Input-Aware Sparse Tensor Storage Format Selection for Optimizing MTTKRP

Qingxiao Sun, Yi Liu, Hailong Yang, Ming Dun, Zhongzhi Luan, Lin Gan, *Member, IEEE*, Guangwen Yang, *Member, IEEE*, and Depei Qian

Abstract—Canonical polyadic decomposition (CPD) is one of the most common tensor computations adopted in many scientific applications. The major bottleneck of CPD is matricized tensor times Khatri-Rao product (MTTKRP). To optimize the performance of MTTKRP, various sparse tensor formats have been proposed such as CSF and HiCOO. However, due to the spatial complexity of the tensors, no single format fits all tensors. To address this problem, we propose *SpTFS*, a framework that automatically predicts the optimal storage format for an input sparse tensor. Specifically, *SpTFS* leverages a set of sampling methods to lower the sparse tensor to fixed-sized matrices and sparsity features. In addition, *SpTFS* adopts both supervised learning based and unsupervised learning based methods to predict the optimal sparse tensor storage formats. For supervised learning, we propose *TnsNet* that combines convolution neural network (CNN) and the feature layer, which effectively captures the sparsity patterns of the input tensors. Once trained, the *TnsNet* can be used with either density or histogram representation of the input tensor for optimal format prediction. Whereas for unsupervised learning, we propose *TnsClustering* that consists of a feature encoder using convolutional layers and fully connected layers, and a *K-means++* model to cluster sparse tensors for optimal tensor format prediction, without massively profiling on the hardware platform. *SpTFS* can use the above two models to predict the optimal tensor storage format for accelerating MTTKRP accurately. The experimental results show that both *TnsNet* and *TnsClustering* can achieve higher prediction accuracy and performance speedup compared to the state-of-the-art works.

Index Terms—Tensor computation, MTTKRP, sparse tensor storage format, convolutional neural network, convolutional autoencoder

1 Introduction

TENSORS can represent high dimensional data with more than two dimensions. Multi-dimensional tensors are commonly used in the fields of scientific computing [1] and numerical analysis [2]. In the meanwhile, tensor decomposition is widely used to understand the relationship of data across multiple dimensions. The concept of tensor decomposition first appeared in the psychometric literature, and later became popular in the field of chemometrics [3]. In recent years, tensor decomposition has received wide attention due to its applicability in broader areas such as neuroscience, recommendation systems, and machine learning [4].

Canonical polyadic decomposition (CPD) [5] is one of the most popular tensor decomposition techniques. CPD is a generalization of singular value decomposition and outputs matrix factors for each mode (a.k.a, dimension) of a tensor. The major performance bottleneck of CPD is matricized tensor times Khatri-Rao product (MTTKRP) [6], which is the primary focus of optimizations in tensor composition. Since real-world tensors are usually large and extremely sparse, many existing works optimize the performance of MTTKRP based on the computation patterns and operation dependency [7].

Although the parallelization can significantly improve the performance of MTTKRP, it is constrained by the sparsity patterns and hardware characteristics. Therefore, different sparse tensor formats have been proposed to improve the computation performance with co-designed storage and algorithm adapt to the sparsity and hardware. Coordinate (COO) [8] is a simple but popular sparse tensor format in which each non-zero value is stored with the indices of all dimensions. Compressed Sparse Fiber (CSF) [9] uses a tree structure to store non-zero values and their index pointers, similar to Compressed Sparse Row (CSR) [10] in matrices. In addition, hardware-specific extensions based on COO and CSF formats have been proposed, such as HiCOO and MM-CSF [11], [12], [13], [14]. However, due to the complex sparsity patterns and diverse hardware characteristics, the optimal tensor format for MTTKRP varies significantly. Therefore, it is challenging to determine the optimal tensor format for MTTKRP running with different tensor inputs on different hardware platforms.

The format selection of sparse tensors can be analogized to the classification problem. For programmers, choosing the optimal format is a daunting task with tedious efforts. Although traditional machine learning methods (e.g.,

- *Qingxiao Sun, Yi Liu, Hailong Yang, Zhongzhi Luan, and Depei Qian are with the School of Computer Science and Engineering, Beihang University, Beijing 100191, China. E-mail: {qingxiaosun, yi.liu, hailong.yang, 07680, depeiq}@buaa.edu.cn.*
- *Ming Dun is with the School of Cyber Science and Technology, Beihang University, Beijing 100191, China. E-mail: dunming0301@buaa.edu.cn.*
- *Lin Gan and Guangwen Yang are with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China. E-mail: {lingan, ygw}@tsinghua.edu.cn.*

Manuscript received 28 Apr. 2021; revised 13 Aug. 2021; accepted 4 Sept. 2021. Date of publication 16 Sept. 2021; date of current version 11 July 2022.
This work was supported by National Key Research and Development Program of China under Grant 2020YFB1506703, and National Natural Science Foundation of China under Grant 62072018.
(Corresponding author: Hailong Yang.)
Recommended for acceptance by L. A. Sousa.
Digital Object Identifier no. 10.1109/TC.2021.3113028

decision tree [15]) are relatively easy to implement, they obtain low accuracy due to the lack of sparsity distribution information [16], [17]. In contrast, the convolutional neural network (CNN) has gained tremendous popularity in classification tasks due to its ability to capture the underlying features of input data without human engineering [18], [19]. Related to this study, previous works applied CNN to sparse matrix format selection for optimizing SpMV [20], [21] and SpGEMM [22]. However, such approaches cannot be directly applied in tensor format selection due to higher dimensional data to deal with. The high-dimensional convolution can neither be used in our work due to the unacceptable prediction overhead caused by the tensor irregularity. In addition, both TensorFlow [23] and PyTorch [24] can only support tensor computations (e.g., 3-D convolution) up to three dimensional, which cannot satisfy the need for higher-order tensors. To apply two-dimensional convolution to tensor format selection, we need to transform any-order tensors into fixed-sized matrices without losing the sparsity patterns. Furthermore, the CNN network needs to be re-designed to compensate for the missing sparsity features during tensor transformation.

Unlike supervised learning based methods, unsupervised learning based methods only require unlabeled training data, which can significantly reduce engineering efforts. Among them, convolutional autoencoder (CAE) [25] have gained attention in classification tasks due to its ability to effectively extract the pixel distribution of an image as a feature vector. After that, the feature vectors are clustered using traditional machine learning methods (e.g., K-means [26]) to achieve image classification. However, the same challenges faced by CNN also apply to CAE. In order to achieve the automatic tensor format selection, the CAE also needs to be re-designed to better learn the sparsity patterns of the input tensor. Moreover, a holistic pipeline including the autoencoder and clustering algorithm needs to be designed to predict the optimal storage format for sparse tensors.

To address the above challenges, we propose an automatic tensor format selection framework *SpTFS*, that effectively predicts the optimal format for an input tensor running MTTKRP on a particular hardware platform. The *SpTFS* first lowers the high dimensional tensors into two-dimensional matrices and then represents the matrices as fixed-size input suitable for the CNN and CAE networks through various scaling methods. For supervised learning, we re-design the CNN network by adding an additional feature layer to compensate for the sparsity features lost during matrix representation. For unsupervised learning, we re-design the CAE by adding the fully connected layers to better extract the spatial distribution of fixed-sized matrices. After that, the encoding outputs are concatenated with sparsity features for K-means++ algorithm to obtain the clusters of sparse tensors. We evaluate *SpTFS* on both CPU and GPU platforms to prove its effectiveness in predicting optimal tensor format.

This paper is an extension of our previous work [27]. Compared to [27], we further implement a holistic pipeline of optimal tensor format prediction named *TnsClustering*, which can effectively predict the optimal tensor format without massively profiling on the hardware platform. To the best of our knowledge, this is the first work to utilize unsupervised learning based method to predict the optimal storage format for sparse tensor computation. The *SpTFS* is open-sourced at https://github.com/sunqingxiao/SpTFS . Specifically, this paper makes the following contributions:

- We comprehensively analyze the impact of sparse tensor format selection on the performance of MTTKRP due to the complex sparsity patterns and diverse hardware characteristics.
- We propose a tensor transformation mechanism that first lowers a tensor into matrices and then represents the matrices to fixed-size inputs to CNN through two different scaling methods.
- We design and implement *TnsNet* that combines CNN and feedforward neural network (FFNN) to obtain better prediction accuracy. TnsNet integrates an additional feature layer that compensates for the sparsity features lost during tensor transformation.
- We design and implement *TnsClustering* that includes *TnsEncoder* and *K-means++* modeling. *TnsEncoder* combines CAE and stacked autoencoder (SAE) to encode the feature vectors of matrices after tensor transformation. *K-means++* modeling utilizes the encoding outputs from *TnsEncoder* concatenated with sparsity features to cluster sparse tensors without massively profiling on the hardware platform.
- We develop an automatic sparse tensor format selection framework *SpTFS* that effectively predicts the optimal format for input tensor data running MTTKRP on different hardware platforms. Experiment results show that *SpTFS* can achieve high prediction accuracy and thus significant speedup for MTTKRP.

The rest of this paper is organized as follows: Section 2 presents the background of this paper. Section 3 presents the details of *SpTFS* methodology. Sections 4 and 5 present the evaluation results of *SpTFS*. Section 6 discusses the related work, and Section 7 concludes this paper.

2 Background

In this section, we present tensor notations and widely used sparse tensor storage formats. Then, we illustrate popular machine learning techniques used for supervised and unsupervised classification tasks. The above background motivates our work in this paper.

