SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Transpose-Free Contraction of Complex Tensors

Matthias Reumann

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Transpose-Free Contraction of Complex Tensors

Transpositionsfreie Kontraktion komplexer Tensoren

Author:	Matthias Reumann
Supervisor:	Prof. Dr. Christian Mendl
Advisor:	Manuel Geiger, M.Sc
Submission Date:	16.8.2023

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, 16.8.2023 Matthias Reumann

Acknowledgments

First and foremost, I'd like to thank my advisor Manuel for the invaluable feedback he provided and the advice he offered me for my future career.

Moreover, I'm eternally grateful for my friends and family, particularly Fabian, Patrick, and Sebastian, without whose support I would not have survived the past few years.

Abstract

Tensor Contraction (TC) is the operation that connects tensors in a Tensor Network (TN). Many scientific applications rely on efficient algorithms for the contraction of large tensors. In this thesis, we aim to develop a transposition-free TC algorithm for complex tensors. Our algorithm fuses high-performance General Matrix-Matrix Multiplication (GEMM), the 1M method for achieving complex with real-valued GEMM, and the Block-Scatter layout for tensors. Consequently, we give an elaborate overview of each. A benchmark for a series of contractions shows that our implementation can compete with the performance of state-of-the-art TC libraries.

Contents

1. Introduction

Due to the *Curse of dimensionality*, the simulation of quantum many-body systems scales exponentially in memory [8]. For example, systems as small as 50 qubits already require 2^{50} complex numbers, bringing current supercomputers to their limits. Hence, the necessity for alternative approaches. Tensor Networks (TN) pose an efficient and viable solution to this problem. A TN is a graph approximating a wavefunction, i.e., a n-qubit quantum system. In this graph, vertices depict tensors, and edges connect these tensors. The operation that connects tensors is called Contraction [2].

Besides the simulation of quantum systems, the technique also finds applications in quantum chemistry [18], string theory [20], and supervised machine learning [24]. Consequently, to fulfill the demands of those research areas, scalable and efficient algorithms are necessary, including algorithms for complex-valued TCs.

Aim. This thesis describes the implementation of a CPU-only transposition-free TC algorithm for complex tensors. The algorithm is based on an array of recent developments: The first is the GotoBLAS algorithm for GEMM [15] and its augmentation by the BLAS-like Library Instantiation Software (BLIS) framework [28]. The benefits of the latter include more opportunities for multithreading and better maintainability. A second is a novel approach to tensor contractions avoiding explicit transposition proposed by Matthews [19]. The third and last is Van Zee's 1M method for performing complex by real-valued matrix-matrix multiplication [27]. This modification improves the portability of the BLIS framework to other CPU architectures. In this work, we seek to fuse these approaches to achieve their combined benefits. The code of the algorithm is published on GitHub as the open-source `C++` library `t1m`.[1]

Structure. The work is structured as follows: In Chapter 2 we convey the required mathematical background. Based on this foundation we introduce the BLIS framework for dense linear algebra, the 1M method, and the Block-Scatter layout for tensors. Chapter 3 relates our work to others as well as discusses and compares alternative approaches. Chapter 4 outlines our implementation of tensor contraction for complex numbers. We report our findings in Chapter 5, as well as present our results for different tensor contraction problems. Lastly, we conclude in Chapter 6, provide an outlook and hint at future work.

[1] `https://github.com/MatthiasReumann/t1m`

2. Background

In this chapter, we provide basic theoretical knowledge to acquaint the reader with the necessary material and dive into the details of most state-of-the-art matrix-matrix multiplication implementations. Furthermore, we show how using these one can obtain two different TC algorithms.

2.1. Mathematical Foundation

This section establishes step-by-step the requisite theory and notation.

2.1.1. General Matrix-Matrix Multiplication (GEMM)

Most modern Basic Linear Algebra Subprograms (BLAS) interfaces adapt the matrix-matrix multiplication as expressed by Dongarra, Du Croz, Hammarling, and Duff [14]. Hence, we do the same. For examples see Goto and R. A. V. D. Geijn [15] and Van Zee and Van De Geijn [28].

Let $A \in \mathbb{C}^{m \times k}, B \in \mathbb{C}^{k \times n}$, and $C \in \mathbb{C}^{m \times n}$. Then, the operation is defined as

$$C = \alpha(A \cdot B) + \beta C \tag{2.1}$$

where $\alpha \in \mathbb{C}$ and $\beta \in \mathbb{C}$ are scalars. As in [19], we assume $\alpha = 1$ and $\beta = 0$ for brevity henceforward. Written element-wise, the equation looks like follows

$$C_{ij} = \sum_{p=0}^{k-1} A_{ip} \cdot B_{pj}. \tag{2.2}$$

The subscript specifies the row and column index, respectively. For example, C_{42} accesses the element in the fourth row and second column. The implicit summation over the same labeled indices yields the elegant Einstein notation:

$$C_{ij} = A_{ip} \cdot B_{pj} \tag{2.3}$$

.

2.1.2. Tensor Contraction

To generalize matrices to higher dimensions we introduce tensors. A Tensor $\mathcal{T}$ is a d-dimensional array, i.e. a 2-dimensional tensor is a matrix. We assign a pair $(l, N_l^{\mathcal{T}})$ to each of the $0, \ldots, d-1$ dimensions, where l is an alphabetic index and $N_l \in \mathbb{N}$ specifies the length of the dimension. We assume the index l to be alphabetic for convenience, however, other index types are also valid. For example, $A_{abc} \in \mathbb{C}^{3\times4\times2}$ is a tensor described by $(a,3)$, $(b,4)$, and $(c,2)$. Figure 2.1 depicts two tensors graphically. We denote the dimensionality of a tensor $\mathcal{T}$ as $d_{\mathcal{T}}$.

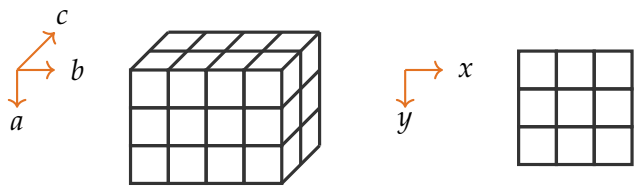


Figure 2.1.: Schematic view of the tensors $A_{abc} \in \mathbb{C}^{3\times4\times2}$ and $B_{yx} \in \mathbb{C}^{3\times3}$.

Following the line of thought that tensors generalize matrices, we define TC analogously to matrix-matrix multiplication. However, this time we sum over a bundle of indices P instead of a single index p as in Equation 2.2.

$$\mathcal{C}_{\pi_{\mathcal{C}}(I,J)} = \sum_{P} \mathcal{A}_{\pi_{\mathcal{A}}(I,P)} \cdot \mathcal{B}_{\pi_{\mathcal{B}}(P,J)} \tag{2.4}$$

where $\pi_{\mathcal{T}}$ is a bijective map $\{0, \ldots, d_{\mathcal{T}}-1\} \to \{0, \ldots, d_{\mathcal{T}}-1\}$ for each of the tensors due to the arbitrary ordering of tensor indices. I is the bundle of indices of $\mathcal{A}$ except those present in P and likewise for J and $\mathcal{B}$. Usually one refers to P as *bound* or *contracted* indices and I as well as J as *free* or *uncontracted* indices. The contraction size is defined as $\prod_{p\in P} N_p$.
Moreover, for $|P| = (d_{\mathcal{A}} + d_{\mathcal{B}} - d_{\mathcal{C}})/2$ the following conditions must hold.

1. $|I| = d_{\mathcal{A}} - |P|$
2. $|J| = d_{\mathcal{B}} - |P|$
3. $\forall p \in P . N_p^{\mathcal{A}} = N_p^{\mathcal{B}}$

As with matrix-matrix multiplication, we can further simplify Equation 2.4 by using the Einstein notation.

$$\mathcal{C}_{\pi_{\mathcal{C}}(I,J)} = \mathcal{A}_{\pi_{\mathcal{A}}(I,P)} \cdot \mathcal{B}_{\pi_{\mathcal{B}}(P,J)} \tag{2.5}$$

2.2. High-performance Matrix-Matrix-Multiplication

In theory, one could simply implement Equation 2.3 using three nested loops and yield correct results. However, this naive approach likely leads to sub-optimal performances due to poor cache utilization and redundant memory operations. Fortunately, this problem can be circumvented. Efficient matrix-matrix multiplication algorithms, such as in the GotoBLAS library and consequently in BLIS, implement an alternative approach and solve the just mentioned problems [15].

In the following sections, we outline the design of efficient matrix-matrix multiplication implementations and demonstrate how complex matrix-matrix multiplication can be achieved with real-valued algorithms.

2.2.1. BLIS Framework

The key insight of Goto et al. was to pack the matrices A and B into smaller blocks $\tilde{A} \in \mathbb{C}^{m_C \times k_C}$, and $\tilde{B} \in \mathbb{C}^{k_C \times n_C}$ such that $\tilde{B}$ resides in L3 and $\tilde{A}$ in L2 cache for spatial locality. The cache-blocking parameters m_C, k_C, and n_C are chosen for specific CPU architectures to fulfill this requirement. A highly-optimized macro kernel performs the matrix-matrix multiplication $\tilde{A}_{ip} \cdot \tilde{B}_{pj}$ and outputs the resulting $m_C \times n_C$ submatrix at the correct location for the C matrix [15].

However, the GotoBLAS approach comes with some shortcomings. One is that GotoBLAS necessarily requires column-major ordering. A complete list of the deficiencies described in detail can be found in Van Zee and Van De Geijn [28].

Row- and column-major ordering are techniques for storing elements of a many-dimensional array in linear contiguous memory. Column-major addresses elements in increasing co-lexicographic order, while row-major specifies them lexicographically. A combination of both is called general-stride ordering. For column-major ordering, the memory offset of a d-dimensional $N_0 \times N_1 \cdots \times N_d$ array, indexed by a tuple $I = (i_0, i_1, \ldots, i_d)$ is given by Equation 2.6. Figure 2.2 illustrates the column-major ordering for a three-dimensional array.

$$\sum_{k=0}^{d} \left(\prod_{l=0}^{k-1} N_l \right) \cdot i_k = i_0 + N_0 \cdot (i_1 + N_1 \cdot (i_2 + N_2 \cdot (\ldots + N_{d-1} i_d))) \tag{2.6}$$

The BLIS framework extends the linear algebra operations such that row- and column-major, as well as general stride ordering can be used. Furthermore, not only can the ordering be different from one operation to the next, but also for a single one. For example, in the matrix-matrix multiplication $C = A \cdot B$, C can be row-stored, A column-stored, and B with general stride ordering [28].

