SIAM J. Sci. Comput. © 2022 Society for Industrial and Applied Mathematics
Vol. 44, No. 4, pp. C287–C309

PERFORMANCE OF THE LOW-RANK TT-SVD FOR LARGE DENSE TENSORS ON MODERN MULTICORE CPUs[*]

MELVEN RÖHRIG-ZÖLLNER[†], JONAS THIES[‡], AND ACHIM BASERMANN[†]

Abstract. There are several factorizations of multidimensional tensors into lower-dimensional components, known as "tensor networks." We consider the popular "tensor-train" (TT) format and ask, How efficiently can we compute a low-rank approximation from a full tensor on current multicore CPUs? Compared to sparse and dense linear algebra, kernel libraries for multilinear algebra are rare and typically not as well optimized. Linear algebra libraries like BLAS and LAPACK may provide the required operations in principle but often at the cost of additional data movements for rearranging memory layouts. Furthermore, these libraries are typically optimized for the compute-bound case (e.g., square matrix operations), whereas low-rank tensor decompositions lead to memory bandwidth limited operations. We propose a "TT singular value decomposition" (TT-SVD) algorithm based on two building blocks: a "Q-less tall-skinny QR" factorization and a fused tall-skinny matrix-matrix multiplication and reshape operation. We analyze the performance of the resulting TT-SVD algorithm using the roofline performance model. In addition, we present performance results for different algorithmic variants for shared-memory as well as distributed-memory architectures. Our experiments show that commonly used TT-SVD implementations suffer severe performance penalties. We conclude that a dedicated library for tensor factorization kernels would benefit the community: Computing a low-rank approximation can be as cheap as reading the data twice from main memory. As a consequence, an implementation that achieves realistic performance will move the limit at which one has to resort to randomized methods that only process part of the data.

Key words. tensor decomposition, performance modeling, high-dimensional problems, higher-order SVD high-perfomence computing, TT-format

AMS subject classifications. 15A23, 15A69, 65F99, 65Y05, 65Y20

DOI. 10.1137/21M1395545

1. Introduction.

The tensor-train (TT) decomposition is a particular form of a tensor network representation of a high-dimensional tensor in which the 3D "core tensors" are aligned in a 1D format and connected by a contraction with their direct neighbors only to represent (or approximate) a d-dimensional tensor. It was introduced as such by Oseledets and Tyrtyshnikov [32, 34]) but in fact has been known to (and used by) computational physicists under the name of matrix product states since the 1980s [1, 2]; see also [41] for a more recent reference. Closely related is the density matrix renormalization group algorithm [42], an optimization method that operates on the space of matrix product states. An overview on numerical algorithms based on low-rank tensor approximations can be found in [19]. Recent research also focuses on applications of TTs in data science see; e.g., [9, 10, 26, 28] for a few examples. The performance of common arithmetic operations in TT format (such as additions and scalar products) is discussed in [12].

One can construct an approximate TT decomposition of high-dimensional data $X \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_d}$ using a high-order singular value decomposition. An algorithm for this, called TT-SVD, is presented in [33]. Given X and a maximum "bond dimension"

[*]Submitted to the journal's Software and High-Performance Computing section February 1, 2021; accepted for publication (in revised form) February 28, 2022; published electronically July 14, 2022. https://doi.org/10.1137/21M1395545

[†]Simulation and Software Technology, German Aerospace Center (DLR), Cologne 51147, Germany (Melven.Roehrig-Zoellner@DLR.de, Achim.Basermann@DLR.de).

[‡]Institute of Applied Mathematics and High Performance Computing Center, TU Delft, Delft, 2628 CD, The Netherlands (J.Thies@TUDelft.nl).

r_{max}, it successively determines the core tensors $T^{(j)} \in \mathbb{R}^{r_{j-1}\times n_j \times r_j}, j = 1 \dots d$, such that $r_0 = r_d = 1$, $r_j \leq r_{max}$, the rows or columns of some matricization of all but one $T^{(j)}$ are orthonormal, and the approximation error (difference between the TT formed by the $T^{(j)}$ and the original data tensor X) is minimized (up to a constant factor) in the Frobenius norm on the manifold of rank-r_{max} TT [33]. Definitions of some of these concepts are obviously needed and will be given in section 2.

The aim of this paper is to develop an efficient formulation and implementation of this algorithm for modern multicore CPUs. We focus on situations where the data is large and dense, but it is feasible to process the complete data set for which a low-rank representation is sought (i.e., to read the data $\mathcal{O}(1)$ times). In contrast, randomized (sampling) algorithms only access part of the data and can be used if the data set is too large [27, 29]. For the deterministic case, error bounds and asymptotic complexity estimates (for the size of the result) exist but differ slightly depending on the desired tensor format; see [19] and the references therein. One usually seeks an approximation with a specific accuracy (in terms of maximal size of the resulting approximation or a tolerance or both). However, common implementations often provide suboptimal performance for this case, as they do not take into account that the computation is limited by data transfers on current computers (see section 5). We investigate the TT-SVD because this is a simple and popular choice, but the ideas can be transferred to other tree tensor networks (see, e.g., [18]) as the algorithmic building blocks are similar. An important ingredient in our implementation is a Q-less "tall-skinny QR" (TSQR, see [13]) variant that is described in detail in section 3.2. The idea to avoid computing and storing the large matrix Q of a QR decomposition was already exploited for, e.g., sparse matrix decompositions and tensor calculus in [6, 16].

Our contribution is twofold. First, based on the example of the TT-SVD algorithm we show that low-rank tensor approximation is a memory-bound problem in high dimensions (in contrast to the SVD in two dimensions for square matrices). Second, we discuss how the TT-SVD algorithm can be implemented efficiently on current hardware. In order to underline our findings, we present performance results for the required building blocks and for different TT-SVD variants and implementations on a small CPU cluster.

The remainder of this paper is organized as follows. In section 2, we introduce the basic concepts and notation for tensor networks and performance engineering that we will use to describe our algorithms and implementation. In section 3 we describe the TT-SVD algorithm with a focus on our tailored Q-less TSQR variant. In section 4 we present a performance model for the two key components of TT-SVD (Q-less TSQR and a "tall-skinny" matrix-matrix multiplication), as well as the overall algorithm. Numerical experiments comparing actual implementations of TT-SVD (including our own optimized version) against the performance model can be found in section 5, and the paper closes with a summary of our findings in section 6.

2. Background and notation.

2.1. Tensor notation and operations.

Classical linear algebra considers matrices and vectors ($n \times 1$ matrices) and provides a notation for operations between them based on matrix-matrix products and matrix transpositions. We make use of this common notation where possible. In this paper, a dense d-dimensional array or tensor is denoted by $X \in \mathbf{R}^{n_1\times\cdots\times n_d}$. We can combine and split dimensions through reshape operations, e.g.,

$$Y = \text{reshape}\left(X, \begin{pmatrix} n_1 & \frac{\bar{n}}{n_1 n_d} & n_d \end{pmatrix}\right) \in \mathbf{R}^{n_1 \times \bar{n}/(n_1 n_d) \times n_d}, \quad \text{with } \bar{n} := \prod_{i=1}^{d} n_i,$$

$$X = \text{reshape}\left(Y, \begin{pmatrix} n_1 & \dots & n_d \end{pmatrix}\right).$$

This assumes that the dimensions of a tensor are ordered and provides a notation for unfolding a d-dimensional tensor into a lower-dimensional tensor, respectively, into a matrix (matricization) and folding it back into a d-dimensional tensor. It only allows us to combine neighboring dimensions, which is sufficient for all cases in this paper. In practice, many tensor algorithms can be written as series of matrix operations of different matricizations of tensors, but more general reshape operations can often be implemented without overhead by just reinterpreting the data in memory.

2.1.1. Matrix decompositions. In two dimensions, the SVD defines the (unique) decomposition of a rectangular matrix $M \in \mathbf{R}^{n_1 \times n_2}$,

$$M = U\Sigma V^T \qquad \Leftrightarrow \qquad M_{i_1,i_2} = \sum_{j=1}^{r} U_{i_1,j}\, \sigma_j\, V_{i_2,j}, \tag{2.1}$$

into the orthonormal matrices of left and right singular vectors $U \in \mathbf{R}^{n_1 \times r}$, $U^T U = I$ and $V \in \mathbf{R}^{n_2 \times r}$, $V^T V = I$ and a diagonal $\Sigma = \text{diag}(\sigma_1, \dots, \sigma_r)$ with singular values $\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_r > 0$. The decomposistion is unique if $\sigma_1 > \sigma_2 > \dots > \sigma_r > 0$. The rank of the matrix is defined as $r = \text{card}(\{\sigma_j > 0\}) \leq \min(n_1, n_2)$.

In the steps of the TT-SVD algorithm, we also use the QR decomposition

$$M = QR, \tag{2.2}$$

with an orthonormal matrix $Q \in \mathbf{R}^{n_1 \times n_2}$, $Q^T Q = I$ and an upper triangular matrix $R \in \mathbf{R}^{n_2 \times n_2}$ and $n_1 \geq n_2$.

2.1.2. TT decomposition. The TT decomposition introduced in [33] generalizes the idea of the SVD to d dimensions:

$$X_{i_1,i_2,\dots,i_d} = \sum_{j_1=1}^{r_1} \sum_{j_2=1}^{r_2} \cdots \sum_{j_{d-1}=1}^{r_{d-1}} T^{(1)}_{1,i_1,j_1}\, T^{(2)}_{j_1,i_2,j_2} \cdots T^{(d)}_{j_{d-1},i_d,1}. \tag{2.3}$$

Here, the 3D tensors $T^{(j)}$ are called "core tensors" of the decomposition and $r_1, \dots, r_{d-1}$ the ranks. In contrast to the SVD, the TT decomposition is not unique, but a best approximation with given maximal rank $r_{\max} \geq r_j$ exists, and the TT-SVD algorithm in section 3.1 calculates a quasi-optimal solution. For a detailed discussion, we refer to [33].

2.2. Performance characteristics on current hardware. Supercomputers consist of a set of compute nodes that are connected by a network (see, e.g., [21]). For the performance modeling, we concentrate on the node-level performance of the required algorithmic building blocks. However, we also show results with a distributed memory variant of the TT-SVD algorithm that allows scaling beyond a single node. Our algorithmic choices and performance optimizations are motivated by hardware characteristics of multicore processors, which we therefore briefly introduce.

Each compute node has one or several multicore CPU sockets with dedicated memory. The CPU cores can access the memory of the complete node, but accesses to the dedicated memory of the socket are faster (ccNUMA architecture). To reduce

TABLE 2.1
Hardware characteristics of a 14*-core Intel Xeon Scalable Processor* Skylake *Gold* 6132*. The data was measured using likwid-bench* [39] *(version* 5.0.1*) on a single socket of a* 4*-socket node. All memory benchmarks use nontemporal stores and AVX*512 *and an array size of* 1 *GByte. For this system, the load bandwidth is approximately twice the store bandwidth. The floating point benchmark uses AVX*512 *fused multiply-add (FMA) instructions.*

Benchmark	Measurement
memory bandwidth (pure load)	93 GByte/s
memory bandwidth (copy)	70 GByte/s
memory bandwidth (STREAM [30])	73 GByte/s
memory bandwidth (pure store)	45 GByte/s
double precision performance (AVX512 FMA)	1009 GFlop/s

the complexity of the shared memory parallelization, we use OpenMP for parallelizing over the cores of one socket and the MPI for communicating between sockets and nodes.

