

# 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 basic tensor operation that is required by various tensor methods such as the ALS or the HOSVD. This paper presents flexible high-performance algorithms for the mode- $q$  tensor-matrix multiplication that computes the product according to the the Loops-over-**gemms** (LOG) approach. Our algorithms can process dense tensors with any linear tensor layout, arbitrary tensor order and dimensions all of which can be runtime variable. We discuss different tensor slicing methods with parallelization strategies and propose six variations of a base algorithm which calls a **gemv**, **gemms** and/or a **gemm\_batch** with subtensors or tensor slices. Their performance is quantified for a large tensor set that contains tensors with various shapes. The best-performing version attains a median performance of 1.37 double 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.

## 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 almost reach the practical peak performance of a computing machine.

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 micro-architecture.

Our work is motivated by the fact that LOG-based implementations of the tensor-matrix and tensor-vector multiplication are similar. To our best knowledge, we are the first to combine the approach in [3] with the findings in [10] and to propose fast in-place tensor-matrix multiplication algorithms that are layout-oblivious. Our algorithms compute the tensor-matrix product in parallel using OpenMP together with highly efficient `gemm` or `gemm_batch` implementations. 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. The parallel versions of the recursive base algorithm execute fused loops in parallel and are able to fully utilize a processors compute units. Despite, the generality of our approach, every proposed algorithm can be 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. We have provided an open and free reference C++ implementation of all algorithms and a python interface for convenience. While we have used Intel’s MKL for our benchmarks, the user is free to choose any other library that provides the BLAS interface.

The following analysis quantifies the impact of the tensor layout, the tensor slicing method and parallel execution of slice-matrix multiplications with varying contraction modes. The runtime measurements of our implementations are compared with those presented in [12, 17] including Libtorch and Eigen. 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 for any linear tensor layout.
- Algorithms with variable loop fusion and parallel subtensor-matrix multiplications achieves the peak performance of a `gemm` with large slice dimensions. Moreover, all our proposed algorithms are layout oblivious and achieve at least a median throughput of [?] 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 execu-

tion 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. 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  $\mathbf{n} = (n_1, n_2, \dots, n_p)$  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

125  $1 \leq u \neq v \neq w \leq p$ . It is sufficient to only provide non-unit modes as the  
 126 remaining single indices correspond to the loop induction variables of the fol-  
 127 lowing algorithms. A subtensor is called a slice  $\underline{\mathbf{A}}'_{u,v}$  if the full range selection  
 128 of  $\underline{\mathbf{A}}$  occurs with only two modes. A fiber  $\underline{\mathbf{A}}'_u$  is a tensor slice with only one  
 129 dimension greater than 1.

130 **Linear Tensor Layouts** We use a layout tuple  $\boldsymbol{\pi} \in \mathbb{N}^p$  to encode all lin-  
 131 ear tensor layouts including the first-order or last-order layout. They contain  
 132 permuted tensor modes whose priority is given by their index. For instance,  
 133 the first- and last-order storage formats are given by  $\boldsymbol{\pi}_F = (1, 2, \dots, p)$  and  
 134  $\boldsymbol{\pi}_L = (p, p-1, \dots, 1)$ . An inverse layout tuple  $\boldsymbol{\pi}^{-1}$  is defined by  $\boldsymbol{\pi}^{-1}(\boldsymbol{\pi}(k)) = k$ .  
 135 Given a layout tuple  $\boldsymbol{\pi}$  with  $p$  modes, the  $\pi_r$ -th element of a stride tuple is given  
 136 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  
 137 mode are contiguously stored in memory.

138 The location of tensor elements within the allocated memory space is deter-  
 139 mined by the tensor layout and the corresponding layout function. For a given  
 140 layout and stride tuple, a layout function  $\lambda_{\mathbf{w}}$  maps a multi-index to a scalar  
 141 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  
 142 position of an element with a multi-index  $\mathbf{i}$ , reading from and writing to memory  
 143 is accomplished with  $j$  and the first element's address of  $\underline{\mathbf{A}}$ .

