# Fast Layout-Oblivious Tensor-Matrix Multiplication with BLAS

Cem Bassoy

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

**Abstract.** The computation of the tensor-matrix product are required in various tensor methods, e.g. for computing the ALS or HOSVD. This paper presents a high-performance algorithm for the mode-$q$ tensor-matrix multiplication using the Loops-over-GEMMs (LOG) approach with dense tensors that can have any linear tensor layout, tensor order and dimensions. The proposed algorithm either directly call efficient implementations of GEMM with tensors or recursively apply GEMM on higher-order tensor slices multiple times. We discuss different strategies for fusing and executing the matrix-matrix multiplication in parallel. Using OpenBLAS, our parallel implementation attains ??? Gflops/s in single precision on a Core i9-7900X Intel Xeon processor. We show that the performance of our implementation is independent of the tensor layout and a performance of ??? can be sustained for any linear tensor format. Our version of the tensor-matrix multiplication is on average ???x and up to ???x faster than state-of-the-art approaches.

## 1 Introduction

Numerical multilinear algebra has become ubiquitous in many scientific domains such as computational neuroscience, pattern recognition, signal processing and data mining [4,11]. Tensors representing large amount of multidimensional data are decomposed and analyzed with the help of basic tensor operations where the contraction of tensors plays a central role [5,6]. To support numeric computations, the development and analysis of high-performance kernels for the tensor contraction have gained greater attention. Based on the Transpose-Transpose-GEMM-Transpose (TGGT) approach, [2,13] reorganize tensors in order to perform a tensor contraction with an optimized matrix-matrix multiplication (GEMM) implementation. A more recent method, GEMM-like Tensor-Tensor Multiplication (GETT), is to design algorithms according to high-performance GEMM [1,9,14]. Other methods are based on the LOG approach in which algorithms utilize GEMM with multiple tensor slices [7,10,12]. Focusing on class 3 compute-bound tensor contractions with free tensor indices, most implementations of the above mentioned approaches reach near peak performance of the computing machine [9,12,14].

In this work, we design and analyze high-performance algorithms for the tensor-vector multiplication that is used in many numerical algorithms, e.g.

the higher-order power method [2, 5, 6]. Our analysis is motivated by the observation that implementations for class 3 tensor contractions do not perform equally well for tensor-vector multiplications. Our approach is akin to the one proposed in [7, 12] but targets the utilization of general matrix-vector multiplication routines (`GEMV`) using `OpenBLAS` [15] without code generation. We present new recursive in-place algorithms that compute the tensor-vector multiplication by executing `GEMV` with slices and fibers of tensors. Moreover, except for few corner cases, we demonstrate that in-place tensor-vector multiplications with any contraction mode can be implemented with one recursive algorithm using multiple slice-vector multiplications and only one `GEMV` parameter configuration. For parallel execution, we propose a variable loop fusion method with respect to the slice order of slice-vector multiplications. Our algorithms support dense tensors with any order, dimensions and any non-hierarchical layouts including the first- and the last-order storage formats for any contraction mode. We have quantified the impact of the tensor layout, tensor slice order and parallel execution of slice-vector multiplications with varying contraction modes. The runtime measurements of our implementations are compared with those presented in [1,9,14]. In summary, the main findings of our work are:

- A tensor-vector multiplication is implementable by an in-place algorithm with 1 `DOT` and 7 `GEMV` parameter configurations supporting all combinations of contraction mode, tensor order, dimensions and non-hierarchical storage format validating the second recipe in [10] with a precise description.
- Algorithms with variable loop fusion and parallel slice-vector multiplications can achieve the peak performance of a `GEMV` with large slice dimensions. The use of order-2 tensor slices helps to retain the performance at a peak level.
- A `LOG`-based implementation is able to compute a tensor-vector product faster than `TTGT`- and `GETT`-based implementations that have been described in [1, 9, 14]. Using symmetrically shaped tensors, an average speedup of 3 to 6x for single and double precision floating point computations can be achieved.

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

## 2 Related Work

The authors in [10] 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.

To cite this paper please use the final published version:
https://dx.doi.org/10.1007/978-3-030-22734-0_3

The work in [7] 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 [14] 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 [9] 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 [14].

