and decreases memory footprint. The computation of a segment is performed by a CUDA thread (in the rest of this paper, thread refers to CUDA thread unless otherwise note) when SpMV is executed on GPU, where a thread is the smallest unit of execution of NIVDIA GPU [9]. As a generalization of matrices, tensors are segmented naturally by slices to utilize data parallelism. The sparse structure of tensors is prone to problems such as discontinuous memory accesses and load imbalance for parallel acceleration [33]. Several sparse tensor pattern capture formats, including COO, CSR, ELL, HYB, and Dense, have been proposed to accelerate the SpTV kernel. Therefore, slices can be calculated in several sparse formats.

For COO format slices, three arrays `cooVal`, `cooRowIdx`, and `cooColIdx` are employed to keep a list of its non-zero coordinates and values. Each non-zero element is assigned to one thread that evaluates some multiplications. Assuming that the storage size of an integer is β_i, and the storage size of a single-precision floating value is β_f. Then the total memory size of COO for $O(nnz)$ non-zero elements is $(2 \times \beta_i + \beta_f) \times O(nnz)$. This only costs negligible time, and no more operations are required to insert non-zero elements.

For CSR format slices, the non-zero elements are stored with three arrays `csrRowPtr`, `csrColIdx`, and `csrVal`. The `csrColIdx` indicates the column indices of the non-zero elements, and the array `csrRowPtr` stores the row pointers to the offsets of each row. The last array, `csrVal`, represents the non-zero elements [40]. The total memory size is $\beta_i \times (F_n + 1 + O(nnz)) + \beta_f \times O(nnz)$, where F_n is the number of fibers. Each thread takes charge of one element in the result matrix, which means the dot multiplication between one fiber and one vector.

For ELL format slices, array `ellVal` is set up to store values of non-zero elements of each row, and array `ellIdx` is set up to store the column indices of each non-zero element. These two arrays store the values and column indices of each non-zero element in the natural order. In Fig. 2, array `ellIdx` is $\{0, 2, 1, 3, 0, 1, 2, 3\}$. The maximum number of non-zero elements in a single row is denoted as H_e, which is 2. The memory size for one row is $(\beta_f + \beta_i) \times H_e$. As is still the case in CSR format, each fiber is assigned to one thread.

For HYB format slices, ELL and COO store non-zero elements together [1]. The threshold H (usually set empirically) is used to determine the format in which non-zeros are stored in the row. If the number of non-zero elements in a row is greater than H, the parts exceeding H are stored using COO, and the rest parts are stored using ELL. Otherwise, all non-zero elements are stored in ELL. In Fig. 2, H is set to 1. The total memory size is $G_{hybcoo} + G_{hybell}$. The memory size cannot be expressed because it involves a different number of non-zero elements per row so we will elaborate on the memory size in Equations (15) and (16).

For Dense format slices, only one array `dnsVal` is used to store the values of each element in a dense slice in a natural order. The memory size of array `dnsVal` is $\beta_f \times J \times K$. With one block as a unit, each thread computes one block of the result matrix.

3. Hybrid tensor format

A single format to store the non-zero elements usually ignores the spatial structure of tensors, resulting in huge memory access delay [27]. The sparsity of the part in the whole tensor has not been well exploited while there are many sparse storage formats. To overcome the sparse locality, we propose a hybrid sparse format that uses multiple formats to store slices and further design an HTF-based pipeline SpTV algorithm.

3.1. HTF storage structure

To address the sparsity of the segments, we propose a compressed format called HTF. This hybrid format has several attractive features: (1) it focuses on the local sparsity of slices instead of the whole sparsity of tensors. (2) stores sparse tensor into various slices with optimal formats, (3) implements optimal kernel to obtain better data locality.

HTF format divides the whole tensor into several slices of the same size. Each slice is available in CSR, COO, ELL, HYB, and dense slice (Dense) formats. For simplicity of notation and space, we only write HTF in terms of operating on the first mode such that the size of one slice is $J \times K$. Fig. 2 shows an example of a tensor with size $5 \times 4 \times 4$. We partition the whole tensor into 5 slices of size 4×4 compressing with the above formats.

HTF employs three arrays to record various slice formats, as shown in Fig. 2. Array `sliceIdx` indicates the index of each slice. Array `sliceNnz` represents the starting position of non-zero elements of each slice (in Fig. 2, the starting positions of non-zero elements of these five slices are $\{0, 3, 11, 19, 25\}$). The last entry of this array is the total number of non-zero elements of all slices, where this value in Fig. 1 is 41. Array `format` stores the corresponding types of formats, which in Fig. 1 is $\{0, 1, 2, 3, 4\}$.

We will introduce the selection way of slice formats in Section 4. Once the slice formats are decided upon, the non-zero elements will be compressed into arrays. In this work, we provide the five formats listed in Section 2.3 to select and determine the way each format is computed on the GPU, taking into consideration the universality of the formats and their performance.

3.2. HTF-based pipeline SpTV algorithm

To improve the transfer bandwidth of GPU and the computational efficiency of SpTV, we design a pipeline SpTV algorithm with HTF. Since our data is a sparse and high-dimensional tensor that yields considerable transfer time, the computation process on the GPU generates a lot of irregular computations and discontinuous accesses. Based on this, we deploy CUDA streams to overlap the transfer time and computation in a multi-pipeline fashion to reduce the overall processing time.

The HTF-based Pipeline SpTV algorithm is shown in Algorithm 1. The main idea is to divide an entire tensor in HTF into multiple sets of slices and transfer them from the CPU to the GPU one batch at a time so that the current batch of data can be transferred while the previous batch is being computed.

Before presenting the specific algorithm, we need to determine the splitting slice mode of SpTV to ensure that the algorithm performs slicing that will result in performance benefits. Inspired by [31], larger slice shapes, more non-zeros, and fewer slices can be produced if the dimension with the smallest size is chosen as the slice mode. More non-zeros in the slice, the higher the probability of data reuse. And fewer slices, lower the overhead of format selection. Thus, before performing SpTV, the algorithm selects the smallest mode other than the computed mode as the split slice mode. The algorithm has four inputs, which are the set of slices in five formats in HTF, the vector of multiplication, the mode of multiplication, and the number of CUDA streams to be started for each of the five formats. Line 1 initializes the storage space of the result matrix. Lines 2 to 4 pack each of the five formatted slice sets, result sets, and stream number sets into a separate array for subsequent traversal. Line 5 transfers the vector v from the CPU to the GPU. Lines 6 and 7 initialize the number of CUDA streams N_{max} based on the maximum number of streams for each format, which makes it easier to reuse CUDA streams. Lines 8 to 23 compute the multiplication of the set of slices and vectors of different formats in turn. The process is executed sequentially, meaning that

Algorithm 1 HTF-based Pipeline SpTV algorithm.

Input:
Five slice sets $\mathbb{S}_{CSR}, \mathbb{S}_{COO}, \mathbb{S}_{ELL}, \mathbb{S}_{HYB}, \mathbb{S}_{Dense}$ of the tensor $\mathcal{X}$ in HTF format;
A vector v;
Mode J;
Stream number $N_{CSR}, N_{COO}, N_{ELL}, N_{HYB}, N_{Dense}$;
Output:
A matrix Y;
1: Let $Y_{CSR}, Y_{COO}, Y_{ELL}, Y_{HYB}, Y_{Dense}$ is the result of five slice sets;
2: Let Y = [$Y_{CSR}, Y_{COO}, Y_{ELL}, Y_{HYB}, Y_{Dense}$];
3: Let S = [$\mathbb{S}_{CSR}, \mathbb{S}_{COO}, \mathbb{S}_{ELL}, \mathbb{S}_{HYB}, \mathbb{S}_{Dense}$];
4: Let N = [$N_{CSR}, N_{COO}, N_{ELL}, N_{HYB}, N_{Dense}$];
5: Transfer v from CPU to GPU;
6: Let N_{max} is the maximum value in the array N;
7: Create N_{max} CUDA streams;
8: **for** $t = 0$ **to** 5 **do**
9: #pipeline parallel execution for
10: **for** $n = 0$ **to** $N[t]$ **do**
11: Let S_e is the subset of slices assigned to each stream in the slice set $S[t]$;
12: Let Y_e is the sub-matrix assigned to each stream in the result matrix $Y[t]$;
13: Asynchronous transfer S_e by stream n from CPU to GPU;
14: Let L is the number of slices in the subset S_e;
15: **for** $l = 0$ **to** L **do**
16: Let $s[l]$ is the l-th slice of the S_e set;
17: Let $y[l]$ is the result of the l-th slice of Y_e;
18: Multiply($s[l]$, v, J, $y[l]$, n) on GPU;
19: **end for**
20: Asynchronous transfer Y_e by stream n from GPU to CPU;
21: **end for**
22: cudaDeviceSynchronize();
23: **end for**
24: Stitch $Y_{CSR}, Y_{COO}, Y_{ELL}, Y_{HYB}, Y_{Dense}$ to Y by slice number;
25: **return** Y;

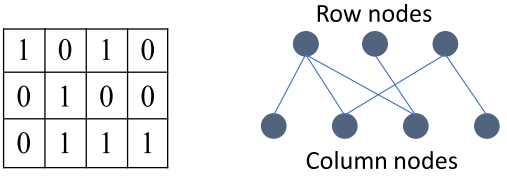
all slices of the current format are processed before moving on to the slices of the following format. Lines 10 to 21 are executed by $N[t]$ pipelines in parallel. Lines 11 and 12 divide the slice set into multiple subsets by the number of streams, which is equivalent to fusing multiple slices into subsets to increase the execution time of the kernel function. Line 13 calls the *cudaMemcpyAsync* function to transfer the slices asynchronously from the page-locked memory on the CPU to the GPU. Line 14 gets the number of slices of the set for this format. Lines 15 to 19 traverse each slice in the subset of slices assigned to stream n and perform the multiplication computation. Line 20 transfers the result asynchronously from the GPU back to the CPU. Line 22 synchronizes waiting for all stream calculations to finish. Line 24 stitches the results of the five formats into the result tensor. Line 25 returns and outputs the result tensor. The format of output $y[l]$ and the stitch tensor in Algorithm 1 is COO format, which stores the indices and values of non-zero elements.

