ScalFrag: Efficient Tiled-MTTKRP with Adaptive Launching on GPUs

Wenqing Lin, Hemeng Wang, Haodong Deng, Qingxiao Sun
SSSLab, Dept. of CST, China University of Petroleum-Beijing, China
{wenqing.lin,hemeng.wang,haodong.deng}@student.cup.edu.cn, qingxiao.sun@cup.edu.cn

***Abstract*—Tensor decomposition, a pivotal technique in mining underlying patterns from voluminous and high-dimensional sparse datasets, plays a crucial role in unraveling latent structures within complex data. Among the various methods employed for tensor decomposition, Canonical Polyadic Decomposition (CPD) stands out as a prominent choice, widely embraced across numerous scientific disciplines and practical applications due to its effectiveness in capturing multi-linear relationships. However, the computational efficacy of CPD is significantly hampered by the Matricized Tensor Times Khatri-Rao Product (MTTKRP) operation, which constitutes its primary bottleneck. While offloading the MTTKRP computation onto Graphics Processing Units (GPUs) has emerged as a prevalent strategy to leverage their parallel processing capabilities for enhancing performance, the inherent sparsity and irregular data access patterns intrinsic to these operations introduce new complexities.**

Addressing this challenge, we introduce an innovative methodology *ScalFrag* designed to accelerate sparse MTTKRP computations on GPU platforms. A key insight underlying our approach is the recognition that the optimal kernel launch configuration—a critical factor influencing GPU performance—varies considerably depending on the unique characteristics of the input tensor. We devise a dynamic kernel launch configuration selection mechanism to tackle this variability. This novel strategy autonomously identifies and applies the most advantageous launch setup tailored to each input tensor, optimizing computational efficiency. Additionally, we present a stream-based algorithm for sparse MTTKRP, further overlapping data access time. By leveraging streaming architectures, our algorithm significantly improves data access efficiency, mitigating the bottlenecks associated with the irregularities of sparse tensor patterns.

The experimental results show that *ScalFrag* performs better than the SOTA library ParTI, and is able to find more suitable kernel launch parameter configurations in a short time.

***Index Terms*—GPU, Tensor Decomposition, MTTKRP, Sparse Tensors, Parallel Computing, Performance Model**

I. Introduction

A tensor, which is a multidimensional array, serves as a higher-level generalization of a matrix, offering a natural way to represent intricate and interconnected data. For example, a 3-way tensor is used in computer vision to represent video data, where each dimension corresponds to different aspects: video frames, frame height, and frame width. The tensors facilitate machine learning operations like convolutions and tasks such as action recognition and object detection. Various essential fields, including data mining [18], recommended systems [29], social network analysis [28], cybersecurity [5], numerical linear algebra, computer vision [37], numerical analysis, and healthcare [12], produce extensive sets of multi-dimensional data in the form of *sparse tensors*. Tensors are also fundamental data structures in deep learning libraries like TensorFlow [1] and PyTorch [26], These libraries provide high-level abstractions and operations for creating, manipulating, and performing computations on tensors, enabling efficient implementation of deep learning models and algorithms. Tensor operations serve as fundamental building blocks and often serve as the decisive operations influencing the performance of tensor algorithms and applications.

Tensors can be efficiently and quickly analyzed using *tensor decomposition* (TD). The utilization of tensor decomposition techniques originated in the field of psychometrics in 1970 [11], with subsequent adoption in chemometrics later [2]. In recent years, these decomposition methods have experienced a surge in popularity due to their effectiveness in various applications such as recommended systems, conversation detection, and so on. The most popular TD method is canonical polyadic decomposition (CPD) model, which is a generalization of singular value decomposition and approximates a tensor as a sum of a finite number of rank-one tensors such that each rank-one tensor corresponds to a useful data property [17].

Executing a complete iteration of CPD involves conducting matricized tensor times Khatri-Rao products (MTTKRP) across every mode and the computation of the CPD for a sparse tensor is predominantly influenced by the MTTKRP operation, accounting for around 90% of the overall execution time [33]. Therefore, it serves as the primary target for optimizations in tensor decomposition. To improve the performance of this memory-constrained workload, recent state-of-the-art research has attempted to create compact representations of the tensor in each mode and has also equipped massively parallel architectures with high bandwidth memory (HBM) (i.e., GPU) to accelerate the MTTKRP kernel.

However, the inherent sparsity of the datasets and the irregularity in data access patterns pose new complexities when attempting to harness GPU power effectively. Sparse data means that only a fraction of the data matrix/tensor contains non-zero elements, leading to inefficient memory usage and complex memory access patterns. Besides, a critical aspect of GPU programming is the kernel launch configuration, which determines how tasks are distributed among GPU threads and blocks. Finding the optimal configuration is vital for maximizing GPU performance. The complexity arises from the fact that the ideal configuration is highly dependent on the specific characteristics of the input tensor, such as its dimensions and sparsity distribution. Traditional

static configurations often fail to adapt to varying inputs, resulting in suboptimal performance. Thus, there's a need for a dynamic mechanism that can automatically identify and apply the best launch setup for each unique input, thereby optimizing computational efficiency.

Another pertinent concern arises from the prevalent focus within the field on optimizations confined within the kernel, neglecting consideration for the comprehensive performance of MTTKRP from start to finish. Through empirical assessments, we have discerned that a substantial portion of the end-to-end MTTKRP process is consumed by data transfer operations, surpassing computational durations in certain tensor instances. Hence, it is our contention that enhancing MTTKRP performance necessitates a dual approach encompassing optimizations within the kernel as well as those addressing end-to-end performance considerations. Furthermore, a comprehensive understanding of the interplay between kernel execution parameters and data transfer dynamics is necessary. This includes identifying the most suitable block and grid sizes that not only expedite computations but also synchronize optimally with the data transfer schedules, thereby minimizing idle times on either the GPU or the CPU.

To address these issues, we propose a new framework *ScalFrag* for sparse tensor decomposition on GPU. First, appropriate parameters are selected based on an auto-tuning launch parameter selection strategy, then, the tensor is divided into segments to process only a portion of the data at a time, and finally, transmission and computation are performed using a form of pipelined parallelism.

Specifically, this paper makes the following key contributions:

- We propose a method for selecting a suitable combination of launch parameters by means of an adaptive launch strategy that selects the optimal combination of launch parameters based on the tensor characteristics.
- We introduce a new blocking approach to generate tensor blocks, based on the resource constraints of hardware, for asynchronous computing to reduce memory usage.
- We introduce a pipelined parallelism strategy, which effectively utilizes the computational resources of the GPU by decomposing the computation process into multiple stages and enabling parallel processing among stages.
- While exposing the fine-grained parallelism within a block to efficiently utilize the GPU hardware, we put the parts with low parallelism to the CPU for execution. Through this CPU-GPU heterogeneous hybrid optimization, substantial efficiency improvement is achieved.

The rest of this paper is organized as follows: Section II presents the background of this paper, describes the related tensor notations, and gives an overview of tensor decomposition. Section III presents the details of the motivation. Section IV describes the methodology of the paper. Section V presents the evaluation results. Section VI discusses the related work, and Section VII concludes this paper.

II. Background

We begin by summarizing the basic tensor notations used in the paper, and then briefly describe the basics of the CPD and MTTKRP computation. A more detailed description of tensor decomposition methods developed over the last few decades, along with their applications, can be found in the survey [17].