## 3 Background

**Notation** An order-$p$ *tensor* is a $p$-dimensional array or hypermatrix with $p$ modes [8]. For instance, scalars, vectors and matrices are order-0, order-1 and order-2 tensors. We write $a$, $\mathbf{a}$, $\mathbf{A}$ and $\underline{\mathbf{A}}$ in order to denote scalars, vectors, matrices and tensors. In general we will assume the order of a tensor to be $p$ and explicitly mention it otherwise. Each dimension $n_r$ of the $r$-th mode shall satisfy $n_r > 1$. The $p$-tuple $\mathbf{n}$ with $\mathbf{n} = (n_1, n_2, \ldots, n_p)$ will be referred to as a *dimension tuple*. We will use round brackets $\underline{\mathbf{A}}(i_1, i_2, \ldots, i_p)$ or $\underline{\mathbf{A}}(\mathbf{i})$ together with a multi-index $\mathbf{i} = (i_1, i_2, \ldots, i_p)$ to identify tensor elements. The set of all multi-indices of a tensor is denoted by $\mathcal{I}$ which is defined as the Cartesian product of all index sets $I_r = \{1, \ldots, n_r\}$. The set $\mathcal{J} = \{0, \ldots, \bar{n} - 1\}$ contains (relative) memory positions of an order-$p$ tensor with $\bar{n} = n_1 \cdot n_2 \cdots n_p$ contiguously stored elements with $|\mathcal{I}| = |\mathcal{J}|$. A *subtensor* denoted by $\underline{\mathbf{A}}'$ shall reference or view a subset of tensor elements where the references are specified in terms of $p$ index ranges. The $r$-th *index range* shall be given by an index pair denoted by $f_r : l_r$ with $1 \leq f_r \leq l_r \leq n_r$. We will use $:_r$ to specify a range with all elements of the $r$-th index set. A subtensor is an order-$p'$ *slice* if all modes of the corresponding order-$p$ tensor are selected either with a full index range or a single index where $p'$ with $p' \leq p$ is the number of all non-singleton dimensions. A *fiber* is a tensor slice with only one dimension greater than 1.

**Non-Hierarchical Storage Formats and Memory Access** Given a dense order-$p$ tensor, we use a *layout tuple* $\boldsymbol{\pi} \in \mathbb{N}^p$ to encode non-hierarchical storage formats such as the well known first-order or last-order layout. They contain permuted tensor modes whose priority is given by their index. For $1 \leq k \leq p$,

To cite this paper please use the final published version:
https://dx.doi.org/10.1007/978-3-030-22734-0_3

an element $\pi_r$ of *k-order layout* tuple is defined as $k - r + 1$ if $1 < r \leq k$ and $r$ in any other case. The well-known first- and last-order storage formats are then given by $\boldsymbol{\pi}_F = (1, 2, \ldots, p)$ and $\boldsymbol{\pi}_L = (p, p-1, \ldots, 1)$. Given a layout tuple $\boldsymbol{\pi}$ with $p$ modes, the $\pi_r$-th element of a *stride tuple* is given by

$$w_{\pi_r} = n_{\pi_1} \cdot n_{\pi_2} \cdots n_{\pi_{r-1}} \quad \text{for } 1 < r \leq p. \tag{1}$$

With $w_{\pi_1} = 1$, tensor elements of the $\pi_1$-th mode are contiguously stored in memory. In contrast to hierarchical storage formats, all tensor elements with one differing multi-index element exhibit the same stride.

The location of tensor elements within the allocated memory space is determined by the storage format of a tensor and the corresponding *layout function*. For a given layout and stride tuple, a layout function $\lambda_{\mathbf{w}} : \mathcal{I} \to \mathcal{J}$ maps a multi-index to a scalar index according to

$$\lambda_{\mathbf{w}}(\mathbf{i}) = \sum_{r=1}^{p} w_r (i_r - 1) \tag{2}$$

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 $\underline{\mathbf{A}}$.

**Tensor-Vector Multiplication** Let $\underline{\mathbf{A}}$ be an order-$p$ input tensor with a dimension tuple $\mathbf{n} = (n_1, \ldots, n_q, \ldots, n_p)$ and let $\mathbf{b}$ be a vector of length $n_q$ with $p > 1$. Let $\underline{\mathbf{C}}$ be a tensor with $p - 1$ modes with a dimension tuple $\mathbf{m} = (n_1, \ldots, n_{q-1}, n_{q+1}, \ldots, n_p)$. A mode-$q$ tensor-vector multiplication is denoted by $\underline{\mathbf{C}} = \underline{\mathbf{A}} \times_q \mathbf{b}$ where

$$c_{i_1,\ldots,i_{q-1},i_{q+1},\ldots,i_p} = \sum_{i_q=1}^{n_q} a_{i_1,\ldots,i_q,\ldots,i_p} \cdot b_{i_q} \tag{3}$$

is an element of $\underline{\mathbf{C}}$. Eq. (3) is an inner product of a fiber of $\underline{\mathbf{A}}$ and $\mathbf{b}$.The mode $q$ is its *contraction mode*. We additionally term $\boldsymbol{\pi}$ as the layout tuple of the input tensor $\underline{\mathbf{A}}$ with a stride tuple $\mathbf{w}$ that is given by Eq. (1). With no transposition of $\underline{\mathbf{A}}$ or $\underline{\mathbf{C}}$, elements of the layout tuple $\boldsymbol{\varphi}$ of the mode-$q$ tensor-vector product $\underline{\mathbf{C}}$ are given by

