STTID: High-Performance Sparse Tensor-Train Interpolative Decomposition

Zhaonan Meng*, Miles Stoudenmire†, Karl Pierce‡, Frank Mueller*, Jiajia Li*
*{zmeng5, fmuelle, jli256}@ncsu.edu, Department of Computer Science, North Carolina State University, Raleigh, USA
†{mstoudenmire}@flatironinstitute.org, Center for Computational Quantum Physics, Flatiron Institute, New York, USA
‡kmp5@umd.edu, Department of Mathematics, University of Maryland, College Park, USA

***Abstract*—Tensor-Train (TT) decomposition provides a low-rank tensor approximation by factorizing a tensor into interconnected three-dimensional core tensors (TT-cores). Conventional TT methods, originally developed for dense tensors, tend to neglect sparsity and produce dense TT-cores, or fail to yield valid results due to their limited capability to handle sparse data. This limitation leads to high computational overhead and increased memory consumption both during decomposition and in downstream TT-core operations. In this work, we propose Sparse Tensor-Train Interpolative Decomposition (STTID), a novel algorithm based on the compressed format that stably decomposes sparse tensors and preserves sparsity in the TT-cores. We address computational bottlenecks in STTID through algorithmic optimizations such as selective data separation and hash-based indexing, and further by developing a hybrid CPU-GPU design. Our evaluation shows that STTID produces sparse TT-cores for sparse input tensors without compromising approximation accuracy. STTID is the first algorithm to efficiently decompose large sparse tensors that existing TT algorithms cannot handle. Our GPU-accelerated implementation achieves speedups of 77.9×–169.9× on an NVIDIA A5000 and 194.3×-550.3× on an H100 GPU over the CPU counterpart for tensors with more than one million nonzeros.**

I. Introduction

Modern data often resides in multidimensional spaces, demanding mathematical tools beyond traditional matrix algebra. Tensors, as multidimensional generalizations of matrices, have become indispensable for modeling such data across fields such as machine learning [64, 102], signal processing [88], neuroimaging analysis [37], and quantum simulation [22, 73]. When processing tensors, a fundamental challenge stems from what is called the *curse of dimensionality* [10, 11]: the number of elements in a tensor grows exponentially with its dimensionality, leading to severe computational and memory demands.

The scaling limitation motivates the development of *tensor decompositions* [43, 88], which can mitigate this curse by replacing full tensors with compact representations. While traditional tensor decompositions such as CANDECOMP/-PARAFAC (CP) and Tucker [43] remain the most widely studied and applied models, Tensor-Train (TT) decomposition [10, 69] has attracted increasing interest as a powerful compressed representation and an effective solution to mitigate the curse of dimensionality in applications such as neural network compression [64, 102], knowledge graph reasoning [57], and high-dimensional partial differential equations (PDEs) [79].

In many scenarios, tensors exhibit sparsity [5, 10, 75], a desirable property that provides significant computational advantages. In domains such as quantum simulation and neural networks, preserving sparsity in tensor factors is particularly valuable for quality assessment and for algorithmic efficiency [4, 22, 64, 101, 103]. The growing prevalence of sparse tensors has spurred extensive research into scalable methods for their processing, especially within CP and Tucker decompositions [8, 41, 49, 93].

However, performing TT decomposition on sparse tensors is non-trivial. The foundational TT algorithm, TT-SVD [69], achieves optimal approximation accuracy for given ranks but disrupts the input structure, leading to a loss of sparsity during the sequential SVD process. The TT-cross [70], designed for partially-accessible tensor functions, inherently preserves input sparsity through the cross decomposition but struggles to achieve accurate TT approximations and often breaks down when applied to highly sparse tensors. *The current lack of a robust method that retains and exploits sparsity during TT decomposition constrains its scalability for sparse tensors. This limitation also hinders potential performance gains in downstream computation and data interpretability by failing to produce sparsity-preserving output.*

Motivated by the benefits of sparse tensors and the limitations of existing methods, we propose the following research objective: ***Develop a TT algorithm that preserves sparsity from the input to the output tensor train, leveraging the computational advantages of sparse tensors to ensure scalable decomposition.*** Achieving this aim requires addressing three main challenges.

Challenge 1: Developing a robust TT algorithm that preserves sparsity in output while maintaining approximation accuracy. As established, current TT methods, such as TT-SVD and TT-cross, struggle with sparse tensors, either by losing sparsity or failing to complete the decomposition. For conventional dense TT methods, sparsity could be enforced via regularization [14, 39]. By enforcing zeros via ℓ^1-norm constraints, one can induce sparsity in TT without severely compromising accuracy. However, the effectiveness of regularization is highly dependent on the data, and regularization offers little improvement in scalability, as the underlying TT decomposition remains dense.

Challenge 2: Designing efficient TT algorithms that operate directly on compressed sparse tensor formats. Compressed

storage formats, such as the coordinate format [48], store only the nonzero entries of a sparse tensor, which represents a small fraction of the entire data. These formats have proven both memory- and computation-efficient for sparse CP and Tucker decompositions [8, 41] and enable the handling of higher-dimensional tensors that would be infeasible with dense representations. As of today, existing TT algorithms, including some modern variants [36, 51, 66, 80], are not designed for sparse formats and rely on dense implementations. Consequently, they have yet to demonstrate scalable processing capabilities for the large sparse tensors encountered in real-world applications.

Challenge 3: The high computational cost of the TT decomposition on large sparse tensors poses a significant demand on computing resources. TT computation is inherently expensive due to the high dimensionality of tensors. For instance, the asymptotic complexity of TT-SVD for a d-dimensional hypercubic tensor with dimension size n is $\mathcal{O}(n^{d+1})$ [69, 80]. While compressed representations reduce the problem size to the number of nonzeros, large sparse tensors with many nonzero entries render serial execution highly inefficient. Thus, leveraging parallel computational power, such as GPUs, is critical for achieving high performance, requiring both parallelization and specialized optimizations strategies that differ from those used for dense tensors.

To address these three research challenges, we propose an efficient sparse TT-ID algorithm, termed STTID, for both CPUs and GPUs, built using a compressed sparse format and enhanced with multiple performance optimization strategies. ***STTID is the first approach to enable efficient TT decomposition of large-scale sparse tensors arising from real applications while preserving sparsity in the TT-cores, thereby fully exploiting sparsity not only during the decomposition process but also in downstream computations.*** In summary, our major contributions are as follows:

- We propose a new Tensor-Train Interpolative Decomposition (TT-ID) algorithm designed for sparse tensors. Leveraging interpolative decomposition (ID) via partial rank-revealing LU (PRRLU), TT-ID can robustly compute optimal TT representations while maintaining sparsity from the decomposition process to output. (Section IV-A)
- We design a sparse TT-ID algorithm, STTID, that operates directly on a compressed format of sparse data and introduce optimizations such as selective data separation and hash table-based Gaussian elimination to enhance its performance. (Section IV-B, Section V)
- We develop a hybrid CPU-GPU implementation of STTID that strategically manages data placement, asynchronous transfers, and on-device hash tables, along with other optimizations, to substantially accelerate large sparse tensor processing. (Section VI)
- We evaluate STTID on tensors of varying sizes and sparsity. (1) Compared with four state-of-the-art baselines, STTID consistently yields the sparsest TT-cores with comparable accuracy and efficiently processes large tensors that are infeasible for the baselines. (2) The hybrid CPU-GPU implementation, enhanced with optimization strategies, further achieves substantial speedups on large real-world tensors with over one million nonzeros, ranging from 77.9× to 169.9× on an NVIDIA A5000 and from 194.3× to 550.3× on an NVIDIA H100. (Section VII)

II. Related Work

TT Algorithms. Beyond the foundational TT-SVD [69] and TT-cross [70] methods, several recent approaches have introduced further advances [2, 36, 51, 66, 80], introducing techniques such as randomization or tall-skinny QR decomposition. For high performance, several parallel TT algorithms for dense tensors have been proposed [13, 17, 80, 86], including initial efforts on GPUs [40, 56]. Most existing works on decomposition of sparse tensors focus on parallel CP or Tucker decompositions [41, 46, 76, 91, 92], as well as on sparse tensor operations such as tensor contractions [7, 35, 55, 71, 95] and tensor-matrix products [6, 32, 42, 49, 54, 63, 93, 94]. However, scalable TT decomposition for sparse tensors has not been studied, which motivates our work.

Sparse Decomposed Factors. A few studies have focused on introducing sparsity into factors of matrix and tensor decompositions. In matrix algebra, QR decomposition has been explored to replace SVD for sparse factorization [103]. Swaminathan et al. [98] proposed a sparse low-rank approximation method, while Yuan and Yang [104] introduced an alternating direction method to achieve sparsity. In the tensor domain, Shah et al. [85] investigated sparse CP decomposition, whereas other works have focused on extracting sparse factors from Tucker decomposition [1, 82]. However, the preservation of sparsity in the output TT remains unexplored.

Interpolative Decomposition. Interpolative decomposition [53], a structure-preserving matrix factorization, has attracted attention in sparse tensor studies such as Tucker and CP decompositions [82, 105]. It is typically achieved by rank-revealing methods such as rank-revealing LU/QR factorizations or nuclear score techniques [23, 31, 34, 38]. In the TT domain, although interpolation has been employed in TT-cross, its application to sparse TT scenarios remains uninvestigated and introduces distinct challenges.

III. Background

This section reviews sparse tensors, the tensor-train decomposition, and the interpolation-based TT algorithm.

A. Sparse Tensors

Tensors generalize vectors and matrices to represent relationships among more than two dimensions [24, 77, 88]. The number of dimensions of a tensor is called its $order$, with each dimension termed a $mode$ [43]. For example, a 4^{th}-order tensor $\boldsymbol{\mathcal{T}} \in \mathbb{R}^{n_1 \times n_2 \times n_3 \times n_4}$ has four modes with sizes n_1, n_2, n_3, n_4 and total size $n_1 n_2 n_3 n_4$. To apply matrix methods, a tensor can be $reshaped$ into a matrix by changing modes while preserving the linear index order of the elements in memory.

Like sparse matrices, sparse tensors are multi-dimensional arrays that contain mostly zero values. Sparse tensors are

$$\boldsymbol{\mathcal{T}}(u_1,\ldots,u_d) \approx \sum^{\mathcal{I},\mathcal{J}} \underbrace{\boldsymbol{\mathcal{T}}(u_1,\mathcal{J}_1)}_{\boldsymbol{\mathcal{G}}_1}(\underbrace{\boldsymbol{\mathcal{T}}(\mathcal{I}_1,\mathcal{J}_1)}_{\mathbf{X}_1})^{-1}\underbrace{\boldsymbol{\mathcal{T}}(\mathcal{I}_1,u_2,\mathcal{J}_2)}_{\boldsymbol{\mathcal{G}}_2}\ldots\underbrace{\boldsymbol{\mathcal{T}}(\mathcal{I}_{d-2},u_{d-1},\mathcal{J}_{d-1})}_{\boldsymbol{\mathcal{G}}_{d-1}}(\underbrace{\boldsymbol{\mathcal{T}}(\mathcal{I}_{d-1},\mathcal{J}_{d-1})}_{\mathbf{X}_{d-1}})^{-1}\underbrace{\boldsymbol{\mathcal{T}}(\mathcal{I}_{d-1},u_d)}_{\boldsymbol{\mathcal{G}}_d}. \quad (2)$$

widely used in various domains where data is naturally high-dimensional and sparse, ranging from computer science fields such as recommendation systems [87], data analytics [21] and completion [96] to the natural sciences, such as quantum chemistry and quantum physics [22, 33, 89]. Sparsity enables compressed representations of sparse matrices and tensors, such as the coordinate (COO) format [50, 81], which explicitly stores all nonzero entries along with their coordinate indices. Other formats, including compressed sparse row/column (CSR/CSC) and compressed sparse fiber (CSF), further reduce memory consumption by compressing repeated indices into pointers [81, 93].

B. Tensor-Train Decomposition

One major obstacle in processing tensors is the exponential growth of multi-dimensional data space, known as the curse of dimensionality [10, 11], which often makes the direct processing of an entire tensor expensive. To overcome this curse, one powerful technique is the Tensor-Train (TT) decomposition, also known as Matrix Product States (MPS) [99].