An important aspect of multicore optimization is the increasing gap between the memory bandwidth and the floating point performance. To alleviate this problem, multiple levels of caches are used, where the larger and slower levels are shared between multiple cores. Efficient algorithms need to exploit spatial and temporal locality (accessing memory addresses close to each other and accessing the same memory address multiple times). In addition, the floating point performance increased due to specialized wider SIMD units as well as optimized out-of-order execution of pipelined instructions. So algorithms can only achieve high performance if they contain many independent instructions for contiguous chunks of data (e.g., no data dependencies/conditional branches).

The actual run-time of a program on a specific hardware may be determined by many factors. Therefore, it is helpful to model the performance based on a few simple characteristics that are anticipated to be potential bottlenecks. For our numerical application, we use the roofline performance model [43], which considers two limiting factors. The algorithm is either compute-bound (limited by the floating point rate) or bandwidth-bound (limited by data transfers). The upper bound for the performance is thus given by

$$P_{\text{Roofline}} = \min\left(P_{\max}, I_c b_s\right). \tag{2.4}$$

Here $P_{\max}$ and b_s characterize the hardware: $P_{\max}$ denotes the applicable peak performance, that is, the maximal performance possible when executing the required floating point operations. b_s is the obtainable bandwidth on the slowest data path (e.g., from the cache or memory that is large enough to contain all data). The bandwidth depends on the access pattern, so we need to measure it with a microbenchmark that reflects the access pattern of the algorithm; see Table 2.1. The algorithm is characterized by its *compute intensity* I_c, which specifies the number of floating point operations per transferred byte. Of course, the roofline model is a simplification: in particular, it assumes that data transfers and calculations overlap, which is not realistic if the compute intensity is close to $P_{\max}/b_s$. However, the model provides insight into the behavior of the algorithm, and it allows us to assess if a specific implementation achieves a reasonable fraction of the possible performance.

To analyze an algorithm in this paper, we usually first estimate the compute intensity I_c and decide whether the algorithm is compute-bound or memory-bound (limited by main memory bandwidth) on the given hardware:

$$I_c \approx \frac{n_{\text{flops}}}{V_{\text{read+write}}}. \tag{2.5}$$

Then, we calculate the ideal run-time $t_{\min}$ from the number of floating point operations n_{flops}, respectively, from the main memory data transfer volume $V_{\text{read+write}}$:

$$t_{\min} = \begin{cases} \frac{n_{\text{flops}}}{P_{\max}} & \text{if} \quad I_c > \frac{P_{\max}}{b_s} \quad \text{(compute-bound)}, \\ \frac{V_{\text{read+write}}}{b_s} & \text{if} \quad I_c < \frac{P_{\max}}{b_s} \quad \text{(memory-bound)}, \end{cases} \tag{2.6}$$

The quotient $\frac{P_{\max}}{b_s}$ is called the *machine intensity*.

Many supercomputers nowadays also feature accelerator hardware such as general purpose graphics processing units (GPUs). We decided not to exploit GPUs in this paper because the TT-SVD accesses the complete data, which typically does not fit into the high bandwidth memory of the device. The slowest data path then is the PCI/e bus, which would make even the most optimized GPU implementation slower than our CPU code.

3. Numerical algorithms and required building blocks.

In this section we discuss different variants of the TT-SVD algorithm from [33]. We focus on algorithmic choices required for an efficient implementation on current hardware that retain numerical accuracy and robustness. As an important building block, we present a Q-less rank-preserving QR implementation for tall-skinny matrices (Q-less TSQR) based on [14].

3.1. TT-SVD.

Based on the original TT format [33], several other formats have been suggested, such as the quantized TT (QTT) format (see, e.g., [25] and the references therein) and the QTT-Tucker format [15]. These formats have interesting numerical properties; however, the required operations for calculating a high-order SVD from dense data in these formats are similar. For simplicity, we describe the algorithm for the TT format, although it is important that the individual dimensions are as small as possible (e.g., two as in the QTT format) to obtain high performance. For other hierarchical formats such as the $\mathcal{H}$-Tucker format (see, e.g., [18]), the rank is defined differently, so the complexity analysis of the algorithm is specific to the (Q)TT format. The algorithmic principles and required building blocks still remain similar for high-order decomposition algorithms for other tree tensor network formats.

3.1.1. Original TT-SVD algorithm.

We first show how the original TT-SVD algorithm from [33] can be implemented; see Algorithm 3.1 For ease of notation, we start with dimension n_d (right-most core tensor in the TT-format).

The idea of the algorithm is as follows: Each core tensor is built subsequently from the singular vectors of a truncated SVD of a matricization (first $(i-1)$ times last $(d-i+1)$ dimensions) of the input/intermediate tensor. In addition, the truncated directions are also removed from the input tensor for subsequent steps. If $r_{\max}$ is big enough, the decomposition approximates the input tensor up to the desired accuracy ϵ. Otherwise, it is less accurate (an a posteriori error bound can be calculated from the truncated singular values of each iteration). The costly operations in this algorithm are computing the SVD in line 7 and evaluating the reduced matrix W for the next iteration in line 10. And, depending on the implementation, the reshape operation in line 6 might require copying or reordering the data in memory. In this algorithm, the total size $\bar{n}$ of the work matrix W is reduced in each step by a factor $\frac{r_{i-1}}{n_i r_i} \leq 1$. And W is reshaped to very tall-skinny matrices in line 6 except for the last iterations,

Algorithm 3.1 TT-SVD.

```
Input: $X \in \mathbf{R}^{n_1\times\cdots\times n_d}$, maximal TT-rank $r_{\max} \geq 1$, tolerance $\epsilon$
Output: TT decomposition $\sum_{j_1,\dots,j_{d-1}} T^{(1)}_{1,i_1,j_1} T^{(2)}_{j_1,i_2,j_2} \cdots T^{(d)}_{j_{d-1},i_d,1} = \tilde{X}_{i_1,\dots,i_d}$ with
    $\|X - \tilde{X}\|_F \leq \epsilon\|X\|_F$ if $r_{\max} \geq r^{(i)}_\delta$
1: $\delta \leftarrow \frac{\epsilon}{\sqrt{d-1}}\|X\|_F$                                   (truncation parameter)
2: $W \leftarrow X$                                                                (temporary tensor)
3: $\bar{n} \leftarrow \prod_{i=1}^d n_i$                                                    (total size of $W$)
4: $r_d \leftarrow 1$
5: for $i = d, \dots, 2$ do
6:     $W \leftarrow$ reshape $(W, (\frac{\bar{n}}{n_i r_i} \quad n_i r_i))$
7:     Calculate SVD: $U\Sigma V^T = W$ with $\Sigma = \operatorname{diag}(\sigma_1, \dots, \sigma_{n_i r_i})$
8:     Choose rank $r_{i-1} = \min(r_{\max}, r^{(i)}_\delta)$, $r^{(i)}_\delta = \min(j : \sigma^2_{j+1} + \sigma^2_{j+2} + \cdots \leq \delta^2)$
9:     $T^{(i)} \leftarrow$ reshape $((V_{:,1:r_{i-1}})^T, \begin{pmatrix} r_{i-1} & n_i & r_i \end{pmatrix})$
10:    $\bar{n} \leftarrow \frac{\bar{n} r_{i-1}}{n_i r_i}$                                                 (new total size of $W$)
11:    $W \leftarrow U_{:,1:r_{i-1}} \operatorname{diag}(\sigma_1, \dots, \sigma_{r_{i-1}})$
12: end for
13: $T^{(1)} \leftarrow$ reshape $(W, \begin{pmatrix} 1 & n_1 & r_1 \end{pmatrix})$
```

where W is much smaller due to the reduction in size $\bar{n}$ in each step. Therefore, it is advisable to apply the QR trick for calculating the SVD:

$$W = U\Sigma V^T \qquad \Leftrightarrow \qquad W = QR,\ R = \bar{U}\Sigma V^T \text{ with } U = Q\bar{U}. \tag{3.1}$$

This idea has been discussed in the literature in similar settings (see, e.g., [11]), but we can exploit some specific details here.

One can also start the iteration in the middle of the TT by reshaping W into an (almost) square matrix of size approximately $\sqrt{\bar{n}} \times \sqrt{\bar{n}}$ and splitting it with an SVD into two independent iterations for a left and a right part. This approach is not advisable because it requires $O(\bar{n}^{\frac{3}{2}})$ floating point operations in contrast to $O(\bar{n}^{1+\frac{1}{d}})$ operations for algorithms that start at the boundaries of the TT (see section 4.2).

3.1.2. Optimized TT-SVD algorithm using TSQR.

Algorithm 3.2 is based on the original TT-SVD (Algorithm 3.1) but avoids some unnecessary computations and data transfers. It has the same numerical properties as Algorithm 3.1 if all required matrix operations are performed accurately: QR and SVD decompositions and multiplications with orthogonal matrices.

In the following, we discuss the three main differences between Algorithm 3.1 and Algorithm 3.2: First, an obvious optimization is to calculate the truncation parameter δ from Σ in the first iteration (line 7 in Algorithm 3.2). This avoids calculating the norm of the input in (line 1 of Algorithm 3.1). Second, using the QR trick (3.1), we replace the large SVD by a QR decomposition followed by a smaller SVD of the triangular factor R (line 5–6). In addition, we can use the matrix of right singular vectors V to calculate the work matrix $W^{(i-1)}$ for the next iteration (line 11 in both algorithms). This has the benefit that we never need the orthogonal factor Q of the QR decomposition which can be exploited in the implementation (see section 3.2). Third, we minimize data transfers that would be required for reshaping with appropriate padding of the data to avoid cache thrashing. So in line 11, we directly store $W^{(i-1)}$ in the desired memory layout for the next iteration. This replaces the additional