$$\varphi_j = \begin{cases} \pi_k & \text{if } \pi_k < \pi_q \\ \pi_k - 1 & \text{if } \pi_k > \pi_q \end{cases} \quad \text{and} \quad j = \begin{cases} k & \text{if } k < q \\ k - 1 & \text{if } k > q \end{cases} \tag{4}$$

for $k = 1, \ldots, p$. The stride tuple $\mathbf{v}$ is given by Eq. (1) using the shape $\mathbf{m}$ and permutation tuple $\boldsymbol{\varphi}$ of $\underline{\mathbf{C}}$.

## 4 Algorithm Design

### 4.1 Standard Algorithms with Contiguous Memory Access

The control and data flow of the basic tensor-vector multiplication algorithm implements Eq. (3) with a single function. It uses tree recursion with a control flow

To cite this paper please use the final published version:
https://dx.doi.org/10.1007/978-3-030-22734-0_3

```
1  tensor_times_vector_recursive(A, b, C, n, i, q, q̂, r)
2      if r = q̂ then
3          tensor_times_vector_recursive(A, b, C, n, i, q, q̂, r − 1)
4      else if r > 1 then
5          for i_{π_r} ← 1 to n_{π_r} do
6              tensor_times_vector_recursive(A, b, C, n, i, q, q̂, r − 1)
7      else
8          for i_q ← 1 to n_q do
9              for i_{π_1} ← 1 to n_{π_1} do
10                 C(i_1, ..., i_{q−1}, i_{q+1}, ..., i_p) += A(i_1, ..., i_q, ..., i_p) · b(i_q)
```

**Algorithm 1:** Recursive implementation of the tensor-vector multiplication starting with $r = p$ for $p \geq 2$ and $1 \leq \pi_1 \neq q \leq p$ with better data locality for large dimensions. Iteration along mode $\hat{q}$ with $\hat{q} = (\pi^{-1})_q$ is moved into the inner-most recursion level.

akin to one of Algorithm 1 in [3]. Instead of combining two scalars elementwise in the inner-most loop, the tensor-vector multiplication algorithm computes an inner product and skips the iteration over the $q$-th index set, i.e. the $q$-th loop. The algorithm supports tensors with arbitrary order, dimensions and any non-hierarchical storage format. However, it accesses memory non-contiguously if the storage format does not prioritize the $q$-th mode with $\pi_1 \neq q$ and $w_q > 1$, see Eq. (1). The access pattern can be enhanced by modifying the tensor layout, i.e reordering tensor elements according to the storage format. A reordering however, limits its overall performance of the contraction operation [12].

As proposed in [3], elements can be accessed according to the storage format using a permutation tuple. In this way, the desired index set for a given recursion level can be selected. By inserting the $q$-th (contraction) loop into an already existing branch for $r > 1$ additionally simplifies the algorithm's control-flow. Yet the loop-reordering forces the first $\bar{n}_{k-1} = \prod_{r=1}^{k-1} n_{\pi_r}$ elements of $\underline{\mathbf{C}}$ to be accessed $n_q$-times with $\pi_k = q$. If the number of reaccessed elements exceeds the last-level cache size, cache missus occur resulting in a poor performance of the algorithm with longer execution times.

Algorithm 1 improves the data locality if the number of elements $\bar{n}_{k-1}$ exceeds the cache size. By nesting the $\pi_1$-th loop inside the $i_q$-th loop, the function only reuses $n_{\pi_1}$ elements. This is done by inserting an `if`-statement at the very beginning of the function which skips the $q$-th loop when $r = \hat{q}$ with $\hat{q} = (\pi^{-1})_q$ where $\hat{q}$ is the index position of $q$ within $\boldsymbol{\pi}$. The proposed algorithm constitutes the starting point for `BLAS` utilization.