In general, two challenges are involved with the HTF-based Pipeline SpTV algorithm. The first is the selection of a suitable format for each slice in the HTF. The second is adaptively setting the optimal number of streams for each format. IAP-SpTV consists of the HTF-based Pipeline SpTV algorithm and its solutions to these challenges. Hence, we will present our proposed corresponding solutions in the following two sections.

4. Input-aware format selection via Slice-GCN

To overcome the challenge of selecting a suitable format for each slice in the HTF, in this section, we propose a two-stage format selection method to select the best format for slices during the pre-processing stage. If the non-zero element sparsity of a slice is greater than or equal to a threshold sp, the dense format has a significant advantage over all sparse formats.

Thus, **the first stage** judges the sparsity of the slices and sets some of the slices to the dense format. If the non-zero element sparsity of slices is less than the threshold sp, as shown in detail in Section 4.3, the format selection of slices becomes complicated, so we construct Slice-GCN, a graph neural network model for format



(a) Matrix. (b) Bipartite Graph.

Fig. 3. The conversion result of a matrix to a bipartite graph.

selection of sparse slices. Hence, **the second stage** uses Slice-GCN to select the suitable formats for slices.

4.1. Sparse structure representation

Selecting suitable formats for slices is a complex task that usually requires the analysis of a large amount of data. Hence, designing an effective sparse feature extraction method is extremely helpful for the format selection of slices. To achieve this goal, a common approach is to statistically scale an arbitrary-sized slice into a fixed-size matrix for use as an input to a CNN. Nevertheless, this scaling method tends to lose the sparse information of larger slices, so we design a bipartite graph approach to express the sparse features of slices. Specifically, we subdivide the sparse features of slices into two categories: fine-grained features and global features.

Fine-grained features are similar relationships between different rows and columns; therefore, the bipartite graph is used to represent the fine-grained features of the slices. The bipartite graph is a standard model for studying the partitions of rows and columns of a matrix [12,31]. The bipartite graph converted from a tensor is an undirected graph. The graph contains the two vertex sets, representing the rows and columns of the matrix, and the edge connecting the two vertex sets is represented as a non-zero of the matrix. So, the bipartite graph $\mathbb{G} = (\mathbb{H}, \mathbb{C}, \mathbb{E})$ is used to represent the matrix X. If matrix X has a non-zero in the 2nd row and 3rd column, there must be an edge $e_{2,3} = (h_2, c_3)$ in graph $\mathbb{G}$ from node h_2 in the vertex set $\mathbb{H}$ to node c_3 in the vertex set $\mathbb{C}$. The matrix and the bipartite diagram converted from the matrix are shown in Fig. 3 (a) and (b).

Global features number is 9. By traversing the tensor, all 9 features are counted at once, that is, the total number of rows, the total number of columns and the total number of non-zero elements, the mean, maximum, and minimum values of the non-zero numbers of each row, and the mean, maximum and minimum values of the non-zero numbers of each column.

4.2. Slice-GCN design

Graph neural networks are one of the most popular methods for learning graph data and are widely used in areas such as relationship extraction and recommendation systems. Several graph convolution and pooling layers are used by graph neural networks to extract features from graph structures, and a multilayer perceptron (MLP) is applied to classify the feature data [29,37]. Message passing between graph nodes is used by the graph convolution layer to capture the dependencies of the graph. The main idea of message passing is to iteratively aggregate feature messages from neighboring nodes and integrate the aggregated messages with the current central node messages [23]. A multilayer perceptron consists of at least one input layer, one output layer, and one hidden layer, where each node, except the input node, is a neuron using a nonlinear activation function. And the backpropagation method is used by the MLP to learn the mapping between the input and output layers.

Slice-GCN is a redesigned GCN network that applies graph neural networks to select the suitable format for slices. A large number of labeled matrices are demanded training Slice-GCN, so 2041 matrices from SuiteSparse[1] Matrix Collection are used. The process of labeling matrices is divided into two steps. Firstly, the execution time of each matrix of different formats is measured on the target hardware platform. Then the format with the lowest execution time is used as the label of the matrix.

Based on the effective graph structure feature extraction ability of the graph convolution layer and the powerful data classification ability of MLP, we design Slice-GCN to select the optimal format for slices. As shown in Fig. 4, where the GCN is used to extract information about the graph structure, and the MLP is used to classify the feature information extracted by the GCN. To balance the prediction accuracy and computational overhead, we tested each of the commonly used pending options to select the most suitable parameters. Inspired by [34], the pending options for the feature vector size are {16, 32, 64, 128, 256, 512, 1024} and for the hidden layer neurons are {32, 64, 128, 256, 512, 1024, 2048}. After our observation, we finally set the length of the output feature vector to 512 and the number of hidden layer neurons to 128. Specifically, the GCN in Slice-GCN consists of two GraphConv layers, and the length of the output feature vector is set to $512 = 256 \times 2$. The input of the MLP is set to a vector of length $521 = 256 \times 2 + 9$, which is a splice of the output vector of the GCN and the global feature vector of length 9. In addition, since the total number of categories in format is 5, the number of neurons in the hidden layer is set to 128 and the number of neurons in the output layer is set to 5.

4.3. Hyperparameter settings

The sparsity threshold sp of slices is the key hyperparameter of the two-stage format selection method. Suppose Slice-GCN is used to determine whether slices use dense format. In that case, this will increase the computational overhead of Slice-GCN due to a large number of edges in the bipartite graph composed of dense slices and will also introduce a significant error for Slice-GCN that is designed to identify fine-grained sparse structures. In contrast, sparse threshold sp can quickly filter out matrices suitable for using the dense format. We use threshold sp to filter slices in dense format to balance computational overhead and accuracy. To find a suitable sparsity threshold sp, a set of matrices with 500 rows and 500 columns and sparsity from 0.1 to 0.25 with an interval of 0.01 is randomly generated. Thus, all the matrices are divided into 16 groups according to their sparsity, i.e., 0.1 to 0.25, with an interval of 0.01. Each group has 100 matrices. A percentage of a group is recorded, and this percentage is the ratio of the number of matrices whose optimal format is Dense to the number of all matrices in the group. The percentages for all groups are shown in Fig. 5. It is seen that for groups with a sparsity greater than 0.23, the optimal format of all matrices is Dense. As a result, the threshold value of slice sparsity is set to 0.23.

Further, to improve the data stability during training, Slice-GCN uses *LogSoftmax* as the output layer and *NLLLoss* as the loss function, where *LogSoftmax* and *NLLLoss* are open source classes provided by the Pytorch geometry library [8].

5. Performance model for IAP-SpTV

To solve the problem of selecting the optimal number of CUDA streams for each format in HTF, in this section, we quantitatively analyze the transfer and computation execution times of different formats and model the problem of the optimal number of streams to adaptively set the number of streams in IAP-SpTV.

[1] http://sparse.tamu.edu/.

