Near-Optimal Contraction Strategies for the Scalar Product in the Tensor-Train Format

Atte Torri, Przemyslaw Dominikowski, Brice Pointal, Oguz Kaya, Laércio Lima Pilla, Olivier Coulaud

Near-Optimal Contraction Strategies for the Scalar Product in the Tensor-Train Format

Atte Torri[1,2,3](✉), Przemysław Dominikowski[1,4], Brice Pointal[1,2,3],
Oguz Kaya[1,2,3], Laércio Lima Pilla[6], and Olivier Coulaud[5]

[1] Université Paris-Saclay, Gif-sur-Yvette, France
atte.torri@universite-paris-saclay.fr
[2] LISN, Gif-sur-Yvette, France
[3] CNRS, Gif-sur-Yvette, France
[4] Inria, Palaiseau, France
[5] Inria, Bordeaux, France
[6] Univ. Bordeaux, CNRS, Bordeaux INP, Inria, LaBRI, Talence, France

Abstract. Tensor-train (TT) decomposition has garnered tremendous popularity for its efficiency in handling high-dimensional data arising in scientific and quantum computing as well as machine learning applications. It provides a compact representation for matrices and vectors with a Kronecker product-like low-rank structure and enables efficient matrix-vector operations in this compressed form. The vector scalar product is among such key operations, comprising a series of tensor contractions in a specific tensor network topology whose order significantly impacts the computational cost. In this work, we propose efficient algorithms for finding near-optimal contraction orderings for tensor networks representing scalar products in the TT format. We show that our algorithms outperform all existing contraction ordering methods for general tensor networks where the best existing method incurs up to 15% higher cost for $x^T y$, twice the cost for $x^T Ay$, and ten times higher cost for $x^T ABy$ scalar products where x, y and A, B are vectors and matrices expressed in the TT format, respectively.

Keywords: Numerical linear algebra · Multilinear algebra · Tensor decomposition · Tensor-train decomposition · Scalar product · Tensor contraction ordering · Dynamic programming

1 Introduction

In multilinear algebra, a tensor is an element of the tensor product of multiple vector spaces and is capable of representing multidimensional data. Specifically, an N-dimensional tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ has $\prod_{n=1}^{N} I_n$ elements $\mathcal{X}(i_1, \ldots, i_N)$ for $1 \leq i_n \leq I_n$ and $1 \leq n \leq N$. Tensor methods have gained tremendous popularity in numerous domains, including quantum computing, machine learning, scientific computing, signal processing, and many others [7], due to their aptitude in naturally representing high-dimensional data as well as manipulating it efficiently through tensor operations. Among such key operations is the contraction of two tensors, which is reminiscent of the multiplication of two matrices $C = AB$ where $A \in \mathbb{R}^{M \times K}$, $B \in \mathbb{R}^{K \times N}$, and $C \in \mathbb{R}^{M \times N}$ with elements $C(i,j) = \sum_{k=1}^{K} A(i,k)B(k,j)$, which requires $\mathcal{O}(MNK)$ operations. In the tensor case, the contraction of two tensors $\mathcal{A} \in \mathbb{R}^{M_1 \times \cdots \times M_m \times K_1 \times \cdots \times K_k}$ and $\mathcal{B} \in \mathbb{R}^{K_1 \times \cdots \times K_k \times N_1 \times \cdots \times N_n}$ along dimensions $K_1, \ldots, K_k$ yields a tensor $\mathcal{C}$ such that $\mathcal{C}(i_1, \ldots, i_n, j_1, \ldots, j_m) = \sum_{l_1, \ldots, l_k}^{K_1, \ldots, K_k} \mathcal{A}(i_1, \ldots, i_n, l_1, \ldots, l_k)\mathcal{B}(l_1, \ldots, l_k j_1, \ldots, j_m)$. Here, the resulting tensor $\mathcal{C}$ can be computed with $\mathcal{O}(\prod_{i=1}^{m} M_i \prod_{j=1}^{n} N_j \prod_{l=1}^{k} K_l)$ operations, where $\mathcal{C} \in \mathbb{R}^{M_1 \times \cdots \times M_m \times N_1 \times \cdots \times N_n}$.

