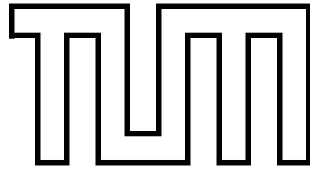


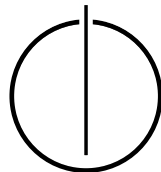
School of Computation, Information and Technology
- Informatics

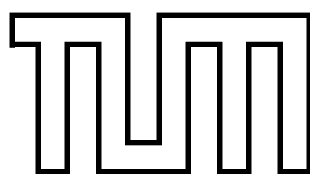
Technical University of Munich

Bachelor's Thesis in Informatics

Implementation of Sparse Tensor Networks Based on Quantum Number Conservation

Felix Leon Griebel





School of Computation, Information and Technology - Informatics

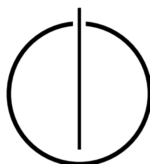
Technical University of Munich

Bachelor's Thesis in Informatics

Implementation of Sparse Tensor Networks Based on Quantum Number Conservation

Implementierung von dünnbesetzten Tensornetzwerken basierend auf Quantenzahlerhaltung

Author:	Felix Leon Griebel
Supervisor:	Univ.-Prof. Dr. Christian Mendl
Advisor:	Manuel Geiger, M.Sc.
Date:	15.03.2024



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

Munich, 15.03.2024 Felix Leon Griebel

Abstract

The states in quantum systems can be described as a wave function. Calculations on tensor networks can approximate this function. As these approximations are not always accurate, especially for tensor networks with large dimensions, symmetries can be exploited. Integrating these in the tensors used for the network, can improve the approximation and reduce the resources needed. One symmetry for this purpose is the U(1) group. We will focus on the integration of this symmetry, specifically for sparse tensors. With this basis, four elementary operations will be discussed, which are crucial for calculating tensor networks: permutation of indices, reshaping tenor indices, multiplying matrices, and decomposing a matrix. However, for the last one, we only cover the theory behind it. Additionally, to the base implementations, we include approaches for improving the performance of these methods. These include changes to the underlying algorithms or using multiple threads to parallelize parts of the methods. As we see in the tests, most of these approaches were not able to increase the performance of the base implementations and are rather decreasing it. This happens most likely due to different reasons, one of the major ones being the overhead that results from the approaches.

Zusammenfassung

Die Zustände in Quantensystemen lassen sich als Wellenfunktion beschreiben. Diese Funktion kann durch Berechnungen mit Tensornetzen approximiert werden. Da diese Approximationen insbesondere bei großen Tensornetzwerken nicht immer genau sind, können Symmetrien ausgenutzt werden. Die Integrierung dieser Symmetrien in die für das Netz verwendeten Tensoren, kann die Approximationen verbessern und den Ressourcenbedarf verringern. Eine Symmetrie hierfür ist die Gruppe U(1). Wir konzentrieren uns auf die Integration dieser Symmetrie, speziell für dünnbesetzte Tensoren. Auf dieser Grundlage werden vier elementare Operationen besprochen, die für die Berechnung von Tensornetzwerken entscheidend sind: Permutation von Indizes, Umformung von Tensorindizes, Multiplikation von Matrizen und Matrixzerlegung. Für die Zerlegung einer Matrix decken wir jedoch nur die Theorie ab. Zusätzlich zu den Basisimplementierungen beziehen wir Ansätze zur Leistungsverbesserung dieser Methoden ein. Dazu gehören Änderungen an den zugrundeliegenden Algorithmen oder die Verwendung mehrerer Threads zur Parallelisierung der Methoden. Wie in den Tests zu sehen ist, konnten die meisten dieser Ansätze die Performance der Basisimplementierungen nicht erhöhen, sondern verringern sie tendenziell eher. Dies geschieht höchstwahrscheinlich aus verschiedenen Gründen, wobei einer der Hauptgründe der zusätzliche Overhead ist, der sich aus den Ansätzen ergibt.

Contents

Abstract vii

Zusammenfassung ix

I. Introduction and Background 1

1. Introduction 2
1.1. Introduction and Background . . . 2
1.2. Tensors . . . 3
1.3. Symmetry . . . 4
1.4. Related Work . . . 6

II. Implementation 7

2. Methods 8
2.1. Class Structure . . . 8
2.2. Primary Operations . . . 9
2.2.1. Base Implementation . . . 9
2.2.2. Algorithmic Improvements . . . 17
2.2.3. Multithreading Improvements . . . 20
2.2.4. Utilizing OpenMP . . . 21
2.2.5. SIMD Improvements . . . 21

III. Results and Discussion 23

3. Results 24
3.1. Benchmarking . . . 24
3.1.1. Testing details . . . 24
3.1.2. Permute . . . 25
3.1.3. Fuse . . . 27
3.1.4. Split . . . 28
3.1.5. Contract . . . 29

4. Discussion 32

5. Conclusion 33

IV. Appendix 34

Bibliography 37

Part I.

Introduction and Background

1. Introduction

1.1. Introduction and Background

The concept of quantum computers promises a significant advantage over conventional computer systems. In comparison to the classical computers which operate using bits, quantum computers utilize qubits. The key here is, that with qubits, a property called ”quantum entanglement” can be exploited, where multiple qubits are influenced by the same operation. This phenomenon can offer us an exponential speed-up in solving important problems. As a consequence of this increase in the performance of problem-solving, quantum computers promise to have a major impact on various aspects of modern society. As they are based on their resource-expensive calculability with conventinal computer systems, one area involved is encryption algorithms and information security protocols.[Rah22]
On the other hand, the medical and chemical fields will also be affected, as quantum computers can help with the simulation of new substances and medication. As a result, testing on animals could be decreased and the overall development process of a drug could be shortened.[CR23]However, the main requirement for these advantages is quantum supremacy. It simply describes when quantum computers will exceed the computational power of current super computers.[ZSW20][BCOT11]
Even before quantum supremacy, it is an important topic to research algorithms for quantum-based systems, to understand the possibilities. We typically achieve this by simulating quantum computers. To state of such computers can be represented, using a wave function. This wave function, on the other hand, can be approximated, using tensor network decompositions such as matrix product states (MPS) or multi-scale entanglement renormalization ansatz (MERA), as seen in figure 1.4. However, with the growing size of tensor network decompositions, the approximation of the wave function is more limited. [BCOT11] Thus, symmetries are being used, to increase the accuracy. Because of this reason, we implemented a library for the manipulation of tensors, that are canonical in regards to a symmetry. In this case, the symmetry included is specifically the abelian U(1) group.
Two major concerns in the practical use might be memory usage and time consumption. As we focus on sparse tensors, we implemented a structure that concentrates on this sparse nature to reduce memory consumption. Following, we can build operations on this basis that approach the performance instead. That way, we cover both concerns.[SPV10][SPV11]

The following section explains the basics of tensors and the use of symmetry. Additionally, the content of the paper will be compared with other related papers and works.
In Chapter II, we will discuss further the implementation itself. Specifically, section 3.1 focuses on the structure of the tensor class itself. Section 3.2 includes the main operations: permutation and reshaping of indices, contraction of two tensors, and the factorization of a tensor of rank 2. These operations are explained in section 3.3, including the basic implementation and its optimization approaches. Chapter III will explain the results, including the discussion and conclusion.

1.2. Tensors

In the following, we will cover the basics regarding tensors, that are relevant for the use in this paper.

In general, tensors describe multi-dimensional arrays. A tensors rank refers to the amount of the tensors indices. For example, a tensor of rank 0 does not contain an index and is referred to as a scalar, while a tensor of rank 2 is a matrix and includes 2 indices. The size of a tensor index win also be called the index dimension. Tensors can also be represented graphically, as seen in figure 1.1, by a circle with legs. Each leg represents an index of the tensor.[SPV10]

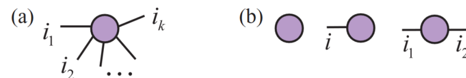


Figure 1.1.: (a) A graphical depiction of a tensor with k legs and therefore k indices. (b) A graphical depiction of tensors with rank 0, 1 and 2. Thus, these tensors represent a scalar, a vector and a matrix. Source: [SPV11]

A tensor may also be manipulated using different operations, such as permutation of the indices, splitting, or fusing. Additionally, multiple tensors can also be combined to form a tensor network, for example by contracting tensors along specific indices. Some of these methods are depicted in figure 1.3. Additionally, in figure 1.2a, 1.2b and 1.2c are the operations depicted in mathematical notation.

$$(\hat{R}')_{adbc} = \hat{R}_{abcd}$$
$$(\hat{S}')_{bcfh} = \hat{S}_{cfbh}$$

(a) The permutation of the indices of Tensors R and S with indices a,b,c,d and c,f,b,h

$$(\hat{R}'')_{uy} = (\hat{R}')_{adbc}$$
$$(\hat{S}'')_{yw} = (\hat{S}')_{bcfh}$$

(b) The fusing of the indices of Tensors R' and S' from (a)

$$(\hat{T}'')_{uw} = \sum_y (\hat{R}'')_{uy}(\hat{S}'')_{yw}$$

(c) The contraction of Tensors R" and S" from (b) along index y

Figure 1.2.: Different tensor operations in mathematical notation. Source: [SPV11]

Figure 1.3.: The different operations permutation of indices (a), and fusing and splitting of indices (b).
Source: [SPV11]

