SC20: International Conference for High Performance Computing, Networking, Storage and Analysis | 978-1-7281-9998-6/20/$31.00 ©2020 IEEE | DOI: 10.1109/SC41405.2020.00022

SpTFS: Sparse Tensor Format Selection for MTTKRP via Deep Learning

Qingxiao Sun*, Yi Liu*, Ming Dun*, Hailong Yang*‡, Zhongzhi Luan*, Lin Gan†,
Guangwen Yang†, and Depei Qian*
Beihang University*, Tsinghua University†, China
State Key Laboratory of Mathematical Engineering and Advanced Computing, Wuxi, China‡
{qingxiaosun,yi.liu,dunming0301,hailong.yang,zhongzhi.luan,depeiq}@buaa.edu.cn*, {lingan,ygw}@tsinghua.edu.cn†

***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 fix-sized matrices and specific features. Then, TnsNet combines CNN and the feature layer to accurately predict the optimal format. The experimental results show that *SpTFS* achieves prediction accuracy of 92.7% and 96% on CPU and GPU respectively.**

***Index Terms*—MTTKRP, Sparse Tensor, Format Selection, Convolutional Neural Network**

I. 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]–[3] and numerical analysis [4]–[6]. 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 [7], and later became popular in the field of chemometrics [8]. In recent years, tensor decomposition has received wide attention due to its applicability in broader areas such as neuroscience [9], recommendation systems [10], and machine learning [11].

Canonical polyadic decomposition (CPD) [12] 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) [13], 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 [13]–[16].

Although the parallelization can significantly improve the performance of MTTKRP, it is constrained by the sparsity patterns and hardware characteristics (e.g., CPU and GPU). Therefore, different sparse tensor formats have been proposed to improve the computation performance with co-designed storage and algorithm that adapts to the sparsity and hardware. Coordinate (COO) [17] 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) [18] uses a tree structure to store non-zero values and their index pointers, similar to Compressed Sparse Row (CSR) [19] in matrices. In addition, hardware-specific extensions based on COO and CSF have been proposed, such as HiCOO and HB-CSF [20]–[22]. 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. However, such a problem is proved to be perfectly suited for deep learning techniques, which have demonstrated their promising success in image classification and object detection. Specifically, 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 [23]–[27]. Despite the great success, how to use CNN to solve the problems in high-performance computing is still an ongoing research field. Related to this study, previous works applied CNN to sparse matrix format selection for optimizing SpMV [28]–[30] and SpGEMM [31]. However, such approaches cannot be directly applied in tensor format selection due to higher dimensional data to deal with.

Since the tensor usually stores higher-dimensional data, it cannot be directly fed to CNN that commonly handles data no more than three dimensions (e.g., weight, height, and channel for image data). Although high-dimensional convolution has been proposed [32]–[34], there are two reasons it cannot be used in our work. Firstly, the irregularity of tensor data deteriorates the computation efficiency of convolution operations, which leads to unacceptable training overhead [35]. Secondly, the popular deep learning frameworks such as TensorFlow [36] and PyTorch [37] can only support up to 3-D convolution layers, which cannot satisfy the need for higher dimensional tensors. Therefore, the format selection of sparse tensor poses special challenges to CNN: *1)* the higher-dimensional tensors need to be lowered into matrices in order to adopt CNN; *2)* the

SC20, November 9-19, 2020, Is Everywhere We Are
978-1-7281-9998-6/20/$31.00 ©2020 IEEE

lowered matrices need to be represented as fixed-size input for CNN, without losing the sparsity patterns; *3)* the CNN network needs to be re-designed to compensate for the missing sparsity features during matrix representation.

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 network through various scaling methods. In addition, we re-design the CNN network by adding an additional feature layer to compensate for the sparsity features lost during matrix representation. We evaluate *SpTFS* on both CPU and GPU platforms to prove its effectiveness in predicting optimal tensor format. To the best of our knowledge, this is the first work to automatically select the optimal sparse storage format for tensor computation.

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. Moreover, TnsNet integrates an additional feature layer that compensates for the sparsity feature lost during tensor transformation.
- 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 II presents the background of this paper. Section III presents the details of *SpTFS* methodology. Section IV presents the evaluation results of *SpTFS*. Section V discusses the related work, and Section VI concludes this paper.

II. Background

A. Tensor Notation

Tensor denotes the array with multiple dimensions [38] and is the generalization of matrix and vector. Specifically, 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 (e.g., Higher-Order Statistics (HOS) for the multivariate cases [39]), chemometrics (e.g., the excitation-emission spectroscopy matrices [8]) and image/video rendering (e.g., RGB color images and 3D light field displays [40]). Since finding the exact rank of a tensor is an NP-hard problem [41], 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 operation to describe the concepts and related mathematics about tensor decomposition without losing generality. All notations for vectors, matrices, and high-dimensional tensors are shown in Table I, 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.

TABLE I: 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.

Canonical polyadic decomposition (CPD) [12] is one of the most widely-applied tensor operations, 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 Equation 1. To solve the CPD, one of the most popular approaches is to utilize Alternating Least Squares (ALS) [38], 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 Equation 2. The main bottleneck in the CPD-ALS algorithm for sparse tensors is MTTKRP [13], which can be formulated as Equation 3.

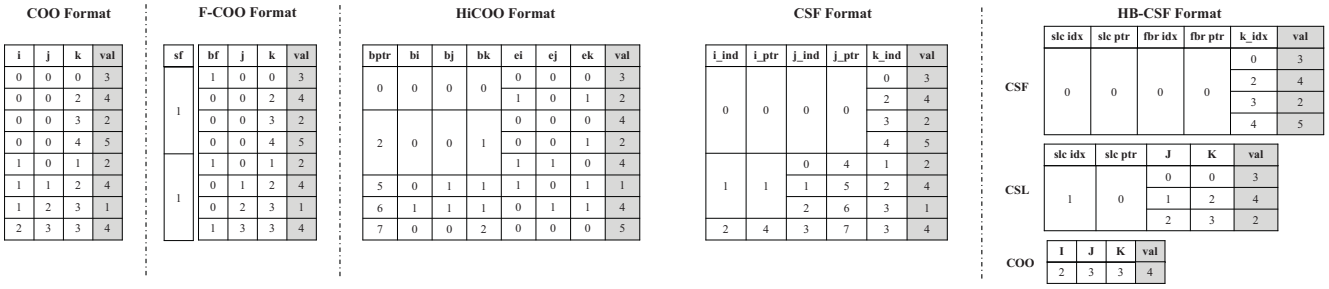
$$\mathcal{X}(i,j,k) = \sum_{f=1}^{F} \mathbf{A}(i,f)\mathbf{B}(j,f)\mathbf{C}(k,f) \tag{1}$$

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

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

B. Sparse Tensor Storage Formats

1) COO and COO-based Formats: Coordinate format (COO) [17] is an intuitive format for storing sparse tensor. 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



(a) COO and COO-based formats

(b) CSF and CSF-based formats

Fig. 1: The comparison of sparse tensor storage formats, where F-COO and HiCOO (where parameter $B = 2$) are COO-based formats, and HB-CSF is CSF-based format. The val stands for the non-zero value.

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 [20] and HiCOO [21], as shown in Figure 1 (a). 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 [42] 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 Equation 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.

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

2) CSF and CSF-based Formats: Compressed Sparse Fiber (CSF) [18] extends the Compress Sparse Row (CSR) [19] format used to store sparse matrices. CSF maintains a tree-based structure and consists of six arrays, as shown in Figure 1 (b). 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. Thus, CSF is extended to HB-CSF [22] for better adaption on GPU, whose structure is shown in Figure 1 (b). 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 efficiency on GPU due to fine-grained tensor partition.

Figure 2 presents the performance comparison of real-world sparse tensors using different storage formats for MTTKRP. The tensor datasets are adopted from FROSTT [43] and HaTen2 [44], including ten 3-D tensors and six 4-D tensors. The detailed descriptions of different tensor formats are provided in Section IV-A. Two observations can be drawn from Figure 2: *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 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. The selection of 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.

