Design of a high-performance tensor-matrix multiplication with BLAS

Cem Savaş Başsoy[a,*]

[a]*Hamburg University of Technology, Schwarzenbergstrasse 95, 21071, Hamburg, Germany*

Abstract

The tensor-matrix multiplication is a basic tensor operation required by various tensor methods such as the HOSVD. This paper presents flexible high-performance algorithms that compute the tensor-matrix product according to the Loops-over-GEMM (LOG) approach. Our algorithms can process dense tensors with any linear tensor layout, arbitrary tensor order and dimensions all of which can be runtime variable. The paper discusses two slicing methods with orthogonal parallelization strategies and propose four algorithms that call BLAS with subtensors or tensor slices. It also provides a simple heuristic which selects one of the four proposed algorithms at runtime. All algorithms have been evaluated on a large set of tensors with various tensor shapes and linear tensor layouts. In case of large tensor slices, our best-performing algorithm achieves a median performance of 2.47 TFLOPS on an Intel Xeon Gold 5318Y and 2.93 TFLOPS an AMD EPYC 9354. Furthermore, it outperforms batched GEMM implementation of Intel MKL by a factor of 2.57 with large tensor slices. Our runtime tests show that our best-performing algorithm is, on average, at least 6.21% and up to 334.31% faster than frameworks implementing state-of-the-art approaches, including actively developed libraries such as Libtorch and Eigen. For the majority of tensor shapes, it is on par with TBLIS which uses optimized kernels for the TTM computation. Our algorithm performs better than all other competing implementations for the majority of real world tensors from the SDRBench, reaching a maximum speedup of 100.80% or more in some tensor instances. This work is an extended version of the article "Fast and Layout-Oblivious Tensor-Matrix Multiplication with BLAS" (Bassoy, 2024)[1].

1. Introduction

Tensor computations are found in many scientific fields such as computational neuroscience, pattern recognition, signal processing and data mining [2, 3, 4, 5, 6]. These computations use basic tensor operations as building blocks for decomposing and analyzing multidimensional data which are represented by tensors [7, 8]. Tensor contractions are an important subset of basic operations that need to be fast for efficiently solving tensor methods.

There are three main approaches for implementing tensor contractions. The Transpose Transpose GEMM Transpose (TGGT) approach reorganizes tensors in order to perform a tensor contraction using optimized implementations of the general matrix multiplication (GEMM) [9, 10]. GEMM-like Tensor-Tensor multiplication (GETT) method implement macro-kernels that are similar to the ones used in fast GEMM implementations [11, 12]. The third method is the Loops-over-GEMM (LOG) or the BLAS-based approach in which Basic Linear Algebra Subprograms (BLAS) are utilized with multiple tensor slices or subtensors if possible [13, 14, 15, 16]. The BLAS are considered the de facto standard for writing efficient and portable linear algebra software, which is why nearly all processor vendors provide highly optimized BLAS implementations. LOG-based and TTGT-based implementation are in general easier to maintain and faster to port than GETT solutions. The latter might need to adapt vector instructions or blocking parameters according to a processor's microarchitecture.

In this work, we present high-performance algorithms for the tensor-matrix multiplication (TTM) which is used in important numerical methods such as the higher-order singular value decomposition and higher-order orthogonal iteration [17, 8, 7] . TTM is a compute-bound tensor operation and has the same arithmetic intensity as a matrix-matrix multiplication which can almost reach the practical peak performance of a computing machine. To our best knowledge, we are the first to combine the LOG approach described in [16, 18] for tensor-vector multiplications with the findings on tensor slicing for the tensor-matrix multiplication in [14]. Our algorithms support dense tensors with any order, dimensions and any linear tensor layout including the first- and the last-order storage formats for any contraction mode all of which can be runtime variable. Supporting arbitrary tensor layouts enables other frameworks non-column-major storage formats to easily integrate our library without tensor reformatting and unnecessary copy operations[1]. Our implementation compute the

*Corresponding author
Email address: `cem.bassoy@gmail.com` (Cem Savaş Başsoy)

[1]For example, Tensorly [19] requires tensors to be stored in the last-order storage format (row-major).

tensor-matrix product in parallel using efficient GEMM without transposing or flattening tensors. In addition to their high performance, all algorithms are layout-oblivious and provide a sustained performance independent of the tensor layout and without tuning. We provide a single algorithm that selects one of the proposed algorithms based on a simple heuristic.

Every proposed algorithm can be implemented with less than 150 lines of C++ code where the algorithmic complexity is reduced by the BLAS implementation and the corresponding selection of subtensors or tensor slices. We have provided an open-source C++ implementation of all algorithms and a python interface for convenience.

The analysis in this work quantifies the impact of the tensor layout, the tensor slicing method and parallel execution of slice-matrix multiplications with varying contraction modes. The runtime measurements of our implementations are compared with state-of-the-art approaches discussed in [11, 12, 20] including LibTorch and Eigen. While our implementation have been benchmarked with the Intel MKL and AMD AOCL libraries, the user is free to select OpenBLAS. In summary, the main findings of our work are:

- Given a row-major or column-major input matrix, the tensor-matrix multiplication with tensors of any linear tensor layout can be implemented by an in-place algorithm with 1 GEMV and 7 GEMM instances, supporting all combinations of contraction mode, tensor order and tensor dimensions.
- The proposed algorithms show a similar performance characteristic across different tensor layouts, provided that the contraction conditions remain the same.
- A simple heuristic is sufficient to select one of the proposed algorithms at runtime, providing a near-optimal performance for a wide range of tensor shapes.
- Our best-performing algorithm is a factor of 2.57 faster than Intel's batched GEMM implementation for large tensor slices.
- Our best-performing algorithm has a median speedup between 6.21% and 334.31% compared to other state-of-the art library implementations, including LibTorch and Eigen when asymmetrically shaped tensors are used.

This work is an extended version of the article "Fast and Layout-Oblivious Tensor-Matrix Multiplication with BLAS" [1]. Compared to our previous publication, we have made several significant additions. We provide runtime tests on a more recent Intel Xeon Gold 5318Y CPU and expanded our study to include AMD's AOCL, running additional benchmarks on an AMD EPYC 9354 CPU. We incorporate a newer version of TBLIS and LibTorch while also testing the TuckerMPI TTM implementation. Furthermore, we extend our implementations to support the column-major matrix storage format and benchmarked our algorithms for both row-major and column-major layouts, analyzing the runtime results in detail. We also present a heuristic that enables the use of a single TTM algorithm, ensuring efficiency across different storage formats and a wide range of tensor shapes. Lastly, we evaluate our and other libraries using real-world tensors from SDRBench [24].

The remainder of the paper is organized as follows. Section 2 presents related work. Section 3 introduces some notation on tensors and defines the tensor-matrix multiplication. Algorithm design and methods for slicing and parallel execution are discussed in Section 4. Section 5 describes the test setup. Benchmark results are presented in Section 6. Conclusions are drawn in Section 8.

2. Related Work

Springer et al. [11] present a tensor-contraction generator TCCG and the GETT approach for dense tensor contractions that is inspired from the design of a high-performance GEMM. Their unified code generator selects implementations from generated GETT, LOG and TTGT candidates. Their findings show that among 48 different contractions 15% of LOG-based implementations are the fastest.

Matthews [12] presents a runtime flexible tensor contraction library that uses GETT approach as well. He describes block-scatter-matrix algorithm which uses a special layout for the tensor contraction. The proposed algorithm yields results that feature a similar runtime behavior to those presented in [11].

Li et al. [14] introduce InTensLi, a framework that generates in-place tensor-matrix multiplication according to the LOG approach. The authors discusses optimization and tuning techniques for slicing and parallelizing the operation. With optimized tuning parameters, they report a speedup of up to 4x over the TTGT-based MATLAB tensor toolbox library discussed in [9].

Başsoy [16] presents LoG-based algorithms that compute the tensor-vector product. They support dense tensors with linear tensor layouts, arbitrary dimensions and tensor order. The presented approach contains eight cases calling GEMV and DOT. He reports average speedups of 6.1x and 4.0x compared to implementations that use the TTGT and GETT approach, respectively.

Pawlowski et al. [18] propose morton-ordered blocked layout for a mode-oblivious performance of the tensor-vector multiplication. Their algorithm iterate over blocked tensors and perform tensor-vector multiplications on blocked tensors. They are able to achieve high performance and mode-oblivious computations.

In [21] the authors present the C++ software package TuckerMPI for large-scale data compression that is used for the tensor tucker decomposition. The library provides a parallel C++ function of the latter containing distributed functions with MPI for the Gram computation and tensor-matrix multiplication. Th latter invokes a local version that contains a multi-threaded `gemm` function, computing the tensor-matrix product with submatrices according to the LOG approach. The presented local TTM corresponds to our `<par-gemm,subtensor>` version.

3. Background

3.1. Tensor Notation

An order-p tensor is a p-dimensional array where tensor elements are contiguously stored in memory[22, 7]. We write a, $\mathbf{a}$, $\mathbf{A}$ and $\underline{\mathbf{A}}$ in order to denote scalars, vectors, matrices and tensors. If not otherwise mentioned, we assume $\underline{\mathbf{A}}$ to have order $p > 2$. The p-tuple $\mathbf{n} = (n_1, n_2, \ldots, n_p)$ will be referred to as the shape or dimension tuple of a tensor where $n_r > 1$. We will use round brackets $\underline{\mathbf{A}}(i_1, i_2, \ldots, i_p)$ or $\underline{\mathbf{A}}(\mathbf{i})$ to denote a tensor element where $\mathbf{i} = (i_1, i_2, \ldots, i_p)$ is a multi-index. For convenience, we will also use square brackets to concatenate index tuples such that $[\mathbf{i}, \mathbf{j}] = (i_1, i_2, \ldots, i_r, j_1, j_2, \ldots, j_q)$ where $\mathbf{i}$ and $\mathbf{j}$ are multi-indices of length r and q, respectively.

3.2. Tensor-Matrix Multiplication (TTM)

Let $\underline{\mathbf{A}}$ and $\underline{\mathbf{C}}$ be order-p tensors with shapes $\mathbf{n}_a = ([\mathbf{n}_1, n_q, \mathbf{n}_2])$ and $\mathbf{n}_c = ([\mathbf{n}_1, m, \mathbf{n}_2])$ where $\mathbf{n}_1 = (n_1, n_2, \ldots, n_{q-1})$ and $\mathbf{n}_2 = (n_{q+1}, n_{q+2}, \ldots, n_p)$. Let $\mathbf{B}$ be a matrix of shape $\mathbf{n}_b = (m, n_q)$. A q-mode tensor-matrix product is denoted by $\underline{\mathbf{C}} = \underline{\mathbf{A}} \times_q \mathbf{B}$. An element of $\underline{\mathbf{C}}$ is defined by

$$\underline{\mathbf{C}}([\mathbf{i}_1, j, \mathbf{i}_2]) = \sum_{i_q=1}^{n_q} \underline{\mathbf{A}}([\mathbf{i}_1, i_q, \mathbf{i}_2]) \cdot \mathbf{B}(j, i_q) \tag{1}$$

with $\mathbf{i}_1 = (i_1, \ldots, i_{q-1})$, $\mathbf{i}_2 = (i_{q+1}, \ldots, i_p)$ where $1 \leq i_r \leq n_r$ and $1 \leq j \leq m$ [14, 8]. The mode q is called the contraction mode with $1 \leq q \leq p$. TTM generalizes the computational aspect of the two-dimensional case $\mathbf{C} = \mathbf{B} \cdot \mathbf{A}$ if $p = 2$ and $q = 1$. Its arithmetic intensity is equal to that of a matrix-matrix multiplication which is compute-bound for large dense matrices.

In the following, we assume that the tensors $\underline{\mathbf{A}}$ and $\underline{\mathbf{C}}$ have the same tensor layout $\boldsymbol{\pi}$. Elements of matrix $\underline{\mathbf{B}}$ can be stored either in the column-major or row-major format. With i_q iterating over the second mode of $\mathbf{B}$, TTM is also referred to as the q-mode product which is a building block for tensor methods such as the higher-order orthogonal iteration or the higher-order singular value decomposition [8]. Please note that the following method can be applied, if indices j and i_q of matrix $\mathbf{B}$ are swapped.

3.3. Subtensors

A subtensor references elements of a tensor $\underline{\mathbf{A}}$ and is denoted by $\underline{\mathbf{A}}'$. It is specified by a selection grid that consists of p index ranges. In this work, an index range of a given mode r shall either contain all indices of the mode r or a single index i_r of that mode where $1 \leq r \leq p$. Subtensor dimensions n'_r are either n_r if the full index range or 1 if a a single index for mode r is used. Subtensors are annotated by their non-unit modes such as $\underline{\mathbf{A}}'_{u,v,w}$ where $n_u > 1$, $n_v > 1$ and $n_w > 1$ for $1 \leq u \neq v \neq w \leq p$. The remaining single indices of a selection grid can be inferred by the loop induction variables of an algorithm. The number of non-unit modes determine the order p' of subtensor where $1 \leq p' < p$. In the above example, the subtensor $\underline{\mathbf{A}}'_{u,v,w}$ has three non-unit modes and is thus of order 3. For convenience, we might also use an dimension tuple $\mathbf{m}$ of length p' with $\mathbf{m} = (m_1, m_2, \ldots, m_{p'})$ to specify a mode-p' subtensor $\underline{\mathbf{A}}'_{\mathbf{m}}$. An order-2 subtensor of $\underline{\mathbf{A}}'$ is a tensor slice $\mathbf{A}'_{u,v}$ and an order-1 subtensor of $\underline{\mathbf{A}}'$ is a fiber $\mathbf{a}'_u$.