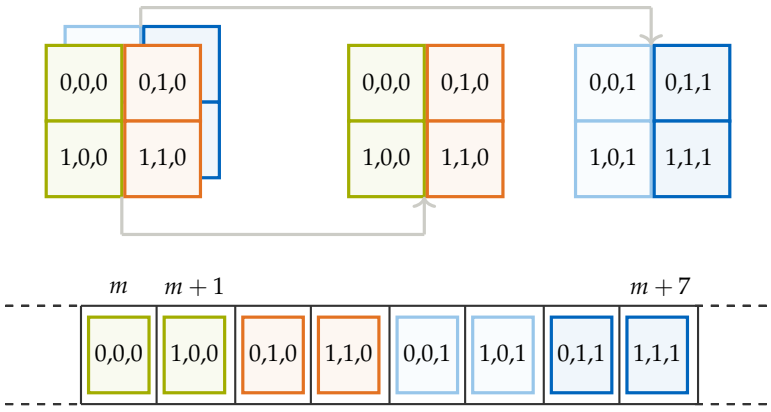


Figure 2.2.: Column-Major ordering for a three-dimensional array in linear memory.

Another key difference between GotoBLAS and BLIS is that the latter further partitions $\tilde{A}$ and $\tilde{B}$ into *slivers* of size $m_R \times k_C$, and $k_C \times n_R$ respectively. The register-blocking size n_R is chosen such that a sliver fits into the L1 cache.

Moreover, BLIS replaces the macro kernel with two loops in a high-level programming language and a small but highly-optimized microkernel written in Assembly. The reason behind this design decision is to move secondary functionality such as edge case handling outside the macro kernel. As a result, the remaining microkernel is easier to maintain and port to different architectures. Nonetheless, since the release of the framework, some edge case handling was again moved inside the microkernel [5]. The authors show that the performance of BLIS is comparable with the open-source libraries OpenBLAS (the successor of the GotoBLAS library), the ATLAS library as well as Intel's MKL library [28]. Performance graphs for a multitude of CPU architectures can be found on their GitHub Repository [7].

Opportunities for multithreading are explored in Smith, R. V. D. Geijn, Smelyanskiy, et al. [21]. At the time of writing, BLIS supports OpenMP and the POSIX thread library [6]. Furthermore, the exposure of its microkernels via a well-documented API promotes code reuse and accelerates the development of high-performance linear algebra software [5]. Optimized microkernel implementations exist for AMD, Intel, and IBM, as well as various ARM architectures [4].

A schematic overview of the approach can be found in Figure 2.3. The illustration intuitively portrays the *loops around the microkernel* as well as the packing of blocks and slivers. In succeeding sections and chapters, we sometimes also refer to it as the *5-Loop Approach*. Algorithm 1 alternatively depicts the approach in pseudocode.

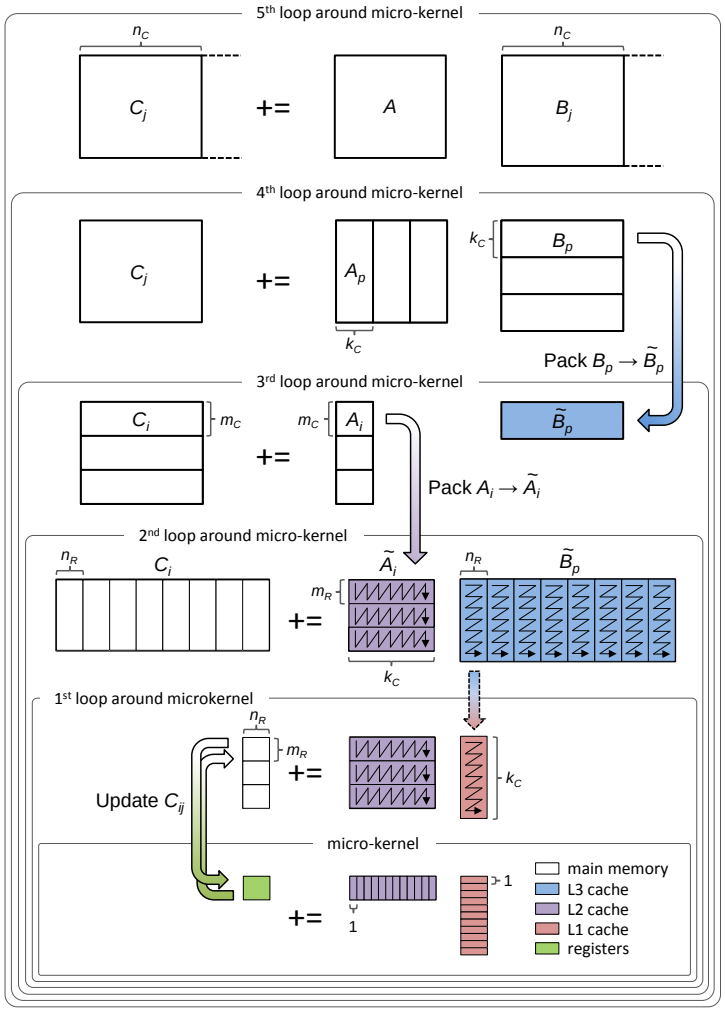


Figure 2.3.: Schematic of BLIS' 5-Loop Approach to Matrix-Matrix Multiplication. Taken from [6] and used with permission by the authors.

Algorithm 1 5-Loop Approach for GEMM

```
1: for j_c := 0, j_c < N, j_c += NC do
2:     for p_c := 0, p_c < K, p_c += KC do
3:         B̃ = pack_B(B, p_c, j_c)
4:         for i_c := 0, i_c < M, i_c += MC do
5:             Ã = pack_A(A, i_c, p_c)
6:             for j_r := 0, j_r < NC, j_r += NR do
7:                 J = j_c + j_r
8:                 for i_r := 0, i_r < MC, i_r += MR do
9:                     I = i_c + i_r
10:                    C[I:I+MR, J:J+NR] = microkernel(Ã, B̃, I, J)
```

2.2.2. 1M Method

The 1M Method by Van Zee is a technique to achieve complex matrix-matrix multiplication by a real-value one [27]. The method has been integrated into BLIS and further strengthens the portability of the framework. This claim can easily be justified by stating that developers must only implement real-value (i.e. `float` and `double`) microkernels by using the 1M Method, therefore, effectively halving the necessary effort. Moreover, any improvement in the performance of real-domain microkernels would immediately lead to benefits in the complex domain.

The multiplication of two complex numbers a and b is generally defined as in Equation 2.7. In the following, superscripts r and i denote the real and imaginary parts of a complex number.

$$\begin{aligned} c^r &= a^r b^r - a^i b^i \\ c^i &= a^r b^i + a^i b^r \end{aligned} \tag{2.7}$$

The 1M method is based on the observation that complex multiplication can be expressed as real-valued matrix-vector multiplication (Equation 2.8).

$$\begin{pmatrix} c^r \\ c^i \end{pmatrix} = \begin{pmatrix} a^r & -a^i \\ a^i & a^r \end{pmatrix} \begin{pmatrix} b^r \\ b^i \end{pmatrix} \tag{2.8}$$

Suppose $A \in \mathbb{C}^{m \times k}$, $B \in \mathbb{C}^{k \times n}$, $C \in \mathbb{C}^{m \times n}$, with $m = 3$, $k = 2$, and $n = 2$. Furthermore, we wish to compute $C = A \times B$. One can easily conclude by inspection that Equation 2.10 equals Equation 2.9 if the output C is stored column-major.

$$\begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \\ C_{20} & C_{21} \end{pmatrix} = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \\ A_{20} & A_{21} \end{pmatrix} \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix} \tag{2.9}$$

$$\begin{pmatrix} C^r_{00} & C^r_{01} \\ C^i_{00} & C^i_{01} \\ C^r_{10} & C^r_{11} \\ C^i_{10} & C^i_{11} \\ C^r_{20} & C^r_{21} \\ C^i_{20} & C^i_{21} \end{pmatrix} = \begin{pmatrix} A^r_{00} & -A^i_{00} & A^r_{01} & -A^i_{01} \\ A^i_{00} & A^r_{00} & A^i_{01} & A^r_{01} \\ A^r_{10} & -A^i_{10} & A^r_{11} & -A^i_{11} \\ A^i_{10} & A^r_{10} & A^i_{11} & A^r_{11} \\ A^r_{20} & -A^i_{20} & A^r_{21} & -A^i_{21} \\ A^i_{20} & A^r_{20} & A^i_{21} & A^r_{21} \end{pmatrix} \begin{pmatrix} B^r_{00} & B^r_{01} \\ B^i_{00} & B^i_{01} \\ B^r_{10} & B^r_{11} \\ B^i_{10} & B^i_{11} \end{pmatrix} \tag{2.10}$$

Note that when we refer to the 1M Format we explicitly refer to the 1M_C format, i.e. the output matrix is stored by columns. An alternative definition of the 1M format using row-major ordering can be found in the original paper [27].

Consequently, we observe that the real-value matrix-matrix multiplication in Equation 2.10 doubles the dimension lengths m and k and thus $\hat{A} \in \mathbb{R}^{2m \times 2k}$, $\hat{B} \in \mathbb{R}^{2k \times n}$, $\hat{C} \in \mathbb{R}^{2m \times n}$. Moreover, we remark that the authors claim that the 1M method exhibits numerical properties similar to the conventional assembly implementation. However, no formal analysis of the numerical stability of the algorithm has been provided [27].

In Equation 2.1 we include the factors α and β for the definition of matrix-matrix multiplication. Since the 1M Method uses real microkernels and these scalars are generally complex, we can not directly use them as arguments for the microkernel. The authors propose the following solution to the problem [27]. If the imaginary part of either one or both factors is zero pass the real part as argument to the microkernel. However, if $\beta^i \neq 0$ scale each element of the resulting matrix C, albeit considerable overhead. When $\alpha^i \neq 0$ scale elements while packing sub matrices of A or B.

We conclude that if complex elements are packed into memory accordingly, complex matrix-matrix multiplication can be achieved with real-valued algorithms.