2.1 Tensor Notation

Tensor denotes the array with multiple dimensions [3] and is the generalization of matrix and vector. A high-order tensor refers to the tensor with more than two dimensions, and the mode-n of a tensor denotes its n^{th} dimension. High-order tensors have been widely used in the fields of signal processing, chemometrics and image/video rendering. Since finding the exact rank of a tensor is an NP-hard problem [28], researchers pay the most attention to ranks that are less than the longest dimension of a sparse tensor. Here, we use the three-dimensional tensor and mode-1 computation to describe the concepts and related mathematics about tensor decomposition without losing generality. All

TABLE 1
Important Tensor Notations

Notation	Definition
$\mathcal{X}$	A high-dimensional tensor.
N	Tensor order.
I, J, K, I_n	Tensor mode sizes.
$\mathcal{X}_{(n)}$	Matricized tensor in mode-n.
$\mathcal{X}(i,j,k)$	An element in a high dimensional tensor.
$\mathcal{X}(i,:,:)$	A slice in a high dimensional tensor.
$\mathcal{X}(i,j,:)$	A fiber in a high dimensional tensor.
$\mathbf{A}$	A matrix.
$\mathbf{A}(i,j)$	An element in a matrix.
$\mathbf{a}$	An vector.
$\mathbf{a}_i$	An element in a vector.
$\odot$	The symbol for Kronecker product.
$*$	The symbol for Hadamard product.
$\dagger$	The symbol for pesudo-inverse.

notations for vectors, matrices, and high-dimensional tensors are shown in Table 1, where a slice denotes the subarray with one index of the tensor fixed, and a fiber denotes the subarray with two indices of the tensor fixed.

Canonical polyadic decomposition (CPD) [5] is one of the most widely-applied tensor computations, which decomposes a tensor $\mathcal{X}$ with rank F into the summation of F rank-one tensors, and the rank-one tensors can be represented as the outer products of vectors. In other words, the CPD models a tensor $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$ with three factor matrices $\mathbf{A} \in \mathbb{R}^{I \times F}$, $\mathbf{B} \in \mathbb{R}^{J \times F}$ and $\mathbf{C} \in \mathbb{R}^{K \times F}$ as formulated in Eq. (1). To solve the CPD, one of the most popular approaches is to utilize Alternating Least Squares (ALS) [3], where a least square problem for each factor matrix is solved iteratively with others fixed. The update process for factor matrix $\mathbf{A}$ is shown in Eq. (2). The main bottleneck in the CPD-ALS algorithm for sparse tensors is MTTKRP [6], which can be formulated as Eq. (3).

$$\mathcal{X}(i,j,k) = \sum_{f=1}^{F} \mathbf{A}(i,f)\mathbf{B}(j,f)\mathbf{C}(k,f) \tag{1}$$

$$\mathbf{A} = \mathcal{X}_{(1)}(\mathbf{B} \odot \mathbf{C})(\mathbf{C}^T\mathbf{C} * \mathbf{B}^T\mathbf{B})^{\dagger} \tag{2}$$

$$\hat{\mathbf{A}} = \mathcal{X}_{(1)}(\mathbf{B} \odot \mathbf{C}). \tag{3}$$

2.2 Sparse Tensor Storage Formats

2.2.1 COO and COO-Based Formats

Coordinate format (COO) [8] is an intuitive format for storing sparse tensor (Fig. 1a). COO consists of tuples, and each tuple stores the indices and value for every nonzero element in the tensor. The MTTKRP algorithm using COO tensor format computes at the granularity of a single element. The advantage of COO is the simplicity and insensitivity of sparsity patterns in tensor. However, it requires a large memory footprint and relies on atomic operations when running on GPU. Therefore, the variants of COO have been proposed to overcome the above drawbacks, including F-COO [11] and HiCOO [12], as shown in Figs. 1b and 1c. F-COO adds two flag arrays named *start-flag* and *bit-flag*, which indicate whether the indices of slices vary at the beginning of a block and at an element, respectively. The two flag arrays are used to guide segment scan for avoiding atomic operations in MTTKRP on GPU. Meanwhile, HiCOO partitions each dimension into chunks with size B and compresses the nonzero tuples with fewer bits. The $bptr$ array stores where the block begins. The indices for every nonzero element can be computed using Eq. (4). Atomic operations in MTTKRP can be avoided through its privatization method. Moreover, HiCOO groups blocks into a large logical superblock with size L. To avoid write conflicts, the blocks within a superblock are always scheduled together and assigned to a single thread.

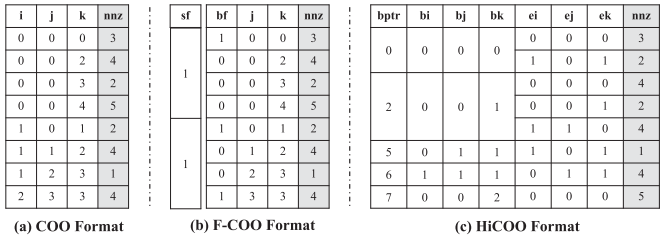


Fig. 1. The comparison of sparse tensor storage format COO and its variants, F-COO and HiCOO (parameter $B = 2$).

$$\begin{aligned} i &= bi * B + ei \\ j &= bj * B + ej \\ k &= bk * B + ek. \end{aligned} \tag{4}$$

2.2.2 CSF and CSF-Based Formats

Compressed Sparse Fiber (CSF) [6] extends the Compress Sparse Row (CSR) [10] format used to store sparse matrices. CSF maintains a tree-based structure and consists of six arrays, as shown in Fig. 2a. The i_ptr array stores the position of the first fibers of the slices, while the j_ptr array stores the position of the first elements of the fibers. Other arrays store the corresponding indices and values of elements. The CSF format is superior in less memory footprint and higher cache hit rate compared to COO, due to its compressed slices and fibers. Moreover, tiling can be used along the second mode to partition matrix factors, which are distributed among threads to eliminate the need for locks. However, the tree-based CSF requires the recursive algorithm when used in MTTKRP, which is not efficient when implemented on GPU.

To address the limitation of CSF, it is extended to HB-CSF [13] for better adaption on GPU, whose structure is shown in Fig. 2b. HB-CSF is a mixture of COO, Compressed Slice (CSL), and CSF. The COO is used when the slice only contains one element, whereas the CSL is applied when the slice contains multiple fibers, and each fiber contains a single element. Except for the above cases, the CSF format is adopted. HB-CSF improves the load balance and memory

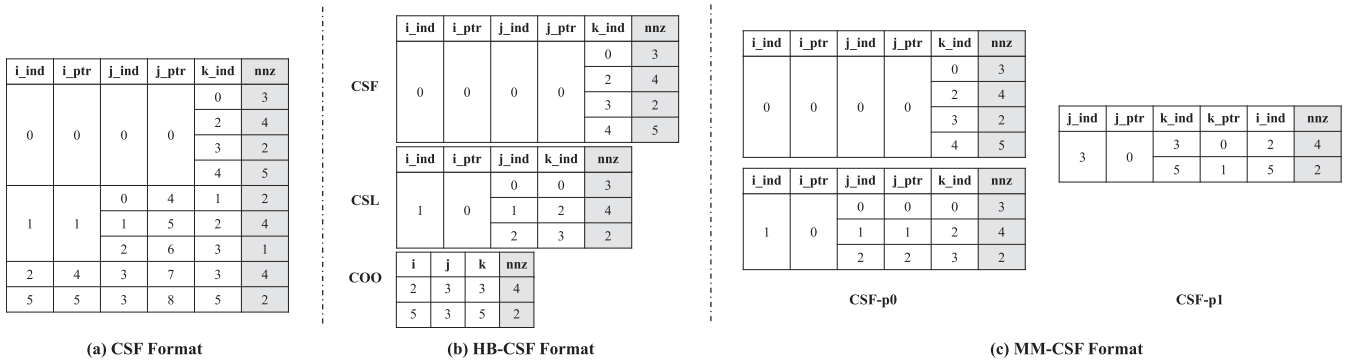


Fig. 2. The comparison of sparse tensor storage formats CSF and its variants, HB-CSF and MM-CSF. For MM-CSF, *p0* means that the slices are originally from CSF tensor for mode-1.

efficiency on GPU due to fine-grained tensor partition. Meanwhile, the original CSF format is mode sensitive as it is compressed along a certain mode. As a consequence, d CSF tensors are needed when conducting CPD of a $d-order$ tensor, whose memory footprint can be unacceptable in practice. Therefore, CSF format has been improved to MM-CSF [14] in order to reduce memory footprint. MM-CSF is a hybrid of multiple CSF tensors compressed along different modes to reach the best compression ratio (Fig. 2c). It adopts a dynamic partitioning scheme: Given a nonzero element $\mathcal{X}(i,j,k)$, it belongs to three fibers in the CSF tensors for three modes, and it is partitioned along the longest fiber, after that, the length of other two fibers it belongs to is reduced by one.

Fig. 3 presents the performance comparison of real-world sparse tensors using different storage formats for MTTKRP. The tensor datasets are adopted from FROSTT [29] and HaTen2 [30], including ten 3-D tensors and six 4-D tensors. The detailed descriptions of different tensors are provided in Section 4.1. Two observations can be drawn from Fig. 3: 1) the execution time of the same tensor in different formats varies significantly; 2) the optimal storage format changes across tensors, there is no single format that fits for all. In fact, the massive amount of real-world tensor data is prohibitive for selecting the optimal format through manual efforts. What is worse, the sparsity distribution of high-dimensional tensors is difficult to be described by traditional machine learning methods, and thus makes such methods less effective. In addition, the selection of a sub-optimal format may exponentially increase the execution time of MTTKRP. The above observations motivate us to predict the optimal storage format for tensors through deep learning methods automatically.

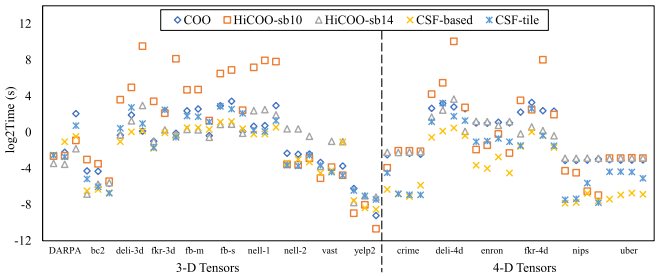


Fig. 3. The execution time of MTTKRP on real-world sparse tensors across all modes on CPU. The number of columns each tensor contains depends on its order.

2.3 Machine Learning-Based Techniques

Supervised learning has attracted lots of attention due to its outstanding performance on image classification tasks [18], [19], [31]. Among the traditional machine learning methods, gradient boosted decision tree (GBDT) [32] has been adopted in many application scenarios. XGBoost [33] is an efficient implementation of GBDT (based on CART [34]) that has been widely used in classification. XGBoost also supports gradient boosting linear classifier (GBLinear). Recently, deep learning methods have achieved great success in scenarios where the input contains non-linear patterns (e.g., images). Convolution neural network (CNN) is one of the most popular deep neural networks (DNNs) and is often used to realize classification or detection tasks. Note that the input size of CNN is typically fixed and equals to the number of nodes of the input layer. If the raw inputs have different sizes, they need to be scaled to a fixed size. During scaling, the features of raw inputs should be retained as much as possible to faithfully represent the patterns.