3.4. Linear Tensor Layouts

We use a layout tuple $\boldsymbol{\pi} \in \mathbb{N}^p$ to encode all linear tensor layouts including the first-order or last-order layout. They contain permuted tensor modes whose priority is given by their index. For instance, the general k-order tensor layout for an order-p tensor is given by the layout tuple $\boldsymbol{\pi}$ with $\pi_r = k - r + 1$ for $1 < r \leq k$ and r for $k < r \leq p$. The first- and last-order storage formats are given by $\boldsymbol{\pi}_F = (1, 2, \ldots, p)$ and $\boldsymbol{\pi}_L = (p, p-1, \ldots, 1)$. An inverse layout tuple $\boldsymbol{\pi}^{-1}$ is defined by $\boldsymbol{\pi}^{-1}(\boldsymbol{\pi}(k)) = k$. Given the contraction mode q with $1 \leq q \leq p$, $\hat{q}$ is defined as $\hat{q} = \boldsymbol{\pi}^{-1}(q)$. Given a layout tuple $\boldsymbol{\pi}$ with p modes, the π_r-th element of a stride tuple $\mathbf{w}$ is given by $w_{\pi_r} = \prod_{k=1}^{r-1} n_{\pi_k}$ for $1 < r \leq p$ and $w_{\pi_1} = 1$. Tensor elements of the π_1-th mode are contiguously stored in memory. Their location is given by the layout function $\lambda_{\mathbf{w}}$ which maps a multi-index $\mathbf{i}$ to a scalar index such that $\lambda_{\mathbf{w}}(\mathbf{i}) = \sum_{r=1}^{p} w_r(i_r - 1)$ [23].

3.5. Reshaping

The reshape operation defines a non-modifying reformatting transformation of dense tensors with contiguously stored elements and linear tensor layouts. It transforms an order-p tensor $\underline{\mathbf{A}}$ with a shape $\mathbf{n}$ and layout $\boldsymbol{\pi}$ tuple to an order-p' view $\underline{\mathbf{B}}$ with a shape $\mathbf{m}$ and layout $\boldsymbol{\tau}$ tuple of length p' with $p' = p - v + u$ and $1 \leq u < v \leq p$. Given a layout tuple $\boldsymbol{\pi}$ of $\underline{\mathbf{A}}$ and contiguous modes $\hat{\boldsymbol{\pi}} = (\pi_u, \pi_{u+1}, \ldots, \pi_v)$ of $\boldsymbol{\pi}$, reshape function $\varphi_{u,v}$ is defined as follows. With $j_k = 0$ if $k \leq u$ and $j_k = v - u$ if $k > u$ where $1 \leq k \leq p'$, the resulting layout tuple $\boldsymbol{\tau} = (\tau_1, \ldots, \tau_{p'})$ of $\underline{\mathbf{B}}$ is then given by $\tau_u = \min(\boldsymbol{\pi}_{u,v})$ and $\tau_k = \pi_{k+j_k} - s_k$ for $k \neq u$ with $s_k = |\{\pi_i \mid \pi_{k+j_k} > \pi_i \wedge \pi_i \neq \min(\hat{\boldsymbol{\pi}}) \wedge u \leq i \leq p\}|$. Elements of the shape tuple $\mathbf{m}$ are defined by $m_{\tau_u} = \prod_{k=u}^{v} n_{\pi_k}$ and $m_{\tau_k} = n_{\pi_{k+j}}$ for $k \neq u$. Note that reshaping is not related to tensor unfolding or the flattening operations which rearrange tensors by copying tensor elements [8, p.459].

4. Algorithm Design

4.1. Baseline Algorithm with Contiguous Memory Access

The tensor-matrix multiplication (TTM) in equation 1 can be implemented with a single algorithm using nested recursion [23]. Such an algorithm consists of two `if` statements with recursive calls and an `else` branch which constitutes the base case. A naive implementation recursively selects fibers of the input and output tensor for the base case that computes a fiber-matrix product. The outer loop iterates over the dimension m and selects an element of $\underline{\mathbf{C}}$'s fiber and a row of $\mathbf{B}$. The inner loop then iterates over dimension n_q and computes the inner product of a fiber of $\underline{\mathbf{A}}$ and the row $\mathbf{B}$. In this case, elements of $\underline{\mathbf{A}}$ and $\underline{\mathbf{C}}$ are accessed non-contiguously whenever $\pi_1 \neq q$ and matrix $\mathbf{B}$ is accessed only with unit strides if it elements are stored contiguously along its rows.

A better approach is illustrated by Algorithm 1 where the loop order is adjusted to the tensor layout $\boldsymbol{\pi}$ and memory is accessed contiguously for $\pi_1 \neq q$ and $p > 1$. The algorithm takes the input order-p tensor $\underline{\mathbf{A}}$, input matrix $\mathbf{B}$, order-p output tensor $\underline{\mathbf{C}}$, the shape tuple $\mathbf{n}$ of $\underline{\mathbf{A}}$, the layout tuple $\boldsymbol{\pi}$ of both tensors, an index tuple $\boldsymbol{\pi}$ of length p, the first dimension m of $\mathbf{B}$, the contraction mode q with $1 \leq q \leq p$ and $\hat{q} = \boldsymbol{\pi}^{-1}(q)$. Initially called with $\mathbf{i} = \mathbf{0}$ and $r = p$, the algorithm increments indices with smaller strides as $w_{\pi_r} \leq w_{\pi_{r+1}}$ with increasing recursion level and decreasing r. This is accomplished in line 5 which uses the layout tuple $\boldsymbol{\pi}$ to select a multi-index element i_{π_r} and to increment it with the corresponding stride w_{π_r}. The two `if` statements in line number 2 and 4 allow the loops over modes q and π_1 to be placed into the base case in which a slice-matrix multiplication is performed. The inner-most loop of the base case increments i_{π_1} with a unit stride and contiguously accesses tensor elements of $\underline{\mathbf{A}}$ and $\underline{\mathbf{C}}$. The second loop increments i_q with which elements of $\mathbf{B}$ are contiguously accessed if $\mathbf{B}$ is stored in the row-major format. The third loop increments j and could be placed as the second loop if $\mathbf{B}$ is stored in the column-major format.

While spatial data locality is improved by adjusting the loop ordering, slices $\underline{\mathbf{A}}'_{\pi_1,q}$, fibers $\underline{\mathbf{C}}'_{\pi_1}$ and elements $\underline{\mathbf{B}}(j, i_q)$ are accessed m, n_q and n_{π_1} times, respectively. While the specified fiber of $\underline{\mathbf{C}}$ might fit into first or second level cache, slice elements of $\underline{\mathbf{A}}$ are unlikely to fit in the local caches if the slice size $n_{\pi_1} \times n_q$ is large, leading to higher cache misses and suboptimal performance. Instead of attempting to improve the temporal data locality, we call high-performance BLAS implementations in the base case. The following subsection explains this approach.

4.2. BLAS-based Algorithms with Tensor Slices

BLAS-based algorithms for the TTM call CBLAS `gemm` function in the base case of Algorithm 1 in order to perform fast slice-matrix multiplications[2]. Function `gemm` denotes a general matrix-matrix multiplication which is defined as `C:=a*op(A)*op(B)+b*C` where `a` and `b` are scalars, `A`, `B` and `C` are matrices, `op(A)` is an `M`-by-`K` matrix, `op(B)` is a `K`-by-`N` matrix and `C` is an `N`-by-`N` matrix. Function `op(x)` either transposes the corresponding matrix `x` such that `op(x)=x'` or not `op(x)=x`. The CBLAS interface also allows users to specify matrix's leading dimension by providing the `LDA`, `LDB` and `LDC` parameters. A leading dimension specifies the number of elements that is required for iterating over the non-contiguous matrix dimension. The leading dimension can be used to perform a matrix multiplication with submatrices or even fibers within submatrices. The leading dimension parameter is necessary for the BLAS-based TTM.

The eighth TTM case in Table 1 contains all arguments that are necessary to perform a CBLAS `gemm` in the base case of Algorithm 1. The arguments of `gemm` are set according to the tensor order p, tensor layout $\boldsymbol{\pi}$ and contraction mode q. If the input matrix $\mathbf{B}$ has the row-major order, parameter `CBLAS_ORDER` of function `gemm` is set to `CblasRowMajor` (`rm`) and `CblasColMajor` (`cm`) otherwise. The eighth case will be denoted as the general case in which function `gemm` is called multiple times with different tensor slices. Next to the eighth TTM case, there are seven corner cases where a single `gemv` or `gemm` call suffices to compute the tensor-matrix product. For instance if $\pi_1 = q$, the tensor-matrix product can be computed by a matrix-matrix multiplication where the input tensor $\underline{\mathbf{A}}$ can be reshaped and interpreted as a matrix without any copy operation. Note that Table 1 supports all linear tensor layouts of $\underline{\mathbf{A}}$ and $\underline{\mathbf{C}}$ with no limitations on tensor order and contraction mode. The following subsection describes all eight TTM cases when the input matrix $\mathbf{B}$ has the row-major ordering.

1 `ttm`($\underline{\mathbf{A}}, \mathbf{B}, \underline{\mathbf{C}}, \mathbf{n}, \boldsymbol{\pi}, \mathbf{i}, m, q, \hat{q}, r$)
2   **if** $r = \hat{q}$ **then**
3   `ttm`($\underline{\mathbf{A}}, \mathbf{B}, \underline{\mathbf{C}}, \mathbf{n}, \boldsymbol{\pi}, \mathbf{i}, m, q, \hat{q}, r-1$)
4   **else if** $r > 1$ **then**
5   **for** $i_{\pi_r} \leftarrow 1$ **to** n_{π_r} **do**
6   `ttm`($\underline{\mathbf{A}}, \mathbf{B}, \underline{\mathbf{C}}, \mathbf{n}, \boldsymbol{\pi}, \mathbf{i}, m, q, \hat{q}, r-1$)
7   **else**
8   **for** $j \leftarrow 1$ **to** m **do**
9   **for** $i_q \leftarrow 1$ **to** n_q **do**
10   **for** $i_{\pi_1} \leftarrow 1$ **to** n_{π_1} **do**
11           $\underline{\mathbf{C}}([\mathbf{i}_1, j, \mathbf{i}_2])$ `+=` $\underline{\mathbf{A}}([\mathbf{i}_1, i_q, \mathbf{i}_2]) \cdot \mathbf{B}(j, i_q)$

Algorithm 1: Modified baseline algorithm for TTM with contiguous memory access. The tensor order p must be greater than 1 and the contraction mode q must satisfy $1 \leq q \leq p$ and $\pi_1 \neq q$. The initial call must happen with $r = p$ where $\mathbf{n}$ is the shape tuple of $\underline{\mathbf{A}}$ and m is the q-th dimension of $\underline{\mathbf{C}}$. Iteration along mode q with $\hat{q} = \boldsymbol{\pi}_q^{-1}$ is moved into the inner-most recursion level.

[2]CBLAS denotes the C interface to the BLAS.