Algorithm 3.2 Optimized TSQR TT-SVD.

```
Input: $X \in \mathbf{R}^{n_1\times\cdots\times n_d}$ stored in suitable memory layout, maximal TT-rank $r_{\max} \geq 1$, tolerance $\epsilon$
Output: TT decomposition $\sum_{j_1,\dots,j_{d-1}} T^{(1)}_{1,i_1,j_1} T^{(2)}_{j_1,i_2,j_2} \cdots T^{(d)}_{j_{d-1},i_d,1} = \tilde{X}_{i_1,\dots,i_d}$
1:  $\bar{n} \leftarrow \prod_{i=1}^{d} n_i$
2:  $r_d \leftarrow 1$
3:  $W^{(d)} \leftarrow$ reshape $(X, (\frac{\bar{n}}{n_d} \quad n_d))$        (only creates a view of $X$)
4:  for $i = d, \dots, 2$ do
5:      Calculate $R$ from the QR decomposition: $QR = W^{(i)}$
6:      Calculate small SVD: $\bar{U}\Sigma V^T = R$ with $\Sigma = \mathrm{diag}(\sigma_1, \dots, \sigma_{n_i r_i})$
7:      In the first iteration, $\delta \leftarrow \frac{\epsilon}{\sqrt{d-1}}\|\Sigma\|_F$
8:      Choose rank $r_{i-1} = \min(r_{\max}, r_\delta^{(i)})$, $r_\delta^{(i)} = \min(j : \sigma_{j+1}^2 + \sigma_{j+2}^2 + \cdots \leq \delta^2)$
9:      $T^{(i)} \leftarrow$ reshape $((V_{:,1:r_{i-1}})^T, (r_{i-1} \quad n_i \quad r_i))$
10:     $\bar{n} \leftarrow \frac{\bar{n} r_{i-1}}{n_i r_i}$
11:     $W^{(i-1)} \leftarrow$ reshape $(W^{(i)} V_{:,1:r_{i-1}}, (\frac{\bar{n}}{n_{i-1} r_{i-1}} \quad n_{i-1} r_{i-1}))$
12: end for
13: $T^{(1)} \leftarrow$ reshape $(W^{(1)}, (1 \quad n_1 \quad r_1))$
```

reshape operation in line 6 of Algorithm 3.1. We assume further that the input tensor X already has a suitable memory layout such that we do not need to copy the data for the first iteration (line 3). The costly operations in this algorithm are the tall-skinny Q-less QR decomposition (line 5) and the tall-skinny matrix-matrix product fused with a reshape operation (line 11).

Memory layout. The chosen memory layout has a significant effect on the performance. A particular problem is cache thrashing (see, e.g., [21]). An example for this is shown in section 5 in Figure 5.4(b). This effect occurs for data accesses with strides that are multiples of $2^k, k \in \mathbf{N}$, with, e.g., $k > 6$. This easily happens in the TT-SVD algorithm if the individual dimensions n_i are multiples of 2, for example, when storing $W^{(i)}$ in a column-major layout. To avoid this problem, one can use padding: that means filling in a few zero entries such that the stride is not close to a multiple of 2^k (in our implementation padding is performed for all matrices such as $W^{(i)}$ to obtain strides of the form $2^6(2l+1), l \in \mathbf{N}$). In addition, the required matrix operations in the TT-SVD algorithm are memory-bound in many cases (see section 4 for a detailed discussion). That means that data locality in these operations plays a crucial role. On older multicore CPUs, a row-major memory layout in operations with tall-skinny matrices operations is favorable; see, e.g., the comparison in [37]. On newer CPUs (Intel Skylake and newer), there is no such performance penalty for using a column-major memory layout (observation of the authors). Therefore, we employ a column-major memory layout for all matrices in Algorithm 3.2. And the leading dimensions of the input tensor X are stored contiguously (Fortran ordering). As indicated above, we thus assume that the stride of the last dimension includes appropriate padding.

3.1.3. Algorithmic variants.

In the following, we discuss some interesting algorithmic variations of the TSQR TT-SVD algorithm.

Thick-bounds variant. If the dimensions n_i in the first iterations of Algorithm 3.2 are small, the required tall-skinny matrix operations become strongly memory-bound. We can increase the compute intensity by combining the right-most dimensions of the input tensor as shown in Algorithm 3.3.

Algorithm 3.3 Thick-bounds TT-SVD.

```
Input: $X \in \mathbf{R}^{n_1 \times \cdots \times n_d}$, minimal dimension $m_{\min}$, minimal reduction factor $f_1^{\min}$,
    estimated TT-ranks $\tilde{r}_i$                    (or simply use $\tilde{r}_i = r_{max}$)
Output: TT decomposition $T^{(1)}, \dots, T^{(d)}$
1: Choose #dimensions $k$ to combine minimal $k \in \{1, \dots, d\}$ with $m \geq$
   $\max(m_{\min}, f_1^{\min}\tilde{r}_{k-1})$, $m := \prod_{i=d-k+1}^{d} n_i$
2: $W \leftarrow$ reshape $(X, (n_1 \quad \cdots \quad n_{d-k} \quad m))$
3: $T^{(1)}, \dots, T^{(d-k)}, \bar{T}^{(d-k+1)} \leftarrow$ TT-SVD($W$)
4: Recover $T^{(d-k+1)}, \dots, T^{(d)}$ from the TT-SVD of $\bar{T}^{(d-k+1)}$
```

This approach allows a more efficient of the compute resources: We suggest a heuristic (line 1) based on estimated TT-ranks on a minimal combined boundary dimension ($m_{\min}$) and on a minimal estimated reduction of the work array size in the first TT-SVD iteration ($f_1^{\min}$). One can choose $m_{\min}$ such that the compute intensity of the first TSQR step is close to the machine intensity. The reduction factor is discussed in section 4.2. The TT cores corresponding to the combined dimensions can be cheaply calculated afterward (line 4).

Two-sided variant. The matrix operations in Algorithm 3.2 become more costly for increasing TT-ranks. And usually, the ranks are smaller near the left and right boundaries of the TT. So we can alternatingly calculate TT cores on the left and on the right as depicted in Algorithm 3.4. The core idea here is to reduce the size of the work array in each iteration with lower computational costs.

Algorithm 3.4 Two-sided TSQR TT-SVD.

```
Input: $X \in \mathbf{R}^{n_1 \times \cdots \times n_d}$
Output: TT decomposition $T^{(1)}, \dots, T^{(d)}$
1: $W^{(d)} \leftarrow$ reshape $(X, (\frac{\bar{n}}{n_d} \quad n_d))$          (with the total size $\bar{n}$)
2: for $i = d, 1, d-1, 2, d-2, 3, \dots$ do
3:     Calculate $R$ from the QR decomposition: $QR = W^{(i)}$
4:     Calculate small SVD: $\bar{U}\Sigma V^T = R$
5:     if $i < d/2$ then
6:         Get the new rank $r_i$ from truncating the SVD          (left case)
7:         $T^{(i)} \leftarrow$ reshape $(V_{:,1:r_i}), (r_{i-1} \quad n_i \quad r_i))$
8:         $\bar{W}^{(i)} \leftarrow W^{(i)} V_{:,1:r_i}$
9:         Reshape and transpose $\bar{W}^{(i)}$ to get $W^{(d-i)}$ for the next iteration
10:    else
11:        Get the new rank $r_{i-1}$ from truncating the SVD          (right case)
12:        $T^{(i)} \leftarrow$ reshape $((V_{:,1:r_{i-1}})^T, (r_{i-1} \quad n_i \quad r_i))$
13:        $\bar{W}^{(i)} \leftarrow W^{(i)} V_{:,1:r_{i-1}}$
14:        Transpose and reshape $\bar{W}^{(i)}$ to get $W^{(d-i+1)}$ for the next iteration
15:    end if
16: end for
17: if $d$ is even then
18:    $T^{(d/2)} \leftarrow$ reshape $(W^{(d/2)}, (r_{d/2-1} \quad n_{d/2} \quad r_{d/2}))$
19: else
20:    $T^{((d+1)/2)} \leftarrow$ reshape $((W^{((d+1)/2)})^T, (r_{(d-1)/2} \quad n_{(d+1)/2} \quad r_{(d+1)/2}))$
21: end if
```

The algorithm includes additional memory operations (line 9 and 14) to reorder data. These can be avoided by directly calculating the QR decomposition of the transposed work matrix ($(W^{(i)})^T$). However, our TSQR implementation requires a specific memory layout which makes the additional reordering necessary. It is also difficult to fuse the reordering with the preceding tall-skinny matrix multiplication efficiently due to complex index calculations. So Algorithm 3.4 illustrates our slightly suboptimal implementation. As the ideas of the thick-bounds variant and the two-sided variant are independent from each other, we can combine them. In our numerical experiments, we thus directly show timings for a two-sided algorithm with thick-bounds.

Distributed TSQR TT-SVD. We can extend Algorithm 3.2 to the case where the input tensor is distributed onto multiple compute nodes. For simplicity, we assume that the tensor is distributed along the first k dimensions and that the number of processes matches the total size of those dimensions. This is sketched in Algorithm 3.5.

Algorithm 3.5 Distributed TSQR TT-SVD.

```
Input: $X \in \mathbf{R}^{n_1\times\cdots\times n_d}$ distributed along the first $k$ dimensions $n_1 \times \cdots \times n_k$, $k \ll d$
    onto $m$ processes $j = 1, \dots, m$ with $m = \prod_{i=1}^{k} n_i$
Output: TT decomposition $T^{(1)}, \dots, T^{(d)}$ (duplicated on all processes)
1: Read local part: $W^{(j)} \leftarrow X_{i_1^{(j)},\dots,i_k^{(j)},:,\dots,:}$ on process $j$
2: $V^{(j)}, T^{(k+1)}, \dots, T^{(d)} \leftarrow$ TSQR-TT-SVD($W$)    (Algorithm 3.2 with global QR)
3: Gather $V \leftarrow \text{reshape}((V^{(1)} \quad \cdots \quad V^{(m)}), (n_1 \quad \dots \quad n_{k-1} \quad n_k r_k))$
4: Recover $T^{(1)}, \dots, T^{(k)}$ from the TT-SVD of $V$
```

The only change required for the distributed case is that the TSQR decomposition in line 5 of Algorithm 3.2 needs to perform an additional global reduction of the (local) triangular factors (see discussion in section 3.2). All other costly operations of Algorithm 3.2 are completely independent on all processes. The work for the small SVDs is duplicated on each process as well as the work for recovering the first few dimensions (line 4 of Algorithm 3.5). Of course, the assumption that the data is distributed along the first dimensions is quite restrictive. For other cases, we could first calculate the TT decomposition with reordered dimensions using this algorithm and in a postprocessing step reorder the dimensions in the TT by swapping dimensions through combining and splitting neighboring TT cores (still efficient if the TT representation is exponentially smaller than the input tensor). We can locally use the thick-bounds variant in the distributed TSQR TT-SVD. However, we cannot efficiently implement the two-sided variant in a distributed setting as the transpose operations would redistribute the data globally.