144 **Non-Modifying Flattening and Reshaping** The flattening operation  $\varphi_{r,q}$   
 145 transforms an order- $p$  tensor  $\underline{\mathbf{A}}$  to another order- $p'$  view  $\underline{\mathbf{B}}$  that has different  
 146 a shape  $\mathbf{m}$  and layout  $\boldsymbol{\tau}$  tuple of length  $p'$  with  $p' = p - q + r$  and  $1 \leq r <$   
 147  $q \leq p$ . It is related to the tensor unfolding operation as defined in [8, p.459]  
 148 but neither changes the element ordering nor copies tensor elements. Given  
 149 a layout tuple  $\boldsymbol{\pi}$  of  $\underline{\mathbf{A}}$ , the flattening operation  $\varphi_{r,q}$  is defined for contigu-  
 150 ous modes  $\hat{\boldsymbol{\pi}} = (\pi_r, \pi_{r+1}, \dots, \pi_q)$  of  $\boldsymbol{\pi}$ . Let  $j = 0$  if  $k \leq r$  and  $j = q - r$   
 151 otherwise for  $1 \leq k \leq p'$ . Then the resulting layout tuple  $\boldsymbol{\tau} = (\tau_1, \dots, \tau_{p'})$   
 152 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  $s_k =$   
 153  $|\{\pi_i \mid \pi_{k+j} > \pi_i \wedge \pi_i \neq \min(\hat{\boldsymbol{\pi}}) \wedge r \leq i \leq p\}|$ . Elements of the corresponding shape  
 154 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  $k \neq r$ .

155 The reshaping operation  $\rho$  transforms an order- $p$  tensor  $\underline{\mathbf{A}}$  to another order- $p$   
 156 tensor  $\underline{\mathbf{B}}$  with different shape  $\mathbf{m}$  and layout  $\boldsymbol{\tau}$  tuples of length  $p$ . In this work,  
 157 it permutes the shape and layout tuple simultaneously without changing the  
 158 element ordering and without copying tensor elements. The operation  $\rho$  uses  
 159 a permutation tuple  $\boldsymbol{\rho} = (\rho_1, \dots, \rho_p)$  to only modify shape and layout tuples.  
 160 Elements of the resulting shape tuple  $\mathbf{m}$  and the layout tuple  $\boldsymbol{\tau}$  are given by  
 161  $m_r = n_{\rho_r}$  and  $\tau_r = \pi_{\rho_r}$ , respectively.

162 **Tensor-Matrix Multiplication (TTM)** Let  $\underline{\mathbf{A}}$  and  $\underline{\mathbf{C}}$  be order- $p$  tensors with  
 163 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  
 164  $\mathbf{B}$  be a matrix of shape  $\mathbf{n}_b = (m, n_q)$ . A mode- $q$  TTM is denoted by  $\underline{\mathbf{C}} = \underline{\mathbf{A}} \times_q \mathbf{B}$

165 where an element of  $\underline{\mathbf{C}}$  is given by

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

166 with  $1 \leq i_r \leq n_r$  and  $1 \leq j \leq m$ . The mode  $q$  is the *contraction mode* of  
 167 the TTM with  $1 \leq q \leq p$ . The tensor-matrix multiplication generalizes the  
 168 computational aspect of the two-dimensional case  $\mathbf{C} = \mathbf{B} \cdot \mathbf{A}$  if  $p = 2$  and  $q = 1$ .  
 169 Its arithmetic intensity is equal to that of a matrix-matrix multiplication and  
 170 is not memory-bound. In the following, we assume that the tensors  $\underline{\mathbf{A}}$  and  $\underline{\mathbf{C}}$   
 171 have the same tensor layout  $\boldsymbol{\pi}$ . Elements of matrix  $\mathbf{B}$  can stored in either the  
 172 column-major or row-major format.

## 173 4 Algorithm Design

### 174 4.1 Sequential Algorithm

175 The sequential baseline algorithm for Eq. 1 can be implemented with a single  
 176 C++ function that supports tensors with arbitrary order, dimensions and any  
 177 linear tensor layout. It consists of nested recursion with a control flow that is akin  
 178 to algorithm 1 in [4] consisting of two **if** statements with an **else** branch. The  
 179 body of the first **if** statement contains a recursive call that skips the iteration  
 180 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  
 181 inverse layout tuple. The second **if** statement contains multiple recursive calls  
 182 for the modes  $1 \leq r \neq \hat{q} \leq p$  with different multi-indices. The **else** branch is the  
 183 base case and consists of two loops that compute a fiber-matrix product. The  
 184 outer loop iterates with  $j$  over the dimension  $m$  of  $\underline{\mathbf{C}}$  and  $\mathbf{B}$ . The inner loop  
 185 iterates with  $i_q$  over the dimension  $n_q$  of  $\underline{\mathbf{A}}$  and  $\mathbf{B}$  computing an inner product.

### 186 4.2 Baseline Algorithm with Contiguous Memory Access

187 The baseline algorithm accesses elements of  $\underline{\mathbf{A}}$  and  $\underline{\mathbf{C}}$  non-contiguously whenever  
 188  $\pi_1 \neq q$ . Matrix  $\mathbf{B}$  is contiguously accessed if  $i_q$  or  $j$  is incremented with unit-  
 189 steps depending on the storage format of  $\mathbf{B}$ . The access pattern can be improved  
 190 by reordering tensor elements according to the storage format. However, copy  
 191 operations reduce the overall throughput of the operation [15].