Supervised learning based methods usually require an enormous amount of labeled training data. In contrast, unsupervised learning based methods do not have such constraints, which greatly reduces the efforts of training data collection. Among unsupervised learning based methods, K-means clustering [26] is favored due to its ease of use. K-means++ [35] is an extension of K-means that improves the initialization of cluster centroids. K-means++ has been widely implemented by popular machine learning libraries (e.g., scikit-learn [36]). More recently, deep learning methods have made great progress in unsupervised image classification scenarios. Among them, convolutional autoencoders (CAE) [25] receive the corrupted image as input and is trained to predict the original uncorrupted image. The generative adversarial network (GAN) [37] learns the pixel distribution of images through its generative model and discriminative model. While the GAN-based structures have achieved impressive results in the field of unsupervised classification, the huge computational overhead is prohibitive in certain scenarios such as tensor format prediction. For example, based on our evaluation results the training and inference time of deep convolutional GAN (DCGAN) [38] is 4.64× and 2.30× larger than that of CAE. Therefore, we do not consider GAN-based approaches in this paper.

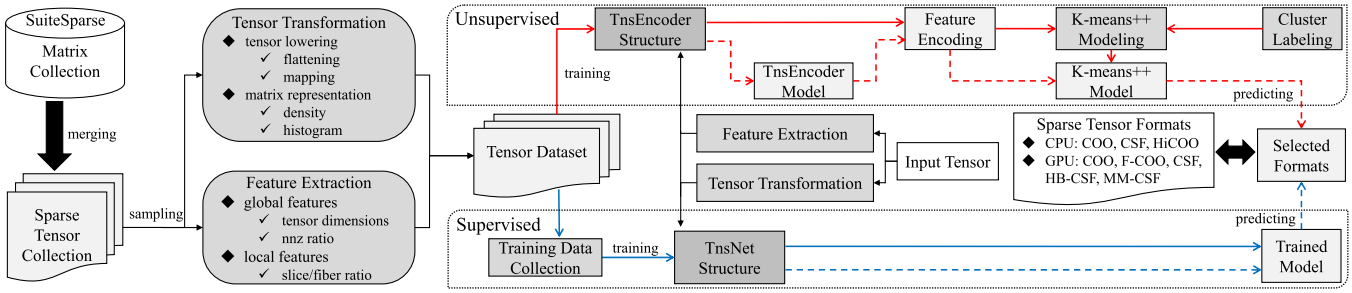


Fig. 4. The design overview of *SpTFS*.

3 Methodology

In this section, we begin with a high-level overview of the *SpTFS* design. After that, we present the key components of the tensor transformation module including tensor lowering and matrix representation. Then, we illustrate the supervised and unsupervised methods implemented by *SpTFS* to achieve the optimal tensor format prediction. Finally, we explain the overall workflow and other implementation details of the *SpTFS* framework.

3.1 Design Overview

In this section, we propose an automatic tensor format selection framework *SpTFS*, that can predict the optimal tensor format for MTTKRP with a re-designed CNN network. As shown in Fig. 4, the *SpTFS* sampling consists of two important components including tensor transformation and feature extraction. The tensor transformation component converts the sparse tensors into fixed-sized matrices through tensor lowering (Section 3.2) and matrix representation (Section 3.3). Inspired by [22], the feature extraction component is used to capture lost tensor features during tensor transformation, which is then fed into the fully connected layer in deep neural networks.

For supervised learning, the performance of each tensor input is profiled with different storage formats on the target hardware platform, and the tensor format with the best performance is labeled. The profiled datasets are then used to train the *TnsNet* (a re-designed CNN network), which predicts the optimal format for a tensor on the target hardware platform. For unsupervised learning, we propose *TnsClustering* that consists of three stages such as feature encoding (*TnsEncoder*), *K-means++* Modeling and cluster labeling. During training, the unlabeled tensor datasets can be directly used to train the *TnsEncoder* (a re-designed CAE) model for feature encoding. Since the training of *TnsEncoder* does not require profiling the performance of different sparse formats on a large volume of tensor datasets, once trained the model can be used across different platforms. After that, the *TnsClustering* uses the *K-means++* algorithm to cluster the input tensors based on their feature vectors. Finally, the *TnsClustering* determines the optimal tensor format for each cluster by profiling the optimal format for the cluster centroid tensor on the target hardware platform. During prediction, the *TnsClustering* obtains the feature vector of each input tensor through tensor transformation and feature encoding (using *TnsEncoder)*. Then, the input tensor is assigned to the nearest cluster by the trained *K-means++* model, with the optimal tensor format predicted the same as the cluster.

Due to the limited number of publicly available datasets of sparse tensors, we randomly select sparse matrices from the SuiteSparse [39] and combine them to generate more datasets of sparse tensors. SuiteSparse has been widely used in evaluating the performance of matrix computation [40] as well as the accuracy of matrix format selection [22]. For the 3-D tensors, we use the elements of the first matrix to form the higher two dimensions, and the elements along the higher indices of the second matrix to form the lowest dimension. Similarly, for the 4-D tensors, the elements of the first matrix and the second matrix form the higher two dimensions and the lower two dimensions, respectively. Then, the generated sparse tensor datasets are processed through tensor transformation and feature extraction in order to be used for training the CNN or CAE networks.

Note that the *SpTFS* can support the format selection of higher-order tensors. Tensors of any order can be transformed into matrices through tensor lowering and matrix representation. In addition to MTTKRP, the *SpTFS* can also support more general tensor computations such as Tensor Times Matrix (TTM) thanks to the versatility of tensor transformation. TTM is the multiplication of a sparse tensor with a dense matrix (called SpTTM) commonly used in Tucker decomposition [3]. SpTTM can be seen as a high-dimensional generalization of the sparse matrix-vector multiplication (SpMV) [11]. Similarly, the *SpTFS* can transform the sparse tensor involved in SpTTM into fixed-sized matrices and sparsity features, which are then fed to *TnsNet* or *TnsClustering* to obtain the optimal tensor format. We leave the adaptation of *SpTFS* to TTM for future work.

3.2 Tensor Lowering

Tensor lowering is based on flattening and mapping techniques, both of which are able to capture the sparsity distribution of tensors, which are important to train the TnsNet network. As shown in Fig. 5, we take mode-1 MTTKRP as an example to explain the flattening and mapping of a third-order tensor. For mapping, we first get the mode-1 slices of the tensor (i.e., $\mathcal{X}_{(i,:,:)}$) (Fig. 5b), which denotes the sub-arrays with the i index fixed. Next, we map the non-zero values of all slices to a slice (Fig. 5c), and the non-zero values of the same position are accumulated to obtain the density of mode-1 slices. Note that in the density distribution, all non-zero values are regarded as 1. The mode-1 mapping can be formulated in Eq. (5), where $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$ and $\mathbf{A} \in \mathbb{R}^{J \times K}$.

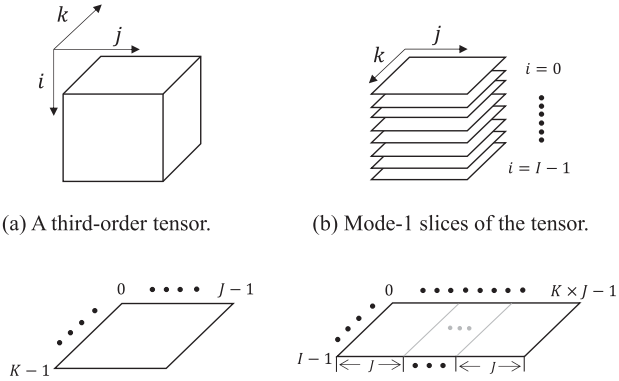


Fig. 5. The process of lowering a high-dimensional tensor into matrices using mapping or flattening.

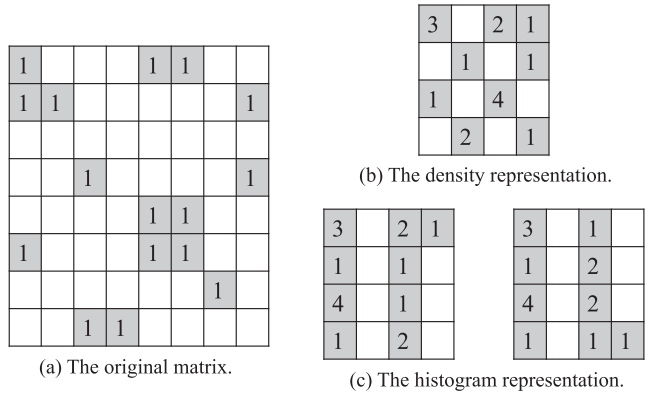


Fig. 6. Different ways for representing a matrix.

$$\mathbf{A} = \sum_{i=1}^{I} \mathcal{X}(i, :, :). \tag{5}$$

Flattening is to unfold the tensor using matricization. Fig. 5d shows the mode-1 flattening of $\mathcal{X}$ is $\mathcal{X}_{(1)}$ in Eq. (3). For any element in $\mathcal{X}$, its flattening to the matrix can be formulated in Eq. (6), where $\mathbf{B} \in \mathbb{R}^{I \times JK}$.

$$\mathbf{B}_{(i,k\times J+j)} = \mathcal{X}(i, j, k). \tag{6}$$

Here, we generalize the method of tensor lowering regarding a Nth-order tensor. For mapping, we map the non-zeros to a slice with $N-2$ indices fixed each time and eventually generate C_N^{N-1} matrices. For flattening, we unfold the tensor along each mode to generate a matrix. The mode-1 mapping can be formulated in Eq. (7), where $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ and $\mathbf{A} \in \mathbb{R}^{I_{N-1} \times I_N}$. The other matrices generated by mode-1 mapping can be deduced similarly. The mode-n flattening of $\mathcal{X}$ to matrix can be formulated in Eq. (8), where tensor element $(i_1, i_2, \ldots, i_N)$ maps to matrix element (i_n, j) [3].

$$\mathbf{A} = \sum_{i_1,\ldots,i_{N-2}=1}^{I_1,\ldots,I_{N-2}} \mathcal{X}(i_1, \ldots, i_{N-2}\, :, :) \tag{7}$$

$$j = 1 + \sum_{\substack{k=1 \\ k\neq n}}^{N} (i_k - 1) J_k, \ \textit{and}\ J_k = \prod_{\substack{m=1 \\ m\neq n}}^{k-1} I_m. \tag{8}$$

Note that flattening and mapping are two separate methods of tensor lowering that can be applied independently. As seen, mapping reflects the vertical distribution of the non-mode indices, and flattening reflects the horizontal distribution of the mode index. Therefore, they can be combined to retain the sparsity distribution of a tensor. With the tensor lowered to matrices, we then transform the matrices into fixed-sized input for the network through matrix representation (Section 3.3).