A. Tensor Notation

Tensors are multi-modal arrays that extend the concepts of vectors and matrices. An N-order tensor is an array with N modes. In this paper, we focus on a three-dimensional tensor and mode-1 operation to explain the concepts and associated mathematics of tensor decomposition. 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, We use the following notation in this paper:

- *Scalars* are written with lowercase letters (e.g., a).
- *Vectors* (first-order tensors) are written with bold lowercase letters (e.g., $a \in \mathbb{R}^I$). The i^th entry of a$\in \mathbb{R}^I$ is denoted a_i.
- *Matrices* (second-order tensors) are written with bold capital letters (e.g., $\mathbf{A} \in \mathbb{R}^{I \times J}$). The $(i,j)^th$ entry of $\mathbf{A} \in \mathbb{R}^{I \times J}$ is denoted $a_{i,j}$.
- *Higher-order tensors* are written with Euler script letters (e.g., $\mathcal{X} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$). The $(i_1, \ldots, i_N)^{th}$ entry of the N-order tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ is denoted $x_{i_1,\ldots,i_N}$.
- *Fibers* are the analog of matrix rows/columns for higher-order tensors. A mode-n fiber of a tensor $\mathcal{X}$ is any vector formed by fixing all indices of $\mathcal{X}$, except the n^th index (e.g., a matrix column is defined by fixing the second index, and is, therefore, a mode-1 fiber).
- *Hadamard* product is an element-wise product between two vectors or matrices, and is denoted by the symbol "*".
- *Kronecker* product between two matrices $\mathbf{A} \in \mathbb{R}^{I \times J}$ and $\mathbf{B} \in \mathbb{R}^{K \times L}$ produces the matrix $\mathbf{C} \in \mathbb{R}^{IJ \times KL}$ where

$$\mathbf{C} = \begin{bmatrix} \mathbf{a}_{1,1}\mathbf{B} & \mathbf{a}_{1,2}\mathbf{B} & \cdots & \mathbf{a}_{1,J}\mathbf{B} \\ \mathbf{a}_{2,1}\mathbf{B} & \mathbf{a}_{2,2}\mathbf{B} & \cdots & \mathbf{a}_{2,J}\mathbf{B} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{a}_{I,1}\mathbf{B} & \mathbf{a}_{I,2}\mathbf{B} & \cdots & \mathbf{a}_{I,J}\mathbf{B} \end{bmatrix}$$

  and is denoted $A \otimes B$.

B. Canonical Polyadic Decomposition

Canonical polyadic decomposition (CPD) is the higher-level generalization of singular value decomposition (SVD), a popular matrix decomposition technique. It 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, and the CPD of a third-order tensor

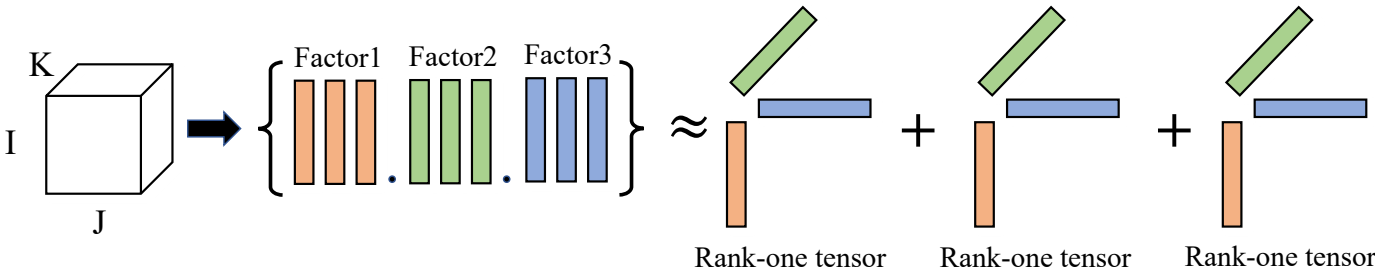

Fig. 1. Canonical polyadic decomposition of a third-order tensor.

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.
$\mathbf{a}$	A vector.
$\mathbf{a}_i$	An element in a vector.
$\otimes$	The symbol for Kronecker product.
$\odot$	The symbol for Khatri-Rao product.
$*$	The symbol for Hadamard product.
$\dagger$	The symbol for pseudo-inverse.

is shown in Figure 1. In other words, the CPD models a tensor $\mathcal{X} \in \mathbb{R}^{I\times J\times K}$ with three factor matrices $A \in \mathbb{R}^{I\times F}$, $B \in \mathbb{R}^{J\times F}$ and $C \in \mathbb{R}^{K\times F}$ as formulated in Equation (1).

To solve the CPD, one of the most popular approaches is to utilize alternating least squares (ALS), 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 symbol $\mathcal{X}_{(1)}(C \odot B)$ is the MTTKRP operation; it contains the Khatri-Rao product of B and C. $X_{(1)}$ is the mode-1 matricization of $\mathcal{X}$. The output of the Khatri-Rao product is then multiplied with $(B^TB * C^TC)^\dagger$, which is the pseudo-inverse of the $R \times R$ matrix generated by B^TB and C^TC. Algorithm 1 demonstrates the steps to update each matrix using the ALS algorithm. Line 3, Line 4, and Line 5 show the MTTKRP operations to update A, B, and C respectively. The main bottleneck in the CPD-ALS algorithm for sparse tensors is MTTKRP, which can be formulated as Equation (3). The reason can be attributed to the access to the large sparse tensor $\mathcal{X}$, and the scattered access to the factor matrices directed from $\mathcal{X}$.

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

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

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

Algorithm 1 The CPD-ALS algorithm

Require: An N-order tensor $\mathcal{X} \in \mathbb{R}^{I_1\times\cdots\times I_N}$, randomly initialized dense factor matrices $\mathbf{A}^{(1)} \in \mathbb{R}^{I_1\times R}$, $\mathbf{A}^{(2)} \in \mathbb{R}^{I_2\times R}$, $\cdots$, $\mathbf{A}^{(N)} \in \mathbb{R}^{I_1\times R}$.
Ensure: Updated factor matrices that approximate $\mathcal{X}$.
1: **repeat**
2: **while** $n = 1, ..., N$ **do**
3: $V \leftarrow \mathbf{A}^{(1)T}\mathbf{A}^{(1)} * \cdots * \mathbf{A}^{(n-1)T}\mathbf{A}^{(n-1)} * \mathbf{A}^{(n+1)T}\mathbf{A}^{(n+1)} * \cdots * \mathbf{A}^{(N)T}\mathbf{A}^{(N)}$
4: $M \leftarrow \mathbf{X}_{(n)}(\mathbf{A}^{(N)} \odot \cdots \odot \mathbf{A}^{(n+1)} \odot \mathbf{A}^{(n-1)} \odot \cdots \odot \mathbf{A}^{(1)})$
5: $\mathbf{A}^{(n)} \leftarrow MV^\dagger$ $\dagger$ denotes the pseudo-inverse
6: **end while**
7: **until** converged
8: **return** $\mathbf{A}^{(1)}, \cdots, \mathbf{A}^{(N)}$

C. Sparse MTTKRP

MTTKRP kernel involves two basic operations:

- *Tensor matricization* is the process in which a tensor is unfolded into a matrix. The mode-n matricization of a tensor $\mathcal{X}$, denoted by $X_{(n)}$, is obtained by laying out the mode-n fibers of $\mathcal{X}$ as the columns of $X_{(n)}$.
- The *Khatri-Rao product* is the "matching column-wise" Kronecker product of two matrices. Given $A \in \mathbb{R}^{I\times F}$ and $B \in \mathbb{R}^{J\times F}$, their Khatri-Rao product is $\mathbf{K} = \mathbf{A} \odot \mathbf{B}$, where $\mathbf{A} \odot \mathbf{B} = [\mathbf{a}_1 \otimes \mathbf{b}_1 \mathbf{a}_2 \otimes \mathbf{b}_2 \ldots \mathbf{a}_R \otimes \mathbf{b}_R] \in \mathbb{R}^{I\times J\times R}$.