TT decomposition represents high-order tensors in a compressed form as a series of low-order tensors. Oseledets [69] introduces the mathematical formulation for decomposing a d^{th}-order tensor $\boldsymbol{\mathcal{T}} \in \mathbb{R}^{n_1\times n_2\times\ldots\times n_d}$ into a TT form as

$$\boldsymbol{\mathcal{T}}(u_1,\ldots,u_d) \cong \sum_{j_0,\ldots,j_d}^{r_0,\ldots,r_d} \boldsymbol{\mathcal{G}}_1(j_0,u_1,j_1)\ldots\boldsymbol{\mathcal{G}}_d(j_{d-1},u_d,j_d), \quad (1)$$

where $\boldsymbol{\mathcal{G}}_k \in \mathbb{R}^{r_{k-1}\times n_k\times r_k}$ for $k=1,2,\ldots,d$ are the third-order tensor-train cores (*TT-cores*), with boundary cores $\boldsymbol{\mathcal{G}}_1$ and $\boldsymbol{\mathcal{G}}_d$ reducing to matrices with $r_0{=}r_d{=}1$. The dimensions of the interconnecting indices between the cores, r_k for $k=1,\ldots,d-1$, are called *TT-ranks*. With suitable TT-ranks, the contraction of TT-cores (*TT-approximation*) can closely approximate or exactly reproduce the tensor $\boldsymbol{\mathcal{T}}$.

In practice, the TT decomposition is primarily used to approximate tensors in a more compressed form by truncating the TT-ranks using a low-rank and error-bounded approximation, such as Tensor-Train Singular Value Decomposition (TT-SVD) [69]. TT-SVD sequentially extracts TT-cores together with their corresponding TT-ranks by iteratively applying the singular value decomposition (SVD) to matricized tensors along the tensor modes. Although TT-SVD yields an optimal Frobenius-norm TT approximation via SVD, it suffers from high computational and memory costs and fails to preserve input properties such as sparsity in the orthogonal cores.

C. TT Algorithm via Cross Interpolation

An alternative to SVD for deriving a TT is to use cross decomposition [9, 28], which factorizes a matrix into factors constructed from its own entries. The cross decomposition of a matrix $\mathbf{M}\in\mathbb{R}^{m\times n}$ approximates $\mathbf{M}$ as $\mathbf{M}\approx\mathbf{C}\mathbf{X}^{\dagger}\mathbf{R}$, where:

- $\mathbf{C}=\mathbf{M}(:,\mathcal{J})$ and $\mathbf{R}=\mathbf{M}(\mathcal{I},:)$ consist of the columns $\mathcal{J}\subseteq\{1,\ldots,n\}$ and rows $\mathcal{I}\subseteq\{1,\ldots,m\}$ of $\mathbf{M}$, respectively, termed the *column/row skeleton*.
- $\mathbf{X}=\mathbf{M}(\mathcal{I},\mathcal{J})$ is the intersection of $\mathbf{C}$ and $\mathbf{R}$, referred to as the *cross matrix*. If $\mathbf{X}$ is square and nonsingular, its pseudo-inverse $\mathbf{X}^{\dagger}$ coincides with the inverse $\mathbf{X}^{-1}$.

If $\mathbf{M}$ is sparse, $\mathbf{C}$, $\mathbf{R}$, and $\mathbf{X}$ naturally preserve the sparsity of $\mathbf{M}$. The quality of the cross approximation for a given rank r depends on how r selected rows and columns span the matrix range. These rows and columns are typically determined by rank-revealing methods [100]. Such a cross approximation can be related to a *one-sided* form by absorbing $\mathbf{X}^{\dagger}$ into either skeleton, yielding an interpolative decomposition (ID) $\mathbf{M}\approx\mathbf{C}\mathbf{Z}$ or $\mathbf{M}\approx\mathbf{X}\mathbf{R}$, where $\mathbf{X}$ and $\mathbf{Z}$ are *interpolation matrices* that express the non-selected rows/columns as linear combinations of the skeletons [53, 58].

TT-cross [66, 70], which incorporates the cross interpolation described above into the TT decomposition, produces a TT in the tensor cross interpolation (TCI) form of Equation (2) [65, 66, 84]. In Equation (2), $\mathcal{I}_k$ and $\mathcal{J}_k$ ($k=1,\ldots,d-1$) denote the global tensor indices. The entries $\boldsymbol{\mathcal{T}}(\mathcal{I}_{k-1},u_k,\mathcal{J}_k)$ and $\boldsymbol{\mathcal{T}}(\mathcal{I}_k,\mathcal{J}_k)$ are used to construct the TT-core $\boldsymbol{\mathcal{G}}_k$ and the intersection matrix $\mathbf{X}_k$, respectively. Compared with the TT form in Equation (1), the TCI form introduces additional matrix inverses between TT-cores. However, these inverses need not be computed or stored explicitly, as $\mathbf{X}_k$ is the subset of $\boldsymbol{\mathcal{G}}_k$ (see details in Section IV-A and Section VII-E). Therefore, in the TCI form, the tensor train can be stored using only the skeleton indices $\mathcal{I}_k,\mathcal{J}_k$ and the corresponding entries of the input tensor.

TT-cross appears promising for preserving sparsity in TT-cores. Since its introduction [70], TT-cross has evolved into various implementations, such as [66]. However, these methods are not well-suited for sparse tensors. They employ a *back-and-forth sweep* optimized for tensor-like functions common in PDEs or optimization problems [18, 97], which are only partially known and cannot be fully evaluated. Each sweep updates the skeleton indices derived from the cross decomposition of matricized tensor slices. For sparse tensors, the evaluated slices, as a small portion of the tensor, could have too few or no nonzero entries, providing little useful information for skeleton updates. This issue can prevent convergence and even lead to unstable behavior, as shown in Section VII.

IV. Sparse Tensor-Train Interpolative Decomposition

This section presents a tensor-train interpolative decomposition algorithm that processes sparse tensors and produces sparse TT-cores, together with its implementation in the coordinate format, STTID.

A. TT-ID Algorithm

Motivated by the need to preserve sparsity in TT decomposition and to overcome the limitations of existing TT algorithms for processing fully-observed sparse tensors, we introduce a new method, Tensor-Train Interpolative Decomposition (TT-ID), adapted from the aforementioned cross interpolation-based TT algorithm. Figure 1 uses the tensor diagram notation [74] to illustrate an overview of our TT-ID when decomposing a four-dimensional tensor $\mathcal{T} \in \mathbb{R}^{n_1 \times \cdots \times n_d}$ ($d = 4$) in a sweep from mode 1 to $d-1$. The core idea is to perform cross decomposition on matricized tensors along each mode within a single sweep, extracting TT-cores that inherit the sparsity of the input tensor.

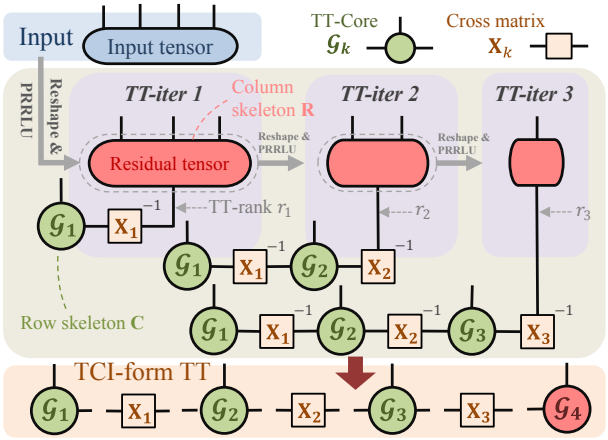

Fig. 1: Iterative computation of TT-cores in TT-ID.

At the k-th TT-ID iteration (*TT-iter*), the following three main operations are performed:
(1) Reshape $\mathcal{T}$ into a matrix $\mathbf{M}$ in shape $(n_k r_{k-1}, \frac{N}{n_k r_{k-1}})$ (Line 4 in Algorithm 1), where N is $\mathcal{T}$'s total size (calculated in Line 1), n_k is the size of mode k, and r_{k-1} denotes the $(k-1)$-th TT-rank determined in the previous iteration (boundary ranks $r_0 = r_d = 1$).
(2) Compute the rank-r cross decomposition of $\mathbf{M}$ ("PRRLU" in Figure 1) to obtain the column/row skeleton $\mathbf{C} \in \mathbb{R}^{n_k r_{k-1} \times r}$ and $\mathbf{R} \in \mathbb{R}^{r \times N/(n_k r_{k-1})}$, and the cross matrix $\mathbf{X}_k \in \mathbb{R}^{r \times r}$ (squares in Figure 1) (Lines 5-8 in Algorithm 1).
(3) Set the k-th TT-rank to r (Line 9), reshape $\mathbf{C}$ into a 3D TT-core $\mathcal{G}_k$ (Line 10, green circles in Figure 1), and take the row skeleton $\mathbf{R}$ as the residual tensor (red ellipses in Figure 1) for the next iteration (Line 12). The last $\mathbf{R}$ obtained in the final TT-iter is reshaped into the terminal TT-core $\mathcal{G}_d$ (Line 14).

Skeleton selection is performed using a partial rank-revealing LU (PRRLU) decomposition [26, 38, 66], which employs Gaussian elimination with complete pivoting to determine the matrix rank and identify the row and column pivots serving as the skeleton indices. Since rank-revealing LU admits existence bounds comparable to those of RRQR [72], and its stability for ID has been demonstrated in practical applications such as Feynman diagram and quantum simulations [65, 83], the choice of PRRLU in TT-ID over alternative methods is motivated by two computational advantages: (1) PRRLU provides both row and column pivots in a single run, whereas RRQR identifies only column skeletons in one pass and requires an additional, more expensive run on the transpose to obtain row pivots. (2) RRQR introduces additional memory overhead due to dense orthogonal factors, while LU-based methods require less memory due to in-place storage.

PRRLU returns a rank-r approximation of $\mathbf{M} \in \mathbb{R}^{m \times n}$ as $\mathbf{P}\mathbf{M}\mathbf{Q}^T \approx \mathbf{L}\mathbf{U}$, where $\mathbf{P}$ and $\mathbf{Q}$ are permutation matrices representing row and column pivoting, $\mathbf{L} \in \mathbb{R}^{m \times r}$ and $\mathbf{U} \in \mathbb{R}^{r \times n}$ are lower and upper trapezoidal matrices. In rank-r PRRLU, the product of $\mathbf{L}$ and $\mathbf{U}$ obtained from the first r Gaussian eliminations exactly reproduces the first r rows and columns of $\mathbf{P}\mathbf{M}\mathbf{Q}^T$. Therefore, we can derive:

$$\mathbf{P}\mathbf{M}\mathbf{Q}^T \approx \underbrace{\mathbf{L}\mathbf{U}(:,1\!:\!r)}_{=\mathbf{P}\mathbf{M}\mathbf{Q}^T(:,1:r)} \underbrace{\mathbf{U}(:,1\!:\!r)^{-1}\mathbf{L}(1\!:\!r,:)^{-1}}_{=(\mathbf{P}\mathbf{M}\mathbf{Q}^T(1:r,1:r))^{-1}} \underbrace{\mathbf{L}(1\!:\!r,:)\mathbf{U}}_{=\mathbf{P}\mathbf{M}\mathbf{Q}^T(1:r,:)}. \quad (3)$$

Since $\mathbf{P}$ and $\mathbf{Q}$ are orthogonal, with inverses equal to their transposes, Equation (3) further simplifies to

$$\mathbf{M} \approx \underbrace{\mathbf{P}^T\mathbf{P}\mathbf{M}\mathbf{Q}^T(:,1\!:\!r)}_{\mathbf{C}=\mathbf{M}(:,\mathcal{J})} \underbrace{(\mathbf{P}\mathbf{M}\mathbf{Q}^T(1\!:\!r,1\!:\!r))^{-1}}_{\mathbf{X}^{-1}=\mathbf{M}(\mathcal{I},\mathcal{J})^{-1}} \underbrace{\mathbf{P}\mathbf{M}\mathbf{Q}\mathbf{Q}^T(1\!:\!r,:)}_{\mathbf{R}=\mathbf{M}(\mathcal{I},:)}. \quad (4)$$

Equation (4) shows that the PRRLU recovery is equivalently expressed in a double-sided interpolative format, yielding a rank-r cross decomposition of $\mathbf{M}$. From PRRLU, we extract just the revealed rank r and the permutation arrays (the vector forms of $\mathbf{P}$ and $\mathbf{Q}$).