### 4.2 Extended Algorithms utilizing BLAS

The number of reused elements in Algorithm 1 can be further minimized by tiling the inner-most loops. Instead of applying loop transformations as proposed

To cite this paper please use the final published version:
https://dx.doi.org/10.1007/978-3-030-22734-0_3

| Case | Order $p$ | Layout $\boldsymbol{\pi}$ | Mode $q$ | Routine | FORMAT | M | N | LDA |
|---|---|---|---|---|---|---|---|---|
| 1 | 1 | - | 1 | DOT | - | $n_1$ | - | - |
| 2 | 2 | $(1,2)$ | 1 | GEMV | ROW | $n_2$ | $n_1$ | $n_1$ |
| 3 | 2 | $(1,2)$ | 2 | GEMV | COL | $n_1$ | $n_2$ | $n_1$ |
| 4 | 2 | $(2,1)$ | 1 | GEMV | COL | $n_2$ | $n_1$ | $n_2$ |
| 5 | 2 | $(2,1)$ | 2 | GEMV | ROW | $n_1$ | $n_2$ | $n_2$ |
| 6 | $> 2$ | any | $\pi_1$ | GEMV | ROW | $\bar{n}_q$ | $n_q$ | $n_q$ |
| 7 | $> 2$ | any | $\pi_p$ | GEMV | COL | $\bar{n}_q$ | $n_q$ | $\bar{n}_q$ |
| 8 | $> 2$ | any | $\pi_2, \pi_3, \ldots, \pi_{p-1}$ | GEMV* | COL | $\hat{n}_q$ | $n_q$ | $\hat{n}_q$ |

**Table 1.** Parameter configuration of the DOT- and GEMV with eight cases executing a tensor-vector multiplication with respect to the order $p$, layout $\boldsymbol{\pi}$ and contraction mode $q$. All three parameters determine the values of FORMAT, M, N and LDA. GEMV* denotes a multiple execution of GEMV with different tensor slices. In case of order-2 and order-$\hat{q}$ slices, the number of rows must be equal to $\hat{n}_q = n_{\pi_1}$ and $\hat{n}_q = w_q$, respectively. The number of rows for case 6 and 7 is given by $\bar{n}_q = \prod_{r=1}^{p} n_r / n_q$.

in [9, 14], we apply highly optimized routines to fully or partly execute tensor contractions as it is done in [7, 12] for class 3 tensor operations. The function and parameter configurations for the tensor multiplication can be divided into eight cases.

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

*Case 2-5* ($p = 2$)*:* Let $\mathbf{A}$ be an order-2 tensor, i.e. matrix with dimensions $n_1$ and $n_2$. If $m = 2$ and if $\mathbf{A}$ is stored according to the column-major $\boldsymbol{\pi} = (1,2)$ or row-major format $\boldsymbol{\pi} = (2,1)$, the tensor-vector multiplication can be trivially executed by a GEMV routine using the tensor's storage format. The two remaining cases for $m = 1$ require an interpretation of the order-2 tensor. In case of the column-major format $\boldsymbol{\pi} = (1,2)$, the tensor-vector product can be computed with a GEMV routine, interpreting the columns of the matrix as rows with permuted dimensions. Analogously, a GEMV routine executes a tensor-vector multiplication with $\boldsymbol{\pi} = (2,1)$.

*Case 6-7* ($p > 2$)*:* General tensor-vector multiplications with higher-order tensors execute the GEMV routine multiple times over different slices of the tensor. There are two exceptions to the general case.If $\pi_1 = q$, a single GEMV routine is sufficient for any storage layout. The tensor can be interpreted as a matrix with $\bar{n}_q = \prod_{r=1}^{p} n_r / n_q$ rows and $n_q$ columns. The leading dimension LDA for $\pi_1 = q$ is $n_q$. Tensor fibers with contiguously stored elements are therefore interpreted as matrix rows. In case of $\pi_p = q$, the leading dimension LDA is given by $\bar{n}_q$ where all fibers with the exception of the dimension $\pi_p$ are interpreted as matrix columns. The interpretation of tensor objects does not copy data elements.

*Case 8* ($p > 2$)*:* For the last case with $\pi_1 \neq q$ and $\pi_p \neq q$, we provide two methods that loop over tensor slices. Lines 8 to 10 of Algorithm 1 perform a

To cite this paper please use the final published version:
https://dx.doi.org/10.1007/978-3-030-22734-0_3

slice-vector multiplication of the form $\mathbf{c}' = \mathbf{A}' \cdot \mathbf{b}$. It is executed with a `GEMV` with no further adjustment of the algorithm. The vector $\mathbf{c}'$ denotes a fiber of $\underline{\mathbf{C}}$ with $n_u$ elements and $\mathbf{A}'$ denotes an order-2 slice of $\underline{\mathbf{A}}$ with dimensions $n_u$ and $n_v$ such that

$$\mathbf{A}' = \underline{\mathbf{A}}(i_1, \ldots, :_u, \ldots, :_v, \ldots, i_p) \quad \text{and} \quad \mathbf{c}' = \underline{\mathbf{C}}(i_1, \ldots, :_u, \ldots, i_p) \tag{5}$$