3.3 Matrix Representation

The matrices generated through tensor lowering are irregular in size, and we need to scale the matrices into fixed-sized input for the network. Inspired by [20], we consider two ways for matrix scaling: Density representation and histogram representation. Both methods can represent the coarse-grained patterns of the original matrix with acceptable sizes. The density representation captures detailed variations among different regions of the original matrix. The histogram representation [20] further captures the distance between an element and the diagonal of the original matrix while losing parts of the sparsity distribution.

As shown in Fig. 6, we illustrate the matrix representation methods by the example of mapping the 8×8 matrix to 4×4 matrices. For the density representation (Fig. 6b), each block counts non-zero elements and fills into the new matrix. For histogram representation, row histogram and column histogram are used to express the diagonal information of the original matrix (Fig. 6c). The steps of scaling with histograms are illustrated in Algorithm 1. The $rowDim$ and the $colDim$ are the row and column indices of the elements mapped to the new matrix, which are also used in the density representation. The $dist$ reflects the distance between the element and the diagonal. However, the distance does not fully reflect the sparsity distribution of an element. For example, if the elements distributed on both sides of the diagonal have the same distance from the diagonal, they will be counted at the same position of the histogram. Finally, the values of the new matrices are normalized to the range of [0,1] by dividing by the maximum value.

Algorithm 1. Matrix Representation Using Histograms

Input: input matrix IM and output resolution r
Output: row matrix RM and column matrix CM
$rowRatio = IM.height/r$
$colRatio = IM.width/r$
$maxDim = max(IM.height,\ IM.width)$
for each non-zero nz in IM **do**
  $dist = r \times abs(nz.row - nz.col)/maxDim$
  $rowDim = nz.row/rowRatio$
  $colDim = nz.col/colRatio$
  $RM[rowDim][dist] += 1$
  $CM[colDim][dist] += 1$
end for

Note that both representation methods may lose the potential patterns of a matrix, which will ultimately affect the accuracy of tensor format selection. Therefore, we

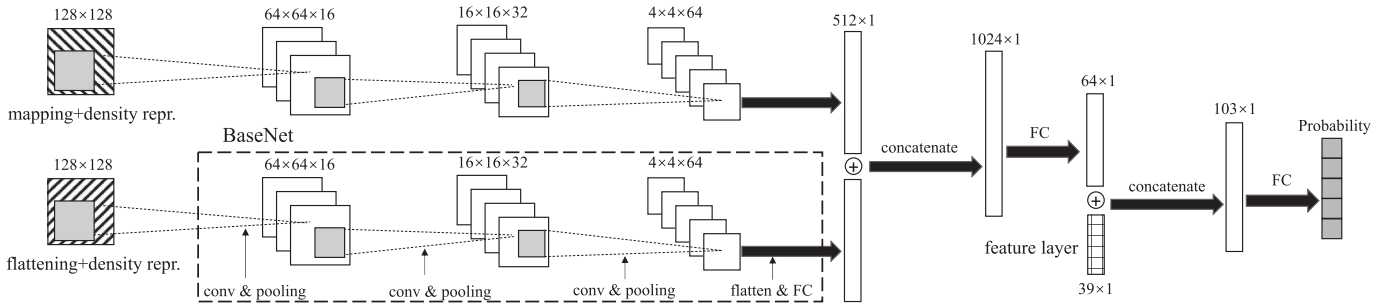


Fig. 7. The structure design of *TnsNet*.

choose to add the feature layer to the network structure to compensate for the loss of tensor features (Section 3.4).

3.4 Network Structure Design

We utilize *TnsNet* (supervised learning) and *TnsClustering* (unsupervised learning) for optimal format prediction of sparse tensors. As shown in Fig. 7, *TnsNet* combines CNN and feedforward neural network (FFNN) to predict the optimal storage format for a tensor. The inputs of *TnsNet* include the fixed-sized matrices and sparsity features through tensor transformation and feature extraction, respectively. Here, we take the density representation as an example, where the *BaseNet* includes all convolution and pooling layers of *TnsNet*. For the matrices generated after tensor transformation, we use them as inputs to the *BaseNets* separately and concatenate the outputs as the joint features. The joint features are further concatenated with the feature layer, which can supplement the missing sparsity features of the original tensors (Table 2). Due to page limit, readers can refer to [27] for more *TnsNet* details.

As shown in Fig. 8, *TnsEncoder* consists of fully connected layers, convolutional layers and deconvolutional layers. The original input is encoded by the 2nd∼6th layers and decoded by the 7th∼11th layers, which eventually derives the reconstructed input. The *TnsEncoder* is trained with the cost function minimizing the mean squared error between the original values and the reconstructed values. After training, the feature vector of an input tensor can be obtained from the 6th layer of the trained model.

TABLE 2
The Candidate Feature Set of a Tensor

Feature	Meaning
I_n	Tensor mode sizes.
slice, fiber	Number of slices, fibers.
sliceRatio, fiberRatio	The ratio of slices, fibers.
nnz	Number of non-zeros.
sparsity	Density of NNZ in the tensor.
aveNnzPerRow_n	Average NNZ per row-n.
maxNnzPerRow_n	Maximum NNZ per row-n.
minNnzPerRow_n	Minimum NNZ per row-n.
devNnzPerRow_n	The deviation of of NNZ per row-n.
adjNnzPerRow_n	Average NNZ difference of adjacent row-ns.
aveNnzPerSlice	Average NNZ per slice.
maxNnzPerSlice	Maximum NNZ per slice.
minNnzPerSlice	Minimum NNZ per slice.
adjNnzPerSlice	Average NNZ difference of adjacent slices.
devNnzPerSlice	The deviation of NNZ per slice.
aveNnzPerFiber	Average NNZ per fiber.
maxNnzPerFiber	Maximum NNZ per fiber.
minNnzPerFiber	Minimum NNZ per fiber.
devNnzPerFiber	The deviation of NNZ per fiber.
adjNnzPerFiber	Average NNZ difference of adjacent fibers.
aveFibersPerSlice	Average number of fibers per slice.
maxFibersPerSlice	Maximum number of fibers per slice.
minFibersPerSlice	Minimum number of fibers per slice.
devFibersPerSlice	The deviation of number of fibers per slice.
adjFibersPerSlice	Average fiber difference of adjacent slices.

Fig. 9 illustrates the design of *TnsClustering* using density representation for predicting the optimal tensor format. For the matrices generated after tensor transformation, we feed them into the *TnsEncoders* separately and concatenate the encoding outputs with the sparsity features (Table 2). The resulting feature vectors are then clustered by the *K-means++* algorithm. After the clustering model is trained, we profile the performance of the centroid tensor for each cluster with different storage formats on the target hardware platform. The optimal tensor format is assigned as a label for each cluster. During the prediction, the optimal storage format for the input tensor is determined according to the label of the cluster which it belongs to. Note that, the performance profiling of the centroid tensors only needs to be done once for each target platform.

Moreover, *TnsClustering* can be easily adapted to the histogram representation or higher-order tensors. In detail, the fixed-size matrices generated after tensor transformation are fed into the TnsEncoders, and the output feature vectors and sparsity features are merged as input to the *K-means++* clustering algorithm. The encoding length in *TnsEncoder* can be adjusted to ensure that the combined feature vector and sparsity features have a similar impact on the clustering results. In such a way, stable prediction performance can be achieved. When adapting the *TnsClustering* to a different hardware platform, only the optimal storage format of the centroid tensor of each cluster needs to be profiled without re-training the clustering model.

3.5 Implementation Details

Fig. 10 shows the overall workflow of *SpTFS*. We implement tensor transformation and feature extraction in C++ code to reduce the preprocessing overhead. The fixed-size matrices and sparsity features after tensor preprocessing are then fed into *TnsNet* structure or *TnsClustering* pipeline. The *TnsNet* and *Tnsclustering* are implemented with popular machine learning frameworks (i.e., TensorFlow [23] and sklearn [36]). However, the ideas behind *TnsNet* and *Tnsclustering* are general to be implemented in other frameworks as well.

The optimal sparse format of the input tensor is determined according to the classification result of *SpTFS*. Note that the above process of predicting the optimal format for a

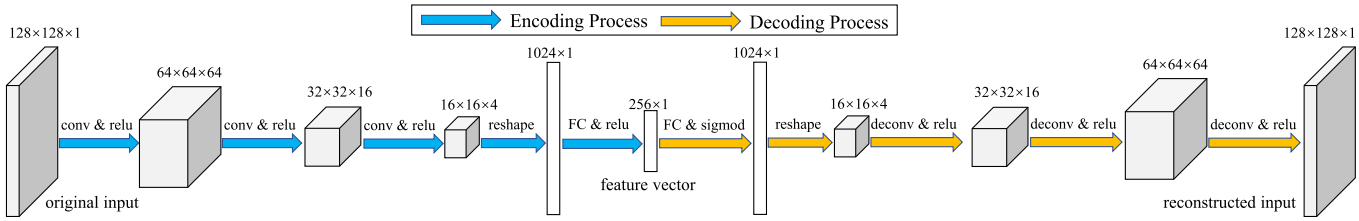


Fig. 8. The structure design of *TnsEncoder*.

single tensor only needs to be done once. After that, the tensor library such as ParTI!, SPLATT, HB-CSF and MM-CSF is invoked to perform tensor computations such as MTTKRP and CPD. Our design decouples the tensor format prediction from tensor library so that the *SpTFS* can easily support more sparse tensor formats by incorporating more tensor libraries.

