

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 multiplication 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 that compute the tensor-matrix product according to the Loops-over-GEMM (LoG) approach. Our algorithms can process dense tensors with any linear tensor layout, arbitrary tensor order and dimensions all of which can be runtime variable. We discuss different tensor slicing methods with parallelization strategies and propose six algorithm versions which call `gemv`, `gemm` and/or `gemm_batch` with subtensors or tensor slices. Their performance is quantified for a set of tensors with various shapes and tensor orders. Our best performing version attains a median performance of 1.37 double precision Tflops 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 for a range of linear tensor formats. Our fastest implementation is on average at least by 14.05% and up to 3.79 x faster than other state-of-the-art approaches and actively developed libraries like Libtorch and Eigen.

1 Introduction

Tensor computations are found in many scientific fields such as computational neuroscience, pattern recognition, signal processing and data mining [5, 12]. Tensors representing large amount of multidimensional data are decomposed and analyzed with the help of basic tensor operations [6, 7]. The decomposition and analysis led to the development and optimization 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 [6, 7]. 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 implementation (`gemm`) [1, 14]. Implementations of the GEMM-like Tensor-Tensor multiplication (GETT) methods have

macro-kernels that are similar to the ones used in fast GEMM implementations [10, 15]. A different method is the Loops-over-GEMM (LoG) approach in which BLAS are utilized with multiple tensor slices or subtensors if possible [2, 8, 11, 13]. Implementations of the LoG and TTGT approaches are in general easier to maintain and faster to port than GETT implementations which might need to adapt vector instructions or blocking parameters according to a processor’s microarchitecture.

To our best knowledge, we are the first to combine the LoG approach described in [2] with the findings on tensor slicing for the tensor-matrix multiplication in [8]. Our proposed algorithms support dense tensors with any order, dimensions and any linear tensor layout including the first- and the last-order storage formats for any contraction mode all of which can be runtime variable. They compute the tensor-matrix product in parallel using highly efficient `gemm` or `gemm_batch` without transposing or flattening tensors. Despite their high performance, all algorithms are layout-oblivious and provide a sustained performance independent of the tensor layout without tuning. Moreover, every proposed algorithm can be implemented with less than 100 lines of C++ code where the algorithmic complexity is reduced by the BLAS implementation and the corresponding selection of subtensors or tensor slices. We have provided an open 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 [10, 15] including Libtorch and Eigen. In summary, the main findings of our work are:

- A tensor-matrix multiplication can be implemented by an in-place algorithm with 1 `gemv` and 7 `gemm` calls, supporting all combinations of contraction mode, tensor order and dimensions for any linear tensor layout.
- Our algorithm with variable loop fusion and parallel slice-matrix multiplications is on average 17% faster than Intel’s `gemm_batch` when the contraction and leading dimensions of the tensors are greater than 256.
- Our algorithms are layout oblivious. The fastest implementations achieve at least a median throughput of [?] for any linear tensor layout.
- Our fastest algorithm computes the tensor-matrix multiplication on average, by at least 14.05% and up to a factor of 3.79 faster than other state-of-the-art library implementations, including LibTorch and Eigen.

The remainder of the paper is organized as follows. Section 2 presents related work. Section 3 introduces some notation on tensors and defines the tensor-matrix multiplication. Algorithm design and methods for slicing and parallel execution are discussed in Section 4. Section 5 describes the test setup. Benchmark results are presented in Section 6. Conclusions are drawn in Section 7.

2 Related Work

The authors in [11] 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 [8] 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 [1]. 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 [15] 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 [10] 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 [15].

3 Background

Notation An order- p tensor is a p -dimensional array [9] 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 $1 \leq u \neq v \neq w \leq p$. 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 $\boldsymbol{\pi}_L = (p, p-1, \dots, 1)$. The general k -order tensor layout for an order- p tensor is given by the layout tuple $\boldsymbol{\pi}$ with $\pi_r = k - r + 1$ for $1 < r \leq k$ and r for $k < r \leq p$. An inverse layout tuple $\boldsymbol{\pi}^{-1}$ is defined by $\boldsymbol{\pi}^{-1}(\boldsymbol{\pi}(k)) = k$. Given a layout tuple $\boldsymbol{\pi}$ with p modes, the π_r -th element of a stride tuple is given by $w_{\pi_r} = \prod_{k=1}^{r-1} n_{\pi_k}$ for $1 < r \leq p$ and $w_{\pi_1} = 1$. Tensor elements of the π_1 -th mode are contiguously stored in memory. The location of tensor elements is determined by the tensor layout and the layout function. For a given tensor layout and stride tuple, a layout function $\lambda_{\mathbf{w}}$ maps a multi-index to a scalar index with $\lambda_{\mathbf{w}}(\mathbf{i}) = \sum_{r=1}^p w_r(i_r - 1)$. With $j = \lambda_{\mathbf{w}}(\mathbf{i})$ being the relative memory position of an element with a multi-index \mathbf{i} , reading from and writing to memory is accomplished with j and the first element's address of $\underline{\mathbf{A}}$.