For an N-order tensor $\mathcal{X}$ and factor matrices $\mathbf{A}^{(1)}, \mathbf{A}^{(2)}, \cdots, \mathbf{A}^{(N)}$, the mode-$n$ MTTKRP is given by Equation (4).

$$\mathbf{M} = \mathbf{X}_{(n)}(\mathbf{A}^{(N)} \odot \cdots \odot \mathbf{A}^{(n+1)} \odot \mathbf{A}^{(n-1)} \odot \cdots \odot \mathbf{A}^{(1)}) \tag{4}$$

D. Sparse Tensor Formats

Sparse tensor formats are broadly categorized into two main families: coordinate-based and tree-based, as shown in the Figure 2. An elementary method for sparse tensor representation involves storing the indices along each dimension and the value for each non-zero element, known as the Coordinate (COO) format.

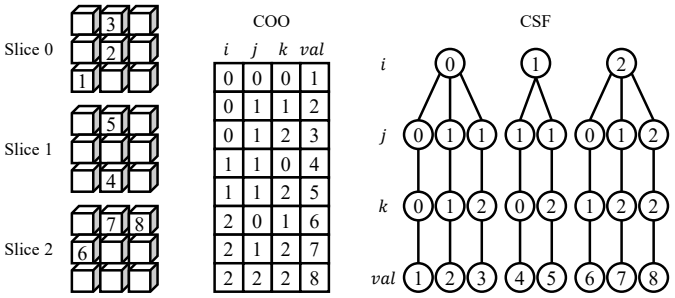

Fig. 2. COO and CSF (mode 1) formats for an example third-order tensor.

For a third-order tensor $\mathcal{X}$, the COO format consists of $M(i, j, k, val)$ entries, where each entry represents the indices (i, j, k) and the corresponding value of a non-zero element. The COO format's simplicity and suitability for parallelization over non-zero elements make it a popular choice. Nonetheless, a drawback of the parallel COO format is its requirement for atomic operations to update the output matrix. The state-of-the-art tensor formats belonging to this family include F-COO [22], which adds flag arrays to eliminate atomic operations. HiCOO [21], which decomposes a sparse tensor into small sparse blocks, reducing the memory required to store tensor nonzeros (and hence memory bandwidth conflicts), and TB-COO [8], which leverages warp shuffle and shared memory on GPU to enable efficient reduction.

The tree-based format family compresses sparse tensor indices into a tree structure. Depending on the input tensor's structure, CSF exhibits the potential to decrease storage demands and required operation counts. Some notable members of this family are CSF, introduced by Smith et al. [31], and BCSF, proposed by Nisa et al. [24], which mainly optimize the load imbalance issue of CSF format and MM-CSF [23] which provides a mixed-mode storage format for sparse tensors of arbitrary dimensions. These formats extend the compressed sparse row (CSR) matrix format to higher-order tensors.

E. CPU–GPU Heterogeneous Computing

Originally, GPU was primarily tailored for gaming purposes, rendering 2D images of 3D triangles and other geometric objects. Each pixel in the output image corresponds to an element, and the GPU employs a cluster of processors to compute the color of these pixels simultaneously. However, modern GPU has evolved into more versatile tools, where rendering is just one of their applications. Their impressive theoretical peak performance makes them appealing for high-performance computing tasks. Given the widespread utilization of both CPUs and GPUs across various applications,

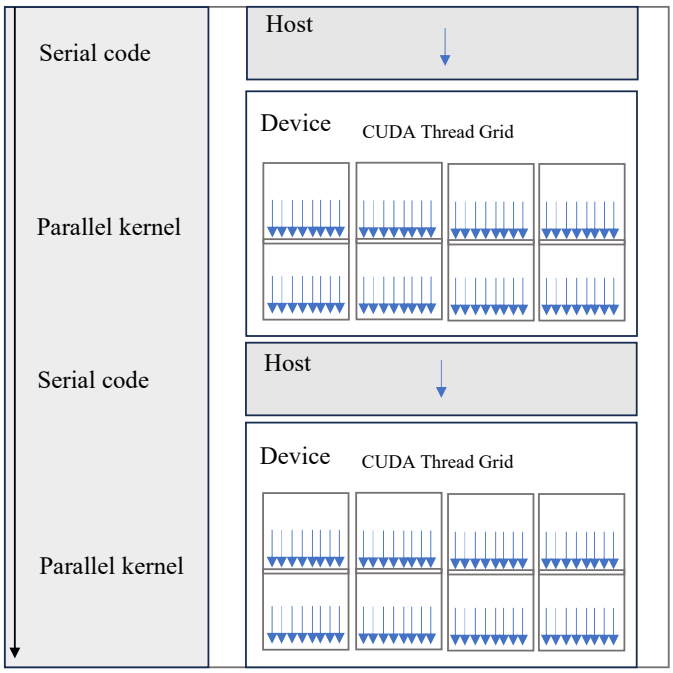

Fig. 3. CPU–GPU heterogeneous computing pattern.

it's acknowledged that each of these processing units (PUs) possesses unique features and strengths.

Modern multicore CPUs typically are equipped with a few tens of cores, which are usually out-of-order, multi-instruction issue cores. Additionally, CPU cores operate at high frequencies and utilize large-sized caches to minimize the latency of a single thread, making them well-suited for latency-critical applications. In contrast, GPUs leverage a significantly larger number of cores, which are typically in order and share their control units. Furthermore, GPU cores operate at lower frequencies and use smaller-sized caches, making them more suitable for throughput-critical applications.

In view of these differences, collaboration between CPU and GPU is considered to be the key to realizing high-performance computing, and Figure 3 illustrates a heterogeneous computing model for CPUs and GPUs. This allows the advantages of both types of processors to be fully utilized to build heterogeneous computing environments, thereby effectively compensating for the respective shortcomings of CPUs and GPUs and optimizing the performance of various applications.

III. Motivation

A. Parameter Sensitivity Analysis

Our MTTKRP computations performed using the NVIDIA GeForce RTX 3090 platform revealed that different combinations of `gridSize` and `blockSize` have a significant impact on the performance of MTTKRP computations. Where `gridSize` and `blockSize` are execution parameters of the MTTKRP kernel on the GPU, `gridSize` denotes the number of threads in the grid, while `blockSize` refers to the number