where $u = \pi_1$ and $v = q$ or vice versa. Algorithm 1 needs a minor modification in order to loop over order-$\hat{q}$ slices. With $\hat{q} = (\pi^{-1})_q$, the conditions in line 2 and 4 are changed to $1 < r \leq \hat{q}$ and $\hat{q} < r$, respectively. The modified algorithms therefore omits the first $\hat{q}$ modes $\pi_1, \ldots, \pi_{\hat{q}}$ including $\pi_{\hat{q}} = q$ where all elements of an order-$\hat{q}$ slice are contiguously stored. Choosing the first-order storage format for convenience, the order-$\hat{q}$ and order-$(\hat{q}-1)$ slices of both tensors are given by

$$\underline{\mathbf{A}}' = \underline{\mathbf{A}}(:_1, \ldots, :_q, i_{q+1}, \ldots, i_p) \text{ and } \underline{\mathbf{C}}' = \underline{\mathbf{C}}(:_1, \ldots, :_{q-1}, i_{q+1}, \ldots, i_p). \tag{6}$$

The fiber $\mathbf{c}'$ of length $w_q = n_1 \cdot n_2 \cdots n_{q-1}$ is the one-dimensional interpretation of $\underline{\mathbf{C}}'$ and the order-2 slice $\mathbf{A}'$ with dimensions $w_q$ and $n_q$ the two-dimensional interpretation of $\underline{\mathbf{A}}'$. The slice-vector multiplication in this case can be performed with a `GEMV` that interprets the order-$\hat{q}$ slices as order-2 according to the description. Table 1 summarizes the call parameters of the `DOT` or `GEMV` for all order, storage format and contraction mode combinations.

### 4.3 Parallel Algorithms with Slice-Vector Multiplications

A straight-forward approach for generating a parallel version of Algorithm 1 is to divide the outer-most $\pi_p$-th loop into equally sized iterations and execute them in parallel using the `OpenMP parallel for` directive [3]. With no critical sections and synchronization points, all threads within the parallel region execute their own sequential slice-vector multiplications. The outer-most dimension $n_{\pi_p}$ determines the degree of parallelism, i.e. the number of parallel threads executing their own instruction stream.

Fusing additional loops into a single one improves the degree of parallelism. The number of fusible loops depends on the tensor order $p$ and contraction mode $q$ of the tensor-vector multiplication with $\hat{q} = (\pi^{-1})_q$. In case of mode-$q$ slice-vector multiplications, loops $\pi_{\hat{q}+1}, \ldots, \pi_p$ are not involved in the multiplications and can be transformed into one single loop. For mode-2 slice-vector multiplications all loops except $\pi_1$ and $\pi_{\hat{q}}$ can be fused. When all fusible loops are lexically present and both parameters are known before compile time, loop fusion and parallel execution can be easily accomplished with the `OpenMP collapse` directive. The authors of [7] use this approach to generate parallel tensor-matrix functions.

With variable number of dimensions and a variable contraction mode, the iteration count of slice-vector multiplications and the slice selection needs to be determined at compile or run time. If $\bar{n}$ is the number of tensor elements of $\underline{\mathbf{A}}$, the total number of slice-vector multiplications with mode-$\hat{q}$ slices is given by $\bar{n}' = \bar{n}/w_q$. Using Eq. (1), the strides for the iteration are given by $w_{\pi_{\hat{q}+1}}$ for $\underline{\mathbf{A}}$ and

To cite this paper please use the final published version:
https://dx.doi.org/10.1007/978-3-030-22734-0_3

$v_{\pi_{\hat{q}}}$ for $\underline{\mathbf{C}}$. In summary, one single parallel outer loop with an iteration count $\bar{n}'$ and an increment variable $j$ iteratively calls mode-$\hat{q}$ slice-vector multiplications with adjusted memory location $j \cdot w_{\pi_{\hat{q}+1}}$ and $j \cdot v_{\pi_{\hat{q}}}$ for $\underline{\mathbf{A}}$ and $\underline{\mathbf{C}}$, respectively. The degree of parallelism $\prod_{r=\hat{q}+1}^{p} n_r$ decreases with increasing $\hat{q}$ and corresponds for $\hat{q} = p-1$ to the first parallel version. Tensor-vector multiplications with mode-2 slice-vector multiplications are further optimized by fusing additional $\hat{q}-2$ loops.

## 5 Experimental Setup

**Computing System** The experiments were carried out on a Core i9-7900X Intel Xeon processor with 10 cores and 20 hardware threads running at 3.3 GHz. It has a theoretical peak memory bandwidth of 85.312 GB/s resulting from four 64-bit wide channels with a data rate of 2666MT/s. The sizes of the L3-cache and each L2-cache are 14MB and 1024KB. The source code has been compiled with `GCC v7.3` using the highest optimization level `-Ofast` and `-march=native`, `-pthread` and `-fopenmp`. Parallel execution for the general case (8) has been accomplished using `GCC`'s implementation of the `OpenMP v4.5` specification. We have used the `DOT` and `GEMV` implementation of the `OpenBLAS` library `v0.2.20`. The benchmark results of each function are the average of 10 runs.