192 A better approach is to access tensor elements according to the tensor layout  
 193 using the tensor layout tuple  $\boldsymbol{\pi}$  as proposed in [4]. The modified algorithm 1  
 194 contiguously accesses memory for  $\pi_1 \neq q$  and  $p > 1$ . Each recursion level adjusts  
 195 only one multi-index element  $i_{\pi_r}$  with a stride  $w_{\pi_r}$  in line 5. With increasing re-  
 196 cursion level and decreasing  $r$ , indices are incremented with smaller step sizes as  
 197  $w_{\pi_r} \leq w_{\pi_{r+1}}$ . The condition of the second **if** statement in line 4 is changed from  
 198  $r \geq 1$  to  $r > 1$ . In this way, the mode- $\pi_1$  loop with index  $i_{\pi_1}$  and the minimum  
 199 stride  $w_{\pi_1}$  are included in the base case which contains three loops performing  
 200 a slice-matrix multiplication. The loop ordering are adjusted according to the

---

```

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  $\underline{\mathbf{C}}$ .

201 tensor and matrix layout. The inner-most loop increments  $i_{\pi_1}$  and contiguously  
 202 accesses tensor elements of  $\underline{\mathbf{A}}$  and  $\underline{\mathbf{C}}$ . The second loop increments  $i_q$  with which  
 203 elements of  $\mathbf{B}$  are contiguously accessed if  $\mathbf{B}$  is stored in the row-major format.  
 204 The third loop increments  $j$  and could be placed as the second loop if  $\mathbf{B}$  is stored  
 205 in the column-major format.

206 While spatial data locality is improved by adjusting the loop ordering, the  
 207 temporal data locality of tensors  $\underline{\mathbf{A}}$  and  $\underline{\mathbf{C}}$  differ. Note that slice  $\underline{\mathbf{A}}'_{\pi_1, q}$  is accessed  
 208  $m$  times, fiber  $\underline{\mathbf{C}}_{\pi_1}$  is accessed  $\mathbf{n}(q)$  times and element  $\underline{\mathbf{B}}(j, i_q)$  is accessed  $\mathbf{n}(\pi_1)$   
 209 times. While the specified fiber of  $\underline{\mathbf{C}}$  can fit into first or second level cache, slice  
 210 elements of  $\underline{\mathbf{A}}$  are unlikely to fit in the local caches if the slice size  $n_{\pi_1} \times n_q$   
 211 is large leading to higher cache misses and suboptimal performance. Optimized  
 212 tiling for better temporal data locality has been discussed in [6] which suggests  
 213 to use existing high-performance BLAS implementations for the base case.

### 214 4.3 BLAS-based Algorithms with Tensor Slices

215 Algorithm 1 is the starting point for the BLAS-based algorithm which computes  
 216 the tensor-matrix product with a `gemm` routine. Besides the illustrated algorithm,  
 217 we have identified seven other cases where a single `gemm` call suffices to compute  
 218 the tensor-matrix product even if the tensor order  $p > 2$ . In summary, there  
 219 are eight cases with a single `gemm` call using different arguments which are listed  
 220 in table 1. The list of `gemm` calls supports all linear tensor layout and has no  
 221 limitation on tensor order and contraction mode. The arguments of `gemm` are  
 222 chosen depending on the tensor order  $p$ , tensor layout  $\boldsymbol{\pi}$  and contraction mode  
 223  $q$  except for the `CBLAS_ORDER` which is `CblasRowMajor`.

Case	Order $p$	Layout $\pi$	Mode $q$	Routine	T	M	N	K	A	LDA	B	LDB	LDC
1	1	-	1	<code>gemv</code>	-	$m$	$n_1$	-	$\mathbf{B}$	$n_1$	$\underline{\mathbf{A}}$	-	-
2	2	(1, 2)	1	<code>gemm</code>	$\mathbf{B}$	$n_2$	$m$	$n_1$	$\underline{\mathbf{A}}$	$n_1$	$\mathbf{B}$	$n_1$	$m$