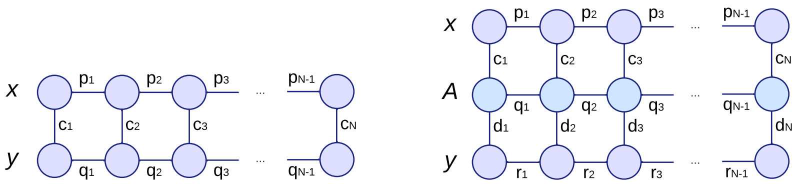
A fundamental limitation of tensor computations is the exponential growth of the computational and memory cost with the number of tensor dimensions, a phenomenon known as *the curse of dimensionality* [7]. Specifically, a tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ requires $\mathcal{O}(I_1 \ldots I_N)$ space, which grows exponentially with N. Fortunately, a high-dimensional tensor can often be represented as or approximated by a network of small-dimensional tensors, called a *tensor decomposition* [7]. The contraction of these tensors over their shared dimensions reconstructs the original tensor. Among various tensor decomposition network topologies, tensor-train (TT) decomposition [12] has garnered great interest from numerous applications. Formally, it represents a tensor $\mathcal{X}$ using N three-dimensional tensors $\mathcal{G}_1, \ldots, \mathcal{G}_N$ in a decomposed form $\mathcal{X} = \mathcal{G}_1 \times_{r_1} \mathcal{G}_2 \times_{r_2} \cdots \times_{r_{N-1}} \mathcal{G}_N$ with $\mathcal{G}_n \in \mathbb{R}^{r_{n-1} \times I_n \times r_n}$ and $r_0 = r_N = 1$. Here, the contraction of $\mathcal{G}_n$ along common/inner network dimensions $r_1, \ldots, r_{N-1}$ yields $\mathcal{X}$ with elements $\mathcal{X}(i_1, i_2, \ldots, i_N) = \sum \mathcal{G}_1(i_1, \alpha_1)\mathcal{G}_2(\alpha_1, i_2, \alpha_2) \ldots \mathcal{G}_N(\alpha_{N-1}, i_N)$ for all $1 \leq \alpha_n \leq r_n$ and $1 \leq n \leq N$. This representation effectively reduces the storage complexity from $\mathcal{O}(I^N)$ to $\mathcal{O}(NIR^2)$ where $R = \max_{1 \leq n \leq N} r_n$ and r_n is called the *rank* of the decomposition. To effectively compress the data, R should remain low, in which case the tensor $\mathcal{X}$ is said to be of *low rank.* This applies in many quantum computing applications where vectors representing a quantum state inherently exhibit a Kronecker product structure, e.g., $x = \sum_{j=1}^{R} \bigotimes_{i=1}^{N} x_i$ for $x_i \in \mathbb{R}^{I_i}$, which can be effectively represented using a low-rank N-dimensional TT decomposition [14]. Similarly, in many scientific computing applications, such Kronecker-like structures exist naturally or can be imposed on a vector $x \in \mathbb{R}^M$ to obtain an $N = \log_2 M$-dimensional tensor representation $\mathcal{X} \in \mathbb{R}^{2 \times \cdots \times 2}$ of vector elements via a process called *quantization* [6]. When done correctly, this preserves a low TT decomposition rank and thereby allows effective compression. We refer to this as the *TT-vector* representation of a vector x. Similarly to vectors, matrices/operators can also have an inherent low-rank structure, e.g., $A = \sum_{j=1}^{R} \bigotimes_{i=1}^{N} A_i$ where $A_i \in \mathbb{R}^{I_i \times J_i}$, or can be likewise *quantized* into such a structure. An N-dimensional TT decomposition can then be used to compactly represent these matrices, which we refer to as the *TT-matrix* representation where each tensor in the decomposition has two free dimensions instead of one. The first and second free dimensions of all tensors in a TT-matrix represent the rows and columns of the full matrix A, respectively.

Once matrices and vectors are represented in this way, the TT framework provides efficient algorithms to carry out all matrix-vector operations, including basic matrix/vector arithmetic and matrix-vector multiplication, within the TT-matrix and TT-vector formats [12]. This capability enables the development of high-dimensional numerical algorithms that leverage the efficiency of compact TT representations. Among these operations is the *scalar product* of two TT-vectors, which is an essential step in many applications. In numerical solvers, the vector scalar product is a key operation used to compute the vector norm ($\sqrt{x^T x}$) or Rayleigh coefficient ($x^T Ax/x^T x$), to name a few examples. In quantum chemistry applications, CholeskyQR algorithm is used to orthogonalize a basis $B = [b_1, \ldots, b_s]$ of s TT-vectors, necessitating the formation of the Gram matrix $G = B^T B$. This requires $\mathcal{O}(s^2)$ scalar products involving TT-vectors in B, which constitutes one of the most computationally expensive steps in the application [14]. In machine learning, kernel methods have recently been extended to use TT-vectors where kernel functions leverage scalar products $x^T y$ or $x^T Ay$ of TT-vectors x and y where A is a TT-matrix [1].

The main focus of this work is to perform such TT scalar product computations near-optimally in order to accelerate the applications whose performance heavily depends on this fundamental operation. In figs. 1a and 1b, we provide the tensor network diagrams corresponding to the scalar products $x^T y$ and $x^T Ay$ where x and y are TT-vectors and A is a TT-matrix with corresponding dimensions. In this diagram, each node represents a tensor in the network, and each outgoing edge from a node represents a dimension of the tensor. A common dimension connecting two tensors indicates a contraction to be performed between them along this dimension. In fig. 1a, p_n and q_n respectively represent the ranks of TT-vectors x and y, while c_n represent common vector dimensions. Similarly, in fig. 1b, the TT-matrix A comprises N four-dimensional tensors with row dimensions c_n, column dimensions d_n, and rank dimensions q_n. In both cases, the scalar product is obtained by the pairwise contraction of all tensors in the network in any order, yielding a 0-dimensional tensor, i.e., a scalar, as no free edge would remain in the network after contractions. However, the order in which these contractions are performed has a significant impact on the total contraction cost. For a general tensor network, finding an optimal contraction order is an NP-hard problem [8]. Our goal in this work is to design efficient algorithms tailored specifically for TT scalar product networks to find a near-optimal contraction ordering in a reasonable time.

2 Related Work

Optimal Algorithms: There are three main approaches to finding an optimal contraction order in a general tensor network: depth-first approaches [8,13], breadth-first strategies [5,13], and a dynamic programming formulation with memoization [13]. Cost capping and outer product restrictions can be leveraged to make optimal algorithms more efficient by pruning unwanted subtrees that are deemed too expensive or involve contractions of unconnected tensors [13].



(a) $x^T y$ (b) $x^T A y$

Fig. 1. Tensor network diagram for a TT scalar product

The `cotengra` library [4] provides an optimized implementation based on the depth-first strategy, which we refer to as *Optimal* in the rest of the paper.

Greedy Algorithms: A simple greedy heuristic consists of sorting all possible contractions in the network by their costs at each step and then carrying out the least expensive one. Since this approach ignores the impact of this choice on the cost of remaining contractions, it is susceptible to causing a high total contraction cost in the end. To address this limitation, another metric for the greedy approach is proposed in [4], which sorts all possible immediate contractions with a heuristic cost function involving the size difference between the contracted tensors $\mathcal{A}$ and $\mathcal{B}$ and the resulting tensor $\mathcal{C}$ of the contraction. This method also includes a weighting parameter α that multiplies the size of the input tensors, resulting in the heuristic cost function $cost(\mathcal{A},\mathcal{B}) = size(\mathcal{C}) - \alpha(size(\mathcal{A}) + size(\mathcal{B}))$. Furthermore, to allow the algorithm to sample multiple greedy paths, a probabilistic *Boltzmann factor* τ is introduced. The algorithm that tunes the parameter α and performs Boltzmann sampling on the factor τ to return one of the best contraction paths is part of the `cotengra` library [4], which we refer to as *Hyper-Greedy*. Recent work [11] extends this approach by using multiple cost functions where each function provides a different contraction order, and the best one is kept. We denote this strategy as *Cgreedy*.