Case	Order p	Layout $\boldsymbol{\pi}_{\underline{\mathbf{A}},\underline{\mathbf{C}}}$	Layout $\boldsymbol{\pi}_{\mathbf{B}}$	Mode q	Routine	`T`	`M`	`N`	`K`	`A`	`LDA`	`B`	`LDB`	`LDC`
1	1	-	`rm/cm`	1	`gemv`	-	m	n_1	-	$\mathbf{B}$	n_1	$\underline{\mathbf{A}}$	-	-
2	2	`cm`	`rm`	1	`gemm`	$\mathbf{B}$	n_2	m	n_1	$\underline{\mathbf{A}}$	n_1	$\mathbf{B}$	n_1	m
	2	`cm`	`cm`	1	`gemm`	-	m	n_2	n_1	$\mathbf{B}$	m	$\underline{\mathbf{A}}$	n_1	m
3	2	`cm`	`rm`	2	`gemm`	-	m	n_1	n_2	$\mathbf{B}$	n_2	$\underline{\mathbf{A}}$	n_1	n_1
	2	`cm`	`cm`	2	`gemm`	$\mathbf{B}$	n_1	m	n_2	$\underline{\mathbf{A}}$	n_1	$\mathbf{B}$	m	n_1
4	2	`rm`	`rm`	1	`gemm`	-	m	n_2	n_1	$\mathbf{B}$	n_1	$\underline{\mathbf{A}}$	n_2	n_2
	2	`rm`	`cm`	1	`gemm`	$\mathbf{B}$	n_2	m	n_1	$\underline{\mathbf{A}}$	n_2	$\mathbf{B}$	m	n_2
5	2	`rm`	`rm`	2	`gemm`	$\mathbf{B}$	n_1	m	n_2	$\underline{\mathbf{A}}$	n_2	$\mathbf{B}$	n_2	m
	2	`rm`	`cm`	2	`gemm`	-	m	n_1	n_2	$\mathbf{B}$	m	$\underline{\mathbf{A}}$	n_2	m
6	> 2	any	`rm`	π_1	`gemm`	$\mathbf{B}$	$\bar{n}_q$	m	n_q	$\underline{\mathbf{A}}$	n_q	$\mathbf{B}$	n_q	m
	> 2	any	`cm`	π_1	`gemm`	-	m	$\bar{n}_q$	n_q	$\mathbf{B}$	m	$\underline{\mathbf{A}}$	n_q	m
7	> 2	any	`rm`	π_p	`gemm`	-	m	$\bar{n}_q$	n_q	$\mathbf{B}$	n_q	$\underline{\mathbf{A}}$	$\bar{n}_q$	$\bar{n}_q$
	> 2	any	`cm`	π_p	`gemm`	$\mathbf{B}$	$\bar{n}_q$	m	n_q	$\underline{\mathbf{A}}$	$\bar{n}_q$	$\mathbf{B}$	m	$\bar{n}_q$
8	> 2	any	`rm`	$\pi_2, .., \pi_{p-1}$	`gemm*`	-	m	n_{π_1}	n_q	$\mathbf{B}$	n_q	$\underline{\mathbf{A}}$	w_q	w_q
	> 2	any	`cm`	$\pi_2, .., \pi_{p-1}$	`gemm*`	$\mathbf{B}$	n_{π_1}	m	n_q	$\underline{\mathbf{A}}$	w_q	$\mathbf{B}$	m	w_q

Table 1: Eight TTM cases implementing the mode-q TTM with the `gemm` and `gemv` CBLAS functions. Arguments of `gemv` and `gemm` (`T`, `M`, `N`, *dots*) are chosen with respect to the tensor order p, layout $\boldsymbol{\pi}$ of $\underline{\mathbf{A}}$, $\mathbf{B}$, $\underline{\mathbf{C}}$ and contraction mode q where `T` specifies if $\mathbf{B}$ is transposed. Function `gemm*` with a star denotes multiple `gemm` calls with different tensor slices. Argument $\bar{n}_q$ for case 6 and 7 is defined as $\bar{n}_q = (\prod_r^p n_r)/n_q$. Input matrix $\mathbf{B}$ is either stored in the column-major or row-major format. The storage format flag set for `gemm` and `gemv` is determined by the element ordering of $\mathbf{B}$.

4.2.1. Row-Major Matrix Multiplication

The following paragraphs introduce all TTM cases that are listed in Table 1.

Case 1: If $p = 1$, The tensor-vector product $\underline{\mathbf{A}} \times_1 \mathbf{B}$ can be computed with a `gemv` operation where $\underline{\mathbf{A}}$ is an order-1 tensor $\mathbf{a}$ of length n_1 such that $\mathbf{a}^T \cdot \mathbf{B}$.

Case 2-5: If $p = 2$, $\underline{\mathbf{A}}$ and $\underline{\mathbf{C}}$ are order-2 tensors with dimensions n_1 and n_2. In this case the tensor-matrix product can be computed with a single `gemm`. If $\mathbf{A}$ and $\mathbf{C}$ have the column-major format with $\boldsymbol{\pi} = (1, 2)$, `gemm` either executes $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}^T$ for $q = 1$ or $\mathbf{C} = \mathbf{B} \cdot \mathbf{A}$ for $q = 2$. Both matrices can be interpreted $\mathbf{C}$ and $\mathbf{A}$ as matrices in row-major format although both are stored column-wise. If $\mathbf{A}$ and $\mathbf{C}$ have the row-major format with $\boldsymbol{\pi} = (2, 1)$, `gemm` either executes $\mathbf{C} = \mathbf{B} \cdot \mathbf{A}$ for $q = 1$ or $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}^T$ for $q = 2$. The transposition of $\mathbf{B}$ is necessary for the TTM cases 2 and 5 which is independent of the chosen layout.

Case 6-7 : If $p > 2$ and if $q = \pi_1$(case 6), a single `gemm` with the corresponding arguments executes $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}^T$ and computes a tensor-matrix product $\underline{\mathbf{C}} = \underline{\mathbf{A}} \times_{\pi_1} \mathbf{B}$. Tensors $\underline{\mathbf{A}}$ and $\underline{\mathbf{C}}$ are reshaped with $\varphi_{2,p}$ to row-major matrices $\mathbf{A}$ and $\mathbf{C}$. Matrix $\mathbf{A}$ has $\bar{n}_{\pi_1} = \bar{n}/n_{\pi_1}$ rows and n_{π_1} columns while matrix $\mathbf{C}$ has the same number of rows and m columns. If $\pi_p = q$ (case 7), $\underline{\mathbf{A}}$ and $\underline{\mathbf{C}}$ are reshaped with $\varphi_{1,p-1}$ to column-major matrices $\mathbf{A}$ and $\mathbf{C}$. Matrix $\mathbf{A}$ has n_{π_p} rows and $\bar{n}_{\pi_p} = \bar{n}/n_{\pi_p}$ columns while $\mathbf{C}$ has m rows and the same number of columns. In this case, a single `gemm` executes $\mathbf{C} = \mathbf{B} \cdot \mathbf{A}$ and computes $\underline{\mathbf{C}} = \underline{\mathbf{A}} \times_{\pi_p} \mathbf{B}$. Noticeably, the desired contraction are performed without copy operations, see also Section 3.5.

Case 8 ($p > 2$): If the tensor order is greater than 2 with $\pi_1 \neq q$ and $\pi_p \neq q$, the modified baseline Algorithm 1 is used to successively call $\bar{n}/(n_q \cdot n_{\pi_1})$ times `gemm` with different tensor slices of $\underline{\mathbf{C}}$ and $\underline{\mathbf{A}}$. Each `gemm` computes one slice $\underline{\mathbf{C}}'_{\pi_1,q}$ of the tensor-matrix product $\underline{\mathbf{C}}$ using the corresponding tensor slices $\underline{\mathbf{A}}'_{\pi_1,q}$ and the matrix $\mathbf{B}$. The matrix-matrix product $\mathbf{C} = \mathbf{B} \cdot \mathbf{A}$ is performed by interpreting both tensor slices as row-major matrices $\mathbf{A}$ and $\mathbf{C}$ which have the dimensions (n_q, n_{π_1}) and (m, n_{π_1}), respectively.

4.2.2. Column-Major Matrix Multiplication

The tensor-matrix multiplication is performed with the column-major version of `gemm` when the input matrix $\mathbf{B}$ is stored in column-major order. Although the number of `gemm` cases remains the same, the `gemm` arguments must be rearranged. The argument arrangement for the column-major version can be derived from the row-major version that is provided in Table 1.

The CBLAS arguments of `M` and `N`, as well as `A` and `B` is swapped and the transposition flag for matrix $\mathbf{B}$ is toggled. Also, the leading dimension argument of `A` is adjusted to `LDB` or `LDA`. The only new argument is the new leading dimension of `B`.

Given case 4 with the row-major matrix multiplication in Table 1 where tensor $\underline{\mathbf{A}}$ and matrix $\mathbf{B}$ are passed to `B` and `A`. The corresponding column-major version is attained when tensor $\underline{\mathbf{A}}$ and matrix $\mathbf{B}$ are passed to `A` and `B` where the transpose flag for $\mathbf{B}$ is set and the remaining dimensions are adjusted accordingly.

4.2.3. Matrix Multiplication Variations

The column-major and row-major versions of `gemm` can be used interchangeably by adapting the storage format. This means that a `gemm` operation for column-major matrices can compute the same matrix product as one for row-major matrices, provided that the arguments are rearranged accordingly. While the argument rearrangement is similar, the arguments associated with the matrices `A` and `B` must be interchanged. Specifically, `LDA` and `LDB` as well as M and N are swapped along with the corresponding matrix pointers. In addition, the transposition flag must be set for A or B in the new format if B or A is transposed in the original version.

For instance, the column-major matrix multiplication in case 4 of Table 1 requires the arguments of A and B to be tensor $\underline{\mathbf{A}}$ and matrix $\mathbf{B}$ with $\mathbf{B}$ being transposed. The arguments of an equivalent row-major multiplication for A, B, M, N, LDA, LDB and T are then initialized with $\mathbf{B}$, $\underline{\mathbf{A}}$, m, n_2, m, n_2 and $\mathbf{B}$.

Another possible matrix multiplication variant with the same product is computed when, instead of $\mathbf{B}$, tensors $\underline{\mathbf{A}}$ and $\underline{\mathbf{C}}$ with adjusted arguments are transposed. We assume that such reformulations of the matrix multiplication do not outperform the variants shown in Table 1, as we expect BLAS libraries to have optimal blocking and multiplication strategies.

4.3. Matrix Multiplication with Subtensors

Algorithm 1 can be slightly modified in order to call gemm with reshaped order-$\hat{q}$ subtensors that correspond to larger tensor slices. Given the contraction mode q with $1 < q < p$, the maximum number of additionally fusible modes is $\hat{q} - 1$ with $\hat{q} = \boldsymbol{\pi}^{-1}(q)$ where $\boldsymbol{\pi}^{-1}$ is the inverse layout tuple. The corresponding fusible modes are therefore $\pi_1, \pi_2, \dots, \pi_{\hat{q}-1}$.

The non-base case of the modified algorithm only iterates over dimensions that have indices larger than $\hat{q}$ and thus omitting the first $\hat{q}$ modes. The conditions in line 2 and 4 are changed to $1 < r \leq \hat{q}$ and $\hat{q} < r$, respectively. Thus, loop indices belonging to the outer π_r-th loop with $\hat{q} + 1 \leq r \leq p$ define the order-$\hat{q}$ subtensors $\underline{\mathbf{A}}'_{\boldsymbol{\pi}'}$ and $\underline{\mathbf{C}}'_{\boldsymbol{\pi}'}$ of $\underline{\mathbf{A}}$ and $\underline{\mathbf{C}}$ with $\boldsymbol{\pi}' = (\pi_1, \dots, \pi_{\hat{q}-1}, q)$. Reshaping the subtensors $\underline{\mathbf{A}}'_{\boldsymbol{\pi}'}$ and $\underline{\mathbf{C}}'_{\boldsymbol{\pi}'}$ with $\varphi_{1,\hat{q}-1}$ for the modes $\pi_1, \dots, \pi_{\hat{q}-1}$ yields two tensor slices with dimension n_q or m with the fused dimension $\bar{n}_q = \prod_{r=1}^{\hat{q}-1} n_{\pi_r}$ and $\bar{n}_q = w_q$. Both tensor slices can be interpreted either as row-major or column-major matrices with shapes $(n_q, \bar{n}_q)$ or $(w_q, \bar{n}_q)$ in case of $\underline{\mathbf{A}}$ and $(m, \bar{n}_q)$ or $(\bar{n}_q, m)$ in case of $\underline{\mathbf{C}}$, respectively.

The gemm function in the base case is called with almost identical arguments except for the parameter M or N which is set to $\bar{n}_q$ for a column-major or row-major multiplication, respectively. Note that neither the selection of the subtensor nor the reshaping operation copy tensor elements. This description supports all linear tensor layouts and generalizes lemma 4.2 in [14] without copying tensor elements, see section 3.5. The division in large subtensors has also been described in [21] for tensors with a first-order layout.

