Analysis of high-performance tensor-matrix multiplication with BLAS

Cem Savaş Başsoy[a,*]

[a]*Hamburg University of Technology, Schwarzenbergstrasse 95, 21071, Hamburg, Germany*

Abstract

The tensor-matrix multiplication is a basic tensor operation required by various tensor methods such as the ALS and 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 that call BLAS with subtensors or tensor slices. Their performance is quantified on 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 tensor layout does not affect the performance significantly. Our fastest implementation is on average at least 14.05% and up to 3.79x 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 [1, 2]. These computations use basic tensor operations as building blocks for decomposing and analyzing multidimensional data which are represented by tensors [3, 4]. Tensor contractions are an important subset of basic operations that need to be fast for efficiently solving tensor methods.

There are three main approaches for implementing tensor contractions. The Transpose-Transpose-GEMM-Transpose (TGGT) approach reorganizes (flattens) tensors in order to perform a tensor contraction using optimized General Matrix Multiplication (GEMM) implementations [5, 6]. Implementations of the GEMM-like Tensor-Tensor multiplication (GETT) method have macro-kernels that are similar to the ones used in fast GEMM implementations [7, 8]. The third method is the Loops-over-GEMM (LoG) approach in which BLAS are utilized with multiple tensor slices or subtensors if possible [9, 10, 11, 12]. 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.

In this work, we present high-performance algorithms for the tensor-matrix multiplication which is used in many numerical methods such as the alternating least squares method [3, 4]. 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.

To our best knowledge, we are the first to combine the LoG approach described in [12, 13] for tensor-vector multiplications with the findings on tensor slicing for the tensor-matrix multiplication in [10]. Our algorithms support dense tensors with any order, dimensions and any linear tensor layout including the first- and the last-order storage formats for any contraction mode all of which can be runtime variable. They compute the tensor-matrix product in parallel using efficient GEMM or batched GEMM without transposing or flattening tensors. Despite their high performance, all algorithms are layout-oblivious and provide a sustained performance independent of the tensor layout and without tuning.

Moreover, every proposed algorithm can be implemented with less than 150 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-source `C++` implementation of all algorithms and a python interface for convenience. While Intel's MKL is used for our benchmarks, the user is free to select any other library that provides the BLAS interface and even integrate it's own implementation to be library independent.

The analysis in this work 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 state-of-the-art approaches discussed in [7, 8, 14] including Libtorch and Eigen. In summary, the main findings of our work are:

- A tensor-matrix multiplication can be implemented

[*]Corresponding author
Email address: `cem.bassoy@gmail.com` (Cem Savaş Başsoy)

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 fastest algorithm with tensor slices is on average 17% faster than Intel's batched GEMM implementation when the contraction and leading dimensions of the tensors are greater than 256.
- The proposed algorithms are layout-oblivious. Their performance does not vary significantly for different tensor layouts if the contraction conditions remain the same.
- 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

Springer et al. [7] 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.

Matthews [8] presents a runtime flexible tensor contraction library that uses GETT approach as well. 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 [7].

Li et al. [10] introduce InTensLi, a framework that generates in-place tensor-matrix multiplication according to the LOG approach. The authors discusses optimization and tuning techniques for slicing and parallelizing the operation. With optimized tuning parameters, they report a speedup of up to 4x over the TTGT-based MATLAB tensor toolbox library discussed in [5].

Başsoy [12] presents LoG-based algorithms that compute the tensor-vector product. They support dense tensors with linear tensor layouts, arbitrary dimensions and tensor order. The presented approach is to divide into eight cases calling GEMV and DOT. He reports average speedups of 6.1x and 4.0x compared to implementations that use the TTGT and GETT approach, respectively.

Pawlowski et al. [13] propose morton-ordered blocked layout for a mode-oblivious performance of the tensor-vector multiplication. Their algorithm iterate over blocked tensors and perform tensor-vector multiplications on blocked tensors. They are able to achieve high performance and mode-oblivious computations.

3. Background

3.1. Tensor Notation

An order-p tensor is a p-dimensional array where tensor elements are contiguously stored in memory[15, 3]. 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 order $p > 2$. The p-tuple $\mathbf{n} = (n_1, n_2, \ldots, n_p)$ will be referred to as the shape or dimension tuple of a tensor where $n_r > 1$. We will use round brackets $\underline{\mathbf{A}}(i_1, i_2, \ldots, i_p)$ or $\underline{\mathbf{A}}(\mathbf{i})$ to denote a tensor element where $\mathbf{i} = (i_1, i_2, \ldots, i_p)$ is a multi-index. For convenience, we will also use square brackets to concatenate index tuples such that $[\mathbf{i}, \mathbf{j}] = (i_1, i_2, \ldots, i_r, j_1, j_2, \ldots, j_q)$ where $\mathbf{i}$ and $\mathbf{j}$ are multi-indices of length r and q, respectively.

3.2. Subtensors

A subtensor references elements of a tensor $\underline{\mathbf{A}}$ and is denoted by $\underline{\mathbf{A}}'$. It is specified by a selection grid that consists of p index ranges. In this work, an index range of a given mode r shall either contain all indices of the mode r or a single index i_r of that mode where $1 \leq r \leq p$. Subtensor dimensions n'_r are either n_r if the full index range or 1 if a a single index for mode r is used. Subtensors are annotated by their non-unit modes such as $\underline{\mathbf{A}}'_{u,v,w}$ where $n_u > 1$, $n_v > 1$ and $n_w > 1$ for $1 \leq u \neq v \neq w \leq p$. The remaining single indices of a selection grid can be inferred by the loop induction variables of an algorithm. The number of non-unit modes determine the order p' of subtensor where $1 \leq p' < p$. In the above example, the subtensor $\underline{\mathbf{A}}'_{u,v,w}$ has three non-unit modes and is thus of order 3. For convenience, we might also use an dimension tuple $\mathbf{m}$ of length p' with $\mathbf{m} = (m_1, m_2, \ldots, m_{p'})$ to specify a mode-p' subtensor $\underline{\mathbf{A}}'_{\mathbf{m}}$. An order-2 subtensor of $\underline{\mathbf{A}}'$ is a tensor slice $\mathbf{A}'_{u,v}$ and an order-1 subtensor of $\underline{\mathbf{A}}'$ is a fiber $\mathbf{a}'_u$.

3.3. 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 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 $k < r \leq p$. The first- and last-order storage formats are given by $\boldsymbol{\pi}_F = (1, 2, \ldots, p)$ and $\boldsymbol{\pi}_L = (p, p-1, \ldots, 1)$. An inverse layout tuple $\boldsymbol{\pi}^{-1}$ is defined by $\boldsymbol{\pi}^{-1}(\boldsymbol{\pi}(k)) = k$. Given a layout tuple $\boldsymbol{\pi}$ with p modes, the π_r-th element of a stride tuple is given by $w_{\pi_r} = \prod_{k=1}^{r-1} n_{\pi_k}$ for $1 < r \leq p$ and $w_{\pi_1} = 1$. Tensor elements of the π_1-th mode are contiguously stored in memory. The location of tensor elements 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 $\lambda_{\mathbf{w}}(\mathbf{i}) = \sum_{r=1}^{p} w_r(i_r - 1)$, see [16, 13].

3.4. Flattening and Reshaping

The following two operations define non-modifying reformatting transformations of dense tensors with contiguously stored elements and linear tensor layouts.

The flattening operation $\varphi_{u,v}$ transforms an order-p tensor $\underline{\mathbf{A}}$ with a shape $\mathbf{n}$ and layout $\boldsymbol{\pi}$ tuple to an order-p' view $\underline{\mathbf{B}}$ with a shape $\mathbf{m}$ and layout $\boldsymbol{\tau}$ tuple of length p' with $p' = p - v + u$ and $1 \leq u < v \leq p$. It is akin to tensor unfolding, also known as matricization and vectorization [4, p.459]. However, it neither modifies the element ordering nor copies tensor elements. Given a layout tuple $\boldsymbol{\pi}$ of $\underline{\mathbf{A}}$, the flattening operation $\varphi_{u,v}$ is defined for contiguous modes $\hat{\boldsymbol{\pi}} = (\pi_u, \pi_{u+1}, \ldots, \pi_v)$ of $\boldsymbol{\pi}$. With $j_k = 0$ if $k \leq u$ and $j_k = v - u$ if $k > u$ where $1 \leq k \leq p'$, the resulting layout tuple $\boldsymbol{\tau} = (\tau_1, \ldots, \tau_{p'})$ of $\underline{\mathbf{B}}$ is then given by $\tau_u = \min(\boldsymbol{\pi}_{u,v})$ and

$$\tau_k = \pi_{k+j_k} - s_k \quad \text{for } k \neq u$$

with $s_k = |\{\pi_i \mid \pi_{k+j_k} > \pi_i \wedge \pi_i \neq \min(\hat{\boldsymbol{\pi}}) \wedge u \leq i \leq p\}|$. Elements of the shape tuple $\mathbf{m}$ are defined by $m_{\tau_u} = \prod_{k=u}^{v} n_{\pi_k}$ and $m_{\tau_k} = n_{\pi_{k+j}}$ for $k \neq u$.