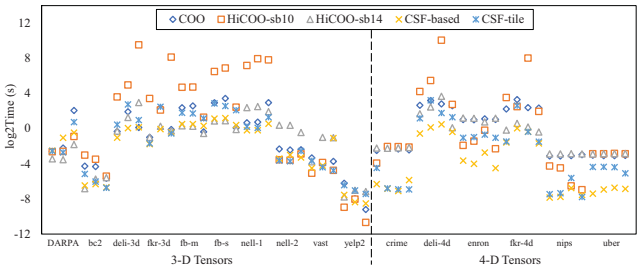


Fig. 2: The execution time of MTTKRP on real-world sparse tensors across all modes on CPU. The number of columns each tensor contains depends on its order.

C. Supervised Learning-based Techniques

Supervised learning has attracted lots of attention due to its outstanding performance on image classification tasks [23]–[27], [45]. Among the machine learning methods, gradient

boosted decision tree (GBDT) [46] has been adopted in many application scenarios. XGBoost [47] is an efficient implementation of GBDT (based on CART [48]), which 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 tasks such as classification and detection. The input of CNN can be a matrix with non-linear patterns (e.g., pixels of an image), and the output is the probability of the input belonging to each class. However, 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, in order to faithfully represent the patterns.

III. Methodology

A. Design Overview

In this section, we propose an automatic tensor format selection framework *SpTFS*, that can predict the optimal tensor format for MTTKRP with re-designed CNN network. As shown in Figure 3, the *SpTFS* consists of three important components including tensor transformation, feature extraction and TnsNet (a re-designed CNN network). The tensor transformation component converts the sparse tensors into fix-sized matrices through tensor lowering (Section III-B) and matrix representation (Section III-C). Inspired by [31], feature extraction is used to capture lost tensor features during tensor transformation, which is then fed into the fully connected layer in TnsNet (Section III-D). 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, which predicts the optimal format for a tensor on the target hardware platform.

Due to the limited number of publicly available datasets of sparse tensors, we randomly select sparse matrices from the SuiteSparse [49] and combine them to generate more datasets of sparse tensors. SuiteSparse has been widely used in evaluating the performance of matrix computation [50]–[54] as well as the accuracy of matrix format selection [29]–[31], [55]–[57]. 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. After that, the generated sparse tensor datasets are processed through tensor transformation and feature extraction in order to be used for training the CNN network.

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.

B. 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 Figure 4, 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., $\mathcal{X}_{(i,:,:)}$) (Figure 4 (b)), which denotes the subarrays with the i index fixed. Next, we map the non-zero values of all slices to a slice (Figure 4 (c)), 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 Equation 5, where $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$ and $\mathbf{A} \in \mathbb{R}^{J \times K}$.

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

Flattening is to unfold the tensor using matricization. Figure 4 (d) shows the mode-1 flattening of $\mathcal{X}$ is $\mathcal{X}_{(1)}$ in Equation 3. For any element in $\mathcal{X}$, its flattening to the matrix can be formulated in Equation 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 Equation 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 Equation 8, where tensor element $(i_1, i_2, \cdots, i_N)$ maps to matrix element (i_n, j) [38].

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

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

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 fix-sized input for the network through matrix representation (Section III-C).

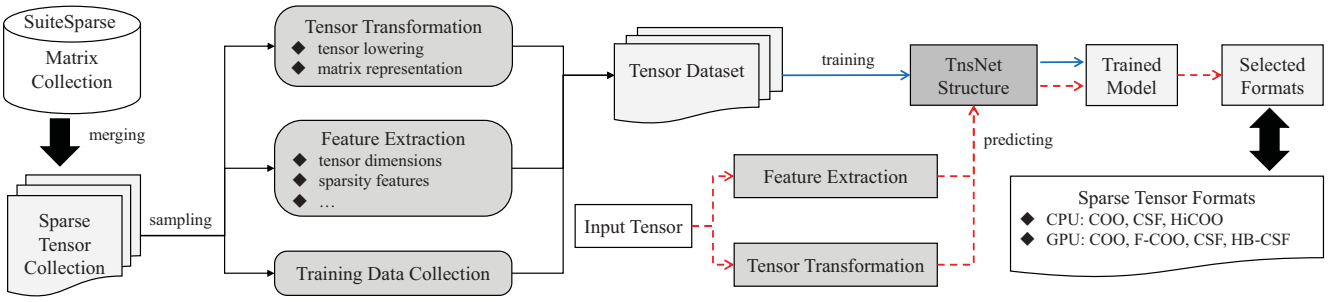


Fig. 3: The design overview of *SpTFS*.

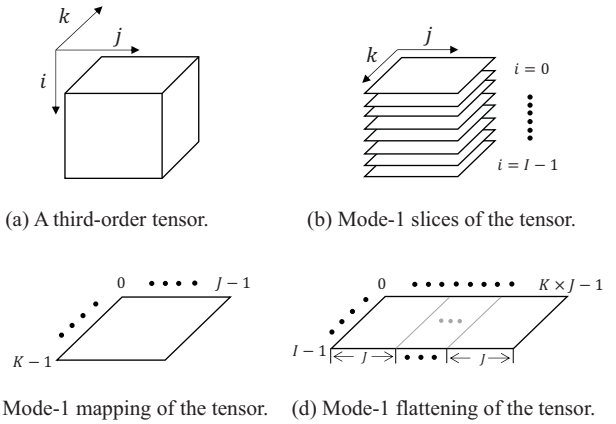


Fig. 4: The process of lowering a high-dimensional tensor into matrices using mapping or flattening.

C. Matrix Representation

The matrices generated through tensor lowering are irregular in size, and we need to scale the matrices into fix-sized input (e.g., 128×128) for the network. Inspired by [28], 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 [28] further captures the distance between an element and the diagonal of the original matrix while losing parts of the sparsity distribution.

As shown in Figure 5, we illustrate the matrix representation methods by the example of mapping the 8×8 matrix to 4×4 matrices. For the density representation (Figure 5 (b)), 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 (Figure 5 (c)). 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.

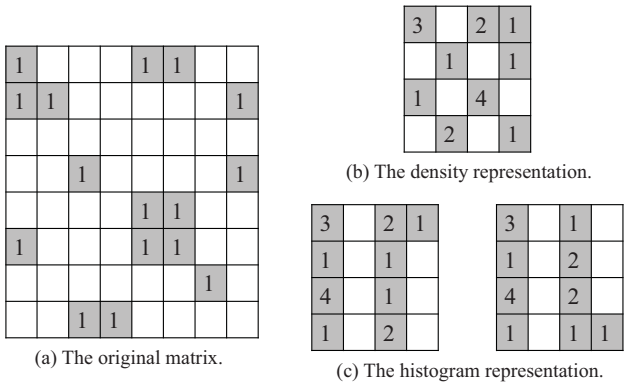


Fig. 5: Different ways for representing a matrix.

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 choose to add a feature layer to the network structure to compensate for the loss of tensor features (Section III-D).

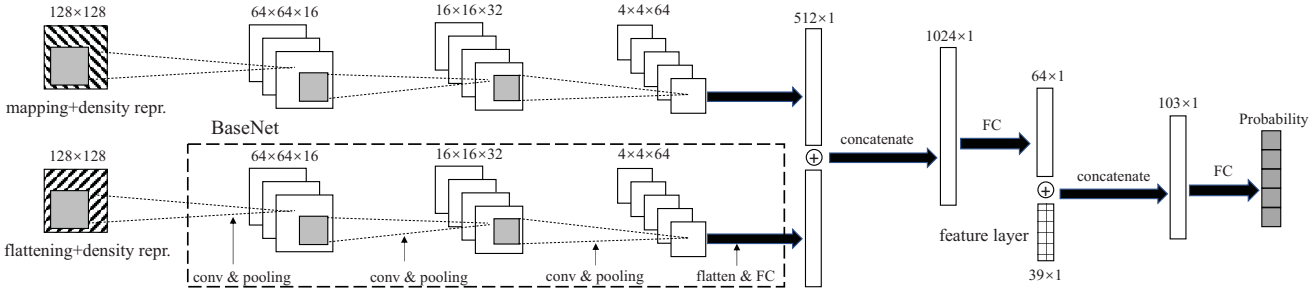


Fig. 6: The structure design of TnsNet.