3.2. Rank-preserving Q-less TSQR algorithm.

In this section, we present our highly efficient rank-preserving TSQR decomposition based on the communication-avoiding QR factorization in [14]. The QR decomposition is rank preserving in the following sense: It does not break down if the input matrix is rank-deficient and the resulting triangular matrix R has the same (numerical) rank as the input matrix. For numerical robustness, we choose an implementation based on Householder reflections [23]. As we do not need the matrix Q in any form, its data is not even stored implicitly as in common LAPACK [3] routines to reduce the memory traffic. The core building block transforms a rectangular matrix with zero lower left triangle to upper triangular form by an orthogonal transformation Q:

$$\begin{pmatrix} M \\ R \end{pmatrix} = Q\bar{R}, \qquad \text{with } M \in \mathbf{R}^{n_b \times m}, R, \bar{R} \in R^{m\times m}. \tag{3.2}$$

Here, n_b denotes a block size that is chosen as a multiple of the SIMD width such that the data fits into the CPU caches (e.g., M and R fit into L2, multiple Householder reflectors fit into L1 for internal blocking over columns). This building block is similar to the LAPACK routine `dtpqrt2` (for the special case that M is rectangular). Our implementation differs in the following three points: First, `dtpqrt2` overwrites the input matrix M with the Householder reflection vectors. We do not modify M and store reflection vectors as long as they are needed in an internal buffer. Second, we assume a special memory layout and alignment of M and R; R is overwritten by $\bar{R}$. In contrast, LAPACK routines cope with inputs of arbitrary strides and alignment. Third, our implementation is branchless and uses fewer flops than the LAPACK reference implementation as discussed below. Based on this building block, we implement a hybrid-parallel (MPI+OpenMP) TSQR scheme. The TSQR algorithm is explained in detail in [14]. The main idea is that, with the building block above, one can calculate triangular factors for blocks of the input matrix and combine them, e.g., for a flat tree reduction,

$$\begin{pmatrix} M_1 \\ M_2 \\ M_3 \end{pmatrix} = \begin{pmatrix} Q_1R_1 \\ M_2 \\ M_3 \end{pmatrix} = \begin{pmatrix} \begin{pmatrix} & Q_1 \\ I & \end{pmatrix} \begin{pmatrix} M_2 \\ R_1 \end{pmatrix} \\ M_3 \end{pmatrix} = \begin{pmatrix} Q_{12}R_{12} \\ M_3 \end{pmatrix} = \cdots = Q_{123}R_{123}.$$

Each OpenMP thread performs a flat tree reduction (minimizing data transfers). The resulting triangular $m \times m$ matrices are combined on the master thread (negligible overhead if the number of rows of the input matrix on each thread is large). The results on multiple MPI processes are combined using an `MPI_Allreduce` operation with a commutative MPI user reduction. So the MPI library implementation decides about the actual reduction graph.

Some details of our main TSQR building block are illustrated in Algorithm 3.6.

Algorithm 3.6 Householder QR of a rectangular and a triangular matrix.

```
Input: $M \in \mathbf{R}^{n_b\times m}$, triangular $R \in \mathbf{R}^{m\times m}$, $\epsilon_{FP} > 0$
    ($\epsilon_{FP}$ is the smallest positive normalized floating point number)
Output: triangular $\bar{R} \in \mathbf{R}^{m\times m}$ that satisfies (3.2)
 1: $W_{1:n_b,:} \leftarrow (M; R)$
 2: for $j = 1, \dots, m$ do
 3:     $u \leftarrow W_{j:n_b+j,j}$                                   ($w := W_{j:n_b+j,j}$)
 4:     $t \leftarrow \|u\|_2^2 + \epsilon_{FP}$,   $\alpha \leftarrow \sqrt{t + \epsilon_{FP}}$          ($\Rightarrow \alpha^2 = \|w\|_2^2 + 2\epsilon_{FP}$)
 5:     $\alpha \leftarrow (-1)\cdot\alpha$ if $u_1 > 0$ else $1\cdot\alpha$        (implemented without branches)
 6:     $t \leftarrow t - \alpha u_1$,   $u_1 \leftarrow u_1 - \alpha$,   $\beta \leftarrow 1/\sqrt{t}$      ($\Rightarrow t = \|w\|_2^2 + \epsilon_{FP} - w_1\alpha$)
 7:     $v \leftarrow \beta u$                                         ($\Rightarrow v = (w - \alpha e_1)/\sqrt{t}$)
 8:     $W_{n_b+1:n_b+j,j} \leftarrow (W_{1:j-1,j}; \alpha)$
 9:     for $k = j+1, \dots, m$ do
10:         $\gamma \leftarrow v^T W_{j:j+n_b,k}$
11:         $W_{j:j+n_b,k} \leftarrow W_{j:j+n_b,k} - \gamma v$
12:     end for
13: end for
14: $\bar{R} \leftarrow W_{n_b+1:n_b+m,:}$
```

There are two numerical differences with respect to the LAPACK reference implementation: First, we calculate scaled Householder reflection vectors v with $\|v\|_2 = \sqrt{2}$ to avoid some additional multiplications. Second, we add the term ϵ_{FP} in line 4 to prevent a breakdown (division by zero in line 6). In contrast, the reference implementation (`dlarfg`) checks if $\|u\|_2$ is equal to zero or almost zero and performs different (expensive) steps depending on that. So our implementation avoids a conditional branch at the cost of some numerical robustness. We emphasize that, through adding ϵ_{FP} twice as in line 4, we obtain in exact arithmetic

$$\|v\|_2^2 = \frac{\|w - \alpha e_1\|_2^2}{t} = \frac{\|w\|_2^2 - 2w_1\alpha + \alpha^2}{\|w\|_2^2 + \epsilon_{FP} - w_1\alpha} = \frac{2\|w\|_2^2 - 2w_1\alpha + 2\epsilon_{FP}}{\|w\|_2^2 + \epsilon_{FP} - w_1\alpha} = 2. \tag{3.3}$$

In inexact arithmetic, this also holds approximately as long as $2\|u\|_2^2 + \epsilon_{FP}$ is in the range where the floating point arithmetic is accurate (no denormal numbers, e.g., $2\|u\|_2^2 \lesssim 10^{308}$ and $\epsilon_{FP} \approx 10^{-308}$ for double precision). So $I - vv^T$ is a valid Householder reflection even for $\|u\|_2 \approx 0$.

The actual implementation looks more complicated as it uses a recursive blocking of columns: On each recursion level, it splits the matrix into a left block and a right block and first processes the left block, then applies reflections to the right block and proceeds with the right block. This is numerically equivalent to the algorithm shown here as it only reorders the loop iterations. In addition, we avoid the copy in line 1 by just pointing to the actual data. The conditional sign flip in line 5 is compiled to floating point instructions (masked blending). Moreover, the vector operations in all iterations use vectors of the same length $(n_b + 1)$ which facilitates the SIMD parallelization.

4. Performance analysis.

In this section we first analyze the performance of the building blocks and then model the run-time of the complete TT-SVD algorithm. We assume that the dense input tensor is stored in main memory. If we read the input data from the disk, the same principles apply, but the gap between the bandwidth and the floating point performance is even larger.

4.1. Building blocks.

The main building blocks in Algorithm 3.2 are TSQR decompositions and matrix-matrix multiplications that we discuss in the following.

4.1.1. Q-less TSQR algorithm.

For $X \in \mathbf{R}^{n\times m}$ with $n \gg m$, the TSQR algorithm described in section 3.2 calculates the triangular matrix $R \in \mathbf{R}^{m\times m}$ of the QR decomposition of X. A cache-friendly implementation only reads X once from main memory ($V_{\text{read}} = 8nm$ bytes for double precision). Thus, a pure load benchmark shows the upper bound for the possible bandwidth $b_s = b_{\text{load}}$. We estimate the required floating point operations of the Householder QR reduction by considering lines 4, 7, 10, and 11 in Algorithm 3.6. We can simplify this to $\sum_{k=1}^{m}(m-k+1) = \frac{m(m+1)}{2}$ dot products and scaled vector additions (axpy) of length n_b+1. This results in $m(m+1)(n_b+1)$ fused multiply-add instructions, respectively, $2m(m+1)(n_b+1)$ floating point operations. We need to perform n/n_b such reduction steps assuming a flat TSQR reduction scheme. In practice, we perform some additional reduction steps with a different block size n_b depending on the number of OpenMP threads and MPI processes, but these are negligible for large n. Overall, we obtain

$$n_{\text{flops}} \approx \frac{n}{n_b}\left(2m(m+1)(n_b+1)\right) \approx \left(1 + \frac{1}{n_b}\right) 2nm^2 \tag{4.1}$$

$$\Rightarrow \qquad I_c = \frac{n_{\text{flops}}}{V_{\text{read}}} \approx \left(1 + \frac{1}{n_b}\right)\frac{m}{4}. \tag{4.2}$$

The compute intensity shows that the algorithm is memory-bound for m up to ~ 50 (assuming $n_b \gg 1$) on the considered hardware (see Table 2.1).

4.1.2. Tall-skinny matrix-matrix multiplication (TSMM). For matrices $X \in \mathbf{R}^{n\times m}$, $M \in \mathbf{R}^{m\times k}$, and $Y \in \mathbf{R}^{\hat{n}\times\hat{m}}$ with $n \gg m$ and $\hat{n}\hat{m} = nk$, the fused kernel for a TSMM and a reshape operation calculates

$$Y \leftarrow \text{reshape}\left(XM, \begin{pmatrix}\hat{n} & \hat{m}\end{pmatrix}\right).$$

The reshape operation just modifies the memory layout of the result and has no influence on the performance. The matrix-matrix multiplication requires $2nmk$ floating point operations and can exploit modern fused multiply-add instructions. The operation reads X ($8nm$ bytes for double precision) and writes Y ($8nk$ bytes) using nontemporal stores. The ratio of read to write volume is defined by m/k. In our experiments, we usually have $m/k \approx 2$, so we approximate the limiting bandwidth with a STREAM benchmark: $b_s = b_{\text{STREAM}}$. The resulting double precision compute intensity is $I_c = \frac{mk}{4(m+k)} \approx \frac{m}{12}$ for $m/k \approx 2$. So on the considered hardware, this operation is memory-bound for m up to ~ 150 (see Table 2.1).

4.2. Complete TT-SVD algorithm. We only analyze the optimized TSQR TT-SVD algorithm depicted in Algorithm 3.2. The analysis includes the idea of the thick-bounds variant in order to adjust algorithmic parameters.