3	2	(1, 2)	2	<code>gemm</code>	-	$m$	$n_1$	$n_2$	$\mathbf{B}$	$n_2$	$\underline{\mathbf{A}}$	$n_1$	$n_1$
4	2	(2, 1)	1	<code>gemm</code>	-	$m$	$n_2$	$n_1$	$\mathbf{B}$	$n_1$	$\underline{\mathbf{A}}$	$n_2$	$n_2$
5	2	(2, 1)	2	<code>gemm</code>	$\mathbf{B}$	$n_1$	$m$	$n_2$	$\underline{\mathbf{A}}$	$n_2$	$\mathbf{B}$	$n_2$	$m$
6	$> 2$	any	$\pi_1$	<code>gemm</code>	$\mathbf{B}$	$\bar{n}_q$	$m$	$n_q$	$\underline{\mathbf{A}}$	$n_q$	$\mathbf{B}$	$n_q$	$m$
7	$> 2$	any	$\pi_p$	<code>gemm</code>	-	$m$	$\bar{n}_q$	$n_q$	$\mathbf{B}$	$n_q$	$\underline{\mathbf{A}}$	$\bar{n}_q$	$\bar{n}_q$
8	$> 2$	any	$\pi_2, \dots, \pi_{p-1}$	<code>gemm*</code>	-	$m$	$n_{\pi_1}$	$n_q$	$\mathbf{B}$	$n_q$	$\underline{\mathbf{A}}$	$w_q$	$w_q$

**Table 1.** Parameter configuration of the `gemv`- and `gemm` routines with eight cases computing a tensor-matrix product in which  $\mathbf{B}$  has the row-major format. The BLAS arguments T, M, N, etc. are chosen with respect to the tensor order  $p$ , tensor layout  $\pi$  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  $\bar{n}_q = (n_1 \cdots n_p)/n_q$ .

224 *Case 1* ( $p = 1$ ): The tensor-vector product  $\underline{\mathbf{A}} \times_1 \mathbf{B}$  can be computed with a  
 225 `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  $n_1$ .

226 *Case 2-5* ( $p = 2$ ): If  $\underline{\mathbf{A}}$  and  $\underline{\mathbf{C}}$  are order-2 tensors, i.e. a matrix  $\mathbf{A}$  with  
 227 dimensions  $n_1$  and  $n_2$ , then a single `gemm` suffices to compute the tensor-matrix  
 228 product. If  $\mathbf{A}$  and  $\mathbf{C}$  have the column-major format with  $\pi = (1, 2)$ , `gemm` either  
 229 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  
 230  $\mathbf{C}$  and  $\mathbf{A}$  as matrices using the reshaping operation  $\rho$  with  $\rho = (2, 1)$  in row-  
 231 major format even though both are stored column-wise. If  $\mathbf{A}$  and  $\mathbf{C}$  have the  
 232 row-major format with  $\pi = (2, 1)$ , `gemm` either executes  $\mathbf{C} = \mathbf{B} \cdot \mathbf{A}$  for  $q = 1$  or  
 233  $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}^T$  for  $q = 2$ . Note that the transposition of  $\mathbf{B}$  is necessary for the cases  
 234 2,5 and independent of the chosen storage format.

235 *Case 6-7* ( $p > 2$ ): If the order of  $\underline{\mathbf{A}}$  and  $\underline{\mathbf{C}}$  is greater than 2 and if the  
 236 contraction mode  $q$  is equal to  $\pi_1$  (case 6), a single `gemm` with the depicted  
 237 parameters executes  $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}^T$  and computes a tensor-matrix product  $\underline{\mathbf{C}} =$   
 238  $\underline{\mathbf{A}} \times_{\pi_1} \mathbf{B}$  for any storage layout of  $\underline{\mathbf{A}}$  and  $\underline{\mathbf{C}}$ . Tensors  $\underline{\mathbf{A}}$  and  $\underline{\mathbf{C}}$  are flattened with  
 239  $\varphi_{2,p}$  to row-major matrices  $\mathbf{A}$  and  $\mathbf{C}$ . Matrix  $\mathbf{A}$  has  $\bar{n}_{\pi_1} = \bar{n}/n_{\pi_1}$  rows and  $n_{\pi_1}$   
 240 columns while matrix  $\mathbf{C}$  has the same number of rows and  $m$  columns. If  $\pi_p = q$   
 241 (case 7), Tensors  $\underline{\mathbf{A}}$  and  $\underline{\mathbf{C}}$  are flattened with  $\varphi_{1,p-1}$  to column-major matrices  
 242  $\mathbf{A}$  and  $\mathbf{C}$ . Matrix  $\mathbf{A}$  has  $n_{\pi_p}$  rows and  $\bar{n}_{\pi_p} = \bar{n}/n_{\pi_p}$  columns while matrix  $\mathbf{C}$   