Recursive Tensor Network Partitioning: Another strategy introduced in [4] involves building the contraction tree with a top-down approach by recursively partitioning the network into multiple subnetworks, contracting tensors within each subnetwork among themselves, and finally combining the resulting tensors with another set of contractions. Specifically, the algorithm starts with the root vertex corresponding to the set V containing all tensors in the network. Then, it partitions V into k vertex sets ($V = V_1 \cup V_2 \cup \cdots \cup V_k$) each forming a subnetwork, which are then contracted within themselves with the same partitioning algorithm used recursively. When the size of a subnetwork $|V_k|$ falls below a threshold, another algorithm such as *Optimal* or *Hyper-Greedy* can be used instead. Once each subnetwork is contracted into a single tensor, the resulting tensors are contracted among themselves to obtain the final contraction, for which the order can be determined using one of these two algorithms. One such state-of-the-art partitioner is provided by the `KaHyPar` library [15]. The `cotengra` library [4] uses this partitioner by repeatedly sampling contraction paths and tuning the parameters k and ϵ to keep the path with the minimum cost. We denote this approach as *Hyper-Kahypar*.

Tree Decomposition Approaches: Some strategies perform a *tree decomposition* on the line graph of the tensor network [9]. A tree decomposition maps a graph into a tree structure where each node (bag) represents a subset of vertices from the original graph, capturing its connectivity structure. An optimal tree decomposition has minimal *width* [9] (i.e. equal to *treewidth*). Given an optimal tree decomposition for the line graph of the tensor network, one can use a polynomial-time deterministic algorithm to find an optimal contraction ordering [9]. However, finding an optimal tree decomposition is an NP-hard problem [9] for which there exist heuristics such as *QuickBB* [3] and *FlowCutter* [2,17]. *QuickBB* employs a branch-and-bound strategy to find a tree decomposition, with the aim of effectively pruning the search space. *FlowCutter* uses a graph partitioning approach, leveraging the equivalence between multilevel partitionings and tree decompositions [17]. It performs recursive bisections on the line graph to find a partitioning and compares it with previous results until the treewidth can no longer be notably improved.

Learning Strategies: Reinforcement learning (RL) has emerged as a promising approach for addressing combinatorial optimization problems. In the context of graph-based problems, Graph Neural Networks (GNNs) demonstrate a significant potential. This approach has been applied to the tensor network contraction ordering problem [10], where the problem is modeled as a Markov decision process. In this formulation, the state space comprises all possible graphs, the action space consists of potential edge contractions, the transition function maps the current graph to the subsequent graph with the selected edge contracted, and the reward function quantifies the cost of the chosen contraction. The RL agent is initialized with a graph representing a tensor network and, at each step, selects an edge to contract based on a probability distribution. The agent is trained to minimize contraction costs using proximal policy optimization (PPO) [16,10]. The authors introduce additional techniques to enhance performance, such as path pruning to handle the extensive search space and incorporating existing algorithms to solve part of the problem. We trained and evaluated this method on TT scalar product networks, which we refer to as *RL-TNCO*.

Standard Approach for TT Scalar Product Networks: The standard approach for the TT scalar product performs all contractions involving tensors in the leftmost dimension in a fixed order, which effectively reduces the network's width by one, then repeats the same process on the remaining network. Such an ordering prevents the formation of high-dimensional intermediate tensors, indirectly bounding the contraction cost. In fig. 1a, this approach corresponds to contracting the edge c_1 first and p_1 next. This would eliminate tensors in the first two dimensions yet add an edge q_1 parallel to c_2 in the second dimension. The algorithm similarly proceeds to eliminate the following dimensions (contracting the edge q_1c_2 first, p_2 next, etc.) until the entire network is contracted. We extend this strategy to the scalar product x^TAy with respect to a TT-matrix A. In fig. 1b, this corresponds to the contraction of edges c_1, d_1, and q_1 in order, which eliminates the first dimension while similarly adding parallel edges p_1 to c_2 and r_1 to d_2. The algorithm then proceeds to contract the tensors in the remaining dimensions in the same order (by contracting edges p_1c_2, r_1d_2, and q_2, etc.). In both cases, the algorithm runs on both sides of the network and picks the best of two orderings. We denote this approach as the *Sweep* method, which is widely used in tensor libraries such as TT-Toolbox [12].

3 Effective Contraction Ordering Strategies for TT Scalar Product Networks

In this section, we propose two new algorithms, namely *Sweep-opt* and *Δ-opt*, which take advantage of the special structure of TT scalar product networks to find near-optimal contraction orderings using efficient dynamic programming.

3.1 Optimal Sweep Algorithm

For computing x^Ty, the *Sweep* method eliminates each dimension i by contracting the edge c_i followed by p_i, resulting in the addition of an edge q_i parallel to c_{i+1} in the network. However, this dimension can also be eliminated by contracting c_i and q_i or p_i and q_i, which adds edge p_i or c_i parallel to c_{i+1}, respectively. In all cases, parallel edges are equivalent to having one edge whose size is the product of the sizes of parallel edges. The goal of the *Sweep-opt* algorithm is to eliminate one dimension at a time similarly while making optimal decisions.