4 Evaluation of Supervised Learning

In this section, we evaluate *SpTFS* by comparing *TnsNet* with other supervised learning based methods. Due to page limit, we present the experimental results that are not included in [27]. In addition, we also present the evaluation results of MM-CSF, which gives novel insights to the analysis of *TnsNet* performance on GPU.

4.1 Experiment Setup

4.1.1 Hardware and Software Platforms

As shown in Table 3, we evaluate the effectiveness of *SpTFS* on a server, which consists of an Intel Xeon Silver CPU with 16 cores and an Nvidia RTX Turing GPU with 68 SMs. The GPU is also utilized to speed up the process of training and prediction. *TnsNet* is built using TensorFlow release version 1.15 [23].

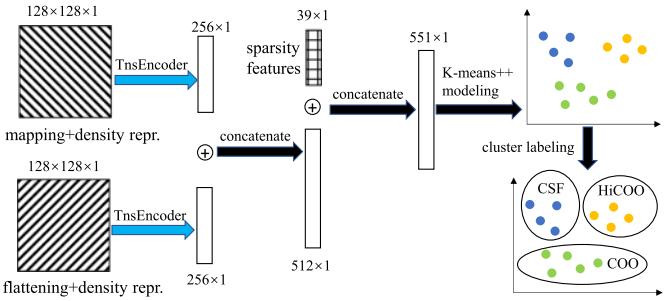


Fig. 9. The design of *TnsClustering* using density representation for predicting the optimal tensor format.

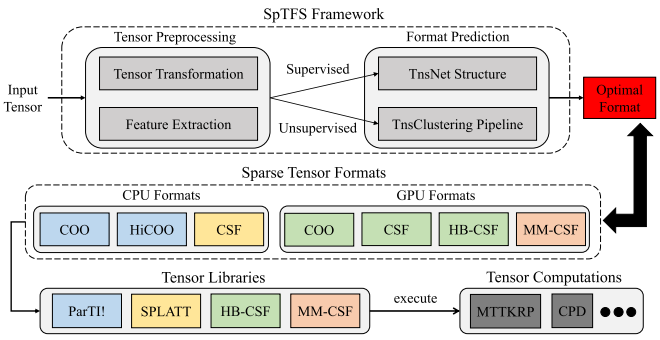


Fig. 10. The overall workflow of *SpTFS*.

4.1.2 Sparse Tensor Formats and Datasets

For MTTKRP on CPU, we evaluate the sparse tensor storage formats, including COO, CSF, and HiCOO. Specifically, the CSF implementation is adopted from SPLATT [6], whereas the COO and HiCOO implementations are adopted from ParTI! [12]. SPLATT provides the optimization option for tiling. We use CSF-tile and CSF-based to denote enabling and disabling this option. For HiCOO, we use two superblock sizes (i.e., HiCOO-sb10 and HiCOO-sb14) according to [12]. For example, HiCOO-sb10 means that the size of the superblock is 2^{10}. For MTTKRP on GPU, the tensor formats we evaluate include COO, F-COO, CSF, HB-CSF and MM-CSF. We adopt the implementations of all tensor formats from [13] except F-COO and MM-CSF. For MM-CSF, the implementation is adopted from [14]. Since there is no public F-COO implementation available, we develop our F-COO implementation based on [11]. Due to severe load imbalance, F-COO exhibits relatively poor performance, which has also been confirmed in [13]. Among all GPU formats during our evaluation, F-COO only accounts for 0.4% of the cases with the best performance, whereas in 59.3% of the cases, it leads to the worst performance. Therefore, we do not consider F-COO for tensor format selection. To ensure fairness, we use the best parameter configurations reported in each implementation and compile with the "O3" option.

For the evaluation dataset, we generate 9,855 third-order tensors and 9,793 fourth-order tensors based on 2,726 matrices selected from the SuiteSparse Matrix Collection [39]. The number of non-zero elements ranges from 3 to 9,953,208. In addition, we add 16 real-world tensors (10 for 3-D and 6 for 4-D) from FROSTT [29] and HaTen2 [30] to the evaluation dataset (listed in Fig. 3). Note that the evaluation dataset is randomly divided into a training set and a validation set during cross validation (Section 4.1.3).

TABLE 3
Hardware Platforms Used for Evaluation

	Intel CPU	Nvidia GPU
Core	Intel Xeon Silver 4110 2 processors, 16 cores	GeForce RTX 2080 Ti 68 SMs
Frequency	2.1GHz	1.3GHz
Caches	32KB L1, 1MB L2, 11MB L3	5.5MB L2
Memory	192GB DDR4, 230.4GB/s	11GB GDDR6, 616.0GB/s
OS/Driver	CentOS Linux release 7.6	GPU Driver 440.33
Compiler	gcc-4.8	nvcc-10.2

TABLE 4
Prediction Accuracy of *TnsNet* and *GBDT* on GPU

MTTKRP Mode	SpTensor Format	Ground Truth	TnsNet		GBDT	
			recall	prec.	recall	prec.
Mode-1	COO	5044	0.98	0.97	0.89	0.8
	CSF	2771	0.92	0.93	0.5	0.52
	HB-CSF	2001	0.94	0.93	0.59	0.75
	MM-CSF	39	0.69	0.82	0.03	0.5
	Total	9855	**0.95**		0.72	
Mode-2	COO	4075	0.95	0.96	0.74	0.71
	CSF	2096	0.9	0.9	0.51	0.49
	HB-CSF	1706	0.91	0.9	0.64	0.7
	MM-CSF	1978	0.96	0.96	0.74	0.77
	Total	9855	**0.94**		0.67	
Mode-3	COO	3521	0.97	0.97	0.94	0.97
	CSF	2534	0.84	0.82	0.54	0.54
	HB-CSF	1958	0.83	0.81	0.56	0.65
	MM-CSF	1842	0.97	0.98	0.94	0.91
	Total	9855	**0.89**		0.71	
All-Mode	Overall		**0.93**		0.7	

4.1.3 Cross Validation

We use the 5-fold cross validation method to evaluate the accuracy of the models. The validation method is widely used in literatures [20], [21], [41]. Specifically, we randomly divide the evaluation dataset into 5 folds. In each round, a single fold is selected as the validation set, and the other 4 folds are used as the training set. The above process is repeated for 5 rounds, each of which selects a different fold as the validation set. The validation results are averaged over the rounds to reduce variability.

We compare the design of *TnsNet* with the XGBoost [33] machine learning methods, including *GBDT* and *GBlinear*, which are based on CART [34] and the linear model, respectively. The input features of *GBDT* and *GBLinear* are listed in Table 2, which are the same to the feature layer of TnsNet. The *GBDT* and *GBLinear* are trained for CPU and GPU platforms separately. We have carefully tuned the hyperparameters of *GBDT* and *GBLinear*. The hyperparameters of *TnsNet* include the batch size, the convolution filter size and the learning rate. We empirically determine the hyperparameter settings across the validations. We set the batch size to 100 and the convolution filter size to 3×3. We select the Adam stochastic optimizer with 0.0001 learning rate. During training data collection, we set the runtime parameters according to previous works [6], [12], [13]. Specifically, we set the number of threads to 16 when on CPU, and the thread block size to 256 when on GPU. The rank size is set to 16.

4.2 Results for Prediction

As seen from [27], the *TnsNet* with density representation (*density repr.*) and feature layer (*FL*) achieves the highest prediction accuracy on CPU. In adition, *GBDT* performs better than *GBlinear* due to its ability to process nonlinar data. Therfore, we will only present the evaluation results of *TnsNet* (*density repr.+FL*) and *GBDT* in the rest of the paper.

Table 4 reports the prediction accuracy on GPU. Similarly, *TnsNet* achieves better prediction accuracy compared to *GBDT* in all modes across all tensor formats. Specifically, *TnsNet* achieves 93% prediction accuracy on average, whereas *GBDT* only achieves 70%. The prediction accuracy of *GBDT* on GPU is significantly lower than that on CPU (86%). This is because the tensor formats (e.g., HiCOO-sb10 and CSF-tile) on CPU are all derived from three basic tensor formats (i.e., COO, HiCOO and CSF) with minor configuration changes. Therefore, they share similar computation patterns as the basic tensor formats. Whereas on GPU, there are more basic tensor formats (four in our experiments) available. Due to the lack of spatial distribution information, the prediction accuracy of *GBDT* becomes worse when more tensor formats need to be considered. The above drawback of *GBDT* in turn demonstrates *TnsNet* is more advantageous with more sparse tensor formats emerging in the future.

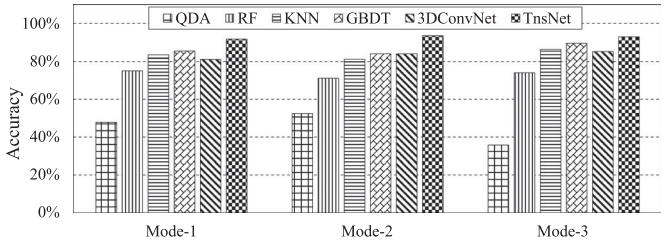


Fig. 11. Prediction accuracy comparison between *TnsNet* and other supervised learning based methods on CPU.

In addition, as shown in Table 4 the proportion of MM-CSF achieving the best performance in mode-1 is low. This is because the tensor generated by matrix combination in mode-1 has the same number of non-zeros in each fiber. In this case, although MM-CSF's re-partitioning of non-zeros obtains better data compression and reduces memory usage, it may lead to load imbalance among warps. Therefore, for most tensors, the performance of MM-CSF is worse than CSF and HB-CSF in mode-1.

4.3 Comparison With Other Supervised Methods

To further evaluate the superiority of *SpTFS*, we compare *TnsNet* with more supervised learning based methods (Fig. 11). We implement three traditional machine learning methods using sklearn [36], including Quadratic Discriminant Analysis (*QDA*), Random Forest (*RF*) and K-Nearest Neighbor (*KNN*). The input features of these methods are consistent with *GBDT*. In addition, we implement a 3-D convolutional network (*3DConvNet*) (with *conv3d* operator) using TensorFlow [23]. Specifically, we transform a tensor into a fixed size of $32 \times 32 \times 32$ through the density representation, which is fed into *3DConvNet* together with sparsity features. Similar to *TnsNet*, the output of the fixed-size tensor after 3-D convolutions and fully connected layers is concatenated with the feature layer. We have carefully tuned all hyperparameters of the above methods.