4.4. Parallel BLAS-based Algorithms

Most BLAS libraries provide an option to change the number of threads. Hence, functions such as gemm and gemv can be run either using a single or multiple threads. The TTM cases one to seven contain a single BLAS call which is why we set the number of threads to the number of available cores. The following subsections discuss parallel versions for the eighth case in which the outer loops of Algorithm 1 and the gemm function inside the base case can be run in parallel. Note that the parallelization strategies can be combined with the aforementioned slicing methods.

ttm<par-loop><slice>($\underline{\mathbf{A}}$, $\mathbf{B}$, $\underline{\mathbf{C}}$, $\mathbf{n}$, $\boldsymbol{\pi}$, m, q, p)
    $[\underline{\mathbf{A}}', \underline{\mathbf{C}}', \mathbf{n}', \mathbf{w}'] =$ **reshape** ($\underline{\mathbf{A}}$, $\underline{\mathbf{C}}$, $\mathbf{n}$, m, $\boldsymbol{\pi}$, q, p)
    parallel for $i \leftarrow 1$ **to** n'_4 **do**
        parallel for $j \leftarrow 1$ **to** n'_2 **do**
            gemm(m, n'_1, n'_3, 1, $\mathbf{B}$,n'_3, $\underline{\mathbf{A}}'_{ij}$,$w'_3$, 0, $\underline{\mathbf{C}}'_{ij}$,$w'_3$)

Algorithm 2: Function ttm<par-loop><slice> is an optimized version of Algorithm 1. The reshape function transforms the order-p tensors $\underline{\mathbf{A}}$ and $\underline{\mathbf{C}}$ with layout tuple $\boldsymbol{\pi}$ and their respective dimension tuples $\mathbf{n}$ and $\mathbf{m}$ into order-4 tensors $\underline{\mathbf{A}}'$ and $\underline{\mathbf{C}}'$ with layout tuple $\boldsymbol{\pi}'$ and their respective dimension tuples $\mathbf{n}'$ and $\mathbf{m}'$ where $\mathbf{n}' = (n_{\pi_1}, \hat{n}_{\pi_2}, n_q, \hat{n}_{\pi_4})$ and $m'_3 = m$ and $n'_k = m'_k$ for $k \neq 3$. Each thread calls multiple single-threaded gemm functions each of which executes a slice-matrix multiplication with the order-2 tensor slices $\underline{\mathbf{A}}'_{ij}$ and $\underline{\mathbf{C}}'_{ij}$. Matrix $\mathbf{B}$ has the row-major storage format.

4.4.1. Sequential Loops and Parallel Matrix Multiplication

Algorithm 1 is run for the eighth case and does not need to be modified except for enabling gemm to run multi-threaded in the base case. This type of parallelization strategy might be beneficial with order-$\hat{q}$ subtensors where the contraction mode satisfies $q = \pi_{p-1}$, the inner dimensions $n_{\pi_1}, \dots, n_{\hat{q}}$ are large and the outer-most dimension n_{π_p} is smaller than the available processor cores. For instance, given a first-order storage format and the contraction mode q with $q = p - 1$ and $n_p = 2$, the dimensions of reshaped order-q subtensors are $\prod_{r=1}^{p-2} n_r$ and n_{p-1}. This allows gemm to perform with large dimensions using multiple threads increasing the likelihood to reach a high throughput. However, if the above conditions are not met, a multi-threaded gemm operates on small tensor slices which might lead to an suboptimal utilization of the available cores. This algorithm version will be referred to as <par-gemm>. Depending on the subtensor shape, we will either add <slice> for order-2 subtensors or <subtensor> for order-$\hat{q}$ subtensors with $\hat{q} = \pi_q^{-1}$.

4.4.2. Parallel Loops and Sequential Matrix Multiplication

Instead of sequentially calling multi-threaded gemm, it is also possible to call single-threaded gemms in parallel. Similar to the previous approach, the matrix multiplication can be performed with tensor slices or order-$\hat{q}$ subtensors.

Matrix Multiplication with Tensor Slices. Algorithm 2 with function ttm<par-loop><slice> executes a single-threaded gemm with tensor slices in parallel using all modes except π_1 and $\pi_{\hat{q}}$. The first statement of the algorithm calls the reshape function which transforms tensors $\underline{\mathbf{A}}$ and $\underline{\mathbf{C}}$ without copying elements by calling the reshaping operation $\varphi_{\pi_{\hat{q}+1},\pi_p}$ and $\varphi_{\pi_2,\pi_{\hat{q}-1}}$. The resulting tensors $\underline{\mathbf{A}}'$

and $\underline{\mathbf{C}}'$ are of order 4. Tensor $\underline{\mathbf{A}}'$ has the shape $\mathbf{n}' = (n_{\pi_1}, \hat{n}_{\pi_2}, n_q, \hat{n}_{\pi_4})$ with the dimensions $\hat{n}_{\pi_2} = \prod_{r=2}^{\hat{q}-1} n_{\pi_r}$ and $\hat{n}_{\pi_4} = \prod_{r=\hat{q}+1}^{p} n_{\pi_r}$. Tensor $\underline{\mathbf{C}}'$ has the same shape as $\underline{\mathbf{A}}'$ with dimensions $m'_r = n'_r$ except for the third dimension which is given by $m_3 = m$.

The following two `parallel for` loops iterate over all free modes. The outer loop iterates over $n'_4 = \hat{n}_{\pi_4}$ while the inner one loops over $n'_2 = \hat{n}_{\pi_2}$ calling `gemm` with tensor slices $\underline{\mathbf{A}}'_{2,4}$ and $\underline{\mathbf{C}}'_{2,4}$. Here, we assume that matrix $\mathbf{B}$ has the row-major format which is why both tensor slices are also treated as row-major matrices. Notice that `gemm` in Algorithm 2 will be called with exact same arguments as displayed in the eighth case in Table 1 where $n'_1 = n_{\pi_1}$, $n'_3 = n_q$ and $w_q = w'_3$. For the sake of simplicity, we omitted the first three arguments of `gemm` which are set to `CblasRowMajor` and `CblasNoTrans` for `A` and `B`. With the help of the reshaping operation, the tree-recursion has been transformed into two loops which iterate over all free indices.

Matrix Multiplication with Subtensors. An alternative algorithm is given by combining Algorithm 2 with order-$\hat{q}$ subtensors that have been discussed in 4.3. With order-$\hat{q}$ subtensors, only the outer modes $\pi_{\hat{q}+1}, \ldots, \pi_p$ are free for parallel execution while the inner modes $\pi_1, \ldots, \pi_{\hat{q}-1}, q$ are used for the slice-matrix multiplication. Therefore, both tensors are reshaped twice using $\varphi_{\pi_1, \pi_{\hat{q}-1}}$ and $\varphi_{\pi_{\hat{q}+1}, \pi_p}$. Note that in contrast to tensor slices, the first reshaping also contains the dimension n_{π_1}. The reshaped tensors are of order 3 where $\underline{\mathbf{A}}'$ has the shape $\mathbf{n}' = (\hat{n}_{\pi_1}, n_q, \hat{n}_{\pi_3})$ with $\hat{n}_{\pi_1} = \prod_{r=1}^{\hat{q}-1} n_{\pi_r}$ and $\hat{n}_{\pi_3} = \prod_{r=\hat{q}+1}^{p} n_{\pi_r}$. Tensor $\underline{\mathbf{C}}'$ has the same dimensions as $\underline{\mathbf{A}}'$ except for $m_2 = m$.

Algorithm 2 needs a minor modification for supporting order-$\hat{q}$ subtensors. Instead of two loops, the modified algorithm consists of a single loop which iterates over dimension $\hat{n}_{\pi_3}$ calling a single-threaded `gemm` with subtensors $\underline{\mathbf{A}}'$ and $\underline{\mathbf{C}}'$. The shape and strides of both subtensors as well as the function arguments of `gemm` have already been provided by the previous Section 4.3. This `ttm` version will referred to as `<par-loop><subtensor>`.

Note that functions `<par-gemm>` and `<par-loop>` implement opposing versions of the `ttm` where either `gemm` or the fused loop is performed in parallel. Version `<par-loop-gemm` executes available loops in parallel where each loop thread executes a multi-threaded `gemm` with either subtensors or tensor slices.

4.4.3. Combined Matrix Multiplication

The combined matrix multiplication calls one of the previously discussed functions depending on the number of available cores. The heuristic assumes that function `<par-gemm>` is not able to efficiently utilize the processor cores if subtensors or tensor slices are too small. The corresponding algorithm switches between `<par-loop>` and `<par-gemm>` with subtensors by first calculating the parallel and combined loop count $\hat{n} = \prod_{r=1}^{\hat{q}-1} n_{\pi_r}$ and $\hat{n}' = \prod_{r=1}^{p} n_{\pi_r}/n_q$, respectively. Given the number of physical processor cores as `ncores`, the algorithm executes `<par-loop>` with `<subtensor>` if `ncores` is greater than or equal to $\hat{n}$ and call `<par-loop>` with `<slice>` if `ncores` is greater than or equal to $\hat{n}'$. Otherwise, the algorithm will default to `<par-gemm>` with `<subtensor>`. Function `par-gemm` with tensor slices is not used here. The presented strategy is different to the one presented in [14] that maximizes the number of modes involved in the matrix multiply. We will refer to this version as `<combined>` to denote a selected combination of `<par-loop>` and `<par-gemm>` functions.

4.4.4. Multithreaded Batched Matrix Multiplication

The multithreaded batched matrix multiplication version calls in the eighth case a single `gemm_batch` function that is provided by Intel MKL's BLAS-like extension. With an interface that is similar to the one of `cblas_gemm`, function `gemm_batch` performs a series of matrix-matrix operations with general matrices. All parameters except `CBLAS_LAYOUT` requires an array as an argument which is why different subtensors of the same corresponding tensors are passed to `gemm_batch`. The subtensor dimensions and remaining `gemm` arguments are replicated within the corresponding arrays. Note that the MKL is responsible of how subtensor-matrix multiplications are executed and whether subtensors are further divided into smaller subtensors or tensor slices. This algorithm will be referred to as `<batched-gemm>`.

5. Experimental Setup

5.1. Computing System

The runtime benchmark have been executed on a dual socket Intel Xeon Gold 5318Y CPU with an Ice Lake architecture and a dual socket AMD EPYC 9354 CPU with a Zen4 architecture. With two NUMA domains, the Intel CPU consists of 2×24 cores which run at a base frequency of 2.1 GHz. Assuming a peak AVX-512 Turbo frequency of 2.5 GHz, the CPU is able to process 3.84 TFLOPS in double precision. We have measured a peak double-precision floating-point performance of 3.8043 TFLOPS (79.25 GFLOPS/core) and a peak memory throughput of 288.68 GB/s using the Likwid performance tool. The AMD EPYC 9354 CPU consists of 2×32 cores running at a base frequency of 3.25 GHz. Assuming an all-core boost frequency of 3.75 GHz, the CPU is theoretically capable of performing 3.84 TFLOPS in double precision. We measured a peak double-precision floating-point performance of 3.87 TFLOPS (60.5 GFLOPS/core) and a peak memory throughput of 788.71 GB/s.

All libraries have been compiled with the GNU compiler v11.2.0 using the highest optimization level `-O3` together with the `-fopenmp` and `-std=c++17` flags. Loops within the eighth case have been parallelized using GCC's OpenMP v4.5 implementation. In case of the Intel CPU,

Dataset	Tensor Shape Ex.	Matrix Shape Ex.
N_1	$65536 \times 1024 \times 2$	65536×1024
	$2048 \times 1024 \times 2 \times 2 \times 2$	2048×1024
N_2	$1024 \times 65536 \times 2$	65536×1024
	$1024 \times 2048 \times 2 \times 2 \times 2$	2048×1024
N_3	$1024 \times 2 \times 65536$	65536×1024
	$1024 \times 2 \times 2048 \times 2 \times 2$	2048×1024
N_{10}	$1024 \times 2 \times 65536$	65536×1024
	$1024 \times 2 \times 2 \times 2 \times 2048$	2048×1024
M	$256 \times 256 \times 256$	256×256
	$32 \times 32 \times 32 \times 32 \times 32$	32×32

Dataset Q (orig. Name)	Tensor Shape	Matrix Shape Ex.
CESM ATM	$26 \times 1800 \times 3600$	1800×26
ISABEL	$100 \times 500 \times 500 \times 13$	500×100
NYX	$512 \times 512 \times 512 \times 6$	512×512
SCALE-LETK	$98 \times 1200 \times 1200 \times 13$	1200×98
QMCPACK	$69 \times 69 \times 115 \times 288$	69×69
Miranda	$256 \times 384 \times 384 \times 7$	384×256
SP	$500 \times 500 \times 500 \times 11$	500×500
EXAFEL	$986 \times 32 \times 185 \times 388$	32×986

Table 2: Tensor shape sets and example dimension tuples that are used in our runtime benchmarking. The first 4 shape sets N_1, N_2, N_3 and N_{10} are used to generate asymmetrically shaped tensors, each consisting of 72 dimension tuples. Shape set M contains 48 tensor shapes that are used to generate symmetrically shaped tensors. Shape set Q contains 8 tensor shapes that are part of SDRBench [24]. Note that all matrix shapes depend on the input tensor shapes and contraction mode.

the Intel Math Kernel Library 2022 (MKL) and its threading library `mkl_intel_thread`, threading runtime library `libiomp5` has been used for the three BLAS functions `gemv`, `gemm` and `gemm_batch`. For the AMD CPU, the AMD library AOCL v4.2.0 has been compiled with the `zen4` flag.

5.2. OpenMP Parallelization

The loops in the `par-loop` algorithms have been parallelized using the OpenMP directive `omp parallel for` together with the `schedule(static)`, `num_threads(ncores)` and `proc_bind(spread)` clauses. In case of tensor-slices, the `collapse(2)` clause has been added for transforming both loops into one loop which has an iteration space of the first loop times the second one. We also had to enable nested parallelism using `omp_set_nested` to toggle between single- and multi-threaded `gemm` calls for different TTM cases when using AMD AOCL.