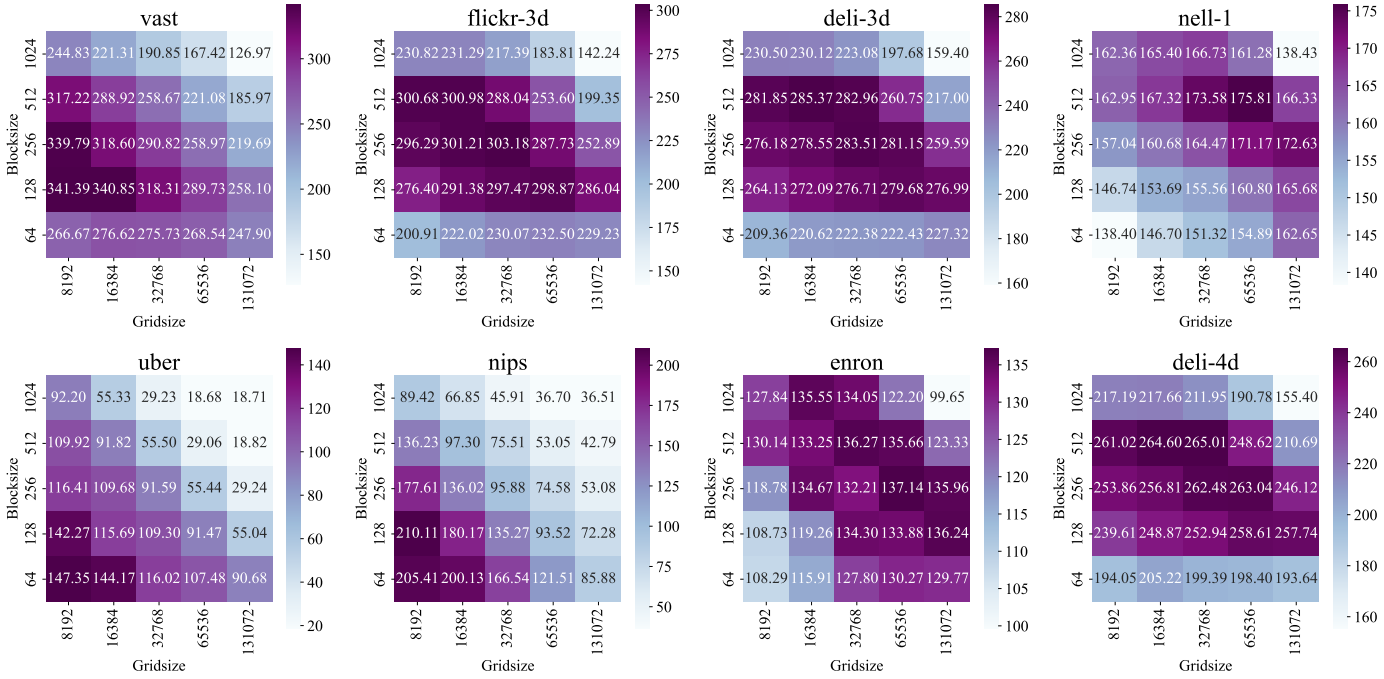

Fig. 4. GFlops of MTTKRP kernel with different launch settings.

of threads per thread block (TB). The results are illustrated in Figure 4.

It can be seen that the color distribution of heatmaps is not the same for different tensors. In general, when the `gridSize` and `blockSize` are small, the performance is poor. With the increase of parameter values, the performance will gradually improve. However, when the `gridSize` and `blockSize` reach a certain value, the performance decreases. This indicates that the optimal values of `gridSize` and `blockSize` are not the larger the better, but need to be determined according to the characteristics of the tensor and the actual situation of the computing environment. The performance of the same tensor under different parameters has a big gap, and the optimal performance parameters of different tensors are significant different. These parameters determine the allocation of computing resources, load balancing of tasks, and memory access efficiency, which have a large impact on MTTKRP computational performance.

Due to the diversity and complexity of tensor data, as well as the variability of computing environments, it is almost impossible to find a common and optimal set of parameter configurations for MTTKRP computation. When we deal with large-scale and complex tensor data, different tensors may have different dimensionality, sparsity, data distribution, etc., and the hardware environments may also have significant differences in terms of computing capability and memory bandwidth, which make it impossible to simply apply a fixed set of parameter configurations to fit all cases. Therefore, a method for searching the high-performance launch parameters is needed, which can dynamically adjust the parameter configurations according to the actual situation of the tensor data and the hardware environment, and achieve the performance optimization of MTTKRP parallel computing.

B. Waste of Computational Resources

During the execution of MTTKRP for tensors, data transfer is a critical aspect, especially between the host and the device. The host and the device are connected via PCIe with 24.3 GB/s bandwidth. As shown in Figure 5, transferring data from the host to the device (H2D) takes a lot of time. Because tensor data is typically large, H2D takes up the vast majority of the time. In contrast, kernel computation takes less time and transferring data from the device to the host (D2H) takes less time.

However, the device cannot start computation until the data is fully transferred and must wait for the data loading to complete. The idle waiting time indicate that the computational resources cannot be fully utilized. Moreover, The waiting time increases as the amount of data increases, leading to a decrease in computational performance. Therefore, the data transfer time becomes a bottleneck in MTTKRP computation. In addition, the bandwidth for data transfer between host and device is limited. When the amount of data transferred exceeds the bandwidth capacity, the speed of data transfer is limited, which leads to an increase in the waiting time of the device and a decrease in the computational performance.

In this case, we need to find an effective way to reduce the waiting time of the device so that MTTKRP computation can fully utilize the hardware resources. To this end, we propose a new solution: partitioning tensor data and transferring the data while computing. This pipelined transmission method can greatly reduce the waiting time of the device and realize the parallel execution of data transmission and computation.

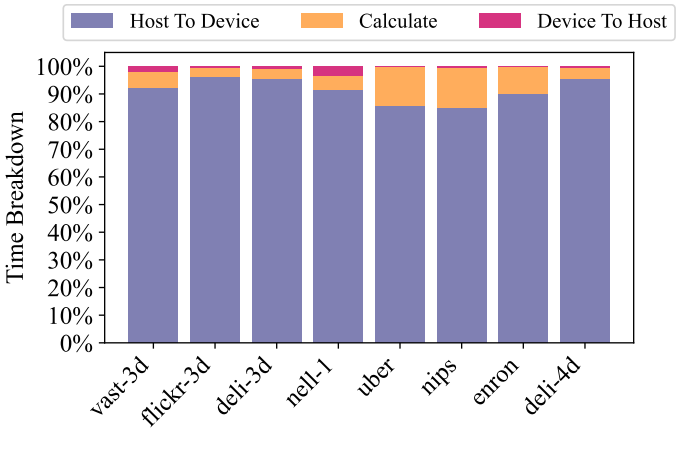

Fig. 5. Time breakdown of MTTKRP processing.

IV. Methodology

A. Design Overview

In this section, we detail an efficient end-to-end MTTKRP acceleration framework *ScalFrag* for GPU platforms. *ScalFrag* aims to fully utilize the parallel processing capability of GPU to achieve efficient acceleration of MTTKRP computation. The *ScalFrag* framework consists of two parts: adaptive launching strategy, and pipelined parallel processing. The design overview of *ScalFrag* is shown in Figure 6.

ScalFrag first matricizes the input tensor and then chunks the matricized tensor. The chunked matrix is pipelined to the device side. Before the kernel function starts, *ScalFrag* uses a trained model to select the best combination of launch parameters based on the sparsity features of the matrixed tensor after chunking. The core of the adaptive launching strategy is to dynamically select the optimal parameter combinations to adapt to the different characteristics of the matricized tensor. *ScalFrag* utilizes the parallel processing capability of GPU to automatically adjust the launch parameters by analyzing the dimensions and distribution characteristics of the tensor to maximize the computational throughput. During kernel function computation, the frequently accessed data in the kernel and intermediate results (e.g., computation result `mvals`, factor matrices `times_mat`) are stored in shared memory to reduce the latency of data accesses. After kernel computation, *ScalFrag* passes the results back to the host side.

B. Adaptive Launching Strategy

Typically, MTTKRP has unique optimal launch parameter configuration for each sparse tensor. In order to fully utilize the computational GPU resources, we propose an adaptive launching strategy to select the optimal parameter combinations based on the sparsity distribution of the tensor. The adaptive launching strategy is illustrated in Figure 7.

At first, we need to preprocess the input tensor. The key feature parameters of the tensor are extracted, which reflect the basic properties and computational requirements of the tensor. The feature parameters we focus on mainly include tensor size (dimension and number of elements) and sparsity (distribution and proportion of non-zero elements). For example, the feature parameters include `numSlices`, `numFibers`, `sliceRatio`, `fiberRatio`, `maxNnzPerSlice`, `maxNnzPerSlice`, and so on. these parameters not only determine the size of the computation, but also affect the efficiency of the GPU parallel computation and the overhead of data transfer.

In the training phase, we use the extracted key feature parameters to train a parameter selection model whose output is the best combination of starting parameters for input tensor features. To obtain the best prediction performance, we try various machine learning models such as DecisionTree, SVM, AdaBoost, Bagging, etc., each of which has its own unique advantages. After that, we evaluate the trained model in terms of prediction accuracy, training and inference time. For example, the DecisionTree regressor has the lowest MAPE (less than 15%), which can be a good guide for the selection of launch parameters. For these lightweight models, the training time is less than 0.5 seconds. Since the training needs to be performed only once, the cost can be considered negligible. Even for online overhead, the inference time is less than 1% of the MTTKRP computation. On the other hand, the iterative CPD process involves many MTTKRP operations, further diluting the inference overhead.