We first consider the case that the number of columns m in the required building blocks is small enough such that they operate in the memory-bound regime (small $r_{\max}$ and small n_i). For this case, we can estimate a lower bound for the run-time by considering the data transfers in the main building blocks: One TSQR TT-SVD iteration first reads the work matrix (TSQR) and then reads it again and writes a reduced work matrix (TSMM). So for each iteration $j = 1, \dots, d-1$, we obtain the data volume: $V_{\text{read+write}} = 2\bar{n} + f_j\bar{n}$. Here, $\bar{n}$ denotes the total size of the input data of that iteration and $f_j \in (0,1]$ a reduction factor ($f_j = \frac{r_{i-1}}{n_i r_i}$ with $i = d-j+1$ in Algorithm 3.2). This is the lowest data transfer volume possible for one step in the TT-SVD algorithm if we assume that we need to consider all input data before we can compress it (*global* truncated SVD or QR decomposition). *Local* transformations are possible by, e.g., calculating truncated SVDs of blocks of the input matrix that fit into the cache and combining them later. Such a *local* approach could at best improve the performance by roughly a factor of two, as it would only read the data once instead of twice. However, this reduces the accuracy of the approximation (combining multiple *local* approximations instead of one *global* approximation for each step). For the complete TSQR TT-SVD algorithm, we sum up the data transfers of all iterations:

$$(4.3) \quad \bar{V}_{\text{read+write}} = 2\bar{n}(1 + f_1 + f_1 f_2 + \cdots) + \bar{n}(f_1 + f_1 f_2 + \cdots) \lesssim \frac{2\bar{n}}{1-\bar{f}} + \frac{\bar{f}\bar{n}}{1-\bar{f}},$$

with $1 > \bar{f} \geq f_j$ and the total size of the input tensor $\bar{n}$. To optimize data transfers, we thus need a significant reduction $f_1 \ll 1$ of the size of the work matrix in the first step. This is exactly the idea of the thick-bounds variant discussed in section 3.1.3. Overall, this indicates that small reduction factors f_j would be beneficial. However, by combining dimensions to reduce f_j in the steps of the algorithm, the compute intensity increases, and at some point the building blocks become compute-bound. For a rank-1 approximation, we can choose a small reduction factor (e.g., $\bar{f} = 1/16$ in our implementation), and for larger maximal rank, we use the choice $\bar{f} = 1/2$. This results in overall transfer volumes of

$$\bar{V}_{\text{read+write}} \lesssim \begin{cases} 2.2\bar{n} & \text{for } \bar{f} = \frac{1}{16}, \\ 5.0\bar{n} & \text{for } \bar{f} = \frac{1}{2}. \end{cases} \tag{4.4}$$

So for strongly memory-bound cases (small $r_{\max}$ and n_i), we expect a run-time in the order of the time required for copying the input tensor X (in memory).

In contrast, for larger ranks, the problem becomes compute-bound. The building blocks need approximately $2nm^2$ (TSQR), respectively, $2nmk$ (TSMM) floating point operations for an input matrix of size $n \times m$, respectively, the multiplication of an $n \times m$ with an $m \times k$ matrix. In iteration j, we have dimensions $nm = \bar{n} \prod_{l=1}^{j-1} f_j$, $m = k/f_j$, and $k = r_{i-1}$. So for the complete algorithm, we obtain with $f_j \approx \bar{f}$ and $f_j \leq r_{\max}$

$$n_{\text{flops}} \approx 2\bar{n} \left(r_{d-1} \frac{1+f_1}{f_1} + f_1 r_{d-2} \frac{1+f_2}{f_2} + \cdots \right) \lesssim 2\bar{n} r_{\max} \left(\frac{1}{\bar{f}} + \frac{2}{1-\bar{f}} \right). \tag{4.5}$$

This shows that combining more dimensions to reduce f_1 in the first step increases the work. The optimal reduction factor to minimize the number of operations is roughly $\bar{f} \approx 0.4$. With the choices for $\bar{f}$ from above, we obtain

$$n_{\text{flops}} \lesssim \begin{cases} 36\bar{n} r_{\max} & \text{for } \bar{f} = \frac{1}{16}, \\ 12\bar{n} r_{\max} & \text{for } \bar{f} = \frac{1}{2}. \end{cases} \tag{4.6}$$

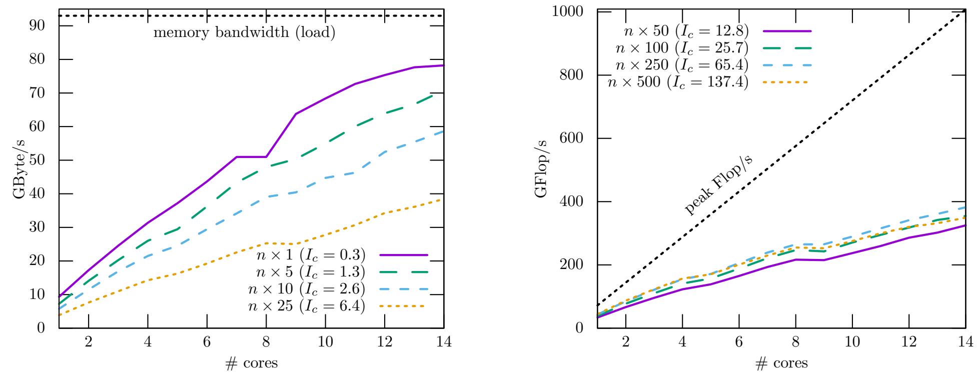
This approximation neglects the operations of the small SVD calculations of the triangular factors. So it is only valid for higher dimensions, e.g., for $\bar{n} := \prod n_i \gg (\max n_i)^3$. For the compute-bound case, we expect roughly a linear increase in run-time for increasing values of $r_{\max}$ given fixed dimensions and a fixed reduction factor $\bar{f}$ (this requires combining more dimensions). For large dimensions n_i the reduction factors become very small ($f_j \sim 1/n_i$ without splitting dimensions), and thus the computational complexity increases. In our implementation (see Algorithm 3.3), we only combine dimensions at the boundary, so we can only influence the first reduction factor f_1.

5. Numerical experiments.

In this section, we first discuss the performance of the building blocks and then consider different variants and implementations of the complete TT-SVD algorithm. We perform all measurements on a small CPU cluster; see Table 2.1 for information on the hardware. For most of the experiments, we only use a single CPU socket to avoid NUMA effects (accessing memory from another CPU socket). We implemented all required algorithms in a templated C++ library [36] based on MPI, OpenMP, and CPU SIMD intrinsics. The library includes scripts for all experiments. Comparisons of building blocks with the Intel Math Kernel Library (MKL) are written in Python using `NumPy`. We set up comparisons with other software very carefully: In particular, we ran benchmarks multiple times and ignored the first runs to avoid measuring initialization overhead. Furthermore, we checked that a high fraction of the computing time is spent in appropriate building blocks (like MKL functions) and not in some (Python) layer above (using the Linux tool `perf`). All calculations use double precision. The input data in all experiments is uniformly random, and we prescribe the dimensions, respectively, TT-ranks.

5.1. Building blocks.

The important building blocks are the Q-less TSQR algorithm and the TSMM (fused with a reshape of the result). Depending on the desired TT-rank in the TT-SVD algorithm, the number of columns m changes for the tall-skinny matrices in the building blocks. Therefore, we need to consider the performance for varying numbers of columns.



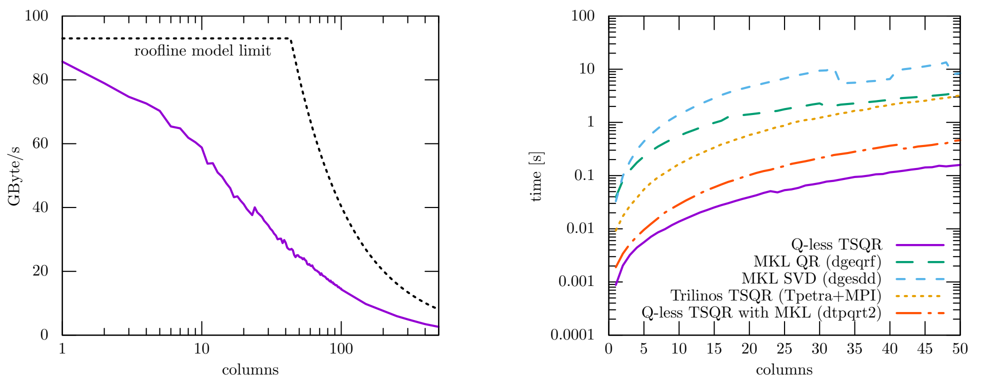
(a) Memory-bound case ($I_c \lesssim 11$). (b) Compute-bound case ($I_c \gtrsim 11$).

FIG. 5.1. *Single socket Q-less TSQR compared with the peak bandwidth, respectively, the peak Flop/s. Based on Table* 2.1*, the machine intensity for this operation (pure load) is* $1009/93 \approx 11$ *[flops/byte]. The input dimensions are chosen such that the matrix has a total size of* $\sim 3/14$ *GByte per core. The TSQR block size is* $n_b = 592$ *for* $m \lesssim 160$ *columns and then reduced linearly with* m *(e.g.,* $n_b = 192$ *for* $m = 500$*).*

5.1.1. Q-less TSQR algorithm. As analyzed in section 4.1.1, the Q-less TSQR algorithm is memory-bound for m up to ~ 50 columns on the hardware used. As we do not store the Q matrix of the TSQR decomposition, its run-time is limited by the load memory bandwidth. We expect a saturating behavior of the measured bandwidth up to the peak load bandwidth on 1–14 cores. However, in Figure 5.1(a) we see that the bandwidth is not fully saturated on 14 cores except for the case $n \times 1$. So our implementation partly seems to be limited by the in-core performance even for the memory-bound cases. This effect increases with the number of columns, respectively, with the compute intensity. This indicates that our implementation is still suboptimal. In addition, the simple roofline model based on the number of floating point operations is too optimistic for this case because the TSQR algorithm includes data dependencies as well as costly sqrt operations. Overall we obtain more than 50% of the peak bandwidth for small numbers of columns.

For the compute-bound case ($m \geq 50$ on this hardware), we observe the expected linear scaling with the number of cores (see Figure 5.1(b)). Our implementation achieves roughly 35% of the peak performance here, independent of the number of columns.

Figure 5.2(a) shows the obtained bandwidth on a full socket and the roofline limit depending on the number of columns m. The kink in the roofline limit denotes the point where the operation (theoretically) becomes compute-bound. We see that the obtained bandwidth of our implementation decreases with the number of columns even in the memory-bound regime. However, our specialized TSQR implementation is still significantly faster than just calling some standard QR algorithm that is not optimized for tall-skinny matrices. This is illustrated by Figure 5.2(b). The comparison with MKL QR is fair concerning the hardware setting (single socket with 14 cores, no NUMA effects). However, it is unfair from the algorithmic point of view because we can directly discard Q and exploit the known memory layout, whereas the MKL QR algorithm must work for all matrix shapes and any given memory layout and strides. We also show the run-time of the MKL SVD calculation for the same



(a) Obtained memory bandwidth of our Q-less TSQR implementation compared to the roofline limit: $b_{\text{roofline}} = \min(P_{\text{peak}}/I_c, b_s)$.

(b) Run-time for the QR decomposition (respectively a SVD) of a double precision $10^7 \times m$ matrix for $m = 1, \dots, 50$ with our Q-less TSQR implementation, Intel MKL 2020.3, and Trilinos 13.0.0.

