Fast Layout-Oblivious Tensor-Matrix Multiplication with BLAS

Cem Savaş Başsoy

Hamburg University of Technology, Schwarzenbergstrasse 95, Germany,
cem.bassoy@gmail.com

Abstract. The tensor-matrix product is a compute-bound tensor operations and required in various tensor methods, e.g. for computing the ALS or HOSVD. This paper presents a high-performance algorithm for the mode-q tensor-matrix multiplication using the Loops-over-GEMMs (LOG) approach with dense tensors that can have any linear tensor layout, tensor order and dimensions. The proposed algorithm either directly calls efficient implementations of GEMM with tensors or recursively apply GEMM on higher-order tensor slices multiple times. We discuss different tensor slicing methods and parallelization strategies using OpenMP with GEMM and batched GEMM. Our best implementation attains a median performance of 1.37 single precision Tflops/s on an Intel Xeon Gold 6248R processor using Intel's MKL. We show that the performance is only slightly affected by the tensor layout and the median performance is between [**?**] and [**?**] Tflops/s for a range of linear tensor formats. Our fastest version of the tensor-matrix multiplication is on average at least 14.05% and up to 3.79 x faster than other state-of-the-art implementations, including Libtorch and Eigen.

Keywords: First keyword · Second keyword · Another keyword.

1 Introduction

Tensor computations are found in many scientific fields such as computational neuroscience, pattern recognition, signal processing and data mining [7,14]. Tensors representing large amount of multidimensional data are decomposed and analyzed with the help of basic tensor operations [8,9]. The decomposition and analysis led to the development and analysis of high-performance kernels for tensor contractions. In this work, we present and analyze a high-performance algorithm for the tensor-matrix multiplication that is used in many numerical algorithms such as the alternating least squares method [8,9]. It is a compute-bound tensor operation and has the same arithmetic intensity as a matrix-matrix multiplication which can reach near peak performance of a computing machine.

To our best knowledge, there has been three main approaches for implementing tensor contractions. The Transpose-Transpose-GEMM-Transpose (TGGT) approach reorganizes (flattens) tensors in order to perform a tensor contraction with an optimized matrix-matrix multiplication (GEMM) implementation [2,16].

Implementations of a more recent method (GETT) are based on high-performance GEMM-like algorithms [1,12,17]. A different method is the LOG approach in which BLAS are utilized with multiple tensor slices or subtensors if possible [10,13,15]. Implementations of the LOG and TTGT approaches are in general easier to maintain and faster to port than GETT implementations which might need to adapt vector instructions or blocking parameters according to a processor's microarchitecture.

Our analysis is motivated by the similarity of LOG-based algorithms for the tensor-matrix and tensor-vector multiplications with the former operation being the generalization of the latter. We therefore applied the approach described in [3] and combined it with the findings presented in [10]. Our proposed recursive in-place algorithms compute the tensor-matrix multiplication by executing GEMMs with general subtensors or tensor slices. They 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. Input and output tensors do not need to be transposed or flattened into two-dimensional matrices. Despite, the generality of our approach, the proposed algorithms are implemented with less than 100 lines of C++ code where the complexity is hidden by the BLAS implementation and the corresponding selection of subtensors or tensor slices.

The following discussion analysis if hopefully providing performance of GETT implementations.

with only eight cases executing

while

The following discussion introduces the base algorithm and variations thereof.

Moreover, except for few corner cases, we demonstrate that our algorithm is able to perform the multiplication with any contraction mode using multiple slice-matrix multiplications and only one GEMM parameter configuration.

, only requiring few cases calling GEMMs routines without transposing the input and output tensors.

While we have used Intel's MKL for our benchmarks, the user is free to choose any library that provides the BLAS interface.

For parallel execution, we propose a variable loop fusion method with respect to the slice order of slice-vector multiplications.

We have quantified the impact of the tensor layout, tensor slice order and parallel execution of slice-matrix multiplications with varying contraction modes. The runtime measurements of our implementations are compared with those presented in [1,12,17]. In summary, the main findings of our work are:

- A tensor-matrix multiplication is implementable by an in-place algorithm with 1 GEMV and 7 GEMM parameter configurations supporting all combinations of contraction mode, tensor order and dimensions.
- Algorithms with variable loop fusion and parallel slice-matrix multiplications can achieve the peak performance of a GEMM with large slice dimensions. Moreover, the proposed algorithm is layout oblivious and is able to achieve a sustainable performance throughput for any linear tensor layout.

– A `LOG`-based tensor-times-matrix implementation can be faster than `TTGT`- and `GETT`-based implementations that have been described in [12, 17]. Using symmetrically shaped tensors, an average speedup of [?] x to [?] x for single and double precision floating point computations can be achieved.

The remainder of the paper is organized as follows. Section 2 presents related work. Section 3 introduces the terminology used in this paper and defines the tensor-vector multiplication. Algorithm design and methods for parallel execution is discussed in Section 4. Section 5 describes the test setup and discusses the benchmark results in Section 6. Conclusions are drawn in Section 7.

2 Related Work

The authors in [13] discuss the efficient tensor contractions with highly optimized `BLAS`. Based on the `LOG` approach, they define requirements for the use of `GEMM` for class 3 tensor contractions and provide slicing techniques for tensors. The slicing recipe for the class 2 categorized tensor contractions contains a short description with a rule of thumb for maximizing performance. Runtime measurements cover class 3 tensor contractions.

The work in [10] presents a framework that generates in-place tensor-matrix multiplication according to the `LOG` approach. The authors present two strategies for efficiently computing the tensor contraction applying `GEMM`s with tensors. They report a speedup of up to 4x over the `TTGT`-based `MATLAB` tensor toolbox library discussed in [2]. Although many aspects are similar to our work, the authors emphasize the code generation of tensor-matrix multiplications using high-performance `GEMM`'s.