Algorithm 1 Tensor-Train Interpolative Decomposition

Input: Tensor $\mathcal{T} \in \mathbb{R}^{n_1 \times n_2 \times \ldots \times n_d}$, maximum allowable TT-rank $r_{\max} \geq 1$, rank truncation tolerance $\epsilon > 0$.
Output: TT cores $\mathcal{G}_1, \ldots, \mathcal{G}_d$, with intermediate cross matrices $\mathbf{X}_1, \ldots, \mathbf{X}_{d-1}$.

1: $N \leftarrow \prod_{k=1}^{d} n_k$
2: $r_0, r_d \leftarrow 1$ ▷ Boundary TT-ranks
3: **for** $k = 1$ **to** $d-1$ **do**
4: $\quad \mathbf{M} \leftarrow \texttt{reshape}(\mathcal{T}, [n_k r_{k-1}, \frac{N}{n_k r_{k-1}}])$
5: $\quad r, \boldsymbol{perm\_r}, \boldsymbol{perm\_c} \leftarrow \texttt{PRRLU}(\mathbf{M}, r_{\max}, \epsilon)$
6: $\quad \mathcal{I}_k \leftarrow \boldsymbol{perm\_r}(1\!:\!r), \mathcal{J}_k \leftarrow \boldsymbol{perm\_c}(1\!:\!r)$
7: $\quad \mathbf{C} \leftarrow \mathbf{M}(:, \mathcal{J}_k), \mathbf{R} \leftarrow \mathbf{M}(\mathcal{I}_k, :)$ ▷ Row/Col skeleton
8: $\quad \mathbf{X}_k \leftarrow \mathbf{M}(\mathcal{I}_k, \mathcal{J}_k)$ ▷ Cross matrix
9: $\quad r_k \leftarrow r$ ▷ k-th TT-rank
10: $\quad \mathcal{G}_k \leftarrow \texttt{reshape}(\mathbf{C}, [r_{k-1}, n_k, r_k])$ ▷ k-th TT-core
11: $\quad N \leftarrow \frac{N r_k}{n_k r_{k-1}}$
12: $\quad \mathcal{T} \leftarrow \mathbf{R}$
13: **end for**
14: $\mathcal{G}_d \leftarrow \texttt{reshape}(\mathbf{R}, [r_{d-1}, n_d, r_d])$
15: **return** $\mathcal{G}_1, \mathbf{X}_1, \mathcal{G}_2, \ldots, \mathbf{X}_{d-1}, \mathcal{G}_d$

TT-ID returns an interpolation form of TT as Equation (2) with *left-nested* [66, 84] skeleton indices. In practice, the cross $\mathbf{X}_k$ need not be stored, since (1) it is a subset of $\mathcal{G}_k$ obtainable via selecting $\mathcal{I}_k$ rows in $\mathcal{G}_k$, and (2) $\mathcal{G}_k \mathbf{X}_k^{-1}$ can be evaluated efficiently without an explicit inverse, as shown in Section VII-E. Despite forfeiting the orthogonality preserved by TT-SVD, TT-ID ensures a controllable approximation error with left-nested interpolation [78, 84], which diminishes with the growth of the maximum allowable TT-rank ($r_{\max}$). By sequentially decomposing the entire tensor, TT-ID avoids the

risk of failing to identify optimal interpolation skeletons on sparse tensors in traditional TT-cross methods discussed in Section III-C and verified in Section VII-B.

B. STTID Algorithm

Although our TT-ID preserves the sparsity of the input throughout the decomposition and ultimately generates sparse TT-cores, Algorithm 1 still outlines a dense implementation, operating on inputs storing all zero entries, like existing TT algorithms. ***To enable TT-ID to fully leverage sparsity for practical decomposition of large sparse tensors, we develop a sparse TT-ID algorithm, termed* STTID*, which directly operates on tensors in the COO format.*** The COO format provides structural flexibility essential for our algorithm, including *efficient reshaping*, *dimension-agnostic pivoting*, and *incremental construction* (as elaborated below), that cannot be achieved with CSR/CSC or CSF formats, even though they typically obtain higher compression ratios.

STTID applies the procedure of Algorithm 1 to a COO-format tensor. For a sparse tensor of order d with nnz nonzero elements, the COO format stores the nonzero values in a length-nnz array and their corresponding position indices in d separate coordinate arrays. During reshaping steps such as $\mathcal{T} \to \mathbf{M}$ (line 4 in Algorithm 1), matrices are stored in three arrays of length nnz, namely row, col, and val, representing row indices, column indices, and nonzero values, respectively. The shape of $\mathbf{M}$ is $(n_k r_{k-1}, \frac{N}{n_k r_{k-1}})$, which implies that splitting and/or merging of dimensions may occur. *When tensors are stored in the COO format, such reshaping can be efficiently performed by iterating over the coordinates and directly computing the new indices.* In contrast, with CSC/CSR or CSF formats, reshaping requires recomputing pointer structures, which introduces additional overhead of nontrivial magnitude.

Beyond reshaping, the central component of STTID is the sparse partial rank-revealing LU (sparse PRRLU), as detailed in Algorithm 2. The s-th sparse PRRLU iteration can be divided into three phases:

Phase 1: Find the Pivot Entry. PRRLU employs a complete pivoting by permuting linearly independent columns and rows to leading positions. The s-th PRRLU iteration selects the maximum absolute value in the submatrix $\mathbf{M}(s{:}m,\ s{:}n)$ and records its position (μ, λ) as the row and column pivots (Lines 4,5). In an unordered COO $\mathbf{M}$, this pivot can be located through an $\mathcal{O}(\mathbf{M}.nnz)$ traversal over all nonzero entries.

The "partial" aspect of PRRLU refers to terminating the iteration at rank r once either of the following criteria is met: (1) The pivot at step $r+1$ is smaller than a prescribed tolerance ϵ (roundoff makes exact zeros unlikely) [26]; or (2) the rank reaches the upper bound $\min\{r_{\max}(\text{specified maximum rank}), m, n\}$ (Lines 7-9).

Phase 2: Row and Column Pivoting. The row-μ and column-λ entries are swapped with row-s and column-s by updating the indices in $\mathbf{M}.row$ and $\mathbf{M}.col$ accordingly (Lines 13-14). Such swaps are recorded by exchanging the s-th index with the pivot index in the permutation arrays $\boldsymbol{perm\_r}$ and $\boldsymbol{perm\_c}$. *In such two-dimensional pivoting, the COO format offers a clear advantage: a single traversal of all nonzeros suffices to complete the pivoting with complexity $\mathcal{O}(\mathbf{M}.nnz)$.* In contrast, the direction-oriented layout and indirect pointer structures of CSC/CSR formats make *dimension-agnostic pivoting* (across both row and column directions) nontrivial, raising its complexity to $\mathcal{O}(\mathbf{M}.nnz^2)$.

Algorithm 2 Sparse Partial Rank-Revealing LU Decomposition (Sparse PRRLU).

Input: $\mathbf{M} \in \mathbb{R}^{m \times n}$ in COO format, maximum rank $r_{\max} \geq 0$, rank truncation tolerance $\epsilon > 0$.
Output: Revealed rank r, pivot vectors $\boldsymbol{perm\_r}$ and $\boldsymbol{perm\_c}$.

1: $r \leftarrow 0$, $\boldsymbol{perm\_r} \leftarrow [1, \ldots, m]$, $\boldsymbol{perm\_c} \leftarrow [1, \ldots, n]$
2: **for** $s = 1$ to N **do**
3: /* Phase 1: Find the pivot indices */
4: Find nonzero p, s.t. $|\mathbf{M}.val(p)| = \max\{|\mathbf{M}.val(i)| : \mathbf{M}.row(i) \in [s, m], \mathbf{M}.col(i) \in [s, n]\}$
5: $(\mu, \lambda, piv\_val) \leftarrow (\mathbf{M}.row(p), \mathbf{M}.col(p), \mathbf{M}.val(p))$
6: /* Iteration termination */
7: **if** $|piv\_val| < \epsilon$ **or** $r = \min\{r_{\max}, m, n\}$ **then**
8: break
9: **end if**
10: /* Phase 2: Row/Column pivoting */
11: $\boldsymbol{perm\_r}(s) \leftrightarrow \boldsymbol{perm\_r}(\mu)$, $\boldsymbol{perm\_c}(s) \leftrightarrow \boldsymbol{perm\_r}(\lambda)$
12: **for** $i = 1$ to $\mathbf{M}.nnz$ **do**
13: $\mathbf{M}.row(i) \leftarrow \mu$ **or** s if $\mathbf{M}.row(i) = s$ **or** μ
14: $\mathbf{M}.col(i) \leftarrow \lambda$ **or** s if $\mathbf{M}.col(i) = s$ **or** λ
15: **end for**
16: /* Phase 3: Gaussian elimination */
17: Sparse vectors $\boldsymbol{u} \leftarrow \mathbf{M}(s+1{:}m, s)$, $\boldsymbol{v} \leftarrow \mathbf{M}(s, s+1{:}n)$
18: **for** $i = 1$ to $\boldsymbol{u}.nnz$ **do**
19: **for** $j = 1$ to $\boldsymbol{v}.nnz$ **do**
20: $outer\_prod \leftarrow -\boldsymbol{u}.data(i) \times \boldsymbol{v}.data(j)/piv\_val$
21: $\mathbf{M}.addupdate(\boldsymbol{u}.idx(i), \boldsymbol{v}.idx(j), outer\_prod)$
22: **end for**
23: **end for**
24: $r \leftarrow r + 1$ ▷ Rank increment
25: **end for**
26: **return** r, $\boldsymbol{perm\_r}$, $\boldsymbol{perm\_c}$

Phase 3: Gaussian Elimination. Following the pivoting, Gaussian elimination is performed using the outer-product formulation (right-looking strategy in the classical LU decomposition [25]). We adopt the outer-product formulation for several reasons: (1) Complete pivoting is necessary to accurately reveal the rank and skeleton indices; (2) the $\mathbf{L}$ factor in PRRLU lacks an exploitable sparsity pattern, making the left-looking method [19] unsuitable; and (3) pivoting strategies (e.g., partial [16], static [30]) and optimizations such as supernodal [16] or multifrontal [15] methods, though effective in general sparse LU solvers, such as SuperLU, UMFPACK, and MUMPS [3, 15, 52], would compromise the numerical stability and rank-revealing property, thereby failing to produce valid skeletons for ID. Nevertheless, the fill-ins do *not* affect the sparsity of the ID output, as the TT-cores and residual tensors are formed from the skeleton of the input tensor (Lines 7-8 in Algorithm 1).

Two sparse vectors, $\boldsymbol{u}$ and $\boldsymbol{v}$, used to form the outer product, are extracted from $\mathbf{M}$ simultaneously in a single

traversal (Line 17). Each outer-product update is then applied to $\mathbf{M}(s{+}1{:}m,\ s{+}1{:}n)$ using the `addUpdate` function (Lines 20,21). If the target location $(\boldsymbol{u}.idx(i), \boldsymbol{v}.idx(j))$ already contains a nonzero entry, `addUpdate` accumulates the contribution; otherwise, the value is inserted as a new nonzero element with the corresponding indices. Since this process involves repeated traversals, searches, insertions, and updates in $\mathbf{M}$, Phase 3 becomes the most computationally expensive step in a PRRLU iteration, with complexity $\mathcal{O}(\boldsymbol{u}.nnz \times \boldsymbol{v}.nnz \times \mathbf{M}.nnz)$.

Although COO's coordinate-list structure supports incremental insertion, a feature absent in the static CSR/CSC, the Gaussian elimination phase poses significant computational challenges due to its intensive search overhead. Moreover, the fill-ins introduced by PRRLU can further increase the overhead of subsequent iterations. To address these challenges and improve the scalability of sparse PRRLU operations, we introduce two algorithmic optimizations together with a parallel STTID implementation on GPUs, as detailed in Sections V and VI.