D. Network Structure Designs

As shown in Figure 6, TnsNet combines CNN and feed-forward neural network (FFNN) [58] to predict the optimal storage format for a tensor. The inputs of TnsNet include the fix-sized matrices generated through tensor lowering and matrix representation. Besides, tensor features are also extracted and fed to TnsNet. Here, we take the density representation as an example, where the BaseNet includes all convolution and pooling layers of TnsNet. The advantage of using BaseNet is to provide better portability when handling higher-order tensors. In such a case, the design and hyper-parameters of the BaseNet can be re-used without any modification. For the matrices generated after tensor transformation, we use them as inputs to the BaseNets separately and concatenate the outputs at the fully connected layer.

After BaseNet, the outputs from two tensor lowering methods merge together as joint features of matrices and flow into the fully connected layer. The joint features are further concatenated with the feature layer, where the feature layer can supplement the missing sparsity features of the original tensor. The feature layer reflects the memory layout (e.g., sparsity) and computation characteristics (e.g., nnz) using the specific tensor format. Table II summarizes all the sparsity features used in the feature layer, including the global and local features of the tensor. The global features include the tensor's mode sizes, nnz, sparsity, and features related to nnz per row-n (e.g., aveNnzPerRow_n). Whereas, the local features are those related to slices and fibers, such as nnz per slice, nnz per fiber and fibers per slice. The local features can significantly affect the performance of MTTKRP of CSF-based tensor formats. Since the value of the input feature (e.g., nnz) may vary significantly across tensors, we normalize each input feature to the range of [0,1] by dividing each value by the maximum value of the feature. Ultimately, the outputs of TnsNet are the probability of each tensor format to achieve the optimal performance. For a particular tensor, the tensor format with the highest probability is predicted to be optimal.

As shown in Figure 7, TnsNet using the histogram representation replaces each input matrix with a row matrix and a column matrix. These two matrices can be seen as different channels of an image (e.g., RGB). For the matrices generated after the same method of tensor transformation, we adopt the late-merging structure [57] in our TnsNet. That is, each matrix is used as the input of BaseNet, and then the output features are merged into the fully connected layer (similar to handling higher-order tensors). After merging the features of the row and column, the remaining structure stays unchanged.

TABLE II: 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 number of NNZ per row-n.
maxNnzPerRow_n	Maximum number of NNZ per row-n.
minNnzPerRow_n	Minimum number of NNZ per row-n.
devNnzPerRow_n	The deviation of the number of NNZ per row-n.
adjNnzPerRow_n	Average difference between NNZ of adjacent row-ns.
aveNnzPerSlice	Average number of NNZ per slice.
maxNnzPerSlice	Maximum number of NNZ per slice.
minNnzPerSlice	Minimum number of NNZ per slice.
adjNnzPerSlice	Average difference between NNZ of adjacent slices.
devNnzPerSlice	The deviation of number of NNZ per slice.
aveNnzPerFiber	Average number of NNZ per fiber.
maxNnzPerFiber	Maximum number of NNZ per fiber.
minNnzPerFiber	Minimum number of NNZ per fiber.
devNnzPerFiber	The deviation of number of NNZ per fiber.
adjNnzPerFiber	Average difference between NNZ 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 difference between fibers of adjacent slices.

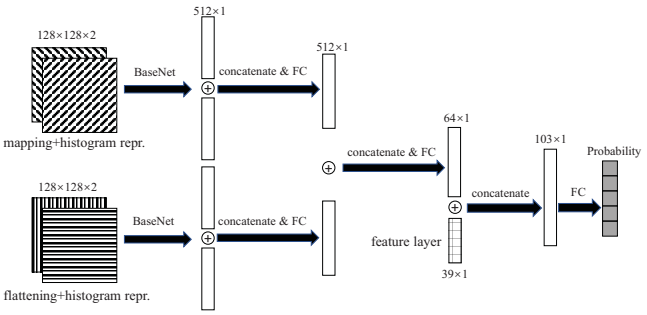


Fig. 7: The design of TnsNet using histogram representation.

When porting the TnsNet to a hardware platform different from where it is trained, new training data needs to be collected

to reflect the new hardware characteristics. When adding new tensor data to the dataset, the same tensor transformation procedures are applied to convert the tensor data into fix-sized matrices to re-train TnsNet. Then, the TnsNet is re-trained with the new data collected on the particular platform.

IV. Evaluation

A. Experiment Setup

1) Hardware and Software Platforms: As shown in Table III, we evaluate the effectiveness of *SpTFS* on two hardware platforms, where the CPU is an Intel Xeon processor and the GPU is an Nvidia RTX Turing processor. The GPU is also utilized to speed up the process of training and prediction. TnsNet is built using TensorFlow release version 1.15 [36].

TABLE III: Hardware platforms used for evaluation.

	CPU	**GPU**
Model	Intel Xeon Silver 4110 CPU	GeForce RTX 2080 Ti
Frequency	2.1GHz	1.3GHz
Cores	16	4352 (68 SMs)
Cache	32KB L1, 1MB L2, 11MB L3	5.5 MB L2
Memory	192GB DDR4	11GB GDDR6
Bandwidth	230.4GB/s	616.0GB/s
OS/Driver	CentOS Linux release 7.6	GPU Driver 440.33
Compiler	gcc-4.8	nvcc-10.2

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 [59], whereas the COO and HiCOO implementations are adopted from ParTI! [21]. 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 [21]. For example, HiCOO-sb10 means that the size of the superblock is 2^{10}. For MTTKRP on GPU, the tensor formats we evaluate are COO, F-COO, CSF, and HB-CSF. Except for F-COO, we adopt the implementations of all tensor formats from [22]. Since there is no public F-COO implementation available, we develop our F-COO implementation based on [20]. Due to severe load imbalance, F-COO exhibits relatively poor performance, which has also been confirmed in [22]. 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 on GPU. 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 [49]. 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 [43] and HaTen2 [44] to the evaluation dataset (details listed in Figure 2).

3) Cross Validation: We use the 5-fold cross validation method to determine hyperparameters and evaluate the accuracy of the models. The validation method is widely used in literatures [28]–[30]. We compare the design of TnsNet with the XGBoost [47] machine learning methods, including GBDT and GBlinear, which are based on CART [48] and the linear model, respectively. The input features of GBDT and GBLinear are listed in Table II, which are the same to the feature layer of TnsNet. The GBDT and GBLinear are trained for CPU and GPU platforms separately. For TnsNet, 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 rank size to 16. Besides, we set the number of threads to 16 when on CPU, and the thread block size to 256 when on GPU.

B. Results for Prediction

1) Prediction Accuracy: Table IV reports the prediction accuracy of the four designs (density/histogram representation w/o feature layer) of TnsNet and two machine learning methods (GBDT and GBLinear) on CPU. The *recall* represents the percentage of correct results returned and *prec.* represents the precision. The third column in Table IV represents the number of sparse tensors achieving the optimal performance in the corresponding format. TnsNet with density representation (*density repr.*) and feature layer (*FL*) achieves the highest prediction accuracy in general. The overall prediction accuracy of TnsNet (*density repr.+FL*) in mode-1, mode-2 and mode-3 is 92%, 93% and 93%, respectively. In comparison, the machine learning method GBDT only achieves 85%, 84%, and 89% prediction accuracy. In addition, TnsNet (*density repr.+FL*) also exhibits stable recall rate and prediction accuracy for all tensor formats. In comparison, the recall rate and precision accuracy of GBDT deteriorate to 49% (in mode-1 with HiOO-sb14 format) and 74% (in mode-3 with CSF-tile format) in the worst case. For the histogram representation, the training accuracy is high whereas the prediction accuracy is low, which indicates it suffers from overfitting.