2.3. High-performance Tensor Contraction

Extending the argument for GEMM algorithms, implementing TC using $|P|$ nested loops is likely inefficient. Hence, alternative approaches are necessary. In this section, we describe two such alternatives, namely Transpose-Transpose-GEMM-Transpose (TTGT) and Block-Scatter-Matrix Tensor Contraction (BSMTC). Both use that under certain conditions tensors are equivalent to matrices in memory.

2.3.1. Tensor Contraction as Matrix-Matrix Multiplication

Equation 2.3 and 2.5 hint at the similarities between matrix-matrix multiplication and TC. In fact, under certain conditions, tensors are *structurally equivalent* to matrices in memory [19]. If elements of a tensor $\mathcal{T}_{\pi_\mathcal{T}(QR)}$ are stored by columns and $\pi_\mathcal{T}$ is the identity map, the index sets Q and R both collapse into a range of single continuous indices $\bar{Q}$ and $\bar{R}$.

Let $\mathcal{A}_{abc} \in \mathbb{C}^{4\times4\times3}$. Suppose $I = (a, b)$, $P = (c)$, and $\pi_\mathcal{A}$ maps each index to itself. Hence, $\mathcal{A}_{\pi_\mathcal{A}(IP)} = \mathcal{A}_{IP}$. Since we assume that $\mathcal{A}$ is stored column-major, the location of individual elements given by Equation 2.6 is

$$loc(\mathcal{A}_{abc}) = a + b \cdot N_a + c \cdot N_a N_b \tag{2.11}$$

Notice that for I the range of values of (a, b) is $0 \leq \bar{I} < N_a N_b = 16$ and $0 \leq \bar{P} < N_c = 3$ for P. Consequently, the tensor $\mathcal{A}_{IP}$ becomes the structural equivalent matrix $\tilde{\mathcal{A}}_{\bar{I}\bar{P}}$. Then, if these conditions hold for the tensors $\mathcal{C}$, $\mathcal{A}$, and $\mathcal{B}$, TC is *functionally equivalent* to matrix-matrix multiplication for the *sequentially continuous* indices $\bar{I}$,$\bar{J}$, and $\bar{P}$ [19].

$$\tilde{\mathcal{C}}_{\bar{I}\bar{J}} = \tilde{\mathcal{A}}_{\bar{I}\bar{P}} \cdot \tilde{\mathcal{B}}_{\bar{P}\bar{J}} \tag{2.12}$$

2.3.2. TTGT

Unfortunately, often tensors are not stored in memory as described in the previous section. A possible solution to this problem is to transpose each of the tensors $\mathcal{A}$ and $\mathcal{B}$ such that each is structurally equivalent to a matrix. Then, perform a matrix-matrix multiplication and transpose the resulting matrix `C` into its tensor layout $\mathcal{C}$. Algorithm 2 describes the TTGT procedure concisely.

Algorithm 2 TTGT Approach

```
1: function TTGT(𝒜, ℬ, 𝒞, permA, permB, permC)
2:     A = permute(𝒜, permA)
3:     B = permute(ℬ, permB)
4:     C = gemm(A, B)
5:     𝒞 = permute(C, permC)
```

However, the transposition of each tensor comes with a cost in memory. Since full temporary copies of $\mathcal{A}, \mathcal{B}, \mathcal{C}$ are needed, memory workspace increases by a factor of two for the TTGT approach. Furthermore, the percentage of total time spent on transpositions can be as high as 50% [19].

2.3.3. BSMTC

An alternative approach was proposed by Matthews [19]. The key principle: Utilize the BLIS framework to pack small submatrices of structurally equivalent tensors directly without transposition.

To map a tensor $\mathcal{A}$ to its structurally equivalent matrix $\tilde{\mathcal{A}}_{\hat{I}\hat{P}}$ Matthews introduces *scatter vectors*. Given the index bundles I and P and let P be the logical x axis. Then, specify the row and column scatter vectors $rscat_{\bar{I}}(\mathcal{A})$ and $cscat_{\bar{P}}(\mathcal{A})$ as follows

$$\begin{aligned} rscat_{\bar{I}}(\mathcal{A}) &= i_0 \cdot s_{i_0}(\mathcal{A}) + i_1 \cdot s_{i_1}(\mathcal{A}) + \cdots + i_{|I|-1} \cdot s_{i_{|I|-1}}(\mathcal{A}), \\ cscat_{\bar{P}}(\mathcal{A}) &= p_0 \cdot s_{p_0}(\mathcal{A}) + p_1 \cdot s_{p_1}(\mathcal{A}) + \cdots + p_{|P|-1} \cdot s_{p_{|P|-1}}(\mathcal{T}). \end{aligned} \tag{2.13}$$

where $s_i(\mathcal{A})$ is the stride of the index i for tensor $\mathcal{A}$. The location of elements of a tensor as if it were a matrix is then given by

$$loc(\tilde{\mathcal{A}}_{\bar{I}\bar{P}}) = rscat_{\bar{I}}(\mathcal{A}) + cscat_{\bar{P}}(\mathcal{A}). \tag{2.14}$$

Figure 2.4 illustrates the scatter vectors for a tensor $\mathcal{A}_{bacd}$ with index bundles $I = (b, a)$ and $P = (c, d)$. The numbers in the grid cells depict the location of elements for the structural equivalent matrix $\tilde{\mathcal{A}}$ in contiguous memory. For example, the offset 35 in the bottom right corner is given by $loc(\tilde{A}_{5,5}) = rscat_5(\mathcal{T}) + cscat_5(\mathcal{T}) = 5 + 30$. Moreover, one can notice by inspection that the difference in the values for both scatter vectors is constant. In our example, this holds for all the values of both vectors. However, since it's unlikely that this is true for bigger tensors, we partition each scatter vector of length l into blocks of length b and calculate the difference in these blocks. Therefore giving rise to the *Block-Scatter* vectors *rbs* and *cbs* for *rscat* and *cscat*, respectively. The i-th entry denotes either a constant stride for a block of length b or 0 otherwise.

$$bs_i(\mathcal{T}) = \begin{cases} scat_j(\mathcal{T}) - scat_{j-1}(\mathcal{T}) = s & \text{If s is const for all } i \cdot b < j < \min(l, (i+1) \cdot b) \\ 0 & \text{Otherwise} \end{cases} \tag{2.15}$$

Note that the length of the *bs* vector is $\lceil \frac{l}{b} \rceil$. Furthermore, we call the parameter b a *blocking parameter*. Now, if for a given index i both $rbs_i(\mathcal{T}) > 0$ and $cbs_i(\mathcal{T}) > 0$, the elements in this block can be accessed more efficiently with constant row and column strides. In Figure 2.4 the highlighted 3×3 square is an example of an efficiently accessible block. In this block, each element can be accessed in code by `A[r*1 + c*6]`, where 1 is the row and 6 is the column stride. Consequently, this allows vectorization on modern CPUs.

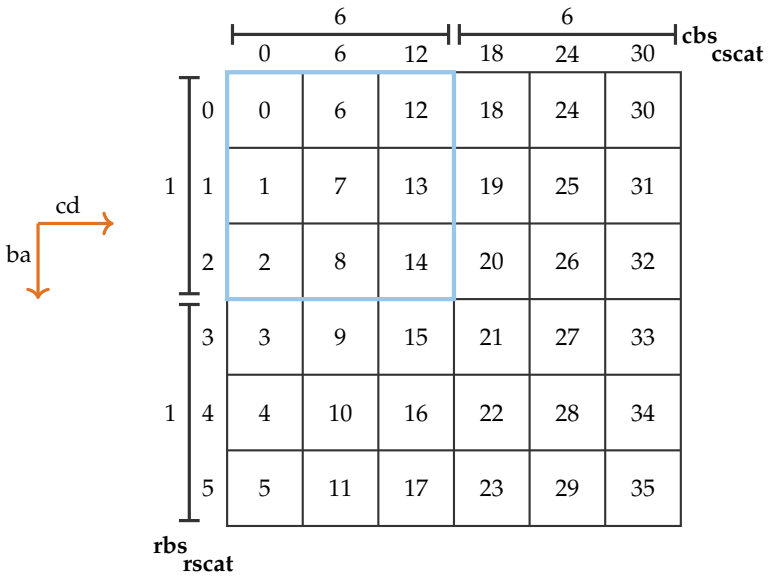


Figure 2.4.: Schematic representation of the scatter and block-scatter layout for the Tensor $\mathcal{A}_{bacd} \in \mathbb{C}^{3\times2\times2\times3}$ stored by columns. $I = (b, a)$, $P = (c, d)$. The blocking size is 3 for each axis.

Finally, we present the transposition-free tensor contraction algorithm BSMTC as implemented by the TBLIS library [19]. As eluded to at the beginning of this section, TBLIS bases its implementation on BLIS's 5-Loop Approach for GEMM. The transformation of the GEMM algorithm into a TC requires numerous changes.

First, rewrite the packing functions `pack_A` and `pack_B` of Algorithm 1 such that the Block-Scatter Layout is used. Thus, calculate the scatter vectors $rscat_{\bar{I}}$, $cscat_{\bar{P}}$ for $\mathcal{A}$, $rscat_{\bar{P}}$, $cscat_{\bar{J}}$ for $\mathcal{B}$, and $rscat_{\bar{I}}$, $cscat_{\bar{J}}$ for $\mathcal{C}$. Due to the integration into the BLIS framework, the blocking parameters can be derived from BLIS smallest partition unit: the sliver. For the $\mathcal{C}$ tensor we simply use m_R and n_R as blocking parameters for the row and column Block-Scatter vectors. For the $\mathcal{A}$ tensor it is theoretically possible to use m_R and k_C as blocking parameters. However, since k_C is usually large (256 for Intel's Haswell architecture for double precision) Matthews proposes to further divide k_C into $k_P \approx m_R$ chunks [19]. Therefore, effectively making m_R and k_P the blocking parameters for $\mathcal{A}$. Likewise, we define the blocking parameters for $\mathcal{B}$ as k_P and n_R.

Once both $\mathcal{A}$ and $\mathcal{B}$ are packed into $\tilde{A}$ and $\tilde{B}$ in their respective BLIS format, invoke the microkernel. BLIS' microkernels assume a constant stride for the output submatrix $\tilde{C}$. Given the strides $rs = rbs(\mathcal{C})$ and $cs = cbs(\mathcal{C})$, differentiate between two cases. If $rs > 0$ and $cs > 0$ use these strides and a correctly offset pointer to $\mathcal{C}$ as arguments for

the microkernel. If not, write the result of the microkernel invocation to a temporary buffer first and then unpack it using the Scatter-Layout.