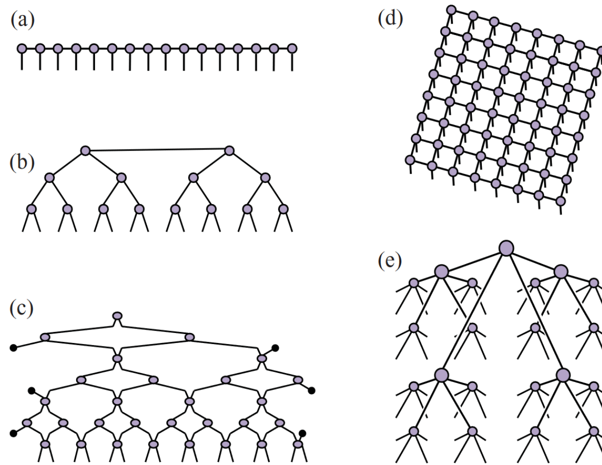


Figure 1.4.: Visualized examples of tensor network states: (a) matrix product state (MPS), (b) tree tensor network (TTN), (c) multiscale entanglement renormalization ansatz (MERA), (d) projected entangled-pair state (PEPS), and (e) 2D TTN
Source: [SPV11]

1.3. Symmetry

Different kinds of symmetries might be used to improve the approximation accuracy of a wave function. However, we will focus on the abelian symmetry of the U(1) group. To include this symmetry in the concept of the tensors, we add two additional attribute of the tensor's indices. Firstly, we add to each index a tuple of particle numbers, also referred to as charges or quantum numbers, with the same dimension as the index. That way, we have a quantum number for every coordinate of an index. These quantum numbers are used to describe a specific state of a system. In our case, the quantum numbers can be used to describe the symmetry.[Pfe17] Additionally, we assign the directions ”incoming” or ”outgoing” to each index (figure 1.6). To preserve the quantum numbers, we only allow elements in the tensor to be non-zero, if the sum of the corresponding quantum numbers of the incoming indices equals the sum the ones of the outgoing indices (figure 1.5). (TODO: Add further explanation through graphics) [SPV11]

$$\begin{array}{c|cccc} & 3 & 4 & 1 & 4 \\ \hline 4 & 0 & X & 0 & X \\ 1 & 0 & 0 & X & 0 \\ 4 & 0 & X & 0 & X \\ 2 & 0 & 0 & 0 & 0 \end{array}$$

Figure 1.5.: A matrix with quantum numbers along the sides (purple). Consistent with the symmetry, the matrix only contains non-zero values, where the quantum numbers match.

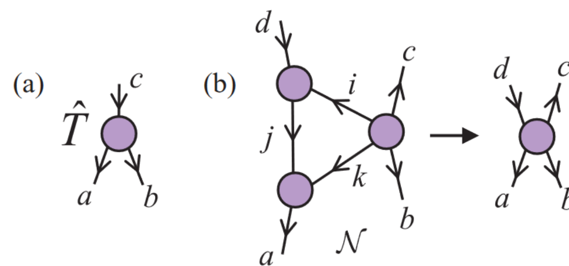


Figure 1.6.: (a) a grapical representation of a tensor with one incoming and two outgoing indices. (b) The representation of a tensor network, with directions.
Source: [SPV11]

Given a specific index, we can accumulate the elements with the same quantum number to a block. Furthermore, in case we have a tensor of rank 2 with this symmetry, we can assign each non-zero element a block, as the corresponding quantum numbers in both indices have to be equal. An example of a block can be seen in figure 1.7.

(a)

$$\begin{array}{c|cccc} \text{incoming} \backslash \text{outgoing} & 3 & \boxed{4} & 1 & \boxed{4} \\ \hline \boxed{4} & 0 & 3 & 0 & 4 \\ 1 & 0 & 0 & 2 & 0 \\ \boxed{4} & 0 & 5 & 0 & -3 \\ 2 & 0 & 0 & 0 & 0 \end{array}$$

(b)

$$\begin{pmatrix} 3 & 4 \\ 5 & -3 \end{pmatrix}$$

Figure 1.7.: (a) A matrix with quantum numbers along the sides (purple).
(b) A submatrix of the matrix in (a), corresponding to the block of the quantum number with value 4.

1.4. Related Work

The work in the implementation is in major parts based on the knowledge from the two papers [SPV11] and [SPV10]. These papers focus on discussing the theory behind the calculation of tensors in the presence of symmetries from a physics point of view. While covering the basics of tensors, the authors explain how to integrate the symmetries in them and their operations. While the first paper does not specify one symmetry only, the second paper focuses on the U(1) group symmetry. The latter also explains that the contraction of multiple tensors, a fundamental step for the building of tensor networks, can be broken into elementary steps, given as permutation of a tensors indices, reshaping the indices by fusing them, multiplying to matrices to obtain a new one, reshaping the resulting matrix again by splitting the indices and lastly permutating the resulting indices again. The manipulation of a tensor network can then be described based on four different operations: 1. permutation of the indices of a tensor, 2. reshaping the indices of a tensor, 3. multiplication of two matrices, and 4. decomposition of a matrix. Contrary to these two papers, we will just include the theory, where needed for the understanding of this project.[SPV11][SPV10]

The paper "Implementing global Abelian symmetries in projected entangled-pair state algorithms" by B Bauer et al. also takes an approach for implementing symmetries in tensor networks but focuses on projected entangled-pair state (PEPS) algorithms. Additionally, it is not specified, whether the relying tensors, used for the calculation, are mostly dense or sparse.[BCOT11]

Similar projects may include "TensorFlow" [AAB+15] or "TensorNetwork" [RMG+19], which are both implemented and applicable in Python. However, while TensorNetwork supports tensors with symmetry and charges similar to this project, TensorFlow does not. While these projects focus on the work with the Python programming language, this project is implemented and supports its use in C++.

In comparison to these different projects, we will discuss a practical implementation. It focuses on the calculation with sparse tensors, that include the U(1) symmetry. For this purpose, we concentrated on implementing the four different operations on tensor networks, discussed in the paper "Tensor network states and algorithms in the presence of a global U(1) symmetry" above. [SPV11]

Part II.

Implementation

2. Methods

2.1. Class Structure

```
1  class Tensor {
2      private:
3      int rank;
4
5      vector <bool> isIndexOutgoing;
6      vector <vector<int>> quantumNumbers;
7
8      vector<vector<int>> positionTable;
9      vector <complex<double>> tensorElements;
10
11     ...
12
13 }
```

Listing 2.1: The attributes of the tensor class

The Tensor class consists of multiple basic attributes. Firstly, even though the value could be derived from other attributes, the *rank* of the tensor is stored on its own. This adds helpful redundancy to the attributes. Especially during some methods, the other attributes that could be used to calculate the *rank* might be changed and therefore unreliable during these periods. With the rank saved separately, we don't depend on the correctness of said variables and can use the attribute without calculating or temporarily saving it.

The elements of the tensor are saved in two different structures. Firstly, the complex values themselves are stored in a vector, called *tensorElements*, without a specific order. The position of the values in the tensor is stored in another structure, the matrix *positionTable*. Each row, later also referred to as an element of the table, contains the coordinates based on the indices and an additional value at the end. The last value points to the position of the complex value in the *tensorElements* structure. Therefore, if we have a specific row in the matrix, we can access the corresponding complex number in $\mathbf{O(1)}$. Nevertheless, the overall access of an element over the coordinates would be in $\mathbf{O(n)}$. In this structure, only non-zero values are stored. A similar approach is conventionally used for sparse matrices or vectors, to not store all elements. We exploit the same principle here as well. The separation of the values and their coordinates has the background that the result is a simpler data structure and multiple operations only need to modify the *positionTable*.

The next variables describe the attributes specific to the U(1) symmetry. Firstly, the class contains a structure for the quantum numbers. This is achieved by building a vector of vectors, which represents a list of each index's quantum numbers. As the quantum numbers of an index have the same dimension as the index itself, we can derive the index sizes from this variable. Theoretically, we could also calculate the rank of the tensor based on this structure. Lastly, we have a vector, called *isIndexOutgoing*, that describes for each of the

indices the direction. The value true indicates, that the index is outgoing and otherwise incoming.

This is the foundation that is used for further calculations. During the development, other attributes were considered. Another vector previously included the current order of the indices, naming them with the numbers from 0 to rank-1. However, this would add unnecessary complexity to the calculations. Furthermore, the user would need to know the current order of the indices to permute them based on that order. The two variables for the complex values in the tensor were also changed to the current state. The *positionTable* was intended as a hashmap with the coordinates as keys and the pointer to the correct complex number in the other structure as values. With this approach, acccessing the value to specific coordinates would be in $\mathbf{O(1)}$ instead of $\mathbf{O(n)}$. However, during further development, this also proved to be not suitable, as the access would be fast, but the different operations on the structure would be more complex and time-consuming.

2.2. Primary Operations

To simulate tensor networks, we need multiple different operations. These operations can be summed up in four different groups. Firstly, we can permute the order of the tensor indices. Secondly, the indices can be reshaped. More specifically, multiple indices can either be fused into one, or one can be split into multiple ones again. Furthermore, two tensors of rank 2 can be contracted along different indices. The result will be a new tensor with the indices of both preceding tensors combined. Lastly, we can use matrix factorization to decompose a tensor of rank 2 into multiple others.[SPV11]

These operations can be combined in different ways to achieve desired states. How the operations can or should be combined in this library, will be discussed within the description of the specific methods.