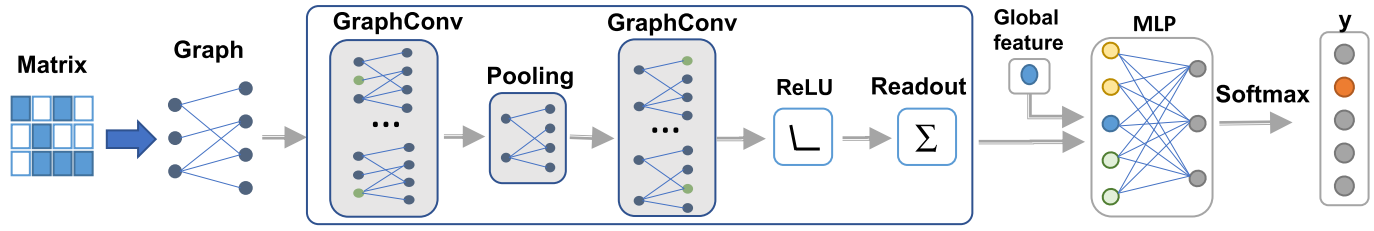
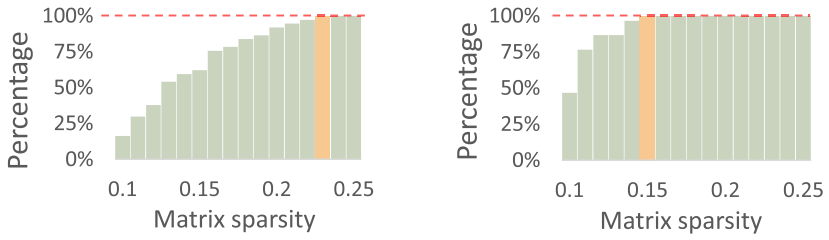


Fig. 4. Slice-GCN model structure diagram.



(a) V100. (b) TITAN.

Fig. 5. The relationship between matrix sparsity and the selected Dense as the best format.

5.1. Performance analysis for HTF

Multiple kernels being launched for diverse formats is a time-consuming operation, which incurs the additional cost of contexts, thereby degrading performance. With the help of streams, GPU programs can efficiently perform memory access and computing operations in parallel, thus improving data throughput. To be concrete, we employ CUDA streams to execute kernels in parallel on the device and to synchronize the stream only when its batch slices are completely executed. Hence, a fiber-wise static analysis method is used to evaluate the time consumed by the above formats on CPU-GPU.

The kernel computation time is influenced by the memory access time (MA), the CUDA core computation time (CC), and the slice format. In the case of the COO format, the kernel computation time is predominantly affected by the memory access time, resulting in $CT = \text{Max}(MA, CC)$. Conversely, for the ELL, CSR, HYB, and Dense formats, which effectively utilize memory bandwidth, the computation time is given by $CT = MA + CC$, as follows:

$$CT = \begin{cases} MA + CC, & \text{For CSR, ELL, HYB, Dense;} \\ \text{Max}(MA, CC), & \text{For COO.} \end{cases} \tag{4}$$

The total execution time T for synchronous SpTV on the CPU-GPU is

$$T = TT + CT, \tag{5}$$

where transfer time (TT) and computation time (CT) are closely related to the number of non-zero elements. For the asynchronous case, we will discuss it further in Section 5.2. The underlying compilation optimization on the GPU usually affects the overlap between MA and CC. The quantitative analysis of this overlap cannot be done by directly analyzing the hardware. Therefore, the effect of this overlap is simplified in this paper in order to facilitate model-solving. Inspired by [19], a distribution function is defined to analyze the number of non-zero elements of fibers within a tensor. The distribution function represents the proportion of the total sample of fibers with a given length in the sample space for a tensor. Given a random variable ξ, denoting the number of non-zero elements of a fiber, the sample space $\Omega = \{1, 2, ..., E\}$, where E denotes the length of the fiber. Thus, the distribution function denoting the number of non-zero elements of the fiber is

$$P(\xi = z) = \Upsilon_z / F_n, z \in \Omega, \tag{6}$$

where Υ_z denotes the number of fibers whose non-zero elements number is z, F_n represents the total number of fibers. The expected value of the number of non-zero elements in the fiber is

$$\mathbb{E}(\xi) = \sum_{z=1}^{z \in \Omega} (z \times P(\xi = z)). \tag{7}$$

Thus, the total number of non-zero elements of F_n fibers is denoted as

$$O = \mathbb{E}(\xi) \times F_n. \tag{8}$$

Since the result is a dense two-order tensor $Y \in \mathbb{R}^{I \times K}$ as shown in Equation (1), assuming that the bandwidth size of PCIe is B, PCIe transfer time for the five different formats is expressed as

$$TT = TT_{h2d} + TT_{d2h} = \frac{\beta_i \times A_i + \beta_f \times A_v}{B} + \frac{\beta_f \times I \times K}{B}, \tag{9}$$

where A_i is the index amount and A_v is the value amount.

For each format, we have described the corresponding memory size in Section 2.3. Then,

$$TT_{COO} = \frac{(2 \times \beta_i + \beta_f) \times O}{B}, \tag{10}$$

$$TT_{CSR} = \frac{\beta_i \times (F_n + 1 + O) + \beta_f \times O}{B}, \tag{11}$$

$$TT_{ELL} = \frac{(\beta_f + \beta_i) \times F_n \times H_e}{B}, \tag{12}$$

$$TT_{Dense} = \frac{\beta_f \times J \times K}{B}; \tag{13}$$

$$TT_{HYB} = \frac{G_{hybcoo} + G_{hybell}}{B}, \tag{14}$$

where H_e is the maximum value of non-zero elements in all fibers. The HYB storage format is evaluated using a hybrid approach, where for each fiber of the slice, the part above the threshold H is stored using the COO format, and the rest is stored using the ELL format. Assuming H is less than $\mathbb{E}(\xi)$, the total number of non-zero elements in the COO part of the HYB format is denoted as $F_n \times (\mathbb{E}(\xi) - H)$ and the total number of non-zero elements in the ELL part is denoted as $F_n \times H$. Thus,

$$G_{hybcoo} = (\beta_f + 2 \times \beta_i) \times F_n \times (\mathbb{E}(\xi) - H), \tag{15}$$

$$G_{hybell} = (\beta_f + \beta_i) \times F_n \times H. \tag{16}$$

The memory accessing time (MA) contains two parts: reading and writing, which mainly considers the time of accessing global memory. The optimization effects of Cache and shared memory are disturbed by complex factors such as intermediate auxiliary arrays, making them difficult to evaluate. To focus on modeling and optimizing the problem of choosing the number of CUDA streams, we have simplified the analysis of computational performance. Contiguous data with a memory transaction length is read on global memory simultaneously. If the data access of one warp is continuous, the latency of access to the global memory can be hidden. Increasing the coalescing of memory accesses by compressing irregular arrays is one of our purposes in choosing a suitable sparse format for slicing. Therefore, we assume that all access operations coalesce. MA of a thread is expressed by

$$MA = \left\lceil \frac{G_{v+s}}{M} \right\rceil \times \frac{1}{R}, \tag{17}$$

where M is the size of a memory transaction, and R is the clock rate of the global memory. G_{v+s} includes two parts, the vector memory (G_v) and the slice memory (G_s) which contains different formats. N_{sp} is the number of stream processors (SP). N_w is the number of threads per warp. N_b is the number of threads per block.

For COO format, the memory accessed by each thread includes three portions, one nonzero element with two indices of `cooRowIdx` and `cooColIdx`, the element in `cooVal` and the corresponding element in the vector, where the memory size for slices is $(2 \times \beta_i + \beta_f) \times O$ as shown in Section 2.3 and for the input vector is $\beta_f \times J$. According to Equation (17), MA is expressed by

$$MA = \left\lceil \frac{(2 \times \beta_i + \beta_f) \times O + \beta_f \times J}{M} \right\rceil \times \frac{1}{R}. \tag{18}$$

Each warp of the GPU performs an inner product operation between two vectors to process a row. Therefore, the computation time required for a warp to execute a sliced row is $\left\lceil \frac{\mathbb{E}(\xi)}{N_w} \right\rceil \times \frac{1}{R_{fma}}$, where $\frac{1}{R_{fma}}$ represents the computation time for float-multiply-add (FMA) operation per thread, and the expected value $\mathbb{E}(\xi)$ represents the number of non-zero values per row. A slice consists of J rows, and if only one warp is used to compute J rows, the computation time required is $\left\lceil \frac{\mathbb{E}(\xi)}{N_w} \right\rceil \times \frac{1}{R_{fma}} \times J$. Fortunately, the GPU can execute $\frac{N_{sp}}{N_w}$ warps simultaneously. Hence, the computation time CC for the kernel is expressed as:

$$CC = \left\lceil \frac{\mathbb{E}(\xi)}{N_w} \right\rceil \times \frac{1}{R_{fma}} \times J \times \frac{N_w}{N_{sp}}. \tag{19}$$

The reasoning process for the other four formats of MA and CC is similar to the COO format, so they are not described in detail.

5.2. Performance model for pipeline

Before measuring the overlap achieved on streams, we only consider 2 copy engines and no implicit synchronization. GPU with 2 copy engines can use the PCIe bus in a full duplex, which completes the full overlap in both directions of computation and communication. No implicit synchronization involves the data being transferred after the kernel is computed. As shown in Fig. 6, the dominant component determines which kind of pieces overlap would select.