The `num_threads(ncores)` clause specifies the number of threads within a team where `ncores` is equal to the number of processor cores. Hence, each OpenMP thread is responsible for computing $\bar{n}'/$`ncores` independent slice-matrix products where $\bar{n}' = n_2' \cdot n_4'$ for tensor slices and $\bar{n}' = n_4'$ for mode-$\hat{q}$ subtensors.

The `schedule(static)` instructs the OpenMP runtime to divide the iteration space into equally sized chunks, except for the last chunk. Each thread sequentially computes $\bar{n}'/$`ncores` slice-matrix products. We have decided to use this scheduling kind as all slice-matrix multiplications exhibit the same number of floating-point operations with a regular workload where one can assume negligible load imbalance. Moreover, we wanted to prevent scheduling overheads for small slice-matrix products were data locality can be an important factor for achieving higher throughput.

The `OMP_PLACES` environment variable has not been explicitly set and thus defaults to the OpenMP `cores` setting which defines an OpenMP place as a single processor core. Together with the clause `num_threads(ncores)`, the number of OpenMP threads is equal to the number of OpenMP places, i.e. to the number of processor cores. We did not measure any performance improvements for a higher thread count.

The `proc_bind(spread)` clause additionally binds each OpenMP thread to one OpenMP place which lowers inter-node or inter-socket communication and improves local memory access. Moreover, with the `spread` thread affinity policy, consecutive OpenMP threads are spread across OpenMP places which can be beneficial if the user decides to set `ncores` smaller than the number of processor cores.

5.3. Data sets

We have evaluated the performance of our algorithms with asymmetrically and symmetrically shaped tensors to account for a wide range of use cases. Their corresponding tensor shapes are divided into 12 sets N_1, N_2, …, N_{10}, M and Q. Table 2 contains example dimension tuples for the input tensor and matrix. The shape of the latter is (n_2, n_q) if $q = 1$ and (n_1, n_q) otherwise where q is the contraction mode with $1 \leq q \leq p$. The computation of the output tensor dimensions is described in Section 3.2.

The first shape 10 sets N_1 to N_{10} contain 9×8 tensor shapes all of which generate asymmetrically shaped tensors. Within one set N_k, dimension tuples are arranged within 10 two-dimensional shape arrays $\mathbf{N}_k$ of size 9×8 with $1 \leq k \leq 10$. A dimension tuple $\mathbf{n}_{r,c}$ within $\mathbf{N}_k$ is of length $r + 1$ with $1 \leq r \leq 9$ and $1 \leq c \leq 8$. Its i-th element is either 1024 for $i = 1 \land k \neq 1$ or $i = 2 \land k = 1$, or $c \cdot 2^{15-r}$ for $i = \min(r + 1, k)$ or 2 otherwise. A special feature of this test set is that the contraction dimension and the leading dimension are disproportionately large.

The second shape set M contains 48 tensor shapes that generate symmetrically shaped tensors. The shapes are arranged within one two-dimensional shape array $\mathbf{M}$ of size 6×8. Similar to the previous setup, the row number r is equal to the tensor order $r + 1$ with $1 \leq r6$. A row of the tensor shape array consists 8 dimension tuples of the same length $r + 1$ where elements of one dimension tuple are equal such that $m_{r,c} = \mathbf{m}_{r,c}(i) = \mathbf{m}_{r,c}(j)$ for $1 \leq i, j \leq r + 1$. With eight shapes and the step size of each row $s_r = (m_{r,8} - m_{r,1})/8$, the respective intermediate dimensions $m_{r,c}$ are given by $m_{r,c} = m_{r,1} + (c - 1)s_r$ with

$1 \leq c \leq 8$. Symmetrically and asymmetrically shaped tensors have also been used in [16, 23].

We have also benchmarked with eight tensors that are part of the scientific data reduction benchmark (SDRBench) [24]. The scientific datasets in SDRBench mainly consist of order-3 tensors with different tensor shapes and number of data fields, originating from various real-world simulations. Tensors from the SP dataset for instance has been used for benchmarking the truncated Tucker decomposition in [21] We perform runtime tests with order-4 tensors that are generated with dimension tuples of the tensor shape set Q. Their first three dimensions correspond to the respective ones mentioned in the original data sets and the last dimension to the number of data fields. All tensor shapes are provided in Table 2.

5.4. Profiling setup

Our benchmark suite iterates through one of tensor shape sets for one contraction mode q with $1 \leq q \leq \max_p$ where $\max_p$ is the maximum tensor order within the shape set. Tensor and matrix elements are randomly generated single-precision floating-point numbers in case of the data set Q. In all other cases double-precision is used. The profiler first sweeps through tensor shapes belonging to one tensor order and then iteratively selects one larger tensor order for the next sweep. It should be noted that if $q > p$, the contraction mode q is set to p. Given a dimension tuple of length, the profiler generates two tensors and a matrix, executes a mode-q TTM implementation 20 times and finally computes the median runtime of the benchmarked TTM implementation. To prevent caching of the output tensor, we invalidate caches which is excluded from the timing.

The runtime results for one contraction mode and one TTM implementation are stored in a two-dimensional array with shape $\max_p \times k$ where k is either 8 in case of asymmetrically and symmetrically shaped tensors or 1 in case of the set Q. Hence, our profiler generates 10 runtime arrays of shape 9×8 with asymmetrically shaped tensors for 10 contraction modes using the shape sets N_1, N_2, $\ldots$, N_{10}. Generating symmetrically shaped tensors with the shape set M, the profiler returns 7 runtime arrays of shape 6×8 for 7 contraction modes. Using the shape set Q, 4 one-dimensional runtime arrays for 4 contraction modes are computed.

The three-dimensional runtime data generated with the data sets N and M can be used to create two dimensional performance maps, as it is done in the following Section 6. Each value in a performance map corresponds to a mean or median value over tensor sizes (i.e. dimension tuples with the same length), over tensor orders or contraction modes.

6. Experimental Results and Discussion

The runtime results within the following subsections are executed with asymmetrically and symmetrically shaped tensors. The last subsection also considers tensors with real-world tensor shapes. The corresponding tensor shapes and their shape sets have been described in the previous section 5.

6.1. Slicing Methods

This section analyzes the performance of the two proposed slicing methods <slice> and <subtensor> that have been discussed in section 4.4. Fig. 1 contains eight performance contour plots of four ttm functions <par-loop> and <par-gemm>. Both functinos either compute the slice-matrix product with subtensors <subtensor> or tensor slices <slice> on the Intel Xeon Gold 5318Y CPU. Each contour level within the plots represents a mean GFLOPS/core value that is averaged across tensor sizes.

Every contour plot contains all applicable TTM cases listed in Table 1. The first column of performance values is generated by gemm belonging to the TTM case 3, except the first element which corresponds to TTM case 2. The first row, excluding the first element, is generated by TTM case 6 function. TTM case 7 is covered by the diagonal line of performance values when $q = p$. Although Fig. 1 suggests that $q > p$ is possible, our profiling program ensures that $q = p$. TTM case 8 with multiple gemm calls is represented by the triangular region which is defined by $1 < q < p$.

With asymmetrically shaped tensors, <par-loop,slice> runs on average with 34.96 GFLOPS/core (1.67 TFLOPS). With a maximum performance of 57.805 GFLOPS/core (2.77 TFLOPS), it performs on average 89.64% faster than <par-loop,subtensor>. The slowdown with subtensors at $q = p - 1$ or $q = p - 2$ can be explained by the small loop count of the function that are 2 and 4, respectively. While function <par-loop,slice> is affected by the tensor shapes for dimensions $p = 3$ and $p = 4$ as well, its performance improves with increasing order due to the increasing loop count. Function <par-loop,slice> achieves on average 17.34 GFLOPS/core (832.42 GFLOPS) if symmetrically shaped tensors are used. If subtensors are used, function <par-loop,subtensor> achieves a mean throughput of 17.62 GFLOPS/core (846.16 GFLOPS) and is on average 9.89% faster than <par-loop,slice>. The performances of both functions are monotonically decreasing with increasing tensor order, see plots (1.c) and (1.d) in Fig. 1.

Function <par-gemm,slice> averages 36.42 GFLOPS/-core (1.74 TFLOPS) and achieves up to 57.91 GFLOPS/-core (2.77 TFLOPS) with asymmetrically shaped tensors. Using subtensors, function <par-gemm,subtensor> exhibits almost identical performance characteristics and is on average 3.42% slower than its counterpart with tensor slices. For symmetrically shaped tensors, <par-gemm> with subtensors and tensor slices achieve a mean throughput 15.98 GFLOPS/core (767.31 GFLOPS) and 15.43 GFLOPS/-core (740.67 GFLOPS), respectively. However, function <par-gemm,subtensor> is on average 87.74% faster than <par-gemm,slice> which is hardly visible due to small performance values around 5 GFLOPS/core or less whenever $q < p$ and the dimensions are smaller than 256. The

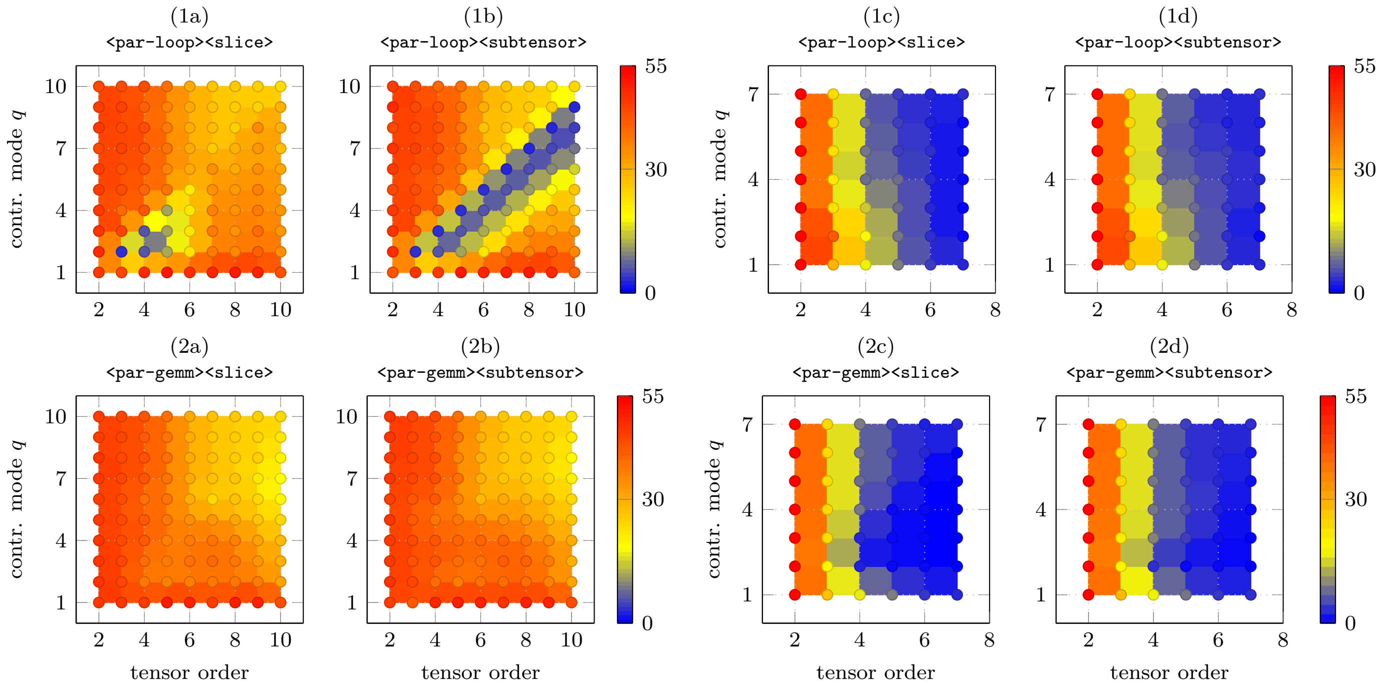

Figure 1: Performance contour plots in double-precision GFLOPS/core of the proposed TTM algorithms <par-loop> and <par-gemm> with varying tensor orders p and contraction modes q. The top row of maps (1x) depict measurements of the <par-loop> versions while the bottom row of maps with number (2x) contain measurements of the <par-gemm> versions. Tensors are asymmetrically shaped on the left four maps (a,b) and symmetrically shaped on the right four maps (c,d). Tensor $\underline{\mathbf{A}}$ and $\underline{\mathbf{C}}$ have the first-order while matrix $\mathbf{B}$ has the row-major ordering. All functions have been measured on an Intel Xeon Gold 5318Y.

speedup of the `<subtensor>` version can be explained by the smaller loop count and slice-matrix multiplications with larger tensor slices.

Our findings indicate that, regardless of the parallelization method employed, subtensors are most effective with symmetrically shaped tensors, whereas tensor slices are preferable with asymmetrically shaped tensors when both the contraction mode and leading dimension are large.

6.2. Parallelization Methods

This subsection compares the performance results of the two parallelization methods, `<par-gemm>` and `<par-loop>`, as introduced in Section 4.4 and illustrated in Fig. 1.

