

# Fast Layout-Oblivious Tensor-Matrix Multiplication with BLAS

Cem Bassoy

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 strategies for fusing and executing the matrix-matrix multiplication in parallel. Using OpenBLAS, our parallel implementation attains [?] Gflops/s in single precision on a Core i9-7900X Intel Xeon processor. We show that the performance of our implementation is independent of the tensor layout and a performance of [?] can be sustained for any linear tensor format. Our version of the tensor-matrix multiplication is on average [?] x and up to [?] x faster than state-of-the-art approaches.

## 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 approach to implement tensor contractions. The Transpose-Transpose-GEMM-Transpose (TGGT) approach reorganizes (flatens) 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 algorithms utilize GEMM with multiple tensor slices if possible [10, 13, 15].

Our analysis is motivated by the observation that LOG implementations of the tensor-matrix multiplication has not been fully thoroughly investigated. Our approach is akin to the one proposed in [10, 15] but targets the utilization of general matrix-matrix multiplication routines (GEMM) using OpenBLAS, Intel MKL and BLIS without code generation. The recursive in-place algorithms is similar to the one presented in [3] and computes the tensor-matrix multiplication by executing GEMM with slices and fibers of tensors. However, the presented algorithm requires twice as many cases and has also additional implementation options which has not been previously discussed. 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. For parallel execution, we propose a variable loop fusion method with respect to the slice order of slice-vector multiplications. 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. 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  $[?] \times$  to  $[?] \times$  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 GEMMs 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. In general we assume a tensor  $\underline{\mathbf{A}}$  to have a tensor order with  $p > 2$ . The  $p$ -tuple  $\mathbf{n}$  with  $\mathbf{n} = (n_1, n_2, \dots, n_p)$  will be referred to as a dimension tuple with  $n_r > 1$ . We will use round brackets  $\underline{\mathbf{A}}(i_1, i_2, \dots, i_p)$  or  $\underline{\mathbf{A}}(\mathbf{i})$  to denote a tensor element where  $\mathbf{i} = (i_1, i_2, \dots, i_p)$  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  $\boldsymbol{\pi}_F = (1, 2, \dots, p)$  and

$\pi_L = (p, p-1, \dots, 1)$ . An inverse layout tuple  $\pi^{-1}$  is defined by  $\pi^{-1}(\pi(k)) = k$ . Given a layout tuple  $\pi$  with  $p$  modes, the  $\pi_r$ -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  $\pi_1$ -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  $\tau$  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  $\pi$  of  $\underline{\mathbf{A}}$ , the flattening operation  $\varphi_{r,q}$  is defined for contiguous modes  $\hat{\pi} = (\pi_r, \pi_{r+1}, \dots, \pi_q)$  of  $\pi$ . Let  $j = 0$  if  $k \leq r$  and  $j = q - r$  otherwise for  $1 \leq k \leq p'$ . Then the resulting layout tuple  $\tau = (\tau_1, \dots, \tau_{p'})$  of  $\underline{\mathbf{B}}$  is given by  $\tau_r = \min(\pi_{r,q})$  and  $\tau_k = \pi_{k+j} + s_k$  if  $k \neq r$  where  $s_k = |\{\pi_i \mid \pi_{k+j} > \pi_i \wedge \pi_i \neq \min(\hat{\pi}) \wedge r \leq i \leq p\}|$ . 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  $\rho$  transforms an order- $p$  tensor  $\underline{\mathbf{A}}$  to another order- $p$  tensor  $\underline{\mathbf{B}}$  with different shape  $\mathbf{m}$  and layout  $\tau$  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  $\rho$  uses a permutation tuple  $\rho = (\rho_1, \dots, \rho_p)$  to only modify shape and layout tuples. Elements of the resulting shape tuple  $\mathbf{m}$  and the layout tuple  $\tau$  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  $\mathbf{n}_a = (n_1, \dots, n_q, \dots, n_p)$  and  $\mathbf{n}_c = (n_1, \dots, n_{q-1}, m, n_{q+1}, \dots, n_p)$ . Let  $\underline{\mathbf{B}}$  be a matrix of shape  $\mathbf{n}_b = (m, n_q)$ . A mode- $q$  TTM is denoted by  $\underline{\mathbf{C}} = \underline{\mathbf{A}} \times_q \underline{\mathbf{B}}$  where an element of  $\underline{\mathbf{C}}$  is given by