As shown in Fig. 11, *TnsNet* still achieves the highest prediction accuracy in all modes. Among traditional machine learning methods, *GBDT* achieves higher accuracy in general. This indicates the effectiveness of *GBDT* for processing nonlinear data. *TnsNet* performs better than *3DConvNet* because it retains the mode-specific sparsity distribution as much as possible. Although increasing the transformed tensor size (i.e., input resolution) may improve the prediction accuracy of *3DConvNet*, the training time and inference time of *3DConvNet* would become prohibitively large (1.51× and 1.20× larger than that of *TnsNet* in current implementation). In addition, increasing the input resolution also exacerbates

TABLE 5
Clustering Accuracy Comparison of Unsupervised Learning Based Methods

Architecture & Dimension	MTTKRP Mode	TnsClustering			FcClustering			PureClustering		
		Top-1 acc.	Top-2 acc.	H-score	Top-1 acc.	Top-2 acc.	H-score	Top-1 acc.	Top-2 acc.	H-score
CPU & 3-D	Mode-1	**0.75**	**0.95**	**0.58**	0.69	0.93	0.5	0.69	0.9	0.46
	Mode-2	**0.75**	**0.95**	**0.6**	0.73	0.95	0.57	0.7	0.91	0.49
	Mode-3	**0.77**	0.94	**0.5**	0.77	**0.95**	0.5	0.75	0.92	0.41
GPU & 3-D	Mode-1	**0.72**	0.94	**0.41**	0.71	**0.95**	0.4	0.67	0.92	0.29
	Mode-2	**0.66**	**0.91**	**0.41**	0.62	0.89	0.36	0.6	0.86	0.3
	Mode-3	**0.64**	**0.89**	**0.43**	0.61	0.87	0.39	0.57	0.84	0.31
CPU & 4-D	Mode-1	**0.64**	**0.9**	**0.38**	0.62	0.89	0.36	0.61	0.89	0.34
	Mode-2	**0.65**	**0.88**	**0.44**	0.62	0.87	0.4	0.61	0.87	0.39
	Mode-3	**0.63**	**0.89**	**0.41**	0.6	0.88	0.37	0.61	0.88	0.38
	Mode-4	**0.62**	**0.88**	0.4	0.6	0.88	0.38	0.62	0.88	**0.4**

tensor preprocessing overhead. Therefore, we believe that *TnsNet* works better for tensor format selection than 3-D convolution network.

5 Evaluation of Unsupervised Learning

In this section, we run a series of experiments to evaluate the effectiveness of *SpTFS*. In addition, we compare *TnsClustering* with other unsupervised methods in terms of prediction accuracy and performance speedup. Moreover, we analyze the overhead of *SpTFS* from both training and prediction aspects. We also report the sensitivity of *TnsClustering* to the number of cluster centroids and encoding length.

5.1 Experiment Setup

We use the same hardware platforms listed in Table 3 to evaluate the unsupervised learning based method in *SpTFS*. The autoencoders are built using the Keras APIs [42] of TensorFlow v1.15 with eager execution enabled. In eager mode, TensorFlow evaluates operations immediately without constructing a computational graph in advance. This design provides simplicity and flexibility, thereby making programming and model debugging easier. Unlike supervised learning based method, the autoencoders are trained directly using unlabeled sparse tensor datasets (illustrated in Section 4.1.2). The *K-means*++ clustering algorithm is implemented using scikit-learn v0.20.4 [36].

We use the 5-fold cross validation method to evaluate the accuracy of the models. We compare *TnsClustering* with *FcClustering*, which uses a stacked autoencoder [43] without convolutional and deconvolutional layers. We also evaluate the performance of the method that uses the *K-means*++ algorithm to cluster sparse tensors based on their sparsity features (Table 2) without autoencoders (named as *PureClustering*). The above three methods are trained only once on each target hardware platform. For each target platform, the cluster labeling is done to determine the optimal storage format for each cluster. For both autoencoders, we empirically determine the hyperparameter settings across the validations. Specifically, we select the Adam stochastic optimizer with 0.001 learning rate and a batch size of 100. For *TnsEncoder*, we set the convolution filter size to 3×3 and the stride to 2. For *K-means*++ algorithm, we set the number of cluster centroids to 256 and the batch size to 200. Since the density representation outperforms the histogram representation (Section 4), all experiments for unsupervised learning are conducted with the density representation.

5.2 Prediction Accuracy

We use top-N accuracy to measure the effectiveness of clustering model, where top-N accuracy is how often the predicted class falls in the top N values of the probability distribution. The homogeneity score (H-score) [36] is used to check whether all of the clusters contain only samples that are members of a single class. H-score is independent of the absolute values of the labels. A higher H-score means the better homogeneity of the clustering.

Table 5 reports the clustering accuracy of three unsupervised learning based methods on different hardware platforms. For 3-D tensors on CPU, *TnsClustering* achieves the highest H-score in all modes, which reflects the effectiveness of *TnsEncoder* for clustering. Compared to *FcClustering* and *PureClustering*, *TnsClustering* achieves higher Top-1/2 accuracy in most modes on both hardware platforms. Specifically, *TnsClustering* achieves 76% Top-1 accuracy on average, whereas *FcClustering* and *PureClustering* only achieve 73% and 71%, respectively. The results of Top-1 accuracy on GPU are similar to that on CPU. *TnsClustering* achieves an average Top-2 accuracy of 95% and 92% on CPU and GPU respectively, which is close to the Top-1 accuracy of *TnsNet* (93% and 93%). This indicates that *TnsEncoder* can effectively encode the spatial information of sparse tensors into feature vectors, and thus improve the clustering accuracy.

For 4-D tensors, we adjust the encoding length of autoencoders to provide stable clustering accuracy. For the matrices generated after flattening and mapping, we set the encoding length to 128 and 32, respectively. As shown in Table 5, *TnsClustering* achieves the highest Top-1/2 accuracy across all modes. Specifically, *TnsClustering* achieves 63.4% Top-1 accuracy on average, whereas *FcClustering* and *PureClustering* achieve 60.9% and 61.2%, respectively. We can observe that *PureClustering* has even better clustering accuracy than *FcClustering*. This is because the sparsity patterns of higher-order tensors are difficult to be accurately represented with only fully connected layers. This also proves the effectiveness of sparsity features extracted by *SpTFS* for *K-means*++ clustering.

Note that the supervised learning and unsupervised learning methods satisfy different user scenarios regarding the cost of labeling training data. When the cost of labeling

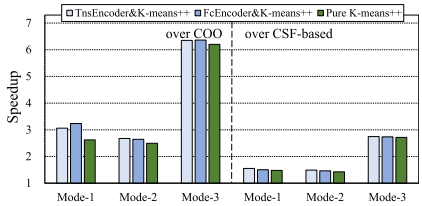


Fig. 12. Speedup of unsupervised learning based methods over the COO and CSF-based format on CPU.

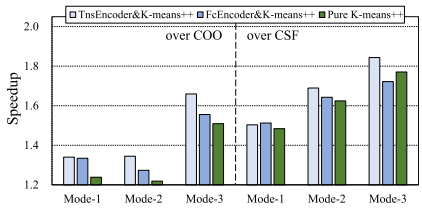


Fig. 13. Speedup of unsupervised learning based methods over the COO and CSF format on GPU.

training data is acceptable, it is always recommended to choose the supervised learning method (*TnsNet*) for better prediction accuracy.

5.3 Performance Speedup

The performance of MTTKRP using the tensor format predicted by the unsupervised learning based methods on CPU and GPU is presented in Figs. 12 and 13, respectively. The performance using the COO format and CSF-based/CSF format is chosen as the baseline, respectively. We can observe that *TnsClustering* achieves higher performance speedup than *FcClustering* and *PureClustering* in most modes on both platforms.

On CPU, *TnsClustering* achieves an average speedup of 4.03× and 1.93× over COO format and CSF-based format, respectively. The low performance speedup of *TnsClustering* over CSF-based format is due to the fact that CSF format performs better than the COO format in most cases [27]. On GPU, *TnsClustering* achieves 1.45× and 1.68× speedup over COO format and CSF format, respectively. The low performance speedup of *TnsClustering* over COO format is due to the fact that COO format already achieves optimal performance in most cases (refer to Table 4). Although the Top-1 prediction accuracy of *TnsClustering* is significantly lower than that of *TnsNet*, it does not lead to a large gap in performance speedup [27]. This indicates that even if *TnsClustering* does not accurately predict the optimal tensor format in some cases, it can still select a good tensor format with performance speedup.

The performance speedup for 4-D tensors is shown in Fig. 14. *TnsClustering* achieves the highest performance speedup in most modes. On average, *TnsClustering* achieves 17.0× and 1.70× speedup over COO format and CSF-based format, respectively. This is because the distribution of the optimal formats is highly skewed. For example, the COO format only accounts for 4.1% of the cases with the best performance. The performance gap among different formats of 4-D tensors is much larger than that of 3-D tensors. This demonstrates the necessity of our proposed methods for selecting the optimal format to speedup tensor computations.

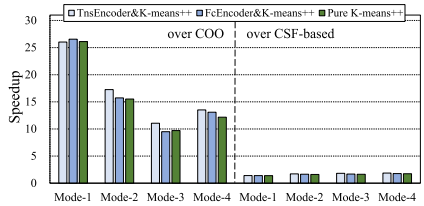


Fig. 14. Speedup of unsupervised learning based methods over the COO and CSF format for 4-D tensors.

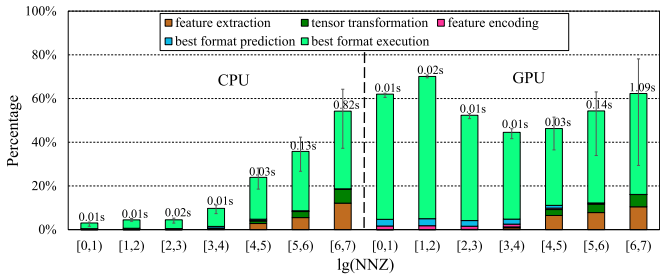


Fig. 15. Performance breakdown using *TnsClustering* for optimal format selection normalized to the COO format.