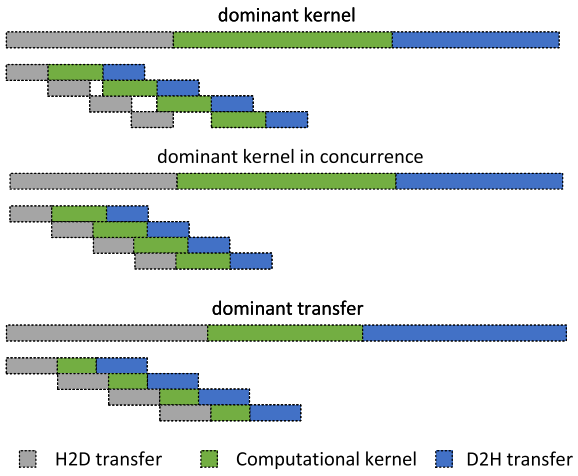


Fig. 6. Overlap with no implicit synchronization and 2 copy engines when using 4 streams.

The additional overhead introduced by start-up streams is relatively large if the execution time is fast, which leads to an inability to optimize the overlap between transfer time and computation time. To balance the start-up overhead and execution time, we transfer them all simultaneously in multiple streams for a slice set of the same format to increase the execution time. Furthermore, for the entire tensor, i.e., a family of slice sets in multiple formats, we transfer the slice sets of each format in batches. In short, one batch transfers the slice set of one format, and multiple batches transfer the whole tensor. Due to the longer kernel computing, streams other than the first stream need to wait for the data from the host. For transfer dominant, each stream gets data from the host fast, so the wait time for all streams is minimized. To sum up, the total execution time $T(n)$ with all streams is modeled as

$$T(n) = \begin{cases} \frac{TT_{h2d}}{n} + CT + \frac{TT_{d2h}}{n} + \lambda n, & CT \geq TT_{h2d}; \\ TT_{h2d} + \frac{CT}{n} + \frac{TT_{d2h}}{n} + \lambda n, & CT < TT_{h2d}, \end{cases} \tag{20}$$

where n is the number of streams and λ is the penalty coefficient to quantify the overhead of creating streams.

As a point of note, when the dominant kernel case is computation, we need to consider whether the kernel functions of each stream completely occupy the computational resources of SMs. The top part of Fig. 6 indicates sufficient computational tasks per stream to occupy the SMs. When enough tasks are unavailable for the streams, concurrent execution of the streams is generated in the middle part of Fig. 6. This overall processing time is

$$T(n) = TT_{h2d} + \frac{CT}{n} + \frac{TT_{d2h}}{n} + \lambda n \tag{21}$$

which is the same as the transfer-dominated case, so the optimal results of concurrent execution can be obtained by consulting the transfer-dominated one.

The pipeline optimization is organized as follows:

$$\arg\min_{n \in [1,N]} T(n), \tag{22}$$

where N is the maximum number of concurrent streams in CUDA. Actually, limited by contexts, registers, etc., the number of streams that can be executed concurrently is finite instead of a creation of infinity in streams.

Theorem 1. *From the function form of (22), the optimization objective $T(n)$ for n is a convex function.*

Table 2
Experimental platforms configuration.

Platform attr	V100		TITAN	
Parameters	Intel(R) Xeon(R) Gold 6248 CPU	NVIDIA V100 GPU	Intel(R) Xeon(R) Silver 4110 CPU	NVIDIA TITAN RTX
Microarchitecture	Cascade Lake	Volta	Skylake	Turing
Frequency	2.50 GHz	1.455 GHz	2.10 GHz	1.77 GHz
#Physical cores	20	5120	8	4608
Peak FP32	–	15 TFLOP/s	–	16.3 TFLOP/s
Last-level cache	27.5 M	16 M	11 M	6 M
Memory size	132 G	32 G	132 G	24 G
Max bandwidth	131 GB/s	900 GB/s	107 GB/s	672 GB/s
Compiler	gcc 5.5.0	nvcc 10.1	gcc 5.4.0	nvcc 10.0

Proof. Given two variables n_1, n_2 and arbitrary rational numbers $\theta \in (0,1)$, then define a function $f(n_1, n_2)$ that

$$f(n_1,n_2) = T(\theta n_1 + (1-\theta)n_2) - \theta T(n_1) - (1-\theta)T(n_2) = \begin{cases} \frac{(TT_{h2d}+TT_{d2h})\times\theta(\theta-1)(n_1-n_2)^2}{[\theta n_1+(1-\theta)n_2]n_1 n_2}, & CT \geq TT_{h2d}; \\ \frac{(CT+TT_{d2h})\times\theta(\theta-1)(n_1-n_2)^2}{[\theta n_1+(1-\theta)n_2]n_1 n_2}, & CT < TT_{h2d}. \end{cases} \tag{23}$$

Obviously, $\theta - 1 < 0$, $f(n_1, n_2) \leq 0$, hence $T(n)$ satisfies the definition of the convex function [15] in these two cases. □

Consequently, there must be an optimal number of streams to minimize the overall GPU execution time in terms of Theorem 1.

Theorem 2. *By Theorem 1, there exist solutions to Equation (22), defined by n_{opt}, satisfying the maximization of the overlap between computational and transfer time. The optimal stream number n_{opt} is*

$$n_{opt} = \begin{cases} \sqrt{\frac{TT_{h2d}+TT_{d2h}}{\lambda}}, & CT \geq TT_{h2d}; \\ \sqrt{\frac{CT+TT_{d2h}}{\lambda}}, & CT < TT_{h2d}. \end{cases} \tag{24}$$

Proof. It is known based on Theorem 1 that the optimization objective of n is a convex function and the derivative of $T(n)$ is obtained for

$$\frac{\mathrm{d}T(n)}{\mathrm{d}n} = \begin{cases} \lambda - \frac{TT_{h2d}+TT_{d2h}}{n^2}, & CT \geq TT_{h2d}; \\ \lambda - \frac{CT+TT_{d2h}}{n^2}, & CT < TT_{h2d}. \end{cases} \tag{25}$$

Setting $\frac{\mathrm{d}T(n)}{\mathrm{d}n} = 0$, then, Theorem 2 is proved. □

According to the aforementioned formats analysis in Section 3, we consider the adaptive number of streams for each format aimed at minimizing T. Each slice has an optimal format, and the execution for a set of slices with the same format is divided into n_{opt} groups according to the number of streams, with a minimum granularity of one slice, to achieve full overlap using pipeline techniques. We balance the transfer time and computation time of each format executed for adaptive streams.

6. Experimental analysis

In this section, a significant number of experiments are conducted to assess the performance of IAP-SpTV on CPU-GPU. All comparative experiments utilize single-precision floating numbers, and the results are the mean of ten runs.

Table 3
Details of the tensor datasets.

Abbr.	Tensor Dataset	I	J	K	Non-zeros
Mov	MovieLens Ratings	71568	65134	6	10000054
Het	Hetrec2011-lastfm-2k	2100	18744	12647	186479
Ube	Uber Pickups	4392	1140	1717	3309490
Chi	Chicago Crime	148464	77	32	5330673
Nip	NIPS Publications	42194	2862	14036	3101609

6.1. Platforms and datasets

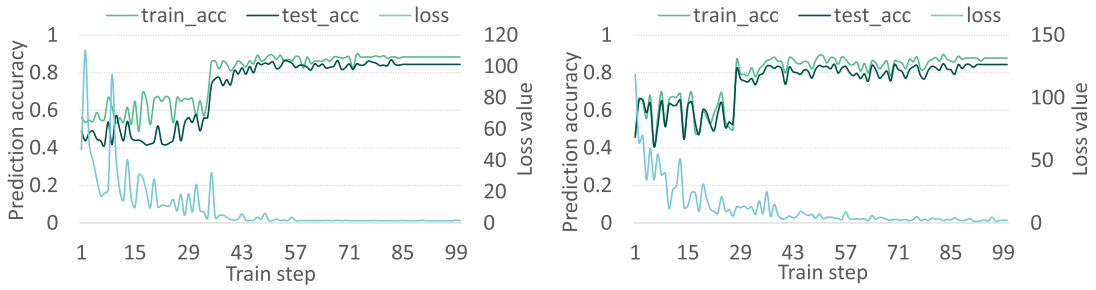
We evaluate our work on two distinct CPU-GPU systems, the full hardware configuration is shown in Table 2. The Slice-GCN code is implemented using Python 3.7, PyTorch 1.4.0 and PyTorch Geometric 1.6.0 (PyG).

Listed in Table 3, the tensor datasets used in this work are publicly available and most of them are from FROSTT [32], with the Hetrec2011-lastfm-2k [4] dataset, the MovieLens Ratings [11] dataset, the Uber Pickups [32] dataset, the Chicago Crime [32] dataset, and the NIPS Publications [10] dataset. Furthermore, Slice-GCN utilizes train data from 2041 matrices inside the publicly available dataset SuiteSparse Matrix Collection. The M-format of the matrix indicates the format that requires the least time to conduct 10 iterations of SpMV, with M-format 0, 1, 2, 3, and 4 representing CSR, COO, ELL, HYB, and Dense, respectively.