V. High-Performance Optimizations for PRRLU

This section presents a high-performance implementation of the sparse PRRLU, the primary computational bottleneck in STTID. Figure 2 illustrates the three-phase workflow of a single sparse PRRLU iteration, incorporating two optimizations: ❶ selective data separation and ❷ hash table-based pivoted Gaussian elimination.

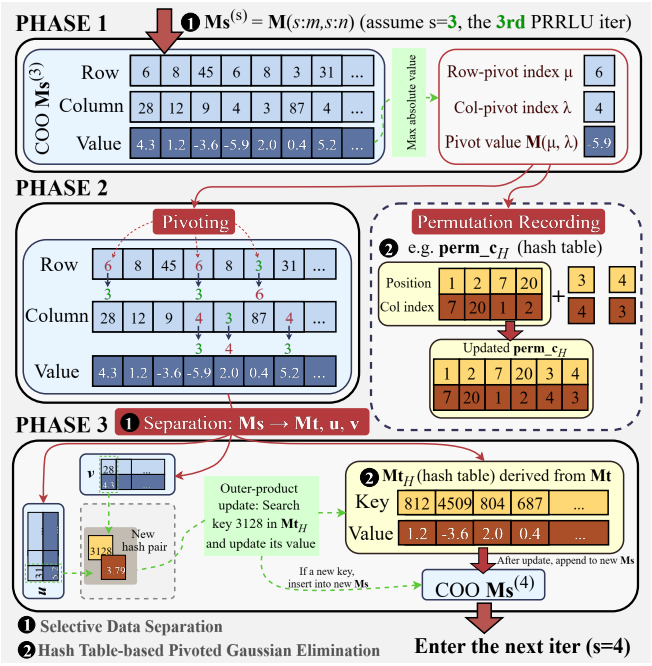

Fig. 2: Example of a sparse PRRLU iteration equipped with data separation and hash table indexing.

A. Selective Data Separation Strategy

All three phases in Algorithm 2 traverse every nonzero entry of $\mathbf{M}$ due to the unordered COO representation that appears from the incremental construction in Phase 3 from prior iteration. However, in any s-th iteration, the rank-revealing procedure only involves the submatrix $\mathbf{M}(s{:}m, , s{:}n)$. To mitigate redundant traversal, we introduce a *selective data separation strategy* (❶ in Figure 2) that avoids global scanning and reduces iteration complexity.

In each PRRLU iteration, the separation strategy extracts the submatrix as $\mathbf{Ms}$ from $\mathbf{M}$. This partitioning is performed concurrently with the pivoting in Phase 2 (red box labeled "❶ *Separation*" in Figure 2): The matrix $\mathbf{Ms}^{(s)} = \mathbf{M}(s{:}m,\ s{:}n)$, derived from the previous $(s{-}1)$-th iteration, is classified into two categories during pivoting: (1) nonzeros in $\mathbf{M}(s{+}1{:}m,\ s)$ and $\mathbf{M}(s,\ s{+}1{:}n)$, which are extracted as two sparse vectors $\boldsymbol{u}$ and $\boldsymbol{v}$ in one-dimensional COO format; (2) elements in $\mathbf{M}(s{+}1{:}m,\ s{+}1{:}n)$, stored as $\mathbf{Mt}$, which require Gaussian elimination updates. With this categorization, the `addUpdate` function now only operates on the submatrix $\mathbf{Mt}$ rather than on the entire $\mathbf{M}$. The updated $\mathbf{Mt}$ then becomes $\mathbf{Ms}^{(s+1)}$ for the next iteration. Consequently, the search and update operations of Phases 1, 2, and 3 are confined to $\mathbf{Ms}^{(s)}$ in the s-th iteration. The complexity of Phases 1 and 2 decreases from $\mathcal{O}(\mathbf{M}.nnz)$ to $\mathcal{O}(\mathbf{Ms}.nnz)$. And the overall complexity, which is dominated by Phase 3, is reduced from $\mathcal{O}(\boldsymbol{u}.nnz \times \boldsymbol{v}.nnz \times \mathbf{M}.nnz)$ to $\mathcal{O}(\boldsymbol{u}.nnz \times \boldsymbol{v}.nnz \times \mathbf{Mt}.nnz)$.

B. Hash Table–based Pivoted Gaussian Elimination

Even with the data separation strategy, Gaussian elimination in Phase 3 still dominates the overhead of sparse PRRLU because of its extensive searches and updates. Moreover, the permutation vectors $\boldsymbol{perm\_r}$ and $\boldsymbol{perm\_c}$, which store row and column pivots (initialized in Line 1 and updated in Line 11 of Algorithm 2), have sizes m and n that correspond to the reshaped matrix $\mathbf{M}$. These vectors can become extremely large after tensor reshaping. For example, reshaping a 5^{th}-order tensor with mode size 500 in the first TT-iter yields a matrix with $n{=}500^4$ columns, requiring 465 Gigabytes for the dense int64 $\boldsymbol{perm\_c}$ recording all column positions. To further mitigate these computational and storage challenges, we employ a *hash table–based pivoted Gaussian elimination*.

Hash tables are data structures offering efficient key-value storage with expected $\mathcal{O}(1)$ complexity for lookup, insertion, and deletion [60]. A hash function maps input keys to slots in the table. Collisions, where multiple keys map to the same slot, are resolved by techniques such as chaining or probing.

Gaussian Elimination. In Phase 3, each (i, j) pair from vectors $\boldsymbol{u}$ and $\boldsymbol{v}$ requires a search on the separated matrix $\mathbf{Mt}$ to locate the position of the outer-product value, *outer_prod*. If the position exists, the entry $\mathbf{Mt}(\boldsymbol{u}.idx(i), \boldsymbol{v}.idx(j))$ is updated; otherwise, a new entry with value *outer_prod* is inserted. We construct a hash table $\mathbf{Mt}_H$ (❷ in Phase 3 in Figure 2) from $\mathbf{Mt}$, which stores nonzeros as key-value pairs with keys formed by linearizing row and column indices.

The hash table $\mathbf{Mt}_H$ enables fast entry lookup for updates, reducing the complexity of Phase 3 to $\mathcal{O}(\boldsymbol{u}.nnz \times \boldsymbol{v}.nnz)$. However, when an entry is new, inserting it into the hash table incurs additional overhead, and the inserted key-value

pairs still require conversion to COO format when constructing $\mathbf{Ms}^{(s+1)}$ for the next iteration. To avoid this cost, we bypass inserting new outer products into $\mathbf{Mt}_H$ and instead write them directly to $\mathbf{Ms}^{(s+1)}$ in COO format. After completing the Gaussian elimination, the values in $\mathbf{Mt}_H$ are also appended to $\mathbf{Ms}^{(s+1)}$ to preserve the values.

Permutation Vectors. To address the storage challenge mentioned earlier, we leverage hash tables to record row and column permutations (❷ in Phase 2 of Figure 2). This approach exploits the fact that some rows and columns remain in their original positions throughout the PRRLU process, so their permutations do not need to be explicitly stored. Taking column pivoting as an example, we initialize an empty hash map $\boldsymbol{perm\_c}_H$. In the s-th PRRLU iteration, when a new pivot index λ is found, we check for the key-value pairs (s, λ) and (λ, s) in $\boldsymbol{perm\_c}_H$. If the keys are absent, they are inserted; otherwise, their values are updated to reflect the swap. Row permutations are recorded similarly to construct $\boldsymbol{perm\_r}_H$.

After PRRLU, the first r row/column indices after pivoting (as the skeleton in Line 6 of Algorithm 1) are retrieved by querying keys 1 to r in the hash table, where an absent key indicates an unpermuted index. This overhead is minimal, as r is relatively small and the lookup occurs only once per TT iteration ($d-1$ times). This hash table-based pivot recording adds at most two key-value pairs per vector in each iteration. Using the earlier example for $\mathbf{M}$ in shape $(500, 500^4)$, the final $\boldsymbol{perm\_c}_H$ contains at most 500×2 key-value pairs when PRRLU terminates (up to 500 iterations, as $m = 500$), instead of a dense array of size $n = 500^4$.

VI. Hybrid CPU-GPU STTID

As the sparse PRRLU step remains the dominant computational bottleneck in STTID even after optimization, we introduce a hybrid CPU-GPU implementation to further accelerate this step on NVIDIA GPU.

A. Data Placement

We develop a hybrid CPU-GPU STTID by offloading and parallelizing the sparse PRRLU on NVIDIA GPUs using CUDA [62]. In such hybrid systems, allocation memory must be carefully managed for all participating data objects. To minimize the GPU memory burden, we restrict GPU-resident data to the PRRLU-processed submatrix $\mathbf{Ms}$, along with the intermediate hash table. The remaining objects, such as the original tensor, output permutation vectors, and TT-cores, are managed in CPU memory. $\mathbf{M}$ (the $\mathbf{Ms}$ of the 1^{st} PRRLU iteration) is transferred from CPU to GPU and processed by the parallel sparse PRRLU at each TT iteration. PRRLU outputs the new pivot indices (μ, λ) to $\boldsymbol{perm\_r}_H$ and $\boldsymbol{perm\_c}_H$ on the host via asynchronous data transfers, as detailed in Section VI-B. With the permutation vectors on the host, STTID assembles the outputs $\mathbf{G}_k$ and $\mathbf{X}_k$, together with the residual tensor for the next TT iteration.

B. CPU-GPU Asynchronous Pivoting

With data separation, pivot entry identification (Phase 1) is efficiently performed using the `cublasIdamax_64` routine from the cuBLAS library [67] to find the entry of $\mathbf{Ms}$ with the maximum absolute value. The located position (μ, λ) is used for row and column pivoting on the device. At the same time, it is copied to the host to update the permutation vectors, enabling an asynchronous pivoting process.

On the GPU, data parallelism is employed by assigning each CUDA thread to independently pivot a single nonzero entry. Each thread independently executes coordinate permutation on the COO row and column arrays, along with data separation. Meanwhile, $\boldsymbol{perm\_r}_H$ and $\boldsymbol{perm\_c}_H$ are updated on the CPU following the procedure described in Section V-B. This update is performed on the host because (i) the operations are lightweight and inherently serial, involving only a few conditional if-else statements on hash tables, which makes them inefficient on the GPU, and (ii) the permutation vectors are required on the host for output as skeleton indices.

C. On-Device Hash Table

In Phase 3, we implement parallel operations on the hash table constructed using NVIDIA's cuCollections library [12], which provides on-device hash tables supporting high-throughput concurrent operations. We employ the highest-performing implementation in cuCollections[1], `cuco::static_map`, to construct the hash table $\mathbf{Mt}_H$ on the device. $\mathbf{Mt}_H$ is a fixed-capacity hash table using linear probing, with its capacity initialized to the number of nonzeros in $\mathbf{Mt}$. The implementation maintains identical key-value pair representations to its CPU counterpart described in Section V-B.

The $\mathbf{Mt}_H$ implemented using cuCollections demonstrates limitations not only due to the entry insertion overhead discussed in Section V-B but because of a concurrency issue: The current on-device operations in `cuco::static_map` do not support updating an existing key-value pair while simultaneously inserting a new one, causing atomicity violations under concurrent access. To address this, threads invoke the *fetch_add* method of *cuco::find* (on-device lookup reference) in our device CUDA kernel solely to update existing key-value pairs, while new pairs are directly decoded into coordinate entries and appended to $\mathbf{Ms}$. This approach eliminates both the atomicity problem and the overhead associated with inserting and extracting pairs in $\mathbf{Mt}_H$.

D. Kernel Fusion and Shared Memory Optimization.

To further optimize performance, we employ kernel fusion to combine Phases 2 and 3. The fused kernel avoids materializing the intermediate tensor $\mathbf{Mt}$ by having threads directly insert the entries into $\mathbf{Mt}_H$ after pivoting and separating data objects, thereby reducing memory footprints. The computation of $outer\_prod$ is fully independent across CUDA threads,

[1] `multi_map` and `dynamic_map` incur higher overhead and remain under active development (e.g., `dynamic_map` currently only supports host-bulk APIs) in cuCollection (commit e3aec27).