**Tensor Shapes** We have used *asymmetrically-shaped* and *symmetrically-shaped* tensors in order to provide a comprehensive test coverage. *Setup 1* performs runtime measurements with *asymmetrically-shaped* tensors. Their dimension tuples are organized in 10 two-dimensional arrays $\mathbf{N}_q$ with 9 rows and 32 columns where the dimension tuple $\mathbf{n}_{r,c}$ of length $r+1$ denotes an element $\mathbf{N}_q(r,c)$ of $\mathbf{N}_q$ with $1 \leq q \leq 10$. The dimension $\mathbf{n}_{r,c}(i)$ of $\mathbf{N}_q$ is 1024 if $i = 1$, $c \cdot 2^{15-r}$ if $i = \min(r+1, q)$ and 2 for any other index $i$ with $1 < q \leq 10$. The dimension $\mathbf{n}_{r,c}(i)$ of $\mathbf{N}_1$ is given by $c \cdot 2^{15-r}$ if $i = 1$, 1024 if $i = 2$ and 2 for any other index $i$. Dimension tuples of the same array column have the same number of tensor elements. Please note that with increasing tensor order (and row-number), the contraction mode is halved and with increasing tensor size, the contraction mode is multiplied by the column number. Such a setup enables an orthogonal test-set in terms of tensor elements ranging from $2^{25}$ to $2^{29}$ and tensor order ranging from 2 to 10. *Setup 2* performs runtime measurements with *symmetrically-shaped* tensors. Their dimension tuples are organized in one two-dimensional array $\mathbf{M}$ with 6 rows and 8 columns where the dimension tuple $\mathbf{m}_{r,c}$ of length $r+1$ denotes an element $\mathbf{M}(r,c)$ of $\mathbf{M}$. For $c = 1$, the dimensions of $\mathbf{m}_{r,c}$ are given by $2^{12}$, $2^8$, $2^6$, $2^5$, $2^4$ and $2^3$ with descending row number $r$ from 6 to 1. For $c > 1$, the remaining dimensions are given by $\mathbf{m}_{r,c} = \mathbf{m}_{r,c} + k \cdot (c-1)$ where $k$ is $2^9$, $2^5$, $2^3$, $2^2$, 2, 1 with descending row number $r$ from 6 to 1. In this setup, shape tuples of a column do not yield the same number of subtensor elements.

**Performance Maps** Measuring a single tensor-vector multiplication with the first setup produces $2880 = 9 \times 32 \times 10$ runtime data points where the tensor

To cite this paper please use the final published version:
https://dx.doi.org/10.1007/978-3-030-22734-0_3

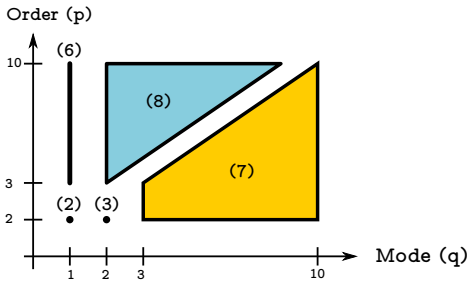


**Fig. 1.** Schematic contour view of the following average performance maps for the tensor-vector multiplication with tensors that are stored according to the first-order storage format. Each case $x$ in Table 1 affects a different region $x$ within the performance map. Performance values are the arithmetic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

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

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

## 6 Results and Discussion

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

To cite this paper please use the final published version:
https://dx.doi.org/10.1007/978-3-030-22734-0_3