3.5. Tensor-Matrix Multiplication

Let $\underline{\mathbf{A}}$ and $\underline{\mathbf{C}}$ be order-p tensors with shapes $\mathbf{n}_a = ([\mathbf{n}_1, n_q, \mathbf{n}_2])$ and $\mathbf{n}_c = ([\mathbf{n}_1, m, \mathbf{n}_2])$ where $\mathbf{n}_1 = (n_1, n_2, \ldots, n_{q-1})$ and $\mathbf{n}_2 = (n_{q+1}, n_{q+2}, \ldots, n_p)$. Let $\mathbf{B}$ be a matrix of shape $\mathbf{n}_b = (m, n_q)$. A q-mode tensor-matrix product is denoted by $\underline{\mathbf{C}} = \underline{\mathbf{A}} \times_q \mathbf{B}$. An element of $\underline{\mathbf{C}}$ is defined by

$$\underline{\mathbf{C}}([\mathbf{i}_1, j, \mathbf{i}_2]) = \sum_{i_q=1}^{n_q} \underline{\mathbf{A}}([\mathbf{i}_1, i_q, \mathbf{i}_2]) \cdot \mathbf{B}(j, i_q) \qquad (1)$$

with $\mathbf{i}_1 = (i_1, \ldots, i_{q-1})$, $\mathbf{i}_2 = (i_{q+1}, \ldots, i_p)$ where $1 \leq i_r \leq n_r$ and $1 \leq j \leq m$ [10, 4]. Mode q is called the contraction mode with $1 \leq q \leq p$. The tensor-matrix multiplication generalizes the computational aspect of the two-dimensional case $\mathbf{C} = \mathbf{B} \cdot \mathbf{A}$ if $p = 2$ and $q = 1$. Its arithmetic intensity is equal to that of a matrix-matrix multiplication and is not memory-bound.

In the following, we assume that the tensors $\underline{\mathbf{A}}$ and $\underline{\mathbf{C}}$ have the same tensor layout $\boldsymbol{\pi}$. Elements of matrix $\underline{\mathbf{B}}$ can be stored either in the column-major or row-major format. The tensor-matrix multiplication with i_q iterating over the second mode of $\mathbf{B}$ is also referred to as the q-mode product which is a building block for tensor methods such as the higher-order orthogonal iteration or the higher-order singular value decomposition [4]. Please note that the following method can be applied, if indices j and i_q of matrix $\mathbf{B}$ are swapped.

3.6. BLAS

The Basic Linear Algebra Subprograms (BLAS) are Fortran routines that perform basic vector and matrix operations. They are considered as a de-facto standard for writing efficient and portable linear algebra software which is why most processor vendors provide efficient machine-specific BLAS implementations. The BLAS are subdivided into three groups. While routines in the first level and second level operate on matrices and vectors, third level routines perform matrix operations. With a high arithmetic intensity, level three routines are compute-bound.

CBLAS denotes the C interface to the BLAS that can be used by C or C++ programs where each C function is prefixed with `cblas`. For the sake of simplicity, we omit the `cblas` prefix of CBLAS functions. The following algorithms implementing the tensor-matrix multiplication uses the general matrix-matrix multiplication `gemm` and only for one special case the matrix-vector multiplications `gemv`. Function `gemm` is defined as `C:=a*op(A)*op(B)+b*C` where `a` and `b` are scalars, `A`, `B` and `C` are matrices, `op(A)` is an `M`-by-`K` matrix, `op(B)` is a `K`-by-`N` matrix and `C` is an `N`-by-`N` matrix. Function `op(x)` either transposes the corresponding matrix `x` such that `op(x)=x'` or not `op(x)=x`. In CBLAS matrices are passed by reference using pointers to contiguously stored matrix elements.

The BLAS interfaces also allow the user to specify the leading dimension of a matrix by providing the `lda`, `ldb` and `ldc` parameters. A leading dimension determines the number of elements that is required for iterating over the non-contiguous matrix dimension. This allows matrix operations to perform on submatrices or even matrix fibers where fiber elements are contiguously stored in memory. The leading dimension parameter is necessary for implementing BLAS-based tensor-matrix multiplication. The CBLAS interface also provides an additional layout parameter `CBLAS_LAYOUT` with which one can specify the column-major or row-major ordering. The selection of the storage format prevents unnecessary transpose operations for the tensor-matrix multiplication by choosing the best format for tensor slices.

4. Algorithm Design

4.1. Baseline Algorithm with Contiguous Memory Access

The tensor-times-matrix multiplication in equation 1 can be implemented with one sequential algorithm using a nested recursion [16]. It consists of two `if` statements with an `else` branch that computes a fiber-matrix product with two loops. The outer loop iterates over the dimension m of $\underline{\mathbf{C}}$ and $\mathbf{B}$, while the inner iterates over dimension n_q of $\underline{\mathbf{A}}$ and $\mathbf{B}$ computing an inner product with fibers of $\underline{\mathbf{A}}$ and $\mathbf{B}$.

```
1  ttm($\underline{\mathbf{A}}, \mathbf{B}, \underline{\mathbf{C}}, \mathbf{n}, \boldsymbol{\pi}, \mathbf{i}, m, q, \hat{q}, r$)
2    if $r = \hat{q}$ then
3      ttm($\underline{\mathbf{A}}, \mathbf{B}, \underline{\mathbf{C}}, \mathbf{n}, \boldsymbol{\pi}, \mathbf{i}, m, q, \hat{q}, r-1$)
4    else if $r > 1$ then
5      for $i_{\pi_r} \leftarrow 1$ to $n_{\pi_r}$ do
6        ttm($\underline{\mathbf{A}}, \mathbf{B}, \underline{\mathbf{C}}, \mathbf{n}, \boldsymbol{\pi}, \mathbf{i}, m, q, \hat{q}, r-1$)
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           $\underline{\mathbf{C}}([\mathbf{i}_1, j, \mathbf{i}_2])$ += $\underline{\mathbf{A}}([\mathbf{i}_1, i_q, \mathbf{i}_2]) \cdot \mathbf{B}(j, i_q)$
```

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

While matrix $\mathbf{B}$ can be accessed contiguously depending on its storage format, elements of $\underline{\mathbf{A}}$ and $\underline{\mathbf{C}}$ are accessed non-contiguously if $\pi_1 \neq q$.

A better approach is illustrated in algorithm 1 where the loop order is adjusted to the tensor layout $\boldsymbol{\pi}$ and memory is accessed contiguously for $\pi_1 \neq q$ and $p > 1$. The adjustment of the loop order is accomplished in line 5 which uses the layout tuple $\boldsymbol{\pi}$ to select a multi-index element i_{π_r} and to increment it with the corresponding stride w_{π_r}. Hence, with increasing recursion level and decreasing r, indices are incremented with smaller strides as $w_{\pi_r} \leq w_{\pi_{r+1}}$. The second `if` statement in line number 4 allows the loop over mode π_1 to be placed into the base case which contains three loops performing a slice-matrix multiplication. In this way, the inner-most loop is able to increment i_{π_1} with a unit stride and contiguously accesses tensor elements of $\underline{\mathbf{A}}$ and $\underline{\mathbf{C}}$. The second loop increments i_q with which elements of $\mathbf{B}$ are contiguously accessed if $\mathbf{B}$ is stored in the row-major format. The third loop increments j and could be placed as the second loop if $\mathbf{B}$ is stored in the column-major format.

While spatial data locality is improved by adjusting the loop ordering, slices $\underline{\mathbf{A}}'_{\pi_1,q}$, fibers $\underline{\mathbf{C}}'_{\pi_1}$ and elements $\underline{\mathbf{B}}(j, i_q)$ are accessed m, n_q and n_{π_1} times, respectively. The specified fiber of $\underline{\mathbf{C}}$ might fit into first or second level cache, slice elements of $\underline{\mathbf{A}}$ are unlikely to fit in the local caches if the slice size $n_{\pi_1} \times n_q$ is large, leading to higher cache misses and suboptimal performance. Instead of optimizing for better temporal data locality, we use existing high-performance BLAS implementations for the base case. The following subsection explains this approach.