Non-Modifying Flattening and Reshaping The flattening operation $\varphi_{r,q}$ transforms an order- p tensor $\underline{\mathbf{A}}$ to another order- p' view $\underline{\mathbf{B}}$ that has different a shape \mathbf{m} and layout $\boldsymbol{\tau}$ tuple of length p' with $p' = p - q + r$ and $1 \leq r < q \leq p$. It is related to the tensor unfolding operation as defined in [6, p.459] but neither changes the element ordering nor copies tensor elements. Given a layout tuple $\boldsymbol{\pi}$ of $\underline{\mathbf{A}}$, the flattening operation $\varphi_{r,q}$ is defined for contiguous modes $\hat{\boldsymbol{\pi}} = (\pi_r, \pi_{r+1}, \dots, \pi_q)$ of $\boldsymbol{\pi}$. Let $j = 0$ if $k \leq r$ and $j = q - r$ otherwise for $1 \leq k \leq p'$. Then the resulting layout tuple $\boldsymbol{\tau} = (\tau_1, \dots, \tau_{p'})$ of $\underline{\mathbf{B}}$ is given by $\tau_r = \min(\boldsymbol{\pi}_{r,q})$ and $\tau_k = \pi_{k+j} + s_k$ if $k \neq r$ where $s_k = |\{\pi_i \mid \pi_{k+j} > \pi_i \wedge \pi_i \neq \min(\hat{\boldsymbol{\pi}}) \wedge r \leq i \leq p\}|$. Elements of the corresponding shape tuple \mathbf{m} are given by $m_{\tau_r} = \prod_{k=r}^q n_{\pi_k}$ and $m_{\tau_k} = n_{\pi_{k+j}}$ if $k \neq r$.

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

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

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

with $1 \leq i_r \leq n_r$ and $1 \leq j \leq m$. Mode q is the *contraction mode* of the TTM with $1 \leq q \leq p$. The tensor-matrix multiplication generalizes the computational

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

172 4 Algorithm Design

173 4.1 Sequential Algorithm

174 The sequential baseline algorithm for Eq. 1 can be implemented with a single
 175 C++ function that supports tensors with arbitrary order, dimensions and any
 176 linear tensor layout. It consists of nested recursion with a control flow that is akin
 177 to algorithm 1 in [3] consisting of two **if** statements with an **else** branch. The
 178 body of the first **if** statement contains a recursive call that skips the iteration
 179 over the dimension n_q when $r = \hat{q}$ with $\pi_r = q$ and $\hat{q} = \pi_q^{-1}$ where π^{-1} is the
 180 inverse layout tuple. The second **if** statement contains multiple recursive calls
 181 for the modes $1 \leq r \neq \hat{q} \leq p$ with different multi-indices. The **else** branch is the
 182 base case and consists of two loops that compute a fiber-matrix product. The
 183 outer loop iterates with j over the dimension m of $\underline{\mathbf{C}}$ and $\underline{\mathbf{B}}$. The inner loop
 184 iterates with i_q over the dimension n_q of $\underline{\mathbf{A}}$ and $\underline{\mathbf{B}}$ computing an inner product.

185 4.2 Baseline Algorithm with Contiguous Memory Access

186 The baseline algorithm accesses elements of $\underline{\mathbf{A}}$ and $\underline{\mathbf{C}}$ non-contiguously whenever
 187 $\pi_1 \neq q$. Matrix $\underline{\mathbf{B}}$ is contiguously accessed if i_q or j is incremented with unit-
 188 steps depending on the storage format of $\underline{\mathbf{B}}$. The access pattern can be improved
 189 by reordering tensor elements according to the storage format. However, copy
 190 operations reduce the overall throughput of the operation [13].

191 A better approach is to access tensor elements according to the tensor layout
 192 using the tensor layout tuple $\boldsymbol{\pi}$ as proposed in [3]. The modified algorithm 1
 193 contiguously accesses memory for $\pi_1 \neq q$ and $p > 1$. Each recursion level adjusts
 194 only one multi-index element i_{π_r} with a stride w_{π_r} in line 5. With increasing re-
 195 cursion level and decreasing r , indices are incremented with smaller step sizes as
 196 $w_{\pi_r} \leq w_{\pi_{r+1}}$. The condition of the second **if** statement in line 4 is changed from
 197 $r \geq 1$ to $r > 1$. In this way, the mode- π_1 loop with index i_{π_1} and the minimum
 198 stride w_{π_1} are included in the base case which contains three loops performing
 199 a slice-matrix multiplication. The loop ordering are adjusted according to the
 200 tensor and matrix layout. The inner-most loop increments i_{π_1} and contiguously
 201 accesses tensor elements of $\underline{\mathbf{A}}$ and $\underline{\mathbf{C}}$. The second loop increments i_q with which
 202 elements of $\underline{\mathbf{B}}$ are contiguously accessed if $\underline{\mathbf{B}}$ is stored in the row-major format.
 203 The third loop increments j and could be placed as the second loop if $\underline{\mathbf{B}}$ is stored
 204 in the column-major format.