Concluding, the combination of the Block-Scatter Layout and BLIS' 5-Loop Approach yields a transposition-free TC algorithm.

3. Related Work

Recent work circumvents the problem of memory expensive transpositions. Most notably, TBLIS from which this thesis draws heavy inspiration [19]. Unfortunately, at the time of writing, TBLIS does not include BLIS's optimized microkernel for complex numbers. Instead, matrix-matrix multiplication is implemented by three nested loops in `C++`. Compared to the performance for real-valued contractions, the library suffers from this insufficiency for complex numbers. For this thesis, we adapt the TBLIS approach and apply it to complex tensor contractions.

Furthermore, Huang et al. extend TBLIS by using Strassen's matrix multiplication and show promising results [17]. For the sake of comparison to the original TBLIS, we do not implement Strassen's algorithm.

In [22], Springer et al. presents another novel transposition-free approach, namely *GEMM-like Tensor-Tensor multiplication (GETT)*. GETT and TBLIS share the same core principle: Utilize efficient matrix-matrix multiplication kernels to perform tensor contraction by reformalizing sub-matrix-packing.

On the downside, GETT is not a standalone library but part of the *Tensor Contraction Code Generator (TCCG)*. TCCG combines multiple methods of tensor contraction and chooses one of those based on heuristics and previous contractions at compile time. This design decision makes it cumbersome to apply the approach in practice. Firstly, one needs to specify the contraction in text form with an additional constraint that the free indices of each input tensor need to be a multiple of 24 [16]. Secondly, two compilation steps are necessary since the code needs to be generated and compiled. Moreover, the TCCG is written in `Python 2`, an unsupported version of `Python` since 2020 [25].

More traditionally, the MATLAB Tensor Toolbox by Bader et al. and similarly the Tensor Contraction Library (TCL) by Springer et al. implement the TTGT method [1]. TCL, which is also part of the TCCG, uses the High-Performance Tensor Transpose (HPTT) library for transpositions [23]. Nonetheless, due to the inherent nature of TTGT, both the Tensor Toolbox and TCL require double the amount of temporary memory. Here we use these libraries as reference implementations.

Lastly, we hint at the current development of GPU tensor libraries such as cuTENSOR [10]. Since our implementation is purely CPU-based, we do not investigate this avenue any further.

4. Implementation

The purpose of this chapter is to outline the implementational details of our complex tensor contraction algorithm using real-valued matrix-matrix multiplication. We provide the reader with a top-down description of our approach, stepping down one abstraction at a time, i.e. from the interface to the core of our algorithm.

4.1. Interface

Listing 4.1 presents the `C++` interface for our tensor contraction library. Besides the ones listed, overloaded functions for real-valued contraction also exist.

```
1  void contract(Tensor<std::complex<float>> A, std::string labelsA,
2                Tensor<std::complex<float>> B, std::string labelsB,
3                Tensor<std::complex<float>> C, std::string labelsC);
4
5  void contract(Tensor<std::complex<double>> A, std::string labelsA,
6                Tensor<std::complex<double>> B, std::string labelsB,
7                Tensor<std::complex<double>> C, std::string labelsC);
```

Listing 4.1: Outgoing `C++` interface for complex tensor contractions as part of our `t1m` library.

The `Tensor` class conveniently wraps marray, a flexible multidimensional array library [12]. Marray provides many useful utilities such as stride calculation for ease of development. The interested reader is referred to the documentation [11]. For example, the Tensor $\mathcal{A} \in \mathbb{C}^{3\times4\times5}$ is initialized with `Tensor<std::complex<float>>({3, 4, 5}, ptrA)`, where `ptrA` is a `std::complex<float>>` pointer to a complex array of size $3 \cdot 4 \cdot 5$.

4.2. Preparation

Before we execute the loops described in Chapter 2, a handful of preliminary steps are necessary. In particular, we wrap the provided `Tensor` objects into `BlockScatterMatrix` objects which calculate Scatter and Block-Scatter vectors according to Equations 2.13

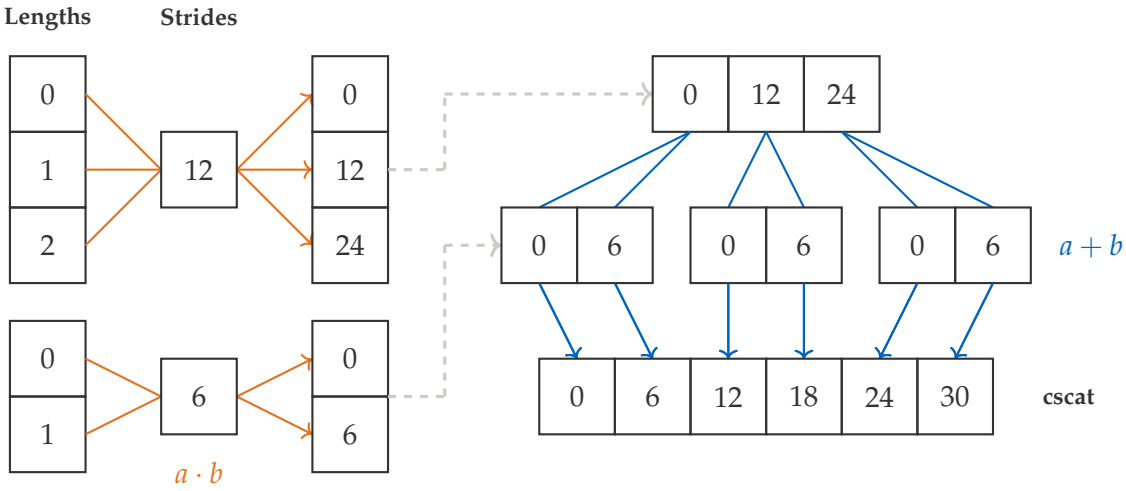


Figure 4.1.: Graphical representation of our algorithm for the calculation of the *cscat* vector for $\mathcal{A}_{bacd} \in \mathbb{C}^{3\times2\times2\times3}$ and the index bundle $P = (c, d)$.

and 2.15. Algorithmically, the Scatter vectors can be calculated by traversing a tree-like structure and collecting the results in the end. Figure 4.1 depicts our algorithm for the calculation of Scatter vectors. Given the Scatter, we simply calculate associated Block-Scatter vectors with nested loops in `C++` code.

The TBLIS library delays the transition to the Block-Scatter layout until all input and output tensors have been partitioned into fixed-sized blocks [19]. The rationale behind this decision includes the provision of precomputed scatter vectors and unwanted `malloc` calls. Even though worthwhile considerations, since we neither allow the first nor found significant evidence for the second argument, we went for the simple route of allocating the Scatter and Block-Scatter vectors at the start of the tensor contraction.

Then, we initialize a `gemm_context` object which holds all relevant data for the GEMM-like TC such as the `BlockScatterMatrix` objects for $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$, cache and register blocking parameters, as well as a function pointer to the respective BLIS microkernel. BLIS's `bli_cntx_get_l3_sup_blksz_def_dt` function provides the blocking sizes `MC`, `KC` etc. for the underlying hardware, given a constant such as `BLIS_DOUBLE`. For a full list of parameter types see [3].

Listing 4.2 shows the `gemm_context` structure in `C++` code. The template type `U` specifies the floating point type of the complex type of `T`. For example, if `T` equals `std::complex<double>`, `U` would be `double`. This separation is necessary for the usage of real microkernels for the 1M Method. The combination of `C++` variadic templates and a context object allows us to overload the function executing the five loops around

the microkernel, therefore improving code reuse and readability.

Lastly, we allocate $\tilde{A} \in \Theta(m_C \times k_C)$, $\tilde{B} \in \Theta(k_C \times n_C)$, and $\tilde{C} \in \Theta(m_R \times n_R)$ additional real-valued workspace for the packing operations described in Section 4.4.

```
1  template<typename T, typename U>
2  struct gemm_context
3  {
4    /* BLIS-specific context object */
5    const cntx_t *cntx;
6    /* Cache and register blocking sizes */
7    const dim_t NC; const dim_t KC; const dim_t MC;
8    const dim_t NR; const dim_t MR; const dim_t KP;
9    /* Tensors */
10   BlockScatterMatrix<T> *A;
11   BlockScatterMatrix<T> *B;
12   BlockScatterMatrix<T> *C;
13   /* Scalars */
14   U *alpha; U *beta;
15   /* Function pointer to BLIS microkernel */
16   void (*kernel);
17 };
```

Listing 4.2: Context object for the GEMM-like TC of complex numbers.

4.3. Loops around Micro-kernel

After all the preparations have been made, we apply the 5-loop approach as described in Section 2.2.1. However, the combination of BLIS, the Block-Scatter layout, and the 1M method require the following adjustments.

1. Halve the step sizes `KC` and `MC` such that after packing, $\tilde{B}$ has size `KC` $\cdot$ `NC` and $\tilde{A}$ has size `MC` $\cdot$ `KC`.
2. Pack blocks of the tensors $\mathcal{B}$ and $\mathcal{A}$ in their respective 1M format using the Block-Scatter Layout (See Section 4.4).
3. Invoke a real microkernel and unpack the result from $\tilde{C}$ to the actual location in memory using the Block-Scatter Layout.

A comparison between Algorithm 1 and Algorithm 3 illustrates the changes succinctly. Note that Algorithm 3 depicts a simplified version, i.e., a version without edge case handling, for educational purposes. Moreover, the `rmicrokernel` function call refers to the kernels described in Section 4.5.

Algorithm 3 1M GEMM for TC

```
1: for j_c := 0, j_c < N, j_c += NC do
2:     for p_c := 0, p_c < K, p_c += KC/2 do
3:         B̃ = pack_B(B, p_c, j_c)
4:         for i_c := 0, i_c < M, i_c += MC/2 do
5:             Ã = pack_A(A, i_c, p_c)
6:             for j_r := 0, j_r < NC, j_r += NR do
7:                 J = j_c + j_r
8:                 for i_r := 0, i_r < MC, i_r += MR do
9:                     I = i_c + (i_r/2)                    ▷ /2 due to 1M Format
10:                    C̃[I:I+MR, J:J+NR] = rmicrokernel(Ã, B̃, I, J)
11:                    unpack(C̃[I:I+MR, J:J+NR], C, I, J)
```

4.4. Packing

We use the Block-Scatter layout of a tensor to address individual complex elements as if the tensor was a matrix. If possible, access elements continuously with constant row and column strides provided by the Block-Scatter vectors. We hint the compiler to vectorize these operations by `#pragma omp simd` directives. Otherwise, use the less efficient Scatter layout.