6.2. Results of model training

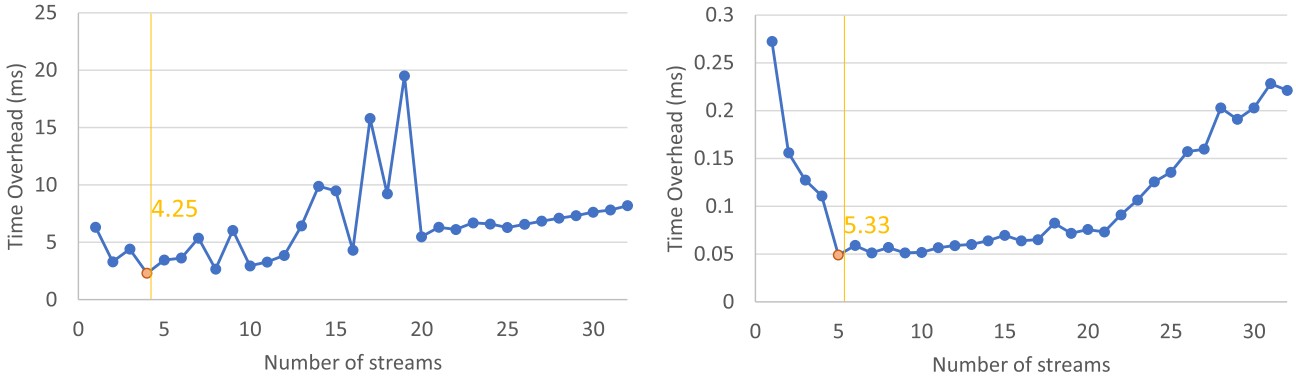
To perform a more comprehensive evaluation of Slice-GCN in Section 4, we show the loss, test accuracy, and train accuracy of Slice-GCN during training. During the training phase, Slice-GCN updates the weight of the neural network based on the loss, which is used to indicate the degree of deviation of the prediction from the target value. Train accuracy and test accuracy show the proportion of correctly predicted cases relative to the total number of cases in the train and test sets, respectively, where a binary value (true/false) indicates whether the prediction is correct.

We divide all matrices of SuiteSparse Matrix Collection into a train set and a test set in the ratio of 80% and 20%, and we label the format in which the matrix performs best on the GPU for SpMV as M-format. Considering the variations in computational performance among slices on different GPUs, we compare the optimal formats for all matrices in our experiments on the Tesla V100 GPU and TITAN RTX GPU. Among the 2041 matrices analyzed, 701 matrices exhibited different optimal formats on these two GPUs. The intricate nature of these differences has motivated us to develop methods that can effectively discern and address them. In summary, the M-format derived from our tests is GPU-specific and cannot be universally applied across GPUs with different architectures. Retraining using the M-format specific to the corresponding GPU is a viable solution to resolve this issue.



(a) V100 GPU M-format. (b) TITAN RTX GPU M-format.

Fig. 7. Loss value, accuracy on the train set, and accuracy on the test set for Slice-GCN during the training phase.



(a) CSR. (b) ELL.

Fig. 8. Comparison of the number of streams analyzed theoretically and measured practically.

To demonstrate the training process of Slice-GCN, the loss values are counted at each step of Slice-GCN training, and the accuracy of the model on the train and test sets are also recorded after each training step, as shown in Fig. 7 (a) and (b). The figures show that as the number of training steps increases, the loss value decreases and stabilizes, and the accuracy rate on the train and test sets increases and eventually stabilizes. A gradual decrease in the loss value means that the prediction of the network for the matrix format of the training data is gradually closer to the best matrix format, which usually leads to a gradual increase in the accuracy of the training set. Specifically, for V100, the model reaches a stable state after about 85 epochs, with a total training time of 1.01 hours. For V100, the model reaches a stable state after about 92 epochs, with a total training time of 2.29 hours. After the model stabilizes, the train set accuracy is 87.81% on the V100 GPU and 88.35% on the TITAN GPU. And the test set accuracy is 84.31% on the V100 GPU and 85.78% on the TITAN GPU.

6.3. Optimal number of streams

To verify the correctness of the optimal number of streams derived from our pipeline performance model in Section 5, we compare the execution time of executing SpTV using different numbers of streams in mode-1 of the Nip dataset for CSR and ELL formats. The experiments are run on the TITAN RTX GPU, and the results are shown in Fig. 8. For different formats, we record the execution time for the number of streams from 1 to 32. And for each format, the results of the theoretical calculation of the optimal number of streams are shown as yellow vertical lines. It is seen that the number of streams obtained from the theoretical calculation (yellow line) is very close to the number of streams with the lowest execution time in practice (red dot). Therefore, the results of our pipeline performance model are correct and valid.

Additionally, the pattern of execution time changes in both formats, falling first and then rising as the number of streams grows. This is a situation because only a few streams can successfully hide the data transfer time. The benefits of hiding data transfer, however, are eventually balanced by the extra overhead of opening streams when more streams are opened, and the amount of processing required for each stream is reduced, as well as the transfer time and computation time.

The CSR format has better adaptability than ELL, which also leads to significant differences in computational execution time between slices of different CSR formats. The substantial oscillations in the line graph of CSR relative to ELL in Fig. 8 demonstrate that CSR is more susceptible to load imbalance as the number of streams rises, and the computational granularity of each stream grows finer.

6.4. Performance

To evaluate the effectiveness of the technique, we compare the SpTV speed of the method based on the adaptive pipeline technique with the current state-of-the-art method on GPUs. Thus, our baseline includes ParTI! [18] and HOCFS [39], where ParTI! is the prevalent tensor library on GPU and HOCFS is the latest research work on pipeline-based implementation for SpTV. In brief, the ParTI! library is a synchronous single-format (COO) library. The HOCFS is a two-format hybrid, pipeline asynchronous, empirically dependent, and static. In contrast, the IAP-SpTV we have presented is a multi-format hybrid, pipeline asynchronous, adaptive, and conceptually analytic.

Furthermore, to demonstrate the progress of Slice-GCN for slice format selection, the currently widely used CNN [41] was made as a comparison method. To facilitate the distinction, the IAP-SpTV implemented based on Slice-GCN is denoted as AdaptGCN, and the SpTV implemented based on CNN and our proposed adaptive

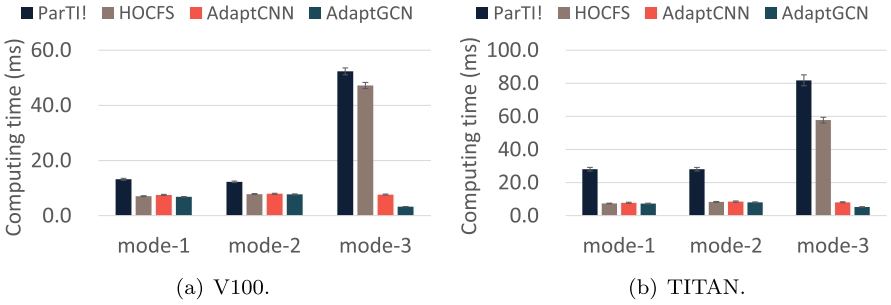


Fig. 9. Average execution time for SpTV on Mov.

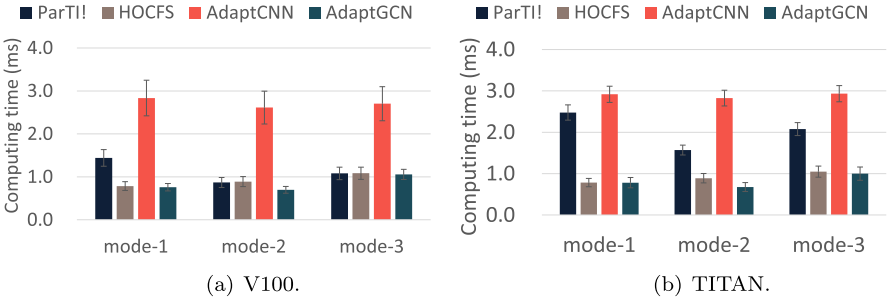


Fig. 10. Average execution time for SpTV on Het.

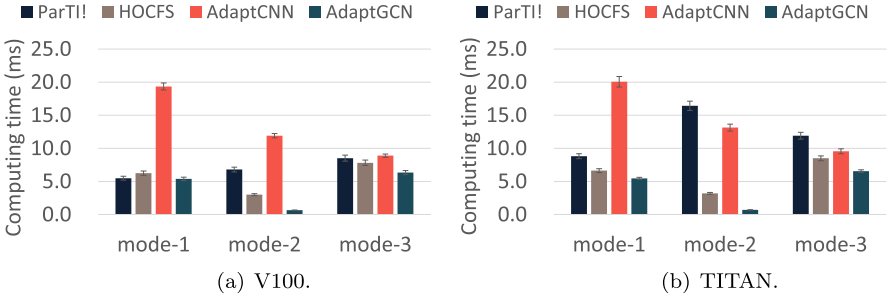


Fig. 11. Average execution time for SpTV on Ube.