205 While spatial data locality is improved by adjusting the loop ordering, the
 206 temporal data locality of tensors $\underline{\mathbf{A}}$ and $\underline{\mathbf{C}}$ differ. Note that slice $\underline{\mathbf{A}}'_{\pi_1, q}$ is accessed

```

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

207 m times, fiber $\underline{\mathbf{C}}_{\pi_1}$ is accessed $\mathbf{n}(q)$ times and element $\underline{\mathbf{B}}(j, i_q)$ is accessed $\mathbf{n}(\pi_1)$
208 times. While the specified fiber of $\underline{\mathbf{C}}$ can fit into first or second level cache, slice
209 elements of $\underline{\mathbf{A}}$ are unlikely to fit in the local caches if the slice size $n_{\pi_1} \times n_q$
210 is large leading to higher cache misses and suboptimal performance. Optimized
211 tiling for better temporal data locality has been discussed in [4] which suggests
212 to use existing high-performance BLAS implementations for the base case.

213 4.3 BLAS-based Algorithms with Tensor Slices

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

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

225 *Case 2-5* ($p = 2$): If $\underline{\mathbf{A}}$ and $\underline{\mathbf{C}}$ are order-2 tensors, i.e. a matrix \mathbf{A} with
226 dimensions n_1 and n_2 , then a single `gemm` suffices to compute the tensor-matrix
227 product. If \mathbf{A} and \mathbf{C} have the column-major format with $\pi = (1, 2)$, `gemm` either
228 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
229 \mathbf{C} and \mathbf{A} as matrices using the reshaping operation ρ with $\rho = (2, 1)$ in row-
230 major format even though both are stored column-wise. If \mathbf{A} and \mathbf{C} have the

Case	Order p	Layout π	Mode q	Routine	T	M	N	K	A	LDA	B	LDB	LDC
1	1	-	1	gemv	-	m	n_1	-	B	n_1	<u>A</u>	-	-
2	2	(1, 2)	1	gemm	B	n_2	m	n_1	<u>A</u>	n_1	B	n_1	m
3	2	(1, 2)	2	gemm	-	m	n_1	n_2	B	n_2	<u>A</u>	n_1	n_1
4	2	(2, 1)	1	gemm	-	m	n_2	n_1	B	n_1	<u>A</u>	n_2	n_2
5	2	(2, 1)	2	gemm	B	n_1	m	n_2	<u>A</u>	n_2	B	n_2	m
6	> 2	any	π_1	gemm	B	\bar{n}_q	m	n_q	<u>A</u>	n_q	B	n_q	m
7	> 2	any	π_p	gemm	-	m	\bar{n}_q	n_q	B	n_q	<u>A</u>	\bar{n}_q	\bar{n}_q
8	> 2	any	π_2, \dots, π_{p-1}	gemm*	-	m	n_{π_1}	n_q	B	n_q	<u>A</u>	w_q	w_q

Table 1. Eight cases with **gemv** and **gemm** for the mode- q tensor-matrix multiplication. Arguments T, M, N, etc. of the BLAS are chosen with respect to the tensor order p , layout π and contraction mode q where T specifies if **B** is transposed. **gemm*** denotes multiple **gemm** calls with different tensor slices. Argument \bar{n}_q for case 6 and 7 is given by $\bar{n}_q = 1/n_q \prod_r^p n_r$. Matrix **B** has the row-major format.

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 **B** is necessary for the cases 2,5 and independent of the chosen storage format.

Case 6-7 ($p > 2$): If the order of **A** and **C** is greater than 2 and if the contraction mode q is equal to π_1 (case 6), a single **gemm** with the depicted parameters executes $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}^T$ and computes a tensor-matrix product **C** = **A** \times_{π_1} **B** for any storage layout of **A** and **C**. Tensors **A** and **C** are flattened with $\varphi_{2,p}$ to row-major matrices **A** and **C**. Matrix **A** has $\bar{n}_{\pi_1} = \bar{n}/n_{\pi_1}$ rows and n_{π_1} columns while matrix **C** has the same number of rows and m columns. If $\pi_p = q$ (case 7), Tensors **A** and **C** are flattened with $\varphi_{1,p-1}$ to column-major matrices **A** and **C**. Matrix **A** has n_{π_p} rows and $\bar{n}_{\pi_p} = \bar{n}/n_{\pi_p}$ columns while matrix **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 **C** = **A** \times_{π_p} **B** for any storage layout of **A** and **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 **C** and **A** in the base case. Each **gemm** computes one slice **C**' $_{\pi_1,q}$ of the tensor-matrix product **C** using the corresponding tensor slices **A**' $_{\pi_1,q}$ and the matrix **B**. The matrix-matrix product $\mathbf{C} = \mathbf{B} \cdot \mathbf{A}$ is performed by interpreting both tensor slices as row-major matrices **A** and **C** which have the dimensions (n_q, n_{π_1}) and (m, n_{π_1}) , respectively.