Since the choice of two contractions (p_i/c_i, q_i/c_i, or p_i/q_i) to eliminate a dimension modifies the remaining network, a straightforward approach would consider all $\mathcal{O}(3^N)$ possibilities and keep the best, which is infeasible for large N. Fortunately, we observe that the number of possible modifications to the remaining network is limited, restricting the search space and creating a small number of overlapping subproblems. Performing a contraction on p_i/c_i or q_i/c_i eliminates all existing parallel edges on c_i, adding solely the edge q_i or p_i parallel to c_{i+1}, respectively. However, a p_i/q_i contraction propagates all current parallel edges of c_i as well as c_i itself onto c_{i+1}. Therefore, for any dimension i, the set of edges parallel to c_i can only be of the form $\{p_k, c_{k+1}, \dots, c_{i-1}\}$ or $\{q_k, c_{k+1}, \dots, c_{i-1}\}$ for $0 \le k < i$ (fig. 2). We define $\mathcal{F}_p(i,k)$ as the optimal contraction cost of the *Sweep-opt* strategy for the subnetwork involving dimensions $[i, N]$, given that c_i has parallel edges $\{p_k, c_{k+1}, \dots, c_{i-1}\}$. We define $\mathcal{F}_q$ in an analogous manner. This creates $\mathcal{O}(N^2)$ subproblems with an overlapping structure depending on the choice of two contractions per dimension as follows:

$$\begin{tikzcd}
 & \mathcal{F}_q(i+1,i) & & \mathcal{F}_q(i+1,i) \\
\mathcal{F}_p(i,k) \arrow[ru, "p_i/c_i"] \arrow[r, "q_i/c_i"] \arrow[rd, "p_i/q_i"] & \mathcal{F}_p(i+1,i) & \mathcal{F}_q(i,k) \arrow[ru, "p_i/c_i"] \arrow[r, "q_i/c_i"] \arrow[rd, "p_i/q_i"] & \mathcal{F}_p(i+1,i) \\
 & \mathcal{F}_p(i+1,k) & & \mathcal{F}_q(i+1,k)
\end{tikzcd} \tag{1}$$

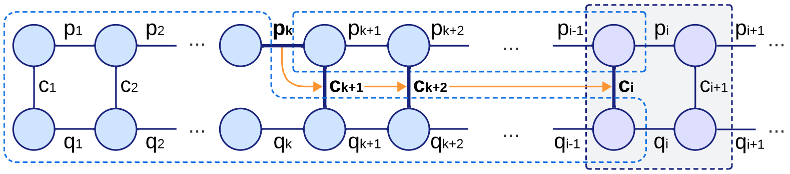


Fig. 2. Formation of the edge set $\{p_k, c_{k+1}, \ldots, c_{i-1}\}$ parallel to c_i

Let $\mathcal{C}_{pc}(i,k)$ be the minimum of the cost of contracting p_i first and c_i next or vice versa on a subnetwork $[i,N]$ having edges $\{p_k, c_{k+1}, \ldots, c_{i-1}\}$ parallel to c_i. We define $\mathcal{C}_{qc}$ and $\mathcal{C}_{pq}$ similarly. Following eq. (1), we can then define the optimal sweeping contraction cost for this subnetwork as

$$\mathcal{F}_p(i,k) = \min\{\mathcal{F}_q(i{+}1,i){+}\mathcal{C}_{pc}(i,k), \mathcal{F}_p(i{+}1,i){+}\mathcal{C}_{qc}(i,k), \mathcal{F}_p(i{+}1,k){+}\mathcal{C}_{pq}(i,k)\}$$

and express $\mathcal{F}_q(i,k)$ analogously.

Complexity: If the values $p_k c_{k+1} \ldots c_i$ and $q_k c_{k+1} \ldots c_i$ are precomputed for all $0 \leq k < i \leq N$ in $\mathcal{O}(N^2)$ time, each evaluation of $\mathcal{C}$ can be performed in constant time. The same holds then for the computation of $\mathcal{F}_p(i,k)$ and $\mathcal{F}_q(i,k)$, yielding $\mathcal{O}(N^2)$ time and space complexity for the *Sweep-opt* algorithm. We run the algorithm from both ends of the tensor network and keep the best solution.

3.2 Optimal Δ-Window Algorithm

Despite being very efficient, *Sweep-opt* has two potential drawbacks. First, for networks with very irregular ranks or dimension sizes, eliminating a single dimension at a time, albeit optimally, might still diverge from the optimal contraction order. Second, it cannot be generalized to scalar products involving matrices (e.g., $x^T A y$) with the same quadratic complexity. To remedy these, we propose another algorithm, *Δ-opt*, which aims to optimally eliminate multiple dimensions at a time within a small window of Δ dimensions.

Consider the tensor network for $x^T y$ whose tensors in the first $i{-}1$ dimensions are already contracted. Regardless of the order of these contractions, we note that this creates an intermediate tensor of size $p_{i-1} \times q_{i-1}$ connected to the rest of the network that involves tensors in dimensions $[i,N]$ and remains unmodified. The aim of *Δ-opt* is to find a near-optimal contraction cost from this starting configuration to contract all tensors up to dimension $j \geq i$, which we represent as $\mathcal{G}_{LR}(i,j)$. A key observation is that performing these contractions creates the same initial configuration for the dimension $j{+}1$ this time, with an intermediate tensor of size $p_j \times q_j$ connected to the rest of the network for dimensions $[j{+}1,N]$ which remain unmodified. If the window size is deemed small (i.e., $j - i < \Delta$), we resort to an optimal solver. Otherwise, we consider all possible splits and take the minimum, that is, $\mathcal{G}_{LR}(i,j) = \min\limits_{k=i\ldots j-1}(\mathcal{G}_{LR}(i,k) + \mathcal{G}_{LR}(k+1,j))$. However, since $\mathcal{G}_{LR}(i,k)$ itself may be composed of splits, these splits eventually produce a series of windows smaller than Δ, which are solved optimally. Therefore, it is unnecessary to consider $k \geq i + \Delta$, which simplifies the formulation to

$$\mathcal{G}_{LR}(i,j) = \begin{cases} \mathcal{G}^{(opt)}_{LR}(i,j), & j - i < \Delta \\ \min\limits_{k=i \ldots \min(i+\Delta-1,N)} (\mathcal{G}_{LR}(i,k) + \mathcal{G}_{LR}(k+1,j)) & \text{otherwise} \end{cases} \tag{2}$$

where $\mathcal{G}^{(opt)}_{LR}(i,j)$ is the contraction cost obtained from an optimal solver. We similarly compute all $\mathcal{G}_{RL}(i,j)$ from the other end of the network.