The authors of [17] 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. However, their tests do not include the tensor-vector multiplication where the contraction exhibits at least one free tensor index.

Using also the `GETT` approach, the author presents in [12] a runtime flexible tensor contraction library. 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 [17].

3 Background

Notation An order-p tensor is a p-dimensional array [11] where tensor elements are contiguously stored in memory. 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 a tensor order that is greater than 2. The p-tuple $\mathbf{n}$ with $\mathbf{n} = (n_1, n_2, \ldots, n_p)$ will be referred to as a dimension tuple with $n_r > 1$. 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. A subtensor is denoted by $\underline{\mathbf{A}}'$ and references elements of a tensor $\underline{\mathbf{A}}$. They are specified with p index ranges and form a selection grid. In this work, the index range shall either address all indices of a given mode or a single element that are given by single indices i_r with $1 \leq r \leq p$. Elements n'_r of a subtensor's dimension tuple $\mathbf{n}'$ are therefore n_r if all indices of mode r are selected and 1 otherwise. We will annotate subtensors using only their non-unit modes such as $\underline{\mathbf{A}}'_{u,v,w}$ where $n_u > 1$,$n_v > 1$ and $n_w > 1$ and $1 \leq u \neq v \neq w \leq p$. It is sufficient to only provide non-unit modes as the remaining single indices correspond to the loop induction variables of the following algorithms. A subtensor is called a slice $\underline{\mathbf{A}}'_{u,v}$ if the full range selection of $\underline{\mathbf{A}}$ occurs with only two modes. A fiber $\underline{\mathbf{A}}'_u$ is a tensor slice with only one dimension greater than 1.

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 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 a layout tuple $\boldsymbol{\pi}$ with p modes, the π_r-th element of a stride tuple 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.

The location of tensor elements within the allocated memory space is determined by the tensor layout and the corresponding layout function. For a given layout and stride tuple, a layout function $\lambda_{\mathbf{w}}$ maps a multi-index to a scalar index with $\lambda_{\mathbf{w}}(\mathbf{i}) = \sum_{r=1}^{p} w_r(i_r - 1)$. With $j = \lambda_{\mathbf{w}}(\mathbf{i})$ being the relative memory position of an element with a multi-index $\mathbf{i}$, reading from and writing to memory is accomplished with j and the first element's address of $\underline{\mathbf{A}}$.

Non-Modifying Flattening and Reshaping The flattening operation $\varphi_{r,q}$ transforms an order-p tensor $\underline{\mathbf{A}}$ to another order-p' view $\underline{\mathbf{B}}$ that has different a shape $\mathbf{m}$ and layout $\boldsymbol{\tau}$ tuple of length p' with $p' = p - q + r$ and $1 \leq r < q \leq p$. It is related to the tensor unfolding operation as defined in [8, p.459] but neither changes the element ordering nor copies tensor elements. Given a layout tuple $\boldsymbol{\pi}$ of $\underline{\mathbf{A}}$, the flattening operation $\varphi_{r,q}$ is defined for contiguous modes $\hat{\boldsymbol{\pi}} = (\pi_r, \pi_{r+1}, \ldots, \pi_q)$ of $\boldsymbol{\pi}$. Let $j = 0$ if $k \leq r$ and $j = q - r$ otherwise for $1 \leq k \leq p'$. Then the resulting layout tuple $\boldsymbol{\tau} = (\tau_1, \ldots, \tau_{p'})$ of $\underline{\mathbf{B}}$ is given by $\tau_r = \min(\boldsymbol{\pi}_{r,q})$ and $\tau_k = \pi_{k+j} + s_k$ if $k \neq r$ where $s_k = |\{\pi_i \mid \pi_{k+j} > \pi_i \wedge \pi_i \neq \min(\hat{\boldsymbol{\pi}}) \wedge r \leq i \leq p\}|$. Elements of the corresponding shape tuple $\mathbf{m}$ are given by $m_{\tau_r} = \prod_{k=r}^{q} n_{\pi_k}$ and $m_{\tau_k} = n_{\pi_{k+j}}$ if $k \neq r$.

The reshaping operation ρ transforms an order-p tensor $\underline{\mathbf{A}}$ to another order-p tensor $\underline{\mathbf{B}}$ with different shape $\mathbf{m}$ and layout $\boldsymbol{\tau}$ tuples of length p. In this work, it permutes the shape and layout tuple simultaneously without changing the element ordering and without copying tensor elements. The operation ρ uses a permutation tuple $\boldsymbol{\rho} = (\rho_1, \ldots, \rho_p)$ to only modify shape and layout tuples.

Elements of the resulting shape tuple $\mathbf{m}$ and the layout tuple $\boldsymbol{\tau}$ are given by $m_r = n_{\rho_r}$ and $\tau_r = \pi_{\rho_r}$, respectively.

Tensor-Matrix Multiplication (TTM) Let $\underline{\mathbf{A}}$ and $\underline{\mathbf{C}}$ be order-p tensors with shapes $\mathbf{n}_a = (n_1, \ldots, n_q, \ldots, n_p)$ and $\mathbf{n}_c = (n_1, \ldots, n_{q-1}, m, n_{q+1}, \ldots, n_p)$. Let $\mathbf{B}$ be a matrix of shape $\mathbf{n}_b = (m, n_q)$. A mode-q TTM is denoted by $\underline{\mathbf{C}} = \underline{\mathbf{A}} \times_q \mathbf{B}$ where an element of $\underline{\mathbf{C}}$ is given by

$$\underline{\mathbf{C}}(i_1, \ldots, i_{q-1}, j, i_{q+1}, \ldots, i_p) = \sum_{i_q=1}^{n_q} \underline{\mathbf{A}}(i_1, \ldots, i_q, \ldots, i_p) \cdot \mathbf{B}(j, i_q) \tag{1}$$

with $1 \leq i_r \leq n_r$ and $1 \leq j \leq m$. The mode q is the *contraction mode* of the TTM with $1 \leq q \leq p$. The tensor-matrix multiplication 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 and is not memory-bound. 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 stored in either the column-major or row-major format.