4.4 BLAS-Based Algorithms with Subtensors

Case 8 can be optimized by utilizing larger subtensors instead of tensor slices. This can be done by adding mergeable modes to the slice-matrix multiplication in which the subtensor can be flattened into a matrix without reordering ten-

257 sor elements. The flattening operation does not copy or reorder elements, see
 258 section 3 and lemma 4.1 in [8]. The number of mergeable modes is $\hat{q} - 1$ with
 259 $\hat{q} = \pi^{-1}(q)$ and the corresponding modes are $\pi_1, \pi_2, \dots, \pi_{\hat{q}-1}$. Applying flatten-
 260 ing $\varphi_{1,q-1}$ and reshaping ρ with $\rho = (2, 1)$ on a subtensor of $\underline{\mathbf{A}}$ with dimensions
 261 $n_{\pi_1}, \dots, n_{\pi_{\hat{q}-1}}, n_q$ yields a row-major matrix \mathbf{A} with shape $(n_q, \prod_{r=1}^{\hat{q}-1} n_{\pi_r})$. Anal-
 262 ogously, tensor $\underline{\mathbf{C}}$ becomes a row-major matrix with the shape $(m, \prod_{r=1}^{\hat{q}-1} n_{\pi_r})$.
 263 This description supports all linear tensor layouts and generalizes lemma 4.2
 264 in [8].

265 Algorithm 1 needs a minor modification so that `gemm` can be used with flat-
 266 tened subtensors instead of tensor slices. The modified algorithm therefor iter-
 267 ates only over modes larger than \hat{q} in the non-base case and hence omits the
 268 first \hat{q} modes $\pi_{1,\hat{q}} = (\pi_1, \dots, \pi_{\hat{q}})$ with $\pi_{\hat{q}} = q$. The conditions in line 2 and 4 are
 269 changed to $1 < r \leq \hat{q}$ and $\hat{q} < r$, respectively. The single indices of the subtensors
 270 $\underline{\mathbf{A}}'_{\pi_{1,\hat{q}}}$ and $\underline{\mathbf{C}}'_{\pi_{1,\hat{q}}}$ are given by the loop induction variables that belong to the
 271 π_r -th loop with $\hat{q} + 1 \leq r \leq p$.

272 4.5 Parallel BLAS-based Algorithms

273 The following paragraphs discuss three parallel approaches for the eighth case.
 274 Cases 1 to 7 already call a multi-threaded `gemm` and cannot be further optimized.

275
 276 **Sequential Loops and Multithreaded Matrix Multiplication** One straight
 277 forward approach is to use algorithm 1 as it is and to sequentially call a multi-
 278 threaded `gemm` in the base case of the algorithm as described in subsection 4.3.
 279 This is beneficial if $q = \pi_{p-1}$, the inner dimensions n_{π_1}, \dots, n_q are large or the
 280 outer-most dimension n_{π_p} is smaller than the available processor cores. How-
 281 ever, if the above conditions are not met, the processor cores might not be fully
 282 utilized where each multi-threaded `gemm` is executed with small subtensors. We
 283 will refer to this algorithm version as `<seq-loops,par-gemm>` that is executable
 284 with subtensors or tensor slices.

285 **Parallel Loops and Single or Multithreaded Matrix Multiplication** A
 286 more advanced version of the above algorithm executes a single-threaded `gemm`
 287 in parallel including all available (free) modes which depend on the slicing. If
 288 subtensors are used, all $\pi_{\hat{q}+1}, \dots, \pi_p$ modes are free. In case of tensor slices, only
 289 π_1 and $\pi_{\hat{q}}$ are free modes. The corresponding maximum degree of parallelism for
 290 both cases are $\prod_{r=\hat{q}+1}^p n_{\pi_r}$ and $\prod_{r=1}^p n_r / (n_{\pi_1} n_{\pi_{\hat{q}}})$, respectively.

291 Using tensor slices for the multiplication, $\underline{\mathbf{A}}$ and $\underline{\mathbf{C}}$ are flattened twice with
 292 $\varphi_{\pi_{\hat{q}+1}, \pi_p}$ and $\varphi_{\pi_2, \pi_{\hat{q}-1}}$. The resulting tensor is of order 4 with dimensions $n_{\pi_1},$