4.2. BLAS-based Algorithms with Tensor Slices

Algorithm 1 computes the mode-q tensor-matrix product in a recursive fashion. The base case multiplies different tensor slices of $\underline{\mathbf{A}}$ with the matrix $\mathbf{B}$. Instead of optimizing the slice-matrix multiplication in the base case, one can use a `gemm` routine instead. Note that algorithm 1 is the general case for $p \geq 2$ but only executable for $\pi_1 \neq q$. For $\pi_1 = q$, the tensor-matrix product can be computed by a matrix-matrix multiplication where the input tensor $\underline{\mathbf{A}}$ can be flattened into a matrix without any copy operation. The same can be applied when $\pi_p = q$ and five other cases where the input tensor is either one or two-dimensional. In summary, there are seven other corner cases to the general case where a single `gemv` or `gemm` call suffices to compute the tensor-matrix product. All eight cases per storage format are listed in table 1. The arguments of the CBLAS routines `gemv` or `gemm` are set according to the tensor order p, tensor layout $\boldsymbol{\pi}$ and contraction mode q. If the input matrix $\mathbf{B}$ has the row-major order, parameter `CBLAS_ORDER` of function `gemm` is set to `CblasRowMajor` and `CblasColMajor` otherwise. Note that table 1 supports all linear tensor layouts of $\underline{\mathbf{A}}$ and $\underline{\mathbf{C}}$ with no limitations on tensor order and contraction mode. The following subsection describes all eight cases when the input matrix $\mathbf{B}$ has the row-major order.

4.2.1. Row-Major Matrix Multiplication

Case 1: If $p = 1$, The tensor-vector product $\underline{\mathbf{A}} \times_1 \mathbf{B}$ can be computed with a `gemv` operation where $\underline{\mathbf{A}}$ is an order-1 tensor $\mathbf{a}$ of length n_1 such that $\mathbf{a}^T \cdot \mathbf{B}$.

Case 2-5: If $p = 2$, $\underline{\mathbf{A}}$ and $\underline{\mathbf{C}}$ are order-2 tensors with dimensions n_1 and n_2. In this case the tensor-matrix product can be computed with a single `gemm`. If $\mathbf{A}$ and $\mathbf{C}$ have the column-major format with $\boldsymbol{\pi} = (1, 2)$, `gemm` either executes $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}^T$ for $q = 1$ or $\mathbf{C} = \mathbf{B} \cdot \mathbf{A}$ for $q = 2$. Both matrices can be interpreted $\mathbf{C}$ and $\mathbf{A}$ as matrices in row-major format although both are stored column-wise. If $\mathbf{A}$ and $\mathbf{C}$ have the row-major format with $\boldsymbol{\pi} = (2, 1)$, `gemm` either executes $\mathbf{C} = \mathbf{B} \cdot \mathbf{A}$ for $q = 1$ or $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}^T$ for $q = 2$. The transposition of $\mathbf{B}$ is necessary for the cases 2 and 5 which is independent of the chosen layout.

Case 6-7 : If $p > 2$ and if $q = \pi_1$(case 6), a single `gemm` with the corresponding arguments executes $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}^T$ and computes a tensor-matrix product $\underline{\mathbf{C}} = \underline{\mathbf{A}} \times_{\pi_1} \mathbf{B}$. Tensors $\underline{\mathbf{A}}$ and $\underline{\mathbf{C}}$ are flattened with $\varphi_{2,p}$ to row-major matrices $\mathbf{A}$ and $\mathbf{C}$. Matrix $\mathbf{A}$ has $\bar{n}_{\pi_1} = \bar{n}/n_{\pi_1}$ rows and n_{π_1} columns while matrix $\mathbf{C}$ has the same number of rows and m columns. If $\pi_p = q$ (case 7), $\underline{\mathbf{A}}$ and $\underline{\mathbf{C}}$ are flattened with $\varphi_{1,p-1}$ to column-major matrices $\mathbf{A}$ and $\mathbf{C}$. Matrix $\mathbf{A}$ has n_{π_p} rows and $\bar{n}_{\pi_p} = \bar{n}/n_{\pi_p}$ columns while $\mathbf{C}$ has m rows and the same number of columns. In this case, a single `gemm` executes $\mathbf{C} = \mathbf{B} \cdot \mathbf{A}$ and computes $\underline{\mathbf{C}} = \underline{\mathbf{A}} \times_{\pi_p} \mathbf{B}$. Noticeably, the desired contraction are performed without copy operations, see subsection 3.4.

Case 8 ($p > 2$): If the tensor order is greater than 2 with $\pi_1 \neq q$ and $\pi_p \neq q$, the modified baseline algorithm 1 is used to successively call $\bar{n}/(n_q \cdot n_{\pi_1})$ times `gemm` with different tensor slices of $\underline{\mathbf{C}}$ and $\underline{\mathbf{A}}$. Each `gemm` computes one slice $\underline{\mathbf{C}}'_{\pi_1,q}$ of the tensor-matrix product $\underline{\mathbf{C}}$ using the corresponding tensor slices $\underline{\mathbf{A}}'_{\pi_1,q}$ and the matrix $\mathbf{B}$. The matrix-matrix product $\mathbf{C} = \mathbf{B} \cdot \mathbf{A}$ is performed by interpreting both tensor slices as row-major matrices $\mathbf{A}$ and $\mathbf{C}$

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

Table 1: Eight cases of CBLAS functions `gemm` and `gemv` implementing the mode-q tensor-matrix multiplication with a row-major or column-major format. Arguments `T`, `M`, `N`, etc. of `gemv` and `gemm` are chosen with respect to the tensor order p, layout $\boldsymbol{\pi}$ of $\underline{\mathbf{A}}$, $\mathbf{B}$, $\underline{\mathbf{C}}$ and contraction mode q where `T` specifies if $\mathbf{B}$ is transposed. Function `gemm*` with a star denotes multiple `gemm` calls with different tensor slices. Argument $\bar{n}_q$ for case 6 and 7 is defined as $\bar{n}_q = (\prod_r^p n_r)/n_q$. Input matrix $\mathbf{B}$ is either stored in the column-major or row-major format. The storage format flag set for `gemm` and `gemv` is determined by the element ordering of $\mathbf{B}$.

which have the dimensions (n_q, n_{π_1}) and (m, n_{π_1}), respectively.

4.2.2. Column-Major Matrix Multiplication

The tensor-matrix multiplication is performed with the column-major version of `gemm` when the input matrix $\mathbf{B}$ is stored in column-major order. Although the number of `gemm` cases remains the same, the `gemm` arguments must be rearranged. The argument arrangement for the column-major version can be derived from the row-major version that is provided in table 1.

Firstly, the BLAS arguments of `M` and `N`, as well as `A` and `B` must be swapped. Additionally, the transposition flag for matrix $\mathbf{B}$ is toggled. Also, the leading dimension argument of `A` is swapped to `LDB` or `LDA`. The only new argument is the new leading dimension of `B`.

Given case 4 with the row-major matrix multiplication in table 1 where tensor $\underline{\mathbf{A}}$ and matrix $\mathbf{B}$ are passed to `B` and `A`. The corresponding column-major version is attained when tensor $\underline{\mathbf{A}}$ and matrix $\mathbf{B}$ are passed to `A` and `B` where the transpose flag for $\mathbf{B}$ is set and the remaining dimensions are adjusted accordingly.

4.2.3. Matrix Multiplication Variations

The column-major and row-major versions of `gemm` can be used interchangeably by adapting the storage format. This means that a `gemm` operation for column-major matrices can compute the same matrix product as one for row-major matrices, provided that the arguments are rearranged accordingly. While the argument rearrangement is similar, the arguments associated with the matrices `A` and `B` must be interchanged. Specifically, `LDA` and `LDB` as well as `M` and `N` are swapped along with the corresponding matrix pointers. In addition, the transposition flag must be set for `A` or `B` in the new format if `B` or `A` is transposed in the original version.