The difference in prediction accuracy among the tensor formats is due to the amount of training data and the complexity of the format patterns. With more training data, TnsNet can better learn the data patterns in the corresponding tensor format. The prediction accuracy of HiCOO-sb10 is better than HiCOO-sb14, because the training dataset contains more profiling data for HiCOO-sb10 than HiCOO-sb14 (4,454 vs. 250 on average across all modes). Besides, the complex format patterns also affect the prediction accuracy of TnsNet. In mode-1, the number of training data for COO and CSF-based format is similar, but the prediction accuracy of COO is much higher than CSF-based. This is because the performance of CSF is strongly correlated with the spatial distribution of the tensor data. However, the spatial distribution of tensor data is partially lost during the tensor transformation. Due to space constraint, we will only present the evaluation results of TnsNet (*density repr.+FL*) and GBDT in the rest of the paper.

TABLE IV: Prediction accuracy of different designs of TnsNet and machine learning methods on CPU.

MTTKRP Mode	SpTensor Format	Ground Truth	only density repr.		only histogram repr.		density repr.+FL		histogram repr.+FL		GBDT		GBLinear	
			recall	prec.	recall	prec.	recall	prec.	recall	prec.	recall	prec.	recall	prec.
Mode-1	COO	740	0.95	0.88	0.6	0.65	0.97	0.91	0.56	0.68	0.91	0.78	0.9	0.66
	HiCOO-sb10	4143	0.9	0.91	0.86	0.82	0.95	0.92	0.83	0.82	0.88	0.88	0.87	0.68
	HiCOO-sb14	328	0.66	0.87	0.63	0.7	0.73	0.88	0.67	0.66	0.49	0.82	0.22	0.91
	CSF-based	3912	0.93	0.88	0.84	0.83	0.91	0.94	0.84	0.81	0.9	0.86	0.56	0.82
	CSF-tile	732	0.67	0.93	0.77	0.83	0.78	0.85	0.78	0.77	0.52	0.88	0.27	0.72
	Total	9855	0.89		0.8		**0.92**		0.79		0.85		0.71	
Mode-2	COO	656	0.92	0.93	0.58	0.59	0.95	0.96	0.6	0.57	0.87	0.78	0.57	0.64
	HiCOO-sb10	2544	0.89	0.91	0.86	0.9	0.9	0.93	0.86	0.89	0.82	0.9	0.54	0.86
	HiCOO-sb14	170	0.94	0.85	0.77	0.76	0.96	0.85	0.77	0.78	0.86	0.83	0.42	0.82
	CSF-based	5527	0.94	0.92	0.84	0.81	0.96	0.94	0.83	0.83	0.89	0.85	0.94	0.67
	CSF-tile	958	0.8	0.92	0.75	0.78	0.86	0.94	0.75	0.75	0.6	0.77	0.03	0.44
	Total	9855	0.91		0.8		**0.93**		0.8		0.84		0.69	
Mode-3	COO	831	0.67	0.94	0.63	0.71	0.84	0.95	0.68	0.7	0.83	0.81	0.37	0.7
	HiCOO-sb10	6675	0.97	0.92	0.87	0.86	0.96	0.95	0.87	0.88	0.95	0.94	0.97	0.72
	HiCOO-sb14	253	0.75	0.91	0.76	0.74	0.77	0.89	0.74	0.8	0.63	0.84	0.24	0.56
	CSF-based	1912	0.84	0.84	0.7	0.7	0.9	0.86	0.75	0.67	0.8	0.81	0.13	0.73
	CSF-tile	184	0.85	0.92	0.72	0.67	0.88	0.91	0.76	0.71	0.69	0.74	0.04	0.23
	Total	9855	0.9		0.79		**0.93**		0.8		0.89		0.71	
All-Mode	Overall		0.9		0.8		**0.93**		0.8		0.86		0.7	

Table V 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 96% prediction accuracy on average, whereas GBDT only achieves 90%. Since there are fewer tensor formats available on GPU, the prediction accuracy of both TnsNet and GBDT is higher than on CPU.

Due to the limited number of real-world tensors, we repeat the 5-fold cross validation of the entire datasets for 10 times and obtain the average accuracy of 93% and 92% on CPU and GPU, respectively. The results further prove the effectiveness of TnsNet on real-world tensors.

TABLE V: Prediction accuracy of TnsNet and GBDT on GPU.

MTTKRP Mode	SpTensor Format	Ground Truth	TnsNet		GBDT	
			recall	prec.	recall	prec.
Mode-1	COO	6063	0.98	0.98	0.94	0.98
	CSF	1640	0.95	0.91	0.87	0.73
	HB-CSF	2152	0.96	0.98	0.88	0.91
	Total	9855	**0.97**		0.92	
Mode-2	COO	6191	0.97	0.99	0.93	0.98
	CSF	1650	0.94	0.9	0.89	0.71
	HB-CSF	2014	0.97	0.94	0.87	0.91
	Total	9855	**0.97**		0.91	
Mode-3	COO	5967	0.97	0.97	0.94	0.97
	CSF	1796	0.85	0.86	0.81	0.66
	HB-CSF	2092	0.92	0.9	0.78	0.88
	Total	9855	**0.94**		0.88	
All-Mode	Overall		**0.96**		0.9	

2) Training/Testing Loss: The loss is used to indicate the degree of prediction deviation from the true value. During training, the loss decreases, and the accuracy increases. Figure 8 compares the losses and accuracies during training and testing on CPU and GPU. We can observe that the loss curve during training is more oscillating than during testing, but the overall trend is similar. As the number of training steps increases, the accuracy and loss of TnsNet gradually converge to an almost constant value. Compared to CPU, the training on GPU converges faster due to fewer tensor formats. The results show that TnsNet quickly learns the tensor features and maintains convergence on both CPU and GPU platforms. The results also indicate that the training data after applying our proposed tensor transformation is still able to retain the important features for identifying the optimal tensor format.

C. Results for Speedup

The performance speedup of MTTKRP using the tensor format predicted by TnsNet on CPU and GPU is presented in Figure 9 and Figure 10, respectively. The performance using the COO format and CSF-based/CSF format are chosen as the baseline. We can observe that TnsNet achieves higher performance speedup than GBDT in all modes on both platforms. On CPU, TnsNet achieves an average speedup of 4.56× and 2.06× over COO format and CSF-based format, respectively. The low performance speedup of TnsNet over CSF-based format is due to the fact that CSF format performs better than the COO format in most cases. On GPU, TnsNet achieves an average speedup of 1.58× and 2.08× over COO format and CSF format, respectively. The low performance speedup of TnsNet over COO format is due to the fact that COO format already achieves optimal performance in most cases (refer to Table V). Nevertheless, the results further demonstrate that choosing the right tensor format is critical to achieving the optimal performance for MTTKRP.

Figure 12 shows the speedup distribution of TnsNet compared to GBDT, with the cases where the two models give different predictions for the optimal tensor formats. The left part of the vertical dash line indicates the cases when using the formats predicted by TnsNet achieves better performance than GBDT. TnsNet is able to improve the performance of 72% of the tensors. For 4.8% of the tensors, TnsNet achieves 5× speedup over GBDT. Moreover, the speedup of TnsNet can reach up to 420×, 341× and 557× in mode-1, mode-2, and mode-3, respectively. The results show that TnsNet is more accurate for predicting the optimal tensor format, and thus leads to significant speedup compared to GBDT.