 293 $\hat{n}_{\pi_2}, n_q, \hat{n}_{\pi_4}$ 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-
 294 recursion has been transformed in two loops. The outer loop iterates over \hat{n}_{π_4}
 295 while the inner loop iterates over \hat{n}_{π_2} calling `gemm` with slices $\underline{\mathbf{A}}'_{\pi_{1,q}}$ and $\underline{\mathbf{C}}'_{\pi_{1,q}}$.
 296 Both loops are parallelized using `omp parallel for` together with the `collapse(2)`
 297 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}_{π_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 batched Matrix Multiplication 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. We measured a peak performance of 1.78 double precision Tflops 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**.

If not otherwise mentioned, both tensors $\underline{\mathbf{A}}$ and $\underline{\mathbf{C}}$ are stored according to the first-order linear tensor layout with $\pi = (1, \dots, p)$ whereas matrix \mathbf{B} has the row-major storage format.

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

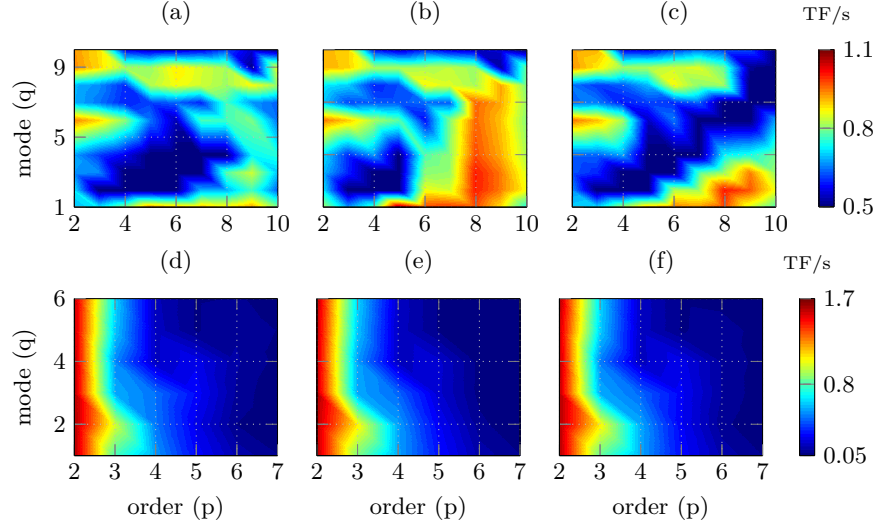


Fig. 1. Performance maps in double-precision Tflops of the proposed algorithms with varying tensor orders p and contraction modes q . Tensors are asymmetrically-shaped on the upper plots and symmetrically-shaped on the lower plots. In (a) and (d) function `<gemm_batch>` is executed, in (b) and (e) `<par-loops,seq-gemm>` with tensor slices, in (c) and (f) `<par-loops,seq-gemm>` with subtensors.

340 tuples where the tensor order ranges from 2 to 7 and has 8 dimension tuples for
 341 each order. Each tensor dimension within the second set is 2^{12} , 2^8 , 2^6 , 2^5 , 2^4 and
 342 2^3 . A detailed explanation of the tensor shape setup is given in [2,3].

343 6 Results and Discussion

344 **Slicing Methods** The following paragraphs analyze the two proposed slicing
 345 methods by benchmarking the functions `<par-loops,seq-gemm>` and `<gemm_batch>`
 346 using asymmetrically (top) and symmetrically (bottom) shaped tensors. Fig. 1
 347 contains six contour plots (performance maps) in which `<par-loops,seq-gemm>`
 348 either uses subtensors or tensor slices and `<gemm_batch>` loops over subtensors
 349 only. Each point within the performance map represents a mean value that has
 350 been averaged over tensor sizes for a tensor order¹.

351 For asymmetrically shaped tensors, function `<par-loops,seq-gemm>` with ten-
 352 sor slices performs on average 18% better than with subtensors. Our function
 353 `<par-loops,seq-gemm>` with tensor slices is on average 11% faster than Intel’s
 354 `gemm_batch` routine and reaches almost 1.1 Tflops for non-edge cases with $q > 2$
 355 and $p > 6$. This suggests that the Intel’s implementation does not divide sub-
 356 tensors into smaller blocks.

¹ Note that Fig. 2 suggests that the contraction mode q can be greater than p which is not possible. Our profiling program sets $q = p$ in such cases.

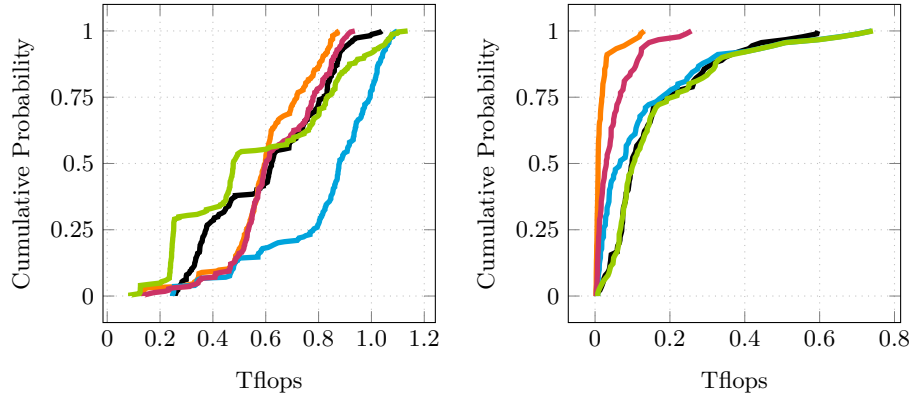


Fig. 2. Cumulative performance distributions of the proposed algorithms for the eighth case. Each distribution line belongs to one algorithm: `<gemm_batch>` (black), `<seq-loops,par-gemm>` (orange) and `<par-loops,seq-gemm>` (blue) using tensor slices, `<seq-loops,par-gemm>` (pink) and `<par-loops,seq-gemm>` (green) using subtenors. Tensors are asymmetrically (left plot) and symmetrically shaped (right plot).

357 With symmetrically shaped tensors, `<par-loops,seq-gemm>` with tensor slices
 358 performs almost identical as `<gemm_batch>` with 221.52 Gflops and 236.21 Gflops,
 359 respectively. Moreover, the slicing method seems to have only little affect on the
 360 overall runtime behavior of `<par-loops,seq-gemm>`. In contrast to the perfor-
 361 mance maps with asymmetrically shaped tensors, all functions almost reach the
 362 attainable peak performance of 1.7 Tflops when $p = 2$. This can be by the fact that