In the practical application stage, we first extract the characteristic parameters of the input tensor. These parameters include key information such as tensor size and sparsity. Then, we input these parameters into the trained launch parameter selection model. The model will output an optimal launch parameter combination based on the input feature parameters. Finally, the optimal launch parameter combination is passed to the GPU to initiate the MTTKRP computation. Since we select the optimal launch parameter combination based on the feature parameters of the tensor, the GPU is able to process the tensor data more efficiently, achieving higher computational throughput and hardware resource utilization.

C. Pipeline Parallelism

In order to reduce the waiting time incurred by transferring data between a host and a device (e.g., a GPU), we have designed and implemented a pipeline parallelization method to improve the effectiveness of MTTKRP. Figure 8 visualizes the whole flow of the segmented pipeline parallelism.

In the data preprocessing stage, we segment the COO format tensor based on the pre-designed index and the number of segments with non-zero element values. This segmentation method is able to decompose the originally huge tensor data into several small segments, each of which contains a portion of non-zero elements and their corresponding coordinate information. In this way, we are able to manage and transfer these data chunks more efficiently, thus reducing the overall time of data transfer.

Next, we reasonably allocate storage space to accommodate the entire decomposed tensor data according to the performance and storage capacity of the GPU. At the same

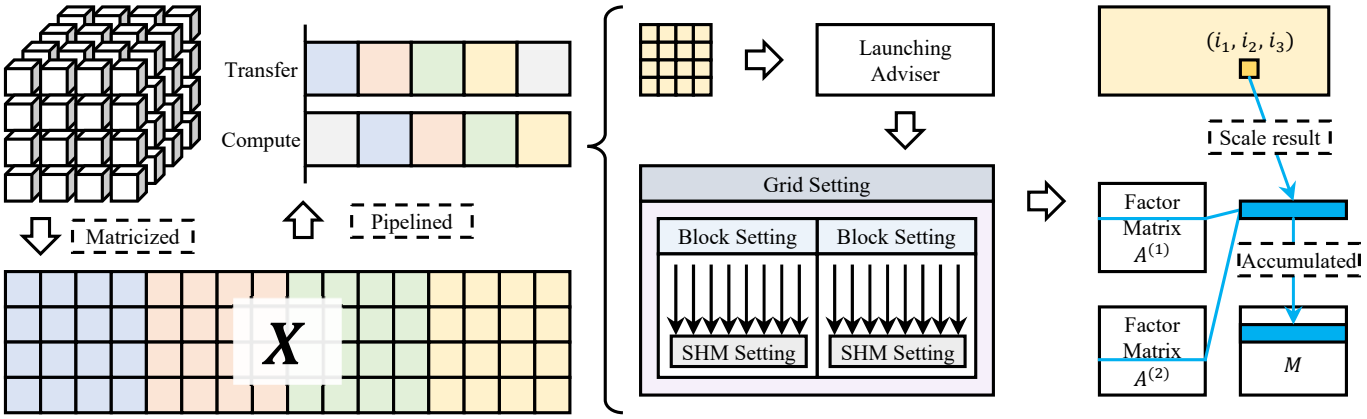

Fig. 6. The design overview of *ScalFrag*.

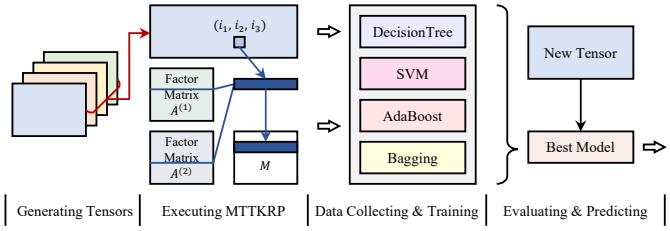

Fig. 7. The adaptive launching parameter selection.

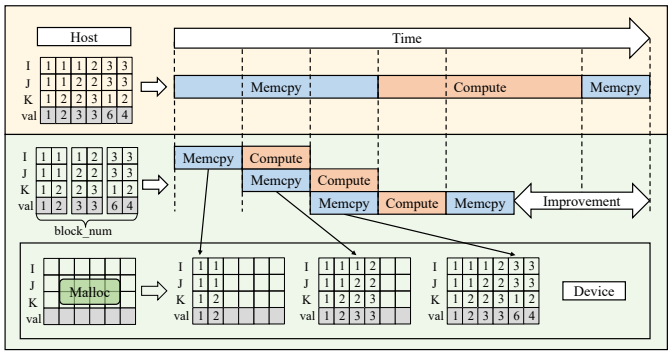

Fig. 8. Pipeline parallelism for MTTKRP computation.

time, we create a corresponding number of CUDA streams based on the number of segments. Each stream is responsible for transferring and processing one or more specific data segments, thus realizing parallel data transfer and computation. This parallel processing method can fully utilize the GPU resources and reduce the end-to-end execution time.

In the data transfer phase, we assign the transfer operation for each data segment to the corresponding CUDA stream for asynchronous transfer. This means that when data is transferred as data segments to the GPU, the GPU can start processing the previously completed data segment without having to wait for the transfer of the current data segment to complete. This asynchronous transfer increase the temporal utilization of hardware resources caused by the data transfer, further improving overall performance.

When segmented transmission is performed in practical applications, the data transmission time may be greater than the calculation time. At this time, after the data calculation of the previous segment is completed, the next segment is still being transmitted. This would indicate that the communication is not fully covered. Therefore, we empirically determine the appropriate number of segments and streams as much as possible to minimize the device waiting time.

When the computation of the data segment is completed, we synchronize the results from the device memory back to the host memory. On the host, we can perform further data processing and storing of the results. This synchronized return ensures data accuracy and consistency, and provides reliable data support for the tensor application.

V. Evaluation

A. Experiment Setup

1) Platform: As depicted in Table II, we have conducted experiments on the platform equipped with Intel Core i7 CPU and NVIDIA RTX 3090 GPU. We adopt GCC-9.4.0 and NVCC-12.2 compilers within an Ubuntu 20.04.6 environment.

TABLE II
Hardware Specifications.

	CPU	GPU
Model	Intel Core i7-11700K	NVIDIA GeForce RTX 3090
Frequency	3.6GHz	1.4GHz
Processing Units	8C16T	10496 (82 SMs)
Cache	80KB L1, 512KB L2, 16MB L3	128KB L1 (per SM), 6MB L2
Memory	32GB	24GB
Bandwidth	31.2 GB/s	936.2 GB/s

2) Dataset: We use 3D and 4D sparse tensors collected from real-world applications, all of them from The formidable repository of open sparse tensors and tools (FROSTT) [30].

Table III lists the order, dimension, and number of nonzeroes of the tensors as well as the densities of these tensors.

3) Implementations: We compare *ScalFrag* with publicly available framework ParTI [19], which is designed for fast sparse tensor operations and tensor decompositions on multicore CPU and GPU architecture. ParTI supports a variety of tensor operations, including arithmetic operations, SpTTM, SpMTTKRP, SpCPD, sparse Tucker decomposition, and so on. We use the optimal parameter configuration suggested by the authors and thoroughly resize the grid and thread blocks.

TABLE III
TENSORS USED FOR EVALUATION.