4 Algorithm Design

4.1 Sequential Baseline Algorithm

The sequential baseline algorithm implementing Eq. 1 can be implemented with a single C++ function. It consists of nested recursion with a control flow that resembles algorithm 1 in [4], consisting of two `if` statements with an `else` branch. The body of the first `if` statement contains a recursive call that skips the iteration over the dimension n_q when $r = \hat{q}$ with $\pi_r = q$ and $\hat{q} = \boldsymbol{\pi}_q^{-1}$ where $\boldsymbol{\pi}^{-1}$ is the inverse layout tuple. The second `if` statement contains multiple recursive calls for the modes $1 \leq r \neq \hat{q} \leq p$ with different multi-indices. Note that the second `if` statement is skipped for $q = \pi_1$ as the condition of the first one is evaluated to true. The `else` branch is the base case and consists of two loops that compute a fiber-matrix product. The inner loop iterates over the dimension n_q of $\underline{\mathbf{A}}$ and $\mathbf{B}$ with index $1 \leq i_q \leq n_q$ computing an inner product. The outer loop iterates over the dimension m of $\underline{\mathbf{C}}$ and $\mathbf{B}$ with index $1 \leq j \leq m$. The baseline algorithm supports tensors with arbitrary order, dimensions and any non-hierarchical storage format.

4.2 Modified Baseline Algorithm with Contiguous Memory Access

The baseline algorithm accesses memory of $\underline{\mathbf{A}}$ and $\underline{\mathbf{C}}$ non-contiguously whenever $\pi_1 \neq q$ so that indices i_q and j are incremented with steps greater than one. Matrix $\mathbf{B}$ is contiguously accessed if i_q or j is incremented with unit-steps depending on the storage format of $\underline{\mathbf{B}}$. The access pattern could be improved

```
tensor_times_matrix($\underline{\mathbf{A}}, \mathbf{B}, \underline{\mathbf{C}}, \mathbf{n}, \mathbf{i}, m, q, \hat{q}, r$)
    if $r = \hat{q}$ then
        tensor_times_matrix($\underline{\mathbf{A}}, \mathbf{B}, \underline{\mathbf{C}}, \mathbf{n}, \mathbf{i}, m, q, \hat{q}, r-1$)
    else if $r > 1$ then
        for $i_{\pi_r} \leftarrow 1$ to $n_{\pi_r}$ do
            tensor_times_matrix($\underline{\mathbf{A}}, \mathbf{B}, \underline{\mathbf{C}}, \mathbf{n}, \mathbf{i}, m, q, \hat{q}, r-1$)
    else
        for $j \leftarrow 1$ to $m$ do
            for $i_q \leftarrow 1$ to $n_q$ do
                for $i_{\pi_1} \leftarrow 1$ to $n_{\pi_1}$ do
                    $\underline{\mathbf{C}}(i_1, ..., i_{q-1}, j, i_{q+1}, ..., i_p)$ += $\underline{\mathbf{A}}(i_1, ..., i_q, ..., i_p) \cdot \mathbf{B}(j, i_q)$
```

Algorithm 1: Modified baseline algorithm with contiguous memory access for the tensor-matrix multiplication. The tensor order must be greater than one and for the contraction mode $1 \leq q \leq p$ and $\pi_1 \neq q$ must hold. The algorithm needs to be initially called with $r = p$ where $\mathbf{n}$ is the shape tuple of $\underline{\mathbf{A}}$ and m is the q-th dimension of $\underline{\mathbf{C}}$.

by reordering tensor elements according to the storage format which results in copy operations reducing the overall throughput of the operation [15].

A better approach is to access tensor elements according to the tensor layout using the permutation tuple $\boldsymbol{\pi}$ as proposed in [4]. The modified algorithm with contiguous memory accesses is given in algorithm 1 for $\pi_1 \neq q$ and $p > 1$. Each recursion level adjusts only one multi-index element i_{π_r} with a stride w_{π_r} as depicted in line 5. With increasing recursion level and decreasing r, indices are incremented with smaller step sizes as $w_{\pi_r} \leq w_{\pi_{r+1}}$. The condition of the second `if` statement in line 4 is changed from $r \geq 1$ to $r > 1$. In this way, the loop incrementing with index i_{π_1} and the minimum stride w_{π_1} can be included in the base case which contains three loops performing a slice-matrix multiplication. The ordering of the three loops within the base case are adjusted according to the tensor and matrix layout. The inner-most loop increments i_{π_1} and therefore 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. The simple ordering of the three loops is discussed in [5].

While spatial data locality is improved by adjusting the loop ordering, the temporal data locality of tensors $\underline{\mathbf{A}}$ and $\underline{\mathbf{C}}$ differ. Note that slice $\underline{\mathbf{A}}'_{\pi_1,q}$ is accessed m times, fiber $\underline{\mathbf{C}}_{\pi_1}$ is accessed $\mathbf{n}(q)$ times and element $\underline{\mathbf{B}}(j, i_q)$ is accessed $\mathbf{n}(\pi_1)$ times. While the specified fiber of $\underline{\mathbf{C}}$ can 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. Optimized

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

Table 1. Parameter configuration of the `GEMV`- and `GEMM` routines with eight cases computing a tensor-matrix product. The arguments `T`, `M`, `N`, etc. of the BLAS are chosen with respect to the tensor order p, tensor layout $\boldsymbol{\pi}$ and contraction mode q where `T` specifies if $\mathbf{B}$ is transposed. `GEMM*` denotes multiple `GEMM` calls with different tensor slices. The number of rows for case 6 and 7 is given by $\bar{n}_q = (n_1 \cdots n_p)/n_q$.

tiling for better temporal data locality has been discussed in [6] which suggests to use existing high-performance BLAS implementations for the base case.