With asymmetrically shaped tensors, both `<par-gemm>` functions with subtensors and tensor slices compute the tensor-matrix product on average with ca. 36 GFLOPS/core and outperform function `<par-loop,subtensor>` on average by a factor of 2.31. The speedup can be explained by the performance drop of function `<par-loop,subtensor>` to 3.49 GFLOPS/core at $q = p - 1$ while both versions of `<par-gemm>` operate around 39 GFLOPS/core. Function `<par-loop,slice>` performs better for reasons explained in the previous subsection. However, it is on average 30.57% slower than function `<par-gemm,slice>` due to the aforementioned performance drops.

In case of symmetrically shaped tensors, `<par-loop>` with subtensors and tensor slices outperform their corresponding `<par-gemm>` counterparts by 23.3% and 32.9%, respectively. The speedup mostly occurs when $1 < q < p$ where the performance gain is a factor of 2.23. This performance behavior can be expected as the tensor slice sizes decreases for the eighth case with increasing tensor order causing the parallel slice-matrix multiplication to perform on smaller matrices. In contrast, `<par-loop>` can execute small single-threaded slice-matrix multiplications in parallel.

In summary, function `<par-loop,subtensor>` with symmetrically shaped tensors performs best. If the leading and contraction dimensions are large, both versions of function `<par-gemm>` outperform `<par-loop>` with any type of slicing.

6.3. LoG Variants

The contour plots in Fig. 1 contain performance data that are generated by all applicable TTM cases of each `ttm` function. Yet, the presented slicing or parallelization methods only affect the eighth case, while all other TTM cases apply a single multi-threaded `gemm` with the same configuration. The following analysis will consider performance values of the eighth case in order to have a more fine grained visualization and discussion of the loops over `gemm` implementations. Fig. 2 contains cumulative performance distributions of all the proposed algorithms including the functions `<batched-gemm>` and `<combined>` for the eighth TTM case only. Moreover, the experiments have been additionally executed on the AMD EPYC processor and with the column-major ordering of the input matrix as well.

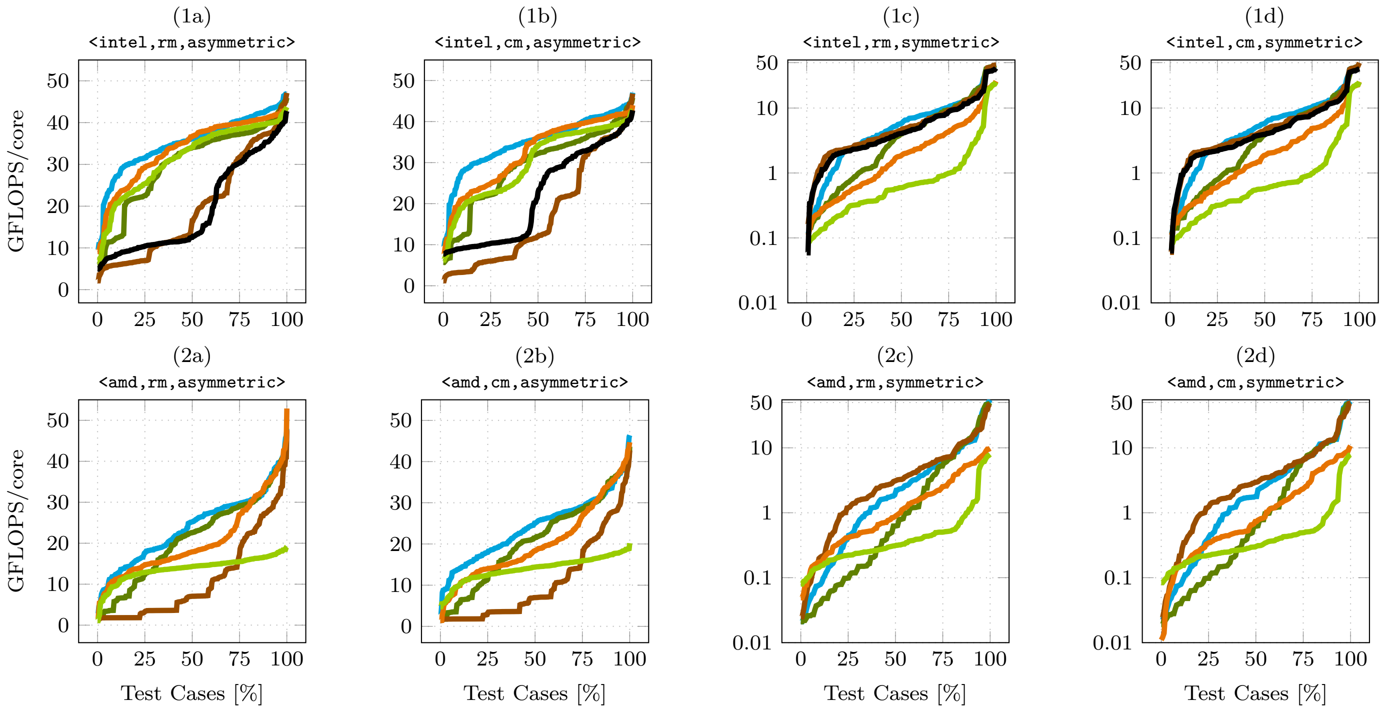

Figure 2: Cumulative performance distributions in double-precision GFLOPS/core of the proposed algorithms for the eighth case. Each distribution belongs to one algorithm: `<batched-gemm>` (black), `<combined>` (blue), `<par-gemm,slice>` (light green) and `<par-loop,slice>` (dark green), `<par-gemm,subtensor>` (orange) and `<par-loop,subtensor>` (brown). The top row of maps (1x) depict measurements performed on an Intel Xeon Gold 5318Y with the MKL while the bottom row of maps with number (2x) contain measurements performed on an AMD EPYC 9354 with the AOCL. Tensors are asymmetrically shaped in (a) and (b) and symmetrically shaped in (c) and (d). Input matrix has the row-major ordering (rm) in (a) and (c) and column-major ordering (cm) in (b) and (d).

The probability x of a point (x, y) of a distribution function for a given algorithm corresponds to the number of test instances for which that algorithm that achieves a throughput of either y or less. For instance, function `<batched-gemm>` computes the tensor-matrix product with asymmetrically shaped tensors in 25% of the tensor instances with equal to or less than 10 GFLOPS/core. Please note that the four plots on the right, plots (c) and (d), have a logarithmic y-axis for a better visualization.

6.3.1. *Combined Algorithm and Batched GEMM*

This subsection discusses the performance of function `<batched-gemm>` and `<combined>` against those of `<par-loop>` and `<par-gemm>` for the eighth TTM case.

Given a row-major matrix ordering, the combined function `<combined>` achieves on the Intel processor a median throughput of 36.15 and 4.28 GFLOPS/core with asymmetrically and symmetrically shaped tensors. Reaching up to 46.96 and 45.68 GFLOPS/core, it is on par with `<par-gemm,subtensor>` and `<par-loop,slice>` and outperforms them for some tensor instances. Note that both functions run significantly slower either with asymmetrically or symmetrically shaped tensors. The observable superior performance distribution of `<combined>` can be attributed to the heuristic which switches between `<par-loop>` and `<par-gemm>` depending on the inner and outer loop count as explained in section 4.4.

Function `<batched-gemm>` of the BLAS-like extension library has a performance distribution that is akin to the `<par-loop,subtensor>`. In case of asymmetrically shaped tensors, all functions except `<par-loop,subtensor>` outperform `<batched-gemm>` on average by a factor of 2.57 and up to a factor 4 for $2 \leq q \leq 5$ with $q + 2 \leq p \leq q + 5$. In contrast, `<par-loop,subtensor>` and `<batched-gemm>` show a similar performance behavior in the plot (1c) and (1d) for symmetrically shaped tensors, running on average 3.55 and 8.38 times faster than `<par-gemm>` with subtensors and tensor slices, respectively.

In summary, `<combined>` performs as fast as, or faster than, `<par-gemm,subtensor>` and `<par-loop,slice>`, depending on the tensor shape. Conversely, `<batched-gemm>` underperforms for asymmetrically shaped tensors with large contraction modes and leading dimensions.

6.3.2. *Matrix Formats*

This subsection discusses if the input matrix storage formats have any affect on the runtime performance of the proposed functions. The cumulative performance distributions in Fig. 2 suggest that the storage format of the input matrix has only a minor impact on the performance. The Euclidean distance between normalized row-major and column-major performance values is around 5 or less with a maximum dissimilarity of 11.61 or 16.97, indicating a moderate similarity between the corresponding

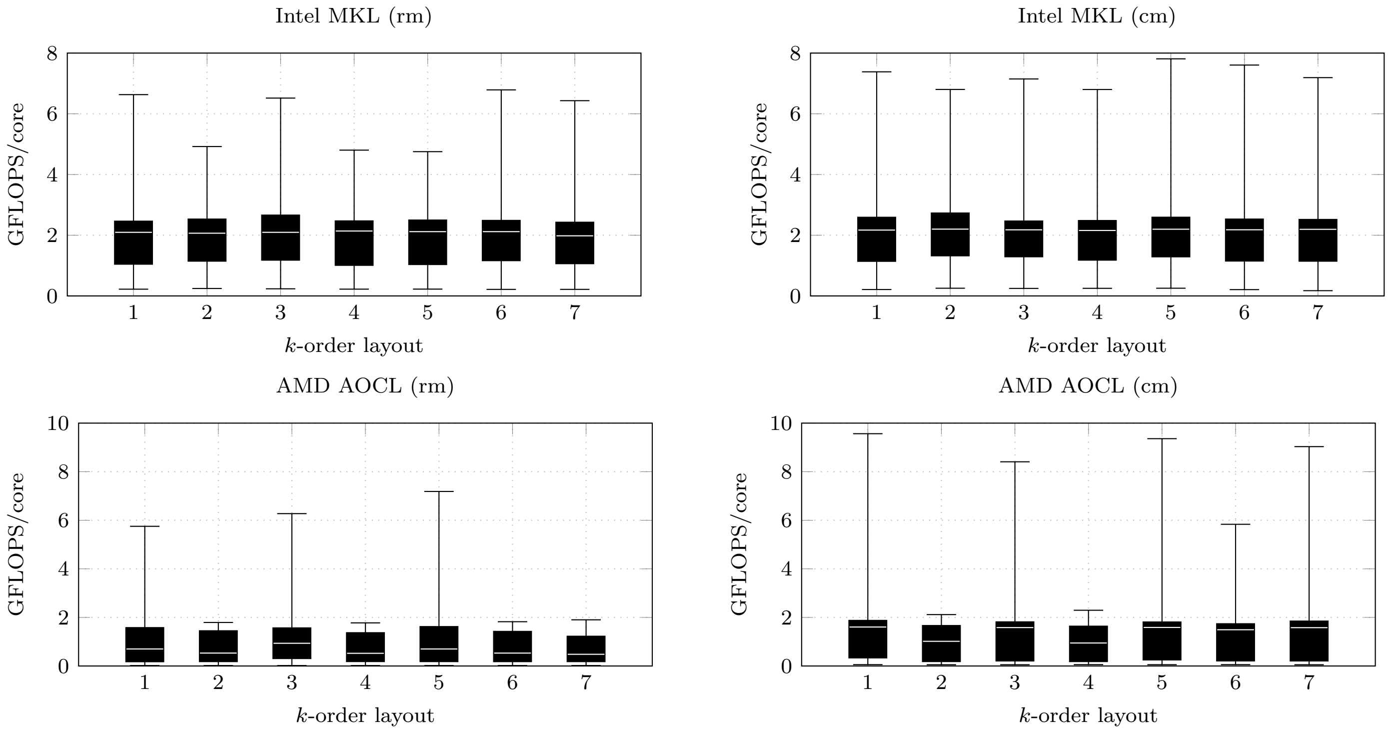

Figure 3: Box plots visualizing performance statics in double-precision GFLOPS/core of the function with row-major (left) or column-major matrices (right). Box plot number k denotes the k-order tensor layout of symmetrically shaped tensors with order 7.

row-major and column-major data sets. Moreover, their respective median values with their first and third quartiles differ by less than 5% with three exceptions where the difference of the median values is between 10% and 15%.

6.3.3. BLAS Libraries

This subsection compares the performance of functions that use Intel's Math Kernel Library (MKL) on the Intel Xeon Gold 5318Y processor with those that use the AMD Optimizing CPU Libraries (AOCL) on the AMD EPYC 9354 processor. Comparing the performance per core and limiting the runtime evaluation to the eighth case, MKL-based functions with asymmetrically shaped tensors run on average between 1.48 and 2.43 times faster than those with the AOCL. For symmetrically shaped tensors, MKL-based functions are between 1.93 and 5.21 times faster than those with the AOCL. In general, MKL-based functions on the respective CPU achieve a speedup of at least 1.76 and 1.71 compared to their AOCL-based counterpart when asymmetrically and symmetrically shaped tensors are used.