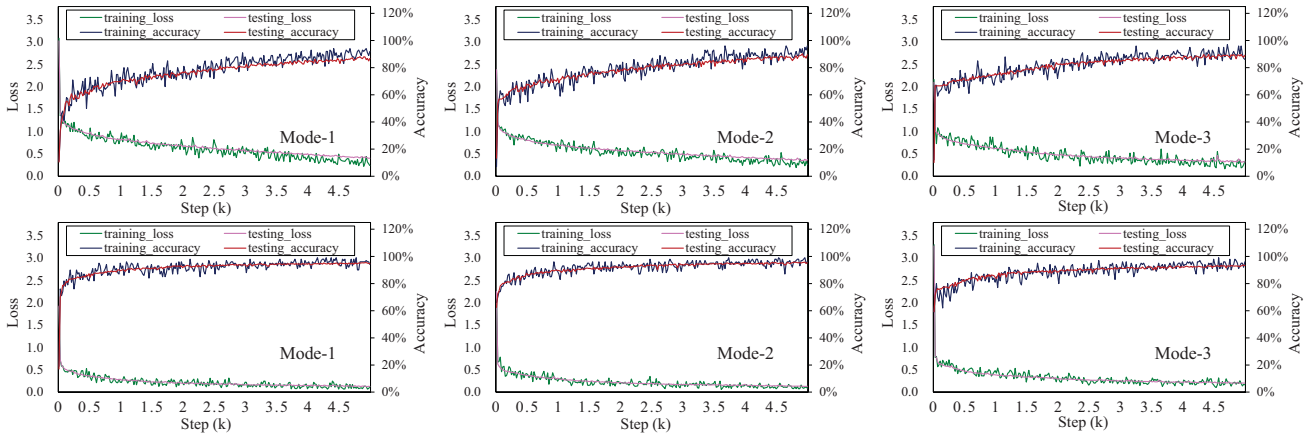


Fig. 8: Loss and accuracy of TnsNet during training and testing on CPU (above) and GPU (below).

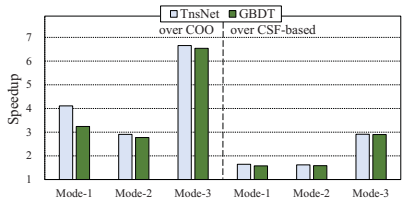


Fig. 9: Speedup of TnsNet and GBDT over the baseline COO format and the CSF-based format on CPU.

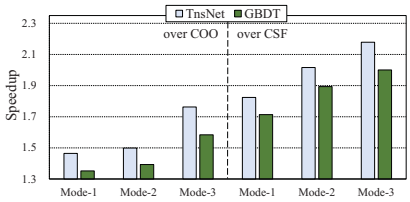


Fig. 10: Speedup of TnsNet and GBDT over the baseline COO format and the CSF format on GPU.

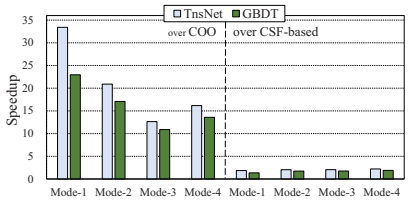


Fig. 11: Speedup of TnsNet and GBDT over the baseline COO format and the CSF-based format for 4-D tensors.

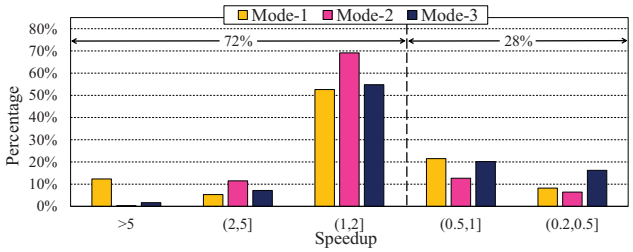


Fig. 12: Speedup distribution of TnsNet over GBDT on CPU.

D. Applying to Higher-order Tensors

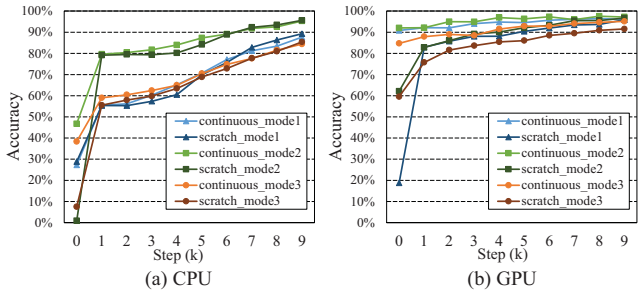
Since the tensor lowering method can be applied to tensors with any dimension, it is easy for our approach to be used with higher-order tensors. To demonstrate, we apply our approach to predict the optimal storage format for fourth-order tensors. Unlike the third-order tensor, a fourth-order tensor will eventually generate $C_3^2 = 3$ matrices after mapping in a certain mode. Specifically, the TnsNet achieves an average accuracy of 88% across all modes, whereas GBDT only achieves 58%. Since the sparsity distribution of higher-order tensors is more complicated, it is difficult to capture the accurate correlations between the data patterns and the optimal format due to the loss of sparsity features. In addition, TnsNet leverages the feature layer to compensate for the loss of sparsity features during tensor transformation, which can retain high prediction accuracy. For the real-world tensors, TnsNet achieves an average accuracy of 91% across all modes.

The performance speedup for fourth-order tensors is shown in Figure 11. On average, TnsNet and GBDT achieve 20.8× and 16.1× speedup over COO format across all modes, respectively. And for the CSF-based format, TnsNet and GBDT only achieve an average speedup of 2.05× and 1.69×, 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. Moreover, the performance gap among different storage formats on the fourth-order tensors is larger than that on third-order tensors. Such a performance gap might become even larger as the tensor order increases. This proves the necessity of our work for accurately predicting the optimal storage format for tensors.

E. Applying to more Hardware Platforms

To demonstrate the generality of our approach, we evaluate *SpTFS* on two new hardware platforms, including an Intel Xeon E5-2650v4 CPU with 24 cores, and an Nvidia Tesla V100 GPU with 80 SMs. Specifically, we collect training data on these new platforms and use TnsNet to re-train the model to predict the optimal tensor formats. Here, we utilize two methods for re-training: *1)* training the models from scratch, and *2)* re-training the models based on pre-trained ones.

Figure 13 shows the prediction accuracy when applying TnsNet on these new platforms with the above two re-training methods. We can observe that both methods eventually achieve similar prediction accuracy of around 90%. On GPU, using



(a) CPU (b) GPU

Fig. 13: Prediction accuracy of different re-training methods on new hardware platforms, where accuracy at $step = 0$ is without re-training. *scratch_modeX* and *continuous_modeX* mean training the model from scratch and re-training from a pre-trained model, respectively for *mode-X* computation.

continuous training, the prediction accuracy already reaches over 85% across all modes when re-training starts. However, on CPU, the *continuous* training has no significant advantage over training from scratch. The reason can be attributed to the different similarity of the training data collected on the platforms. We measure the similarity by comparing the optimal formats for tensors on the new and original platforms. On GPUs, the similarities of the training data across all three modes are 91%, 93%, and 90%, respectively. In contrast, on CPUs, the similarities only reach 29%, 47%, and 37%, respectively. In general, the *SpTFS* can be applied to other hardware platforms with stable prediction accuracy.

F. Overhead Analysis

We discuss the overhead of *SpTFS* from both training and prediction aspects. For training, it takes about 7 minutes to train TnsNet with 7,886 input tensors (5-fold cross validation) on our experiment server. Since the training is only performed once, the training overhead is negligible. For prediction, the overhead of *SpTFS* can be further divided into three parts: *1)* tensor lowering and matrix representation, *2)* tensor feature extraction, and *3)* tensor format prediction using TnsNet. Illustrated in Section II-A, the number of iterations of CPD-ALS (MTTKRP algorithm for our interest) depends on when it reaches the convergence (i.e., the error is less than the $threshold$). Here we set the $threshold$ to 10^{-5} in accord with existing research [59]. Based on our empirical study, the average number of iterations for CPD-ALS with the real-world tensors is 15.8. Therefore, we analyze the prediction overhead of *SpTFS* amortized to 15 iterations of CPD-ALS.