4.3 BLAS-based Algorithms with Tensor Slices

The proposed algorithm 1 is the starting point for the BLAS-based algorithm which computes the tensor-matrix product with a `GEMM` routine. Besides the illustrated algorithm, we have identified seven other cases where a single `GEMM` call suffices to compute the tensor-matrix product even if the tensor order p is greater than two. In summary there are eight cases with a single `GEMM` call using different arguments which are listed in table 1. The list of `GEMM` calls is complete with no limitation on tensor order and contraction mode, supporting all linear tensor layout. `GEMM` arguments are chosen depending on the tensor order p, tensor layout $\boldsymbol{\pi}$ and contraction mode q except for the `CBLAS_ORDER` which is `CblasRowMajor`.

Case 1 ($p = 1$)*:* The tensor-vector product $\underline{\mathbf{A}} \times_1 \mathbf{B}$ can be computed with a `GEMV` operation $\mathbf{a}^T \cdot \mathbf{B}$ where $\underline{\mathbf{A}}$ is an order-1 tensor, i.e. a vector $\mathbf{a}$ of length n_1.

Case 2-5 ($p = 2$)*:* If $\underline{\mathbf{A}}$ and $\underline{\mathbf{C}}$ are order-2 tensors, i.e. a matrix $\mathbf{A}$ with dimensions n_1 and n_2, then a single `GEMM` suffices to compute the tensor-matrix product. 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 $q = 2$. Note that `GEMM` interprets $\mathbf{C}$ and $\mathbf{A}$ as matrices using the reshaping operation ρ with $\boldsymbol{\rho} = (2,1)$ in row-major format even though 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 $q = 2$. Note that the transposition of $\mathbf{B}$ is necessary for the cases 2,5 and independent of the chosen storage format.

Case 6-7 ($p > 2$)*:* If the order of $\underline{\mathbf{A}}$ and $\underline{\mathbf{C}}$ is greater than 2 and if the contraction mode q is equal to π_1 (case 6), a single `GEMM` with the depicted parameters executes $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}^T$ and computes a tensor-matrix product $\underline{\mathbf{C}} = \underline{\mathbf{A}} \times_{\pi_1} \mathbf{B}$ for any storage layout of $\underline{\mathbf{A}}$ and $\underline{\mathbf{C}}$. Tensors $\underline{\mathbf{A}}$ and $\underline{\mathbf{C}}$ are flattened 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), Tensors $\underline{\mathbf{A}}$ and $\underline{\mathbf{C}}$ are flattened 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 matrix $\mathbf{C}$ has m rows and the same number of columns. A single `GEMM` executes $\mathbf{C} = \mathbf{B} \cdot \mathbf{A}$ and computes the tensor-matrix product $\underline{\mathbf{C}} = \underline{\mathbf{A}} \times_{\pi_p} \mathbf{B}$ for any storage layout of $\underline{\mathbf{A}}$ and $\underline{\mathbf{C}}$. Note that in all cases no copy operation is performed in order to compute the desired contraction, see subsection 3.

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 $\underline{\mathbf{A}}$ in the base case. 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.

4.4 BLAS-Based Algorithms with Subtensors

It is possible to further optimize case 8 by selecting larger subtensors instead of slices and enable higher processor utilization by executing multiple `GEMM`s with larger matrices. Note that the base case of the modified baseline algorithm 1 calls slice-matrix multiplication with slices of which two dimensions are greater than one, i.e. n_q, n_{π_1} and m, n_{π_1} for $\underline{\mathbf{A}}$ and $\underline{\mathbf{C}}$, respectively. In order to use larger subtensors, additional mergeable modes must be selected that still allow the subtensor to be flattened into a matrix without reordering tensor elements, see section 3. The maximum number of mergeable modes is $\hat{q}-1$ with $\hat{q} = \boldsymbol{\pi}^{-1}(q)$ and the corresponding modes are $\pi_1, \pi_2, \ldots, \pi_{\hat{q}-1}$. Applying flattening $\varphi_{1,q-1}$ and reshaping ρ with $\boldsymbol{\rho} = (2,1)$ on a subtensor of $\underline{\mathbf{A}}$ with dimensions $n_{\pi_1}, \ldots, n_{\pi_{\hat{q}-1}}, n_q$ yields a row-major matrix $\mathbf{A}$ with shape $(n_q, \prod_{r=1}^{\hat{q}-1} n_{\pi_r})$. This is done analogously for $\underline{\mathbf{C}}$ resulting in a row-major matrix with shape $(m, \prod_{r=1}^{\hat{q}-1} n_{\pi_r})$.

Algorithm 1 needs a minor modification that allow each `GEMM` to be called with flattened subtensors of $\underline{\mathbf{A}}$ and $\underline{\mathbf{C}}$. The modified algorithm is omitted the first $\hat{q}$ modes $\boldsymbol{\pi}_{1,\hat{q}} = (\pi_1, \ldots, \pi_{\hat{q}})$ including $\pi_{\hat{q}} = q$. This is done by only iterating over modes in the non-base case of the algorithm that are larger than $\hat{q}$. The conditions in line 2 and 4 are therefore changed to $1 < r \leq \hat{q}$ and $\hat{q} < r$, respectively. The single indices of the subtensors $\underline{\mathbf{A}}'_{\boldsymbol{\pi}_{1,\hat{q}}}$ and $\underline{\mathbf{C}}'_{\boldsymbol{\pi}_{1,\hat{q}}}$ within the base case of the algorithm are given by the loop induction variables that belong to the π_r-th loop with $\hat{q}+1 \leq r \leq p$. Subtensor elements are contiguously stored and the number of non-unit dimensions is equal to $\hat{q}$.