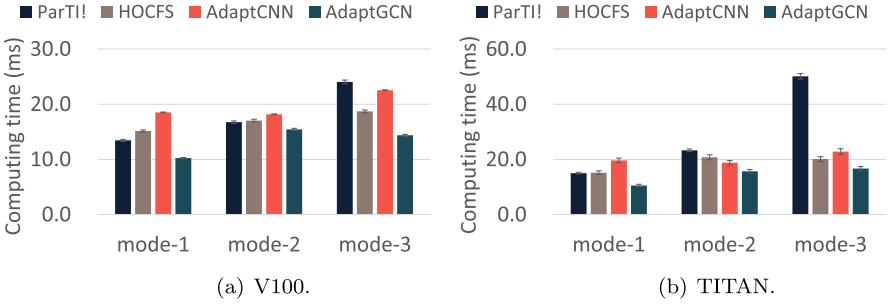


Fig. 12. Average execution time for SpTV on Chi.

pipeline technique is denoted as AdaptCNN. For a fairer comparison, the CNN in the experiments is obtained by re-training using the same train set as Slice-GCN. Arbitrarily sized matrices are scaled into fixed size matrices using statistical methods and then fed into CNN.

Figs. 9, 10, 11, 12, and 13 are shown to give the results on Tesla V100 GPU and TITAN RTX GPU to verify the portability of our proposed method. (1) Compared to the execution time of ParTI!, AdaptGCN achieves significant performance gains over most tests. The reason is that ParTI! implements fine-grained SpTV in a non-zero element-wise manner but executes only on the GPU default stream, which is synchronous, so it has more execution time than AdaptGCN. (2) Compared to the execution time of HOCFS, AdaptGCN improves performance over almost half of the tests. The reason is that HOCFS only distinguishes between CSR and ELL formats for slices and cannot improve the performance of slices whose optimal formats are COO, HYB, and Dense. (3) Compared to the execution time of AdaptCNN, AdaptGCN shows significant performance improvement in some of the tests. The reason for this is that the format chosen for the slices using CNN is often not optimal, and the unsuitable slicing format leads to a significant increase in the overall execution time of SpTV. Moreover, since the matrix size of the input to the convolutional neural network is fixed, a scaling operation is performed on the slices before they

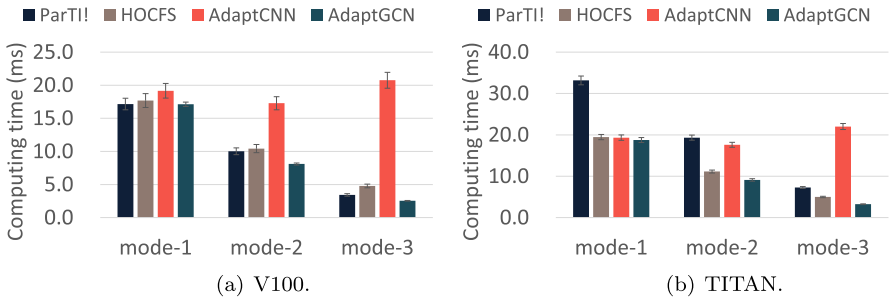


Fig. 13. Average execution time for SpTV on Nip.

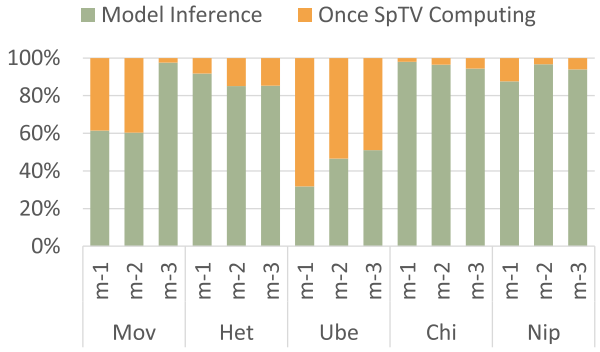


Fig. 14. The cost time of once SpTV computing compared with the model inference for sparse formats.

are fed into the convolutional neural network. The scaling operation loses the locally sparse features of the slices [41]. In contrast, the sparse matrix can be converted into a bipartite graph for input into the graph neural network without scaling. Therefore, graph neural networks are more suitable than convolutional neural networks for the sparse format selection of slices. (4) Besides, the computational performance of the different methods for executing SpTV on the V100 and TITAN platforms differs due to the different CPU and GPU processors of the two platforms. Further, the computational performance of all SpTV tests was better on the V100 platform than on the TITAN platform, which is also consistent with the difference in the two hardware configurations.

Furthermore, ParTI! uses only a single format, so the format selection is not required. Compared with HOCFS, which selects slice formats based on the threshold of non-empty fiber density [38], our designed selection method has better adaptability. Both HOCFS and our proposed IAP-SpTV approach use hybrid storage formats, so both have to pre-process overhead. The pre-processing overhead is the time spent selecting and converting the slices to suitable formats. In practical applications such as tensor decomposition, SpTV is heavily iterated, and the pre-processing execution time can be cost-shared by these iterations. Specifically, we compare the inference time of the model with the results of one actual SpTV execution time, as shown in Fig. 14. It can be seen that the inference time is more than the actual SpTV execution time if the SpTV is executed only once. Moreover, we repeated the execution of SpTV to determine the number of iterations needed for our method to perform better than the other methods. As shown in Fig. 15, the extra inference time of our method compared to PARTI! typically requires 100 iterations to amortize. The comparison with HOCFS generally requires 200 iterations of amortization.

Overall, in most cases, we can see that the AdaptGCN method outperforms the other methods. Specifically, the AdaptGCN method improves computing performance over the ParTI! library by 58.42% on average and over HOCFS by 24.85% on average on the TITAN CPU-GPU platform. And the AdaptGCN method improves computing performance over the ParTI! library by 32.23% on average and over HOCFS by 24.87% on average on the V100 CPU-GPU platform.

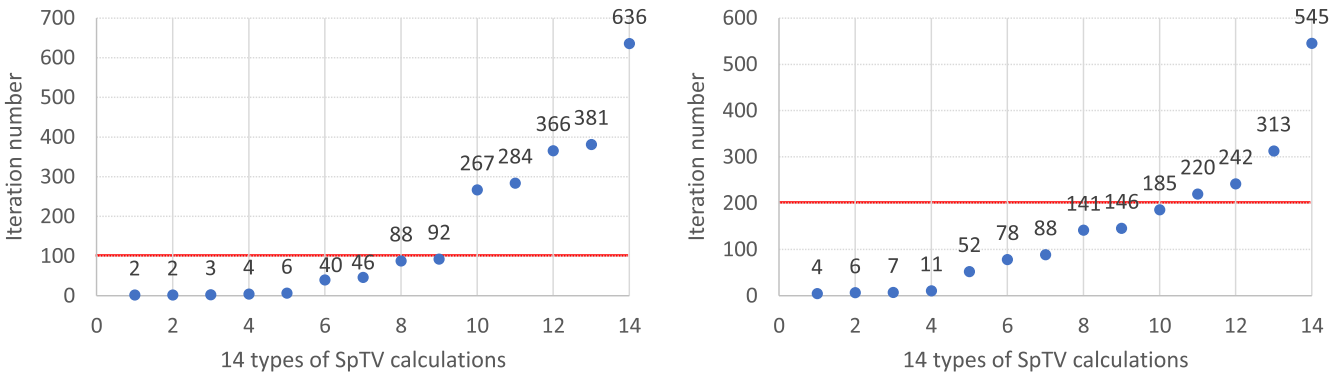
6.5. Analysis of results

To evaluate our proposed method more comprehensively, we first show the execution time of each format of HTF, as shown in Fig. 16 (a) and (b). The m-1, m-2, and m-3 are abbreviations of mode-1, mode-2, and mode-3. From the figures, we can see that CSR is the main format for slicing, followed by ELL. This shows the rationality of choosing CSR and ELL formats as the main computational formats for HOCFS. And, COO, HYB, and Dense also have a certain proportion for the calculation, which is an important reason for the performance improvement of our method relative to the HOCFS method. In particular, most of the mode-3 of Mov is computed in Dense and ELL formats, which can be seen in Fig. 9 (a) and (b) that the performance improvement of AdaptGCN over HOCFS on mode-3 of Mov is significant.

Second, we compare the differences between CNN and Slice-GCN by the M-format during the format selection phase. If the prediction format of Slice-GCN is 1, and the prediction format of CNN is 0, the M-format difference is 1. If the format is the same, the M-format difference is 0, and so on. We compare the differences in prediction results between CNN and Slice-GCN on different platforms. On the V100 platform, there are differences in the prediction results for 15.86% of the slices, while on the TITAN platform, there are differences in the prediction results for 17.64% of the slices. These differences in the prediction results of CNN and Slice-GCN affect the performance of the subsequent execution of SpTV, as demonstrated in the comparison of AdaptCNN and AdaptGCN in Figs. 9, 10, 11, 12, and 13.