Figure 14 and Figure 15 show the performance breakdown of *SpTFS* normalized to the performance using baseline COO format on CPU and GPU, respectively. Note that the cases where COO is already the optimal format are not taken into account. The average performance is presented for 7 groups, where the number of non-zeros ranges from 10^0 to 10^7 with a stride of $\times 10$. The absolute value and error bar for each performance result are also presented. The results less than 100% mean *SpTFS* achieves better performance even with prediction overhead taken into account. For CPU, the performance results for all groups are under 100%. However, as the number of non-zeros increases, the performance advantage using *SpTFS* diminishes (normalized results closing to 100%). This is because the tensor transformation and feature extraction overhead increase proportionally as the number of non-zeros. However, the performance benefit of adopting optimal tensor format becomes less due to the increasing non-zeros (the tensor becomes dense). Together, the above trends offset the performance profit of predicting the optimal tensor format with *SpTFS*. In addition, the format prediction overhead using histogram representation is 2.1× higher on average than using density representation.

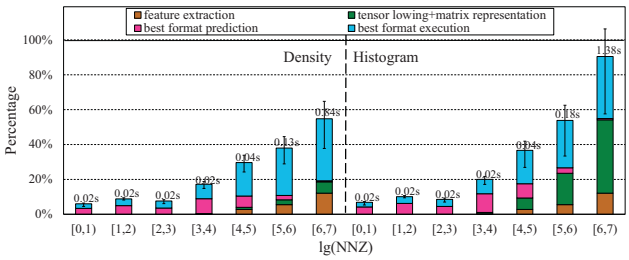


Fig. 14: Performance breakdown of *SpTFS* normalized to the baseline COO format on CPU.

For GPU, the results of most groups are still below 100%. However, the performance improvement using *SpTFS* becomes smaller on GPU. This is because the performance gap among different GPU formats is not as large as that of CPU formats. Different from CPU, the trend of performance speedup is not monotonically decreasing with the increasing number of non-zeros. This is because as the number of non-zeros increases, more GPU threads are launched for better parallelism until the hardware resources are saturated. Whereas similar to CPU, the sampling overhead of *SpTFS* takes a larger portion of the execution time when the number of non-zeros increases.

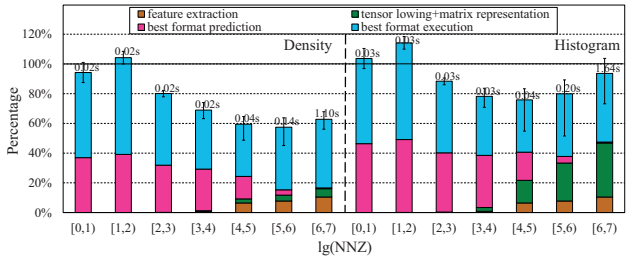


Fig. 15: Performance breakdown of *SpTFS* normalized to the baseline COO format on GPU.

In sum, it is effective to improve performance by using *SpTFS* to predict the optimal storage format for sparse tensors. Especially when the convergence condition of CPD-ALS becomes stricter (which means more iterations are required), the prediction overhead will be further diluted compared to the performance gain. In addition, when the rank size is larger, the proportion of prediction overhead will reduce significantly

compared to the execution time of MTTKRP. Currently, we do not consider the format conversion overhead in our evaluation. Such overhead can be amortized if the converted tensors are to be used by multiple users. In such a case, only a one-time process is required to convert the tensors into all formats, which eliminates the overhead for runtime format conversion. We leave the tensor format prediction mechanism with input awareness for future work.

G. Analysis of Network Designs

1) Mapping vs. Flattening: Mapping and flattening reflect the vertical and horizontal sparsity distribution of the tensors in a certain mode, respectively. Figure 16 presents the prediction accuracy of TnsNet with only mapping or flattening method. For reference, the prediction accuracy of the best TnsNet design (mapping+flattening+density repr.) is also shown in the figure (pink bars). We can observe that every single method alone is unable to achieve the best prediction accuracy. Specifically, the average prediction accuracy when using mapping or flattening alone is 82% and 84%, respectively. Therefore, it is necessary to combine mapping and flattening in order to capture more tensor patterns for better prediction accuracy.

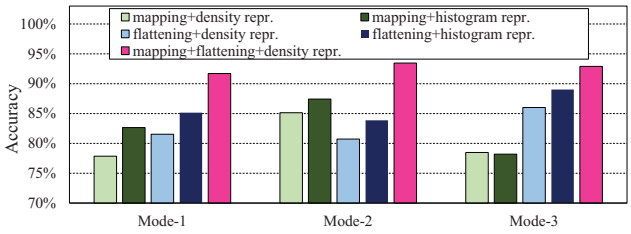


Fig. 16: The prediction accuracy of TnsNet using only mapping or flattening method (+ density/histogram representation).

2) Density vs. Histogram: Density and histogram are the two methods considered in our approach for matrix scaling. Figure 16 presents a detailed comparison of the two representations when using one of the lowering method (mapping or flattening). When using only one tensor lowering method, the prediction accuracy of TnsNet with histogram representation is 3.4% higher than the density representation on average. This is because using only one tensor lowering method, it is able to reduce the number of weights and biases to be learned during training. The low number of weights and biases avoids the overfitting problem in Table IV.

3) CNN Ablation Study: To study the importance of the CNN component in TnsNet, we conduct experiments by removing the CNN and only using the feature layer as input to TnsNet. The evaluation results show that TnsNet achieves 54% prediction accuracy on average, which proves the necessity of CNN in TnsNet for improving prediction accuracy. In addition, we modify TnsNet to a deeper network by adding one extra feature layer after the original feature layer. The two feature layers have the same configuration. The evaluation results show that TnsNet achieves 93.9% prediction accuracy on average, but with 5.2% increase of training time. Therefore, when TnsNet goes deeper, there is a tradeoff between prediction accuracy and training overhead.

4) Input Resolution: The input resolution is the size of the matrices fed to the network. Higher resolution will retain more tensor patterns, and ultimately improve the prediction accuracy. However, the high resolution also means high overhead during the processes of tensor lowering and matrix representation. When the input resolution increases from 64×64 to 128×128, the accuracy of TnsNet improves from 83.5% to 92.7%. However, as the input resolution increases to 256×256, the accuracy improves slightly (93.3%). Therefore, to balance the prediction accuracy and tensor transformation overhead, we choose the input resolution of 128×128 for TnsNet.

V. Related Work

Optimization for CPD Algorithm. For optimizing the CPD algorithm on CPUs, Kang *et al.* [14] utilized the MapReduce framework [60] to implement the CPD algorithm on CPUs. Choi *et al.* [61] implemented both Alternating Least Squares (ALS) and Gradient Descent (GD) algorithm for CPD in DFacTo on CPUs. Smith *et al.* [59] developed SPLATT that accelerated the CPD algorithm through efficient cache-friendly tilting and reordering under CSF format [18]. Choi *et al.* [13] further optimized the MTTKRP process in SPLATT through fine-grained blocking techniques to improve the cache hit rate. Li *et al.* [21] developed the HiCOO format that is compact and mode-generic to improve the performance of the CPD algorithm. Vervliet *et al.* [62] proposed a fully randomized sampling approach to combine the CPD with Stochastic Gradient Descent (SGD). Cheng *et al.* [15] proposed SPALS, which samples the MTTKRP process to reduce the computational power. For optimizing the CPD algorithm on GPUs, Liu *et al.* [20] introduced F-COO, which adds flag arrays for eliminating atomic operations. Nisa [22] developed HB-CSF to alleviate the load imbalance caused by tree-based CSF.

Sparse Matrix Format Selection. Methods for handling this problem can be divided into traditional machine learning methods [55]–[57], [63] and deep learning methods [28]–[31]. SMAT [63] generated the learning model offline based on the ruleset classifier to predict the best combination of parameters. Sedaghati *et al.* [55] developed the learning model using the decision tree classifier to select the best matrix format on GPUs. Benatia *et al.* [56] leveraged the multi-class Support Vector Machine (SVM) classifier to select the best sparse format for each input matrix. Zhao *et al.* [57] proposed a two-stage scheme using regression tree-based models to construct overhead-conscious selectors of sparse matrix formats. For deep learning methods, Zhao *et al.* [28] used CNN for the first time to implement sparse matrix format selection through histogram representation. Pichel *et al.* [29] overcame the class imbalance problem in the context of sparse matrix format selection on GPUs through cost-sensitive methods. Xie *et al.* [31] used CNN to predict the best format and algorithm for SpGEMM through matrix features and density representation.