4.5 Parallel BLAS-based Algorithms

Note that cases 1 to 7 only call a single `GEMM` and cannot be further parallelized. Hence, we focus on the previously described algorithm which sequentially calls multi-threaded `GEMM`s. This is beneficial if $q = \pi_{p-1}$, the inner dimensions $n_{\pi_1}, \ldots, n_q$ are large or the outer-most dimension n_{π_p} is smaller than the available processor cores. If the above conditions are not met, the processor cores might fully utilized where each multi-threaded `GEMM` is executed with small subtensors. The algorithm will be referred to as the **multithreaded gemm** with **subtensors** or tensor **slices**.

A more advanced version of the above algorithm is to execute single-threaded `GEMM`s in parallel including all available (free) modes for the parallelization. The general subtensor-matrix approach contains $\pi_{\hat{q}+1}, \ldots, \pi_p$ free modes whereas in case of the slice-matrix multiplications, all modes except π_1 and $\pi_{\hat{q}}$ are free. Their respective maximum degree of parallelism is $\prod_{r=\hat{q}+1}^{p} n_{\pi_r}$ and $\prod_{r=1}^{p} n_r / (n_{\pi_1} n_{\pi_{\hat{q}}})$. These free modes can be utilized by directly parallelizing the corresponding (free) loops with the tasking concept of OpenMP. We have placed the `taskloop` directive in front of the loop in line 5 which allows to partition all free loops into tasks for parallel execution. This allows to collapse iterations of all free loops within the desired recursion levels into one larger divisible iteration space. One advantage of this approach is that both tensors or subtensors do not need to be flattened or reshaped and the existing algorithm can be reused. We will refer to as the **taskloops** over **sequential gemm** with subtensors or slices.

Alternatively, it is possible to flatten the tree-recursion and collapse all loops into one or two loops depending on the available fusible loops. Considering the slice-matrix approach, both tensors are flattened twice with $\varphi_{\pi_{\hat{q}+1},\pi_p}$ and $\varphi_{\pi_2,\pi_{\hat{q}-1}}$. The resulting tensor is of order 4 with dimensions $n_{\pi_1}, \hat{n}_{\pi_2}, n_q, \hat{n}_{\pi_4}$ where $\hat{n}_{\pi_2} = \prod_{r=2}^{\hat{q}-1} n_{\pi_r}$ and $\hat{n}_{\pi_4} = \prod_{r=\hat{q}+1}^{p} n_{\pi_r}$. The corresponding algorithm consists of two nested loops. The outer loop iterates over $\hat{n}_{\pi_4}$ while the inner loop iterates over $\hat{n}_{\pi_2}$ calling `GEMM` with multiple slices $\underline{\mathbf{A}}'_{\pi_1,q}$ and $\underline{\mathbf{C}}'_{\pi_1,q}$. The two loops are parallelized using the `parallel for` together with the `collapse(2)` clause. In case of the general subtensor-matrix approach, both tensors are flattened twice with $\varphi_{\pi_{\hat{q}+1},\pi_p}$ and $\varphi_{\pi_1,\pi_{\hat{q}-1}}$. The resulting tensor is of order 3 with dimensions $\hat{n}_{\pi_1}, n_q, \hat{n}_{\pi_4}$ where $\hat{n}_{\pi_1} = \prod_{r=1}^{\hat{q}-1} n_{\pi_r}$ and $\hat{n}_{\pi_4} = \prod_{r=\hat{q}+1}^{p} n_{\pi_r}$. The corresponding algorithm consists of one loops which iterates over $\hat{n}_{\pi_4}$ calling `GEMM` with multiple subtensors $\underline{\mathbf{A}}'_{\boldsymbol{\pi}',q}$ and $\underline{\mathbf{C}}'_{\boldsymbol{\pi}',q}$ with $\boldsymbol{\pi}' = (\pi_1, \ldots, \pi_{\hat{q}-1})$. We will refer to as the **parallel loops** over **sequential gemm** with subtensors or slices.

The next algorithm is a modified version of the general subtensor-matrix approach that uses a single batched `GEMM` for the eighth case. We will refer to as the **batched gemm** with subtensors.

5 Experimental Setup

Computing System The experiments have been carried out on an Intel Xeon Gold 6248R processor with a Cascade micro-architecture. The processor consists of 24 cores operating at a base frequency of 3 GHz for non-AVX512 instructions. With 24 cores and a peak AVX-512 boost frequency of 2.5 GHz, the processor achieves a theoretical data throughput of ca. 1.92 double precision TFlops/s We measured a peak performance of 1.78 double precision Tflops/s using the likwid performance tool [?] .

The source code has been compiled with `GCC v10.2` using the highest optimization level `-Ofast` and `-march=native`, `-pthread` and `-fopenmp`. Loops within for the eighth case have been parallelized using `GCC`'s `OpenMP v4.5` implementation. We have used the `GEMV` and `GEMM` implementation of the 2024.0 Intel oneAPI Math Kernel Library (oneMKL) and its own threading library `mkl_intel_thread` together with the threading runtime library `libiomp5`. The benchmark results of each function are the average of 10 runs.

Tensor Shapes We have used asymmetrically-shaped and symmetrically-shaped tensors in order to cover many possible use cases. The dimension tuples of both shape types are organized within two three-dimensional arrays with which tensors are initialized. The dimension array for the first shape type contains $720 = 9 \times 8 \times 10$ dimension tuples where the row number is the tensor order ranging from 2 to 10. For each tensor order 8 tensor instances with increasing tensor size is generated. The second set consists of $336 = 6 \times 8 \times 7$ dimensions tuples where the tensor order ranges from 2 to 7 and has 8 dimension tuples for each order. Each tensor dimension within the second set is 2^{12}, 2^8, 2^6, 2^5, 2^4 and 2^3. A detailed explanation of the tensor shape setup is given in [3, 4].