6.4. Tensor Layouts

Fig. 3 contains four box plots summarizing the performance distribution of the `<combined>` function using the AOCL and MKL. Every k-th box plot has been computed from benchmark data with symmetrically shaped order-7 tensors that has a k-order tensor layout. The 1-order and 7-order layout, for instance, are the first-order and last-order storage formats of an order-7 tensor.

The reduced performance of around 1 and 2 GFLOPS can be attributed to the fact that contraction and leading dimensions of symmetrically shaped subtensors are at most 48 and 8, respectively. When `<combined>` is used with MKL, the relative standard deviations (RSD) of its median performances are 2.51% and 0.74%, with respect to the row-major and column-major formats. The RSD of its respective interquartile ranges (IQR) are 4.29% and 6.9%, indicating a similar performance distributions. Using `<combined>` with AOCL, the RSD of its median performances for the row-major and column-major formats are 25.62% and 20.66%, respectively. The RSD of its respective IQRs are 10.83% and 4.31%, indicating a similar performance distributions. A similar performance behavior can be observed also for other `ttm` variants such as `<par-loop,slice>`. The runtime results demonstrate that the function performances stay within an acceptable range independent for different k-order tensor layouts and show that our proposed algorithms are not designed for a specific tensor layout.

6.5. Comparison with Related Work

This subsection compares our best performing algorithm with libraries that do not use the LOG approach. **TCL** implements the TTGT approach with a high-perform tensor-transpose library **HPTT** which is discussed in [11]. TCL has been used with the same BLAS libraries as TLIB

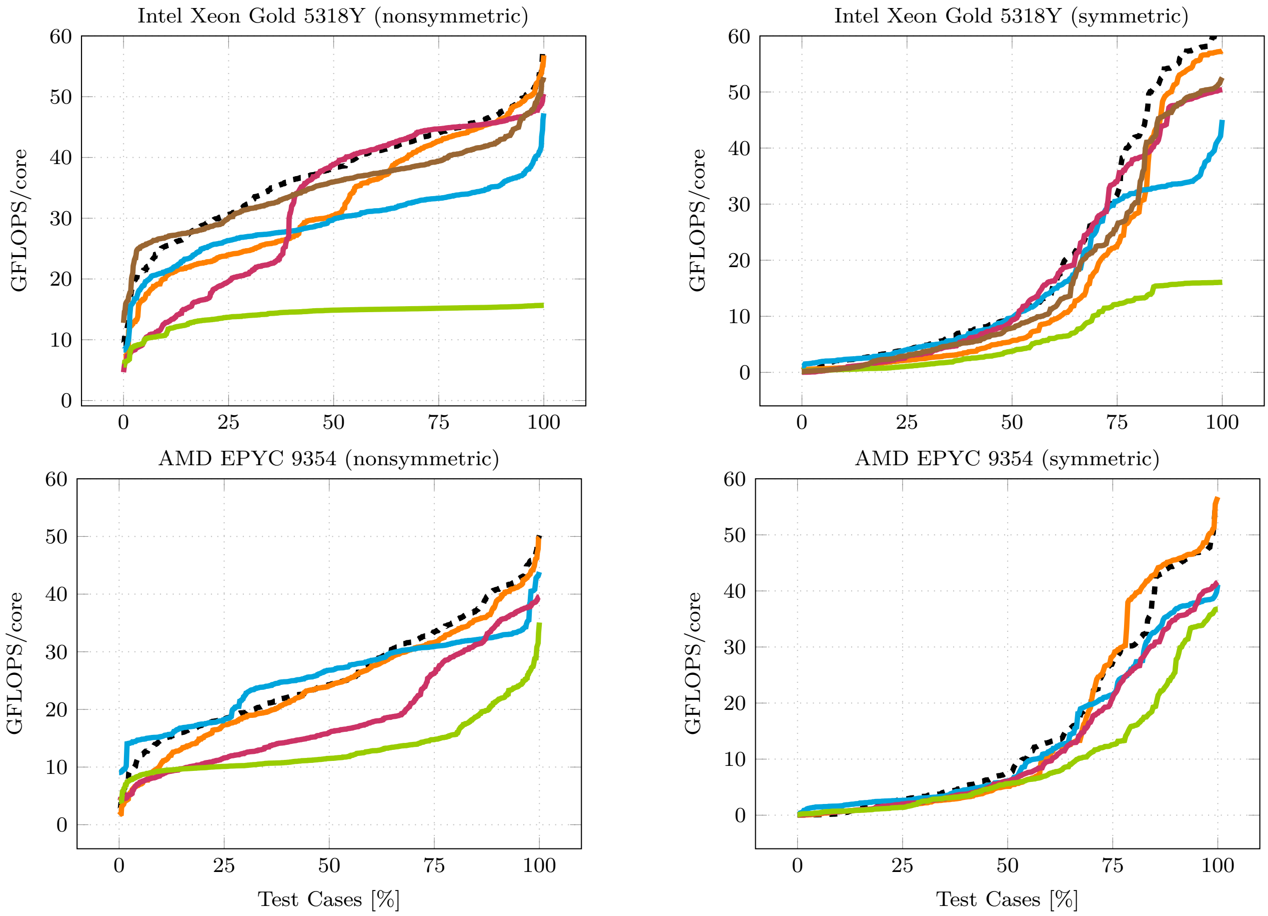

Figure 4: Cumulative performance distributions of TTM implementations in double-precision GFLOPS/core. Each distribution corresponds to a library: **TLIB**[ours] ( ), **TCL** ( ), **TBLIS** ( ), **LibTorch** ( ), **Eigen** ( ), **TuckerMPI** ( ). Libraries have been tested with asymmetrically-shaped (left plot) and symmetrically-shaped tensors (right plot).

to ensure a fair comparison. **TBLIS** (v1.2.0) implements the GETT approach that is akin to BLIS' algorithm design for the matrix multiplication [12]. The library has been compiled with the `zen4` and `skx-2` to enable architecture specific optimization. The tensor extension of **Eigen** (v3.4.90) is used by the Tensorflow framework. Library **LibTorch** (v2.5.0) is the C++ distribution of PyTorch [20]. The **TuckerMPI** library is a parallel C++ software package for large-scale data compression which provides a local and distributed TTM function [21]. The local version implements the LOG approach using a BLAS implementation and computes the TTM product similar to our function `<par-gemm,subtensor>`. Note that we were use the current TuckerMPI version with Intel's MKL, which is why TuckerMPI's TTM has only been executed on the Intel CPU. **TLIB** denotes our library and the previously discussed `<combined>` algorithm. If not otherwise stated, all of the following performance and comparisons numbers represent medians across a specified tensor set.

6.5.1. Artificial Tensor Shapes

Fig. 2 compares the performance distribution of our implementation with the previously mentioned libraries. The benchmark is executed with asymmetrically and symmetrically shaped tensors. The corresponding tensor shapes and their shape sets have been described in Section 5.

Using MKL on the Intel CPU, TLIB achieves a performance of 38.21 GFLOPS/core (1.83 TFLOPS) and reaches with asymmetrically shaped tensors at most 57.68 GFLOPS/core (2.76 TFLOPS), given the the shape sets N_k with $1 \leq k \leq 10$. TLIB is in at least 2.03x as many tensor instances faster than other libraries and achieves a speedup of at least 6.36%. LibTorch and TuckerMPI have almost the same performance of 38.17 and 35.98 GFLOPS/core, yet only reach a peak performance 50.48 and 53.21 GFLOPS/core. Both are 17.47% and 6.97% slower than TLIB. In case of symmetrically shaped tensors from the shape set M, TLIB's computes the TTM with 8.99 GFLOPS/core (431.52 GFLOPS). Except for TBLIS, TLIB achieves a speedup for at least 33% more tensor instances and is at least 3.08% faster. Moreover, TLIB achieves a median speedup of 12.98% and 6.23% compared to LibTorch and TuckerMPI. With a higher performance of 9.73 GFLOPS/core, TBLIS is faster than TLIB for about the same amount of tensor instances and is 1.38% slower than TLIB.

On the AMD CPU, TLIB computes the tensor prod-

Library	Performance [GFLOPS/core]			Speedup [%]
	Min	Median	Max	Median
TLIB	9.45	**38.27**	**57.87**	-
TCL	7.14	30.46	56.81	6.36
TBLIS	8.33	29.85	47.28	23.96
LibTorch	4.65	38.17	50.48	17.47
Eigen	5.85	14.89	15.67	170.77
TuckerMPI	**12.79**	35.98	53.21	6.97
TLIB	0.14	8.99	**58.14**	-
TCL	0.36	5.64	57.35	3.08
TBLIS	**1.11**	**9.73**	45.03	1.38
LibTorch	0.02	9.31	50.44	12.98
Eigen	0.21	3.80	16.06	216.69
TuckerMPI	0.12	7.91	52.57	6.23

Library	Performance [GFLOPS/core]			Speedup [%]
	Min	Median	Max	Median
TLIB	2.71	24.28	50.18	-
TCL	1.67	24.11	49.85	0.57
TBLIS	**9.06**	**26.81**	47.83	0.43
LibTorch	0.63	16.04	**50.84**	29.68
Eigen	4.06	11.49	35.08	117.48
TLIB	0.02	**7.75**	54.16	-
TCL	0.01	5.14	**56.75**	6.10
TBLIS	0.06	6.14	41.11	13.64
LibTorch	0.06	6.04	41.65	12.37
Eigen	**0.07**	5.58	36.76	114.22

Table 3: The table presents the minimum, median, and maximum runtime performances in GFLOPS/core alongside the median speedup of TLIB compared to other libraries. The tests were conducted on an Intel Xeon Gold 5318Y CPU (left) and an AMD EPYC 9354 CPU (right). The performance values on the upper and lower rows of one table were evaluated using asymmetrically and symmetrically shaped tensors, respectively.

uct with 24.28 GFLOPS/core (1.55 TFLOPS), reaching with asymmetrically shaped tensors a maximum performance of 50.18 GFLOPS/core (3.21 TFLOPS). TBLIS and TCL execute the TTM with 26.81 and 24.11 GFLOPS/core, executing the TTM equally fast as TLIB with a speedup percentage of 0.57 and 0.43. Moreover, TLIB is faster than TBLIS and TCL in the same number of tensor instances as in the opposite case. The three libraries are 29.68% and a factor of 2.17 faster than LibTorch and Eigen, respectively. In case of symmetrically shaped tensors, TLIB has a median performance of 7.52 GFLOPS/core (481.39 GFLOPS). Compared to the second-fastest library TCL, TLIB speeds up the computation by 6.10% and is in 43.66% more tensor instances faster than TCL. TBLIS, LibTorch and Eigen are slower than TLIB by at least 12.37%.

In most instances, TLIB is faster than other libraries across all TTM cases with few exceptions. On the AMD CPU, TCL achieves a higher throughput of about 9% for the second and third TTM cases when asymmetrically shaped tensors are used. TBLIS is 12.63% faster than TLIB for the eighth TTM case with the same tensor set. On the Intel CPU, LibTorch is in the 7th TTM case 16.94% faster than TLIB. The TCL library runs on average as fast as TLIB in the 6th and 7th TTM cases . The performances of TLIB, TBLIS and TuckerMPI in the 8th TTM case are almost on par, TLIB executing the TTM about 3.2% faster. In case of symmetrically shaped tensors, TBLIS and LibTorch outperform TLIB in the 7th TTM case by 38.5% and 219.5%.

6.5.2. Real-World Tensor Shapes

We have additionally conducted performance tests with an order-3 and seven order-4 tensors that have also been used in SDRBench [24]. The corresponding tensor shape set Q and the tensor shapes are given in Table 2. With a maximum tensor order of 4, every tensor is multiplied with a matrix along every mode using a TTM implementation. Note that the multiplication over the first and fourth mode corresponds to the sixth and seventh TTM case in Table 1 for which TLIB will call a single `gemm`. The multiplication over the second and third mode corresponds to the eighth TTM case where a `gemm` is called multiple times.

Fig. 5 contains bar plots for all tensor shapes of set Q. The size of each bar is the total running time of the respective TTM implementation over all modes that is executed on an Intel Xeon Gold 5318Y CPU and an AMD EPYC 9354 CPU. Note that TCL was not able to compute the TTM for the EXAFEL data set which is why the runtime is set to zero.

On the Intel Xeon Gold 5318Y CPU, TLIB is for most tensor instances faster and reaches a maximum speedup of 137.32% (TCL), 100.80% (TBLIS), 210.71% (LibTorch), 798.91% (Eigen), 581.73% (TuckerMPI). TCL is on par with TLIB for the CESM-ATM data set. TuckerMPI is for the CESM-ATM and Miranda data sets 46.8% and 13.7% faster than TLIB The TTMs of TuckerMPI and LibTorch compute the tensor product for the fourth mode faster than TLIB, independent of the tensor instance.