 243 has  $m$  rows and the same number of columns. A single `gemm` executes  $\mathbf{C} = \mathbf{B} \cdot \mathbf{A}$   
 244 and computes the tensor-matrix product  $\underline{\mathbf{C}} = \underline{\mathbf{A}} \times_{\pi_p} \mathbf{B}$  for any storage layout  
 245 of  $\underline{\mathbf{A}}$  and  $\underline{\mathbf{C}}$ . Note that in all cases no copy operation is performed in order to  
 246 compute the desired contraction, see subsection 3.

247 *Case 8* ( $p > 2$ ): If the tensor order is greater than 2 with  $\pi_1 \neq q$  and  $\pi_p \neq q$ ,  
 248 the modified baseline algorithm 1 is used to successively call  $\bar{n}/(n_q \cdot n_{\pi_1})$  times  
 249 `gemm` with different tensor slices of  $\underline{\mathbf{C}}$  and  $\underline{\mathbf{A}}$  in the base case. Each `gemm` computes  
 250 one slice  $\underline{\mathbf{C}}'_{\pi_1,q}$  of the tensor-matrix product  $\underline{\mathbf{C}}$  using the corresponding tensor

251 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  
 252 by interpreting both tensor slices as row-major matrices  $\mathbf{A}$  and  $\mathbf{C}$  which have  
 253 the dimensions  $(n_q, n_{\pi_1})$  and  $(m, n_{\pi_1})$ , respectively.

#### 254 4.4 BLAS-Based Algorithms with Subtensors

255 Case 8 can be optimized by selecting larger subtensors instead of tensor slices  
 256 which might lead to a better processor utilization. Larger subtensors can be  
 257 used by selecting additional mergeable modes that still allow the subtensor to be  
 258 flattened into a matrix without reordering tensor elements using the description  
 259 provided in section 3, see also lemma 4.1 in [10]. The maximum number of  
 260 mergeable modes is  $\hat{q} - 1$  with  $\hat{q} = \pi^{-1}(q)$  and the corresponding modes are  
 261  $\pi_1, \pi_2, \dots, \pi_{\hat{q}-1}$ . Applying flattening  $\varphi_{1, q-1}$  and reshaping  $\rho$  with  $\boldsymbol{\rho} = (2, 1)$  on  
 262 a subtensor of  $\underline{\mathbf{A}}$  with dimensions  $n_{\pi_1}, \dots, n_{\pi_{\hat{q}-1}}, n_q$  yields a row-major matrix  
 263  $\mathbf{A}$  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-  
 264 major matrix with shape  $(m, \prod_{r=1}^{\hat{q}-1} n_{\pi_r})$ . This description supports all linear  
 265 tensor layouts and generalizes lemma 4.2 in [10].

266 Algorithm 1 needs a minor modification for calling `gemm` flattened subtensors  
 267 instead of tensor slices. The modified algorithm must therefor omit the first  
 268  $\hat{q}$  modes  $\pi_{1, \hat{q}} = (\pi_1, \dots, \pi_{\hat{q}})$  including  $\pi_{\hat{q}} = q$ . This is done by only iterating  
 269 over modes larger than  $\hat{q}$  in the non-base case. The conditions in line 2 and 4 are  
 270 changed to  $1 < r \leq \hat{q}$  and  $\hat{q} < r$ , respectively. The single indices of the subtensors  
 271  $\underline{\mathbf{A}}'_{\pi_{1, \hat{q}}}$  and  $\underline{\mathbf{C}}'_{\pi_{1, \hat{q}}}$  are given by the loop induction variables that belong to the  
 272  $\pi_r$ -th loop with  $\hat{q} + 1 \leq r \leq p$ .

#### 273 4.5 Parallel BLAS-based Algorithms

274 Note that cases 1 to 7 only call a single `gemm` and cannot be further parallelized.  
 275 Hence, the following paragraphs discuss parallel methods for the eighth case.

276 **Sequential Loops and Multithreaded `gemm`** One straight forward approach  
 277 is to use algorithm 1 as it is and to sequentially call a multi-threaded `gemm` in  
 278 the base case of the algorithm as described in subsection 4.4. This is beneficial if  