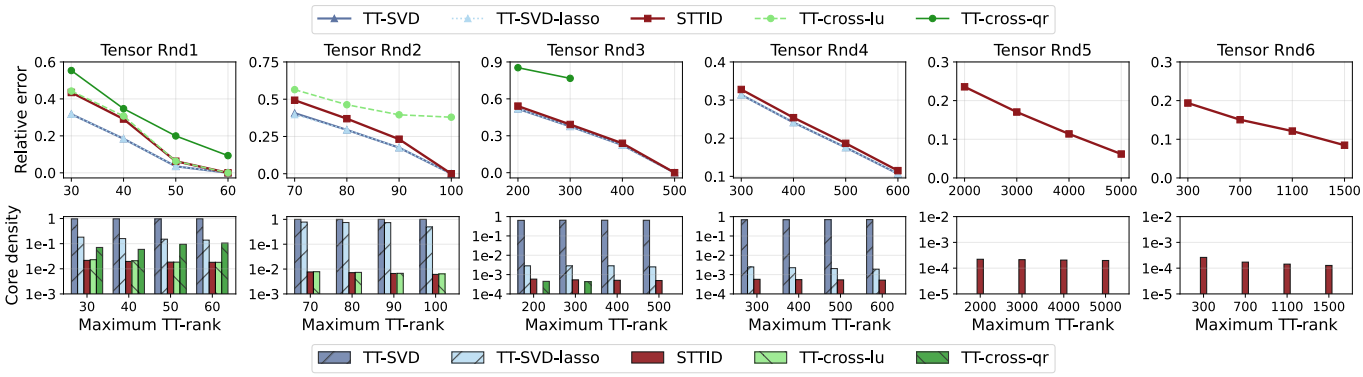

Fig. 3: Reconstruction error (upper panel) and average TT-core density (lower panel) versus maximum TT-rank ($r_{\max}$) for TT-SVD, TT-SVD-lasso, TT-cross-qr, TT-cross-lu, and STTID on six tensors Rnd1-Rnd6. Missing data points for some algorithms indicate that they failed to complete for various reasons (see Section VII-B for details).

where one-dimensional thread blocks are assigned, and each thread maps to a pair of elements in $\boldsymbol{u}$ and $\boldsymbol{v}$ using its global thread index via quotient and remainder operations. Updates to the hash table $\mathbf{Mt}_H$ are protected using *cuda::atomic_ref*, while insertions into the COO-formatted $\mathbf{Ms}^{(s+1)}$ are not. We employ atomic operations to ensure safe concurrent append operations to $\mathbf{Ms}^{(s+1)}$. These data structures — $\boldsymbol{u}$, $\boldsymbol{v}$, $\mathbf{Mt}_H$, and $\mathbf{Ms}^{(s+1)}$ — reside in global memory due to their large sizes and limited data reuse. However, atomic operations on global memory are expensive. To mitigate this overhead, we leverage shared memory to localize atomic operations and subsequently update global memory after synchronization, thereby improving the efficiency of insertions.

VII. Evaluation

A. Experiment Setup

Experiment Platforms. All CPU-based experiments are conducted on a Linux server (Ubuntu 22.04), equipped with an AMD Ryzen Threadripper PRO 3975WX processor (32 cores) and 503 GB of memory. GPU experiments are conducted on two different NVIDIA GPUs: NVIDIA RTX A5000 (Ampere architecture, 8,192 CUDA cores, 24 GB GDDR6 memory, 768 GB/s memory bandwidth, 27.8 TFLOPS (FP32) peak performance); NVIDIA H100 NVL (Hopper architecture, 16,896 CUDA cores, 94 GB HBM2e memory, 3.9 TB/s bandwidth, 60 TFLOPS (FP32) peak performance).

Software Environments. Our program uses double-precision floating-point values and is compiled with g++ 9.4.0 and NVCC 12.0. On the CPU, the hash table is implemented using the C++ STL `unordered_map` [61], with `std::hash` as the hash function and separate chaining for collision resolution. On the GPU, we use 512 CUDA threads per block, and the on-device hash table is implemented via `static_map` in cuCollections (commit e3aec27) with linear probing to resolve collisions.

Experimental Data. We evaluate our algorithm on random and real-world tensors, whose characteristics (e.g., modes, density) are summarized in Table I. *Random tensors:* Rnd1–Rnd6 are generated from a power-law distribution [45, 47], reflecting the heavy-tailed phenomena commonly observed in graph and social network data. *Real-world tensors:* We additionally test tensors collected from real applications. WikiPeople-3, WikiPeople-4, JF17K-3, and JF17K-4 are widely used knowledge graphs that represent n-ary relations ($n = 3, 4$) among entities [57]. We evaluate four 4^{th}- and 5^{th}-order tensors from the FROSTT dataset [90] to benchmark the performance of our method: Uber pickup data (Uber, "latitude-longitude-date-hour"), NIPS publication data (NIPS, "paper-author-word-year"), and two Chicago crime datasets (CC-comm and CC-geo, "day-hour-community-crimetype" and "day-hour-latitude-longitude-crimetype"). [2]

Tensor	Modes	NNZs	Density
Rnd1	$(10, 10, 10, 10)$	91	9.10×10^{-3}
Rnd2	$(10, 10, 10, 10, 10)$	428	4.28×10^{-3}
Rnd3	$(50, 50, 50, 50)$	576	9.22×10^{-5}
Rnd4	$(50, 50, 50, 50)$	893	1.42×10^{-4}
Rnd5	$(100, 100, 100, 100)$	12,729	1.27×10^{-4}
Rnd6	$(50, 50, 50, 50, 50)$	27,843	8.91×10^{-5}
Wiki3	$(66, 12270, 12270, 12270)$	25,820	2.1×10^{-10}
Wiki4	$(50, 9528, 9528, 9528, 9528)$	15,188	3.7×10^{-14}
JF17K3	$(104, 11541, 11541, 11541)$	34,544	2.2×10^{-10}
JF17K4	$(23, 6536, 6536, 6536, 6536)$	9,509	2.3×10^{-13}
Uber	$(183, 24, 1140, 1717)$	3,309,490	3.9×10^{-4}
NIPS	$(2482, 2862, 14036, 17)$	3,101,609	1.8×10^{-6}
CC-comm	$(6186, 24, 77, 32)$	5,330,673	1.5×10^{-2}
CC-geo	$(6185, 24, 380, 395, 32)$	6,327,013	8.9×10^{-6}

TABLE I: Evaluation datasets: random tensors (top panel) and real-world tensors from knowledge graphs (middle panel) and FROSTT tensor collection (bottom panel).

B. Quality Comparison

Figure 3 demonstrates the superior capability of STTID in stably decomposing diverse sparse tensors (Rnd1-Rnd6) while preserving TT sparsity and maintaining error convergence as maximum TT-ranks increase, outperforming four state-of-the-art baselines that are limited in handling sparse data. The first two are SVD-based methods: TT-SVD and TT-SVD with ℓ^1-regularized TT-cores (TT-SVD-lasso), implemented by TensorLy's functionality [44]; the other two are TT-cross

[2] While STTID can process relatively large tensors, tensors with an excessive number of nonzeros, such as Flickr (112,890,310 nonzeros) from FROSTT, can exceed GPU memory capacity during runtime.

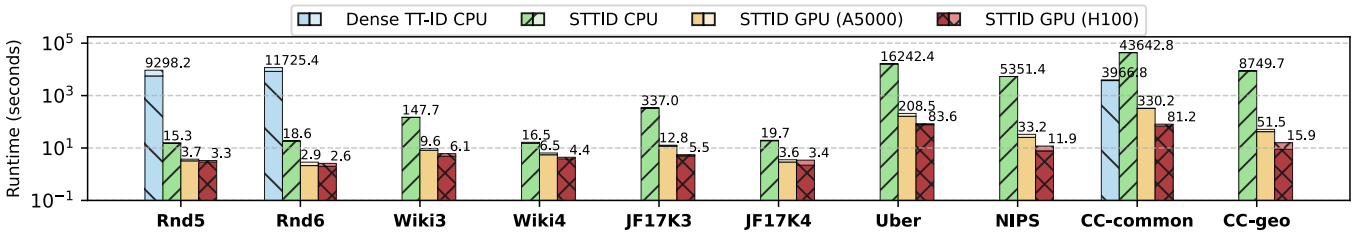

Fig. 4: Overall runtime (in seconds) of the dense TT-ID on CPU, STTID on CPU, and the hybrid CPU-GPU STTID on A5000 and H100 GPUs. The color gradient of the stacked bars represents the runtime of PRRLU (dark/bottom, accounts for over 95% of runtime) and other operations (bright/top).

methods: the baseline version *TT-cross-qr* [18, 70] (available in TensorLy) and the variant *TT-cross-lu* [66] implemented in Python. Two metrics are used in Figure 3 to assess the quality of TT decomposition of sparse tensors: (1) reconstruction error and (2) average core density.

- Reconstruction error: We evaluate TT accuracy by the reconstruction error, defined as the relative Frobenius-norm difference $E_r = \|\mathcal{T} - \hat{\mathcal{T}}\|_F / \|\mathcal{T}\|_F$, which quantifies how closely the TT-approximation $\hat{\mathcal{T}}$ reproduces the original $\mathcal{T}$.
- Average core density: The fraction of nonzeros in the TT-cores, computed as total nonzeros divided by the sum of core sizes, measuring the sparsity of a TT decomposition.

TT-SVD Methods. Due to SVD's closest low-rank approximation [20], TT-SVD can achieve the minimal reconstruction error, but its orthogonal TT-cores are typically dense. By tuning the ℓ^1 regularization to introduce zeros in these TT-cores without obviously degrading reconstruction accuracy, TT-SVD-lasso achieves up to two orders of magnitude higher sparsity on tensors Rnd3 and Rnd4. Such highly sparse inputs with abundance of near-zero entries result in many negligible values in TT-cores to be zeroed by regularization. Nevertheless, this method has two drawbacks: (i) It is not robust and is highly input-dependent (failing to introduce much sparsity in the TT-cores of Rnd1 and Rnd2). Moreover, altering TT-cores destroys the orthogonality guaranteed by TT-SVD and may amplify undesired approximation errors, requiring careful selection of the regularization strength. (ii) Being built on TT-SVD, it does not support sparse computation on compressed storage, leading to scalability issues like out-of-memory for larger tensors Rnd5 and Rnd6.

TT-cross Methods. Limited by the partial-evaluation scheme described in Section III-C, both TT-cross implementations struggle to decompose sparse tensors. They succeed only on small cases, Rnd1 and Rnd2, producing sparse TT-cores but with the highest errors. For larger tensors, the truncated green line in Figure 3 indicate frequent runtime failures, arising either from singular matrix inversion (TT-cross-qr) or from non-convergence caused by repeated iterations without updating new interpolation skeletons (TT-cross-lu). Thus, while both methods can preserve sparsity in TT-cores when successful, they remain unstable on sparse tensors and scale poorly.

STTID. Compared with baselines achieving similar reconstruction error at the same maximum TT-ranks, STTID achieves the sparsest TT-cores across different tensors. Because TT-cores are small in size, their average density is typically higher than that of the input tensor, although they remain highly sparse across all tested tensors. While limited by the approximation capability of ID, STTID sacrifices a small amount of accuracy compared with SVD-based methods, yet it can still converge to the exact decomposition, revealing uncompressed TT-ranks of tensors (e.g., Rnd1: $r_{\max} = 60$, $E_r = 1.7\times10^{-16}$; Rnd2: $r_{\max} = 100$, $E_r = 2.4\times10^{-16}$). The small accuracy trade-off yields not only significantly sparser TT-cores but also improved computational efficiency. As tensor size increases (Rnd5 and Rnd6), STTID becomes the only practical method of decomposing the tensors, as all other baselines fail for various reasons.

C. Performance Evaluation

Figure 4 compares the runtimes of the three implementations, showing STTID's superior scalability on highly sparse tensors and the significant speedup of the GPU implementation over the CPU version on large datasets. The three implementations are: (1) the dense TT-ID, (2) STTID on CPU with the optimizations from Section V, and (3) its GPU-accelerated counterpart described in Section VI on two different GPUs.