VI. Conclusion

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 fix-sized matrices through tensor lowering and matrix representation. Besides, we re-design the CNN network by adding the feature layer to compensate for the sparsity features lost during matrix representation. 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.

Acknowledgements

This work is supported by National Key Research and Development Program of China (Grant No. 2016YFB1000304 and 2016YFB0200100), National Natural Science Foundation of China (Grant No. 61502019 and 61732002) and the Open Project Program of the State Key Laboratory of Mathematical Engineering and Advanced Computing (Grant No. 2019A12). Hailong Yang is the corresponding author.

References

[1] B. N. Khoromskij, "Tensors-structured numerical methods in scientific computing: Survey on recent advances," *Chemometrics and Intelligent Laboratory Systems*, vol. 110, no. 1, pp. 1–19, 2012.

[2] N. Vervliet, O. Debals, L. Sorber, and L. De Lathauwer, "Breaking the curse of dimensionality using decompositions of incomplete tensors: Tensor-based scientific computing in big data analysis," *IEEE Signal Processing Magazine*, vol. 31, no. 5, pp. 71–79, 2014.

[3] A. Cichocki, D. Mandic, L. De Lathauwer, G. Zhou, Q. Zhao, C. Caiafa, and H. A. Phan, "Tensor decompositions for signal processing applications: From two-way to multiway component analysis," *IEEE signal processing magazine*, vol. 32, no. 2, pp. 145–163, 2015.

[4] T. Zhang and G. H. Golub, "Rank-one approximation to high order tensors," *SIAM Journal on Matrix Analysis and Applications*, vol. 23, no. 2, pp. 534–550, 2001.

[5] T. Gerstner and M. Griebel, "Dimension–adaptive tensor–product quadrature," *Computing*, vol. 71, no. 1, pp. 65–87, 2003.

[6] F. Le Gall, "Powers of tensors and fast matrix multiplication," in *Proceedings of the 39th international symposium on symbolic and algebraic computation*, 2014, pp. 296–303.

[7] R. A. Harshman *et al.*, "Foundations of the parafac procedure: Models and conditions for an" explanatory" multimodal factor analysis," 1970.

[8] A. Smilde, R. Bro, and P. Geladi, *Multi-way analysis: applications in the chemical sciences*. John Wiley & Sons, 2005.

[9] F. Cong, Q.-H. Lin, L.-D. Kuang, X.-F. Gong, P. Astikainen, and T. Ristaniemi, "Tensor decomposition of eeg signals: a brief review," *Journal of neuroscience methods*, vol. 248, pp. 59–69, 2015.

[10] P. Symeonidis, "Matrix and tensor decomposition in recommender systems," in *Proceedings of the 10th ACM Conference on Recommender Systems*, 2016, pp. 429–430.

[11] N. D. Sidiropoulos, L. De Lathauwer, X. Fu, K. Huang, E. E. Papalexakis, and C. Faloutsos, "Tensor decomposition for signal processing and machine learning," *IEEE Transactions on Signal Processing*, vol. 65, no. 13, pp. 3551–3582, 2017.

[12] F. L. Hitchcock, "The expression of a tensor or a polyadic as a sum of products," *Journal of Mathematics and Physics*, vol. 6, no. 1-4, pp. 164–189, 1927.

[13] J. Choi, X. Liu, S. Smith, and T. Simon, "Blocking optimization techniques for sparse tensor computation," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2018, pp. 568–577.

[14] U. Kang, E. Papalexakis, A. Harpale, and C. Faloutsos, "Gigatensor: scaling tensor analysis up by 100 times-algorithms and discoveries," in *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2012, pp. 316–324.

[15] D. Cheng, R. Peng, Y. Liu, and I. Perros, "Spals: Fast alternating least squares via implicit leverage scores sampling," in *Advances in neural information processing systems*, 2016, pp. 721–729.

[16] N. Srivastava, H. Jin, S. Smith, H. Rong, D. Albonesi, and Z. Zhang, "Tensaurus: A versatile accelerator for mixed sparse-dense tensor computations."

[17] O. Kaya and B. Uçar, "Scalable sparse tensor decompositions in distributed memory systems," in *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2015, pp. 1–11.

[18] S. Smith and 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.

[19] E. F. D'Azevedo, M. R. Fahey, and R. T. Mills, "Vectorized sparse matrix multiply for compressed row storage format," in *International Conference on Computational Science*. Springer, 2005, pp. 99–106.

[20] B. Liu, C. Wen, A. D. Sarwate, and M. M. Dehnavi, "A unified optimization approach for sparse tensor operations on gpus," in *2017 IEEE international conference on cluster computing (CLUSTER)*. IEEE, 2017, pp. 47–57.

[21] J. Li, J. Sun, and 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.

[22] I. Nisa, J. Li, A. Sukumaran-Rajam, R. Vuduc, and P. Sadayappan, "Load-balanced sparse mttkrp on gpus," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2019, pp. 123–133.

[23] Q. Li, W. Cai, X. Wang, Y. Zhou, D. D. Feng, and M. Chen, "Medical image classification with convolutional neural network," in *2014 13th international conference on control automation robotics & vision (ICARCV)*. IEEE, 2014, pp. 844–848.

[24] Y. Wei, W. Xia, M. Lin, J. Huang, B. Ni, J. Dong, Y. Zhao, and S. Yan, "Hcp: A flexible cnn framework for multi-label image classification," *IEEE transactions on pattern analysis and machine intelligence*, vol. 38, no. 9, pp. 1901–1907, 2015.

[25] J. Wang, Y. Yang, J. Mao, Z. Huang, C. Huang, and W. Xu, "Cnn-rnn: A unified framework for multi-label image classification," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 2285–2294.

[26] S. Yu, S. Jia, and C. Xu, "Convolutional neural networks for hyperspectral image classification," *Neurocomputing*, vol. 219, pp. 88–98, 2017.

[27] M. Zhang, W. Li, and Q. Du, "Diverse region-based cnn for hyperspectral image classification," *IEEE Transactions on Image Processing*, vol. 27, no. 6, pp. 2623–2634, 2018.

[28] Y. Zhao, J. Li, C. Liao, and 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, pp. 94–108.

[29] 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.

[30] 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," *The Journal of Supercomputing*, pp. 1–18, 2020.

[31] Z. Xie, G. Tan, W. Liu, and N. Sun, "Ia-spgemm: an input-aware autotuning framework for parallel sparse matrix-matrix multiplication," in *Proceedings of the ACM International Conference on Supercomputing*, 2019, pp. 94–105.

[32] K. Jnawali, M. R. Arbabshirani, N. Rao, and A. A. Patel, "Deep 3d convolution neural network for ct brain hemorrhage classification," in *Medical Imaging 2018: Computer-Aided Diagnosis*, vol. 10575. International Society for Optics and Photonics, 2018, p. 105751C.

[33] J. Li, S. Zhang, and T. Huang, "Multi-scale 3d convolution network for video based person re-identification," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, 2019, pp. 8618–8625.

[34] S. Dong, Z. Gao, S. Pirbhulal, G.-B. Bian, H. Zhang, W. Wu, and S. Li, "Iot-based 3d convolution for video salient object detection," *Neural computing and applications*, vol. 32, no. 3, pp. 735–746, 2020.

[35] J. Bouvrie, "Notes on convolutional neural networks," 2006.