 279  $q = \pi_{p-1}$ , the inner dimensions  $n_{\pi_1}, \dots, n_q$  are large or the outer-most dimension  
 280  $n_{\pi_p}$  is smaller than the available processor cores. However, if the above conditions  
 281 are not met, the processor cores might not be fully utilized where each multi-  
 282 threaded `gemm` is executed with small subtensors. We will refer to this algorithm  
 283 version as `<seq-loops, par-gemm>` that is executable with subtensors or tensor  
 284 slices.

285 **Parallel Loops and Single- or Multithreaded `gemm`** A more advanced ver-  
 286 sion of the above algorithm executes a single-threaded `gemm` in parallel including  
 287 all available (free) modes which depend on the slicing. If subtensors are used,  
 288 all  $\pi_{\hat{q}+1}, \dots, \pi_p$  modes are free. In case of tensor slices, only  $\pi_1$  and  $\pi_{\hat{q}}$  are free  
 289 modes. The corresponding maximum degree of parallelism for both cases are  
 290  $\prod_{r=\hat{q}+1}^p n_{\pi_r}$  and  $\prod_{r=1}^p n_r / (n_{\pi_1} n_{\pi_{\hat{q}}})$ , respectively.



Using tensor slices for the multiplication,  $\underline{\mathbf{A}}$  and  $\underline{\mathbf{C}}$  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  $\hat{n}_{\pi_4} = \prod_{r=\hat{q}+1}^p n_{\pi_r}$ . In this way the tree-recursion has been transformed in two 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  $\underline{\mathbf{C}}'_{\pi_1, q}$ . Both loops are parallelized using `omp parallel for` together with the `collapse(2)` and the `num_threads` clause which specifies the thread number.

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 single-threaded `gemm` with multiple subtensors  $\underline{\mathbf{A}}'_{\pi', q}$  and  $\underline{\mathbf{C}}'_{\pi', q}$  with  $\pi' = (\pi_1, \dots, \pi_{\hat{q}-1})$ .

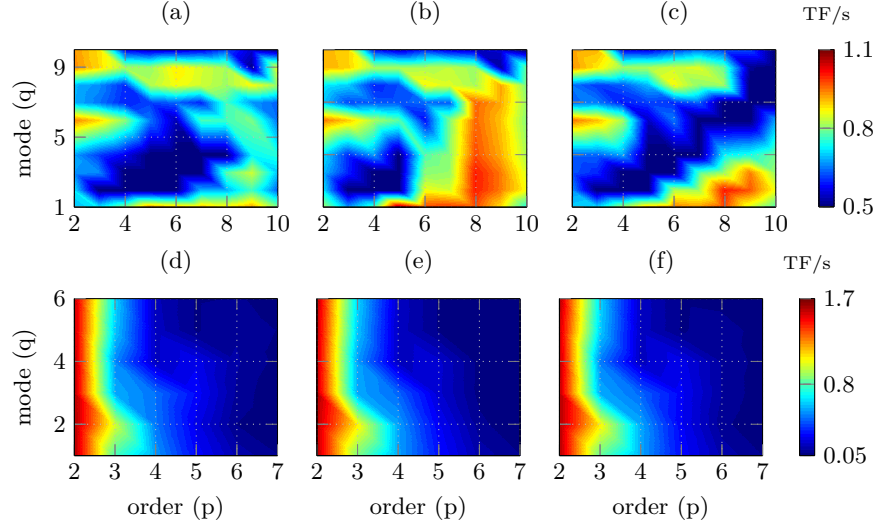
Both algorithm variants will be referred to as `<par-loops, seq-gemm>` which can be used with subtensors or tensor slices. Note that `<seq-loops, par-gemm>` and `<par-loops, seq-gemm>` are opposing versions where either `gemm` or the free loops are performed in parallel. The all-parallel version `<par-loops, par-gemm>` executes available loops in parallel where each loop thread executes a multi-threaded `gemm` with either subtensors or tensor slices.

**Multithreaded `gemm_batch`** The next version of the base algorithm is a modified version of the general subtensor-matrix approach that calls a single batched `gemm` for the eighth case. The subtensor dimensions and remaining `gemm` arguments remain the same. The library implementation is responsible how subtensor-matrix multiplications are executed and if subtensors are further divided into smaller subtensors or tensor slices. This version will be referred to as the `<gemm_batch>` variant.

## 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 `-O3` 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 MKL and its own threading library `mk1_intel_thread` together with the threading runtime library `libiomp5`. The benchmark results of each function are the average of 10 runs.



**Fig. 1.** Performance maps in double-precision Tflops/s of the proposed tensor-times-matrix algorithms with varying tensor orders  $p$  and contraction modes  $q$ . Tensors are asymmetrically-shaped on the upper plots and symmetrically-shaped on the lower plots. The algorithm of (a) and (e) executes `gemm_batch` with subtensors, (b) and (e) parallel loops over single-threaded `gemm` with tensor slices, (c) and (f) parallel loop over single-threaded `gemm` with subtensors.

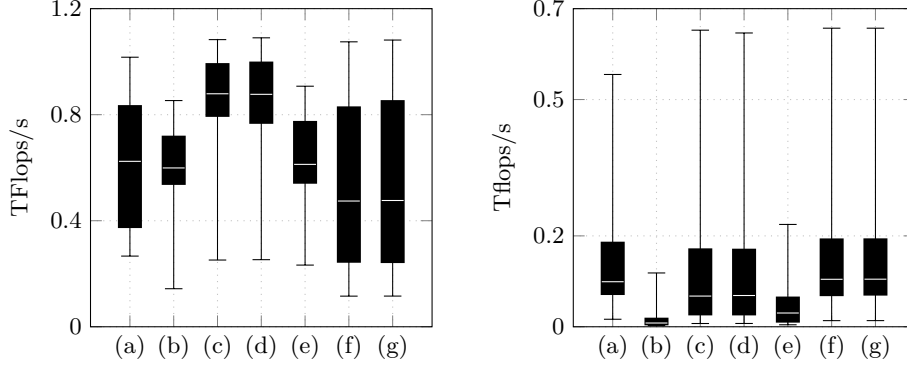