2.2.1. Base Implementation

This subsection will discuss the basic idea behind the implementation of the method. It won't include any further optimizations, such as multithreading or further algorithmic improvements, as this is described in the following segments. Furthermore, the description will not include error checks at the beginning of each method for checking the correctness of the parameters. These versions of the different operations will later be used for building the different optimization approaches. In the implementation and the result section of this paper, the base implementation versions can be recognized by the included "_V0" in the method name.

Consistency

All methods are designed, to have a consistent tensor before and after the operation. In detail, this means that the resulting tensor will not have non-zero elements, whereas the quantum numbers would not allow it. Even though this may limit our options for modifying tensors, we have the advantage of only handling tensors that are consistent, according to the symmetry. Due to this consistency, we can sometimes disregard checks for the correctness of the tensors in the operations themself.

Permute

```
1 bool permute_V0(vector <int> newOrder){...}
```

Listing 2.2: The header for the fuse method

For this method, we use a parameter that will define the new order of the indices. This order will be the desired outcome. Originally, another approach intended, to use a global variable for the order of the indices, so that it can be remembered inside the tensor. However, a user would need to remember it as well. For this reason, the order of the indices after a permute operation, in case it is successful, will be the same as before the operation. This means, we always refer to the indices in ascending order, starting from "0" to "rank-1".

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 2 & 4 \\ 0 & -3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \qquad positionTable : \begin{matrix} (0,1,0) \\ (1,2,1) \\ (1,3,2) \\ (2,1,3) \\ (3,0,4) \end{matrix} \tag{2.1}$$

Example: Given a tensor $\widehat{T}$ of rank 2 with indices i_1, i_2 and the corresponding quantum numbers qn_1, qn_2 and index directions dir_1, dir_2. After calling *permute*({1,0}), the symmetry-related variables will change according to 2.2. If we visualize the matrix as in 2.1, the operation would switch the coordinates of i_1 and i_2 in the *positionTable*, as depicted in 2.3.

$$(qn_1, qn_2) \to (qn_2, qn_1), \qquad (dir_1, dir_2) \to (dir_2, dir_1) \tag{2.2}$$

$$\begin{matrix} (0,1,0) \\ (1,2,1) \\ (1,3,2) \\ (2,1,3) \\ (3,0,4) \end{matrix} \to \begin{matrix} (1,0,0) \\ (2,1,1) \\ (3,1,2) \\ (1,2,3) \\ (0,3,4) \end{matrix} \tag{2.3}$$

In the implementation, we iterate over the *positionTable* variable and build up the coordinates of each entry in the new order. The new order of the entry will be stored back into the structure. Afterward, the variables specific to the symmetry, *quantumNumbers* and *isIndexOutgoing*, will be restructured as well. This happens in the same manner as the coordinates for the entries in *positionTable*. Due to the specific structure of the tensor, there is no need to change something in the *tensorElements* variable.

Another approach, instead of building up the coordinates in the right order and saving them back to the structure again, would be to switch the coordinate numbers. At the start of the method, a sequence could be calculated for what index coordinates to switch, in order to get to the new order. This sequence could be applied on all entries of the *positionTable*, as well as on the *quantumNumbers*- and *isIndexOutgoing*-variables. This approach would result in a more in-place process. However, in addition to the calculation of the sequence, we would still have to apply the procedure to each element. Therefore, even though this approach would be in place, it would add a calculation overhead to the overall operation. As we already use sparse tensors, we decided to use the faster approach, rather than the in-place one.

Fuse

```
1 bool fuse_V0(int startWithPosition, int amountOfIndices, bool makeOutgoing)
      {...}
```

Listing 2.3: The header for the permute method

The Fuse method expects three parameters, the position of the first index for fusing, the amount of indices that will be fused and whether the resulting index will be outgoing or not. The method will fuse the defined amount of indices consecutive, starting with the index according to the first parameter. However, we can not fuse non-consecutive indices with this approach. In order to still be able to do this, a tensor can first be manipulated with the permute operation to put the indices that should be fused in a row. Afterwards fuse can be called and if needed permute again.

Example: Given a tensor $\widehat{T}$ of rank 2 with indices i_1, i_2 and the corresponding quantum numbers qn_1, qn_2 and index directions dir_1 = false, dir_2 = true. After calling *fuse*(0,2, false), the resulting symmetry-related variables are calculated as in 2.4. The new quantum numbers are calculated similarly to a Kronecker product of two vectors. However, instead of multiplying the two values, we subtract or add them (this depends on the directions of the indices). If we visualize the matrix as in 2.1, the *positionTable* would change with the operation as seen in 2.5.

$$qn_{new} = \{qn_1^0 - qn_2^0, qn_1^0 - qn_2^1, ..., qn_1^m - qn_2^0, ..., qn_1^m - qn_2^n\}, \quad dir_{new} = false \tag{2.4}$$

$$\begin{matrix} (0,1,0) & & (1,0) \\ (1,2,1) & & (6,1) \\ (1,3,2) & \rightarrow & (7,2) \\ (2,1,3) & & (9,3) \\ (3,0,4) & & (12,4) \end{matrix} \tag{2.5}$$

In the base implementation, we again iterate over all elements in *positionTable*. For each element, the coordinate for the new index is calculated, based on the previous indices. Afterwards, the *quantumNumbers* for the fused index will be calculated. For each position in the index, a sum is used. If the direction of the desired index is the same as the preceding index, the quantum number will be added to the sum and subtracted otherwise. The quantum numbers of the old indices will then be removed and replaced with the newly calculated ones. The *isIndexOutgoing* and *rank* attributes of the tensor will be adjusted accordingly. After the method, the resulting index will be at the position of the first index for the fusing.

Split

```
1 bool split_V0(int indexToSplit, vector<int> qnA, vector<int> qnB, bool
     makeAout, bool makeBout){...}
```

Listing 2.4: The header for the split method

The split method serves the same purpose as fuse, reshaping the indices of the tensor by splitting an index into two new ones. The method expects five different parameters. Firstly, the position of the index that should be split. Furthermore, for each of the new indices the new quantum numbers and the direction. The reason to define these symmetry-specific variables is mainly that there are multiple possibilities to split an index into two new ones. These parameters are then used to define the chosen option and to check if it is valid.

Example: Given the same tensor $\widehat{T}$ of rank 2 as in 2.2.1 "Fuse", with indices i_1, i_2 and the corresponding quantum numbers qn_1, qn_2 and index directions dir_1 = false, dir_2 = true. If we execute the fuse operation as in 2.2.1 "Fuse", the result would be a vector. We could then split the one index of the vector back into two by applying $split(0, qn_1, qn_2, dir_1, dir_2)$. When the method has checked the correctness of the quantum numbers, the values can just be replaced. The *positionTable* would change back again, as depicted in 2.5.

$$\begin{matrix} (1,0) & & (0,1,0) \\ (6,1) & & (1,2,1) \\ (7,2) & \rightarrow & (1,3,2) \\ (9,3) & & (2,1,3) \\ (12,4) & & (3,0,4) \end{matrix} \tag{2.6}$$

The first major part of the method checks, whether the index can be split into two new indices with the given parameters. Specifically, the two new indices would, if fused, result in the current index that is to split. After the check, the quantum numbers and index direction attributes can be changed to the specified values. The two indices will be consecutive, starting at the position of the previous origin index. Even though the elements of the tensor do not have to be modified, the coordinates in the *positionTable* variable must be changed to fit the new indices.

Another possible approach to the split method would be to only split an index into two or more indices, which were fused in previous operations to create the current one. However, this would need a documented history, in which order the indices were fused, and additionally, the quantum numbers of the original indices. Moreover, it would limit the user's possibilities. With the implemented approach, a user of the framework could split an index into previously existing ones, but also into two indices with completely new quantum number combinations.

Contract

```
1 Tensor multiply_V0(Tensor bTens){...}
```

Listing 2.5: The header for the multiply method

This method is used to contract two tensors of rank 2, or matrices, along specific indices to get a new one. In the implementation, this method is also called multiply. The method expects one parameter. Specifically, the second tensor used for the contraction. The first tensor that is used, will be the one from which the method is called. Additionally, the operation expects a direction of flow in the contraction, meaning one of the indices should be outgoing and the other one incoming. The method also expects the indices that should be contracted to be last in each tensor. Lastly, the sizes of the indices that are contracted should be the same.

Because the indices for the operation are expected to be last, the user may use the permute method to achieve this state. Additionally, if multiple indices should be contracted, the fuse method can be used to get them combined in one index for the multiplication and another index that will not be changed.

Firstly, the entries in the *positionTable* variables of each tensor are ordered ascending based on the element's quantum numbers in the last index.
The algorithm itself is illustrated in algorithm 1 and works as follows. With two pointer on the entries of each sorted *positionTable*, we start comparing the quantum numbers. If the quantum numbers match, the coordinates of the last index itself are compared. If these match as well, the multiplicated result of both elements is added to the entry of the new tensor with the given coordinates. Afterward, we increase the pointer in the second tensor. If the quantum numbers in the second tensor do not match anymore, but the next entry in the first tensor contains the same number, we jump back to the position where they first matched in the second one and increase the pointer in the first tensor. Otherwise, we just increase the pointer in the first tensor and do not jump back. This will be done until both pointers are at the end of each *positionTable* variable.

Afterward, the first tensor is used to create the resulting tensor of the contraction. The last index is removed in all variables and the indices of the second tensor, except the last one, are added. Lastly, the structure of the tensor that saves the elements is adjusted.

An attribute of this operation that we can utilize later on is that each of the quantum number blocks can be calculated on their own.