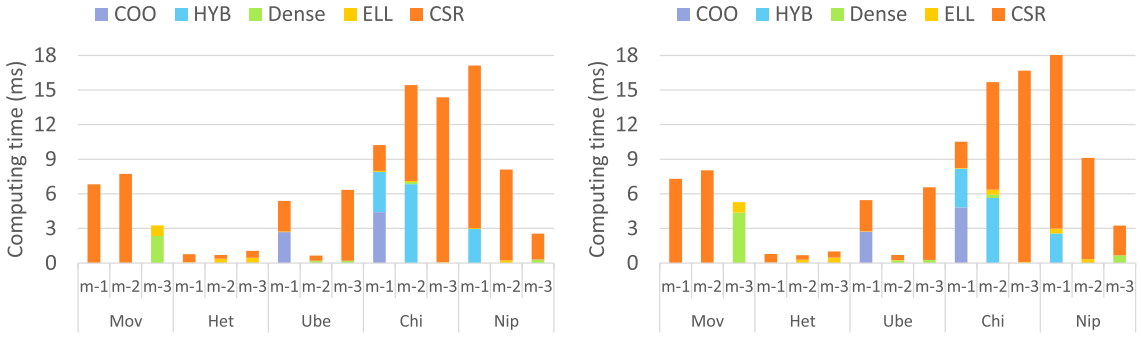
Third, we show the variation in the number of CUDA streams during the process of computation for the adaptive pipeline method, as shown in Fig. 17 (a) and (b). There are variations in the number of CUDA streams created to compute slice sets in the corresponding formats in all modes. Hence, the adaptive pipeline technique is effective for the HTF hybrid format. And, by comparing the number of streams for the same dataset and the same mode on V100 and TITAN platforms, it is seen that the number of streams varies due to the different hardware configurations of the platforms. Therefore, for different hardware platforms, our method can adaptively find a suitable number of streams.

Last, both our approach and the benchmark approach exhibit a low performance of SpTV as a percentage of peak GPU FP32 performance due to two main reasons. Firstly, in addition to considering the computation time, we also take into account the data transfer time between the CPU and GPU, so the overall performance is lowered. Secondly, SpTV is a memory-bound operator. The high-dimensional and intricate sparse structure of the sparse tensor introduces additional memory access overhead, contributing to the lower percentage of peak GPU FP32 performance observed with SpTV. In light of these findings, we plan to conduct further



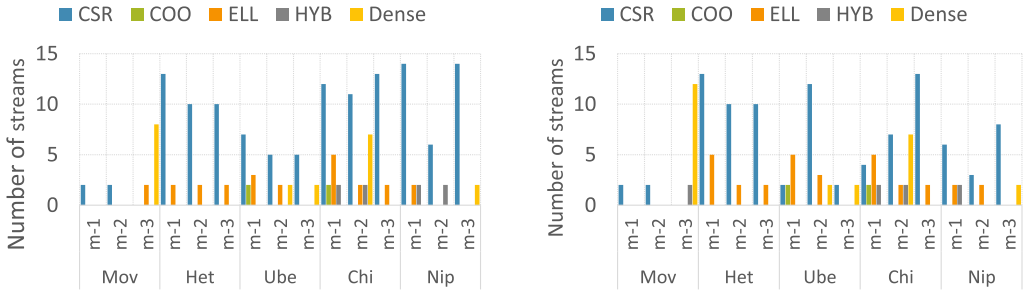
(a) Compared to ParTI!. (b) Compared to HOCFS.

Fig. 15. The number of iterations required to amortize the model inference time for the different types of SpTV calculations. The SpTV calculation types are derived from the SpTV calculations under three modes of the five tensors: Mov, Het, Ube, Chi, and Nip, resulting in a total of 15 types. In subfigure (a), the number of iterations of SpTV for the Nip tensor m-1 of type is 3872, which is beyond the range of the coordinate axes. Therefore, only the SpTV calculations for the other 14 types are presented in subfigure (a). Similarly, in subfigure (b), the number of iterations of SpTV for the Het tensor m-1 of type is 1244, which is beyond the range of the coordinate axes. Thus, subfigure (b) includes the SpTV calculations for the other 14 types.



(a) V100. (b) TITAN.

Fig. 16. The average execution time for each format of HTF.



(a) Experimental results on V100. (b) Experimental results on TITAN.

Fig. 17. Number of streams for different formats of the adaptive pipeline method.

research to enhance the memory access efficiency of sparse tensors in our future work.

7. Related work

With the popularity of big data, the research of large-scale tensors and stream tensors is becoming diversified. With the widespread use of tensors, designing efficient tensor multiplication is a problem worth exploring. Yang et al. [39] designed a pipeline calculation method for third-order SpTV on CPU and GPU. Chen et al. [5] proposed aeSpTV, an adaptive and efficient SpTV framework on Sunway TaihuLight supercomputer, to solve several challenges in optimizing SpTV on high-performance computing platforms. Zheng et al. [42] designed an automatic optimization framework for tensor computation on heterogeneous systems, and used heuristics to optimize the tensor computation program. Besides SpTV, the widely used tensor multiplication operations are Tensor-Times-Matrix (TTM) and Matricized-Tensor-Times-Khatri-Rao-Product (MTTKRP) [6,17,24,25,28,30], which are computationally equivalent to multiple TTVs.

Identifying a suitable sparse computation format to improve the efficiency of parallel computation is a classical research problem, which has a new solution with the development of deep learning. Elafrou et al. [7] explored the distribution of non-zero elements of the matrix and designed methods for predicting performance for SpMV. Yang et al. [38] developed an optimal partitioning strategy based on dynamic programming and distribution functions of non-zero elements to improve the performance of SpMV. Merrill et al. [22] conceived a strictly balanced method for solving SpMV, which

provides predictable performance and is essentially independent of the non-zero distribution between rows. Akrem et al. [2] demonstrated the effectiveness of machine learning for sparse matrix format selection. Zhao et al. [41] and Sun et al. [34,35] revealed the gap between CNN and sparse matrix format selection and designed a CNN structure that delayed the sparsity information to the later training process. Niu et al. [26] devised a fine-grained selection method to find the best format for each tile in the matrix.

8. Conclusion

In this paper, IAP-SpTV, an adaptive pipeline SpTV method based on CPU-GPU is proposed. More specifically, we first designed a hybrid format HTF and then described the HTF-based Pipelined SpTV algorithm. Second, we implemented Slice-GCN for selecting suitable formats for slices. Third, we constructed a performance analysis model for IAP-SpTV. Finally, we demonstrated the correctness, effectiveness, and portability of our approach experimentally.

Besides accelerating IAP-SpTV, our input-aware format selection method is a useful inspiration for the study of hybrid formats, and our performance model provides guidance for the design of heterogeneous pipeline algorithms on CPU-GPU. In future work, we will continue to explore the design and optimization of fine-grained pipeline parallelism.

CRediT authorship contribution statement

Haotian Wang: Conceptualization, Formal analysis, Methodology, Software, Writing – original draft. **Wangdong Yang:** Conceptualization, Funding acquisition, Investigation, Project administration, Supervision. **Rong Hu:** Data curation, Software. **Renqiu Ouyang:** Visualization, Writing – original draft. **Kenli Li:** Resources, Supervision. **Keqin Li:** Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The authors are unable or have chosen not to specify which data has been used.

Acknowledgments

The author sincerely thanks the editors and all the anonymous reviewers for their comments, which are valuable and constructive. The research was partially funded by the Key-Area Research and Development Program of Guangdong Province (Grant no. 2021B0101190004), the Key Program of the National Natural Science Foundation of China (Grant nos. U21A20461, 92055213), the National Key R&D Program of China (Grant no. 2021YFB0300800), and the National Natural Science Foundation of China (Grant no. 61872127).