**Tensor Shapes** We have used asymmetrically-shaped and symmetrically-shaped tensors in order to cover many possible use cases. The dimension tuples of both shape types are organized within two three-dimensional arrays with which tensors are initialized. The dimension array for the first shape type 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

**Slicing and Parallelism** Fig. 1 shows mean performance values of the `<omp-par,seq-gemm>` version with subtensors or tensor slices and the `<gemm-batch>` version with subtensors.

the four versions SB-P1, LB-P1, SB-PN and LB-PN with asymmetrically-shaped tensors.

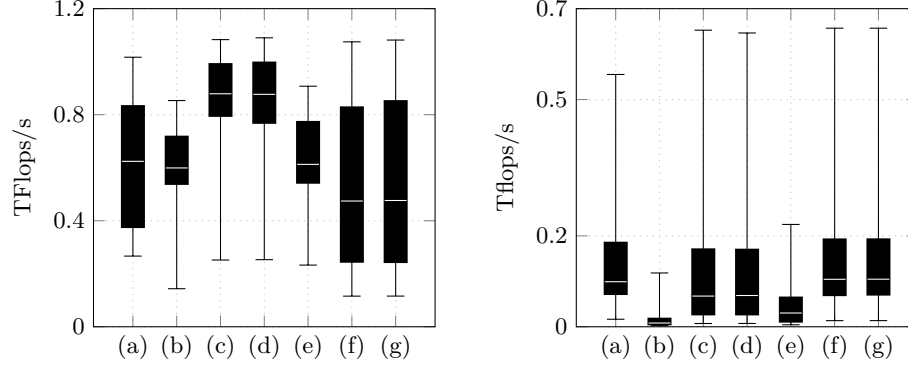
Fig. ?? shows average performance values of the four versions SB-P1, LB-P1, SB-PN and LB-PN with asymmetrically-shaped tensors. In case 2 (region 2), the



**Fig. 2.** Box plots visualizing performance statics in double-precision Tflops/s of the proposed tensor-times-matrix algorithms for the 8-th case. Tensors are asymmetrically-shaped on the left plot and symmetrically-shaped on the right plot. The Algorithm of (a) executes `gemm_batch` with subtensors, (b) and (e) sequential loops over multi-threaded `gemm`, (c) and (f) parallel loops over single-threaded `gemm`, (d) and (g) parallel loops over multi-threaded `gemm`. The algorithms (b,c,d) and (e,f,g) are executed with tensor slices or subtensors, respectively.

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  $p$ . 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 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  $n_p = 2$ .

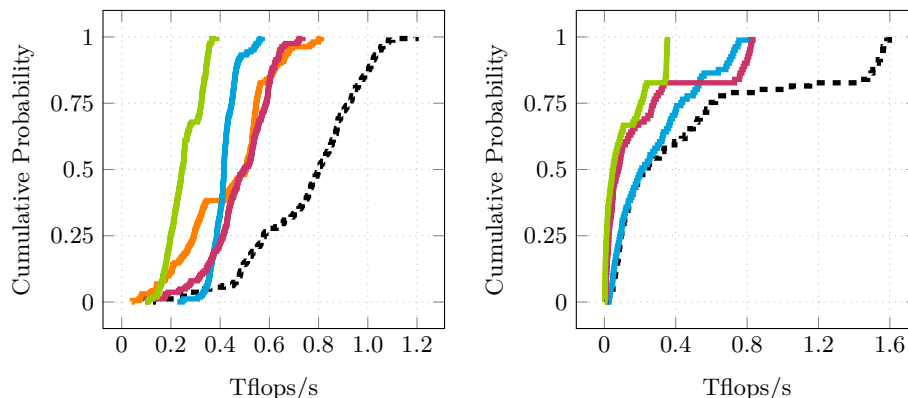


**Fig. 3.** Box plots visualizing performance statics in double-precision Tflops/s of a tensor-times-matrix algorithm for linear  $k$ -order tensor formats. The algorithm loops over single-threaded `gemm` with tensor slices with asymmetrically-shaped tensors on the left plot and with subtensors with symmetrically-shaped tensors on the right plot. Box plot number  $k$  denotes the utilized  $k$ -order storage.

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

382 **Tensor Layouts** Applying the first setup configuration with asymmetrically-  
 383 shaped tensors, we have analyzed the effects of the blocking and paralleliza-  
 384 tion strategy. The **LB-PN** version processes tensors with different storage formats,  
 385 namely the 1-, 2-, 9- and 10-order layout. The performance behavior is almost  
 386 the same for all storage formats except for the corner cases  $q = \pi_1$  and  $q = \pi_p$ .  
 387 Even the performance drop for  $q = p-1$  is almost unchanged. The standard  
 388 deviation from the mean value is less than 10% for all storage formats. Given a  
 389 contraction mode  $q = \pi_k$  with  $1 < k < p$ , a permutation of the inner and outer  
 390 tensor dimensions with their respective indices  $\pi_1, \dots, \pi_{k-1}$  and  $\pi_{k+1}, \dots, \pi_p$   
 391 does influence the runtime where the **LB-PN** version calls **GEMV** with the values  $w_m$   
 392 and  $n_m$ . The same holds true for the outer layout tuple.

393 **Comparison with other Approaches** The following comparison includes  
 394 three state-of-the-art libraries that implement three different approaches. The