Algorithm 1: Multiply base implementation algorithm

Input: Two Tensors $\widehat{A}$ and $\widehat{B}$ with indices a_1, a_2 and b_1, b_2
Output: The new Tensor $\widehat{C}$ with indices c_1 and c_2

1 $\widehat{A}$.positionTable ← $\widehat{A}$.positionTable sorted by quantum numbers of a_2
2 $\widehat{B}$.positionTable ← $\widehat{B}$.positionTable sorted by quantum numbers of b_2
3 **for** *i ← 0* **to** *size of $\widehat{A}$.positionTable* **do**
4   attach quantum number of a_2 to $\widehat{A}$.positionTable[i]
5 **for** *i ← 0* **to** *size of $\widehat{B}$.positionTable* **do**
6   attach quantum number of b_2 to $\widehat{B}$.positionTable[i]
7 $counter_a \leftarrow 0$
8 $counter_b \leftarrow 0$
9 $jumpBack \leftarrow 0$
10 $jumpBack_{already_set} \leftarrow$ false
11 **while** $counter_a$ < *size of $\widehat{A}$.positionTable* & $counter_b$ < *size of $\widehat{B}$.positionTable* **do**
12   **if** *quantum number of a_2 and b_2 are equal for the entries to which the $counter_a$ and $counter_b$ point* **then**
13   **if** `not` $jumpBack_{already_set}$ **then**
14             $jumpBack_{already_set} \leftarrow$ true
15             $jumpBack \leftarrow counter_b$
16   **if** *coordinates of a_2 and b_2 are equal for the entries at $counter_a$ and $counter_b$* **then**
17   add $\widehat{A}$.positionTable[$counter_a$] × $\widehat{B}$.positionTable[$counter_a$] to entry in $\widehat{C}$
18   **if** *element after $counter_b$ in $\widehat{B}$ has the same quantum number* **then**
19             $counter_b \leftarrow counter_b + 1$
20   **else if** *element after $counter_a$ in $\widehat{A}$ has the same quantum number* **then**
21             $counter_a \leftarrow counter_a + 1$
22             $counter_b \leftarrow jumpBack$
23   **else**
24             $counter_a \leftarrow counter_a + 1$
25             $counter_b \leftarrow counter_b + 1$
26   **else if** *quantum number of a_2 at $counter_a$ > quantum number of b_2 at $counter_b$* **then**
27         $jumpBack_{already_set} \leftarrow$ false
28         $counter_b \leftarrow counter_b + 1$
29   **else**
30         $jumpBack_{already_set} \leftarrow$ false
31         $counter_a \leftarrow counter_a + 1$
32 set the isIndexOutgoing and quantumNumbers of $\widehat{C}$ to the corresponding values of a_1 and b_1

Figure 2.1.: The algorithm, used in the base implementation of multiply (*multiply_V0(...)*).

Factorization

The last important operation is the factorization of matrices. The most common factorizations are Singular Value Decomposition (SVD) and Spectral Decomposition. In this section, we will discuss the theory behind the idea of using SVD with quantum numbers. The current implementation includes some basic functionality for these operations but does not contain the complete and correct factorization yet.

$$A = U \cdot \Sigma \cdot V^T \tag{2.7}$$

Firstly, we have to check, whether the Tensor is of rank 2 or not. Because of this requirement, we will from now on call the tensor a matrix. We then can calculate the SVD for the given tensor. The result will be the three matrices U, Σ and V^T (2.7). Similar to the contraction method above, we can divide our matrix into different quantum number blocks. However, we can first calculate the left- and right-symmetric matrix (2.8). When the original tensor has the dimension M x N, the symmetric matrices will have the dimensions M x M and N x N. Additionally, these matrices will have the corresponding quantum numbers. The Matrix U and the Matrix V, which will be transposed, will have the same dimensions and quantum numbers as the left- and right-symmetric matrices. As the Matrix Σ has the same dimensions as the origin matrix according to convention, this will also be the case for our tensor class. Furthermore, Σ will have the same quantum numbers as the origin matrix.

$$Sym_L = A \cdot A^T, \qquad Sym_R = A^T \cdot A \tag{2.8}$$

After calculating the symmetric matrices and setting up the quantum numbers for the resulting matrices, we can divide the two symmetric matrices into the different blocks. These two groups of blocks can be divided into three different classes, blocks that only exist in the left-symmetric matrix, blocks that only exist in the right-symmetric matrix, and blocks, that exist in both. For the first two classes, the blocks will always be empty, as the quantum numbers previously only existed along one index.

For the last class, we calculate for each of the two groups the eigenvalues. Based on these eigenvalues the eigenvectors, and then write these vectors into the corresponding result matrix, We can see the decomposition for a specific block in 2.9. The positions of the values in the resulting matrices will only be in the corresponding block. Additionally, we compare the eigenvalues of both groups afterward and calculate the singular values. These can then be inserted into the diagonal of the blocks in the Σ matrix.

Lastly, we transpose the V matrix and have 3 resulting matrices as the output of the operation. Unfortunately, this procedure will result in a non-diagonal matrix Sigma, which does not comply with the usual convention for SVD of matrices without quantum numbers.

Eigenvalues e_L to Sym_L, Eigenvalues e_R to Sym_R
Eigenvectors v_L to e_L, Eigenvectors v_R to e_R
Singular values σ

$$U_x = \begin{bmatrix} | & | & | \\ v_L^1 & ... & v_L^m \\ | & | & | \end{bmatrix}, \Sigma_x = \begin{bmatrix} \sigma^1 & & \\ & \ddots & \\ & & \sigma^i \end{bmatrix}, V_x^T = \begin{bmatrix} — & v_R^1 & — \\ — & ... & — \\ — & v_R^n & — \end{bmatrix} \tag{2.9}$$

In detail, we need different functionality in our program for this process. Firstly, we need a way to approximate the eigenvalues efficiently and afterward calculate the eigenvectors to these values. For the approximation of the eigenvalues, the QR algorithm was used. Based on this algorithm, a matrix can be factorized via the QR decomposition. These matrices can then be combined in reverse order. This step will then be repeated until we achieve an acceptable approximation of eigenvalues on the diagonal. For the QR decomposition, three different methods were tested. Firstly, the standard Gram-Schmidt process did approximate the QR decomposition to a specific extent, but because of the normalizations of the vectors during the process, the resulting eigenvalues in the QR algorithm converged to values that were off by a few hundredths. Afterwards, the Modified Gram-Schmidt process was implemented. However, even with this process, we were not able to get reliable results for the tests. After testing the approximation with Householder transformations to find the QR decomposition, we were able to get reliable and precise results.
The eigenvectors were calculated based on the eigenvalues. The values were sorted and each eigenvalue was subtracted from the corresponding diagonal elements in the block matrix. The resulting linear system of equations was solved using Gaussian elimination.

2.2.2. Algorithmic Improvements

Contract

The algorithm of the contraction method can be changed slightly in this version, from now on also referred to as "multiply_V0_2". As we know from the base implementation, both, the quantum numbers and the coordinates of the contracted index need to match. The algorithm could be divided based on both requirements. However, dividing it based on the quantum number blocks is not only more intuitive but will also result in a less complex implementation. Furthermore, dividing the algorithm by the coordinates might likely result in a high amount of relatively small blocks.

Because we want to divide our space of elements based on the quantum numbers, we create a map for each tensor to put the different blocks in. Afterward, if a block exists for a number in both maps, we can apply a routine on the corresponding blocks and combine the results of all of these routines. We have multiple options for this routine.

Option 1: We can implement the routine similar to the base implementation algorithm. With this approach, we would have to order the elements based on their coordinate. Afterward, we can again use two pointers to the elements of each tensor. We iterate over the elements in the same way as we do in the base implementation. However, instead of moving the pointers based on the quantum numbers, we do it based on the coordinates of the elements. The disadvantage of this option is the need for sorting. Although, the advantage of having sorted elements could make up for it. A more detailed description of the algorithm is depicted in algorithm 3.

Option 2: The second option would be less complicated. Instead of sorting the two element groups, we ignore the order and for each element in the first group, we iterate over all elements of the second group. Depending on the number of elements in these blocks, it might even be more efficient to not have the overhead of sorting the groups beforehand. The algorithm is illustrated in algorithm 2.

After the routines, we add the elements of all blocks, which have the same coordinates in the resulting tensor, and continue to build the new tensor in the same way as in the base implementation.
The algorithm currently used for version "_V0_2" is depicted in algorithm 4.

Algorithm 2: Subroutine option 1

Input: Two blocks $block_A$ and $block_B$ with indices a_1, a_2 and b_1, b_2
Output: A map $table_C$

```
1  counter_a ← 0
2  counter_b ← 0
3  jumpBack ← 0
4  jumpBack_already_set ← false
5  block_A ← block_A sorted by coordinates of a_2
6  block_B ← block_B sorted by coordinates of b_2
7  while counter_a < size of block_A & counter_b < size of block_B do
8      if coordinates of a_2 and b_2 are equal for the entries at counter_a and counter_b
         then
9          if not jumpBack_already_set then
10             jumpBack_already_set ← true
11             jumpBack ← counter_b
12         add block_A.[counter_a] × block_B.[counter_b] to entry in table_C
13         if element after counter_b in block_B has the same quantum number then
14             counter_b ← counter_b + 1
15         else if element after counter_a in block_A has the same quantum number then
16             counter_a ← counter_a + 1
17             counter_b ← jumpBack
18         else
19             counter_a ← counter_a + 1
20             counter_b ← counter_b + 1
21     else if quantum number of a_2 at position counter_a > quantum number of b_2 at
         position counter_b then
22         jumpBack_already_set ← false
23         counter_b ← counter_b + 1
24     else
25         jumpBack_already_set ← false
26         counter_a ← counter_a + 1
```

Figure 2.2.: The algorithm describes the first option for the subroutine for multiplying two blocks.