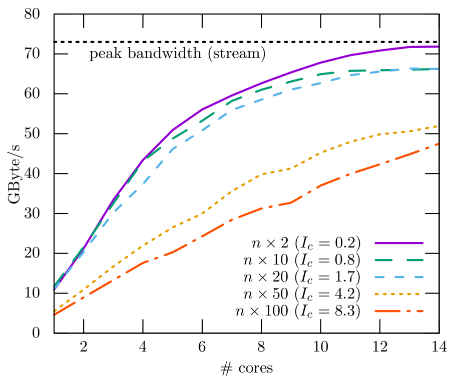
FIG. 5.2. *Single socket performance of tall-skinny matrix decompositions for varying numbers of columns.*

matrix dimensions. Calculating the singular values and the right singular vectors from the resulting R of the TSQR algorithm requires no significant additional time (SVD of $m \times m$ matrix with small m). In addition, we measured the run-time of the Trilinos [40] TSQR algorithm with the Trilinos Tpetra library on one MPI process per core. The Trilinos TSQR algorithm explicitly calculates the matrix Q, and it does not seem to exploit SIMD parallelism: We only obtained scalar fused multiply-add instructions instead of AVX512 (GCC 10.2 compiler with appropriate flags). Due to these two reasons, the Trilinos TSQR is still slower than our almost optimal Q-less TSQR implementation by more than a factor of 10. Finally, we replaced our implementation for reducing a triangular and a rectangular factor to triangular form in our Q-less TSQR implementation by the according low-level MKL routine `dtpqrt2`. In this case, we need to copy a block of the input matrix to a small buffer to avoid overwriting the input matrix (overhead of less than $\sim 10\%$ of the total time). This variant achieves about 1/3 of the performance of our specialized branchless Householder QR implementation. Overall, the QR trick with our Q-less TSQR implementation reduces the run-time of the SVD calculation by roughly a factor of 50 compared to just calling standard LAPACK (MKL) routines.

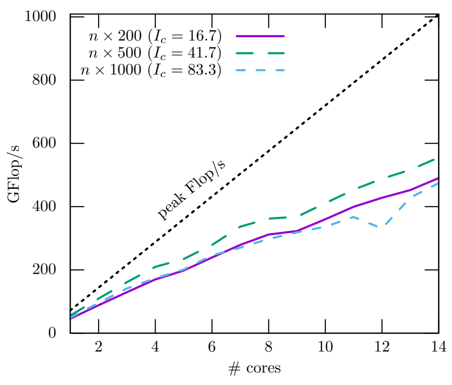
5.1.2. TSMM. As analyzed in section 4.1.2, the fused TSMM and reshape is also memory-bound for m up to ~ 150 columns on the given hardware.

Figure 5.3(a) shows the obtained bandwidth for varying numbers of cores. We observe a saturation of the memory bandwidth for $m < 50$. For $m = 100$, we already see a linear scaling with the number of cores. For the compute-bound case, our implementation roughly obtains 50% of the peak performance (see Figure 5.3(b)).

From Figure 5.4(a), we conclude that our TSMM implementation obtains a high fraction of the maximum possible bandwidth. Near the kink, the roofline model is too optimistic because it assumes that data transfers and floating point operations overlap perfectly. Further insight could be obtained by a more sophisticated performance

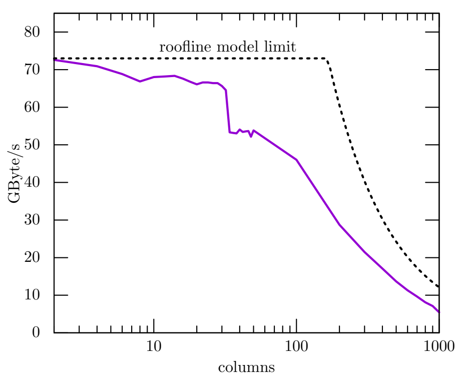


(a) Memory-bound case ($I_c \lesssim 14$) measured with $n = 10^7$.

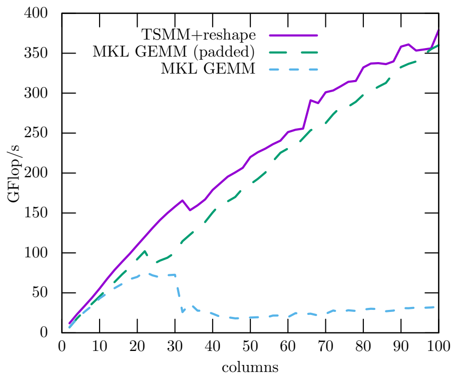


(b) Compute-bound case ($I_c \gtrsim 14$) measured with $n = 5 \cdot 10^6$.

FIG. 5.3. *Single socket TSMM+reshape compared with the peak bandwidth, respectively, peak flop/s. The input matrices have dimensions $n \times m$ and $m \times m/2$; the result is reshaped to $n/2 \times m$. Based on Table* 2.1*, the machine intensity for this operation (load/store ratio of* $2/1 \,\hat{=}\,$ *STREAM) is* $1009/73 \approx 14$ *[flops/byte].*



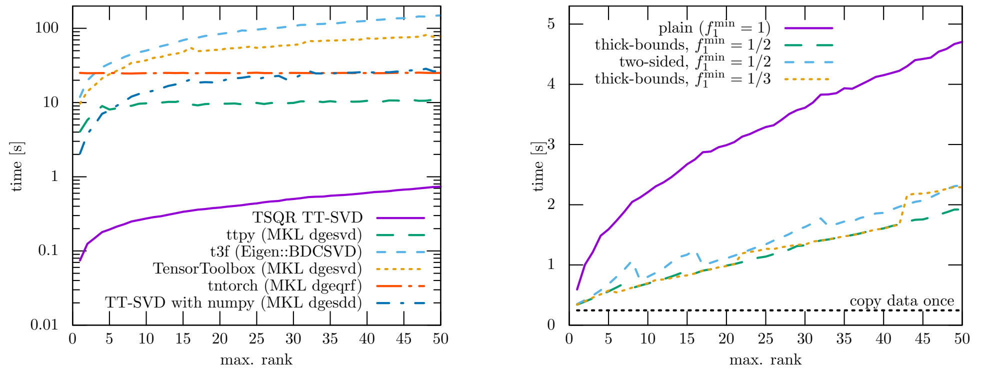
(a) Obtained memory bandwidth of our TSMM+reshape implementation compared to the roofline limit: $b_{\text{roofline}} = \min(P_{\text{peak}}/I_c, b_s)$.



(b) Obtained performance compared to the Intel MKL 2020.3 for $n = 2^{24}$ rows with and without padding to $n^{24} + 64$.

FIG. 5.4. *Single socket performance of TSMM+reshape for varying numbers of columns. The input matrices have dimensions $n \times m$ and $m \times m/2$, and our implementation directly stores the result reshaped to dimensions $n/2 \times m$.* (b) *illustrates the effect of cache thrashing (the leading array dimension is a power of two).*

model such as the execution-cache-memory model; see [38]. For this operation, our implementation and the Intel MKL obtain roughly the same performance, as depicted in Figure 5.4. In contrast to the MKL, our implementation exploits a special memory layout, which might explain the small differences in run-time. So the advantage of our TSMM implementation for the complete TT-SVD algorithm consists mainly in fusing the reshape operation, which ensures a suitably padded memory layout for subsequent



(a) Different implementations of the classical TT-SVD algorithm for a 2^{27} tensor and our TSQR TT-SVD algorithm (without combining dimensions).

(b) Algorithmic variants of TSQR TT-SVD for a 2^{30} tensor with different prescribed reduction factors $f_1^{\min}$.

Fig. 5.5. *Single socket run-time of different TT-SVD algorithms for varying maximal TT-rank.*

operations at no additional cost. Without appropriate padding, the performance can degrade significantly due to cache thrashing (also illustrated in Figure 5.4), in particular for operations from tensor algorithms when individual dimensions are multiples of two.

5.2. TT-SVD. In the following, we consider the complete TT-SVD algorithm and different variants and implementations of it. Figure 5.5(a) illustrates the run-time of the TT-SVD algorithm in different software libraries. All cases show the run-time for decomposing a random double precision 2^{27} tensor on a single CPU socket with 14 cores with a prescribed maximal TT-rank. For several of these libraries, we tested different variants and LAPACK back ends [3, 24]. Here, we only report the timings for the fastest variant that we could find. We show results for the following libraries:

- **TSQR TT-SVD:** The implementation discussed in this paper.
- **ttpy:** A library written in Fortran and Python by the author of [33].
- **t3f:** A library based on the TensorFlow framework [31].
- **TensorToolbox:** A Python library from the author of [7].
- **tntorch:** A library based on PyTorch [5].
- **TT-SVD with `NumPy`:** Simple implementation in `NumPy` [22] inspired by [17].

Both `ttpy` and `TensorToolbox` use the older (and in many cases slower) `dgesvd` routine for calculating SVD decompositions. Our classical TT-SVD implementation with `NumPy` uses the newer LAPACK routine `dgesdd`. The `ttpy` library is still faster in many cases. The `t3f` library is based on TensorFlow, which is optimized for GPUs. It uses the C++ library Eigen [20] as the back end on CPUs. However, only some routines in Eigen are parallelized for multicore CPUs which explains why `t3f` is slow here. In contrast to all other variants, the run-time of the TT decomposition in `tntorch` is almost independent of the maximal TT-rank. `tntorch` does not implement the TT-SVD algorithm but instead first constructs a TT of maximal rank, followed by a left-right orthogonalization step and TT rounding. The computationally costly part is the left-right orthogonalization step, which is based on QR decompositions whose size only depends on the size of the input tensor and not on the desired rank.

Table 5.1

Examples for the resulting effective dimensions and TT-ranks for the different TT-SVD variants for a 2^d tensor. We consider the right-most dimensions and ranks as our implementation calculates the decomposition from right to left.

Case	$r_{\max}$	Effective dim. (n_i)	TT-ranks (r_i)	Reduction factors (f_j)
plain	1	$\ldots, 2, 2$	$\ldots, 1, 1$	$\frac{1}{2}, \frac{1}{2}, \ldots$
$(f_1^{\min} = 1)$	5	$\ldots, 2, 2$	$\ldots, 5, 5, 4, 2$	$1, 1, \frac{5}{8}, \frac{1}{2}, \ldots$
	16	$\ldots, 2, 2$	$\ldots, 16, 16, 8, 4, 2$	$1, 1, 1, 1, \frac{1}{2}, \ldots$
thick-bounds	1	$\ldots, 2, 2, 16$	$\ldots, 1, 1$	$\frac{1}{16}, \frac{1}{2}, \frac{1}{2}, \ldots$
$(f_1^{\min} = 1/2)$	5	$\ldots, 2, 2, 16$	$\ldots, 5, 5$	$\frac{5}{16}, \frac{1}{2}, \frac{1}{2}, \ldots$
	16	$\ldots, 2, 2, 32$	$\ldots, 16, 16$	$\frac{1}{2}, \frac{1}{2}, \ldots$

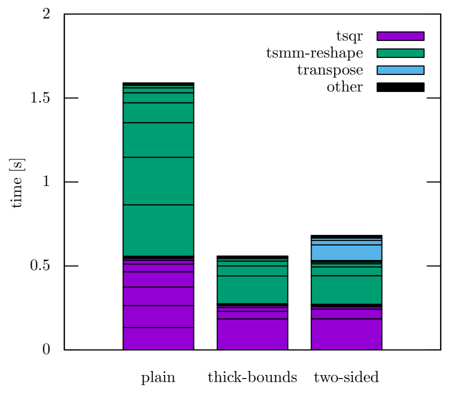
Our TSQR TT-SVD implementation is significantly faster than all other implementations for two reasons. First, there are multiple combinations of basic linear algebra building blocks that calculate the desired result. This is an example of the linear algebra mapping problem as discussed in [35]. Here, we choose a combination of building blocks (Q-less TSQR + multiplication with truncated right singular vectors) that leads to (almost) minimal data transfers. Second, common linear algebra software and algorithms are not optimized for avoiding data transfers. However, for the tall-skinny matrix operations required here, the data transfers determine the performance. For a detailed overview on communication avoiding linear algebra algorithms, see, e.g., [4] and the references therein. An interesting discussion that distinguishes between the effects of reading and modifying data can be found in [8].