[36] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-

scale machine learning," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 265–283.
[37] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems*, 2019, pp. 8024–8035.
[38] T. G. Kolda and B. W. Bader, "Tensor decompositions and applications," *SIAM review*, vol. 51, no. 3, pp. 455–500, 2009.
[39] C. L. Nikias, "Higher-order spectral analysis," in *Proceedings of the 15th Annual International Conference of the IEEE Engineering in Medicine and Biology Societ*. IEEE, 1993, pp. 319–319.
[40] G. Wetzstein, D. Lanman, M. Hirsch, and R. Raskar, "Tensor displays: compressive light field synthesis using multilayer displays with directional backlighting," 2012.
[41] J. Håstad, "Tensor rank is np-complete," *Journal of algorithms (Print)*, vol. 11, no. 4, pp. 644–654, 1990.
[42] S. Sengupta, M. Harris, M. Garland *et al.*, "Efficient parallel scan algorithms for gpus," *NVIDIA, Santa Clara, CA, Tech. Rep. NVR-2008-003*, vol. 1, no. 1, pp. 1–17, 2008.
[43] S. Smith, J. W. Choi, J. Li, R. Vuduc, J. Park, X. Liu, and G. Karypis, "Frostt: The formidable repository of open sparse tensors and tools," 2017.
[44] I. Jeon, E. E. Papalexakis, U. Kang, and C. Faloutsos, "Haten2: Billion-scale tensor decompositions," in *2015 IEEE 31st International Conference on Data Engineering*. IEEE, 2015, pp. 1047–1058.
[45] S. Hershey, S. Chaudhuri, D. P. Ellis, J. F. Gemmeke, A. Jansen, R. C. Moore, M. Plakal, D. Platt, R. A. Saurous, B. Seybold *et al.*, "Cnn architectures for large-scale audio classification," in *2017 ieee international conference on acoustics, speech and signal processing (icassp)*. IEEE, 2017, pp. 131–135.
[46] J. H. Friedman, "Greedy function approximation: a gradient boosting machine," *Annals of statistics*, pp. 1189–1232, 2001.
[47] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, 2016, pp. 785–794.
[48] A. S. Foulkes, "Classifcation and regression trees," in *Applied Statistical Genetics with R*. Springer, 2009, pp. 157–179.
[49] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, pp. 1–25, 2011.
[50] B.-Y. Su and K. Keutzer, "clspmv: A cross-platform opencl spmv framework on gpus," in *Proceedings of the 26th ACM international conference on Supercomputing*, 2012, pp. 353–364.
[51] D. Merrill and M. Garland, "Merge-based sparse matrix-vector multiplication (spmv) using the csr storage format," *ACM SIGPLAN Notices*, vol. 51, no. 8, pp. 1–2, 2016.
[52] B. Xie, J. Zhan, X. Liu, W. Gao, Z. Jia, X. He, and L. Zhang, "Cvr: Efficient vectorization of spmv on x86 processors," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, 2018, pp. 149–162.
[53] C. Liu, B. Xie, X. Liu, W. Xue, H. Yang, and X. Liu, "Towards efficient spmv on sunway manycore architectures," in *Proceedings of the 2018 International Conference on Supercomputing*. ACM, 2018, pp. 363–373.
[54] A. Benatia, W. Ji, Y. Wang, and F. Shi, "Sparse matrix partitioning for optimizing spmv on cpu-gpu heterogeneous platforms," *The International Journal of High Performance Computing Applications*, vol. 34, no. 1, pp. 66–80, 2020.
[55] N. Sedaghati, T. Mu, L.-N. Pouchet, S. Parthasarathy, and P. Sadayappan, "Automatic selection of sparse matrix representation on gpus," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, 2015, pp. 99–108.
[56] A. Benatia, W. Ji, Y. Wang, and F. Shi, "Sparse matrix format selection with multiclass svm for spmv on gpu," in *2016 45th International Conference on Parallel Processing (ICPP)*. IEEE, 2016, pp. 496–505.
[57] Y. Zhao, W. Zhou, X. Shen, and G. Yiu, "Overhead-conscious format selection for spmv-based applications," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2018, pp. 950–959.
[58] D. Svozil, V. Kvasnicka, and J. Pospichal, "Introduction to multi-layer feed-forward neural networks," *Chemometrics and intelligent laboratory systems*, vol. 39, no. 1, pp. 43–62, 1997.
[59] S. Smith, N. Ravindran, N. D. Sidiropoulos, and G. Karypis, "Splatt: Efficient and parallel sparse tensor-matrix multiplication," in *2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2015, pp. 61–70.
[60] J. Dean and S. Ghemawat, "Mapreduce: a flexible data processing tool," *Communications of the ACM*, vol. 53, no. 1, pp. 72–77, 2010.
[61] J. H. Choi and S. Vishwanathan, "Dfacto: Distributed factorization of tensors," in *Advances in Neural Information Processing Systems*, 2014, pp. 1296–1304.
[62] N. Vervliet and L. De Lathauwer, "A randomized block sampling approach to canonical polyadic decomposition of large-scale tensors," *IEEE Journal of Selected Topics in Signal Processing*, vol. 10, no. 2, pp. 284–295, 2015.
[63] J. Li, G. Tan, M. Chen, and N. Sun, "Smat: an input adaptive auto-tuner for sparse matrix-vector multiplication," in *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, 2013, pp. 117–126.

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

Hardware and Software Platforms. We evaluate SpTFS on two hardware platforms, where the CPU is an Intel Xeon Silver 4110 processor and the GPU is a Nvidia GeForce RTX 2080 Ti processor. TnsNet is built using TensorFlow v1.15. We did not use any MPI library.

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 v1.1.0, whereas the COO and HiCOO implementations are adopted from ParTI! v1.1.0. For MTTKRP on GPU, the tensor formats we evaluate are COO, F-COO, CSF and HB-CSF. Except F-COO, we adopt the implementations of all tensor formats from the paper published in IPDPS19 (Nisa et al.). To ensure fairness, we use the best parameter configurations reported in each implementation and compile with "O3" option.

Comparison. We compare the design of TnsNet with the XGBoost v1.1.0 including GBDT and GBlinear. During training data collection, we set the rank size to 16 for all sparse formats. Besides, we set the number of threads to 16 when on CPU, and the thread block size to 256 when on GPU.

ARTIFACT AVAILABILITY

Software Artifact Availability: All author-created software artifacts are maintained in a public repository under an OSI-approved license.

Hardware Artifact Availability: There are no author-created hardware artifacts.

Data Artifact Availability: There are no author-created data artifacts.

Proprietary Artifacts: None of the associated artifacts, author-created or otherwise, are proprietary.

Author-Created or Modified Artifacts:

```
Persistent ID: https://github.com/sunqingxiao/SpTFS\
↪ _info\_for\_SC20;
↪ http://doi.org/10.5281/zenodo.3866029
Artifact name: SpTFS
```

BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

Relevant hardware details: Intel Xeon Silver 4110 CPU; GeForce RTX 2080 Ti

Operating systems and versions: CentOS Linux release 7.6; GPU Driver 440.33

Compilers and versions: gcc 4.8; nvcc 10.2

Applications and versions: TensorFlow v1.15; XGBoost v1.1.0

Libraries and versions: SPLATT v1.1.0; ParTI! v1.1.0; HB-CSF

Key algorithms: convolutional neural network

Input datasets and versions: SuiteSparse; FROSTT; HaTen2

URL to output from scripts that gathers execution environment information.

```
https://github.com/sunqingxiao/SpTFS\_info\_for\_SC2
↪ 0/execution-environment.txt
```

ARTIFACT EVALUATION

Controls, statistics, or other steps taken to make the measurements and analyses robust to variability and unknowns in the system. We have conducted the safety checks when collecting the training data. Specifically, we repeat the execution of MTTKRP under each tensor format for 10 times, and collect the mean execution time and standard deviation for each tensor format. The above procedure is applied to all input sizes. The details of safety checks and data collection have been provided in the readme file of the following url: https://github.com/sunqingxiao/SpTFS_info_for_SC20/blob/ master/software/README.md.