**Fig. 4.** Cumulative performance distributions of tensor-times-matrix algorithms in double-precision Tflops/s. Each distribution line belongs to a library: **tlib** (---), **tcl** (—), **tblis** (—), **libtorch** (—), **eigen** (—). Libraries have been tested with asymmetrically-shaped (left plot) and symmetrically-shaped tensors (right plot).

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. ?? 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. ?? 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)

- 460 2. Bader, B.W., Kolda, T.G.: Algorithm 862: Matlab tensor classes for fast algorithm  
461 prototyping. *ACM Trans. Math. Softw.* **32**, 635–653 (December 2006)
- 462 3. Basso, C.: Design of a high-performance tensor-vector multiplication with blas.  
463 In: *International Conference on Computational Science*. pp. 32–45. Springer (2019)
- 464 4. Basso, C., Schatz, V.: Fast higher-order functions for tensor calculus with tensors  
465 and subtensors. In: *International Conference on Computational Science*. pp. 639–  
466 652. Springer (2018)
- 467 5. Golub, G.H., Van Loan, C.F.: *Matrix Computations*. JHU Press, 4 edn. (2013)
- 468 6. Goto, K., Geijn, R.A.v.d.: Anatomy of high-performance matrix multiplication.  
469 *ACM Transactions on Mathematical Software (TOMS)* **34**(3) (2008)
- 470 7. Karahan, E., Rojas-López, P.A., Bringas-Vega, M.L., Valdés-Hernández, P.A.,  
471 Valdes-Sosa, P.A.: Tensor analysis and fusion of multimodal brain images. *Pro-  
472 ceedings of the IEEE* **103**(9), 1531–1559 (2015)
- 473 8. Kolda, T.G., Bader, B.W.: Tensor decompositions and applications. *SIAM review*  
474 **51**(3), 455–500 (2009)
- 475 9. Lee, N., Cichocki, A.: Fundamental tensor operations for large-scale data analysis  
476 using tensor network formats. *Multidimensional Systems and Signal Processing*  
477 **29**(3), 921–960 (2018)
- 478 10. Li, J., Battaglini, C., Perros, I., Sun, J., Vuduc, R.: An input-adaptive and in-place  
479 approach to dense tensor-times-matrix multiply. In: *High Performance Computing,  
480 Networking, Storage and Analysis, 2015 SC-International Conference for*. pp. 1–12.  
481 IEEE (2015)
- 482 11. Lim, L.H.: Tensors and hypermatrices. In: Hogben, L. (ed.) *Handbook of Linear  
483 Algebra*. Chapman and Hall, 2 edn. (2017)
- 484 12. Matthews, D.A.: High-performance tensor contraction without transposition.  
485 *SIAM Journal on Scientific Computing* **40**(1), C1–C24 (2018)
- 486 13. Napoli, E.D., Fabregat-Traver, D., Quintana-Ortí, G., Bientinesi, P.: Towards an  
487 efficient use of the blas library for multilinear tensor contractions. *Applied Math-  
488 ematics and Computation* **235**, 454 – 468 (2014)
- 489 14. Papalexakis, E.E., Faloutsos, C., Sidiropoulos, N.D.: Tensors for data mining and  
490 data fusion: Models, applications, and scalable algorithms. *ACM Transactions on  
491 Intelligent Systems and Technology (TIST)* **8**(2), 16 (2017)
- 492 15. Shi, Y., Niranjana, U.N., Anandkumar, A., Cecka, C.: Tensor contractions with  
493 extended blas kernels on cpu and gpu. In: *2016 IEEE 23rd International Conference  
494 on High Performance Computing (HiPC)*. pp. 193–202 (Dec 2016)
- 495 16. Solomonik, E., Matthews, D., Hammond, J., Demmel, J.: Cyclops tensor frame-  
496 work: Reducing communication and eliminating load imbalance in massively par-  
497 allel contractions. In: *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th  
498 International Symposium on*. pp. 813–824. IEEE (2013)
- 499 17. Springer, P., Bientinesi, P.: Design of a high-performance gemm-like tensor-tensor  
500 multiplication. *ACM Transactions on Mathematical Software (TOMS)* **44**(3), 28  
501 (2018)