Even though we access complex elements, we pack them as real-valued. Figures 4.2 and 4.3 depict how we store individual complex numbers as real numbers for the 1M method. Note that one complex takes 4 real-valued numbers for the tensor $\mathcal{A}$, and 2 for the tensor $\mathcal{B}$. Adjustment 1 in Section 4.3 stems from these facts.
Then, pack a $m_C \times k_C$ submatrix of $\frac{m_C}{2} \cdot \frac{k_C}{2}$ complex numbers for $\mathcal{A}$ and a $k_C \times n_R$ submatrix of $n_R \cdot \frac{k_C}{2}$ complex numbers for $\mathcal{B}$. Divide these blocks into even smaller $m_R \times k_C$ and $k_C \times n_R$ slivers, respectively. Notice that we store slivers for $\mathcal{A}$ column-major and slivers for $\mathcal{B}$ row-major. We additionally iterate the length k_C in k_P chunks, as described in Section 2.3.3, to utilize block-scattering for the K dimension. Figure 4.4 illustrates this mechanism schematically.

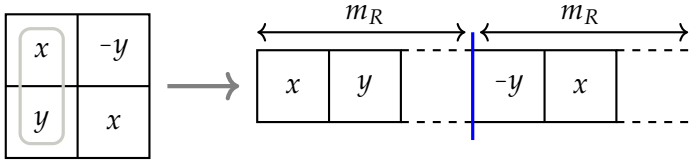


Figure 4.2.: Packing of a single complex number $x + yi$ for the tensor $\mathcal{A}$ in the 1M format.

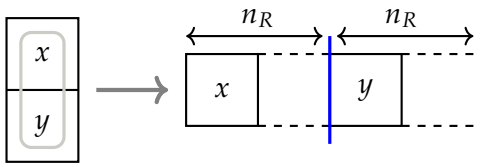


Figure 4.3.: Packing of a single complex number $x + yi$ for the tensor $\mathcal{B}$ in the 1M format.

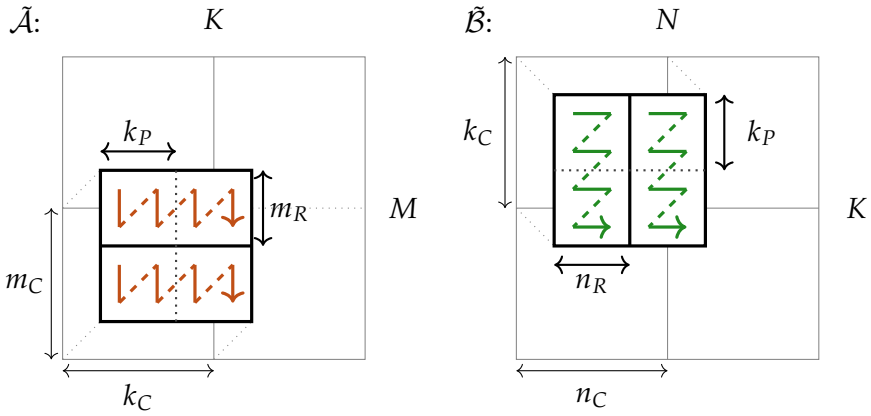


Figure 4.4.: Schematic representation of packing the tensor $\mathcal{A}$ and $\mathcal{B}$ as their structural equivalent matrices $\tilde{\mathcal{A}}$ and $\tilde{\mathcal{B}}$ in the BLIS format. The red and green zigzags denote how elements of a sliver are stored in memory.

4.5. BLIS Micro-kernels

Finally, once the workspaces $\tilde{A}$ and $\tilde{B}$ are packed, we invoke BLIS' micro-kernels for each sliver [5]. More precisely, the functions `bli_sgemm_ukernel` for single, and `bli_dgemm_ukernel` for double precision matrix-matrix multiplication. An example function call can be found in Listing 4.3.

```
1  bli_dgemm_ukernel(MR, NR, KC,
2                    ALPHA, tildeA, tildeB, BETA, tildeC,
3                    1, MR, NULL, CNTX);
```

Listing 4.3: BLIS Microkernel invocation for $m_R \times k_C$ and $k_C \times n_R$ slivers `tildeA` and `tildeB`, respectively. `ALPHA` and `BETA` are constant double pointers. The `1, MR` arguments specify the output row and column stride. `CNTX` is a BLIS-specific context object.

These kernels perform small matrix-matrix multiplications of the provided $m_R \times k_C$ and $k_C \times n_R$ slivers. Figure 4.5 illustrates the procedure graphically. Note the column-major ordering of `tildeA` and row-major ordering of `tildeB` as described in Section 4.4. We do not depict the ordering of the output `tildeC` since it depends on the row and column stride parameters. For the example in Listing 4.3, the arguments `1` for the row and `MR` for the column stride result in column-major ordering.

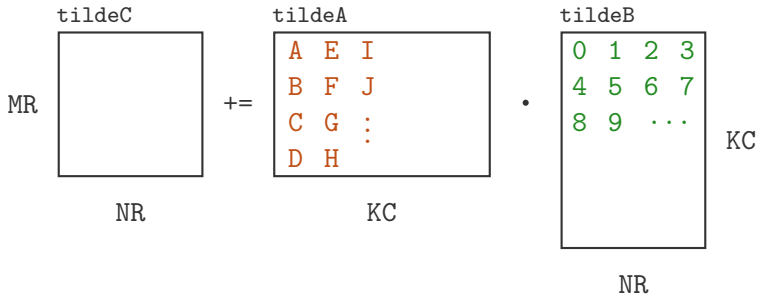


Figure 4.5.: Graphic representation of BLIS's microkernel operation. Inspired by the diagram provided in [5].

Stored in $\tilde{C}$ after the invocation, we unpack the $m_R \times n_R$ result into the output tensor $\mathcal{C}$ with two nested loops by using the Block-Scatter layout.

4.6. Multithreading

Inherited by the BLIS approach are the five parallelizable loops seen in Algorithm 3. Recent work showed that the parallelization of all loops, even though valid, might yield sub-optimal performance [21]. The authors further argue against the parallelization of the p_c loop due to the additional synchronization requirements and the possible reduction of partial results. Additionally, they state that the number of iterations is a useful heuristic for the choice of which loop to parallelize.

Using this assessment, the TBLIS library parallelizes all but the p_c loop [19]. Since both our work and TBLIS are based on the BSMTC approach, the parallelization of the same loops is a viable option. However, with a m_C of 72 and a m_R of 6 for double precision (See Chapter 5), the i_r loop has only $\frac{72}{6} = 12$ iterations. Thus, in alignment with the argument given in [21], we do not parallelize this loop. Furthermore, by testing different configurations we found that the additional parallelization of the i_r loop yielded slightly worse results.

As a result, we decide to parallelize the j_c, i_c, and j_r loop with `#pragma omp parallel for` directives.

5. Evaluation

In this chapter, we describe the limitations of our approach and present various performance measurements for an array of tensor contractions. Particularly, we evaluate our work using the tensor contraction benchmark [22]. We further show how our implementation's runtime scales with contraction size.

5.1. Limitations

Directly inherited by the 1M Method for matrix-matrix multiplication, it is not possible to scale tensors by complex factors directly. The proposed solutions for matrix-matrix multiplication described in Chapter 2 could also work for TC but have not been implemented at the time of writing. Consequently, we set $\alpha = 1$ and $\beta = 0$ for each contraction.

Di Napoli et al. show that specific types of tensor contractions can not be mapped to GEMM interfaces [13]. For example, a requirement is that tensors need to be at least two-dimensional, i.e. matrices. Thus, matrix-vector multiplication can not be performed as GEMM-like TC. Nevertheless, one can handle these edge cases by applying alternative approaches. Since we investigate the application of the 1M Method for TCs, we purposefully neglect these exceptions.

Lastly, since we utilize real microkernels for complex matrix-matrix multiplication, it is not possible to use the Block-Scatter layout for the output tensor $\mathcal{C}$. Suppose $x + yi$ is a complex number in a block with constant row and column stride, i.e. $rs > 0$, $cs > 0$ for the output tensor $\mathcal{C}$. Separate vector-vector multiplications yield the values of x and y due to the 1M method. Furthermore, by column-major ordering, both real values need to be 1-row-stride apart in memory. However, the BLIS interface takes only one row and column stride parameter. That is, we can either provide arguments to fulfill the 1-row-stride requirement or the strides rs and cs. Hence, using rs and cs as arguments is not feasible. As a consequence, we always unpack the results from an additional workspace to the actual location in memory. Nonetheless, it is still possible to unpack more efficiently using constant strides for vectorization.

5.2. Results

This section describes our experimental setup and depicts the results for a plethora of contractions.

5.2.1. Experimental Setup

All experiments run on a single Intel Xeon E52697 v3 @ 2.6 GHz processor using 14 physical cores and 2 hardware threads per core on the CoolMUC-2 HPC cluster at the Leibniz-Rechenzentrum (LRZ) [9]. Cache sizes are 32KB for L1, 256KB for L2, and 19790KB for L3. We compile with Intel's `C++` compiler `icpc` (Version 2021.4.0). BLIS is set up for the `Haswell` configuration and utilizes AVX2 and FM3 extensions. Blocking parameters as described in Section 2.2.1 are: $m_C = 72$, $n_C = 4080$, $k_C = 72$, $n_R = 8$, and $m_R = 6$ which the BLIS Framework provides via the `BLIS_DOUBLE` constant. We set $k_P = 4$ as in Matthews [19].

We compare our work to the TBLIS library which also implements the BSMTC approach and two different implementations of TTGT. The first uses the `ttt` algorithm of the Tensor Toolbox (v3.5) in MATLAB R2022b [1, 26]. An additional invocation of the `permute` function is necessary to transpose the output tensor $\mathcal{C}$ appropriately. The second is the TCL library. Both BSMTC implementations utilize BLIS's microkernels, whereby TCL uses BLIS as the underlying BLAS interface for matrix-matrix multiplication.

Moreover, we execute each contraction 10 times and report the minimum in µs (10^{-9}s). For measurements in `C++` we sandwich the evaluated function between two `std::chrono::high_resolution_clock::now()` calls and calculate the distance with the overloaded minus-operator. In MATLAB we use the integrated `tic toc` stopwatch timer.