Tensor	Order	Dimensions	#nnz	Density
vast	3	165K × 11K × 2	26M	6.9 × 10-3
nell-2	3	12K × 9K × 29K	77M	2.4 × 10-5
flickr-3d	3	320K × 28M × 2M	113M	7.8 × 10-12
deli-3d	3	533K × 17M × 3M	140M	6.1 × 10-12
nell-1	3	2.9M × 2.1M × 25M	144M	9.1 × 10-13
uber	4	183 × 24 × 1140 × 1717	3M	3.9 × 10-4
nips	4	2K × 3K × 14K × 17	3M	1.8 × 10-6
enron	4	6K × 6K × 244K × 1K	54M	5.5 × 10-9
flickr-4d	4	320K × 28M × 2M × 731	113M	1.1 × 10-14
deli-4d	4	533K × 17M × 3M × 1K	140M	4.3 × 10-15

B. Kernel Performance Comparison

We evaluated the performance of *ScalFrag* processing sparse tensor for MTTKRP computation and compared it with ParTI. The results are shown in Figure 9.

From the figure, it is obvious that *ScalFrag* outperforms ParTI for the tested sparse tensor. This phenomenon indicates that different tensors require significantly different hardware resources during computation due to their unique dimensionality, density, and distribution characteristics. *ScalFrag* enables different sparse tensors to achieve their respective better performance and, by means of shared memory that reduces the time of data access during MTTKRP computation.

The results indicate that the performance acceleration of *ScalFrag* is more pronounced for smaller tensors (e.g., *vast*, *uber*, *nips*, etc.) or relatively smaller tensors (e.g. *nell-1*). This is due to the fact that they are significantly different in terms of the number of non-zero elements alone. In particular, for the smaller 4d tensor *nips* in the dataset, the performance is significantly improved with *ScalFrag*, which is nearly 2.2× faster than ParTI; for the smallest 3d tensor *vast* in the dataset, a speedup ratio of about 1.2× is also obtained.

There are many tensors with different characteristics. For example, in the field of image recognition, the tensor is composed of images with different resolutions and color depth. We need to optimize the kernel parameters according to the tensor features to improve the processing speed and accuracy. In the financial data analysis, the tensor is formed by different time periods and different trading varieties. We also need to adjust the kernel parameters specifically in order to better mine the valuable information. This not only illustrates the importance of parameter selection for different tensor features, but also demonstrates the effectiveness of the launch parameter auto-tuning strategy of *ScalFrag*.

C. End-to-end Performance Comparison

We compare *ScalFrag* with ParTI on tensor datasets of different sizes and sparsity patterns. As can be seen from the Figure 10, parallel pipelining improves the performance of executing MTTKRP for both 3D and 4D tensors. This suggests that, in addition to the optimization of kernel functions, end-to-end data transfer, storing, and computation are equally important in the computation of MTTKRP. Especially the data transfer between the device and the host, which has a decisive impact on the efficiency of the whole computation process.

For tensors with fewer non-zero elements, such as the 3d tensor *vast* and *nell-2*, and the 4d tensor *uber* and *nips*, the transfer time from the host to the device is shorter, so the overlapping ratio is larger, indicating higher performance speedup. Especially for the smallest tensor vast, which achieves an approximate 2.0× speedup ratio. In contrast, tensors with more non-zero elments (e.g., *flickr-3d*) take longer transmission time, so pipelining can not completely cover the transmission time. In this case, the computation still needs to wait for

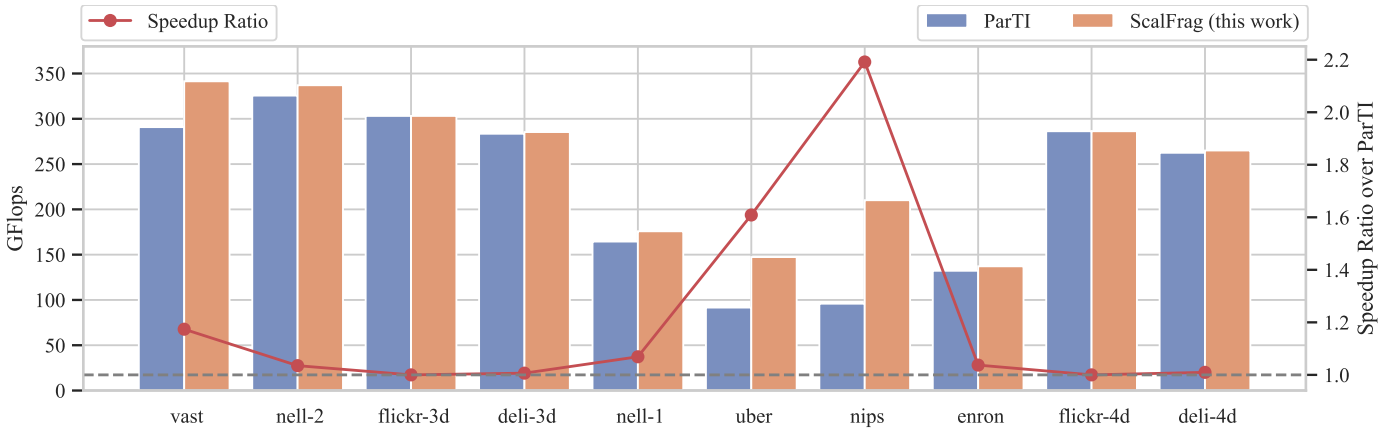

Fig. 9. The performance of MTTKRP kernels with *ScalFrag* and ParTI.

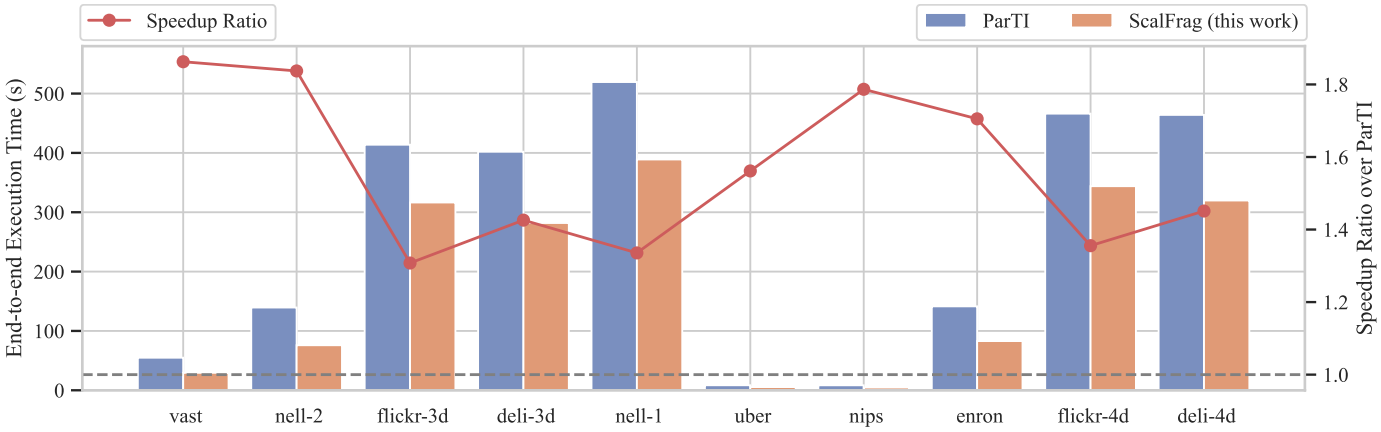

Fig. 10. The end-to-end performance of MTTKRP with *ScalFrag* and ParTI.

the data transmission, meaning the resources remain idle for certain duration. However, *ScalFrag* also achieves more than 1.3× speedup ratios over ParTI. The results prove that *ScalFrag* can be well adapted to tensors of various sizes and sparsity characteristics.

Compared with ParTI, *ScalFrag* improves the performance of MTTKRP by about 1.3× to 2.0×. *ScalFrag* successfully alleviates the problem of long data calculation waiting time mentioned in Section III-B and can be effectively applied to the optimization of CPD procedure.