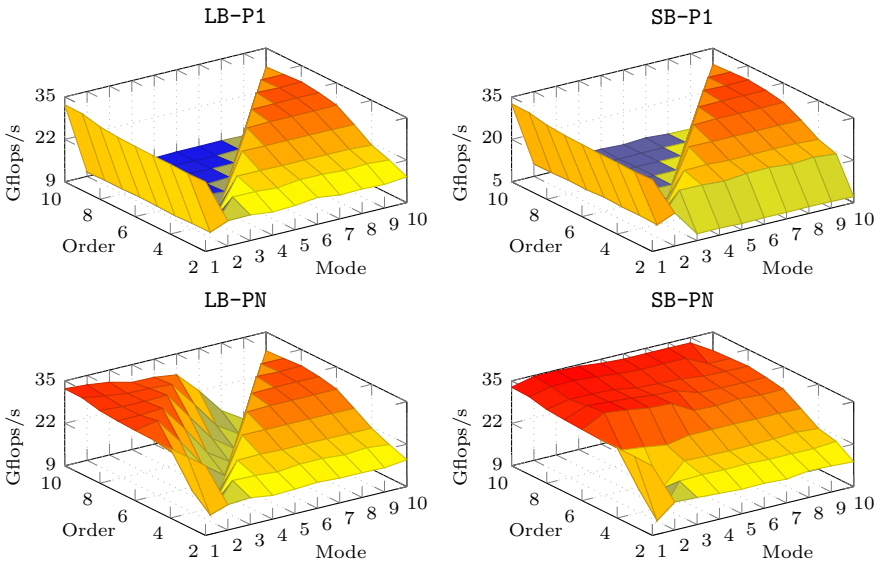


**Fig. 2.** Average performance maps of four tensor-vector multiplications with varying tensor orders $p$ and contraction modes $q$. Tensor elements are encoded in single-precision and stored contiguously in memory according to the first-order storage format. Tensors are *asymmetrically-shaped* with dimensions.

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 $n_{q+1}, \ldots, n_p$ are executed in parallel. The `PN` version speeds up the computation by almost a factor of 4x except for $q = p - 1$. This explains the notch in the left-bottom plot when $q = p - 1$ and $n_p = 2$.

To cite this paper please use the final published version:
https://dx.doi.org/10.1007/978-3-030-22734-0_3

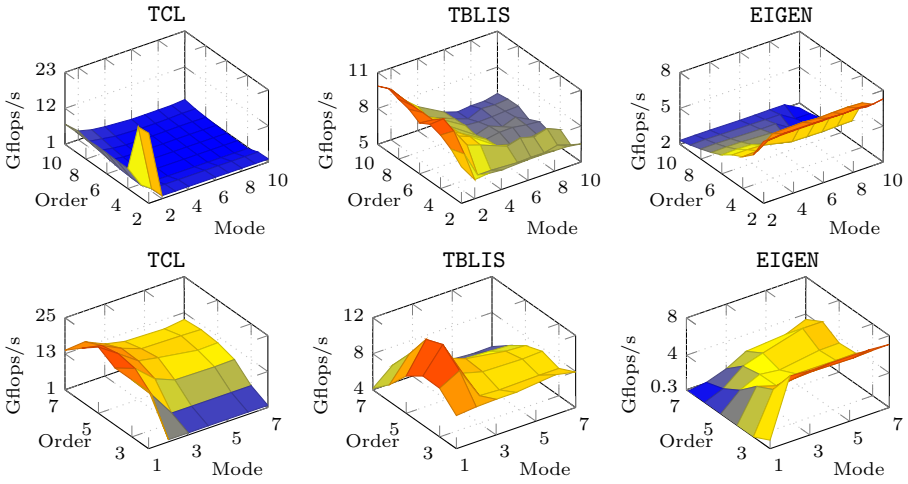


**Fig. 3.** Average performance maps of tensor-vector multiplication implementations using *asymmetrically-shaped* (top) and *symmetrically-shaped* (bottom) tensors with varying contraction modes and tensor order. Tensor elements are encoded in single-precision and stored contiguously in memory according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

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

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

To cite this paper please use the final published version:
https://dx.doi.org/10.1007/978-3-030-22734-0_3

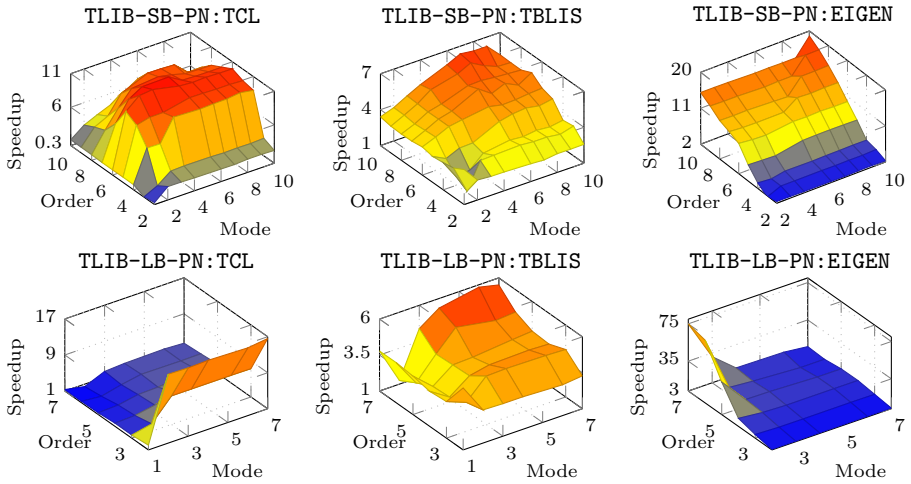


**Fig. 4.** Relative average performance maps of tensor-vector multiplication implementations using *asymmetrically* (top) and *symmetrically* (bottom) shaped tensors with varying contraction modes and tensor order. Relative performance (speedup) is the performance ratio of `TLIB-SB-PN` (top) and `TLIB-LB-PN` (bottom) to `TBLIS`, `TCL` and `EIGEN`, respectively. Tensor elements are encoded in single-precision and stored contiguously in memory according to the first-order storage format.