We calculate the relative speedup factor by

$$\frac{perf_{\{TCL,TTB,TBLIS\}}}{perf_{t1m}}. \tag{5.1}$$

5.2.2. Explicit Tensor Contractions

Figure 5.1 and Figure 5.2 depict the single and multi-core performance of various tensor contractions. We adopt the contractions and tensor sizes from the tensor contraction benchmark [22]. The benchmark ensures that the total memory consumption exceeds 200MiB for each contraction. This is significantly larger than the last level cache (L3) of our system. One alteration necessary is that we halve each dimension length due to the evaluation for complex instead of real numbers. Exact sizes and performance

measurements can be found in Appendix A. Furthermore, we format each contraction as $labels(\mathcal{C})$-$labels(\mathcal{A})$-$labels(\mathcal{B})$. For example, `abcde-efbad-cf` denotes the contraction $\mathcal{C}_{abcde} = \mathcal{A}_{efbad} \cdot \mathcal{B}_{cf}$.

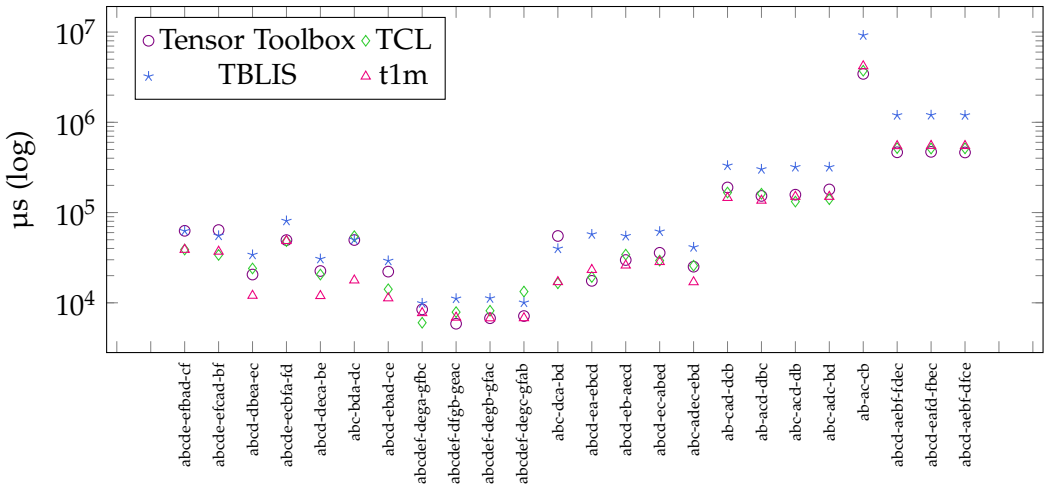


Figure 5.1.: Single-core performance of an array of different double precision complex tensor contractions on an Intel Xeon E52697 processor.

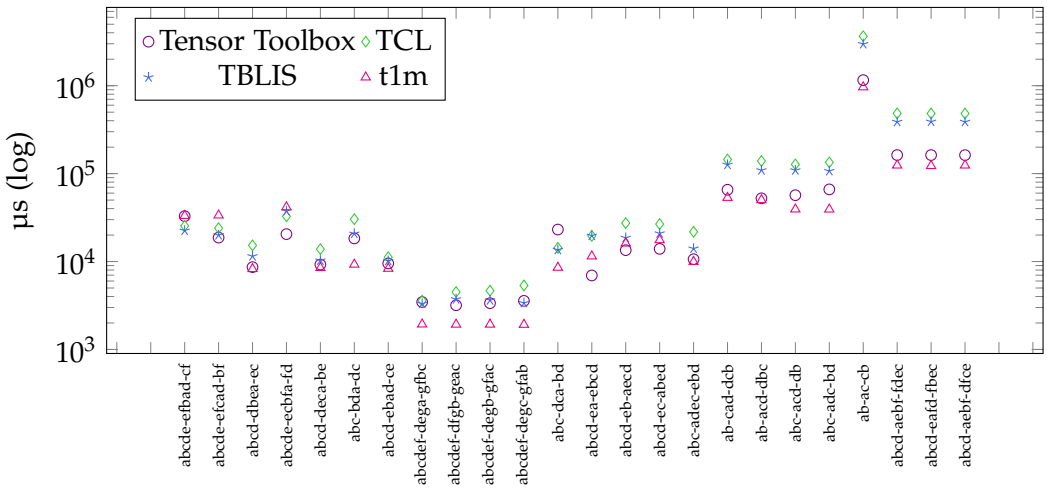


Figure 5.2.: Multi-core performance (14 Cores) of an array of different double precision complex tensor contractions on an Intel Xeon E52697 processor.

As expected, we report an overall improvement compared to TBLIS due to the incorporation of BLIS microkernels and the 1M method. On average, TBLIS takes 2.19 (minimum 1.28, maximum 2.83) times longer on a single core and 1.74 (minimum 0.6, maximum 3.18) longer on multiple cores to compute the same contraction. Exceptions, such as `abcdef-dega-gfbc` are likely due to contraction over small dimension lengths. For the example given, a single label g with dimension length 12 is contracted. In such cases, the overhead of 1M-packing to utilize BLIS microkernels yields poor performances compared to regular packing and TBLIS' triple loop in `C++`.

Furthermore, with only a median speedup factor of 1.02 (minimum 0.78, maximum 3.08), we observe similar performances of our work and TCL for the single-core benchmark. This factor increases to 2.27 (minimum 0.72, maximum 3.93) for the multi-core benchmark. Where our multithreaded implementation is on average 2.6 times faster than the single-thread one, TCL improves only by a factor of 1.36. Hence, the significant difference in multi-core performance.

We notice the biggest difference in single-core performance for the `ab-ac-cb` contraction. Note that in this case the labels of the tensors $\mathcal{A}$ and $\mathcal{B}$ as well as the resulting $\mathcal{C}$ are already in a matrix-form. Therefore it is transposition-free and TGTT reduces to a matrix-matrix multiplication.

On the contrary, for many-transposition contractions such as `abc-bda-dc`, the performance of our approach exceeds both TTGT implementations. The `abc-bda-dc` contraction is additional evidence of the superiority of the Block-Scatter approach for many-transposition contractions. For this benchmark, TBLIS yields similar results, even at bigger contraction sizes (See Section 5.2.3), even though "trivially" implemented.

A full comparison of speedup factors of our implementation and those benchmarked against can be found in Figure 5.3.

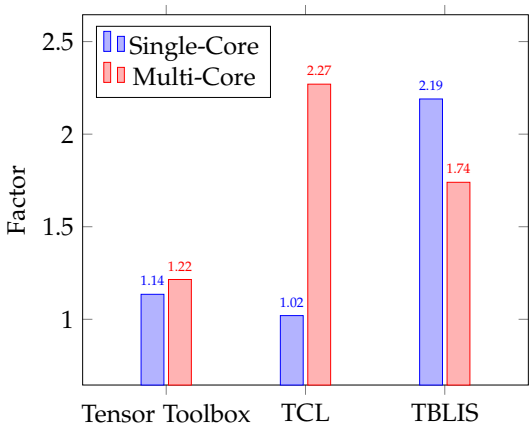
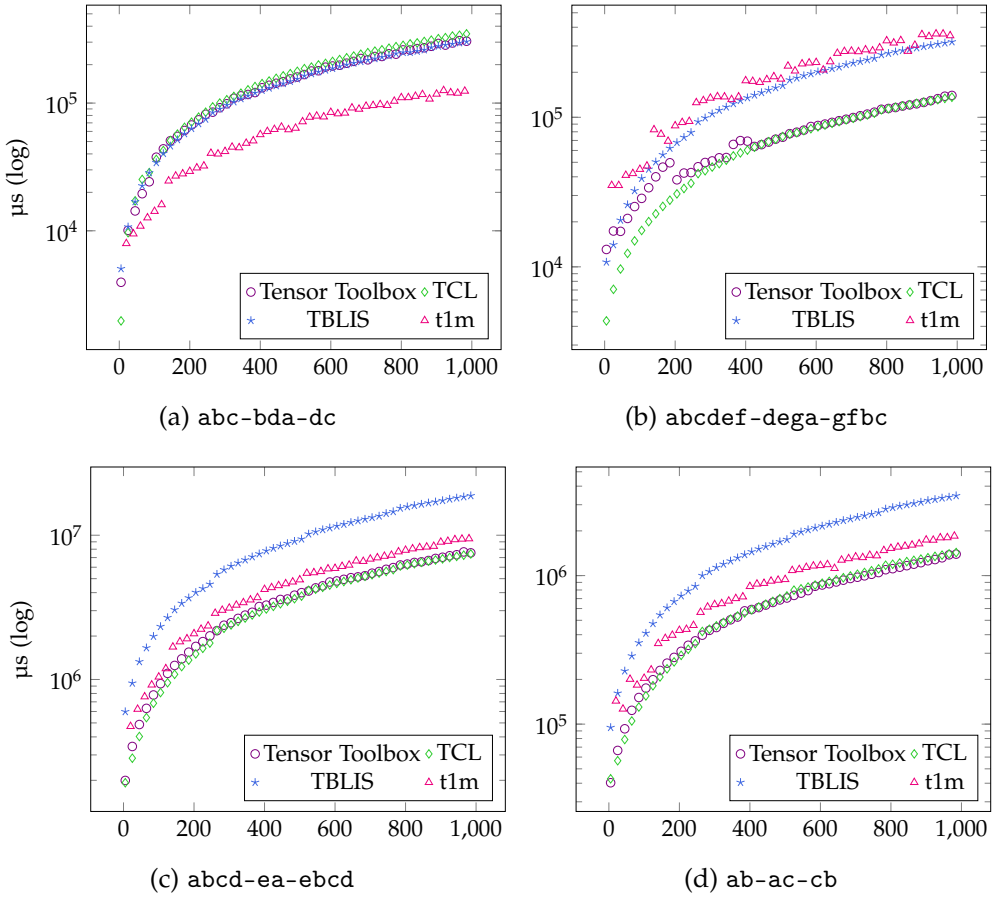


Figure 5.3.: Median speedup factors of our work for single and multi-core benchmarks.