Algorithm 3: Subroutine option 2

Input: Two blocks $block_A$ and $block_B$ with indices a_1, a_2 and b_1, b_2
Output: A map $table_C$

1 **for** *entry*$_a$ *in block*$_A$ **do**
2   **for** *entry*$_b$ *in block*$_B$ **do**
3   **if** *coordinates of* a_2 *for entry*$_a$ = *coordinates of* b_2 *for entry*$_b$ **then**
4   add $entry_a \times entry_b$ to entry in $table_C$

Figure 2.3.: The algorithm describes the second option for the subroutine for multiplying two blocks.

Algorithm 4: Multiply improved implementation algorithm

Input: Two Tensors $\widehat{A}$ and $\widehat{B}$ with indices a_1, a_2 and b_1, b_2
Output: The new Tensor $\widehat{C}$ with indices c_1 and c_2

1 $block_map_A \leftarrow \widehat{A}$.positionTable entries hashed by quantum number of a_2
2 $block_map_B \leftarrow \widehat{B}$.positionTable entries hashed by quantum number of b_2
3 **for** *block*$_A$ *in block_map*$_A$ **do**
4   **if** *block_map*$_B$ *contains correspondence to block*$_A$ **then**
5     $block_B \leftarrow block_map_B$ with same key as $block_A$
6   **while** $counter_a <$ *size of* $\widehat{A}$*.positionTable* & $counter_b <$ *size of* $\widehat{B}$*.positionTable* **do**
7   **for** *entry*$_a$ *in block*$_A$ **do**
8   **for** *entry*$_b$ *in block*$_B$ **do**
9   **if** *coordinates of* a_2 *for entry*$_a$ = b_2 *for entry*$_b$ **then**
10   add $entry_a \times entry_b$ to entry in $\widehat{C}$

11 set the isIndexOutgoing and quantumNumbers of $\widehat{C}$ to the corresponding values of a_1 and b_1

Figure 2.4.: The overall algorithm, used for the version *multiply_V0_2(...)*

2.2.3. Multithreading Improvements

Permute

In the permute operation, we can add multithreading, when we iterate over the elements of the *positionTable*. For each entry, we have to apply the same operation and change the values of the coordinates. As we change different areas of the *positionTable* in each entry, the calculation does not require a specific order of the elements. Therefore, the entries can be modified in different threads.

The first and simple approach, from now on called permute_V1, is to create a new thread for each element. However, as we will see later in the result section, this creates a massive overhead. There are possibly many entries in a tensor and therefore many threads that we would create. Moreover, the number of calculations for each element is rather small.

Instead, in the version permute_V1_2 we do not create threads for each entry, but divide the space of the *positionTable* into multiple subareas. If we have less than 16 entries, we handle the table the same way as in the base implementation. When there are more or equal to 16 and less than 64 entries in *positionTable*, the table is split into two equal parts. In bigger cases, which will be mostly the case, the table is split into four parts. In these examples, we only create a thread for each subspace. This should reduce the overhead of the thread handling.

Fuse

The fuse method includes multiple loops over different variables. However, in all of these loops, some kind of variable is modified. This mostly excludes them from multithreading, as we can not safely access the variables at the same time and would need to implement a locking mechanism for the variable. Even though the implementation of such a mechanism would not be problematic, threads would try to access the variable frequently and therefore spend a significant amount of waiting.

Split

The split method does not offer much opportunity for improvements with multithreading either. The first possibility would be the check, whether the quantum numbers combined, result in the current quantum number values. However, every iteration only needs a constantly small amount of operations. Therefore, the overhead for the multithreading will most likely exceed the benefits of the parallelization itself.

The second loop over the *positionTable* adds coordinates to the structure. For this reason, the loop should not access this variable in multiple threads at the same time. The result is an unsuitable loop for our multithreading approach.

Contract

The contraction method offers multiple different opportunities for multithreading. The reason behind it is the attribute of the operation itself, that we can look at the different blocks separately.
However, one might realize that this does not seem suitable for the base implementation. Therefore, we use the already algorithmic optimized implementation as a basis.

We also include a mutex for the threads, so that the elements can be saved in one structure throughout all threads. For each quantum number block, we create one thread. In the threads itself, we went back to apply a similar process to the block as in the base implementation, except we do not compare quantum numbers anymore. Specifically, the versions multiply_V1 and multiply_V1_2 implement the algorithm 3 in the subroutine and only differ in detail when the calculated elements are added to the element structure across the threads. In the first one, each element that is calculated is immediately added to the structure. The mutex should therefore only be blocked for a short period by one thread but locked many times. In the second version, the calculated elements are stored in the thread temporarily. When all calculations for the block are done, the mutex is locked once and all calculated elements are added to the overall structure. For this reason, the mutex is locked longer by one thread but only once. The version multiply_V1_3 on the other hand implements the algorithm 2 in the subroutine. Here, the overall structure is also accessed immediately after calculating an element.

2.2.4. Utilizing OpenMP

As an alternative to handling the threads manually, versions were added that use OpenMP to parallelize some parts of the operations. We apply the parallelization at the same places, where we used the manual multithreading previously.

For the permute method, we use "#pragma opm parallel for" for the loop, where we iterate over the *positionTable* and calculate the new coordinates.
We also add a version with OpenMP parallelization to the fuse and split methods. For fuse, we add "#pragma omp parallel for" for the loop, when iterating over the *positionTable* and calculating the coordinates to the new format. Additionally, we mark the section where we change the structure of *positionTable* with "#pragma omp critical". The same will be done for the split method. We add "#pragma omp parallel for", when iterating over the quantum numbers to check for equality with the current quantum numbers. Furthermore, we again limit the section where a variable from outside the loop is changed with "#pragma omp critical".

For the contraction method, we use the algorithm, where we already distinguish between the quantum number blocks. With this base, we add again "#pragma omp parallel for" to the loop, where we iterate over the blocks to calculate the subroutines.

All of the versions that use OpenMP, will from this part be marked with "_V3" in the name of the method.

2.2.5. SIMD Improvements

Single instruction, multiple data (SIMD) could also offer some advancement in performance. The overhead when using SIMD intrinsics may not be as big, compared to using multithreading. However, due to the structure of the tensor class, it does not seem suitable for usage in the four primary operations. In the permute method, the same routine is performed on all elements of *positionTable*. However, the values that the steps are performed on are not consecutive in the memory. This is similar to the fuse method, where we additionally change the structure of the *positionTable* variable itself in each step.
The split method would be suitable to include SIMD functionality. The quantum numbers

that we check are aligned in the memory. The overhead of using SIMD in this situation, even if small, may minimize its advantage though.
Contrary, the contraction method does not offer the visible opportunity for using this principle in this class structure. Especially, the fact that the complex values are separated from the coordinates complicates possible use.

Part III.

Results and Discussion

3. Results

3.1. Benchmarking

3.1.1. Testing details

Hardware environment

The tests in this chapter were run on a computer with the following specifications. During the testing, no other major tasks were executed.

CPU: AMD Ryzen 9 5900X 12-Core Processor, 3.70 GHz
RAM: 32 GB, 3.6 GHz
OS: Windows 10 Pro

Software environment

All operations were applied on tensors of rank 2. The value for the non-zero elements is set to 2.0 for the tests. Additionally, the quantum numbers for both indices were generated with the srand() function and additional seeds. These pseudo-random quantum numbers are all integers between -10 and 10.

Additionally, the results of the tests are the average execution time over 15 runs of the same method for each tensor that is included. Even though, we look at this limited fraction of tensors, it is used to represent the different performances of the operations versions. As we also use CMake to compile the project, we will use the release mode for compiling the files for the tests.

Figures

In all the given figures of this chapter, the vertical axis describes the average execution time of the runs in μs. The horizontal axis shows the possible amount of elements in the Matrix, not taking the symmetry into account. A value of 2500 for example, would mean the calculation is done with a 50 x 50 Matrix.

The figures 3.3, 3.5, 3.9 and 3.10 additionally include error bars for the data points, representing the standard error of the runs.

3.1.2. Permute

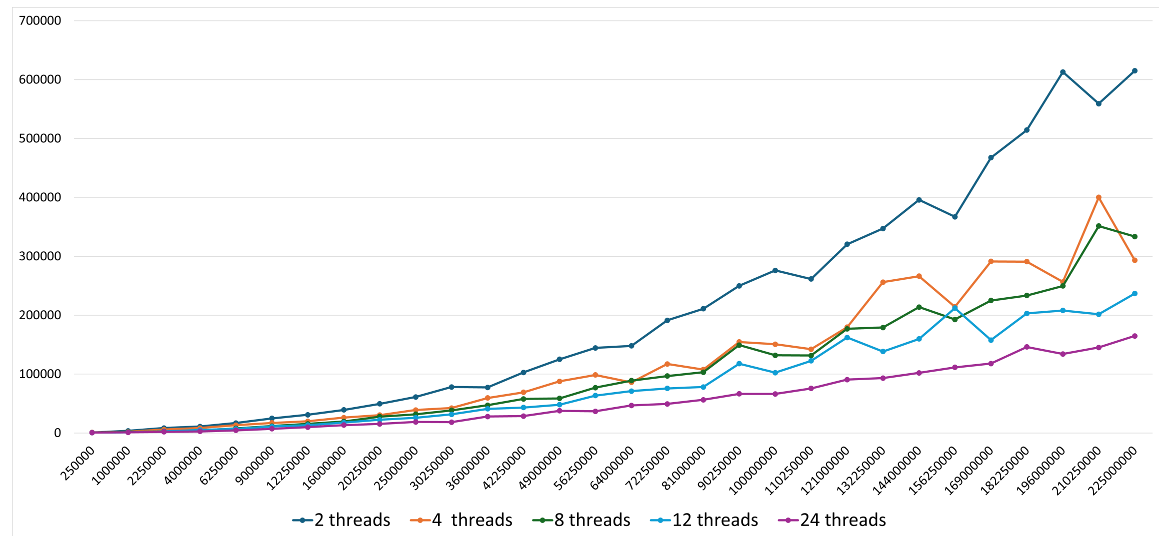


Figure 3.1.: The results from testing the OpenMP version of permute (*permute_V3(...)*) with different numbers of threads.