For instance, the column-major matrix multiplication in case 4 of table 1 requires the arguments of `A` and `B` to be tensor $\underline{\mathbf{A}}$ and matrix $\mathbf{B}$ with $\mathbf{B}$ being transposed. The arguments of an equivalent row-major multiplication for `A`, `B`, `M`, `N`, `LDA`, `LDB` and `T` are then initialized with $\mathbf{B}$, $\underline{\mathbf{A}}$, m, n_2, m, n_2 and $\mathbf{B}$.

Another possible matrix multiplication variant with the same product is computed when, instead of $\mathbf{B}$, tensors $\underline{\mathbf{A}}$ and $\underline{\mathbf{C}}$ with adjusted arguments are transposed. We assume that such reformulations of the matrix multiplication do not outperform the variants shown in Table 1, as we expect highly optimized BLAS libraries to adjust .

4.3. Matrix Multiplication with Subtensors

Algorithm 1 can be slightly modified in order to call `gemm` with flattened order-$\hat{q}$ subtensors that correspond to larger tensor slices. Given the contraction mode q with $1 < q < p$, the maximum number of additionally fusible modes is $\hat{q} - 1$ with $\hat{q} = \boldsymbol{\pi}^{-1}(q)$ where $\boldsymbol{\pi}^{-1}$ is the inverse layout tuple. The corresponding fusible modes are therefore $\pi_1, \pi_2, \dots, \pi_{\hat{q}-1}$.

The non-base case of the modified algorithm only iterates over dimensions that have indices larger than $\hat{q}$ and thus omitting the first $\hat{q}$ modes. The conditions in line 2 and 4 are changed to $1 < r \leq \hat{q}$ and $\hat{q} < r$, respectively. Thus, loop indices belonging to the outer π_r-th loop with $\hat{q} + 1 \leq r \leq p$ define the order-$\hat{q}$ subtensors $\underline{\mathbf{A}}'_{\boldsymbol{\pi}'}$ and $\underline{\mathbf{C}}'_{\boldsymbol{\pi}'}$ of $\underline{\mathbf{A}}$ and $\underline{\mathbf{C}}$ with $\boldsymbol{\pi}' = (\pi_1, \dots, \pi_{\hat{q}-1}, q)$. Flattening the subtensors $\underline{\mathbf{A}}'_{\boldsymbol{\pi}'}$ and $\underline{\mathbf{C}}'_{\boldsymbol{\pi}'}$ with $\varphi_{1,\hat{q}-1}$ for the modes $\pi_1, \dots, \pi_{\hat{q}-1}$ yields two tensor slices with dimension n_q or m and the fused dimension $\bar{n}_q = \prod_{r=1}^{\hat{q}-1} n_{\pi_r}$ with $\bar{n}_q = w_q$. Both tensor slices can be interpreted either as row-major or column-major matrices with shapes $(n_q, \bar{n}_q)$ or $(w_q, \bar{n}_q)$

in case of $\underline{\mathbf{A}}$ and $(m, \bar{n}_q)$ or $(\bar{n}_q, m)$ in case of $\underline{\mathbf{C}}$, respectively.

The `gemm` function in the base case is called with almost identical arguments except for the parameter M or N which is set to $\bar{n}_q$ for a column-major or row-major multiplication, respectively. Note that neither the selection of the subtensor nor the flattening operation copy tensor elements. This description supports all linear tensor layouts and generalizes lemma 4.2 in [10] without copying tensor elements, see section 3.4.

4.4. Parallel BLAS-based Algorithms

Most BLAS libraries provide API functions for adjusting the number of threads. Hence, functions such as `gemm` and `gemv` can be run either using a single or multiple threads. For the cases one to seven with a single BLAS call, the number of threads is always set to the number of available cores. The following subsections discuss parallel versions for the eighth case in which the outer loops of algorithm 1 and the `gemm` function inside the base case can be run in parallel. Note that the parallelization strategies can be combined with the aforementioned slicing methods.

4.4.1. Sequential Loops and Parallel Matrix Multiplication

Similar to the first seven cases, algorithm 1 does not need to be modified except for enabling `gemm` to run multi-threaded in the base case by using BLAS function calls to adjust the thread count. This type of parallelization strategy might be beneficial with order-$\hat{q}$ tensor slices where the contraction mode satisfies $q = \pi_{p-1}$, the inner dimensions $n_{\pi_1}, \ldots, n_{\hat{q}}$ are large and the outer-most dimension n_{π_p} is smaller than the available processor cores. For instance, given a first-order storage format and the contraction mode q with $q = p-1$ and $n_p = 2$, the dimensions of flattened order-q tensor slices are $\prod_{r=1}^{p-2} n_r$ and n_{p-1}. This allows `gemm` to be executed with large dimensions using multiple threads increasing the likelihood to reach a high throughput. However, if the above conditions are not met, a multi-threaded `gemm` operates on small tensor slices which might lead to an suboptimal utilization of the available cores. This algorithm version will be referred to as `<par-gemm>`. Depending on the subtensor shape, we will either add `<slice>` for order-2 subtensors or `<subtensor>` for order-$\hat{q}$ subtensors with $\hat{q} = \pi_q^{-1}$.

4.4.2. Parallel Loops and Sequential Matrix Multiplication

Instead of sequentially calling multi-threaded `gemm`, it is also possible to call single-threaded `gemm`s in parallel. Similar to the previous approach, the matrix multiplication can be performed with tensor slices or order-$\hat{q}$ subtensors.

Matrix Multiplication with Tensor Slices. Algorithm 2 with function `ttm<par-loop><slice>` executes a single-threaded `gemm` with tensor slices in parallel using all modes except π_1 and $\pi_{\hat{q}}$. The first statement of the algorithm calls the `flatten` function which transforms tensors $\underline{\mathbf{A}}$ and $\underline{\mathbf{C}}$

```
ttm<par-loop><slice>($\underline{\mathbf{A}}$, $\mathbf{B}$, $\underline{\mathbf{C}}$, $\mathbf{n}$, $\boldsymbol{\pi}$, $m$, $q$, $p$)
    [$\underline{\mathbf{A}}'$, $\underline{\mathbf{C}}'$, $\mathbf{n}'$, $\mathbf{w}'$] = flatten ($\underline{\mathbf{A}}$, $\underline{\mathbf{C}}$, $\mathbf{n}$, $m$, $\boldsymbol{\pi}$, $q$, $p$)
    parallel for $i \leftarrow 1$ to $n'_4$ do
        parallel for $j \leftarrow 1$ to $n'_2$ do
            gemm($m$, $n'_1$, $n'_3$, 1, $\mathbf{B}$,$n'_3$, $\underline{\mathbf{A}}'_{ij}$,$w'_3$, 0, $\underline{\mathbf{C}}'_{ij}$,$w'_3$)
```

Algorithm 2: Function `ttm<par-loop><slice>` is an optimized version of Algorithm 1. The `flatten` function transforms the order-p tensors $\underline{\mathbf{A}}$ and $\underline{\mathbf{C}}$ with layout tuple $\boldsymbol{\pi}$ and their respective dimension tuples $\mathbf{n}$ and $\mathbf{m}$ into order-4 tensors $\underline{\mathbf{A}}'$ and $\underline{\mathbf{C}}'$ with layout tuple $\boldsymbol{\pi}'$ and their respective dimension tuples $\mathbf{n}'$ and $\mathbf{m}'$ where $\mathbf{n}' = (n_{\pi_1}, \hat{n}_{\pi_2}, n_q, \hat{n}_{\pi_4})$ and $m'_3 = m$ and $n'_k = m'_k$ for $k \neq 3$. Each thread calls multiple single-threaded `gemm` functions each of which executes a slice-matrix multiplication with the order-2 tensor slices $\underline{\mathbf{A}}'_{ij}$ and $\underline{\mathbf{C}}'_{ij}$. Matrix $\mathbf{B}$ has the row-major storage format.

without copying elements by calling the flattening operation $\varphi_{\pi_{\hat{q}+1},\pi_p}$ and $\varphi_{\pi_2,\pi_{\hat{q}-1}}$. The resulting tensors $\underline{\mathbf{A}}'$ and $\underline{\mathbf{C}}'$ are of order 4. Tensor $\underline{\mathbf{A}}'$ has the shape $\mathbf{n}' = (n_{\pi_1}, \hat{n}_{\pi_2}, n_q, \hat{n}_{\pi_4})$ with the dimensions $\hat{n}_{\pi_2} = \prod_{r=2}^{\hat{q}-1} n_{\pi_r}$ and $\hat{n}_{\pi_4} = \prod_{r=\hat{q}+1}^{p} n_{\pi_r}$. Tensor $\underline{\mathbf{C}}'$ has the same shape as $\underline{\mathbf{A}}'$ with dimensions $m'_r = n'_r$ except for the third dimension which is given by $m_3 = m$.