5.2.3. Scaling Contraction Size

Figure 5.4 and Figure 5.5 depict the single and multi-core performance of salient contractions taken from section 5.2.2 with contraction sizes ranging from 5 to 1000. We choose these particular contractions for the following reasons:

- `abc-bda-dc`: Many-transpositions contraction
- `abcdef-dega-gfbc`: Worst average performance in Section 5.2.2
- `abcd-ea-ebcd`: Few-transpositions contraction
- `ab-ac-cb`: GEMM-like contraction, i.e. no transpositions



(a) `abc-bda-dc` (b) `abcdef-dega-gfbc`

(c) `abcd-ea-ebcd` (d) `ab-ac-cb`

Figure 5.4.: Single-core performance for double precision complex tensor contractions with varying contraction size on an Intel Xeon E52697 processor.

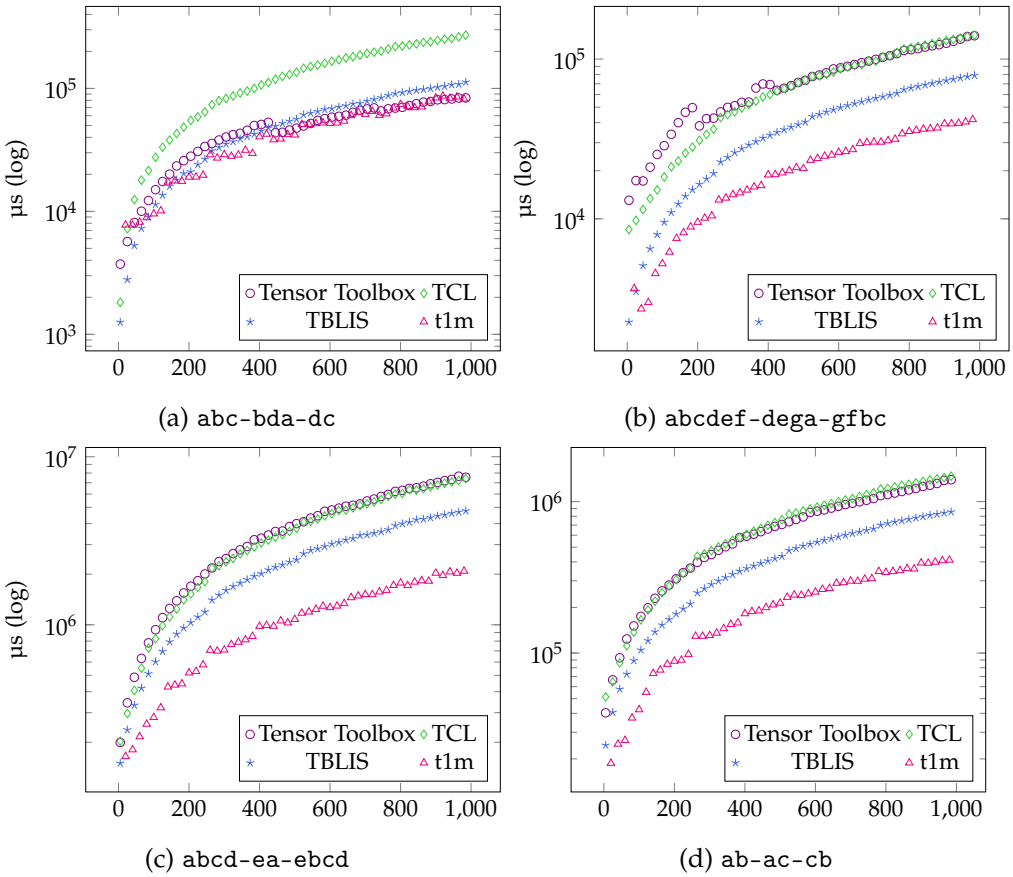


Figure 5.5.: Mutli-core (14 Cores) performance for double precision complex tensor contractions with varying contraction size on an Intel Xeon E52697 processor.

First, we observe that the single-core results for increasing contraction sizes follow the trends described in Section 5.2.2. That is, the TTGT approach excels at few-transposition as well as no-transposition contractions but underperforms for the many-transposition contraction `abc-bda-dc`.

For the multi-core benchmarks, both implementations based on the BSMTC and therefore on the BLIS approach outperform both TTGT libraries, even for the GEMM-like contraction `ab-ac-cb`. A similar result has been reported in Matthews [19]. Moreover, for the many-transposition contraction `abc-bda-dc` we measure an average speedup of 1.5 (minimum: 1.03, maximum: 1.71) for our multithreaded implementation, whereas the runtime of the Tensor Toolbox scales on average by a factor of 2.95 (minimum: 1.06, maximum: 3.71).

6. Conclusion

In this thesis, we introduced the theoretical foundation for matrix-matrix multiplication and its generalization to higher dimensions, namely Tensor Contraction.

We then outlined the design of state-of-the-art high-performance GEMM algorithms, notably, the BLIS approach. Furthermore, we presented the 1M Method for achieving GEMM for complex numbers with real-valued implementations, albeit with minimal overhead.

These algorithms set the stage for two different approaches to TC. Both use the fact that under certain conditions tensor contraction is functionally equivalent to matrix-matrix multiplication. The first, TTGT, transposes tensors such that these conditions hold and invokes a GEMM routine. However, this simple approach increases the memory requirement by a factor of two. To resolve this issue, we illustrated the scatter and block-scatter layouts. Using these techniques, one can directly access tensor elements in memory as if it were a matrix. We show that the combination of these layouts and the BLIS framework yields a high-performance tensor contraction algorithm without memory-intensive transpositions.

We proceeded by pointing out that no efficient transposition-free implementation for complex numbers exists. Therefore, we coalesced the BLIS framework, the 1M Method, and the Block-Scatter layout into one combined algorithm to achieve complex transposition-free tensor contraction.

In the evaluation, we demonstrated that the single- and multi-core performance of our implementation competes with various tensor contraction libraries. Moreover, we highlighted the practical limitations of our approach.

Future research could investigate the utilization of the block-scatter layout for complex tensor contraction on GPUs. Many GPU instruction sets provide interfaces for complex arithmetic, therefore eliminating the overhead of the 1M Method. Furthermore, the optimization of the cache and register blocking sizes potentially lead to greater performance increases.

A. Appendix A

$\mathcal{C}$-$\mathcal{A}$-$\mathcal{B}$	Sizes	t1m	TBLIS	Tensor Toolbox	TCL
abcde-efbad-cf	a:24,b:16,c:12,d:16,e:24,f:16	38799	62500	62774	38928
abcde-efcad-bf	a:24,b:12,c:16,d:16,e:24,f:16	37028	55423	64031	33932
abcd-dbea-ec	a:36,b:36,c:12,d:36,e:36	12022	34050	25096	24017
abcde-ecbfa-fd	a:24,b:16,c:16,d:12,e:24,f:24	48580	80921	49540	48284
abcd-deca-be	a:36,b:12,c:36,d:36,e:36	11931	30643	23090	20558
abc-bda-dc	a:156,b:156,c:12,d:156	17763	50067	44811	54759
abcd-ebad-ce	a:36,b:36,c:12,d:36,e:36	11258	29202	21078	14129
abcdef-dega-gfbc	a:12,b:8,c:8,d:12,e:8,f:8,g:12	7719	9882	8453	6014
abcdef-dfgb-geac	a:12,b:8,c:8,d:12,e:8,f:8,g:12	6874	11126	5864	7886
abcdef-degb-gfac	a:12,b:8,c:8,d:12,e:8,f:8,g:12	6761	11200	6749	8163
abcdef-degc-gfab	a:12,b:8,c:8,d:12,e:8,f:8,g:12	6780	10056	7132	13317
abc-dca-bd	a:156,b:12,c:148,d:156	17026	39816	55022	16466
abcd-ea-ebcd	a:36,b:36,c:36,d:36,e:36	23220	57381	17510	19316
abcd-eb-aecd	a:36,b:36,c:36,d:36,e:36	26004	54937	29795	34213
abcd-ec-abed	a:36,b:36,c:36,d:36,e:36	28408	61588	35819	29368
abc-adec-ebd	a:36,b:36,c:36,d:36,e:36	16977	41270	25130	25790
ab-cad-dcb	a:156,b:148,c:156,d:156	145797	331212	189778	166700
ab-acd-dbc	a:156,b:148,c:148,d:156	136050	301866	152860	161781
abc-acd-db	a:156,b:156,c:148,d:148	150274	318934	157733	132360
abc-adc-bd	a:156,b:156,c:148,d:148	150138	318787	180325	140730
ab-ac-cb	a:2568,b:2560,c:2568	$4.19 \cdot 10^6$	$9.18 \cdot 10^6$	3440621	$3.7 \cdot 10^6$
abcd-aebf-fdec	a:36,b:36,c:36,d:36,e:36,f:36	547265	$1.19 \cdot 10^6$	465560	517955
abcd-eafd-fbec	a:36,b:36,c:36,d:36,e:36,f:36	550024	$1.20 \cdot 10^6$	470656	512435
abcd-aebf-dfce	a:36,b:36,c:36,d:36,e:36,f:36	548595	$1.19 \cdot 10^6$	463423	514995

Table A.1.: Single-core performance in µs of an array of different double precision complex tensor contractions on an Intel Xeon E52697 processor.