On the AMD EPYC 9354 CPU, TLIB performs better than most other libraries except for TCL and LibTorch in some instances. TLIB reaches a maximum speedup of 33.36% (TCL), 117.22% (TBLIS), 221.25% (LibTorch), 205.80% (Eigen). TCL outerperforms TLIB by 16.22% (NYX) and 71.65% (Miranda). In this case, TCL computes the tensor product over the fourth mode for almost all tensor instances faster than TLIB. In that case of the SCALE-LETKF data set TCL is 3.4x faster. LibTorch outperforms TLIB for the CESM-ATM data set by 42.02%.

The runtime tests with tensors from the SDRBench demonstrates that for most tensor instances with real-world tensor shapes, TLIB is able to compute the tensor product faster than other libraries.

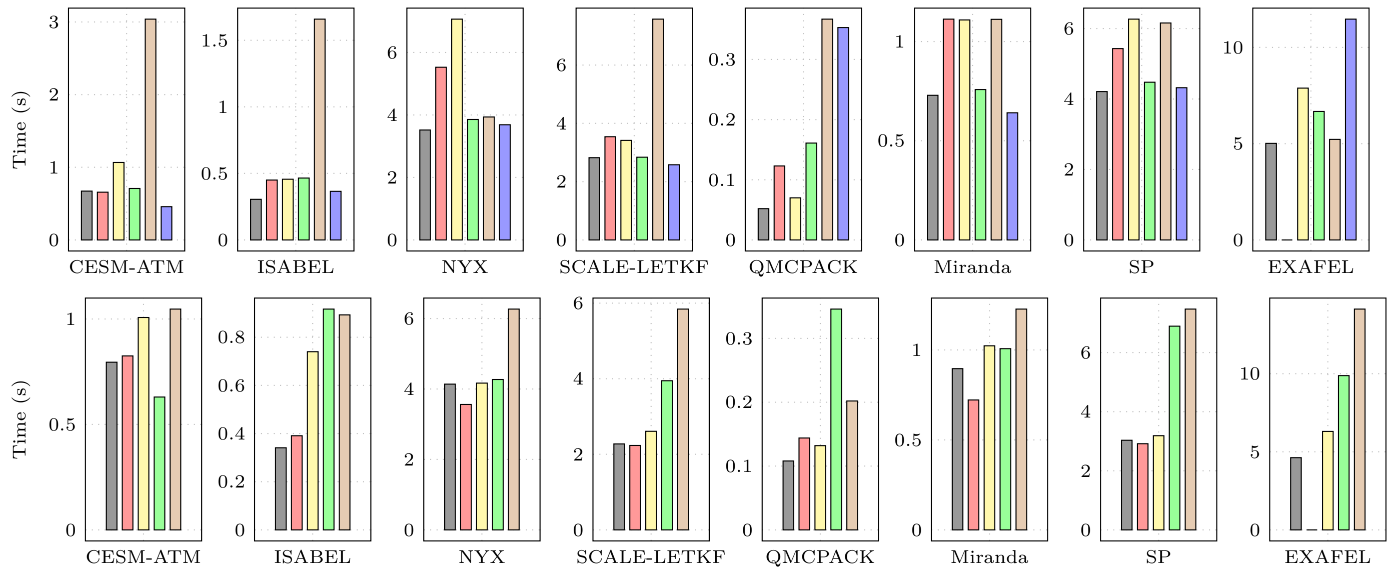

Figure 5: Bar plots contain median runtime in seconds of `TLIB` (), `TCL` (), `TBLIS` (), `LibTorch` (), `Eigen` () and `TuckerMPI` (). The tests were conducted on an Intel Xeon Gold 5318Y CPU (top) and an AMD EPYC 9354 CPU (bottom) using order-3 and order-4 tensors with shapes that are described in Table 2.

7. Summary

We have presented efficient layout-oblivious algorithms for the compute-bound tensor-matrix multiplication that is essential for many tensor methods. Our approach makes use of the LOG method and computes the tensor-matrix product in-place without transposing tensors. It applies the flexible approach described in [16] and generalizes the findings on tensor slicing in [14] for linear tensor layouts. The resulting algorithms are capable of processing dense tensors with arbitrary tensor order, dimensions and with any linear tensor layout all of which can be runtime variable. This degree of flexibility simplifies the integration and application of our library in existing frameworks with different requirements and tensor layouts.

We have presented multiple algorithm variations of the eighth TTM case which either calls a single- or multi-threaded `cblas_gemm` with small or large tensor slices in parallel or sequentially. Additionally, a simple heuristic has been proposed, selecting one of the variants based on the performance evaluation in the original work [1]. We have evaluated all algorithms using a large set of tensor instances with artificial and real-world tensor shapes on an Intel Xeon Gold 5318Y and an AMD EPYC 9354 CPUs. More precisely, we analyzed the impact of performing the `gemm` function with subtensors and tensor slices. Our findings indicate that, subtensors are most effective with symmetrically shaped tensors, irrespective of the parallelization method. Conversely, tensor slices are more advantageous when dealing with asymmetrically shaped tensors, particularly when both the contraction mode and leading dimension are large. Our runtime results demonstrate that parallel executed single-threaded `gemm` performs best with symmetrically shaped tensors. If the leading and contraction dimensions are large, functions with a multi-threaded `gemm` outperforms those with a single-threaded `gemm` for any type of slicing. It can also be observed that function `<combined>` with a simple heuristic performs in most cases as fast as `<par-gemm,subtensor>` and `<par-loop,slice>`, depending on the tensor shape. Function `<batched-gemm>` is less efficient in case of asymmetrically shaped tensors with large contraction and leading dimensions. While matrix storage formats have only a minor impact on TTM performance, runtime measurements show that a TTM using MKL on the Intel Xeon Gold 5318Y CPU achieves higher per-core performance than a TTM with AOCL on the AMD EPYC 9354 processor. We have also demonstrated that our algorithms perform consistently well for k-order tensor layouts, indicating that they are layout-oblivious and do not depend on a specific tensor format.

Our runtime tests with other libraries show that TLIB's `<combined>` version of TTM is either performs equally well or faster than other libraries for the majority of tensor instances. In case of tensors with artificial tensor shapes, TLIB computes the tensor product at least 12.37% faster than LibTorch and Eigen, independent of the processor. TBLIS and TCL achieve a median throughput that is comparable with TLIB when run on the AMD CPU. We observed that most libraries are slower than TLIB for the eighth TTM case across the majority of tensor instances, indicating that our proposed heuristic is efficient. In case of tensors with real-world tensor shapes, TLIB performs better than all libraries for the majority of tensor shapes, reaching a maximum speedup of at least 100.80% in some tensor instances. Exceptions are the CESM-ATM and Miranda data sets where TuckerMPI is 46.8% and 13.7%

faster than TLIB on the Intel CPU. Also TCL is 16.22% and 71.65% faster than TLIB when using the NYX and Miranda data sets on the AMD CPU, respectively.

8. Conclusion and Future Work

The performance tests show that our algorithms are layout-oblivious and do not need layout-specific optimizations, even for different storage ordering of the input matrix. Despite the flexible design, our best-performing algorithm is able to outperform Intel's BLAS-like extension function `cblas_gemm_batch` by a factor of up to 2.57 in case of asymmetrically shaped tensors. Furthermore, the performance results demonstrate that TLIB computes the tensor-matrix product with asymmetrically shaped tensors on average at least 6.21% and up to 334.31% faster than TuckerMPI, LibTorch and Eigen. Our findings leads us to the conclusion that the LOG-based approach is a viable solution for the general tensor-matrix multiplication, capable of matching efficient GETT-based and TGGT-based implementations. Hence, other actively developed libraries such as LibTorch and Eigen might benefit from our algorithm design. Our header-only library provides C++ interfaces and a python module which allows frameworks to easily integrate our library.

In the near future, we intend to incorporate our implementations in TensorLy, a widely-used framework for tensor computations [25, 19]. We also would like to integrate our solution to the TuckerMPI library [21] and investigate the performance of HOSVD algorithms using our approach. Insights provided in [14] could help to further increase the performance. Additionally, we want to explore to what extend our approach can be applied for the general tensor contractions.

8.0.1. Source Code Availability

Project description and source code can be found at `https://github.com/bassoy/ttm`. The sequential tensor-matrix multiplication of TLIB is part of Boost's uBLAS library.

References

[1] C. S. Başsoy, Fast and layout-oblivious tensor-matrix multiplication with blas, in: International Conference on Computational Science, Springer, 2024, pp. 256–271.

[2] E. Karahan, P. A. Rojas-López, M. L. Bringas-Vega, P. A. Valdés-Hernández, P. A. Valdes-Sosa, Tensor analysis and fusion of multimodal brain images, Proceedings of the IEEE 103 (9) (2015) 1531–1559.

[3] E. E. Papalexakis, C. Faloutsos, N. D. Sidiropoulos, Tensors for data mining and data fusion: Models, applications, and scalable algorithms, ACM Transactions on Intelligent Systems and Technology (TIST) 8 (2) (2017) 16.

[4] Q. Song, H. Ge, J. Caverlee, X. Hu, Tensor completion algorithms in big data analytics, ACM Transactions on Knowledge Discovery from Data (TKDD) 13 (1) (Jan. 2019).

[5] H.-M. Rieser, F. Köster, A. P. Raulf, Tensor networks for quantum machine learning, Proceedings of the Royal Society A 479 (2275) (2023) 20230218.

[6] M. Wang, D. Hong, Z. Han, J. Li, J. Yao, L. Gao, B. Zhang, J. Chanussot, Tensor decompositions for hyperspectral data processing in remote sensing: A comprehensive review, IEEE Geoscience and Remote Sensing Magazine 11 (1) (2023) 26–72.

[7] N. Lee, A. Cichocki, Fundamental tensor operations for large-scale data analysis using tensor network formats, Multidimensional Systems and Signal Processing 29 (3) (2018) 921–960.

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

[9] B. W. Bader, T. G. Kolda, Algorithm 862: Matlab tensor classes for fast algorithm prototyping, ACM Trans. Math. Softw. 32 (2006) 635–653.

[10] E. Solomonik, D. Matthews, J. Hammond, J. Demmel, Cyclops tensor framework: Reducing communication and eliminating load imbalance in massively parallel contractions, in: Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on, IEEE, 2013, pp. 813–824.

[11] P. Springer, P. Bientinesi, Design of a high-performance gemm-like tensor–tensor multiplication, ACM Transactions on Mathematical Software (TOMS) 44 (3) (2018) 28.

[12] D. A. Matthews, High-performance tensor contraction without transposition, SIAM Journal on Scientific Computing 40 (1) (2018) C1–C24.

[13] E. D. Napoli, D. Fabregat-Traver, G. Quintana-Ortí, P. Bientinesi, Towards an efficient use of the blas library for multilinear tensor contractions, Applied Mathematics and Computation 235 (2014) 454 – 468.

[14] J. Li, C. Battaglino, I. Perros, J. Sun, R. Vuduc, An input-adaptive and in-place approach to dense tensor-times-matrix multiply, in: High Performance Computing, Networking, Storage and Analysis, 2015, IEEE, 2015, pp. 1–12.

[15] Y. Shi, U. N. Niranjan, A. Anandkumar, C. Cecka, Tensor contractions with extended blas kernels on cpu and gpu, in: 2016 IEEE 23rd International Conference on High Performance Computing (HiPC), 2016, pp. 193–202.

[16] C. Bassoy, Design of a high-performance tensor-vector multiplication with blas, in: International Conference on Computational Science, Springer, 2019, pp. 32–45.

[17] L. De Lathauwer, B. De Moor, J. Vandewalle, A multilinear singular value decomposition, SIAM Journal on Matrix Analysis and Applications 21 (4) (2000) 1253–1278.

[18] F. Pawlowski, B. Uçar, A.-J. Yzelman, A multi-dimensional morton-ordered block storage for mode-oblivious tensor computations, Journal of Computational Science 33 (2019) 34–44.

[19] J. Kossaifi, Y. Panagakis, A. Anandkumar, M. Pantic, Tensorly: Tensor learning in python, Journal of Machine Learning Research 20 (26) (2019) 1–6.

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

[21] G. Ballard, A. Klinvex, T. G. Kolda, Tuckermpi: A parallel c++/mpi software package for large-scale data compression via the tucker tensor decomposition, ACM Transactions on Mathematical Software 46 (2) (6 2020).

[22] L.-H. Lim, Tensors and hypermatrices, in: L. Hogben (Ed.), Handbook of Linear Algebra, 2nd Edition, Chapman and Hall, 2017.

[23] C. Bassoy, V. Schatz, Fast higher-order functions for tensor calculus with tensors and subtensors, in: International Conference on Computational Science, Springer, 2018, pp. 639–652.

[24] K. Zhao, S. Di, X. Lian, S. Li, D. Tao, J. Bessac, Z. Chen, F. Cappello, Sdrbench: Scientific data reduction benchmark for lossy compressors, in: 2020 IEEE International Conference on Big Data (Big Data), 2020, pp. 2716–2724.

[25] J. Cohen, C. Bassoy, L. Mitchell, Ttv in tensorly, Tensor Computations: Applications and Optimization (2022) 11.