The following two `parallel for` loop constructs index all free modes. The outer loop iterates over $n'_4 = \hat{n}_{\pi_4}$ while the inner one loops over $n'_2 = \hat{n}_{\pi_2}$ calling `gemm` with tensor slices $\underline{\mathbf{A}}'_{2,4}$ and $\underline{\mathbf{C}}'_{2,4}$. Here, we assume that matrix $\mathbf{B}$ has the row-major format which is why both tensor slices are also treated as row-major matrices. Notice that `gemm` in Algorithm 2 will be called with exact same arguments as displayed in the eighth case in table 1 where $n'_1 = n_{\pi_1}$, $n'_3 = n_q$ and $w_q = w'_3$. For the sake of simplicity, we omitted the first three arguments of `gemm` which are set to `CblasRowMajor` and `CblasNoTrans` for `A` and `B`. With the help of the flattening operation, the tree-recursion has been transformed into two loops which iterate over all free indices.

Matrix Multiplication with Subtensors. The following algorithm and the flattening of subtensors is a combination of the previous paragraph and subsection 4.3. With order-$\hat{q}$ subtensors, only the outer modes $\pi_{\hat{q}+1}, \ldots, \pi_p$ are free for parallel execution while the inner modes $\pi_1, \ldots, \pi_{\hat{q}-1}, q$ are used for the slice-matrix multiplication. Therefore, both tensors are flattened twice using the flattening operations $\varphi_{\pi_1,\pi_{\hat{q}-1}}$ and $\varphi_{\pi_{\hat{q}+1},\pi_p}$. Note that in contrast to tensor slices, the first flattening also contains the dimension n_{π_1}. The flattened tensors are of order 3 where $\underline{\mathbf{A}}'$ has the shape $\mathbf{n}' = (\hat{n}_{\pi_1}, n_q, \hat{n}_{\pi_3})$ with $\hat{n}_{\pi_1} = \prod_{r=1}^{\hat{q}-1} n_{\pi_r}$ and $\hat{n}_{\pi_3} = \prod_{r=\hat{q}+1}^{p} n_{\pi_r}$. Tensor $\underline{\mathbf{C}}'$ has the same dimensions as $\underline{\mathbf{A}}'$ except for $m_2 = m$.

Algorithm 2 needs a minor modification for supporting order-$\hat{q}$ subtensors. Instead of two loops, the modified algorithm consists of a single loop which iterates over dimension $\hat{n}_{\pi_3}$ calling a single-threaded `gemm` with large tensor slices $\underline{\mathbf{A}}'$ and $\underline{\mathbf{C}}'$. The shape and strides of both tensor slices as well as the function arguments of `gemm` have already been provided by the previous subsection 4.3. This `ttm` version will referred to as `<par-loop><subtensor>`.

Note that functions `<par-gemm>` and `<par-loop>` implement opposing versions of the `ttm` where either `gemm` or the fused loop is performed in parallel. Version `<par-loop-gemm` executes available loops in parallel where each loop thread executes a multi-threaded `gemm` with either subtensors or tensor slices.

4.4.3. 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.

4.4.4. OpenMP Parallelization

The two `parallel for` loops have been parallelized using the OpenMP directive `omp parallel for` together with the `schedule(static)`, `num_threads(ncores)` and `proc_bind (spread)` clauses. In case of tensor-slices, the `collapse(2)` clause is added for transforming both loops into one loop which has an iteration space of the first loop times the second one.

The `num_threads(ncores)` clause specifies the number of threads within a team where `ncores` is equal to the number of processor cores. Hence, each OpenMP thread is responsible for computing $\bar{n}'/$`ncores` independent slice-matrix products where $\bar{n}' = n_2' \cdot n_4'$ for tensor slices and $\bar{n}' = n_4'$ for mode-$\hat{q}$ subtensors.

The `schedule(static)` instructs the OpenMP runtime to divide the iteration space into almost equally sized chunks. Each thread sequentially computes $\bar{n}'/$`ncores` slice-matrix products. We decided to use this scheduling kind as all slice-matrix multiplications have the same number of floating-point operations with a regular workload where one can assume negligible load imbalance. Moreover, we wanted to prevent scheduling overheads for small slice-matrix products were data locality can be an important factor for achieving higher throughput.

We did not set the `OMP_PLACES` environment variable which defaults to the OpenMP `cores` setting defining a place as a single processor core. Together with the clause `num_threads(ncores)`, the number of OpenMP threads is equal to the number of OpenMP places, i.e. to the number of processor cores. We did not measure any performance improvements for a higher thread count.

The `proc_bind(spread)` clause additionally binds each OpenMP thread to one OpenMP place which lowers inter-node or inter-socket communication and improves local memory access. Moreover, with the `spread` thread affinity policy, consecutive OpenMP threads are spread across OpenMP places which can be beneficial if the user decides to set `hwthreads` smaller than the number of processor cores.

5. Experimental Setup

5.0.1. 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. 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.

We have used the GNU compiler v10.2 with 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 `mkl_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 tensor layout. Matrix $\mathbf{B}$ has the row-major storage format.

5.0.2. Tensor Shapes

We have used asymmetrically and symmetrically shaped tensors in order to cover many 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. A special feature of this test set is that the contraction dimension and the leading dimension are disproportionately large. 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 [12, 16].

6. Results and Discussion

6.1. Slicing Methods

This section analyzes the performance of the two proposed slicing methods `<slice>` and `<subtensor>` that have been discussed in section 4.4. Figure 1 contains eight performance contour plots of four `ttm` functions `<par-loop>` and `<par-gemm>` that either compute the slice-matrix product with subtensors `<subtensor>` or tensor slices `<slice>`. Each contour level within the plots represents an average Gflops/core value, averaged across tensor sizes.

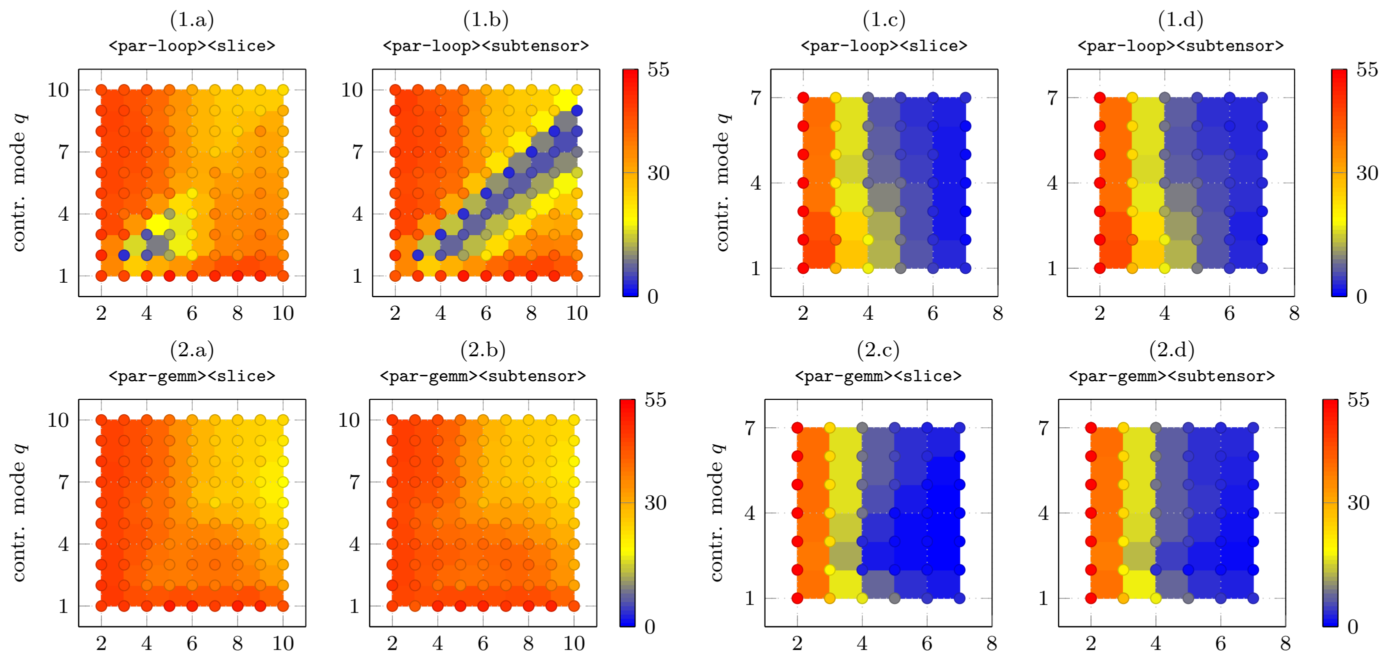

Figure 1: Performance contour plots in double-precision Gflops per core of the proposed TTM algorithms `<par-loop>` and `<par-gemm>` with varying tensor orders p and contraction modes q. The top row of maps (1.x) depict measurements of the `<par-loop>` versions while bottom row of maps with number (2.x) contain measurements of the `<par-gemm>` versions. Tensors are asymmetrically shaped on the left four maps (a,b) and symmetrically shaped on the right four maps (c,d). Tensor $\underline{\mathbf{A}}$ and $\underline{\mathbf{C}}$ have the first-order while matrix $\mathbf{B}$ has the row-major ordering. All functions have been measured on the Intel Xeon Gold 5318Y processor.