Finally, we consider all one-sided or meet-in-the-middle contraction scenarios for the final solution as

$$\min \begin{pmatrix} \mathcal{G}_{LR}(1,N), \\ \mathcal{G}_{RL}(1,N), \\ \min\limits_{i=1 \ldots N-1} (\mathcal{G}_{LR}(1,i) + \mathcal{G}_{RL}(i+1,N) + p_i q_i) \end{pmatrix} \tag{3}$$

where $p_i q_i$ represents the cost of the final contraction when both subnetworks $[1,i)$ and $(i+1,N]$ are contracted beforehand.

Complexity: In the Δ-*opt* implementation, we first precompute the optimal solutions $\mathcal{G}^{(opt)}_{LR}(i,j)$ for all windows $1 \leq i \leq j \leq \min(i+\Delta-1,N)$ and set the base cases for the dynamic programming formulation. Then, we compute each $\mathcal{G}_{LR}(i,j)$ using $O(\Delta)$ table lookups, which gives $\mathcal{O}(N^2\Delta + N\sum_{d=1}^{\Delta} opt_{2d+1})$ time and $\mathcal{O}(N^2)$ space complexity where opt_k is the cost of running an optimal solver on a network with k tensors. The final step in eq. (3) only takes $\mathcal{O}(N)$ time. The algorithm generalizes to products such as $x^T A y$ or $x^T A B y$; we skip the details.

Choice of $\boldsymbol{\Delta}$: If Δ is chosen to be a very small constant, the algorithm has $\mathcal{O}(N^2)$ time and space complexity; however, the cost can escalate with increasing Δ due to the optimal solver. Therefore, we suggest iterative discovery of Δ in practice, starting with a small window ($\Delta = 2$) and increasing it until either the solution no longer improves or the optimal solver becomes too expensive.

4 Experiments

In this section, we compare the contraction cost and execution time of our algorithms against those of *Optimal*, *Hyper-Greedy*, *Cgreedy*, *Hyper-Kahypar*, *Flow-Cutter*, *QuickBB*, *RL-TNCO*, and *Sweep* approaches. The implementation of our algorithms, along with scripts to generate our results, is available on GitHub. [7]

4.1 Setup

For all algorithms except *RL-TNCO*, we used the implementations provided in the `cotengra` library (version `0.6.2`). Additionally, Δ-*opt* incorporates the *Optimal* implementation from `cotengra` as a subroutine.

[7] https://github.com/Blixodus/TT-ScalProdOpt

Our benchmarks consist of TT scalar products of the form $x^T y$, $x^T Ay$, and $x^T ABy$, using TTs with a number of dimensions ranging from 2 to 100, as well as varying rank and dimension size characteristics as follows:

- *rand-rand* (random-random): Dimensions and ranks follow a uniform random distribution, with dimension sizes between 2 and 50, and ranks ranging from 1 to 200.
- *quant-rand* (quantized-random): All dimensions are set to 2 as in the quantized TT formats [6], and the ranks follow a uniform random distribution between 1 and 200.
- *quant-incr* (quantized-increasing): All dimensions are equal to 2, and ranks progressively increase towards the center of TT with some random variation. The peak rank is achieved around the middle of the network for matrices A and B, and around the first and third quartile of dimensions for vectors x and y, respectively. Such patterns are commonly observed in many applications that use TTs to represent low-rank high-dimensional data [6].

In all cases, TTs are ensured to meet the rank feasibility conditions, i.e., $p_i \leq p_{i+1}c_{i+1}$ and $p_i \leq p_{i-1}c_i$, which are enforced in practice by TT-SVD and rank reduction algorithms [12]. We also tested the cases $x^T x$ and $x^T Ax$, which correspond to vector norm and A-norm computations, respectively, and using lower (up to 50) and higher (up to 1000) maximum ranks. The results were comparable to $x^T y$ and $x^T Ay$ experiments with ranks up to 200; therefore, for brevity, we do not report them here.

For each test case and number of TT dimensions, we evaluated 50 instances. For each instance and algorithm, the contraction cost was normalized with respect to the minimum cost obtained for that instance across all algorithms. In our visualizations, the dark lines represent the geometric mean of these 50 normalized results, while the lighter-shaded regions represent the geometric standard deviation from this mean. A solid vertical black line marks the point beyond which the optimal algorithm is no longer executed due to exceeding a computational time threshold of 60 minutes. Beyond this cutoff, cost normalization is performed using the lowest cost achieved among the remaining algorithms.

For *RL-TNCO*, we trained the model on a diverse set of TT scalar product networks. Specifically, for each TT type (*rand-rand*, *quant-rand*, *quant-incr*), we generated 100 training instances, each with 100 dimensions. During training, all reinforcement learning parameters were kept at their default values. Due to machine availability constraints, we evaluated *RL-TNCO* with a subset of the dataset used for other methods, considering only 25 instances per configuration and only networks of type $x^T y$ and $x^T Ay$. For some instances, *RL-TNCO* failed to produce a valid ordering; these cases were excluded from the statistical analysis.