$\mathcal{C}$-$\mathcal{A}$-$\mathcal{B}$	Sizes	t1m	TBLIS	Tensor Toolbox	TCL
abcde-efbad-cf	a:24,b:16,c:12,d:16,e:24,f:16	33174	22471	32837	25431
abcde-efcad-bf	a:24,b:12,c:16,d:16,e:24,f:16	33559	19977	18717	23995
abcd-dbea-ec	a:36,b:36,c:12,d:36,e:36	8246	11501	8636	15273
abcde-ecbfa-fd	a:24,b:16,c:16,d:12,e:24,f:24	41621	37405	20501	32638
abcd-deca-be	a:36,b:12,c:36,d:36,e:36	8471	10279	9249	13834
abc-bda-dc	a:156,b:156,c:12,d:156	9258	20777	18331	30314
abcd-ebad-ce	a:36,b:36,c:12,d:36,e:36	8310	10174	9521	11195
abcdef-dega-gfbc	a:12,b:8,c:8,d:12,e:8,f:8,g:12	1931	3313	3465	3591
abcdef-dfgb-geac	a:12,b:8,c:8,d:12,e:8,f:8,g:12	1922	3726	3182	4500
abcdef-degb-gfac	a:12,b:8,c:8,d:12,e:8,f:8,g:12	1924	3623	3359	4661
abcdef-degc-gfab	a:12,b:8,c:8,d:12,e:8,f:8,g:12	1915	3376	3570	5343
abc-dca-bd	a:156,b:12,c:148,d:156	8535	13557	23108	14298
abcd-ea-ebcd	a:36,b:36,c:36,d:36,e:36	11481	19709	6946	19580
abcd-eb-aecd	a:36,b:36,c:36,d:36,e:36	16091	18610	13475	27349
abcd-ec-abed	a:36,b:36,c:36,d:36,e:36	17564	20908	13937	26691
abc-adec-ebd	a:36,b:36,c:36,d:36,e:36	9958	14053	10630	21772
ab-cad-dcb	a:156,b:148,c:156,d:156	53202	126095	65531	144561
ab-acd-dbc	a:156,b:148,c:148,d:156	50111	109238	52361	139168
abc-acd-db	a:156,b:156,c:148,d:148	39318	109108	56835	127316
abc-adc-bd	a:156,b:156,c:148,d:148	39217	107042	66159	134234
ab-ac-cb	a:2568,b:2560,c:2568	963384	$2.98 \cdot 10^6$	$1.16 \cdot 10^6$	$3.6 \cdot 10^6$
abcd-aebf-fdec	a:36,b:36,c:36,d:36,e:36,f:36	124141	388406	162570	484179
abcd-eafd-fbec	a:36,b:36,c:36,d:36,e:36,f:36	122608	390259	162512	482403
abcd-aebf-dfce	a:36,b:36,c:36,d:36,e:36,f:36	124157	388052	162373	482605

Table A.2.: Multi-core performance (14 Cores) in µs of an array of different double precision complex tensor contractions on an Intel Xeon E52697 processor.

Abbreviations

TN Tensor Network

TC Tensor Contraction

BLIS BLAS-like Library Instantiation Software

BLAS Basic Linear Algebra Subprograms

GETT GEMM-like Tensor-Tensor multiplication

TCCG Tensor Contraction Code Generator

HPTT High-Performance Tensor Transpose

GEMM General Matrix-Matrix Multiplication

BSMTC Block-Scatter-Matrix Tensor Contraction

TTGT Transpose-Transpose-GEMM-Transpose

TCL Tensor Contraction Library

LRZ Leibniz-Rechenzentrum

List of Figures

List of Tables

Bibliography

[1] B. W. Bader and T. G. Kolda. "Algorithm 862: MATLAB tensor classes for fast algorithm prototyping." en. In: *ACM Transactions on Mathematical Software* 32.4 (Dec. 2006), pp. 635–653. ISSN: 0098-3500, 1557-7295. DOI: 10.1145/1186785.1186794. URL: https://dl.acm.org/doi/10.1145/1186785.1186794 (visited on 07/05/2023).

[2] J. Biamonte and V. Bergholm. *Tensor Networks in a Nutshell*. en. arXiv:1708.00006 [cond-mat, physics:gr-qc, physics:hep-th, physics:math-ph, physics:quant-ph]. July 2017. URL: http://arxiv.org/abs/1708.00006 (visited on 03/27/2023).

[3] *blis/docs/BLISObjectAPI.md at master · flame/blis*. en. URL: https://github.com/flame/blis/blob/master/docs/BLISObjectAPI.md (visited on 08/05/2023).

[4] *blis/docs/HardwareSupport.md at master · flame/blis*. en. URL: https://github.com/flame/blis/blob/master/docs/HardwareSupport.md (visited on 08/05/2023).

[5] *blis/docs/KernelsHowTo.md at master · flame/blis*. en. URL: https://github.com/flame/blis/blob/master/docs/KernelsHowTo.md (visited on 07/26/2023).

[6] *blis/docs/Multithreading.md at master · flame/blis*. en. URL: https://github.com/flame/blis/blob/master/docs/Multithreading.md (visited on 07/30/2023).

[7] *blis/docs/Performance.md at master · flame/blis*. en. URL: https://github.com/flame/blis/blob/master/docs/Performance.md (visited on 08/04/2023).

[8] J. C. Bridgeman and C. T. Chubb. "Hand-waving and interpretive dance: an introductory course on tensor networks." en. In: *Journal of Physics A: Mathematical and Theoretical* 50.22 (June 2017), p. 223001. ISSN: 1751-8113, 1751-8121. DOI: 10.1088/1751-8121/aa6dc3. URL: https://iopscience.iop.org/article/10.1088/1751-8121/aa6dc3 (visited on 03/27/2023).

[9] *CoolMUC-2*. Section: Dokumentation. URL: https://doku.lrz.de/coolmuc-2-11484376.html (visited on 07/05/2023).

[10] *cuTENSOR*. en-US. Nov. 2019. URL: https://developer.nvidia.com/cutensor (visited on 08/02/2023).

[11] *CXX Interface · devinamatthews/tblis Wiki*. en. URL: https://github.com/devinamatthews/tblis/wiki/CXX-Interface (visited on 07/26/2023).

[12] *devinamatthews/marray*. en. URL: https://github.com/devinamatthews/marray (visited on 07/26/2023).

[13] E. Di Napoli, D. Fabregat-Traver, G. Quintana-Ortí, and P. Bientinesi. "Towards an efficient use of the BLAS library for multilinear tensor contractions." en. In: *Applied Mathematics and Computation* 235 (May 2014), pp. 454–468. ISSN: 00963003. DOI: 10.1016/j.amc.2014.02.051. URL: https://linkinghub.elsevier.com/retrieve/pii/S0096300314002902 (visited on 07/26/2023).

[14] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff. "A set of level 3 basic linear algebra subprograms." en. In: *ACM Transactions on Mathematical Software* 16.1 (Mar. 1990), pp. 1–17. ISSN: 0098-3500, 1557-7295. DOI: 10.1145/77626.79170. URL: https://dl.acm.org/doi/10.1145/77626.79170 (visited on 07/10/2023).

[15] K. Goto and R. A. V. D. Geijn. "Anatomy of high-performance matrix multiplication." en. In: *ACM Transactions on Mathematical Software* 34.3 (May 2008), pp. 1–25. ISSN: 0098-3500, 1557-7295. DOI: 10.1145/1356052.1356053. URL: https://dl.acm.org/doi/10.1145/1356052.1356053 (visited on 05/18/2023).

[16] *HPAC/tccg: Tensor Contraction Code Generator*. URL: https://github.com/HPAC/tccg/tree/master (visited on 07/05/2023).

[17] J. Huang, D. A. Matthews, and R. A. van de Geijn. *Strassen's Algorithm for Tensor Contraction*. en. arXiv:1704.03092 [cs]. Apr. 2017. URL: http://arxiv.org/abs/1704.03092 (visited on 05/07/2023).

[18] Ö. Legeza, R. Noack, J. Sólyom, and L. Tincani. "Applications of Quantum Information in the Density-Matrix Renormalization Group." en. In: *Computational Many-Particle Physics*. Ed. by H. Fehske, R. Schneider, and A. Weiße. Lecture Notes in Physics. Berlin, Heidelberg: Springer, 2008, pp. 653–664. ISBN: 978-3-540-74686-7. DOI: 10.1007/978-3-540-74686-7_24. URL: https://doi.org/10.1007/978-3-540-74686-7_24 (visited on 08/07/2023).

[19] D. A. Matthews. *High-Performance Tensor Contraction without Transposition*. en. arXiv:1607.00291 [cs]. July 2017. URL: http://arxiv.org/abs/1607.00291 (visited on 03/27/2023).

[20] R. Orus. "A Practical Introduction to Tensor Networks: Matrix Product States and Projected Entangled Pair States." en. In: *Annals of Physics* 349 (Oct. 2014). arXiv:1306.2164 [cond-mat, physics:hep-lat, physics:hep-th, physics:quant-ph], pp. 117–158. ISSN: 00034916. DOI: 10.1016/j.aop.2014.06.013. URL: http://arxiv.org/abs/1306.2164 (visited on 07/31/2023).

[21] T. M. Smith, R. V. D. Geijn, M. Smelyanskiy, J. R. Hammond, and F. G. V. Zee. "Anatomy of High-Performance Many-Threaded Matrix Multiplication." en. In: *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. Phoenix, AZ, USA: IEEE, May 2014, pp. 1049–1059. ISBN: 978-1-4799-3800-1 978-1-4799-3799-8. DOI: 10.1109/IPDPS.2014.110. URL: https://ieeexplore.ieee.org/document/6877334 (visited on 05/21/2023).

[22] P. Springer and P. Bientinesi. *Design of a high-performance GEMM-like Tensor-Tensor Multiplication*. en. arXiv:1607.00145 [cs]. Nov. 2017. URL: http://arxiv.org/abs/1607.00145 (visited on 05/11/2023).

[23] P. Springer, T. Su, and P. Bientinesi. *HPTT: A High-Performance Tensor Transposition C++ Library*. en. arXiv:1704.04374 [cs]. May 2017. URL: http://arxiv.org/abs/1704.04374 (visited on 05/20/2023).

[24] E. M. Stoudenmire and D. J. Schwab. *Supervised Learning with Quantum-Inspired Tensor Networks*. en. arXiv:1605.05775 [cond-mat, stat]. May 2017. URL: http://arxiv.org/abs/1605.05775 (visited on 07/31/2023).

[25] *Sunsetting Python 2 | Python.org*. URL: https://www.python.org/doc/sunset-python-2/ (visited on 07/05/2023).

[26] *Tensor Toolbox for MATLAB*. URL: http://www.tensortoolbox.org/ (visited on 07/17/2023).

[27] F. G. Van Zee. "Implementing High-Performance Complex Matrix Multiplication via the 1M Method." en. In: *SIAM Journal on Scientific Computing* 42.5 (Jan. 2020), pp. C221–C244. ISSN: 1064-8275, 1095-7197. DOI: 10.1137/19M1282040. URL: https://epubs.siam.org/doi/10.1137/19M1282040 (visited on 05/07/2023).

[28] F. G. Van Zee and R. A. Van De Geijn. "BLIS: A Framework for Rapidly Instantiating BLAS Functionality." en. In: *ACM Transactions on Mathematical Software* 41.3 (June 2015), pp. 1–33. ISSN: 0098-3500, 1557-7295. DOI: 10.1145/2764454. URL: https://dl.acm.org/doi/10.1145/2764454 (visited on 05/19/2023).