 363 both dimensions are equal or larger than 4096 enabling `gemm` to operate under
 364 optimal conditions.

365 **Parallelization Methods** The contour plots in Fig. 1 contain performance
 366 data of all cases except for 4 and 5, see Table 1. The effects of the presented slic-
 367 ing and parallelization methods can be better understood if performance data of
 368 only the eighth case is examined. Fig. 2 contains cumulative performance distri-
 369 butions of all the proposed algorithms which are generated `gemm` or `gemm_batch`
 370 calls within case 8. As the distribution is empirically generated, the probability
 371 y of a point (x, y) on a distribution function corresponds to the number of test
 372 cases of a particular algorithm that achieves x or less Tflops. For instance, func-
 373 tion `<seq-loops,par-gemm>` with subtenors computes the tensor-matrix product
 374 with equal to or less than 0.6 Tflops for 50% percent of the test cases using
 375 asymmetrically shaped tensor. Consequently, distribution functions with an ex-
 376 ponential growth is favorable while logarithmic behavior is less desirable. The
 377 test set cardinality for case 8 is 255 for asymmetrically shaped tensors and 91
 378 for symmetrically ones.

379 In case of asymmetrically shaped tensors, `<par-loops,seq-gemm>` with tensor
 380 slices performs best and outperforms `<gemm_batch>`. One unexpected finding is
 381 that function `<seq-loops,par-gemm>` with any slicing strategy performs better

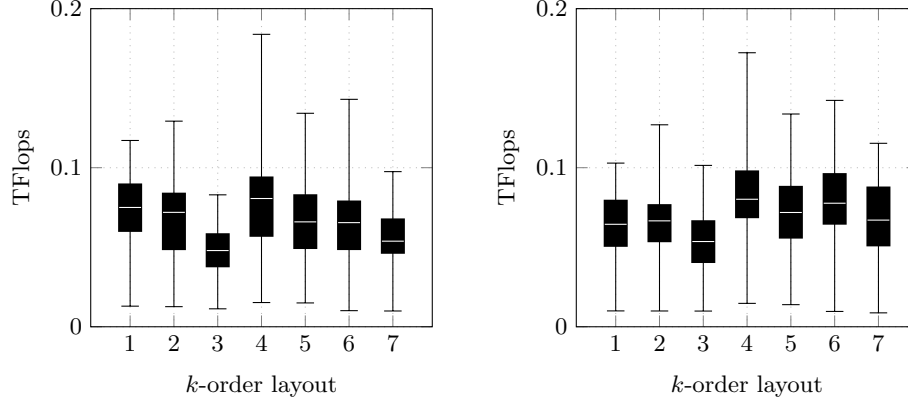


Fig. 3. Box plots visualizing performance statistics in double-precision Tflops of `<gemm_batch>` (left) and `<par-loops, seq-gemm>` with subtensors (right). Box plot number k denotes the k -order tensor layout of symmetrically shaped tensors with order 7.

than `<gemm_batch>` when the tensor order p and contraction mode q satisfy $4 \leq p \leq 7$ and $2 \leq q \leq 4$, respectively. Functions executed with symmetrically shaped tensors reach at most 743 Gflops for the eighth case which is less than half of the attainable peak performance of 1.7 Tflops. This is expected as cases 2 and 3 are not considered. Functions `<par-loops, seq-gemm>` with subtensors and `<gemm_batch>` have almost the same performance distribution outperforming `<seq-loops, par-gemm>` for almost every test case. Function `<par-loops, seq-gemm>` with tensor slices is on average almost as fast as with subtensors. However, if the tensor order is greater than 3 and the tensor dimensions are less than 64, its running time increases by almost a factor of 2.

These observations suggest to use `<par-loops, seq-gemm>` with tensor slices for common cases in which the leading and contraction dimensions are larger than 64 elements. Subtensors should only be used if the leading dimension n_{π_1} of $\underline{\mathbf{A}}_{\pi_1, q}$ and $\underline{\mathbf{C}}_{\pi_1, q}$ falls below 64. This strategy is different to the one presented in [8] that maximizes the number of modes involved in the matrix multiply. We have also observed no performance improvement if `par-gemm` was used with `par-loops` which is why their distribution functions are not shown in Fig. 2. Moreover, in most cases the `seq-loops` implementations are independent of the tensor shape slower than `par-loops`, even for smaller tensor slices.

Layout-Oblivious Algorithms Fig. 3 contains two subfigures visualizing performance statistics in double-precision Tflops of `<gemm_batch>` (left subfigure) and `<par-loops, seq-gemm>` with subtensors (right subfigure). Each box plot with the number k has been computed from benchmark data with symmetrically shaped order-7 tensors with the k -order tensor layout. The 1-order and 7-order layout are the first- and last-order storage formats for the order-7 tensor with $\pi_F = (1, 2, \dots, 7)$ and $\pi_L = (7, 6, \dots, 1)$. The definition of k -order tensor layouts can be found in section 3.