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

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  $\underline{\mathbf{C}} = \underline{\mathbf{B}} \cdot \underline{\mathbf{A}}$  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  $\pi$ . Elements of matrix  $\underline{\mathbf{B}}$  can be 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  $\hat{q} = \pi_q^{-1}$  where  $\pi^{-1}$  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 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  $\pi$  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  $i_{\pi_r}$  with a stride  $w_{\pi_r}$  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  $i_{\pi_1}$  and the minimum stride  $w_{\pi_1}$  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  $i_{\pi_1}$  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  $\mathbf{n}(\pi_1)$

---

```

1 tensor_times_matrix(A, B, C, n, i, m, q, q̂, r)
2   if  $r = \hat{q}$  then
3     | tensor_times_matrix(A, B, C, n, i, m, q, q̂,  $r - 1$ )
4   else if  $r > 1$  then
5     | for  $i_{\pi_r} \leftarrow 1$  to  $n_{\pi_r}$  do
6       | | tensor_times_matrix(A, B, C, n, i, m, q, q̂,  $r - 1$ )
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  $n_{\pi_1}$  do
11          | | | | C( $i_1, \dots, i_{q-1}, j, i_{q+1}, \dots, i_p$ ) += A( $i_1, \dots, i_q, \dots, i_p$ ) · 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  $\underline{\mathbf{C}}$ .

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 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  $\pi$  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  $\pi = (1, 2)$ , **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  $\rho$  with  $\rho = (2, 1)$  in row-major format even though both are stored column-wise. If  $\mathbf{A}$  and  $\mathbf{C}$  have the

Case	Order $p$	Layout $\pi$	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	$\pi_1$	GEMM	$\mathbf{B}$	$\bar{n}_q$	$m$	$n_q$	$\underline{\mathbf{A}}$	$n_q$	$\mathbf{B}$	$n_q$	$m$
7	$> 2$	any	$\pi_p$	GEMM	-	$m$	$\bar{n}_q$	$n_q$	$\mathbf{B}$	$n_q$	$\underline{\mathbf{A}}$	$\bar{n}_q$	$\bar{n}_q$
8	$> 2$	any	$\pi_2, \dots, \pi_{p-1}$	GEMM*	-	$m$	$n_{\pi_1}$	$n_q$	$\mathbf{B}$	$n_q$	$\underline{\mathbf{A}}$	$n_{\pi_1}$	$n_{\pi_1}$

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

row-major format with  $\pi = (2, 1)$ , 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  $\pi_1$  (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  $n_{\pi_1}$  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  $n_{\pi_p}$  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  $(n_q, n_{\pi_1})$  and  $(m, n_{\pi_1})$ , 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 GEMMs 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.  $n_q, n_{\pi_1}$  and  $m, n_{\pi_1}$  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  $\hat{q} = \pi^{-1}(q)$  and the corresponding modes are  $\pi_1, \pi_2, \dots, \pi_{\hat{q}-1}$ . Applying flattening  $\varphi_{1, q-1}$  and reshaping  $\rho$  with  $\rho = (2, 1)$  on a subtensor of  $\underline{\mathbf{A}}$  with dimensions  $n_{\pi_1}, \dots, n_{\pi_{\hat{q}-1}}, n_q$  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  $\pi_{1, \hat{q}} = (\pi_1, \dots, \pi_{\hat{q}})$  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  $\underline{\mathbf{A}}'_{\pi_{1, \hat{q}}}$  and  $\underline{\mathbf{C}}'_{\pi_{1, \hat{q}}}$  within the base case of the algorithm are given by the loop induction variables that belong to the  $\pi_r$ -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

The previously described algorithm sequentially calls multi-threaded **GEMMs**. This is beneficial if the contraction mode  $q$  is large  $q = \pi_{p-1}$ , the inner dimensions  $n_{\pi_1}, \dots, n_q$  are large or the outer-most dimension  $n_{\pi_p}$  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.

Hence, we propose a parallel version of the above algorithm which executes single-threaded **GEMMs** in parallel. Note that cases 1 to 7 only call a single **GEMM** and cannot be further parallelized. A simple approach is to parallelize the outer-most loop  $\pi_p$  using the OpenMP `parallel for` directive which allows OpenMP threads to execute their own sequential subtensor-matrix multiplications independently. However, the degree of parallelism is limited by the outer-most dimension  $n_{\pi_p}$ . Hence, more available modes need to be considered to increase parallelism. The general subtensor-matrix approach contains  $\pi_{\hat{q}+1}, \dots, \pi_p$  free modes whereas in case of the slice-matrix multiplications, all modes except  $\pi_1$  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.

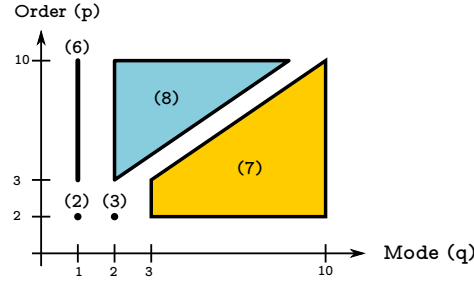


Alternatively, it is possible to fuse available free loops and then to parallelize the resulting loop using either the OpenMP `parallel for` directive with flattened subtensors or to directly call batched matrix multiplication routines, e.g. `BLAS_batched_gemm` if such a routine is available. Considering the slice-matrix algorithm, 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 slices  $\underline{\mathbf{A}}'_{\pi_1, q}$  and  $\underline{\mathbf{C}}'_{\pi_1, q}$ .

## 5 Experimental Setup

**Computing System** The experiments were carried out on a Core i9-7900X Intel Xeon processor with 10 cores and 20 hardware threads running at 3.3 GHz. It has a theoretical peak memory bandwidth of 85.312 GB/s resulting from four 64-bit wide channels with a data rate of 2666MT/s. The sizes of the L3-cache and each L2-cache are 14MB and 1024KB. The source code has been compiled with GCC v7.3 using the highest optimization level `-Ofast` and `-march=native`, `-pthread` and `-fopenmp`. Parallel execution for the general case (8) has been accomplished using GCC's implementation of the OpenMP v4.5 specification. We have used the DOT and GEMV implementation of the OpenBLAS library v0.2.20. 