D. Impact of Block and Stream Settings

Segmenting tensor data can significantly improve the parallelism of data transfer and computation, thus speeding up the overall processing. However, the choice of the number of segments is not arbitrary, and needs to be considered based on a variety of factors such as the size of the tensor, the sparsity ratio, and the computational capability of the GPU. From Figure 11, we can see that the settings of the number of segments and the number of CUDA streams have an important impact on the performance of MTTKRP.

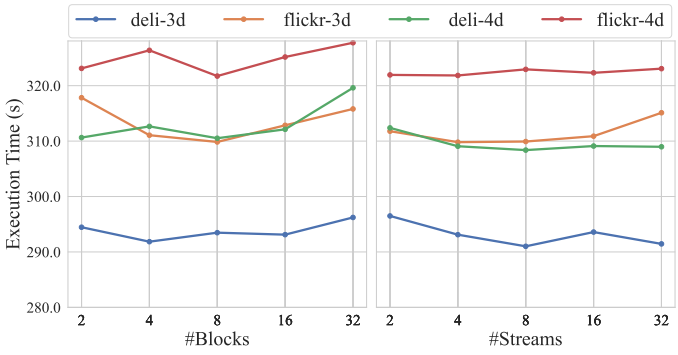

Fig. 11. The performance of MTTKRP with different settings.

When we fix the number of segments to 4 and set the number of CUDA streams to different values, the performance of MTTKRP is different; and when the number of CUDA streams is fixed to 4 and the number of segments is set to different values, the performance of MTTKRP is different. Although the difference among them is not obvious, in practical applications, we may find that the optimal number of segments is required for different tensor datasets. The appropriate number of segments and CUDA streams can improve the efficiency of MTTKRP computation to some extent.

CUDA streams allow us to perform multiple tasks on the GPU at the same time, thus enabling asynchronous data transfer and computation. However, the setting of the number of CUDA streams also requires careful consideration. If the number of CUDA streams is too high, although it can further improve parallelism, it may also lead to too much competition between data transfer and computation, which may cause resource conflicts and performance degradation. On the contrary, if the number of CUDA streams is too small, although it can reduce the possibility of resource conflicts, it may also lead to resource wastage between data transfer and computation, which will also affect performance. We need to flexibly adjust the number of segments and the number of CUDA streams according to the characteristics of different tensor datasets to obtain the best performance.

VI. Related Work

Previous studies have mainly focused on optimizing sparse tensor formats, auto-tuning the tensor computation process, and accelerating sparse tensor computation on different hardware platforms. These studies have proposed various sparse tensor formats and parallel algorithms designed to efficiently process and analyze data on multiple hardware platforms such as CPU, GPU, and other emerging hardware.

A. Tensor Auto-tuning Optimization

There are many prior studies working on optimizing sparse tensor programs. TACO [16] describes the first technique for compiling compound tensor algebra expressions with dense and sparse operands to fast kernels. Stephen et al. [6] describe and implement a new technique for generating tensor algebra kernels that efficiently compute on tensors stored in disparate formats. WACO [40] introduces an innovative approach to co-optimize the format and schedule of a given sparsity pattern

within a sparse tensor program. It introduces a unique feature extractor utilizing a sparse convolutional network, which plays a crucial role in designing a cost model that accounts for different sparsity patterns. Furthermore, it employs a graph-based ANNS, a discretized version of the gradient-based search, efficiently and accurately finds the best format and schedule in the large search space of the co-optimization. Mosaic [3], which is a sparse tensor algebra compiler that can bind tensor expressions to external functions of other tensor algebra libraries and compilers.

Other studies explore format selection based on machine learning models to efficiently leverage existing sparse formats such as SpTFS [36] which adopts both supervised learning-based and unsupervised learning-based methods to predict the best of COO, HiCOO, and CSF formats to compute MTTKRP for a given sparse tensor. Chou et al. [7] provide a method to generate code that efficiently converts sparse tensors between disparate storage formats (data layouts) such as CSR, DIA, ELL, and many others.

B. Tensor Computation Acceleration

There have also been many prior studies working on accelerating sparse tensor programs on multiple hardware platforms. To accelerate computation on GPUs, Li et al. [20] introduce a parallel algorithm and its GPU implementation for SpTTM within ParTI [19], achieved by parallelizing the algorithm across fibers. However, their approach encounters challenges such as load imbalance and warp divergence on GPU platforms due to varying sizes of fibers in sparse tensors. Additionally, they implement the SpMTTKRP algorithm in ParTI [19], dividing data partitions based on tensor non-zeros. Yet, the performance of their method is constrained by the overhead of atomic operations during slice updates.

Sasindu et al. [38] introduce the characteristics of a custom memory controller that can reduce the total memory access time of sparse MTTKRP on FPGAs and develop a custom FPGA accelerator design [39] with (1) PEs consisting of a collection of pipelines that can concurrently process multiple elements of the input tensor and (2) memory controllers to exploit the spatial and temporal locality of the external memory accesses of the computation which achieved great performance compared with the state-of-the-art CPU and GPU implementations.

Other researches have focused on different platforms. Smith et al. [32] enhance the performance of MTTKRP on the Intel Xeon Phi Knights Landing manycore processor. GigaTensor [14] addresses large-scale sparse tensors by offering a scalable framework utilizing the MapReduce framework. Blanco et al. [4] propose cstf to accelerate tensor decompositions by employing a queuing strategy to capitalize on dependency and data reuse with the Spark engine on distributed platforms. Also, Kaya et al. [15] scale CPD on distributed memory systems using the message passing interface (MPI).

C. Customized Tensor Accelerator Architecture

Numerous DL accelerators have recently been proposed for sparse computations [10], [25]. There are also tensor algebra accelerators for scientific applications. The prevalence of tensor operations in big data applications has stimulated research efforts. These efforts focus on accelerating sparse matrix-dense vector (SpMV) [25] and sparse matrix-matrix (SpMM) kernel [9]. By tightly coupling accelerator processing elements (PEs) with multi-core cores, data transfer is avoided, allowing accelerators to reuse the CPU memory system and its virtual addresses. For example, T2S-Tensor [35] is a language and compilation framework used to generate high-performance hardware fore dense tensor conputations such as GEMM, MTTKRP and DTTMc. ExTensor [13] proposes a new approach for performing general tensor algebra using hierarchical and compositional intersection. Tensaurus [34] co-designs the hardware and a sparse storage format, allowing accessing the sparse data in vectorized and streaming fashion and maximizing the utilization of the memory bandwidth. All these show significant speedup and energy benefit.

Additionally, Eric et al. [27] propose hardware extensions to accelerators for supporting numerous format combinations seamlessly and demonstrate better performance.

VII. Conclusion

Canonical polyadic decomposition (CPD) stands out as a prominent choice, widely embraced across numerous scientific disciplines and practical applications due to its effectiveness in capturing multi-linear relationships. With the continuous expansion of big data and deep learning applications, the performance optimization of MTTKRP, as a core operation in many key algorithms, is particularly important. In this paper, MTTKRP optimization methods for GPU platforms are discussed in depth, and an adaptive launch strategy is designed, which dynamically selects the best combination of launch parameters based on the sparsity features of the tensor to ensure that the computational resources of the GPU are fully utilized. We also introduce a pipelined parallelism technique, which enables multiple computation tasks to be executed in parallel by optimizing the dependencies between tasks, thus significantly reducing the overall computation time.

The experimental results show that *ScaleFrag* can obtain high performance on various tensor datasets, which is an improvement to the MTTKRP computation. For future work, we will continue to investigate how to further improve the MTTKRP performance to meet more application scenarios.

Acknowledgements

This work is supported by the National Key R&D Program of China (Grant No. 2023YFB3001604), and the Fundamental Research Funds for the Central Universities (Grant No. 2462023YJRC023). Qingxiao Sun is the corresponding author.