Moreover, each contour plot contains all applicable `gemm` cases listed in Table 1. The first column of performance values is generated by `gemm` belonging to case 3, except the first element which corresponds to case 2. The first row, excluding the first element, is generated by case 6 function. Case 7 is covered by the diagonal line of performance values when $q = p$. Although Figure 1 suggests that $q > p$ is possible, our profiling program sets $q = p$. Finally, case 8 with multiple `gemm` calls is represented by the triangular region which is defined by $1 < q < p$.

Function `<par-loop>` with `<slice>` averages with asymmetrically shaped tensors 26.22 Gflops/core (1.67 Tflops). With a maximum performance of 43.35 Gflops/core (2.77 Tflops), it performs on average 89.64% faster than function `<par-loop>` with subtensors. The slowdown with subtensors at $q = p-1$ or $q = p-2$ can be explained by the small loop count of the function that are in these cases 2 and 4, respectively. While function `<par-loop>` with tensor slices is affected by the tensor shapes for dimensions $p = 3$ and $p = 4$ as well, its performance improves with increasing order due to the increasing loop count.

Function `<par-loop>` with tensor slices achieves on average 13.01 Gflops/core (832.42 Gflops) with symmetrically shaped tensors. In this case, `<par-loop>` with subtensors achieves a mean throughput of 13.22 Gflops/core (846.16 Gflops) and is on average 9.89% faster than the `<slice>` version. The performances of both functions are monotonically decreasing with increasing tensor order, see plots (1.c) and (1.d) in Figure 1. The average performance decrease of both functions can be approximated by a cubic polynomial with the coefficients -35, 640, -3848 and 8011.

Function `<par-gemm>` with tensor slices averages 27.31 Gflops/core (1.74 Tflops) and achieves up to 43.43 Gflops/core (2.77 Tflops). With subtensors, function `<par-gemm>` exhibits almost identical performance characteristics and is on average only 3.42% slower than its counterpart with tensor slices. For symmetrically shaped tensors, `<par-gemm>` with subtensors and tensor slices achieve a mean throughput 11.98 Gflops/core (767.31 Gflops) and 11.57 Gflops/core (740.67 Gflops), respectively. However, function `<subtensor>` is on average 87.74% faster than the `slice` which is hardly visible due to small performance values around 5 Gflops/core or less whenever $q < p$ and the dimensions are smaller than 256. The speedup of function `<subtensor>` version can be explained by the smaller loop count and slice-matrix multiplications with larger tensor slices.

6.2. Parallelization Methods

This section discusses the performance results of the two parallelization methods `<par-gemm>` and `<par-loop>` using the same Figure 1.

With asymmetrically shaped tensors, both `<par-gemm>` functions with subtensors and tensor slices compute the tensor-matrix product with 27 Gflops/core and outperform function `<par-loop><subtensor>` version on average by a factor of 2.31. The speedup can be explained by the

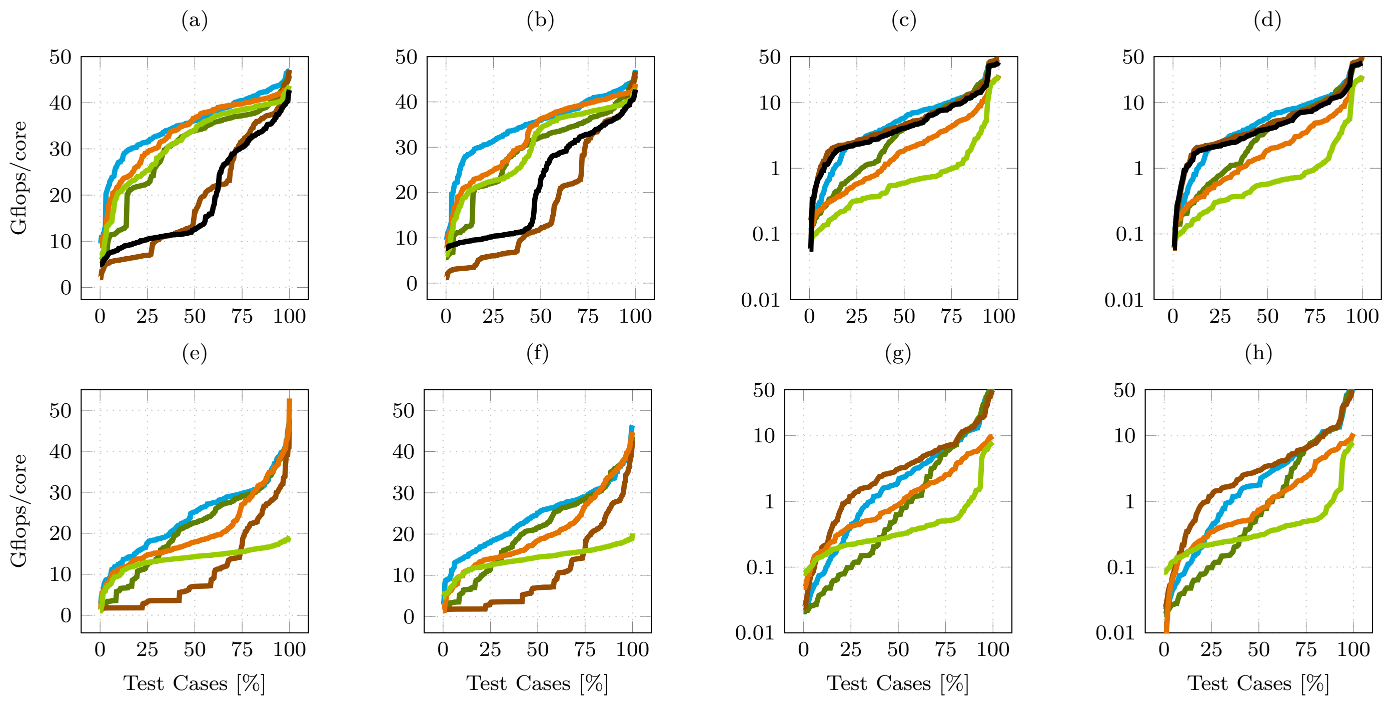

Figure 2: Cumulative performance distributions of the proposed algorithms for the eighth case. Each distribution line belongs to one algorithm: `<gemm_batch>` (▬) , `<par-gemm>` (▬) and `<par-loop>` (▬) using tensor slices, `<par-gemm>` (▬) and `<par-loop>` (▬) using subtensors. Tensors are asymmetrically (left plot) and symmetrically shaped (right plot).

performance drop of function `<par-loop><subtensor>` to 3.49 Gflops/core at $q = p - 1$ while both `<par-gemm>` functions operate around 39 Gflops/core. Function `<par-loop>` with tensor slices performs better for reasons explained in the previous subsection. It is on average 30.57% slower than its `<par-gemm>` version due to the aforementioned performance drops.

In case of symmetrically shaped tensors, `<par-loop>` with subtensors and tensor slices outperform their corresponding `<par-gemm>` counterparts by 23.3% and 32.9%, respectively. The speedup mostly occurs when $1 < q < p$ where the performance gain is a factor of 2.23. This performance behavior can be expected as the tensor slice sizes decreases for the eighth case with increasing tensor order causing the parallel slice-matrix multiplication to perform on smaller matrices. In contrast, `<par-loop>` can execute small single-threaded slice-matrix multiplications in parallel.