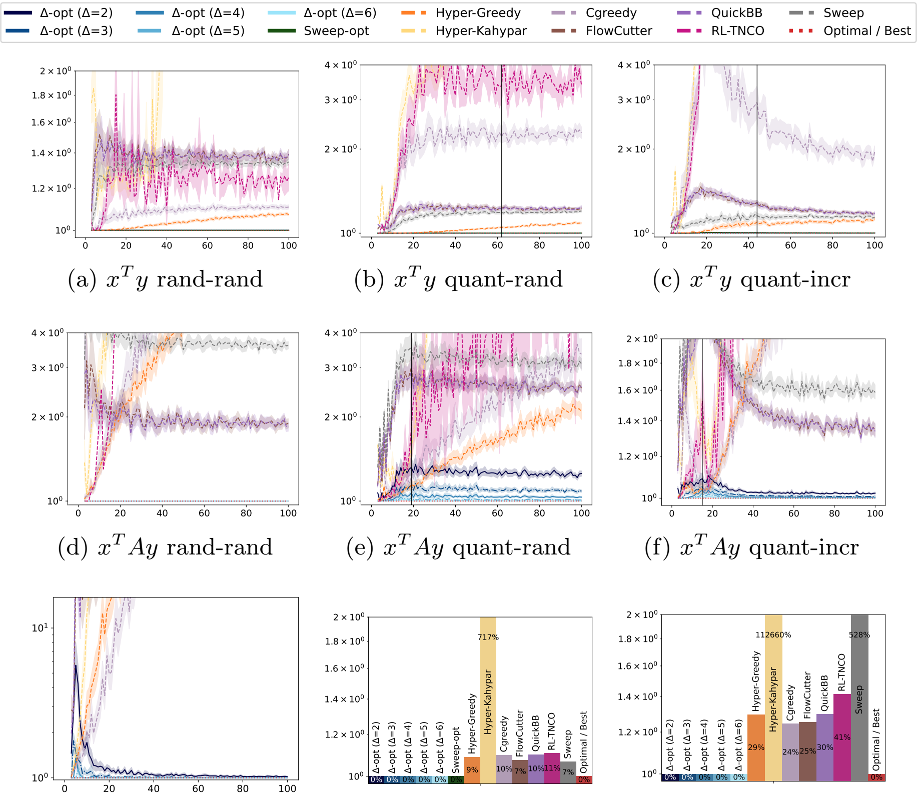
In addition to synthetic benchmarks, we evaluate the algorithms on a quantum chemistry application that aims to compute the vibrational spectra of molecules [14]. This application employs a TT-based iterative eigensolver, where both $x^T y$ and $x^T Ay$ serve as key steps within each iteration. We evaluated using TTs with 60 dimensions generated during the first iteration, using an error tolerance of $\epsilon = 10^{-6}$ for rank truncation.

We performed our experiments, excluding *RL-TNCO*, on machines equipped with two 20-core *Intel® Xeon® Gold 6230* processors operating at a fixed frequency of 2.1 GHz, along with 192 GB of RAM. For *RL-TNCO*, which requires a GPU, we used machines with the same processors but with 768 GB of RAM, and an *Nvidia® Tesla® V100* GPU featuring 32 GB of VRAM. All algorithms, except *RL-TNCO*, were run on a single core using a unified C++ wrapper to mitigate overhead effects associated with parsing the TT descriptor and invoking the respective libraries. The reported execution times exclusively correspond to the function responsible for computing and returning the contraction cost for each algorithm.

4.2 Results

Vector-Vector Product (x^Ty): In figs. 3a to 3c, we report the total contraction cost for computing x^Ty for all algorithms. We first note that in all three cases of *rand-rand*, *quant-rand*, and *quant-incr*, *Sweep-opt* gives results that are indistinguishable from the optimal and provides a dramatic improvement over *Sweep*. This not only suggests that contracting tensors site-by-site is preferable for such TT scalar products, but also demonstrates that choosing the optimal contraction ordering among p_iq_i, p_ic_i, or q_ic_i is crucial for minimizing the total contraction cost, thus advocating the use of *Sweep-opt* for such networks. *Δ-opt* similarly gives near-optimal results in all three cases using a window size Δ as small as 2, as it can similarly identify the ideal contraction order owing to the use of an optimal algorithm in a small window. Among other methods, *Hyper-Greedy* performs the best, starting close to the optimal for small networks, yet slowly diverging from it as the number of dimensions increases, yielding up to 15% higher contraction cost for 100 dimensions. Next, *Cgreedy* performs significantly worse than *Hyper-Greedy* overall, while *FlowCutter* and *QuickBB* converge to a non-optimal plateau in all three cases. *Hyper-Kahypar* incurs a notably high contraction cost overall, while displaying erratic contraction costs beyond 35 dimensions. This suggests that although hypergraph partitioning can be very useful for general tensor networks, it is not an effective strategy for structured networks exhibiting a 1D-like structure, such as our case. Finally, *RL-TNCO* performs poorly, particularly in the *quant-incr* case, which disqualifies it as an effective strategy for such scenarios, especially considering its high training and inference costs for RL, when near-optimal solutions can be computed using *Sweep-opt*.

In fig. 3h, we provide the results using TTs from the quantum chemistry application. We observe that *Hyper-Kahypar* performs very poorly, causing more than eight times higher contraction cost than *Optimal*. Next, similar to figs. 3a to 3c, both *Sweep-opt* and *Δ-opt* using $\Delta \geq 2$ perform identical to *Optimal*. Among the remaining methods, *RL-TNCO* has the worst performance with 11% higher contraction cost than *Optimal*, and surprisingly, *Sweep* is the best with 7% higher cost than *Optimal*. This is mostly due to this network having fixed dimension sizes ($c_i = 15$) and low-rank variation, making the problem “easier”.