6 Results and Discussion

Matrix-Vector Multiplication Fig. 1 shows average performance values of the four versions `SB-P1`, `LB-P1`, `SB-PN` and `LB-PN` with asymmetrically-shaped tensors. In case 2 (region 2), the shape tuple of the two-order tensor is equal to (n_2, n_1) where n_2 is set to 1024 and n_1 is $c \cdot 2^{14}$ for $1 \leq c \leq 32$. In case 6 (region 6), the p-order tensor is interpreted as a matrix with a shape tuple $(\bar{n}_1, n_1)$ where n_1 is $c \cdot 2^{15-r}$ for $1 \leq c \leq 32$ and $2 < r < 10$. The mean performance averaged over the matrix sizes is around 30 Gflops/s in single-precision for both cases. When $p = 2$ and $q > 1$, all functions execute case 3 with a single parallel `GEMV` where the 2-order tensor is interpreted as a matrix in column-major format with a shape tuple (n_1, n_2). In this case, the performance is 16 Gflops/s in region 3 where the first dimension of the 2-order tensor is equal to 1024 for all tensor sizes. The performance of `GEMV` increases in region 7 with increasing tensor order and increasing number of rows $\bar{n}_q$ of the interpreted p-order tensor. In general, `OpenBLAS`'s `GEMV` provides a sustained performance around 31 Gflops/s in single precision for column- and row-major matrices. However, the performance drops with decreasing number of rows and columns for the column-major and row-major format. The performance of case 8 within region 8 is analyzed in the next paragraph.

Slicing and Parallelism Functions with `P1` run with 10 Gflops/s in region 8 when the contraction mode q is chosen smaller than or equal to the tensor order

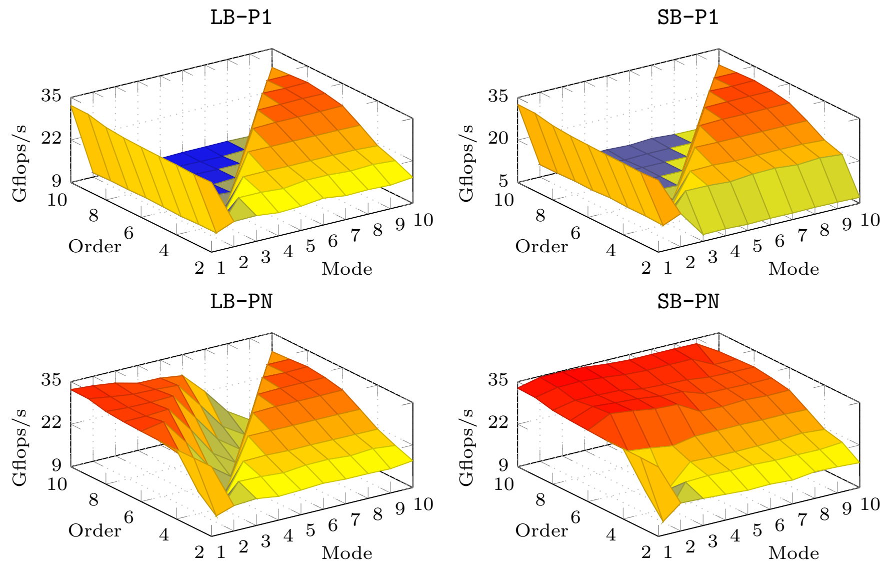

Fig. 1. Average performance maps of four tensor-vector multiplications with varying tensor orders p and contraction modes q. Tensor elements are encoded in single-precision and stored contiguously in memory according to the first-order storage format. Tensors are *asymmetrically-shaped* with dimensions.

p. The degree of parallelism diminishes for $n_p = 2$ as only 2 threads sequentially execute a `GEMV`. The second method `PN` fuses additional loops and is able to generate a higher degree of parallelism. Using the first-order storage format, the outer dimensions $n_{q+1}, \ldots, n_p$ are executed in parallel. The `PN` version speeds up the computation by almost a factor of 4x except for $q = p - 1$. This explains the notch in the left-bottom plot when $q = p - 1$ and $n_p = 2$.

In contrast to the `LB` slicing method, `SB` is able to additionally fuse the inner dimensions with their respective indices $2, 3, \ldots, p-2$ for $q = p - 1$. The performance drop of the `LB` version can be avoided, resulting in a degree of parallelism of $\prod_{r=2}^{p} n_r / n_q$. Executing that many small slice-vector multiplications with a `GEMV` in parallel yields a mean peak performance of up to 34.8(15.5) Gflops/s in single(double) precision. Around 60% of all 2880 measurements exhibit at least 32 Gflops/s that is `GEMV`'s peak performance in single precision. In case of symmetrically-shaped tensors, both approaches achieve similar results with almost no variation of the performance achieving up on average 26(14) Gflops/s in single(double) precision.