The evaluation is conducted on the two largest random tensors (Rnd5 and Rnd6) and eight real-world tensors. For Rnd5 and Rnd6, we set $r_{\max} = 5000$ and $r_{\max} = 1500$ for all implementations, respectively, matching the last maximum TT-ranks in Figure 3. For the four knowledge-graph tensors, exact decomposition is performed with no rank truncation. For the four FROSTT tensors, their high nonzero counts make CPU execution less efficient, so we limit $r_{\max} = 500$ to keep CPU experiments tractable.

Dense TT-ID vs. STTID. Without compressed storage, dense TT-ID can barely handle large tensors. For instance, in-place operations on the double-precision array of Wiki3 would require nearly one million gigabytes of memory. Therefore, dense TT-ID is limited to small-mode tensors such as Rnd5, Rnd6, and CC-comm. The CC-common test, with relatively small mode sizes but an enormous number of nonzeros, marks the boundary of STTID's effectiveness: its high density diminishes the benefits of sparse computation. As a result, STTID on CPU runs even slower than dense TT-ID. For highly sparse data, STTID achieves substantial speedups over dense TT-ID, running $607.7\times$ and $630.4\times$ faster on Rnd5 and Rnd6, respectively.

CPU vs. GPU STTID. For tensors with moderate nonzero counts, such as the knowledge-graph tensors, STTID can efficiently perform exact decomposition. The revealed uncompressed TT-ranks (Wiki3: $(66, 7599, 2949)$, Wiki4: $(50, 3664, 3878, 1630)$, JF17K3: $(104, 2472, 3237)$, JF17K4: $(23, 1498, 2126, 1124)$) demonstrate strong compression capability of TT, as the rank dimensions are significantly smaller than the original tensor dimensions in Table I. For the four FROSTT tensors with large numbers of nonzeros, STTID on CPU is less efficient. The hybrid CPU-GPU implementation delivers significant performance gains as the problem size grows. For these tensors with millions of nonzeros, STTID achieves substantial speedups on GPU over CPU, ranging from $77.9\times$ to $169.9\times$ on NVIDIA A5000 and $194.3\times$ to $550.3\times$ on NVIDIA H100. The efficiency of STTID GPU enables the exact decomposition and the revelation of untruncated TT-ranks of FROSTT tensors, such as NIPS (TT-rank: $2482, 2481, 17$).

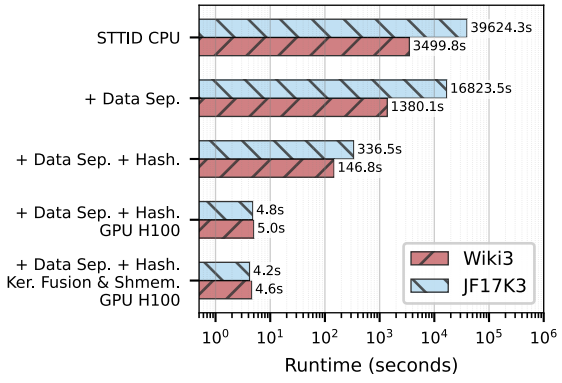

Fig. 5: Runtime comparison of PRRLU at various optimization stages on Wiki3 and JF17K3.

D. Optimization Analysis

Ablation Study. PRRLU accounts for the majority of the runtime in STTID (over 95%, as shown in Figure 4 in dark/bottom stacked bars), highlighting the importance of our optimizations and parallelization efforts focused on it. Figure 5 outlines the runtime of different optimized versions of PRRLU in STTID for the exact decomposition of Wiki3 and JF17K3, demonstrating the effectiveness of each optimization. Our algorithmic optimizations generally yield higher performance gains than low-level GPU optimizations. Data separation reduces runtime by over 50% on the CPU compared to the non-optimized STTID. Furthermore, via its significant complexity reduction, the hash table method delivers speedups of $9.4\times$ and $50.0\times$ over data separation for Wiki3 and JF17K3, respectively. The most significant enhancement comes from the GPU implementation, which provides speedups of $29.4\times$ and $70.1\times$ over optimized CPU version. Finally, kernel fusion with shared memory contributes an additional performance improvement of approximately 10% .

Fill-in Analysis. Figure 6 shows the trend of nonzeros to be processed in PRRLU throughout the TT decomposition.

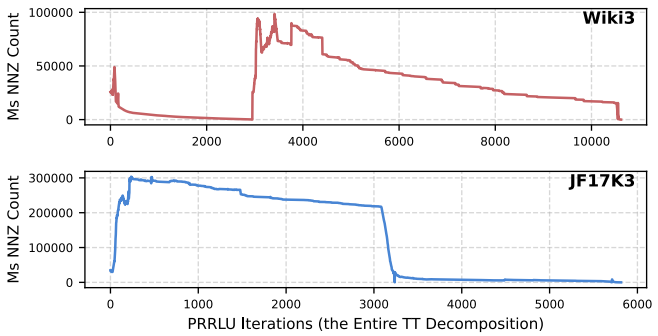

Fig. 6: Nonzero count of $\mathbf{Ms}$ processed by PRRLU during the entire TT decomposition on Wiki3 and JF17K3.

Despite the fill-ins introduced by the rank-revealing process, the data separation strategy (Section V-A) effectively prevents continuous problem-size growth during the iterations on highly sparse tensors, as exemplified by Wiki3 and JF17K3. The number of nonzeros in $\mathbf{Ms}$ does not increase monotonically, as its size gradually decreases during decomposition. However, the behavior is highly data dependent; for JF17K3, a considerable amount of fill-ins still occurs.

The heuristic rank-revealing process can increase the number of nonzeros processed and thus degrade performance. These results highlight the need for sparse rank-revealing methods that identify high-quality skeletons while minimizing fill-ins. Although prior work has recognized this issue [29], practical libraries for sparse and parallel rank-revealing methods tailored to interpolative decomposition remain unavailable—a gap that STTID begins to address. As future work, we plan to incorporate additional structural optimizations, such as block-based strategies, to further reduce fill-ins and improve the scalability of STTID.

E. Case Study: Downstream Computation

After TT decomposition, downstream computations are often needed on the TT-cores, such as single-core contraction, or entire TT reconstruction for predicting missing values in tensor completion tasks or quality measurement [57, 96]. As the reverse of decomposition, TT reconstruction is obtained by inverting the expression in Equation (1) (or Equation (2) for TCI form) as the entire contraction of TT. The TCI-form TT introduces additional evaluation of $\mathcal{G}_k\mathbf{X}_k^{-1}$ during reconstruction. However, as discussed, this can be computed efficiently without explicitly storing or inverting $\mathbf{X}_k$ [65, 66]:

Reshaping the core $\mathcal{G}_k(i_{k-1}, u_k, j_k) = \mathcal{T}(\mathcal{I}_{k-1}, u_k, \mathcal{J}_k)$ to a matrix $\mathbf{G}_k(i_{k-1}\times u_k, j_k)$, the cross $\mathbf{X}_k(i_k, j_k) = \mathcal{T}(\mathcal{I}_k, \mathcal{J}_k)$ is the subset of $\mathbf{G}_k$ ($\mathcal{I}_k \in \mathcal{I}_{k-1} \times u_k$). So $\mathbf{G}_k\mathbf{X}_k^{-1}$ contains an identity matrix at the rows forming $\mathbf{X}_k$. For the remaining rows of $\mathbf{G}_k$, denoted as $\mathbf{G}'_k$, $\mathbf{G}'_k\mathbf{X}_k^{-1}$ can be computed using the LU factors of $\mathbf{G}_k$:

$$\mathbf{G}_k = \begin{pmatrix}\mathbf{X}_k\\ \mathbf{G}'_k\end{pmatrix} = \begin{pmatrix}\mathbf{L}\\ \mathbf{L}'\end{pmatrix}\mathbf{U}, \quad \mathbf{G}_k\mathbf{X}_k^{-1} = \begin{pmatrix}\mathbf{I}\\ \mathbf{L}'\mathbf{L}^{-1}\end{pmatrix}. \tag{5}$$

By solving the lower-triangular system $\mathbf{L}'\mathbf{L}^{-1}$, one can compute $\mathbf{G}_k\mathbf{X}_k^{-1}$, which can then be reshaped back into a 3D TT-core, recovering the TT in Equation (1).

Tensor	Reconstruction time of TT (seconds)		Speedup
	Dense TT	Sparse TT (TCI form)	
Rnd5	47.3 s	3.9 s	12.1×
Rnd6	160.9 s	5.4 s	29.8×

TABLE II: Reconstruction time for dense vs. sparse TT-cores.

Despite the above extra steps, STTID's sparse TCI-form TT-cores largely outperforms dense TT-cores in reconstruction, as shown in Table II. Using the sparse TT of Rnd5 and Rnd6 produced by STTID, we measure the CPU runtime of (1) sparse reconstruction, i.e., solving $\boldsymbol{\mathcal{G}}_k\mathbf{X}_k^{-1}$ via the Intel MKL Direct Sparse Solver and performing sparse tensor contractions using Sparta [55]; and (2) dense reconstruction, i.e., treating TT-cores (after $\boldsymbol{\mathcal{G}}_k\mathbf{X}_k^{-1}$) as dense arrays and contracting them via the dense tensor library TBLIS [59]. Sparse TT reconstruction achieves speedups of $12.1\times$ and $29.8\times$ compared with dense computation.

VIII. Conclusion and Future Work

Motivated by the challenge of sparsity preserving and computational scalability in TT decomposition of sparse tensors, we present STTID, a novel sparse tensor-train algorithm. STTID demonstrates robust capability in processing sparse tensors and generating sparse TT-cores. With a particular focus on accelerating the sparse partial rank-revealing LU decomposition through multiple optimizations and GPU parallelization, STTID enables efficient TT decomposition of large-scale sparse tensors from real applications. Our work lays the foundation for post-decomposition operations on sparse tensors, including computations on sparse TT-cores in downstream applications and enhanced interpretability of sparse data.

The source code of STTID is publicly available at https://github.com/tensorworld/STTID.git to facilitate reproducibility and further research. As future work, we plan to further improve STTID by developing strategies to reduce fill-ins during sparse PRRLU. We hope this work will inspire sparsity-aware designs for other tensor networks and low-rank approximation algorithms, benefiting practical applications such as tensor completion and tensorized neural networks.

Acknowledgment

We thank all the anonymous reviewers for their valuable comments. This work was supported in part by US National Science Foundation under CCF-2316201 and CCF-2504512. The polishing of this manuscript was assisted by Google Gemini [27] and OpenAI ChatGPT [68].

References

[1] Talal Ahmed, Haroon Raja, and Waheed U Bajwa. 2020. Tensor regression using low-rank and sparse Tucker decompositions. *SIAM Journal on Mathematics of Data Science* 2, 4 (2020), 944–966.

[2] Hussam Al Daas, Grey Ballard, Paul Cazeaux, Eric Hallman, Agnieszka Międlar, Mirjeta Pasha, Tim W Reid, and Arvind K Saibaba. 2023. Randomized Algorithms for Rounding in the Tensor-Train Format. *SIAM Journal on Scientific Computing* 45, 1 (2023), A74–A95. doi:10.1137/21M1451191

[3] P.R. Amestoy, I. S. Duff, J. Koster, and J.-Y. L'Excellent. 2001. A Fully Asynchronous Multifrontal Solver Using Distributed Dynamic Scheduling. *SIAM J. Matrix Anal. Appl.* 23, 1 (2001), 15–41.

[4] Baptiste Anselme Martin, Thomas Ayral, François Jamet, Marko J Rančić, and Pascal Simon. 2024. Combining matrix product states and noisy quantum computers for quantum simulation. *Physical Review A* 109, 6 (2024), 062437.

[5] Brett W Bader and Tamara G Kolda. 2008. Efficient MATLAB computations with sparse and factored tensors. *SIAM Journal on Scientific Computing* 30, 1 (2008), 205–231.