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 provide a comprehensive test coverage. *Setup 1* performs runtime measurements with *asymmetrically-shaped* tensors. Their dimension tuples are organized in 10 two-dimensional arrays  $\mathbf{N}_q$  with 9 rows and 32 columns where the dimension tuple  $\mathbf{n}_{r,c}$  of length  $r+1$  denotes an element  $\mathbf{N}_q(r, c)$  of  $\mathbf{N}_q$  with  $1 \leq q \leq 10$ . The dimension  $\mathbf{n}_{r,c}(i)$  of  $\mathbf{N}_q$  is 1024 if  $i = 1$ ,  $c \cdot 2^{15-r}$  if  $i = \min(r+1, q)$  and 2 for any other index  $i$  with  $1 < q \leq 10$ . The dimension  $\mathbf{n}_{r,c}(i)$  of  $\mathbf{N}_1$  is given by  $c \cdot 2^{15-r}$  if  $i = 1$ , 1024 if  $i = 2$  and 2 for any other index  $i$ . Dimension tuples of the same array column have the same number of tensor elements. Please note that with increasing tensor order (and row-number), the contraction mode is halved and with increasing tensor size, the contraction mode is multiplied by the column number. Such a setup enables an orthogonal test-set in terms of tensor elements ranging from  $2^{25}$  to  $2^{29}$  and tensor order ranging from 2 to 10. *Setup 2* performs runtime measurements with *symmetrically-shaped* tensors. Their dimension tuples are organized in one two-dimensional array  $\mathbf{M}$  with 6 rows and 8 columns where the dimension tuple  $\mathbf{m}_{r,c}$  of length  $r+1$  denotes an element  $\mathbf{M}(r, c)$  of  $\mathbf{M}$ . For  $c = 1$ , the dimensions of  $\mathbf{m}_{r,c}$  are given by  $2^{12}$ ,  $2^8$ ,  $2^6$ ,  $2^5$ ,  $2^4$  and  $2^3$  with descending row number  $r$  from 6 to 1. For  $c > 1$ , the remaining dimensions are given by  $\mathbf{m}_{r,c} = \mathbf{m}_{r,c} + k \cdot (c-1)$  where  $k$  is  $2^9$ ,  $2^5$ ,  $2^3$ ,  $2^2$ ,  $2$ ,  $1$  with descending row number  $r$  from 6 to 1. In this setup, shape tuples of a column do not yield the same number of subtensor elements.



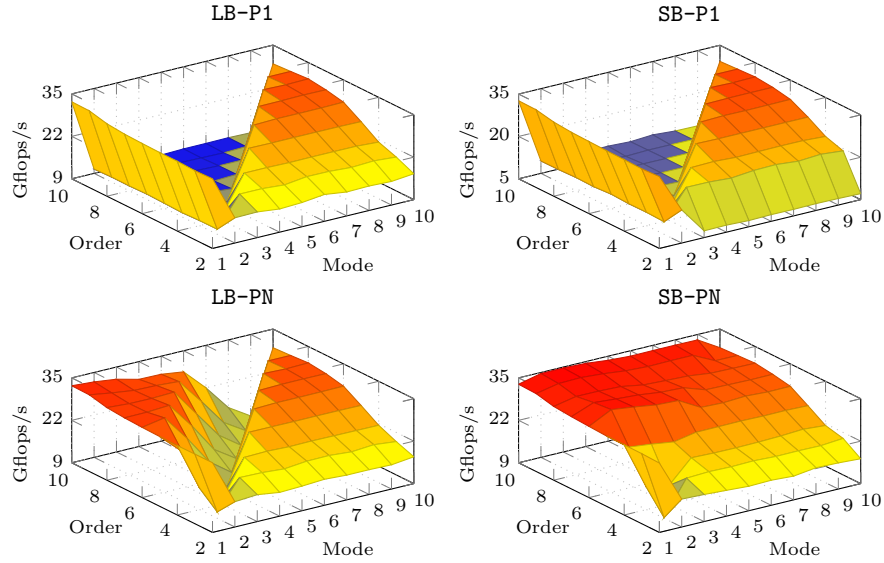
**Fig. 1.** Schematic contour view of the following average performance maps for the tensor-vector multiplication with tensors that are stored according to the first-order storage format. Each case  $x$  in Table 1 affects a different region  $x$  within the performance map. Performance values are the arithmetic mean over the set of tensor sizes with 32 and 8 elements in case of the first and second test setup, respectively. Contraction mode  $q = p$  for  $q > p$  where  $p$  is the tensor order.

**Performance Maps** Measuring a single tensor-vector multiplication with the first setup produces  $2880 = 9 \times 32 \times 10$  runtime data points where the tensor order ranges from 2 to 10, with 32 shapes for each order and 10 contraction modes. The second setup produces  $336 = 6 \times 8 \times 7$  data points with 6 tensor orders ranging from 2 to 7, 8 shapes for each order and 7 contraction modes. Similar to the findings in [4], we have observed a performance loss for small dimensions of the mode with the highest