Tensor Layouts Applying the first setup configuration with asymmetrically-shaped tensors, we have analyzed the effects of the blocking and parallelization strategy. The `LB-PN` version processes tensors with different storage formats, namely the 1-, 2-, 9- and 10-order layout. The performance behavior is almost

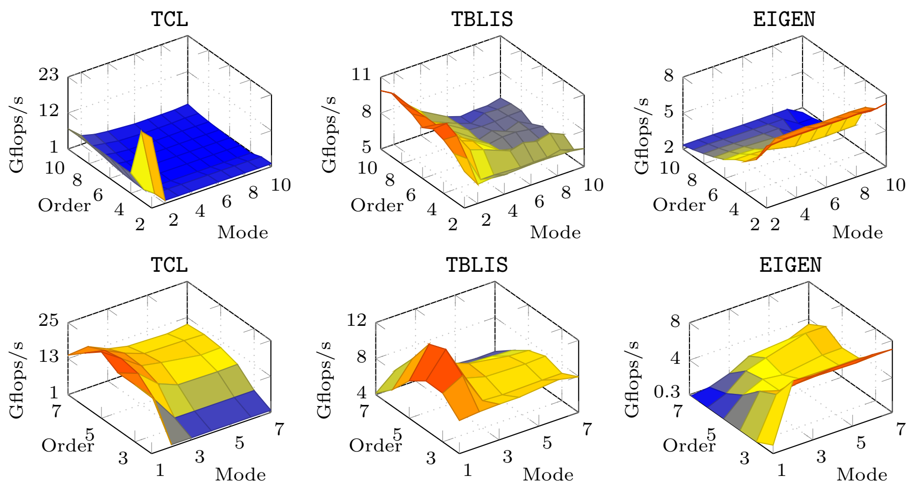

Fig. 2. Average performance maps of tensor-vector multiplication implementations using *asymmetrically-shaped* (top) and *symmetrically-shaped* (bottom) tensors with varying contraction modes and tensor order. Tensor elements are encoded in single-precision and stored contiguously in memory according to the first-order storage format.

the same for all storage formats except for the corner cases $q = \pi_1$ and $q = \pi_p$. Even the performance drop for $q = p - 1$ is almost unchanged. The standard deviation from the mean value is less than 10% for all storage formats. Given a contraction mode $q = \pi_k$ with $1 < k < p$, a permutation of the inner and outer tensor dimensions with their respective indices $\pi_1, \ldots, \pi_{k-1}$ and $\pi_{k+1}, \ldots, \pi_p$ does influence the runtime where the `LB-PN` version calls `GEMV` with the values w_m and n_m. The same holds true for the outer layout tuple.

Comparison with other Approaches The following comparison includes three state-of-the-art libraries that implement three different approaches. The library `TCL` (`v0.1.1`) implements the (`TTGT`) approach with a high-perform tensor-transpose library `HPTT` which is discussed in [17]. `TBLIS` (`v1.0.0`) implements the `GETT` approach that is akin to `BLIS`'s algorithm design for matrix computations [12]. The tensor extension of `EIGEN` (`v3.3.90`) is used by the Tensorflow framework and performs the tensor-vector multiplication in-place and in parallel with contiguous memory access [1]. `TLIB` denotes our library that consists of sequential and parallel versions of the tensor-vector multiplication. Numerical results of `TLIB` have been verified with the ones of `TCL`, `TBLIS` and `EIGEN`.

Fig. 2 illustrates the average single-precision Gflops/s with asymmetrically- and symmetrically-shaped tensors in the first-order storage format. The runtime behavior of `TBLIS` and `EIGEN` with asymmetrically-shaped tensors is almost constant for varying tensor sizes with a standard deviation ranging between 2% and 13%. `TCL` shows a different behavior with 2 and 4 Gflops/s for any order $p \geq 2$ peaking at $p = 10$ and $q = 2$. The performance values however deviate

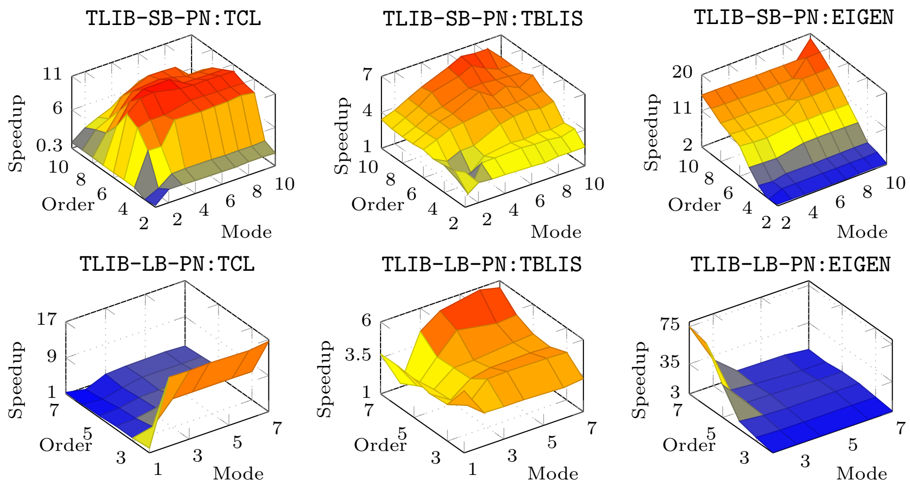

Fig. 3. Relative average performance maps of tensor-vector multiplication implementations using *asymmetrically* (top) and *symmetrically* (bottom) shaped tensors with varying contraction modes and tensor order. Relative performance (speedup) is the performance ratio of `TLIB-SB-PN` (top) and `TLIB-LB-PN` (bottom) to `TBLIS`, `TCL` and `EIGEN`, respectively. Tensor elements are encoded in single-precision and stored contiguously in memory according to the first-order storage format.

from the mean value up to 60%. Computing the arithmetic mean over the set of contraction modes yields a standard deviation of less than 10% where the performance increases with increasing order peaking at $p = 10$. `TBLIS` performs best for larger contraction dimensions achieving up to 7 Gflops/s and slower runtimes with decreasing contraction dimensions. In case of symmetrically-shaped tensors, `TBLIS` and `TCL` achieve up to 12 and 25 Gflops/s in single precision with a standard deviation between 6% and 20%, respectively. `TCL` and `TBLIS` behave similarly and perform better with increasing contraction dimensions. `EIGEN` executes faster with decreasing order and increasing contraction mode with at most 8 Gflops/s at $p = 2$ and $q \geq 2$.