(a) $x^T y$ rand-rand (b) $x^T y$ quant-rand (c) $x^T y$ quant-incr

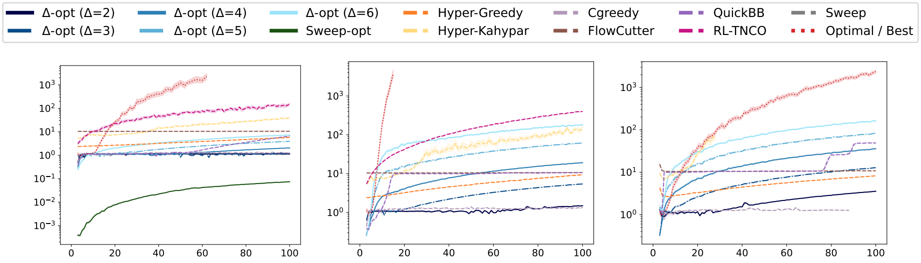
(d) $x^T Ay$ rand-rand (e) $x^T Ay$ quant-rand (f) $x^T Ay$ quant-incr

(g) $x^T ABy$ rand-rand (h) $x^T y$ in real application (i) $x^T Ay$ in real application

Fig. 3. Contraction cost for different types of TT scalar products. The y-axis represents the contraction cost relative to the optimal/best method. For figs. 3a to 3g, the x-axis is the number of TT dimensions. For figs. 3h and 3i, each bar represents a method.

Vector-Matrix-Vector Product ($x^T Ay$): In figs. 3d to 3f, we provide the contraction costs for all algorithms except *Sweep-opt* for the $x^T Ay$ computation. In the *rand-rand* case in fig. 3d, we first note that *Δ-opt* provides near-optimal results for Δ as small as 2 across all tensor network sizes. Next, we observe that *QuickBB* and *FlowCutter* significantly outperform other methods, while incurring up to twice the contraction cost of *Δ-opt*. Finally, *Sweep* converges to approximately four times the contraction cost of *Δ-opt* for larger tensor networks, whereas the remaining methods (*Hyper-Greedy*, *Cgreedy*, *Hyper-Kahypar*, *RL-TNCO*) incur more than ten times the optimal cost for such networks.

Next, in the *quant-rand* case in fig. 3e, we first observe that *Δ-opt* no longer finds near-optimal orderings for $\Delta = 2$, but it converges to a plateau of around 26% higher cost than the optimal. Increasing the window size to 3 reduces this to around 9%, and for $\Delta > 4$, the results become near-optimal for all instances. The best contender in this case is *Hyper-Greedy*, which incurs a higher contrac-



(a) $x^T y$ quant-rand (b) $x^T Ay$ quant-incr (c) $x^T ABy$ rand-rand

Fig. 4. Execution time of algorithms for specific instances. The y-axis is the algorithm's execution time in seconds. The x-axis is the number of TT dimensions.

tion cost as the number of dimensions increases, requiring more than twice the optimal cost for large networks. *Cgreedy* performs worse than *Hyper-Greedy* similarly, *QuickBB* and *FlowCutter* converge to a non-optimal plateau, and *Hyper-Kahypar* and *RL-TNCO* diverge from the optimal.

In fig. 3f, *quant-incr* shows similar tendencies, except that even for $\Delta = 3$, *Δ-opt* manages to find near-optimal solutions in most instances. Finally, fig. 3i provides the results of $x^T Ay$ contraction from the quantum chemistry application, where *Hyper-Kahypar* performs the worst, and *Sweep* incurs more than six times the cost of *Optimal* this time. *Δ-opt* stays indistinguishable from *Optimal* even for $\Delta = 2$, proving to be the method of choice for such networks. *Cgreedy* is the best among the remaining methods with 24% higher cost than *Optimal*.

Vector-Matrix-Matrix-Vector Product ($x^T ABy$): In fig. 3g, we provide the contraction cost for all algorithms except *Sweep-opt* for the $x^T ABy$ *rand-rand* case, to test how adding another layer into the network would affect their performance. We observe that *Δ-opt* provides near-optimal results for large networks even for $\Delta = 2$, whereas we need to have $\Delta > 4$ to guarantee such results for all network sizes. In contrast, all other algorithms rapidly diverge from the optimal, incurring a higher contraction cost of more than 10× as the number of dimensions goes beyond 25. Comparing this result with those in figs. 3a and 3d for $x^T y$ and $x^T Ay$, the advantage of *Δ-opt* seems to increase as the tensor network involves more layers.

Execution Time: In fig. 4, we provide the execution time of all algorithms. Figure 4a shows the execution time for the $x^T y$ computation in the *quant-rand* setting. We first observe that *Sweep-opt* significantly outperforms all other methods, requiring only a fraction of their execution time, which makes it the method of choice for $x^T y$ scalar products given that it also provides near-optimal results as provided in figs. 3a to 3c. Next, we observe that *Δ-opt* remains among the fastest algorithms for $\Delta = 2$ or 3, and the cost progressively increases as Δ grows. The execution cost of *Optimal* grows rapidly, and the algorithm becomes infeasible for medium to large networks. *Cgreedy* is the fastest among our benchmarks and remains competitive with Δ-*opt* using small window sizes. Δ-*opt* stays very competitive for $\Delta \leq 4$ and its cost progressively rises as Δ grows larger. *RL-TNCO* is the slowest among all algorithms except *Optimal*, despite the fact that it uses a GPU for inference, showing that it is not a viable option for this type of structured tensor network.

Next, in fig. 4b, we provide the execution time for the $x^T Ay$ computation in the *quant-incr* setting. Here, we first observe that the optimal algorithm quickly becomes infeasible, even for small networks. *Cgreedy* and Δ-*opt* with $\Delta = 2$ are the fastest in this case, and *Hyper-Greedy* follows suit. Δ-*opt* for $\Delta = 6$ remains comparable to *Hyper-Kahypar* in this case. Δ-*opt* stays very efficient up to $\Delta = 4$, which is sufficient to ensure near-optimal results as shown in fig. 3e, and gets progressively more expensive for larger window sizes. *QuickBB* and *FlowCutter* converge to a plateau and become comparable to *Hyper-Greedy* for large tensor networks. In light of these results, we conclude that Δ-*opt* with a small window size is indeed the method of choice for this type of network.