6.3. Case 8 with different Matrix Formats and Proessors

The contour plots in Fig. **??** contain performance data of all cases except for 4 and 5, see Table **??**. The effects of the presented slicing and parallelization methods can be better understood if performance data of only the eighth case is examined. Fig. 2 contains cumulative performance distributions of all the proposed algorithms which are generated `gemm` or `gemm_batch` calls within case 8. As the distribution is empirically computed, the probability y of a point (x, y) on a distribution function corresponds to the number of test cases of a particular algorithm that achieves x or less Tflops. For instance, function `<seq-loops,par-gemm>` with subtensors computes the tensor-matrix product for 50% percent of the test cases with equal to or less than 0.6 Tflops in case of asymmetrically shaped tensor. Consequently, distribution functions with an exponential growth are favorable while logarithmic behavior is less desirable. The test set cardinality for case 8 is 255 for asymmetrically shaped tensors and 91 for symmetrically ones.

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

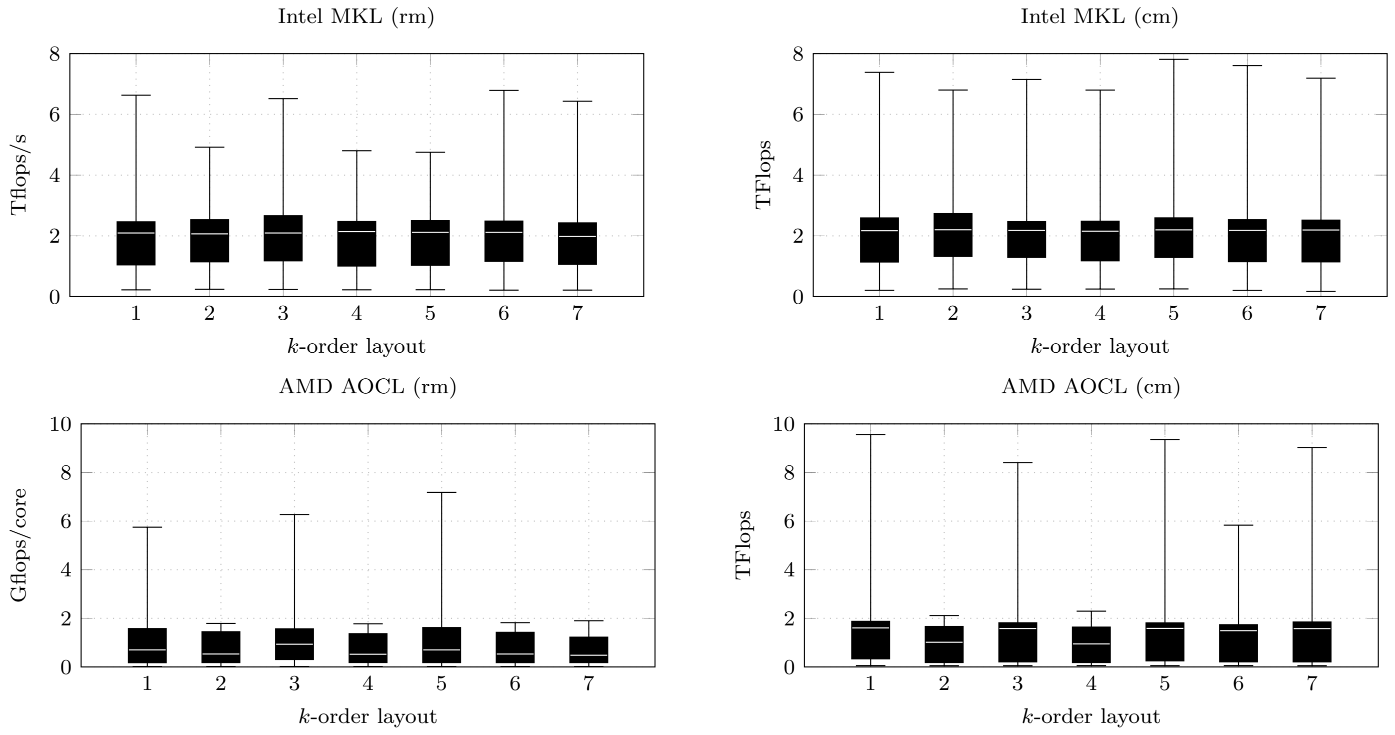

Figure 3: Box plots visualizing performance statics in double-precision Tflops of `<gemm_batch>` (left) and `<par-loop>` with subtensors (right). Box plot number k denotes the k-order tensor layout of symmetrically shaped tensors with order 7.

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 [10] 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.

6.4. Layout-Oblivious Algorithms

Fig. 3 contains two subfigures visualizing performance statics 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, for instance, are the first- and last-order storage formats for the order-7 tensor with $\pi_F = (1, 2, ..., 7)$ and $\pi_L = (7, 6, ..., 1)$. The definition of k-order tensor layouts can be found in section 3.3.

The low performance of around 70 Gflops can be attributed to the fact that the contraction dimension of subtensors of tensor slices of symmetrically shaped order-7 tensors are 8 while the leading dimension is 8 or at most 48 for subtensors. The relative standard deviation of `<gemm_batch>`'s and `<par-loops,seq-gemm>`'s median values are 12.95% and 17.61%. Their respective interquartile range are similar with a relative standard deviation of 22.25% and 15.23%.

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

6.5. Comparison with other Approaches

We have compared the best performing algorithm with four libraries that implement the tensor-matrix multiplication.

Library **tcl** implements the TTGT approach with a high-perform tensor-transpose library **hptt** which is discussed in [7]. **tblis** implements the GETT approach that is akin to Blis' algorithm design for the matrix multiplication [8]. The tensor extension of **eigen** (v3.3.7) is used by the Tensorflow framework. Library **libtorch** (v2.3.0) is the C++ distribution of PyTorch. **tlib** denotes our library using algorithm `<par-loops,seq-gemm>` that have been presented in the previous paragraphs.

Fig. 2 contains cumulative performance distributions for the complete test sets comparing the performance distribution of our implementation with the previously mentioned libraries. Note that we only have used tensor slices for asymmetrically shaped tensors (left plot) and subtensors for symmetrically shaped tensors (right plot). Our implementation with a median performance of 793.75 Gflops outperforms others' for almost every asymmetrically shaped tensor in the test set. The median performances of tcl, tblis, libtorch and eigen are 503.61, 415.33, 496.22 and 244.69