5.4 Overhead Analysis

Considering the training overhead of *TnsClustering*, it takes about 7 minutes to train *TnsEncoder*, and it only takes about 2 seconds to train the *K-means++* model. Profiling the optimal tensor formats for all centroid tensors (cluster labeling) takes about 28 minutes and 2 hours on CPU and GPU, respectively. Compared to *TnsNet* that it takes about 15 hours and 70 hours to collect the performance profiles on CPU and GPU, *TnsClustering* uses unsupervised learning based method and thus only needs to profile the optimal formats for the centroid tensors (256 in our experiments). Therefore, it significantly eliminates the time for preparing the training dataset. Since the training of *TnsEncoder* and *K-means++* model is independent of the hardware platform, and the cluster labeling only needs to be performed once for each target platform, the training overhead of *TnsClustering* is negligible. The prediction overhead of *TnsClustering* can be divided into four parts: 1) feature extraction, 2) tensor transformation, 3) feature encoding using *TnsEncoder*, and *4)* format prediction using *K-means++* model. Our empirical study shows that the prediction overhead can be amortized to 15 iterations of CPD-ALS.

Fig. 15 shows the performance breakdown using *TnsClustering* for optimal format selection normalized to that using the COO format on CPU and GPU. The cases where COO is already the optimal format are not taken into account. The performance of tensor computation is presented in 7 groups, where the nnz ranges from 10^0 to 10^7. The absolute value and error bar for each group are also presented. The results less than 100% indicate achieving better performance even with prediction overhead considered. On CPU, the performance results of all groups are below 100%. The overhead of feature encoding and format prediction is negligible compared to other parts. On GPU, the results of all groups are still below 100%. Due to the shorter tensor computation time, the overhead of feature encoding is significant than that on CPU. Compared to *TnsNet* [27], *TnsClustering* has lower overhead due to the shorter time of feature encoding and format prediction.

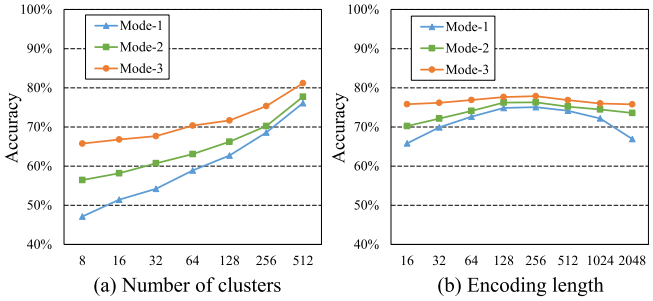


Fig. 16. The prediction accuracy of *TnsClustering* with different (a) number of clusters, and (b) encoding length.

5.5 Parameter Sensitivity Analysis

5.5.1 Number of Cluster Centroids

Fig. 16a presents the prediction accuracy of *TnsClustering* with different number of cluster centroids. The number of cluster centroids ranges from 2^3 to 2^9 with a stride of $\times 2$. As shown, the prediction accuracy is positively correlated with the number of cluster centroids. However, the improvement of prediction accuracy gradually slows down as the number of cluster centroids increases. Note that the overhead of cluster labeling during training is proportional to the number of cluster centroids. Therefore, if the overhead of cluster labeling is acceptable, users can increase the number of cluster centroids within a certain range to improve the prediction accuracy.

5.5.2 Encoding Length

Fig. 16b presents the prediction accuracy with different encoding length of *TnsEncoder*. As shown, a higher prediction accuracy is achieved when the encoding length is set between 128 and 256. Note that the time of *TnsEncoder* forward inference is mainly dominated by the convolution layers. This means that the length of the output layer has little impact on the encoding overhead. Therefore, we choose the output length of 256 for *TnsEncoder* to achieve better prediction accuracy.

6 Related Work

Optimization for CPD Algorithm. For optimizing the CPD algorithm on CPUs, Smith *et al.* [6] developed SPLATT that accelerated the CPD algorithm through efficient cache-friendly tilting and reordering under CSF format. For optimizing the CPD algorithm on GPUs, Liu *et al.* [11] introduced F-COO, which adds flag arrays for eliminating atomic operations. Li *et al.* [12] developed the HiCOO format that is compact and mode-generic to improve the performance of the CPD algorithm. Nisa *et al.* [13] developed HB-CSF to alleviate the load imbalance caused by tree-based CSF. Furthermore, to mitigate overwhelming memory overhead when using CSF to calculate CPD in d^{th} order, MM-CSF was proposed by Nisa *et al.* [14], which a hybrid of multiple CSF tensors compressed along different mode to reach the best compression ratio. Meanwhile, Srivastava *et al.* [7] designed the Compressed Interleaved Sparse Slice (CISS) format, which is the generalization of the Compressed Interleaved Sparse Row (CISR) format [44] for the matrix to match a customized hardware for CPD. Phipps *et al.* [4] utilized Kokkos to build portable CPD software among multicore CPU, GPU and manycore Intel Xeon Phi. Although the above optimizations significantly improve the performance of tensor computation, they cannot effectively adapt to the complex sparsity patterns of tensors. The *SpTFS* is built on top of these optimizations, and automatically selects an optimization scheme suitable for each tensor through deep learning methods.

Sparse Matrix Format Selection. Existing methods for sparse matrix format selection can be divided into traditional machine learning based methods [15], [16], [17] and deep learning based methods [20], [21], [22]. Sedaghati *et al.* [15] developed the learning model using the decision tree classifier to select the best matrix format on GPUs. Zhao *et al.* [16] proposed a two-stage scheme using regression tree-based models to construct overhead-conscious selectors of sparse matrix formats. Chen *et al.* [17] trained a decision tree model for the ARMv8-based manycore architecture to predict the best sparse matrix format and corresponding parameters. For deep learning based methods, Zhao *et al.* [20] used CNN for the first time to implement sparse matrix format selection through histogram representation. Xie *et al.* [22] used CNN to predict the best format and algorithm for SpGEMM through matrix features and density representation. Barred *et al.* [21] proposed a blockwise strategy to make the CNN model independent of matrix sizes and estimated the performance through regression. Since the above works are developed based on supervised learning, they all require large engineering efforts to label training data. In addition, the *SpTFS* supports the format selection for any arrays higher than two-dimensional by transforming them to fixed-sized matrices and sparsity features.

Supervised/Unsupervised Image Classification. Selecting the optimal sparse tensor format can be regarded as a classification problem, similar to recognizing handwritten digits. For supervised learning based methods, CNN-based network design has achieved impressive performance in image classification [18], [19], [31]. For example, Li *et al.* [18] designed a customized CNN for classification of lung image patches. Later, CNN was applied to more complex multi-label classification tasks. Wang *et al.* [19] combined CNN and RNN to model the label co-occurrence dependency in a joint image/label embedding space. Whereas, unsupervised learning based methods generally consist of two stages: 1) extract low-dimensional features of the original input, and 2) apply traditional clustering algorithms (e.g., K-means [26]) based on the extracted features. Stacked autoencoders (SAE) [45] trained a stack of denoising autoencoders in a layer-wise manner and the entire network was fine-tuned to predict pixel states. Convolutional autoencoders (CAE) [25] learned spatial relationships between image pixels in an end-to-end manner without greedily layer-wise pre-training. Generative adversarial networks (GAN) [37] built image representations by training, and reused parts of the generator and discriminator as feature extractors for unsupervised classification. To achieve the tradeoff between prediction accuracy and inference overhead, the *SpTFS* selects CNN and CAE as the basis for supervised and unsupervised methods, respectively. To the best of our knowledge, this is the first work to utilize unsupervised learning for optimal sparse format prediction.

7 Conclusion and Future Work

In this paper, we propose an automatic tensor format selection framework *SpTFS* that effectively predicts the optimal storage format for an input tensor running MTTKRP. The *SpTFS* lowers the high-dimensional tensors into fixed-sized matrices through tensor lowering and matrix representation. Using supervised learning based method, we propose *TnsNet*, a re-designed CNN network with the feature layer added to compensate for the sparsity features lost during matrix representation. Once trained, the *TnsNet* can be used with either density or histogram representation of the input tensor for optimal format prediction. Whereas, using unsupervised learning based method, we propose *TnsClustering*, that consists of a re-designed CAE (*TnsEncoder*) for better feature encoding of input tensor, and a *K-means++* model to cluster sparse tensors for optimal tensor format prediction. The experiment results show that *SpTFS* achieves high prediction accuracy to determine the optimal tensor format on both CPU and GPU platforms, which in turn leads to significant performance speedup for MTTKRP.

For future work, we would like to extend *SpTFS* to support more tensor computations such as TTM, which is the key building block of the ALS-based Tucker decomposition. Since the computational complexity of TTM is lower than that of MTTKRP, we may need to further improve the performance of tensor transformation of *SpTFS* for reduced preprocessing overhead. We would also like to treat the tensor format selection as a regression problem and predict the performance of each tensor format, if there are more tensor formats emerging in the future and the performance gap among different tensor formats is further reduced.

References

[1] A. Cichocki *et al.*, "Tensor decompositions for signal processing applications: From two-way to multiway component analysis," *IEEE Signal Process. Mag.*, vol. 32, no. 2, pp. 145–163, Mar. 2015.

[2] F. Le Gall , "Powers of tensors and fast matrix multiplication," in *Proc. 39th Int. Symp. Symbolic Algebr. Comput.*, 2014, pp. 296–303.

[3] T. G. Kolda and B. W. Bader, "Tensor decompositions and applications," *SIAM Rev.*, vol. 51, no. 3, pp. 455–500, 2009.

[4] E. T. Phipps and T. G. Kolda, "Software for sparse tensor decomposition on emerging computing architectures," *SIAM J. Sci. Comput.*, vol. 41, no. 3, pp. C269–C290, 2019.

[5] F. L. Hitchcock, "The expression of a tensor or a polyadic as a sum of products," *J. Math. Phys.*, vol. 6, no. 1/4, pp. 164–189, 1927.

[6] S. Smith, N. Ravindran, N. D. Sidiropoulos, and G. Karypis, "SPLATT: Efficient and parallel sparse tensor-matrix multiplication," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2015, pp. 61–70.

[7] N. Srivastava, H. Jin, S. Smith, H. Rong, D. Albonesi, and Z. Zhang, "Tensaurus: A versatile accelerator for mixed sparse-dense tensor computations," in *Proc. IEEE Int. Symp. High Perform. Comput. Architecture*, 2020, pp. 689–702.