As described, we have four different versions of the permute method. The base implementation, two implementations with manual multithreading and one with the use of OpenMP. In the first figure 3.1, we see how the method using OpenMP performs with different numbers of threads. We choose 24 as the maximum, as the processor has 24 logical processors. We can observe in the graphic, that the more threads are used, the faster the method is. In the second figure 3.2, we compare all the different alternatives of the permute method. However, we already see that the second version needs immensely more time to execute than the other versions. This is most likely due to the overhead of creating threads for every iteration in the loop. For this reason, we disregard it. Lastly, in the third figure 3.3, to get a better comparison between the other methods, the second version is excluded. We can see that the second multithreading version performs better for large tensors. Even though this option is similar to the first multithreading one that we excluded, this time we only have a maximum of four simultaneous threads. This does not create as huge of an overhead as the second version. Lastly, the OpenMP version performs best on large tensors, which indicates that the multiple threads can be utilized properly.

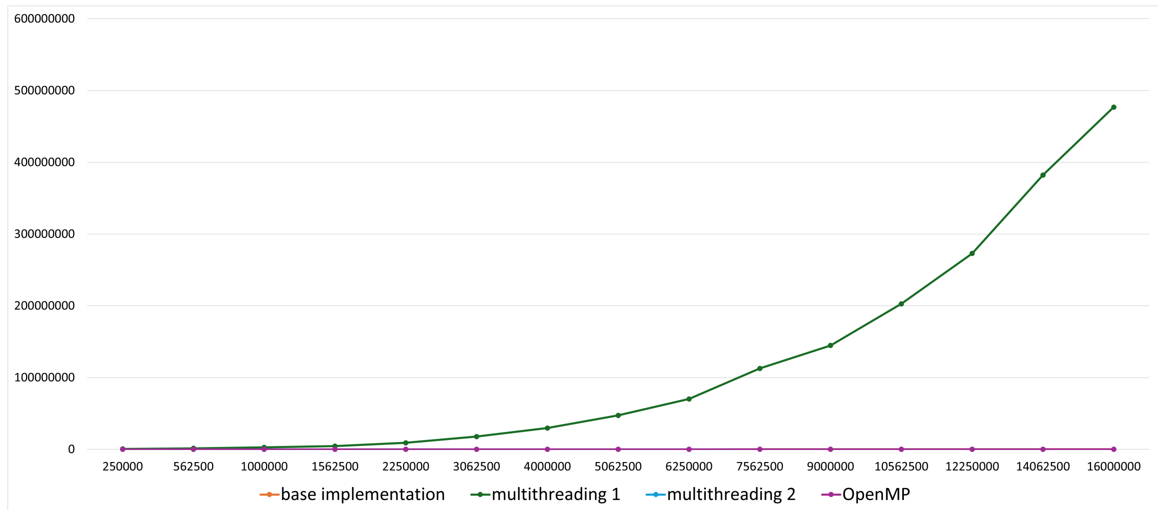


Figure 3.2.: The results from testing the base implementation (*permute_V0(...)*), the first and second multithreading versions (*permute_V1(...)* and *permute_V1_2(...)*), and the method using OpenMP with 24 threads *permute_V3(...)*. The graphs of the base implementation and the second multithreading can not be seen in the picture, as they overlap with the OpenMP version.

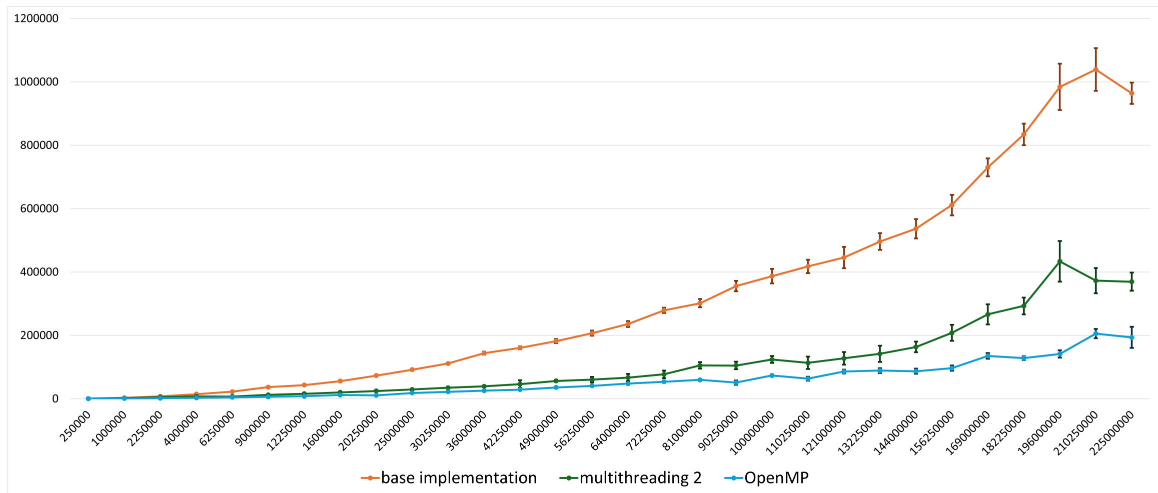


Figure 3.3.: The comparison of the base implementation (*permute_V0(...)*), the second multithreading (*permute_V1_2(...)*) and the OpenMP version (*permute_V3(...)*), excluding the first multithreading option(*permute_V1(...)*).

3.1.3. Fuse

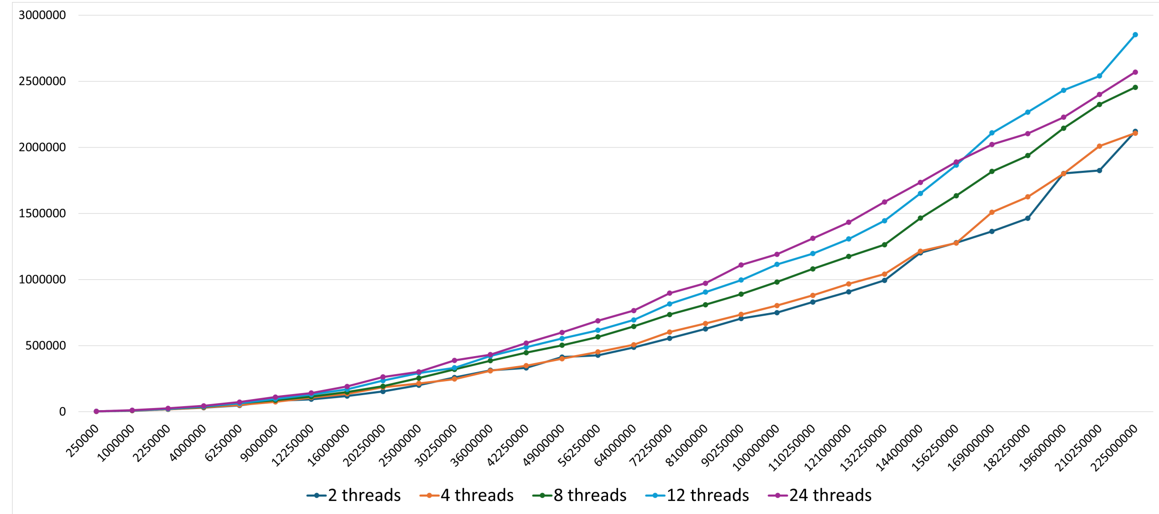


Figure 3.4.: The results from testing the OpenMP version of fuse (*fuse_V3(...)*) with different numbers of threads.

Firstly, we tested the OpenMP version with different amounts of threads again (figure 3.4). This time, the method is more efficient the fewer threads we use. As two threads are our minimum, this can already imply, that the method would perform better with only one thread. For this reason, we compare in figure 3.5 the performance of the OpenMP version with just two threads and the base implementation. As we can see, the method works best without any multithreading. This is most likely due to the threading overhead, for the relatively small amount of calculations per loop iteration, where the threads are used.