Fig. 3 illustrates relative performance maps of the same tensor-vector multiplication implementations. Comparing `TCL` performance, `TLIB-SB-PN` achieves an average speedup of 6x and more than 8x for 42% of the test cases with asymmetrically shaped tensors and executes on average 5x faster with symmetrically shaped tensors. In comparison with `TBLIS`, `TLIB-SB-PN` computes the tensor-vector product on average 4x and 3.5x faster for asymmetrically and symmetrically shaped tensors, respectively.

7 Conclusion and Future Work

Based on the `LOG` approach, we have presented in-place and parallel tensor-vector multiplication algorithms of `TLIB`. Using highly-optimized `DOT` and `GEMV` routines of OpenBLAS, our proposed algorithms is designed for dense tensors with arbitrary order, dimensions and any non-hierarchical storage format. TLIB's algorithms either directly call DOT, GEMV or recursively perform parallel slice-vector multiplications using GEMV with tensor slices and fibers.

Our findings show that loop-fusion improves the performance of TLIB's parallel version on average by a factor of 5x achieving up to 34.8/15.5 Gflops/s in single/double precision for asymmetrically shaped tensors. With symmetrically shaped tensors resulting in small contraction dimensions, the results suggest that higher-order slices with larger dimensions should be used. We have demonstrated that the proposed algorithms compute the tensor-vector product on average 6.1x and up to 12.6x faster than the TTGT-based implementation provided by TCL. In comparison with TBLIS, TLIB achieves speedups on average of 4.0x and at most 10.4x. In summary, we have shown that a LOG-based tensor-vector multiplication implementation can outperform current implementations that use a TTGT and GETT approaches.

In the future, we intend to design and implement the tensor-matrix multiplication with the same requirements also supporting tensor transposition and subtensors. Moreover, we would like to provide an in-depth analysis of LOG-based implementations of tensor contractions with higher arithmetic intensity.

Project and Source Code Availability TLIB has evolved from the Google Summer of Code 2018 project for extending Boost's uBLAS library with tensors. Project description and source code can be found at https://github.com/bassoy/ttv. The sequential tensor-vector multiplication of TLIB is part of uBLAS and in the official release of Boost v1.70.0.

Acknowledgements The author would like to thank Volker Schatz and Banu Sözüar for proofreading. He also thanks Michael Arens for his support.

References

1. Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., ..., Zheng, X.: Tensorflow: A system for large-scale machine learning. In: Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation. pp. 265–283. OSDI'16, USENIX Association, Berkeley, CA, USA (2016)
2. Bader, B.W., Kolda, T.G.: Algorithm 862: Matlab tensor classes for fast algorithm prototyping. ACM Trans. Math. Softw. **32**, 635–653 (December 2006)
3. Bassoy, C.: Design of a high-performance tensor-vector multiplication with blas. In: International Conference on Computational Science. pp. 32–45. Springer (2019)
4. Bassoy, C., Schatz, V.: Fast higher-order functions for tensor calculus with tensors and subtensors. In: International Conference on Computational Science. pp. 639–652. Springer (2018)
5. Golub, G.H., Van Loan, C.F.: Matrix Computations. JHU Press, 4 edn. (2013)
6. Goto, K., Geijn, R.A.v.d.: Anatomy of high-performance matrix multiplication. ACM Transactions on Mathematical Software (TOMS) **34**(3) (2008)

7. Karahan, E., Rojas-López, P.A., Bringas-Vega, M.L., Valdés-Hernández, P.A., Valdes-Sosa, P.A.: Tensor analysis and fusion of multimodal brain images. Proceedings of the IEEE **103**(9), 1531–1559 (2015)
8. Kolda, T.G., Bader, B.W.: Tensor decompositions and applications. SIAM review **51**(3), 455–500 (2009)
9. Lee, N., Cichocki, A.: Fundamental tensor operations for large-scale data analysis using tensor network formats. Multidimensional Systems and Signal Processing **29**(3), 921–960 (2018)
10. Li, J., Battaglino, C., Perros, I., Sun, J., Vuduc, R.: An input-adaptive and in-place approach to dense tensor-times-matrix multiply. In: High Performance Computing, Networking, Storage and Analysis, 2015 SC-International Conference for. pp. 1–12. IEEE (2015)
11. Lim, L.H.: Tensors and hypermatrices. In: Hogben, L. (ed.) Handbook of Linear Algebra. Chapman and Hall, 2 edn. (2017)
12. Matthews, D.A.: High-performance tensor contraction without transposition. SIAM Journal on Scientific Computing **40**(1), C1–C24 (2018)
13. Napoli, E.D., Fabregat-Traver, D., Quintana-Ortí, G., Bientinesi, P.: Towards an efficient use of the blas library for multilinear tensor contractions. Applied Mathematics and Computation **235**, 454 – 468 (2014)
14. Papalexakis, E.E., Faloutsos, C., Sidiropoulos, N.D.: Tensors for data mining and data fusion: Models, applications, and scalable algorithms. ACM Transactions on Intelligent Systems and Technology (TIST) **8**(2), 16 (2017)
15. Shi, Y., Niranjan, U.N., Anandkumar, A., Cecka, C.: Tensor contractions with extended blas kernels on cpu and gpu. In: 2016 IEEE 23rd International Conference on High Performance Computing (HiPC). pp. 193–202 (Dec 2016)
16. Solomonik, E., Matthews, D., Hammond, J., Demmel, J.: Cyclops tensor framework: Reducing communication and eliminating load imbalance in massively parallel contractions. In: Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on. pp. 813–824. IEEE (2013)
17. Springer, P., Bientinesi, P.: Design of a high-performance gemm-like tensor–tensor multiplication. ACM Transactions on Mathematical Software (TOMS) **44**(3), 28 (2018)