Figure 5.5(b) shows the run-time for the different variants of the TSQR TT-SVD algorithm discussed in section 3.1.3. This is the worst case run-time of the algorithm because we approximate a random input matrix and we only prescribe the maximal TT-rank. For the plain case ($f_1^{\min} = 1$), there is no reduction in the data size in the first steps for $r_{\max} > 1$. For the thick-bounds and two-sided variants we set $m_{\min} = 16$ (see Algorithm 3.4). This reduces the run-time for small TT-ranks (difference between plain and other variants for $r_{\max} = 1$). See Table 5.1 for some examples on resulting dimensions and TT-ranks.

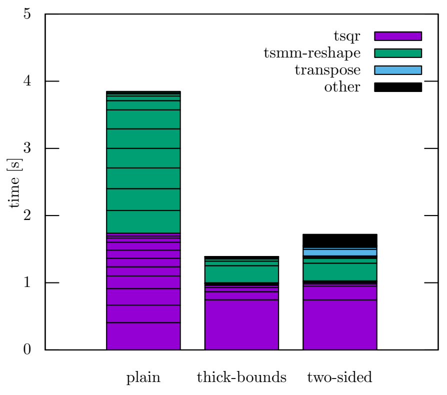
As expected, the plain variant is slower, as it needs to transfer more data in the first iterations. For all cases with a prescribed reduction $f_1^{\min} < 1$, we observe roughly a linear scaling with the maximal TT-rank as predicted by the performance analysis for the compute-bound case. And for small ranks, the run-time is of the order of copying the data in memory. For our implementation the choice $f_1^{\min} = 1/2$ appears to be optimal even though the theoretical analysis indicates that a smaller $f_1^{\min}$ could be beneficial. Decreasing $f_1^{\min}$ increases the number of columns of the matrices in the first step. This leads to more work, and the obtained bandwidth of our TSQR implementation decreases (see Figure 5.2(a)). The two-sided variant uses thick-bounds as well, but it is always slower with our implementation.

The run-time of the individual steps of the algorithm are illustrated in Figure 5.6. We clearly see the effect of combining multiple dimensions: The first TSQR step takes longer, but all subsequent steps are faster. The two-sided variant is only slower due to the additional transpose operation required in our implementation. For real-world problems, the two-sided variant might be faster depending on the resulting TT-ranks.

To validate our assumptions in the performance analysis in section 4.2, we measured data transfers and flops for several cases using CPU performance counters; see Table 5.2. We compare cases where the simple estimates with the global reduction factor $\bar{f}$ fit well, and we observe a good correlation with the measurements. Depending



(a) $r_{\max} = 5$.



(b) $r_{\max} = 35$.

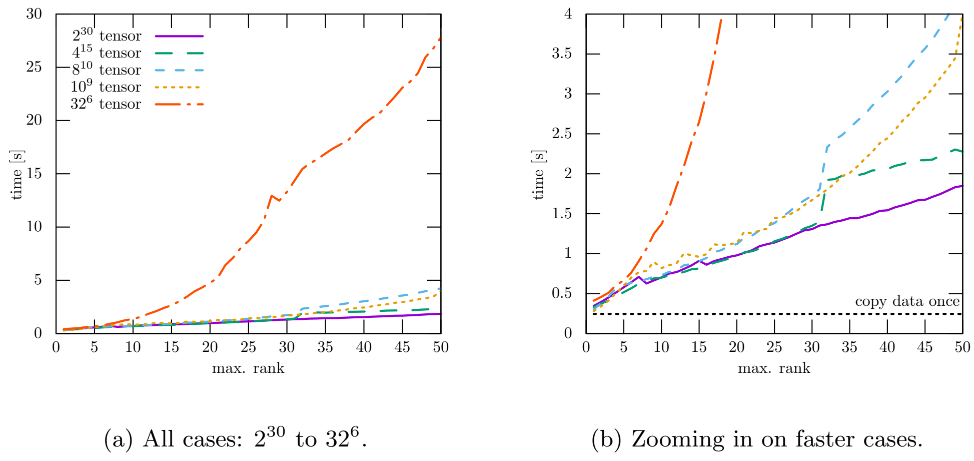
FIG. 5.6. *Timings for the building blocks in different TT-SVD variants for two cases from Figure* 5.5(b). *The* transpose *timings refer to the transpose/reshape operations in lines* 9 *and* 14 *of Algorithm* 3.4.

TABLE 5.2
Measured and estimated number of floating point operations and transferred data volume between the CPU and the main memory for the TSQR TT-SVD algorithm with a 2^{30} tensor in double precision. The measurements were performed with likwid-perfctr [39]. *Estimates based on* (4.3) *and* (4.5) *are shown in parentheses.*

Case	$r_{\max}$	Operations GFlop	Data transfers GByte
plain (estimate with $\bar{f} = 1/2$)	1	14 (13)	43 (43)
thick-bounds (estimate with $\bar{f} = 1/16$)	1	41 (39)	21 (19)
thick-bounds (estimate with $\bar{f} = 1/2$)	31	417 (399)	43 (43)

on the dimensions and the desired maximal rank, the reduction in the first step can differ from the following steps (see Table 5.1) which is not captured by (4.3) and (4.5).

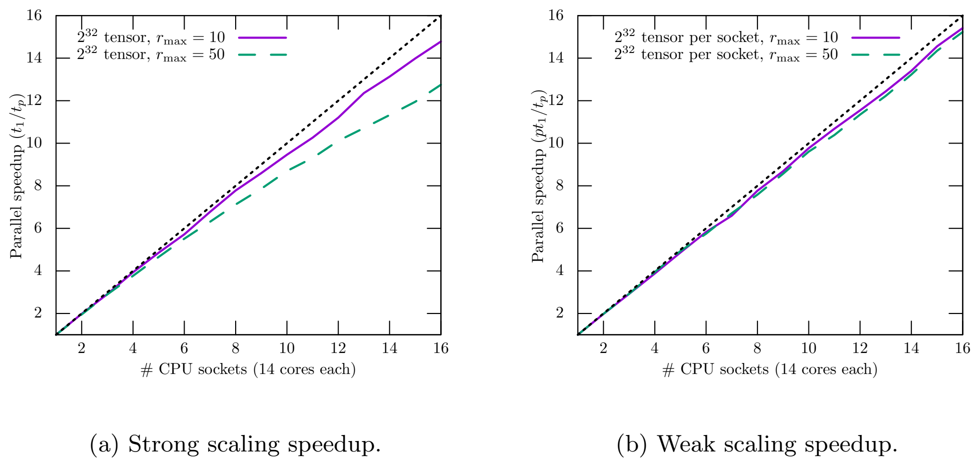
The experiments above use 2^d tensors for simplicity (as in the QTT format). If we increase the size of the individual dimensions, the compute intensity of the TSQR TT-SVD algorithm increases. Figure 5.7 visualizes the run-time for decomposing tensors of different dimensions with approximately the same total size. For very small maximal rank ($r_{\max} < 5$), all cases require similar run-time. For higher maximal ranks, the cases with a higher individual dimension become more costly. Near $r_{\max} = 32$ there are some interesting deviations in the run-time from the expected linear growth. We can explain these deviations by the possible choices for combining dimensions in the thick-bounds algorithm: Depending on $r_{\max}$ and n_i there are only a few discrete choices for the number of columns m of the first step. In particular, we obtain $m = 100$ for $r_{\max} = 10, \dots, 49$ for the 10^9 tensor but $m = 512$ for $r_{\max} = 32, \dots, 255$ for the 8^{10} tensor with a prescribed minimal reduction $f_1^{\min} = 1/2$. This results in a lower run-time for the 10^9 tensor as the first step is the most costly part of the algorithm. As expected, the run-time of the 32^6 case increases linearly with the maximal rank for $r_{\max} \geq 16$, and the run-time is significantly higher than for smaller dimensions as the resulting reduction factors are small ($f_j \approx 1/32$).



(a) All cases: 2^{30} to 32^6.

(b) Zooming in on faster cases.

FIG. 5.7. *Timings for TSQR TT-SVD for varying dimensions on a single socket. Uses the thick-bounds variant with $f_1^{min} = \frac{1}{2}$ where beneficial.*



(a) Strong scaling speedup.

(b) Weak scaling speedup.

FIG. 5.8. *Speedup for the TSQR TT-SVD (thick-bounds variant with $f_1^{min} = \frac{1}{2}$) on a varying number of CPU sockets and nodes with one MPI process per socket. Each node has 4 sockets with 14 cores. The reference time is measured on a single socket.*

Finally, we also tested the distributed variant of the TSQR TT-SVD algorithm using MPI. Figure 5.8 shows strong and weak scaling results for input tensors of dimension 2^{32} (strong scaling) and 2^{32} to 2^{36} (weak scaling). We observe a good weak scaling behavior (parallel efficiency of about $\sim 95\%$). The biggest considered case has an input tensor of size 2^{36} (~ 550 GByte). For strong scaling, the problem size per CPU socket gets smaller. So in particular for bigger TT-ranks, the relative overhead due to duplicating the work of the small SVD increases. The same holds for the relative parallelization overhead in the TSQR algorithm. The TSQR MPI reduction only amounts to about 3% of the overall run-time (with 16 CPU sockets and $r_{\max} = 50$, similar for both strong and weak scaling). Summing up, the distributed variant allows to tackle problems where the dense input tensor is too large for the memory of a single node or where the input tensor is generated by a distributed program on a cluster. The communication overhead is low. Only for strong scaling, we observe a significant overhead due to nonparallelized parts of the algorithm.

6. Conclusion and future work.

In this paper we analyzed the node-level performance of the TT-SVD algorithm that calculates a low-rank approximation of a high-dimensional tensor. The results can also be transferred to other nonrandomized high-order SVD algorithms. We considered the case where the input tensor is large and dense but not too large to be processed completely, i.e., to be read from main memory or disk as a whole. The theoretical minimal run-time depends on the desired accuracy of the approximation. For small TT-ranks (low accuracy), the algorithm is memory-bound, and the ideal run-time on current hardware is approximately twice the time required for reading the data (transferring it from the memory to the CPU). For larger TT-ranks (higher accuracy), the algorithm becomes compute-bound, and the ideal run-time increases linearly with the maximal TT-rank. We presented different variants of the TT-SVD algorithm. In order to reduce the computational complexity, these variants start with the calculation of the TT-cores at the boundaries of the TT and reduce the data size in each step. The key ingredient is a Q-less TSQR decomposition based on Householder reflections that handles rank-deficient matrices without pivoting by clever use of floating point arithmetic. We performed numerical experiments with 2^d tensors of size up to 550 GByte ($d = 36$) on up 224 cores on a small cluster. Our hybrid-parallel (MPI+OpenMP) TT-SVD implementation achieves almost optimal run-time for small ranks and about 25% peak performance for larger TT-ranks. On a single CPU socket, our implementation is about $50\times$ faster compared to TT-SVD algorithms in other libraries. We provide a lower bound for the run-time: reading the data twice from main memory. This also indicates that randomized algorithms can cirumvent this lower bound by not considering all data.