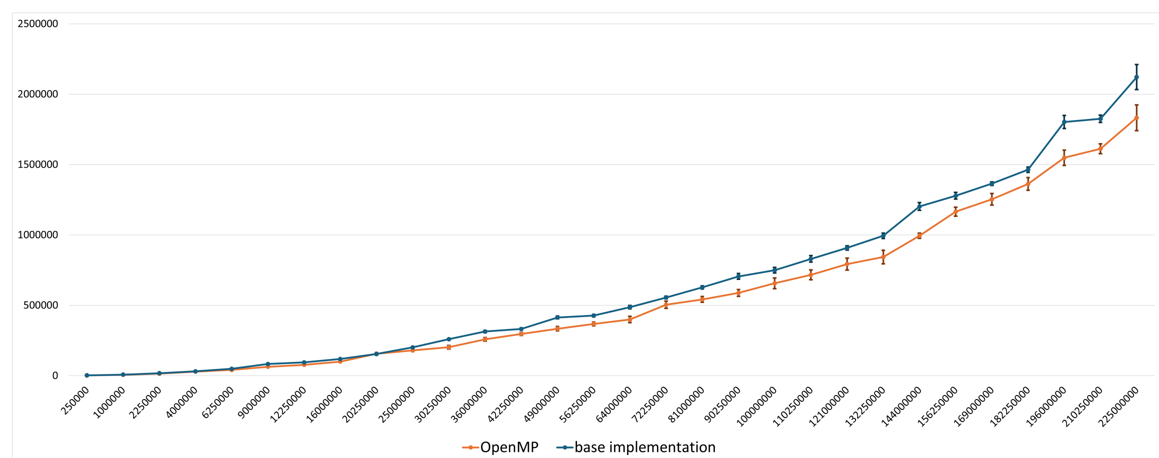


Figure 3.5.: The comparison of the base implementation (*fuse_V0(...)*) and the OpenMP version (*fuse_V3(...)*) with two threads.

3.1.4. Split

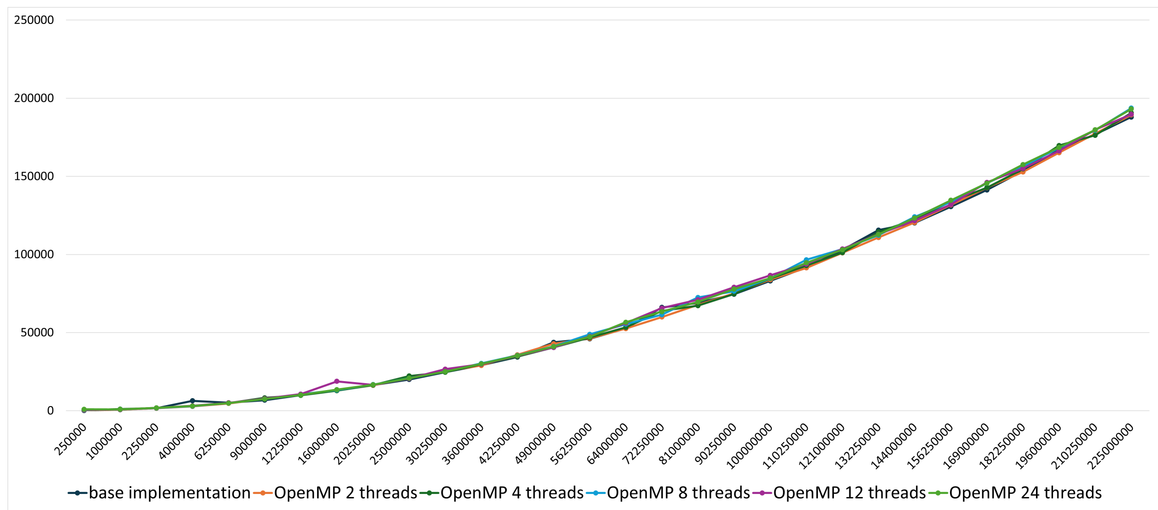


Figure 3.6.: The results of comparing the base implementation with the OpenMP version. The OpenMP version is divided into five options, corresponding to the used number of threads. Graphs that are not visible, are overlapping with the other ones.

For the split method, we included all options in figure 3.6. As we can observe, there is no significant difference in the performance of the different versions.
The OpenMP method also performs the same, no matter what amount of threads are used in our example. This indicates, that even though multiple threads are available, they are not used by the program. The reason for this is most likely the structure of the for-loop that should be parallelized. Therefore, we again regard the base implementation as the best option.

3.1.5. Contract

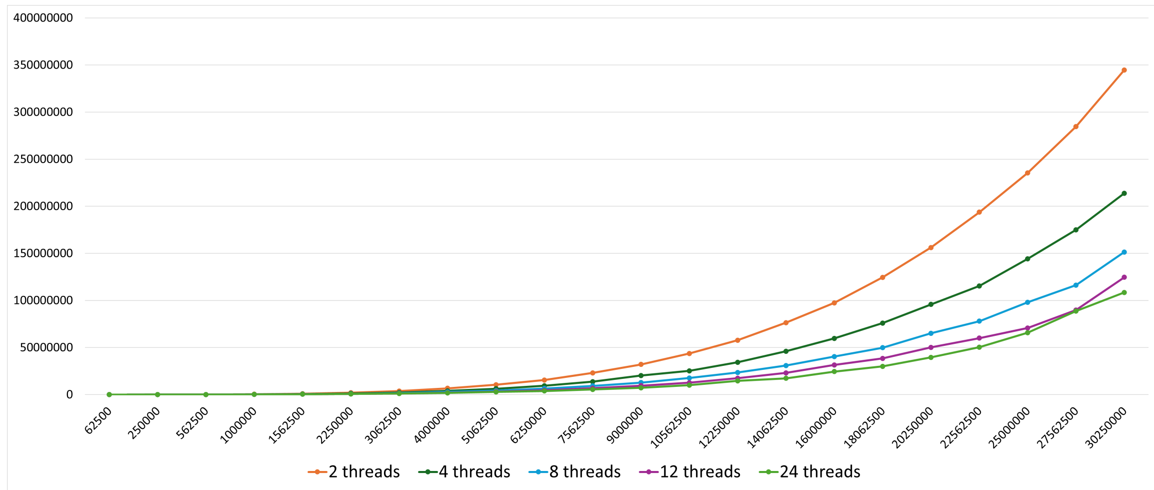


Figure 3.7.: The results from testing the OpenMP version of multiply (*multiply_V3(...)*) with different numbers of threads.

Firstly, we test again how the OpenMP version performs with different numbers of threads. In figure 3.7, we see similar results to the permute method using OpenMP, the more threads are used the more efficient the operation is. Again, this indicates that the different threads can be utilized properly and do not seem to have a problematic overhead.

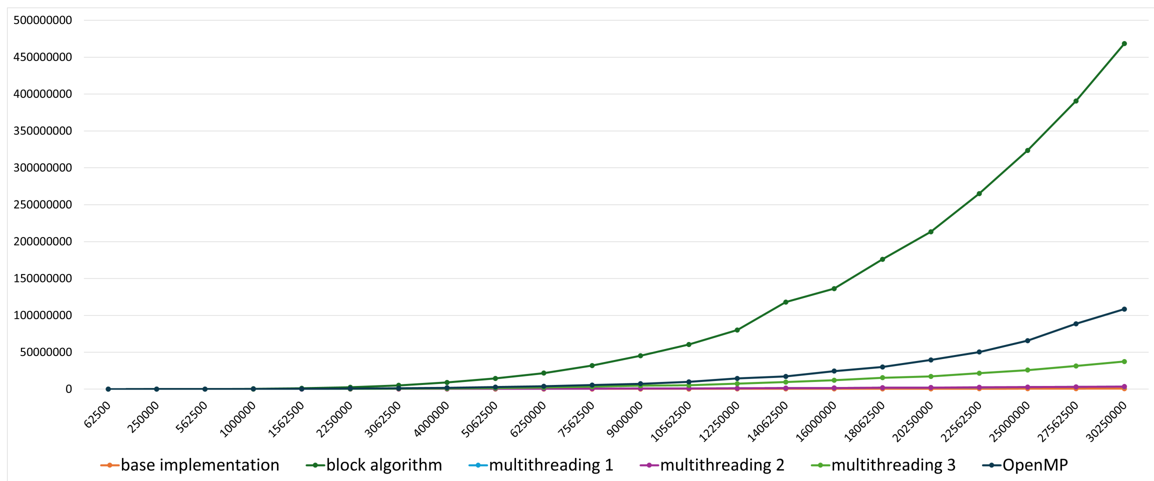


Figure 3.8.: The results from testing the base implementation (*multiply_V0(...)*), the block utilizing one (*multiply_V0_2(...)*), the multithreading versions(*multiply_V1(...)*, *multiply_V1_2(...)* and *multiply_V1_3(...)*), and the method using OpenMP with 24 threads *multiply_V3(...)*.

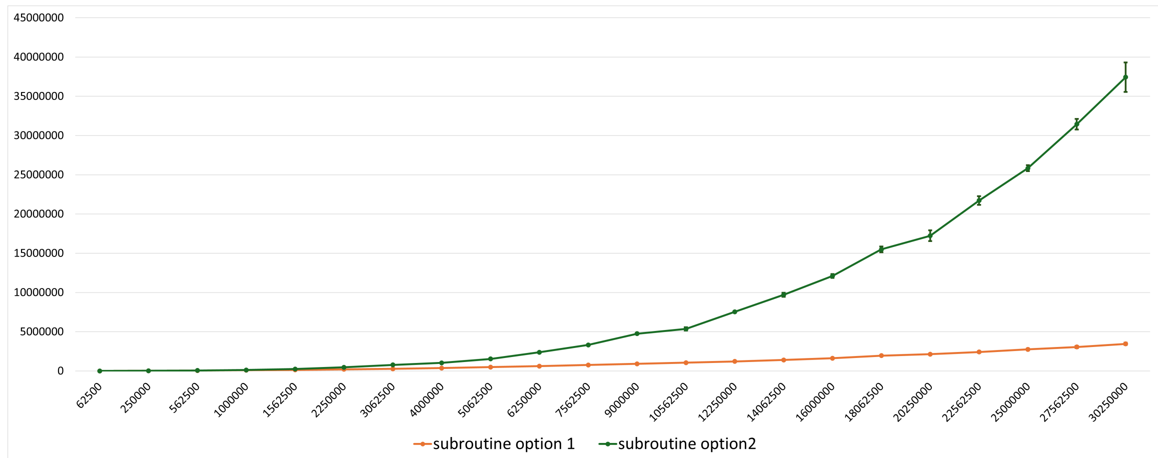


Figure 3.9.: The results from testing the multithreading version of multiply (*multiply_V1(...)* and *multiply_V1_3(...)*), as they implement the different subroutines 3 and 2.