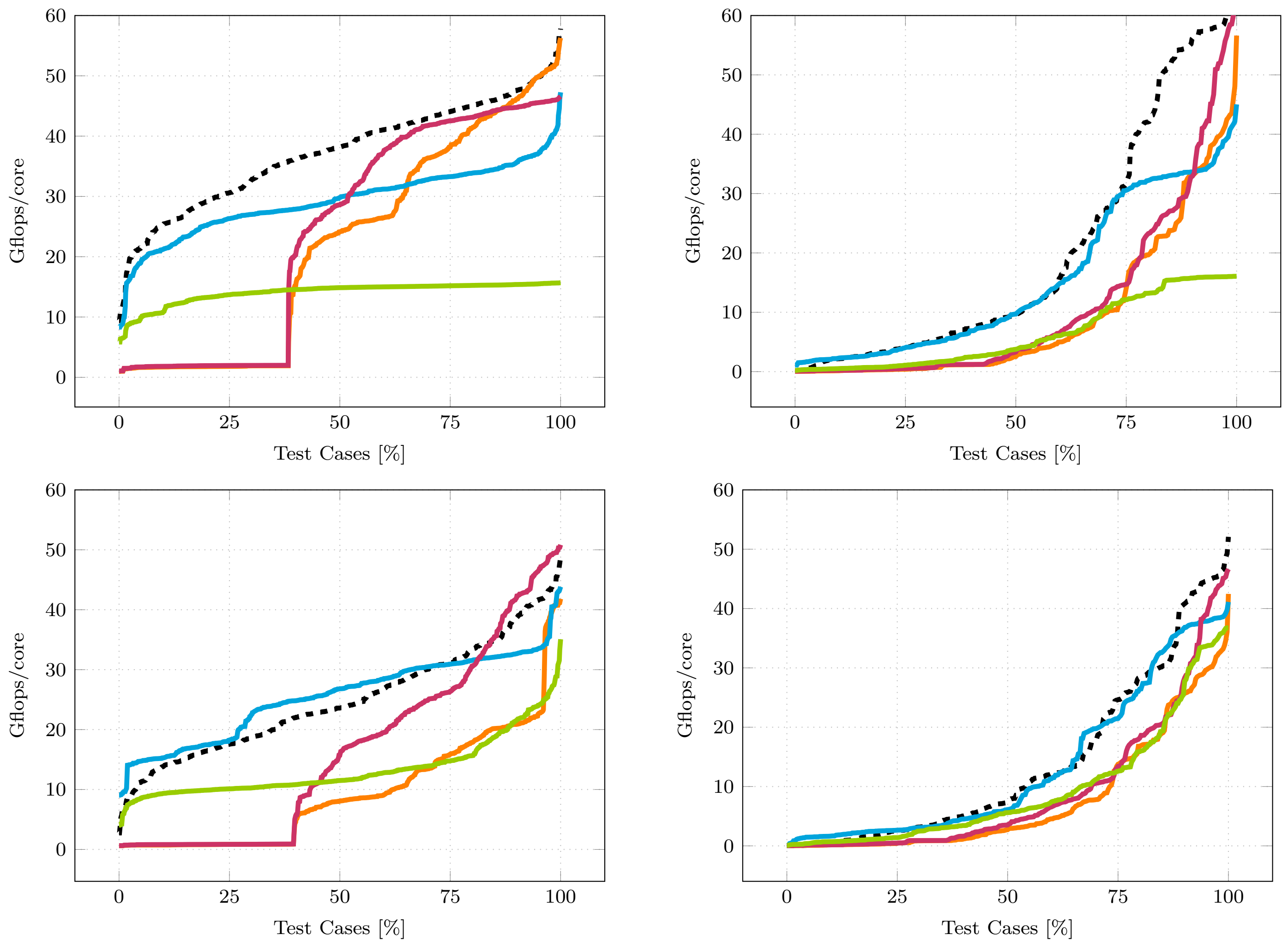

Figure 4: Cumulative performance distributions of tensor-times-matrix algorithms in double-precision Tflops. Each distribution line belongs to a library: **tlib**[ours] (▪ ▪ ▪), **tcl** (━━), **tblis** (━━), **libtorch** (━━), **eigen** (━━). Libraries have been tested with asymmetrically-shaped (left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

Gflops reaching on average 74.11%, 61.14%, 76.68% and 39.34% of tlib's throughputs.

In case of symmetrically shaped tensors the performance distributions of all libraries on the right plot in Fig. 2 are much closer. The median performances of tlib, tblis, libtorch and eigen are 228.93, 208.69, 76.46, 46.25 Gflops reaching on average 73.06%, 38.89%, 19.79% of tlib's throughputs[1]. All libraries operate with 801.68 or less Gflops for the cases 2 and 3 which is almost half of tlib's performance with 1579 Gflops. The median performance and the interquartile range of tblis and tlib for the cases 6 and 7 are almost the same. Their respective median Gflops are 255.23 and 263.94 for the sixth case and 121.17 and 144.27 for the seventh case. This explains the similar performance distributions when their performance is less than 400 Gflops. Libtorch and eigen compute the tensor-matrix product, in median, with 17.11 and 9.64 Gfops/s, respectively. Our library tlib has a median performance of 102.11 Gflops and outperforms tblis with 79.35 Gflops for the eighth case.

7. Conclusion and Future Work

We presented efficient layout-oblivious algorithms for the compute-bound tensor-matrix multiplication which is essential for many tensor methods. Our approach is based on the LOG-method and computes the tensor-matrix product in-place without transposing tensors. It applies the flexible approach described in [12] and generalizes the findings on tensor slicing in [10] for linear tensor layouts. The resulting algorithms are able to process dense tensors with arbitrary tensor order, dimensions and with any linear tensor layout all of which can be runtime variable.

Our benchmarks show that dividing the base algorithm into eight different GEMM cases improves the overall performance. We have demonstrated that algorithms with

[1]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.

parallel loops over single-threaded GEMM calls with tensor slices and subtensors perform best. Interestingly, they outperform a single batched GEMM with subtensors, on average, by 14% in case of asymmetrically shaped tensors and if tensor slices are used. Both version computes the tensor-matrix product on average faster than other state-of-the-art implementations. We have shown that our algorithms are layout-oblivious and do not need further refinement if the tensor layout is changed. We measured a relative standard deviation of 12.95% and 17.61% with symmetrically-shaped tensors for different k-order tensor layouts.

One can conclude that LOG-based tensor-times-matrix algorithms are on par or can even outperform TTGT-based and GETT-based implementations without loosing their flexibility. Hence, other actively developed libraries such as LibTorch and Eigen might benefit from implementing the proposed algorithms. Our header-only library provides `C++` interfaces and a python module which allows frameworks to easily integrate our library.

In the near future, we intend to incorporate our implementations in TensorLy, a widely-used framework for tensor computations [17, 18]. Currently, we lack a heuristic for selecting subtensor sizes and choosing the corresponding algorithm. Using the insights provided in [10] could help to further increase the performance. Additionally, we want to explore to what extend our approach can be applied for the general tensor contractions.

7.0.1. 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` and later.

References

[1] E. Karahan, P. A. Rojas-López, M. L. Bringas-Vega, P. A. Valdés-Hernández, P. A. Valdes-Sosa, Tensor analysis and fusion of multimodal brain images, Proceedings of the IEEE 103 (9) (2015) 1531–1559.

[2] E. E. Papalexakis, C. Faloutsos, N. D. Sidiropoulos, Tensors for data mining and data fusion: Models, applications, and scalable algorithms, ACM Transactions on Intelligent Systems and Technology (TIST) 8 (2) (2017) 16.

[3] N. Lee, A. Cichocki, Fundamental tensor operations for large-scale data analysis using tensor network formats, Multidimensional Systems and Signal Processing 29 (3) (2018) 921–960.

[4] T. G. Kolda, B. W. Bader, Tensor decompositions and applications, SIAM review 51 (3) (2009) 455–500.

[5] B. W. Bader, T. G. Kolda, Algorithm 862: Matlab tensor classes for fast algorithm prototyping, ACM Trans. Math. Softw. 32 (2006) 635–653.

[6] E. Solomonik, D. Matthews, J. Hammond, J. Demmel, Cyclops tensor framework: Reducing communication and eliminating load imbalance in massively parallel contractions, in: Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on, IEEE, 2013, pp. 813–824.

[7] P. Springer, P. Bientinesi, Design of a high-performance gemm-like tensor–tensor multiplication, ACM Transactions on Mathematical Software (TOMS) 44 (3) (2018) 28.

[8] D. A. Matthews, High-performance tensor contraction without transposition, SIAM Journal on Scientific Computing 40 (1) (2018) C1–C24.

[9] E. D. Napoli, D. Fabregat-Traver, G. Quintana-Ortí, P. Bientinesi, Towards an efficient use of the blas library for multilinear tensor contractions, Applied Mathematics and Computation 235 (2014) 454 – 468.

[10] J. Li, C. Battaglino, I. Perros, J. Sun, R. Vuduc, An input-adaptive and in-place approach to dense tensor-times-matrix multiply, in: High Performance Computing, Networking, Storage and Analysis, 2015, IEEE, 2015, pp. 1–12.

[11] Y. Shi, U. N. Niranjan, A. Anandkumar, C. Cecka, Tensor contractions with extended blas kernels on cpu and gpu, in: 2016 IEEE 23rd International Conference on High Performance Computing (HiPC), 2016, pp. 193–202.

[12] C. Bassoy, Design of a high-performance tensor-vector multiplication with blas, in: International Conference on Computational Science, Springer, 2019, pp. 32–45.

[13] F. Pawlowski, B. Uçar, A.-J. Yzelman, A multi-dimensional morton-ordered block storage for mode-oblivious tensor computations, Journal of Computational Science 33 (2019) 34–44.

[14] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al., Pytorch: An imperative style, high-performance deep learning library, Advances in neural information processing systems 32 (2019).

[15] L.-H. Lim, Tensors and hypermatrices, in: L. Hogben (Ed.), Handbook of Linear Algebra, 2nd Edition, Chapman and Hall, 2017.

[16] C. Bassoy, V. Schatz, Fast higher-order functions for tensor calculus with tensors and subtensors, in: International Conference on Computational Science, Springer, 2018, pp. 639–652.

[17] J. Cohen, C. Bassoy, L. Mitchell, Ttv in tensorly, Tensor Computations: Applications and Optimization (2022) 11.

[18] J. Kossaifi, Y. Panagakis, A. Anandkumar, M. Pantic, Tensorly: Tensor learning in python, Journal of Machine Learning Research 20 (26) (2019) 1–6.