[6] Muthu Baskaran, Benoît Meister, Nicolas Vasilache, and Richard Lethin. 2012. Efficient and scalable computations with sparse tensors. In *High Performance Extreme Computing (HPEC), 2012 IEEE Conference on*. 1–6. doi:10.1109/HPEC.2012.6408676

[7] Justus A. Calvin and Edward F. Valeev. 2015. Task-based algorithm for matrix multiplication: A step towards block-sparse tensor computing. *arXiv preprint arXiv:1504.05046* (2015).

[8] Venkatesan T. Chakaravarthy, Jee W. Choi, Douglas J. Joseph, Prakash Murali, Shivmaran S. Pandian, Yogish Sabharwal, and Dheeraj Sreedhar. 2018. On Optimizing Distributed Tucker Decomposition for Sparse Tensors. In *Proceedings of the 32nd ACM International Conference on Supercomputing (ICS '18)*. 10 pages.

[9] Jiawei Chiu and Laurent Demanet. 2013. Sublinear randomized algorithms for skeleton decompositions. *SIAM J. Matrix Anal. Appl.* 34, 3 (2013), 1361–1383.

[10] Andrzej Cichocki, Namgil Lee, Ivan Oseledets, Anh-Huy Phan, Qibin Zhao, Danilo P Mandic, et al. 2016. Tensor networks for dimensionality reduction and large-scale optimization: Part 1 low-rank tensor decompositions. *Foundations and Trends® in Machine Learning* 9, 4-5 (2016), 249–429.

[11] Andrzej Cichocki, Anh-Huy Phan, Qibin Zhao, Namgil Lee, Ivan Oseledets, Masashi Sugiyama, Danilo P Mandic, et al. 2017. Tensor networks for dimensionality reduction and large-scale optimization: Part 2 applications and future perspectives. *Foundations and Trends® in Machine Learning* 9, 6 (2017), 431–673.

[12] NVIDIA Corporation. 2025. cuCollections (Commit e3aec27). Available from https://github.com/NVIDIA/cuCollections.

[13] Hussam Al Daas, Grey Ballard, and Peter Benner. 2022. Parallel Algorithms for Tensor Train Arithmetic. *SIAM*

Journal on Scientific Computing 44, 1 (2022), C25–C53. doi:10.1137/20M1387158

[14] Arnak S. Dalalyan, Mohamed Hebiri, and Johannes Lederer. 2017. On the prediction performance of the Lasso. *Bernoulli* 23, 1 (2 2017). doi:10.3150/15-BEJ756

[15] Timothy A. Davis. 2004. Algorithm 832: UMFPACK V4.3—an unsymmetric-pattern multifrontal method. *ACM Trans. Math. Software* 30, 2 (6 2004), 196–199. doi:10.1145/992200.992206

[16] James W Demmel, Stanley C Eisenstat, John R Gilbert, Xiaoye S Li, and Joseph W H Liu. 1999. A Supernodal Approach to Sparse Partial Pivoting. *SIAM J. Matrix Anal. Appl.* 20, 3 (1999), 720–755. doi:10.1137/S0895479895291765

[17] Sergey Dolgov and Dmitry Savostyanov. 2019. Parallel cross interpolation for high-precision calculation of high-dimensional integrals. (3 2019). doi:10.1016/j.cpc.2019.106869

[18] Sergey Dolgov and Robert Scheichl. 2019. A Hybrid Alternating Least Squares–TT-Cross Algorithm for Parametric PDEs. *SIAM/ASA Journal on Uncertainty Quantification* 7, 1 (2019), 260–291. doi:10.1137/17M1138881

[19] J J Dongarra, S Hammarling, and D W Walker. 1998. Key concepts for parallel out-of-core LU factorization. *Computers & Mathematics with Applications* 35, 7 (1998), 13–31. doi:10.1016/S0898-1221(98)00029-7

[20] Carl Eckart and Gale Young. 1936. The Approximation of One Matrix by Another of Lower Rank. *Psychometrika* 1, 3 (9 1936), 211–218. doi:10.1007/BF02288367

[21] Sofia Fernandes, Hadi Fanaee-T, and João Gama. 2021. Tensor decomposition for analysing time-evolving social networks: an overview. *Artificial Intelligence Review* 54, 4 (4 2021), 2891–2916. doi:10.1007/s10462-020-09916-4

[22] Matthew Fishman, Steven R. White, and E. Miles Stoudenmire. 2022. The ITensor Software Library for Tensor Network Calculations. *SciPost Phys. Codebases* (2022), 4. doi:10.21468/SciPostPhysCodeb.4

[23] Mark Fornace and Michael Lindsey. 2024. Column and row subset selection using nuclear scores: algorithms and theory for Nyström approximation, CUR decomposition, and graph Laplacian reduction. (8 2024). http://arxiv.org/abs/2407.01698

[24] Evgeny Frolov and Ivan Oseledets. 2017. Tensor methods and recommender systems. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 7, 3 (2017), e1201.

[25] Gene H. Golub. 1965. Numerical methods for solving linear least squares problems. *Numer. Math.* 7, 3 (6 1965), 206–216. doi:10.1007/BF01436075

[26] Gene H. Golub and Charles F. Van Loan. 2013. *Matrix Computations - 4th Edition*. Johns Hopkins University Press, Philadelphia, PA. doi:10.1137/1.9781421407944

[27] Google. 2025. Gemini. Large language model. https://deepmind.google/technologies/gemini/ Accessed 2025.

[28] Sergei A Goreinov, Nikolai Leonidovich Zamarashkin, and Evgenii Evgen'evich Tyrtyshnikov. 1997. Pseudo-skeleton approximations by matrices of maximal volume. *Mathematical Notes* 62, 4 (1997), 515–519.

[29] Laura Grigori, Sebastien Cayrols, and James W Demmel. 2018. Low rank approximation of a sparse matrix based on LU factorization with column and row tournament pivoting. *SIAM Journal on Scientific Computing* 40, 2 (2018), C181–C209.

[30] Laura Grigori, James W. Demmel, and Xiaoye S. Li. 2007. Parallel Symbolic Factorization for Sparse LU with Static Pivoting. *SIAM Journal on Scientific Computing* 29, 3 (1 2007), 1289–1314. doi:10.1137/050638102

[31] Ming Gu and Stanley C. Eisenstat. 1996. Efficient Algorithms for Computing a Strong Rank-Revealing QR Factorization. *SIAM Journal on Scientific Computing* 17, 4 (7 1996), 848–869. doi:10.1137/0917055

[32] Ahmed E. Helal, Jan Laukemann, Fabio Checconi, Jesmin Jahan Tithi, Teresa Ranadive, Fabrizio Petrini, and Jeewhan Choi. 2021. Alto: Adaptive linearized storage of sparse tensors. In *Proceedings of the 35th ACM International Conference on Supercomputing*. 404–416.

[33] Thomas Hérault, Yves Robert, George Bosilca, Robert Harrison, Cannada Lewis, and Edward Valeev. 2020. *Distributed-memory multi-GPU block-sparse tensor contraction for electronic structure*. Ph. D. Dissertation. Inria-Research Centre Grenoble–Rhône-Alpes.

[34] Y. P. Hong and C.-T. Pan. 1992. Rank-Revealing QR Factorizations and the Singular Value Decomposition. *Math. Comp.* 58, 197 (1992), 213–232. doi:10.1090/S0025-5718-1992-1106970-4

[35] Rong Hu, Haotian Wang, Wangdong Yang, Renqiu Ouyang, Keqin Li, and Kenli Li. 2024. BCB-SpTC: An Efficient Sparse High-Dimensional Tensor Contraction Employing Tensor Core Acceleration. *IEEE Transactions on Parallel and Distributed Systems* (2024).

[36] Benjamin Huber, Reinhold Schneider, and Sebastian Wolf. 2017. A Randomized Tensor Train Singular Value Decomposition. (10 2017).

[37] Borbála Hunyadi, Patrick Dupont, Wim Van Paesschen, and Sabine Van Huffel. 2017. Tensor decompositions and data fusion in epileptic electroencephalography and functional magnetic resonance imaging data. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 7, 1 (2017), e1197.

[38] Tsung-Min Hwang, Wen-Wei Lin, and Eugene K. Yang. 1992. Rank revealing LU factorizations. *Linear Algebra Appl.* 175 (10 1992), 115–141. doi:10.1016/0024-3795(92)90305-T

[39] Jiahua Jiang, Fatoumata Sanogo, and Carmeliza Navasca. 2022. Low-CP-Rank Tensor Completion via Practical Regularization. *Journal of Scientific Computing* 91, 1 (4 2022), 18. doi:10.1007/s10915-022-01789-9

[40] N. S. Kapralov, A. Yu. Morozov, and S. P. Nikulin. 2023. Parallel Approximation of Multidimensional Tensors Using GPUs. *Programming and Computer Software* 49, 4 (8 2023), 295–301. doi:10.1134/S0361768823040060

[41] Oguz Kaya and Bora Uçar. 2018. Parallel Candecomp/Parafac Decomposition of Sparse Tensors Using Dimension Trees. *SIAM Journal on Scientific Computing* 40, 1 (2018), C99–C130. doi:10.1137/16M1102744 arXiv:https://doi.org/10.1137/16M1102744

[42] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The Tensor Algebra Compiler. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 77 (Oct. 2017), 29 pages. doi:10.1145/3133901

[43] Tamara G Kolda and Brett W Bader. 2009. Tensor decompositions and applications. *SIAM review* 51, 3 (2009), 455–500.

[44] Jean Kossaifi, Yannis Panagakis, Anima Anandkumar, and Maja Pantic. 2019. TensorLy: Tensor Learning in Python. *Journal of Machine Learning Research* 20, 26 (2019), 1–6. http://jmlr.org/papers/v20/18-277.html

[45] Jure Leskovec and Rok Sosič. 2016. Snap: A general-purpose network analysis and graph-mining library. *ACM Transactions on Intelligent Systems and Technology (TIST)* 8, 1 (2016), 1–20.

[46] Jiajia Li, Jee Choi, Ioakeim Perros, Jimeng Sun, and Richard Vuduc. 2017. Model-driven sparse CP decomposition for higher-order tensors. In *2017 IEEE international parallel and distributed processing symposium (IPDPS)*. IEEE, 1048–1057.

[47] Jiajia Li, Yuchen Ma, Xiaolong Wu, Ang Li, and Kevin Barker. 2019. PASTA: a parallel sparse tensor algorithm benchmark suite. *CCF Transactions on High Performance Computing* 1 (2019), 111–130.

[48] Jiajia Li, Yuchen Ma, Chenggang Yan, and Richard Vuduc. 2016. Optimizing sparse tensor times matrix on multi-core and many-core architectures. In *2016 6th Workshop on Irregular Applications: Architecture and Algorithms (IA3)*. IEEE, 26–33.

[49] Jiajia Li, Jimeng Sun, and Richard Vuduc. 2018. HiCOO: Hierarchical storage of sparse tensors. In *Proceedings of the ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*. Dallas, TX, USA.

[50] Jiajia Li, Guangming Tan, Mingyu Chen, and Ninghui Sun. 2013. 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*. 117–126.

[51] Lingjie Li, Wenjian Yu, and Kim Batselier. 2022. Faster tensor train decomposition for sparse data. *J. Comput. Appl. Math.* 405 (5 2022), 113972. doi:10.1016/j.cam.2021.113972

[52] Xiaoye S. Li. 2005. An overview of SuperLU. *ACM Trans. Math. Software* 31, 3 (9 2005), 302–325. doi:10.1145/1089014.1089017

[53] Edo Liberty, Franco Woolfe, Per-Gunnar Martinsson, Vladimir Rokhlin, and Mark Tygert. 2007. Randomized algorithms for the low-rank approximation of matrices. *Proceedings of the National Academy of Sciences* 104, 51 (12 2007), 20167–20172. doi:10.1073/pnas.0709640104

[54] Bangtian Liu, Chengyao Wen, Anand D Sarwate, and Maryam Mehri Dehnavi. 2017. A unified optimization approach for sparse tensor operations on GPUs. In *2017 IEEE international conference on cluster computing (CLUSTER)*. IEEE, 47–57.

[55] Jiawen Liu, Jie Ren, Roberto Gioiosa, Dong Li, and Jiajia Li. 2021. Sparta: High-performance, element-wise sparse tensor contraction on heterogeneous memory. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 318–333.