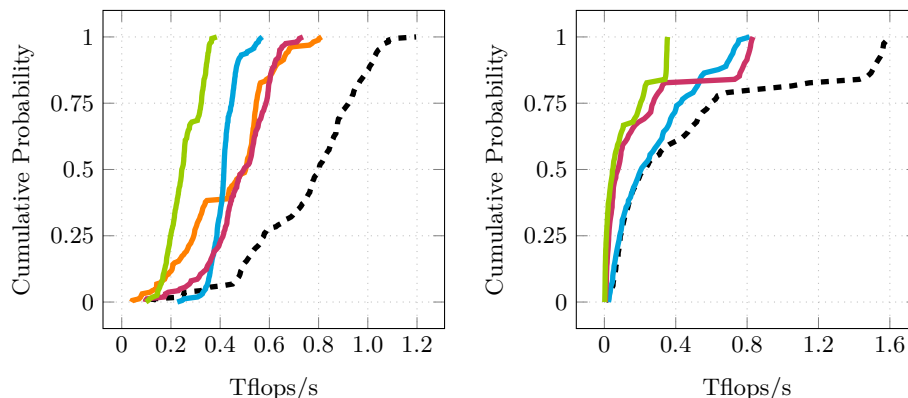


Fig. 4. Cumulative performance distributions of tensor-times-matrix algorithms in double-precision Tflops. Each distribution line belongs to a library: **tlib**[ours] (---), **tcl** (—), **tblis** (—), **libtorch** (—), **eigen** (—). Libraries have been tested with asymmetrically-shaped (left plot)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

409 The low performance can be attributed to the fact that the contraction di-
 410 mension of subtensors of tensor slices of symmetrically shaped order-7 tensors
 411 are 8 while the leading dimension is 8 or at most 48 for subtensors. The relative
 412 standard deviation of `<gemm_batch>`'s and `<par-loops,seq-gemm>`'s median values
 413 are 12.95% and 17.61%. Their respective interquartile range are similar with a
 414 relative standard deviation of 22.25% and 15.23%.

415 The runtime results with different k -order tensor layouts show that the per-
 416 formance of our proposed algorithms is not designed for a specific tensor layout.
 417 Moreover, the performance stays within an acceptable range independent of the
 418 tensor layout.

419 **Comparison with other Approaches** We have compared our best imple-
 420 mentation with four libraries that implement the tensor-matrix multiplication
 421 using different approaches. Library **tcl** implements the TTGT approach with
 422 a high-perform tensor-transpose library **hptt** which is discussed in [15]. **tblis**
 423 implements the GETT approach that is akin to Blis' algorithm design for the
 424 matrix multiplication [10]. The tensor extension of **eigen** (v3.3.7) is used by the
 425 Tensorflow framework. Library **libtorch** (v2.3.0) is the C++ distribution of Py-
 426 Torch. **tlib** denotes our library using algorithm `<par-loops,seq-gemm>` that have
 427 been presented in the previous paragraphs.

428 Fig. 2 contains cumulative performance distributions for the complete test
 429 sets comparing the performance distribution of our implementation with the pre-
 430 viously mentioned libraries. Note that we only have used tensor slices for asym-
 431 metrically shaped tensors (left plot) and subtensors for symmetrically shaped
 432 tensors (right plot). Our implementation with a median performance of 793.75
 433 Gflops outperforms others' for almost every asymmetrically shaped tensor in the
 434 test set. The median performances of **tcl**, **tblis**, **libtorch** and **eigen** are 503.61,

435 415.33, 496.22 and 244.69 Gflops reaching on average 74.11%, 61.14%, 76.68%
 436 and 39.34% of **tlib**'s throughputs.

437 In case of symmetrically shaped tensors the performance distributions of all
 438 libraries on the right plot in Fig. 2 are much closer. The median performances of
 439 **tlib**, **tblis**, **libtorch** and **eigen** are 228.93, 208.69, 76.46, 46.25 Gflops reaching
 440 on average 73.06%, 38.89%, 19.79% of **tlib**'s throughputs². All libraries operate
 441 with 801.68 or less Gflops for the cases 2 and 3 which is almost half of **tlib**'s
 442 performance with 1579 Gflops. The median performance and the interquartile
 443 range of **tblis** and **tlib** for the cases 6 and 7 are almost the same. Their respective
 444 median Gflops are 255.23 and 263.94 for the sixth case and 121.17 and 144.27
 445 for the seventh case. This explains the similar performance distributions when
 446 their performance is less than 400 Gflops. **Libtorch** and **eigen** compute the
 447 tensor-matrix product, in median, with 17.11 and 9.64 Gflops/s, respectively.
 448 Our library **tlib** has a median performance of 102.11 Gflops and outperforms
 449 **tblis** with 79.35 Gflops for the eighth case.

450 7 Conclusion and Future Work

451 We presented efficient layout-oblivious algorithms for the compute-bound tensor-
 452 matrix multiplication which is essential for many tensor methods. Our approach