References

[1] N. Bell, M. Garland, Implementing sparse matrix-vector multiplication on throughput-oriented processors, in: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, 2009, pp. 1–11.
[2] A. Benatia, W. Ji, Y. Wang, F. Shi, Sparse matrix format selection with multiclass SVM for SpMV on GPU, in: 2016 45th International Conference on Parallel Processing, ICPP, 2016, pp. 496–505.
[3] A. Benoit, Y. Robert, Mapping pipeline skeletons onto heterogeneous platforms, J. Parallel Distrib. Comput. 68 (2008) 790–808.
[4] I. Cantador, P. Brusilovsky, T. Kuflik, 2nd workshop on information heterogeneity and fusion in recommender systems (HetRec 2011), in: Proceedings of the 5th ACM Conference on Recommender Systems, RecSys 2011, ACM, New York, NY, USA, 2011.
[5] Y. Chen, G. Xiao, M.T. Özsu, C. Liu, A.Y. Zomaya, T. Li, aeSpTV: an adaptive and efficient framework for sparse tensor-vector product kernel on a high-performance computing platform, IEEE Trans. Parallel Distrib. Syst. 31 (2020) 2329–2345.
[6] Y. Chen, G. Xiao, M.T. Özsu, Z. Tang, A.Y. Zomaya, K. Li, Exploiting hierarchical parallelism and reusability in tensor kernel processing on heterogeneous HPC systems, in: 2022 IEEE 38th International Conference on Data Engineering, ICDE, 2022, pp. 2523–2536.
[7] A. Elafrou, G. Goumas, N. Koziris, Performance analysis and optimization of sparse matrix-vector multiplication on modern multi- and many-core processors, in: 2017 46th International Conference on Parallel Processing, ICPP, 2017, pp. 292–301.
[8] M. Fey, J.E. Lenssen, Fast graph representation learning with PyTorch Geometric, in: ICLR Workshop on Representation Learning on Graphs and Manifolds, 2019.
[9] M. Garland, S.M.L. Grand, J.R. Nickolls, J. Anderson, J. Hardwick, S.A. Morton, E.H. Phillips, Y. Zhang, V. Volkov, Parallel computing experiences with CUDA, IEEE Micro 28 (2008).
[10] A. Globerson, G. Chechik, F.C. Pereira, N. Tishby, Euclidean embedding of co-occurrence data, J. Mach. Learn. Res. (2004).
[11] F.M. Harper, J.A. Konstan, The MovieLens datasets: history and context, ACM Trans. Interact. Intell. Syst. 5 (4) (Dec. 2015).
[12] B. Hendrickson, T.G. Kolda, Partitioning rectangular and structurally unsymmetric sparse matrices for parallel processing, SIAM J. Sci. Comput. 21 (2000) 2048–2072.
[13] T. Kolda, B. Bader, Tensor decompositions and applications, SIAM Rev. 51 (2009) 455–500.
[14] T.G. Kolda, B.W. Bader, Matlab Tensor Toolbox, 2006.
[15] A.J. Kurdila, M. Zabarankin, Convex Functional Analysis, Springer Science & Business Media, 2006.
[16] D. Lee, K. Shin, Robust factorization of real-world tensor streams with patterns, missing values, and outliers, in: 2020 IEEE 36th International Conference on Data Engineering, ICDE, 2021.
[17] J. Li, J. Sun, R. Vuduc, HiCOO: hierarchical storage of sparse tensors, in: SC18: International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE, 2018, pp. 238–252.
[18] J. Li, Y. Ma, R. Vuduc, ParTI!: A Parallel Tensor Infrastructure for Multicore CPUs and GPUs, Oct 2018, last updated: Jan 2020.
[19] K. Li, W. Yang, K. Li, Performance analysis and optimization for SpMV on GPU using probabilistic modeling, IEEE Trans. Parallel Distrib. Syst. 26 (2015) 196–205.
[20] S. Lin, W. Yang, H. Wang, Q. Tsai, K. Li, STM-multifrontal QR: streaming task mapping multifrontal QR factorization empowered by GCN, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2021.
[21] Y. Ma, J. Li, X. Wu, C. Yan, J. Sun, R. Vuduc, Optimizing sparse tensor times matrix on GPUs, J. Parallel Distrib. Comput. 129 (2019) 99–109.
[22] D. Merrill, M. Garland, Merge-based parallel sparse matrix-vector multiplication, in: SC16: International Conference for High Performance Computing, Networking, Storage and Analysis, 2016, pp. 678–689.
[23] C. Morris, M. Ritzert, M. Fey, W.L. Hamilton, J.E. Lenssen, G. Rattan, M. Grohe, Weisfeiler and Leman go neural: higher-order graph neural networks, in: AAAI, 2019.
[24] A.K. Nguyen, A. Helal, F. Checconi, J. Laukemann, J.J. Tithi, Y. Soh, T.M. Ranadive, F. Petrini, J.W. Choi, Efficient, out-of-memory sparse MTTKRP on massively parallel architectures, in: Proceedings of the 36th ACM International Conference on Supercomputing, 2022.
[25] I. Nisa, J. Li, A. Sukumaran-Rajam, R.W. Vuduc, P. Sadayappan, Load-balanced sparse MTTKRP on GPUs, in: 2019 IEEE International Parallel and Distributed Processing Symposium, IPDPS, 2019, pp. 123–133.
[26] Y. Niu, Z. Lu, M. Dong, Z. Jin, W. Liu, G. Tan, TileSpMV: a tiled algorithm for sparse matrix-vector multiplication on GPUs, in: 2021 IEEE International Parallel and Distributed Processing Symposium, IPDPS, 2021, pp. 68–78.
[27] Y. Niu, Z. Lu, H. Ji, S. Song, Z. Jin, W. Liu, TileSpGEMM: a tiled algorithm for parallel sparse general matrix-matrix multiplication on GPUs, in: Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2022.
[28] E. Qin, G. Jeong, W. Won, S.-C. Kao, H. Kwon, S.M. Srinivasan, D. Das, G.E. Moon, S. Rajamanickam, T. Krishna, Extending sparse tensor accelerators to support multiple compression formats, in: 2021 IEEE International Parallel and Distributed Processing Symposium, IPDPS, 2021, pp. 1014–1024.
[29] H. Ramchoun, M.A.J. Idrissi, Y. Ghanou, M. Ettaouil, Multilayer perceptron: architecture optimization and training, Int. J. Interact. Multim. Artif. Intell. 4 (2016) 26–30.
[30] S. Smith, G. Karypis, Tensor-matrix products with a compressed sparse tensor, in: Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms, 2015, pp. 1–7.

[31] S. Smith, N. Ravindran, N. Sidiropoulos, G. Karypis, SpLATT: efficient and parallel sparse tensor-matrix multiplication, in: 2015 IEEE International Parallel and Distributed Processing Symposium, 2015, pp. 61–70.
[32] S. Smith, J.W. Choi, J. Li, R. Vuduc, J. Park, X. Liu, G. Karypis, FROSTT: The Formidable Repository of Open Sparse Tensors and Tools, 2017.
[33] N. Srivastava, H. Jin, S. Smith, H. Rong, D.H. Albonesi, Z. Zhang, Tensaurus: a versatile accelerator for mixed sparse-dense tensor computations, in: 2020 IEEE International Symposium on High Performance Computer Architecture, HPCA, 2020, pp. 689–702.
[34] Q. Sun, Y. Liu, M. Dun, H. Yang, Z. Luan, L. Gan, G. Yang, D. Qian, SpTFS: sparse tensor format selection for MTTKRP via deep learning, in: SC20: International Conference for High Performance Computing, Networking, Storage and Analysis, 2020, pp. 1–14.
[35] Q. Sun, Y. Liu, H. Yang, M. Dun, Z. Luan, L. Gan, G. Yang, D. Qian, Input-aware sparse tensor storage format selection for optimizing MTTKRP, IEEE Trans. Comput. (2021).
[36] R.W. Vuduc, J. Demmel, Automatic Performance Tuning of Sparse Matrix Kernels, 2003.
[37] H. Wang, W. Yang, R. Ouyang, R. Hu, K. Li, K. Li, A heterogeneous parallel computing approach optimizing SpTTM on CPU-GPU via GCN, ACM Trans. Parallel Comput. 10 (2) (2023).
[38] W. Yang, K. Li, K. Li, A parallel computing method using blocked format with optimal partitioning for SpMV on GPU, J. Comput. Syst. Sci. 92 (2018) 152–170.
[39] W. Yang, K. Li, K. Li, A pipeline computing method of SpTV for three-order tensors on CPU and GPU, ACM Trans. Knowl. Discov. Data 13 (2019) 1–27.
[40] Y. Zhang, W. Yang, K. Li, D. Tang, K. Li, Performance analysis and optimization for SpMV based on aligned storage formats on an arm processor, J. Parallel Distrib. Comput. 158 (2021) 126–137.
[41] Y. Zhao, J. Li, C. Liao, X. Shen, Bridging the gap between deep learning and sparse matrix format selection, in: Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2018.
[42] S. Zheng, Y. Liang, S. Wang, R. Chen, K. Sheng, Flextensor: an automatic schedule exploration and optimization framework for tensor computation on heterogeneous system, in: Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, 2020.

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

Wangdong Yang received the Ph.D. degree in computer science from Hunan University, China, and the M.S. degree in computer science from Central South University, China. He is a professor of computer science and technology at 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. He is currently served on the editorial boards of IEEE Internet of Things Journal.

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

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

Kenli Li received the Ph.D. degree in computer science from Huazhong University of Science and Technology, China, in 2003 and the M.S. degree in mathematics from Central South University, China, in 2000. He was a visiting scholar at University of Illinois at Urbana-Champaign from 2004 to 2005. He is a full professor of computer science and technology at 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 served on the editorial boards of IEEE Transactions on Computers. He is an outstanding member of CCF and a member of the IEEE.

Dr. Keqin Li 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 780 journal articles, book chapters, and refereed conference papers, and has received several best paper awards. He holds over 60 patents announced or authorized by the Chinese National Intellectual Property Administration. He is among the world's top 10 most influential scientists in distributed computing 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 IEEE Fellow.