For future work, we see three interesting directions: First, here, we use random input data and prescribe the TT-ranks. In real applications, usually a certain truncation accuracy is prescribed instead, and the TT-ranks depend on the desired accuracy. For optimal performance one needs to combine, rearrange, or split dimensions based on some heuristic such that the first step leads to a sufficient reduction in data size. Second, we only analyzed one TT operation for dense input. Similar performance gains might be possible for other important operations involving large dense data. Handling sparse input data efficiently is more challenging, as the reduction in dimensions in each step does not necessarily lead to a reduction in data size. And finally, it would be interesting to analyze the performance of randomized decomposition algorithms and to deduce lower bounds for their run-time on current hardware.

REFERENCES

[1] I. Affleck, *Large-n limit of* SU(n) *quantum "spin" chains,* Phys. Rev. Lett., 54 (1985), pp. 966–969, https://doi.org/10.1103/PhysRevLett.54.966.

[2] I. Affleck, T. Kennedy, E. H. Lieb, and H. Tasaki, *Rigorous results on valence-bond ground states in antiferromagnets*, Phys. Rev. Lett., 59 (1987), pp. 799–802, https://doi.org/10.1103/PhysRevLett.59.799.

[3] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*, SIAM, Philadelphia, 1999, https://doi.org/10.1137/1.9780898719604.

[4] G. Ballard, E. Carson, J. Demmel, M. Hoemmen, N. Knight, and O. Schwartz, *Communication lower bounds and optimal algorithms for numerical linear algebra*, Acta Numer., 23 (2014), pp. 1–155, https://doi.org/10.1017/s0962492914000038.

[5] R. Ballester-Ripoll, *tntorch - Tensor Network Learning with PyTorch*, revision 8c8lalcb, 2019, https://tntorch.readthedocs.io.

[6] M. W. Berry, S. A. Pulatova, and G. W. Stewart, *Algorithm* 844: Computing Sparse reduced-rank approximations to Sparse Matrices, ACM Trans. Math. Software, 31 (2005), pp. 252–269, https://doi.org/10.1145/1067967.1067972.

[7] D. Bigoni, A. P. Engsig-Karup, and Y. M. Marzouk, *Spectral tensor-train decomposition*, SIAM J. Sci. Comput., 38 (2016), pp. A2405–A2439, https://doi.org/10.1137/15m1036919.

[8] E. Carson, J. Demmel, L. Grigori, N. Knight, P. Koanantakool, O. Schwartz, and H. V. Simhadri, *Write-avoiding algorithms*, in Proceedings of the 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE, 2016, pp. 648–658, https://doi.org/10.1109/ipdps.2016.114.

[9] C. Chen, K. Batselier, C.-Y. Ko, and N. Wong, *A support tensor train machine*, in Proceedings of the 2019 International Joint Conference on Neural Networks (IJCNN), IEEE, 2019, https://doi.org/10.1109/ijcnn.2019.8851985.

[10] C. Chen, K. Batselier, W. Yu, and N. Wong, *Kernelized support tensor train machines*, Pattern Recogn., 122 (2022), p. 108337, https://doi.org/10.1016/j.patcog.2021.108337.

[11] P. G. Constantine, D. F. Gleich, Y. Hou, and J. Templeton, *Model reduction with MapReduce-enabled tall and skinny singular value decomposition*, SIAM J. Sci. Comput., 36 (2014), pp. S166–S191, https://doi.org/10.1137/130925219.

[12] H. A. Daas, G. Ballard, and P. Benner, *Parallel algorithms for tensor train arithmetic*, SIAM J. Sci. Comput., 44 (2022), pp. C25–C53, https://doi.org/10.1137/20m1387158.

[13] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou, *Communication-optimal parallel and sequential QR and LU factorizations*, SIAM J. Sci. Comput., 34 (2012), pp. A206–A239, https://doi.org/10.1137/080731992.

[14] J. W. Demmel, L. Grigori, M. Gu, and H. Xiang, *Communication avoiding rank revealing QR factorization with column pivoting*, SIAM J. Matrix Anal. Appl., 36 (2015), pp. 55–89, https://doi.org/10.1137/13092157x.

[15] S. Dolgov and B. Khoromskij, *Two-level QTT-Tucker format for optimized tensor calculus*, SIAM J. Matrix Anal. Appl., 34 (2013), pp. 593–623, https://doi.org/10.1137/120882597.

[16] H.-Y. Fan, L. Zhang, E. w. Chu, and Y. Wei, *Q-less QR decomposition in inner product spaces*, Linear Algebra Appl., 491 (2016), pp. 292–316, https://doi.org/10.1016/j.laa.2015.08.035.

[17] P. Gelss, S. Klus, M. Scherer, F. Nüske, and M. Lücke, *Scikit-TT Tensor-Train Computations in Python*, revision idfd64a, 2019, https://github.com/PGelss/scikit_tt, 2019, revision 1dfd64a.

[18] L. Grasedyck and W. Hackbusch, *An introduction to hierarchical (H-) rank and TT-rank of tensors with examples*, Comput. Methods Appl. Math., 11 (2011), pp. 291–304, https://doi.org/10.2478/cmam-2011-0016.

[19] L. Grasedyck, D. Kressner, and C. Tobler, *A literature survey of low-rank tensor approximation techniques*, GAMM-Mitt., 36 (2013), pp. 53–78, https://doi.org/10.1002/gamm.201310004.

[20] G. Guennebaud, B. Jacob, et al., *Eigen v*3, version 3.3.9, 2010, http://eigen.tuxfamily.org.

[21] G. Hager and G. Wellein, *Introduction to High Performance Computing for Scientists and Engineers*, CRC Press, Boca Raton, FL, 2010, https://doi.org/10.1201/ebk1439811924.

[22] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, *Array programming with NumPy*, Nature, 585 (2020), pp. 357–362, https://doi.org/10.1038/s41586-020-2649-2.

[23] A. S. Householder, *Unitary triangularization of a nonsymmetric matrix*, J. ACM, 5 (1958), pp. 339–342, https://doi.org/10.1145/320941.320947.

[24] Intel, one API *Math Kernel Library*, version 2020.3, https://software.intel.com/mkl.

[25] B. N. Khoromskij, *$O(d \log N)$-quantics approximation of N-d tensors in high-dimensional numerical modeling*, Constr. Approx., 34 (2011), pp. 257–280, https://doi.org/10.1007/s00365-011-9131-1.

[26] S. Klus and P. Gelss, *Tensor-based algorithms for image classification*, Algorithms, 12 (2019), 240, https://doi.org/10.3390/a12110240.

[27] T. G. Kolda and D. Hong, *Stochastic gradients for large-scale tensor decomposition*, SIAM J. Math. Data Sci., 2 (2020), pp. 1066–1095, https://doi.org/10.1137/19m1266265.

[28] K. Kour, S. Dolgov, M. Stoll, and P. Benner, *Efficient Structure-Preserving Support Tensor Train Machine*, preprint, arXiv:2002.05079, 2020, https://arxiv.org/abs/2002.05079.

[29] B. W. Larsen and T. G. Kolda, *Practical Leverage-Based Sampling for Low-Rank Tensor Decomposition*, preprint, arXiv:2006.16438, 2020, https://arxiv.org/abs/2006.16438.

[30] J. D. McCalpin, *Memory bandwidth and machine balance in current high performance computers*, IEEE Computer Society Technical Committee on Computer Architecture Newsletter, 2 (1995), pp. 19–25.

[31] A. Novikov, P. Izmailov, V. Khrulkov, M. Figurnov, and I. Oseledets, *Tensor train decomposition on TensorFlow (T3F)*, J. Mach. Learn. Res., 21 (2020), pp. 1–7, http://jmlr.org/papers/v21/18-008.html.

[32] I. V. Oseledets, *A new tensor decomposition*, Dokl. Math., 80 (2009), pp. 495–496, https://doi.org/10.1134/S1064562409040115.

[33] I. V. Oseledets, *Tensor-train decomposition*, SIAM J. Sci. Comput., 33 (2011), pp. 2295–2317, https://doi.org/10.1137/090752286.

[34] I. V. Oseledets and E. E. Tyrtyshnikov, *Breaking the curse of dimensionality, or how to use SVD in many dimensions*, SIAM J. Sci. Comput., 31 (2009), pp. 3744–3759, https://doi.org/10.1137/090748330.

[35] C. Psarras, H. Barthels, and P. Bientinesi, *The Linear Algebra Mapping Problem. Current State of Linear Algebra Languages and Libraries*, preprint, arXiv:1911.09421, 2019, https://arxiv.org/abs/1911.09421.

[36] M. Röhrig-Zöllner, *PITTS - parallel iterative tensor-train solvers*, 2021, https://doi.org/10.5281/zenodo.5534544.

[37] M. Röhrig-Zöllner, J. Thies, M. Kreutzer, A. Alvermann, A. Pieper, A. Basermann, G. Hager, G. Wellein, and H. Fehske, *Increasing the performance of the Jacobi–Davidson method by blocking*, SIAM J. Sci. Comput., 37 (2015), pp. C697–C722, https://doi.org/10.1137/140976017.

[38] H. Stengel, J. Treibig, G. Hager, and G. Wellein, *Quantifying performance bottlenecks of stencil computations using the execution-cache-memory model*, in Proceedings of the 29th ACM on International Conference on Supercomputing, ACM Press, 2015, https://doi.org/10.1145/2751205.2751240.

[39] J. Treibig, G. Hager, and G. Wellein, *LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments*, in Proceedings of the 2010 39th International Conference on Parallel Processing Workshops, IEEE, 2010, https://doi.org/10.1109/icppw.2010.38.

[40] T. Trilinos Project Team, *The Trilinos Project Website*, 2020, https://trilinos.github.io, accessed May 22, 2020.

[41] F. Verstraete and J. I. Cirac, *Matrix product states represent ground states faithfully*, Phys. Rev. B, 73 (2006), 094423, https://doi.org/10.1103/PhysRevB.73.094423.

[42] S. R. White, *Density matrix formulation for quantum renormalization groups*, Phys. Rev. Lett., 69 (1992), pp. 2863–2866, https://doi.org/10.1103/PhysRevLett.69.2863.

[43] S. Williams, A. Waterman, and D. Patterson, *Roofline: An insightful visual performance model for multicore architectures*, Comm. ACM, 52 (2009), pp. 65–76, https://doi.org/10.1145/1498765.1498785.