[56] Xiao-Yang Liu, Hao Hong, Zeliang Zhang, Weiqin Tong, Jean Kossaifi, Xiaodong Wang, and Anwar Walid. 2024. High-Performance Tensor-Train Primitives Using GPU Tensor Cores. *IEEE Trans. Comput.* 73, 11 (11 2024), 2634–2648. doi:10.1109/TC.2024.3441831

[57] Yu Liu, Quanming Yao, and Yong Li. 2020. Generalizing tensor decomposition for n-ary relational knowledge bases. In *Proceedings of the web conference 2020*. 1104–1114.

[58] Per-Gunnar Martinsson and Joel A Tropp. 2020. Randomized numerical linear algebra: Foundations and algorithms. *Acta Numerica* 29 (2020), 403–572.

[59] Devin Matthews. 2016. High-Performance Tensor Contraction without BLAS. *CoRR* abs/1607.00291 (2016). arXiv:1607.00291 http://arxiv.org/abs/1607.00291

[60] Ward Douglas Maurer and Ted G Lewis. 1975. Hash table methods. *ACM Computing Surveys (CSUR)* 7, 1 (1975), 5–19.

[61] David R Musser, Gilmer J Derge, and Atul Saini. 2001. *STL tutorial and reference guide: C++ programming with the standard template library*. Addison-Wesley Longman Publishing Co., Inc.

[62] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. 2008. Scalable parallel programming with CUDA. In *ACM SIGGRAPH 2008 classes*. ACM, New York, NY, USA, 1–14. doi:10.1145/1401132.1401152

[63] Israt Nisa, Jiajia Li, Aravind Sukumaran-Rajam, Prasant Singh Rawat, Sriram Krishnamoorthy, and P. Sadayappan. 2019. An Efficient Mixed-mode Representation of Sparse Tensors. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '19)*. ACM, New York, NY, USA, Article 49, 25 pages. doi:10.1145/3295500.3356216

[64] Alexander Novikov, Dmitrii Podoprikhin, Anton Osokin, and Dmitry P Vetrov. 2015. Tensorizing neural networks. *Advances in neural information processing systems* 28 (2015).

[65] Yuriel Núñez Fernández, Matthieu Jeannin, Philipp T. Dumitrescu, Thomas Kloss, Jason Kaye, Olivier Parcollet, and Xavier Waintal. 2022. Learning Feynman Diagrams with Tensor Trains. *Physical Review X* 12, 4 (11 2022), 041018. doi:10.1103/PhysRevX.12.041018

[66] Yuriel Núñez Fernández, Marc K Ritter, Matthieu Jeannin, Jheng-Wei Li, Thomas Kloss, Thibaud Louvet, Satoshi Terasaki, Olivier Parcollet, Jan von Delft, Hiroshi Shinaoka, and Xavier Waintal. 2025. Learning tensor networks with tensor cross interpolation: New algorithms and libraries. *SciPost Physics* 18, 3 (3 2025), 104. doi:10.21468/SciPostPhys.18.3.104

[67] NVIDIA Corporation. [n. d.]. cuBLAS: CUDA Basic Linear Algebra Subroutines (version 12.0.2). https://developer.nvidia.com/cublas.

[68] OpenAI. 2025. ChatGPT. Large language model. https://chat.openai.com/ Accessed 2025.

[69] Ivan V. Oseledets. 2011. Tensor-Train Decomposition. *SIAM Journal on Scientific Computing* 33, 5 (1 2011), 2295–2317. doi:10.1137/090752286

[70] Ivan V. Oseledets and Eugene Tyrtyshnikov. 2010. TT-cross approximation for multidimensional arrays. *Linear Algebra Appl.* 432, 1 (1 2010), 70–88. doi:10.1016/J.LAA.2009.07.024

[71] David Ozog, Jeff R Hammond, James Dinan, Pavan Balaji, Sameer Shende, and Allen Malony. 2013. Inspector-executor load balancing algorithms for block-sparse tensor contractions. In *2013 42nd International Conference on Parallel Processing*. IEEE, 30–39.

[72] C.-T. Pan. 2000. On the existence and computation of rank-revealing LU factorizations. *Linear Algebra Appl.* 316, 1-3 (9 2000), 199–222. doi:10.1016/S0024-3795(00)00120-8

[73] Feng Pan and Pan Zhang. 2022. Simulation of quantum circuits using the big-batch tensor network method. *Physical Review Letters* 128, 3 (2022), 030501.

[74] Roger Penrose and others. 1971. Applications of negative dimensional tensors. *Combinatorial mathematics and its applications* 1, 221-244 (1971), 3.

[75] Ioakeim Perros, Robert Chen, Richard Vuduc, and Jimeng Sun. 2015. Sparse hierarchical tucker factorization and its application to healthcare. In *2015 IEEE International Conference on Data Mining*. IEEE, 943–948.

[76] Ioakeim Perros, Evangelos E. Papalexakis, Fei Wang, Richard Vuduc, Elizabeth Searles, Michael Thompson, and Jimeng Sun. 2017. SPARTan: Scalable PARAFAC2 for Large & Sparse Data. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Halifax, NS, Canada) *(KDD '17)*. ACM, New York, NY, USA, 375–384. doi:10.1145/3097983.3098014

[77] Christos Psarras, Lars Karlsson, Jiajia Li, and Paolo Bientinesi. 2021. The landscape of software for tensor computations. *arXiv preprint arXiv:2103.13756* (2021).

[78] Zhen Qin, Alexander Lidiak, Zhexuan Gong, Gongguo Tang, Michael B Wakin, and Zhihui Zhu. 2022. Error analysis of tensor-train cross approximation. In *Proceedings of the 36th International Conference on Neural Information Processing Systems (NIPS '22)*. Curran Associates Inc., Red Hook, NY, USA.

[79] Lorenz Richter, Leon Sallandt, and Nikolas Nüsken. 2021. Solving high-dimensional parabolic PDEs using the tensor train format. In *International Conference on Machine Learning*. PMLR, 8998–9009.

[80] Melven Röhrig-Zöllner, Jonas Thies, and Achim Basermann. 2022. Performance of the Low-Rank TT-SVD for Large Dense Tensors on Modern MultiCore CPUs. *SIAM Journal on Scientific Computing* 44, 4 (8 2022), C287–C309. doi:10.1137/21M1395545

[81] Youcef Saad. 1990. *SPARSKIT: A basic tool kit for sparse matrix computations*. Technical Report.

[82] Arvind K Saibaba. 2016. HOID: Higher order interpolatory decomposition for tensors based on Tucker representation. *SIAM J. Matrix Anal. Appl.* 37, 3 (2016), 1223–1249.

[83] Kohtaroh Sakaue, Hiroshi Shinaoka, and Rihito Sakurai. 2025. Adaptive sampling-based optimization of quantics tensor trains for noisy functions: Applications to quantum simulations. *SciPost Physics* 19, 2 (2025), 038.

[84] Dmitry V. Savostyanov. 2014. Quasioptimality of maximum-volume cross interpolation of tensors. *Linear Algebra Appl.* 458 (10 2014), 217–244. doi:10.1016/j.laa.2014.06.006

[85] Parikshit Shah, Nikhil Rao, and Gongguo Tang. 2015. Sparse and low-rank tensor decomposition. *Advances in neural information processing systems* 28 (2015).

[86] Tianyi Shi, Maximilian Ruth, and Alex Townsend. 2021. Parallel algorithms for computing the tensor-train decomposition. (11 2021).

[87] Yue Shi, Alexandros Karatzoglou, Linas Baltrunas, Martha Larson, Alan Hanjalic, and Nuria Oliver. 2012. TFMAP: optimizing MAP for top-n context-aware recommendation. In *Proceedings of the 35th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '12)*. Association for Computing Machinery, New York, NY, USA, 155–164. doi:10.1145/2348283.2348308

[88] Nicholas D Sidiropoulos, Lieven De Lathauwer, Xiao Fu, Kejun Huang, Evangelos E Papalexakis, and Christos Faloutsos. 2017. Tensor decomposition for signal processing and machine learning. *IEEE Transactions on signal processing* 65, 13 (2017), 3551–3582.

[89] Ilia Sivkov, Patrick Seewald, Alfio Lazzaro, and Jürg Hutter. 2019. DBCSR: A blocked sparse tensor algebra library. *arXiv preprint arXiv:1910.13555* (2019).

[90] Shaden Smith, Jee W Choi, Jiajia Li, Richard Vuduc, Jongsoo Park, Xing Liu, and George Karypis. 2017. FROSTT: The Formidable Repository of Open Sparse Tensors and Tools. http://frostt.io/

[91] Shaden Smith and George Karypis. 2016. A Medium-

Grained Algorithm for Distributed Sparse Tensor Factorization. In *Parallel and Distributed Processing Symposium (IPDPS), 2016 IEEE International*. IEEE.

[92] Shaden Smith and George Karypis. 2017. Accelerating the Tucker Decomposition with Compressed Sparse Tensors. In *European Conference on Parallel Processing*. Springer.

[93] Shaden Smith, Niranjay Ravindran, Nicholas Sidiropoulos, and George Karypis. 2015. SPLATT: Efficient and Parallel Sparse Tensor-Matrix Multiplication. In *Proceedings of the 29th IEEE International Parallel & Distributed Processing Symposium* (Hyderabad, India) *(IPDPS)*.

[94] Yongseok Soh, Ramakrishnan Kannan, Piyush Sao, and Jee Choi. 2024. Accelerated Constrained Sparse Tensor Factorization on Massively Parallel Architectures. In *Proceedings of the 53rd International Conference on Parallel Processing*. 107–116.

[95] Edgar Solomonik, Devin Matthews, Jeff Hammond, and James Demmel. 2013. Cyclops tensor framework: Reducing communication and eliminating load imbalance in massively parallel contractions. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE, 813–824.

[96] Qingquan Song, Hancheng Ge, James Caverlee, and Xia Hu. 2019. Tensor completion algorithms in big data analytics. *ACM Transactions on Knowledge Discovery from Data (TKDD)* 13, 1 (2019), 1–48.

[97] Konstantin Sozykin, Andrei Chertkov, Roman Schutski, Anh-Huy Phan, Andrzej Cichocki, and Ivan Oseledets. 2022. TTOpt: a maximum volume quantized tensor train-based optimization and its application to reinforcement learning. In *Proceedings of the 36th International Conference on Neural Information Processing Systems (NIPS '22)*. Curran Associates Inc., Red Hook, NY, USA.

[98] Sridhar Swaminathan, Deepak Garg, Rajkumar Kannan, and Frederic Andres. 2020. Sparse low rank factorization for deep neural network compression. *Neurocomputing* 398 (2020), 185–196.

[99] F. Verstraete and J. I. Cirac. 2006. Matrix product states represent ground states faithfully. *Physical Review B* 73, 9 (3 2006), 094423. doi:10.1103/PhysRevB.73.094423

[100] Sergey Voronin and Per-Gunnar Martinsson. 2017. Efficient algorithms for cur and interpolative matrix decompositions. *Advances in Computational Mathematics* 43, 3 (6 2017), 495–516. doi:10.1007/s10444-016-9494-8

[101] Zhiqian Xu, Yi Fan, Chu Guo, and Honghui Shang. 2024. MPS-VQE: A variational quantum computational chemistry simulator with matrix product states. *Computer Physics Communications* 294 (2024), 108897.

[102] Yongxin Yang and Timothy Hospedales. 2016. Deep multi-task representation learning: A tensor factorisation approach. *arXiv preprint arXiv:1605.06391* (2016).

[103] Xiyu Yu, Tongliang Liu, Xinchao Wang, and Dacheng Tao. 2017. On compressing deep models by low rank and sparse decomposition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 7370–7379.

[104] Xiaoming Yuan and Junfeng Yang. 2009. Sparse and low-rank matrix decomposition via alternating direction methods. *preprint* 12, 2 (2009).

[105] Yifan Zhang, Mark Fornace, and Michael Lindsey. 2025. Fast and Accurate Interpolative Decompositions for General, Sparse, and Structured Tensors. *arXiv preprint arXiv:2503.18921* (3 2025). http://arxiv.org/abs/2503.18921