**Comparison with other Approaches** The following comparison includes three state-of-the-art libraries that implement three different approaches. The library `TCL` (`v0.1.1`) implements the (`TTGT`) approach with a high-perform tensor-transpose library `HPTT` which is discussed in [14]. `TBLIS` (`v1.0.0`) implements the `GETT` approach that is akin to `BLIS`'s algorithm design for matrix computations [9]. The tensor extension of `EIGEN` (`v3.3.90`) is used by the Tensorflow framework and performs the tensor-vector multiplication in-place and in parallel with contiguous memory access [1]. `TLIB` denotes our library that consists of sequential and parallel versions of the tensor-vector multiplication. Numerical results of `TLIB` have been verified with the ones of `TCL`, `TBLIS` and `EIGEN`.

Fig. 3 illustrates the average single-precision Gflops/s with asymmetrically- and symmetrically-shaped tensors in the first-order storage format. The runtime behavior of `TBLIS` and `EIGEN` with asymmetrically-shaped tensors is almost constant for varying tensor sizes with a standard deviation ranging between 2% and 13%. `TCL` shows a different behavior with 2 and 4 Gflops/s for any order $p \geq 2$ peaking at $p = 10$ and $q = 2$. The performance values however deviate from the mean value up to 60%. Computing the arithmetic mean over the set of contraction modes yields a standard deviation of less than 10% where the performance increases with increasing order peaking at $p = 10$. `TBLIS` performs best for larger contraction dimensions achieving up to 7 Gflops/s and slower runtimes with decreasing contraction dimensions. In case of symmetrically-shaped tensors, `TBLIS` and `TCL` achieve up to 12 and 25 Gflops/s in single precision with

To cite this paper please use the final published version:
https://dx.doi.org/10.1007/978-3-030-22734-0_3

a standard deviation between 6% and 20%, respectively. TCL and TBLIS behave similarly and perform better with increasing contraction dimensions. EIGEN executes faster with decreasing order and increasing contraction mode with at most 8 Gflops/s at $p = 2$ and $q \geq 2$.

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

## 7 Conclusion and Future Work

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

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

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

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

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

To cite this paper please use the final published version:
https://dx.doi.org/10.1007/978-3-030-22734-0_3

## References


1. Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., ..., Zheng, X.: Tensorflow: A system for large-scale machine learning. In: Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation. pp. 265–283. OSDI'16, USENIX Association, Berkeley, CA, USA (2016)
2. Bader, B.W., Kolda, T.G.: Algorithm 862: Matlab tensor classes for fast algorithm prototyping. ACM Trans. Math. Softw. **32**, 635–653 (December 2006)
3. Bassoy, C., 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. 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)
5. Kolda, T.G., Bader, B.W.: Tensor decompositions and applications. SIAM review **51**(3), 455–500 (2009)
6. 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)
7. Li, J., Battaglino, C., Perros, I., Sun, J., Vuduc, R.: An input-adaptive and in-place approach to dense tensor-times-matrix multiply. In: High Performance Computing, Networking, Storage and Analysis, 2015 SC-International Conference for. pp. 1–12. IEEE (2015)
8. Lim, L.H.: Tensors and hypermatrices. In: Hogben, L. (ed.) Handbook of Linear Algebra. Chapman and Hall, 2 edn. (2017)
9. Matthews, D.A.: High-performance tensor contraction without transposition. SIAM Journal on Scientific Computing **40**(1), C1–C24 (2018)
10. 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)
11. 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)
12. Shi, Y., Niranjan, U.N., Anandkumar, A., Cecka, C.: Tensor contractions with extended blas kernels on cpu and gpu. In: 2016 IEEE 23rd International Conference on High Performance Computing (HiPC). pp. 193–202 (Dec 2016)
13. 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)
14. Springer, P., Bientinesi, P.: Design of a high-performance gemm-like tensor–tensor multiplication. ACM Transactions on Mathematical Software (TOMS) **44**(3), 28 (2018)
15. Wang, Q., Zhang, X., Zhang, Y., Yi, Q.: Augem: automatically generate high performance dense linear algebra kernels on x86 cpus. In: High Performance Computing, Networking, Storage and Analysis (SC), 2013 International Conference for. pp. 1–12. IEEE (2013)



To cite this paper please use the final published version:
https://dx.doi.org/10.1007/978-3-030-22734-0_3