priority. The presented performance values are the arithmetic mean over the set of tensor sizes that vary with the tensor order and contraction mode resulting in a three dimensional performance plot. A schematic countour view of the plots is given in Fig. 1 which is divided into 5 regions. The cases 2, 3, 6 and 7 generate performance values within the regions 2, 3, 6 and 7 where only a single parallel **GEMV** is executed, see Table 1. Please note that the contraction mode  $q$  is set to the tensor order  $p$  if  $q > p$ . Performance values within region 8 result from case 8 which executes **GEMV**'s with tensor slices in parallel.

The following analysis considers four parallel versions **SB-P1**, **LB-P1**, **SB-PN** and **LB-PN**. **SB** (small-block) and **LB** (large-block) denote parallel slice-vector multiplications where each thread recursively calls a single-threaded **GEMV** with mode-2 and mode- $\hat{q}$  slices, respectively. **P1** uses the outer-most dimension  $n_p$  for parallel execution whereas **PN** applies loop fusion and considers all fusible dimensions for parallel execution.

## 6 Results and Discussion

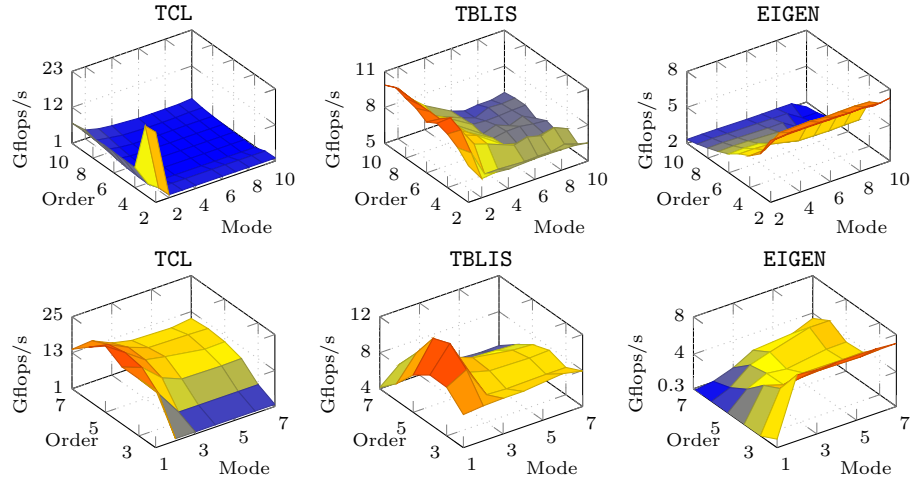
**Matrix-Vector Multiplication** Fig. 2 shows average performance values of the four versions **SB-P1**, **LB-P1**, **SB-PN** and **LB-PN** with asymmetrically-shaped tensors. In case 2 (region 2), the shape tuple of the two-order tensor is equal to  $(n_2, n_1)$  where  $n_2$  is set to 1024 and  $n_1$  is  $c \cdot 2^{14}$  for  $1 \leq c \leq 32$ . In case 6 (region



**Fig. 2.** 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.

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  $p$ . 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  $n_{q+1}, \dots, n_p$  are executed in parallel. The PN version speeds up

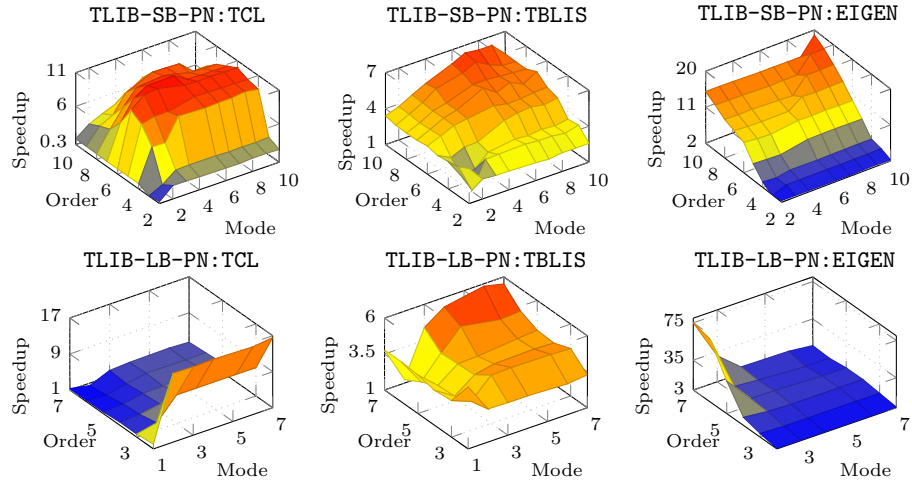


**Fig. 3.** Average performance maps of tensor-vector multiplication implementations using *asymmetrically-shaped* (top) and *symmetrically-shaped* (bottom) tensors with varying contraction modes and tensor order. Tensor elements are encoded in single-precision and stored contiguously in memory according to the first-order storage format.

the 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  $2, 3, \dots, p - 2$  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 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  $\pi_1, \dots, \pi_{k-1}$  and  $\pi_{k+1}, \dots, \pi_p$  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.



**Fig. 4.** 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.

**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. 3 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 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. 4 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 algorithm 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., Battaglini, 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., Niranjana, 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)