[8] O. Kaya and B. Uçar, "Scalable sparse tensor decompositions in distributed memory systems," in *Proc. IEEE Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2015, pp. 1–11.

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

[10] E. F. D'Azevedo, M. R. Fahey, and R. T. Mills, "Vectorized sparse matrix multiply for compressed row storage format," in *Proc. Int. Conf. Comput. Sci.*, 2005, pp. 99–106.

[11] B. Liu, C. Wen, A. D. Sarwate, and M. M. Dehnavi, "A unified optimization approach for sparse tensor operations on GPUs," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2017, pp. 47–57.

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

[13] I. Nisa, J. Li, A. Sukumaran-Rajam , R. Vuduc, and P. Sadayappan, "Load-balanced sparse MTTKRP on GPUs," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2019, pp. 123–133.

[14] I. Nisa, J. Li, A. Sukumaran-Rajam , P. S. Rawat, S. Krishnamoorthy, and P. Sadayappan, "An efficient mixed-mode representation of sparse tensors," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2019, pp. 1–25.

[15] N. Sedaghati, T. Mu, L.-N. Pouchet, S. Parthasarathy, and P. Sadayappan, "Automatic selection of sparse matrix representation on GPUs," in *Proc. 29th ACM Int. Conf. Supercomputing*, 2015, pp. 99–108.

[16] Y. Zhao, W. Zhou, X. Shen, and G. Yiu, "Overhead-conscious format selection for SpMV-based applications," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2018, pp. 950–959.

[17] D. Chen, J. Fang, S. Chen, C. Xu, andZ. Wang, "Optimizing sparse matrix–vector multiplications on an ARMv8-based many-core architecture," *Int. J. Parallel Program.*, vol. 47, no. 3, pp. 418–432, 2019.

[18] Q. Li, W. Cai, X. Wang, Y. Zhou, D. D. Feng, and M. Chen, "Medical image classification with convolutional neural network," in *Proc. IEEE 13th Int. Conf. Control Autom. Robot. Vis.*, 2014, pp. 844–848.

[19] J. Wang, Y. Yang, J. Mao, Z. Huang, C. Huang, and W. Xu, "CNN-RNN: A unified framework for multi-label image classification," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 2285–2294.

[20] Y. Zhao, J. Li, C. Liao, and X. Shen, "Bridging the gap between deep learning and sparse matrix format selection," in *Proc. 23rd ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2018, pp. 94–108.

[21] M. Barreda, M. F. Dolz, M. A. Castaño, P. Alonso-Jordá, and E. S. Quintana-Ortí, "Performance modeling of the sparse matrix–vector product via convolutional neural networks," *J. Supercomputing*, vol. 76, pp. 8883–8900, 2020.

[22] Z. Xie, G. Tan, W. Liu, and N. Sun, "IA-SpGEMM: An input-aware auto-tuning framework for parallel sparse matrix-matrix multiplication," in *Proc. ACM Int. Conf. Supercomputing*, 2019, pp. 94–105.

[23] M. Abadi *et al.*, "TensorFlow: A system for large-scale machine learning," in *Proc. 12th USENIX Symp. Operating Syst. Des. Implementation*, 2016, pp. 265–283.

[24] A. Paszke *et al.*, "PyTorch: An imperative style, high-performance deep learning library," *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2019, pp. 8024–8035.

[25] X. Guo, X. Liu, E. Zhu, and J. Yin, "Deep clustering with convolutional autoencoders," in *Proc. Int. Conf. Neural Inf. Process.*, 2017, pp. 373–382.

[26] J. MacQueen *et al.*, "Some methods for classification and analysis of multivariate observations," in *Proc. 5th Berkeley Symp. Math. Statist. Probability*, 1967, pp. 281–297.

[27] Q. Sun *et al.*, "SpTFS: Sparse tensor format selection for MTTKRP via deep learning," in *Proc. IEEE Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2020, pp. 1–14.

[28] J. Håstad, "Tensor rank is NP-complete," *J. Algorithms Print*, vol. 11, no. 4, pp. 644–654, 1990.

[29] S. Smith *et al.*, "FROSTT: The formidable repository of open sparse tensors and tools," 2017. [Online]. Available: http://frostt.io/

[30] I. Jeon, E. E. Papalexakis, U. Kang, and C. Faloutsos, "HaTen2: Billion-scale tensor decompositions," in *Proc. IEEE 31st Int. Conf. Data Eng.*, 2015, pp. 1047–1058.

[31] M. Zhang, W. Li, and Q. Du, "Diverse region-based CNN for hyperspectral image classification," *IEEE Trans. Image Process.*, vol. 27, no. 6, pp. 2623–2634, Jun. 2018.

[32] J. H. Friedman, "Greedy function approximation: A gradient boosting machine," *Ann. Statist.*, vol. 29, pp. 1189–1232, 2001.

[33] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *Proc. 22nd ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2016, pp. 785–794.

[34] A. S. Foulkes, "Classifcation and regression trees," in *Applied Statistical Genetics With R*. Berlin, Germany: Springer, 2009, pp. 157–179.

[35] D. Arthur and S. Vassilvitskii, "k-means++: The advantages of careful seeding," Stanford Univ., Stanford, CA, 2006.

[36] F. Pedregosa *et al.*, "Scikit-learn: Machine learning in Python," *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, 2011.

[37] I. J. Goodfellow *et al.*, "Generative adversarial networks,"*Commun. ACM*, vol. 63, no. 11, pp. 139–144, 2020.

[38] A. Radford, L. Metz, and S. Chintala, "Unsupervised representation learning with deep convolutional generative adversarial networks," 2016, *arXiv:1511.06434*.

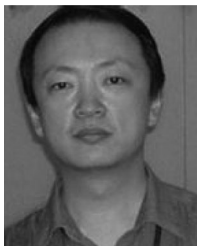
[39] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1–25, 2011.
[40] C. Liu, B. Xie, X. Liu, W. Xue, H. Yang, and X. Liu, "Towards efficient SpMV on Sunway manycore architectures," in *Proc. Int. Conf. Supercomputing*, 2018, pp. 363–373.
[41] J. C. Pichel and B. Pateiro-López, "Sparse matrix classification on imbalanced datasets using convolutional neural networks," *IEEE Access*, vol. 7, pp. 82 377–82 389, 2019.
[42] A. Gulli and S. Pal, *Deep Learning With Keras*. Birmingham, U.K.: Packt Publishing Ltd, 2017.
[43] P. Vincent, H. Larochelle, I. Lajoie, Y. Bengio, P.-A. Manzagol, andL. Bottou, "Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion," *J. Mach. Learn. Res.*, vol. 11, no. 12, pp. 3371–3408, 2010.
[44] J. Fowers, K. Ovtcharov, K. Strauss, E. S. Chung, and G. Stitt, "A high memory bandwidth FPGA accelerator for sparse matrix-vector multiplication," in *Proc. IEEE 22nd Annu. Int. Symp. Field-Programmable Custom Comput. Mach.*, 2014, pp. 36–43.
[45] J. Xie, R. Girshick, and A. Farhadi, "Unsupervised deep embedding for clustering analysis," in *Proc. 33rd Int. Conf. Mach. Learn.*, 2016, pp. 478–487.

Qingxiao Sun is currently working toward the PhD degree in the School of Computer Science and Engineering, Beihang University, China. He is currently working on GPU hardware extension and performance optimization. His research interests include computer architecture, HPC and deep learning.

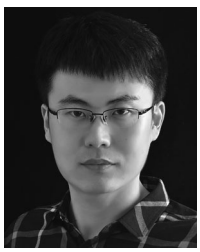
Yi Liu received the PhD degree from the Department of Computer Science, Xi'an Jiaotong University, China, in 2000. He is currently a professor in the School of Computer Science and Engineering, and director of the Sino-German Joint Software Institute (JSI) at Beihang University, China. His research interests include computer architecture, HPC and new generation of network technology.

Hailong Yang received the PhD degree from the School of Computer Science and Engineering, Beihang University, China, in 2014. He is currently an associate professor in the School of Computer Science and Engineering, Beihang University, China. He has been involved in several scientific projects such as performance analysis for big data systems and performance optimization for large scale applications. His research interests include parallel and distributed computing, HPC, performance optimization and energy efficiency.

Ming Dun received the BS degree in electrical engineering from Beihang University, China, in 2019. She is currently working toward the MS degree in the School of Cyber Science and Technology, Beihang University, China. Her research interests include parallel computing, high-performance computing, cloud computing and bioinformatics.



Zhongzhi Luan received the PhD degree from the School of Computer Science, Xi'an Jiaotong University, China. He is currently an associate professor of computer science and engineering, and assistant director of the Sino-German Joint Software Institute (JSI) Laboratory at Beihang University, China, since 2003. His research interests including distributed computing, parallel computing, grid computing, HPC and the new generation of network technology.



Lin Gan (Member, IEEE) received the PhD degree in computer science from Tsinghua University, China. He is currently an assistant researcher with the Department of Computer Science and Technology, Tsinghua University, China, and the assistant director of the National Supercomputing Center in Wuxi. His research interests include high-performance computing solutions based on hybrid platforms such as GPUs, FPGAs, and Sunway CPUs. He was the recipient of 2016 ACM Gordon Bell Prize, 2017 ACM Gordon Bell Prize Finalist, 2018 IEEE-CS TCHPC Early Career Researchers Award for Excellence in HPC, and the Most Significant Paper Award in 25 Years awarded by FPL 2015, etc.

Guangwen Yang (Member, IEEE) received the PhD degree in computer science from Tsinghua University, China. He is currently a professor with the Department of Computer Science and Technology, Tsinghua University, China, and the director of National Supercomputing Center in Wuxi. His research interests include parallel algorithms, cloud computing, and the earth system model. He has received the ACM Gordon Bell Prize in the year of 2016 and 2017, and the Most Significant Paper Award in 25 Years awarded by FPL 2015, etc.

Depei Qian received the master's degree from the University of North Texas, Denton, Texas, in 1984. He is currently a professor at the Department of Computer Science and Engineering, Beihang University, China and also chief scientist of China National High Technology Program (863 Program) on high productivity computer and service environment. His research interests include innovative technologies in distributed computing, high-performance computing and computer architecture. He is also a fellow of China Computer Federation (CCF).