REFERENCES

[1] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G.S., Davis, A., Dean, J., Devin, M., et al.: Tensorflow: Large-scale machine learning on heterogeneous distributed systems. arXiv preprint arXiv:1603.04467 (2016)

[2] Appellof, C.J., Davidson, E.R.: Strategies for analyzing data from video fluorometric monitoring of liquid chromatographic effluents. Analytical Chemistry **53**(13), 2053–2056 (1981)

[3] Bansal, M., Hsu, O., Olukotun, K., Kjolstad, F.: Mosaic: An interoperable compiler for tensor algebra. Proceedings of the ACM on Programming Languages **7**(PLDI), 394–419 (2023)

[4] Blanco, Z., Liu, B., Dehnavi, M.M.: Cstf: Large-scale sparse tensor factorizations on distributed platforms. In: ICPP '18 (2018)

[5] Bruns-Smith, D., Baskaran, M.M., Ezick, J., Henretty, T., Lethin, R.: Cyber security through multidimensional data decompositions. In: CYBERSEC '16 (2016)

[6] Chou, S., Kjolstad, F., Amarasinghe, S.: Format abstraction for sparse tensor algebra compilers. Proceedings of the ACM on Programming Languages **2**, 123:1–123:30 (2018)

[7] Chou, S., Kjolstad, F., Amarasinghe, S.: Automatic generation of efficient sparse tensor format conversion routines. In: PLDI '20 (2020)

[8] Dun, M., Li, Y., Yang, H., Sun, Q., Luan, Z., Qian, D.: An optimized tensor completion library for multiple gpus. In: ICS '21 (2021)

[9] Gerogiannis, G., Yesil, S., Lenadora, D., Cao, D., Mendis, C., Torrellas, J.: Spade: A flexible and scalable accelerator for spmm and sddmm. In: ISCA '23 (2023)

[10] Han, S., Liu, X., Mao, H., Pu, J., Pedram, A., Horowitz, M.A., Dally, W.J.: Eie: Efficient inference engine on compressed deep neural network. ACM SIGARCH Computer Architecture News **44**(3), 243–254 (2016)

[11] Harshman, R.A., et al.: Foundations of the parafac procedure: Models and conditions for an "explanatory" multi-modal factor analysis. UCLA working papers in phonetics **16**(1), 84 (1970)

[12] He, H., Henderson, J., Ho, J.C.: Distributed tensor decomposition for large scale health analytics. In: IW3C2 '19 (2019)

[13] Hegde, K., Asghari-Moghaddam, H., Pellauer, M., Crago, N., Jaleel, A., Solomonik, E., Emer, J., Fletcher, C.W.: Extensor: An accelerator for sparse tensor algebra. In: MICRO '19 (2019)

[14] Kang, U., Papalexakis, E., Harpale, A., Faloutsos, C.: Gigatensor: scaling tensor analysis up by 100 times-algorithms and discoveries. In: KDD '12 (2012)

[15] Kaya, O., Uçar, B.: Scalable sparse tensor decompositions in distributed memory systems. In: SC '15 (2015)

[16] Kjolstad, F., Kamil, S., Chou, S., Lugato, D., Amarasinghe, S.: The tensor algebra compiler. Proceedings of the ACM on Programming Languages **1**, 77:1–77:29 (2017)

[17] Kolda, T.G., Bader, B.W.: Tensor decompositions and applications. SIAM review **51**(3), 455–500 (2009)

[18] Kolda, T.G., Sun, J.: Scalable tensor decompositions for multi-aspect data mining. In: ICDMW '08 (2008)

[19] Li, J., Ma, Y., Yan, C., Sun, J., Vuduc, R.: Parti: a parallel tensor infrastructure for data analysis. In: NIPS, Tensor-Learn Workshop (2016)

[20] Li, J., Ma, Y., Yan, C., Vuduc, R.: Optimizing sparse tensor times matrix on multi-core and many-core architectures. In: IA3 '16 (2016)

[21] Li, J., Sun, J., Vuduc, R.: Hicoo: Hierarchical storage of sparse tensors. In: SC '18 (2018)

[22] Liu, B., Wen, C., Sarwate, A.D., Dehnavi, M.M.: A unified optimization approach for sparse tensor operations on gpus. In: CLUSTER '17' (2017)

[23] Nisa, I., Li, J., Sukumaran-Rajam, A., Rawat, P.S., Krishnamoorthy, S., Sadayappan, P.: An efficient mixed-mode representation of sparse tensors. In: SC '19 (2019)

[24] Nisa, I., Li, J., Sukumaran-Rajam, A., Vuduc, R., Sadayappan, P.: Load-balanced sparse mttkrp on gpus. In: IPDPS '19 (2019)

[25] Nurvitadhi, E., Mishra, A., Marr, D.: A sparse matrix vector multiply accelerator for support vector machine. In: CASES '15 (2015)

[26] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al.: Pytorch: An imperative style, high-performance deep learning library. Advances in neural information processing systems **32** (2019)

[27] Qin, E., Jeong, G., Won, W., Kao, S.C., Kwon, H., Srinivasan, S., Das, D., Moon, G.E., Rajamanickam, S., Krishna, T.: Extending sparse tensor accelerators to support multiple compression formats. In: IPDPS '21 (2021)

[28] Rettinger, A., Wermser, H., Huang, Y., Tresp, V.: Context-aware tensor decomposition for relation prediction in social networks. Social Network Analysis and Mining **2**(4), 373–385 (2012)

[29] Shi, Y., Karatzoglou, A., Baltrunas, L., Larson, M., Hanjalic, A., Oliver, N.: Tfmap: optimizing map for top-n context-aware recommendation. In: SIGIR '12 (2012)

[30] Smith, S., Choi, J.W., Li, J., Vuduc, R., Park, J., Liu, X., Karypis, G.: FROSTT: The formidable repository of open sparse tensors and tools (2017), `http://frostt.io/`

[31] Smith, S., Karypis, G.: Tensor-matrix products with a compressed sparse tensor. In: IA3 '15 (2015)

[32] Smith, S., Park, J., Karypis, G.: Sparse tensor factorization on many-core processors with high-bandwidth memory. In: IPDPS '17 (2017)

[33] Smith, S., Ravindran, N., Sidiropoulos, N.D., Karypis, G.: Splatt: Efficient and parallel sparse tensor-matrix multiplication. In: IPDPS '15 (2015)

[34] Srivastava, N., Jin, H., Smith, S., Rong, H., Albonesi, D., Zhang, Z.: Tensaurus: A versatile accelerator for mixed sparse-dense tensor computations. In: HPCA '20 (2020)

[35] Srivastava, N., Rong, H., Barua, P., Feng, G., Cao, H., Zhang, Z., Albonesi, D., Sarkar, V., Chen, W., Petersen, P., et al.: T2s-tensor: Productively generating high-performance spatial hardware for dense tensor computations. In: FCCM '19 (2019)

[36] Sun, Q., Liu, Y., Yang, H., Dun, M., Luan, Z., Gan, L., Yang, G., Qian, D.: Input-aware sparse tensor storage format selection for optimizing mttkrp. IEEE Transactions on Computers **71**(8), 1968–1981 (2022)

[37] Vasilescu, M.A.O.: Multilinear projection for face recognition via canonical decomposition. In: FG '11 (2011)

[38] Wijeratne, S., Wang, T.Y., Kannan, R., Prasanna, V.: Towards programmable memory controller for tensor decomposition. In: DATA '22 (2022)

[39] Wijeratne, S., Wang, T.Y., Kannan, R., Prasanna, V.: Accelerating sparse mttkrp for tensor decomposition on fpga. In: FPGA '23 (2023)

[40] Won, J., Mendis, C., Emer, J.S., Amarasinghe, S.: Waco: learning workload-aware co-optimization of the format and schedule of a sparse tensor program. In: ASPLOS '23 (2023)