 453 is based on the LOG-method and computes the tensor-matrix product in-place
 454 without transposing tensors. It applies the flexible approach described in [2] and
 455 generalizes the findings on tensor slicing in [8]. The resulting algorithms are able
 456 to process dense tensors with arbitrary tensor order, dimensions and with any
 457 linear tensor layout all of which can be runtime variable.

458 Our benchmarks show that dividing the base algorithm into eight different
 459 **gemm** cases improves the overall performance. We have demonstrated that algo-
 460 rithms with parallel loops over single-threaded **gemm** calls with tensor slices and
 461 subtensors perform best. Interestingly, they outperform a single **gemm_batch** call
 462 with subtensors, on average, by 14% in case of asymmetrically shaped tensors
 463 and if tensor slices are used. Both version computes the tensor-matrix prod-
 464 uct on average at least by 14.05% and up to a factor of 3.79 faster than other
 465 state-of-the-art implementations.

466 Summarizing our findings, LOG-based tensor-times-matrix algorithms are
 467 able to outperform TTGT-based and GETT-based implementations without
 468 sacrificing flexibility or maintainability. Hence, other actively developed libraries
 469 such as LibTorch and Eigen will benefit from implementing the proposed algo-
 470 rithms. Our header-only library provides C++ interfaces and a python module
 471 which allows frameworks to easily integrate our library.

472 In the future, we intend to generalize LOG-based approach for general ten-
 473 sor contractions with the same flexibility that we offered for the tensor-matrix
 474 multiplication. We would like to further optimize the tensor-matrix multiplica-

² We were unable to run tcl with our test set containing symmetrically shaped tensors.
 We suspect a very high memory demand to be the reason.

tion based on benchmark results of matrix-matrix products which might lead to better runtime results for edge cases.

Source Code Availability Project description and source code can be found at <https://github.com/bassoy/ttm>. The sequential tensor-matrix multiplication of TLIB is part of uBLAS and in the official release of Boost v1.70.0 or later.

References

1. Bader, B.W., Kolda, T.G.: Algorithm 862: Matlab tensor classes for fast algorithm prototyping. *ACM Trans. Math. Softw.* **32**, 635–653 (December 2006)
2. Bassoy, C.: Design of a high-performance tensor-vector multiplication with blas. In: *International Conference on Computational Science*. pp. 32–45. Springer (2019)
3. 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)
4. Goto, K., Geijn, R.A.v.d.: Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software (TOMS)* **34**(3) (2008)
5. 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)
6. Kolda, T.G., Bader, B.W.: Tensor decompositions and applications. *SIAM review* **51**(3), 455–500 (2009)
7. 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)
8. Li, J., Battaglino, C., Perros, I., Sun, J., Vuduc, R.: An input-adaptive and in-place approach to dense tensor-times-matrix multiply. In: *High Performance Computing, Networking, Storage and Analysis*, 2015. pp. 1–12. IEEE (2015)
9. Lim, L.H.: Tensors and hypermatrices. In: Hogben, L. (ed.) *Handbook of Linear Algebra*. Chapman and Hall, 2 edn. (2017)
10. Matthews, D.A.: High-performance tensor contraction without transposition. *SIAM Journal on Scientific Computing* **40**(1), C1–C24 (2018)
11. 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)
12. 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)
13. 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)
14. 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)
15. Springer, P., Bientinesi, P.: Design of a high-performance gemm-like tensor-tensor multiplication. *ACM Transactions on Mathematical Software (TOMS)* **44**(3), 28 (2018)