Finally, in fig. 4c, we provide the timings for the $x^T ABy$ case in the *rand-rand* setting, which trends similar to fig. 4b, with Δ-*opt* getting slightly more expensive due to the added layer in the network. Another interesting observation is that *Optimal*, although very expensive, manages to find a solution for larger networks this time. This is mostly due to the randomization in both tensor dimensions and ranks, which provides aggressive pruning possibilities in the branch-and-bound search. Nonetheless, these results again suggest the use of Δ-*opt* with a small Δ to find a near-optimal ordering for such tensor networks.

5 Conclusion

In this work, we introduced two new algorithms, namely *Sweep-opt* and Δ-*opt*, to find a near-optimal contraction ordering for the scalar product of vectors in TT form. These algorithms provide a significant improvement compared to existing algorithms for finding an efficient contraction ordering for a general tensor network, reducing the contraction cost by more than tenfold in some instances. Experiments show that our algorithms provide solutions close to the optimal algorithm in most test cases while necessitating only a fraction of the time the optimal solver requires, thus offering an efficient mechanism for finding a near-optimal contraction ordering for the scalar product of TTs. Owing to this, we aim to accelerate a critical step in numerical solvers that leverage the TT format to represent high-dimensional operators and vectors, as well as kernel methods whose performance heavily relies on such scalar products.

Acknowledgments. This work was supported by the NumPEx Exa-SofT(ANR-22-EXNU-0003) and SELESTE (ANR-20-CE46-0008-01) projects of the French National Research Agency (ANR), and Paris Ile-de-France Region (DIM RFSI RC-TENSOR No. 2021-05). The experiments are carried out using computational resources from the “Mésocentre” computing center of Université Paris-Saclay, CentraleSupélec, and École Normale Supérieure Paris-Saclay supported by CNRS and Région Île-de-France (https://mesocentre.universite-paris-saclay.fr/).

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Chen, C., Batselier, K., Yu, W., Wong, N.: Kernelized support tensor train machines. Pattern Recognition **122**, 108337 (2022). https://doi.org/10.1016/j.patcog.2021.108337
2. Dudek, J.M., Dueñas-Osorio, L., Vardi, M.Y.: Efficient contraction of large tensor networks for weighted model counting through graph decompositions. arXiv (2019). https://doi.org/10.48550/arXiv.1908.04381
3. Gogate, V., Dechter, R.: A complete anytime algorithm for treewidth. In: Proceedings of the 20th Conference on Uncertainty in Artificial Intelligence. pp. 201–208. UAI '04, AUAI Press, Arlington, Virginia, USA (2004). https://doi.org/10.5555/1036843.1036868
4. Gray, J., Kourtis, S.: Hyper-optimized tensor network contraction. Quantum **5**, 410 (2021). https://doi.org/10.22331/q-2021-03-15-410
5. Hartono, A., Sibiryakov, A., Nooijen, M., Baumgartner, G., Bernholdt, D.E., Hirata, S., Lam, C.C., Pitzer, R.M., Ramanujam, J., Sadayappan, P.: Automated operation minimization of tensor contraction expressions in electronic structure calculations. In: Sunderam, V.S., van Albada, G.D., Sloot, P.M.A., Dongarra, J.J. (eds.) Computational Science – ICCS 2005. vol. 3514, pp. 155–164. Springer, Heidelberg (2005). https://doi.org/10.1007/11428831_20
6. Khoromskij, B.N.: O(dlogN)-quantics approximation of N-d tensors in high-dimensional numerical modeling. Constructive Approximation **34**(2), 257–280 (2011). https://doi.org/10.1007/s00365-011-9131-1
7. Kolda, T.G., Bader, B.W.: Tensor decompositions and applications. SIAM Review **51**(3), 455–500 (2009). https://doi.org/10.1137/07070111X
8. Lam, C.C., Sadayappan, P., Wenger, R.: On optimizing a class of multi-dimensional loops with reduction for parallel execution. Parallel Processing Letters **07**(02), 157–168 (1997). https://doi.org/10.1142/S0129626497000176
9. Markov, I.L., Shi, Y.: Simulating quantum computation by contracting tensor networks. SIAM Journal on Computing **38**(3), 963–981 (2008). https://doi.org/10.1137/050644756
10. Merom, E., Maron, H., Mannor, S., Chechick, G.: Optimizing tensor network contraction using reinforcement learning. In: Proceedings of the 39th International Conference on Machine Learning. vol. 162, pp. 15278–15292. PMLR (2022). https://doi.org/10.48550/arXiv.2204.09052
11. Orgler, S., Blacher, M.: Optimizing tensor contraction paths: a greedy algorithm approach with improved cost functions. arXiv (2024). https://doi.org/10.48550/arXiv.2405.09644
12. Oseledets, I.V.: Tensor-train decomposition. SIAM Journal on Scientific Computing **33**(5), 2295–2317 (2011). https://doi.org/10.1137/090752286
13. Pfeifer, R.N.C., Haegeman, J., Verstraete, F.: Faster identification of optimal contraction sequences for tensor networks. Physical Review E **90**(3), 033315 (2014). https://doi.org/10.1103/PhysRevE.90.033315

14. Rakhuba, M., Oseledets, I.V.: Calculating vibrational spectra of molecules using tensor train decomposition. The Journal of Chemical Physics **145**(12), 124101 (2016). https://doi.org/10.1063/1.4962420
15. Schlag, S., Heuer, T., Gottesbüren, L., Akhremtsev, Y., Schulz, C., Sanders, P.: High-quality hypergraph partitioning. ACM Journal of Experimental Algorithmics **27**, 1.9:1–1.9:39 (2023). https://doi.org/10.1145/3529090
16. Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O.: Proximal policy optimization algorithms. arXiv (2017). https://doi.org/10.48550/arXiv.1707.06347
17. Strasser, B.: Computing tree decompositions with FlowCutter: PACE 2017 submission. arXiv (2017). https://doi.org/10.48550/arXiv.1709.08949