Afterward, we can compare all the different versions of the multiply operation in figure 3.8. The worst-performing option is the algorithmic improved method. Even though we divide the operation into different blocks, the subroutine used is fairly bad and we need to use multiple threads to decrease the execution time.
The OpenMP version is not performing as badly but also seems to be a bad option for larger tensors. In figure 3.9, we can compare the two subroutines, discussed in more detail in algorithms 3 and 2. We see that the first version is more efficient in comparison. The second option may be simple in the implementation, but we do not use any algorithm to our advantage and iterate over all elements. It utilizes a similar algorithm of the base implementation and is therefore more complex, but also more efficient.
Lastly, we can compare the versions *multiply_V0(...)*, *multiply_V1(...)* and *multiply_V1_2(...)* in figure 3.10, to get a better contrast between the best versions so far. As seen in the graphic, the first and second multithreading versions perform nearly the same. This is expected, as they implement the same algorithm and only differ in the access of the shared variable as discussed in section 2.2.3. Lastly, the base implementation still performs the best.

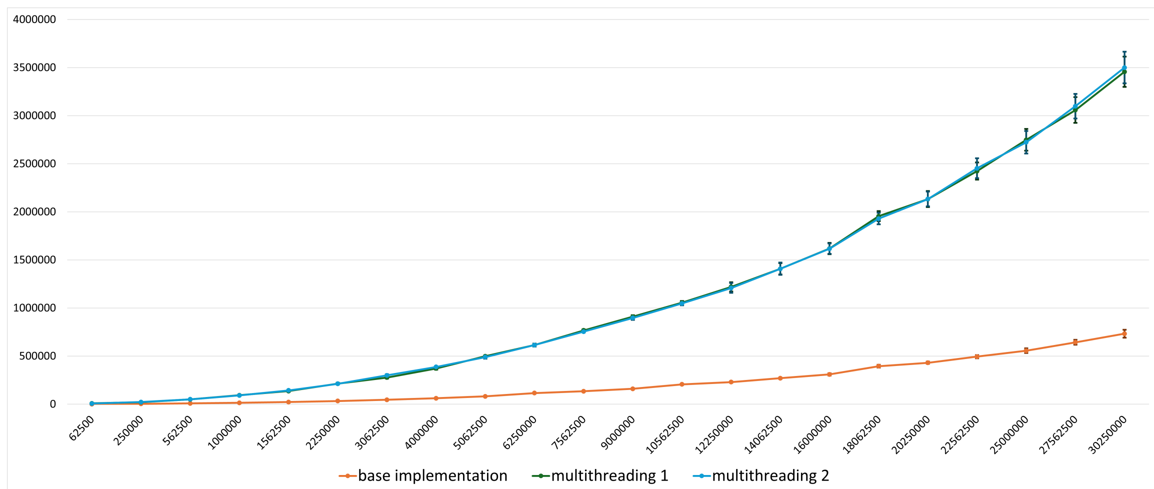


Figure 3.10.: The comparison between versions *multiply_V0(...)*, *multiply_V1(...)* and *multiply_V1_2(...)*. The graph of the first multithreading version is not visible clearly, as it overlaps with the graph of the second multithreading option.

4. Discussion

After looking into the testing results, we see, that the base implementations are still the best performing in nearly all of the methods.

With two very common optimization techniques in mind, using single instruction, multiple data (SIMD) functionality, or creating multiple threads, we tried to improve the base implementations. However, using SIMD seemed not suitable for most operations due to the underlying structure, constructed for sparse tensors. In these cases, the data for which we could apply the same instructions, was not as well aligned as we might want it to be. Moving the data into a bigger vector to perform a given instruction, might be less effective than just applying the instructions to the different values separately. Especially, because we did not use complex calculations in these operations. Using multiple threads at the same time, on the other hand, is more convenient to implement with the specific structure of the tensor class. However, depending on the decisions on how to implement the multithreading, the overhead varies and therefore, the outcome of the specific results will be different. Using OpenMP to simplify the parallelization process may be a good idea, on the other hand, using manual multithreading has been more efficient in the multiply operation, as we were able to better cut it to the requirements of the different operations.

Even though the implementation might build a solid base for future improvement, there are also limitations to it currently. One of them is the missing support of rank 0 tensors or scalars. For many of the operations, this would require additional functionality, specifically for this edge case. Regardless, most, if not all calculations would be based on tensors of a higher rank. Additionally, as discussed in section 2.2 "Factorization", the current implementation still needs a reliable and suitable approach for calculating singular value decomposition and spectral decomposition.

Despite the current limitations, the library offers a good approach to the calculation with sparse tensors with quantum number conservation. It provides users of C++, with the opportunity to implement algorithms based on the tensors with underlying U(1) symmetry.

5. Conclusion

In conclusion, we implemented a well-suited library for sparse tensors with quantum number conservation, which includes major functionalities for algorithms in a tensor network. Even though the operations on the tensors may be suitable for even further improvements, our tests showed that the resulting profit depends heavily on how the optimizations are designed and implemented.
While the outcome of some performance tests may seem discouraging, it indicates that we have built a good basis to improve the current methods and add new functionality.
Overall, during the development process, one goal was to create a library that is simple and intuitive to use, which will be an advantage during the use and future improvements.
However, even though this project offers a good basis, there are still many opportunities for further developments, tests, and changes. Firstly, one major aspect would be the elimination of the current limitations. Furthermore, after receiving feedback from users, other methods could be added that are helpful for the use of the tensor class. The improvements in the different methods also need to be extended and tested. Especially the multiplication and decomposition operations are expensive in regards to computation time.
Even though this would be a major change to the project, the performance of the program could be tested, if we slightly change the structure of the tensor class. That way, SIMD use could be more suitable.
Lastly, other symmetries could also be added to the projects, e.g. the SU(2) group. These changes, despite going in different directions, could contribute positively to these functionalities, implemented in the project.

Part IV.

Appendix

List of Algorithms

1. Multiply base implementation algorithm 14
2. Subroutine option 1 18
3. subroutine option 2 19
4. Multiply improved implementation algorithm 19

List of Figures

1.1. Tensors . . . 3
1.2. Tensor operations in mathematical notation . . . 3
1.3. Tensor operations . . . 4
1.4. Tensor network states . . . 4
1.5. Matrix with quantum numbers . . . 5
1.6. Indices with directions . . . 5
1.7. Quantum number block . . . 5

2.1. The algorithm, used in the base implementation of multiply (*multiply_V0(...)*). 14
2.2. The algorithm describes the first option for the subroutine for multiplying two blocks. . . . 18
2.3. The algorithm describes the second option for the subroutine for multiplying two blocks. . . . 19
2.4. The overall algorithm, used for the version *multiply_V0_2(...)* . . . 19

3.1. Permutation test for OMP version . . . 25
3.2. Permutation test on all version . . . 26
3.3. Permutation test with the best remaining versions . . . 26
3.4. Fuse test on the OMP version . . . 27
3.5. Fuse test on both versions . . . 27
3.6. Split test on all versions . . . 28
3.7. Multiplication test for OMP . . . 29
3.8. Multiplication test on all versions . . . 29
3.9. Multiplication test for subroutines . . . 30
3.10. Multiplication test on best remaining versions . . . 31

Bibliography

[AAB+15] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[BCOT11] B. Bauer, P. Corboz, R. Orús, and M. Troyer. Implementing global abelian symmetries in projected entangled-pair state algorithms. *Physical Review B*, 83(12), March 2011.

[CR23] Samantha Chen and Aditya Rajawali. Leveraging the power of quantum computing and machine learning to disrupt drug development. *Quarterly Journal of Computational Technologies for Healthcare*, 8(4):1–11, Dec. 2023.

[Pfe17] Wolfgang Pfeiler. *Band 5 Quanten, Atome, Kerne, Teilchen.* De Gruyter, Berlin, Boston, 2017.

[Rah22] Fazal Raheman. The future of cybersecurity in the age of quantum computers. *Future Internet*, 14(11), 2022.

[RMG+19] Chase Roberts, Ashley Milsted, Martin Ganahl, Adam Zalcman, Bruce Fontaine, Yijian Zou, Jack Hidary, Guifre Vidal, and Stefan Leichenauer. Tensornetwork: A library for physics and machine learning, 2019.

[SPV10] Sukhwinder Singh, Robert N. C. Pfeifer, and Guifré Vidal. Tensor network decompositions in the presence of a global symmetry. *Physical Review A*, 82(5), November 2010.

[SPV11] Sukhwinder Singh, Robert N. C. Pfeifer, and Guifre Vidal. Tensor network states and algorithms in the presence of a global u(1) symmetry. *Phys. Rev. B*, 83:115125, Mar 2011.

[ZSW20] Yiqing Zhou, E. Miles Stoudenmire, and Xavier Waintal. What limits the simulation of quantum computers? *Phys. Rev. X*, 10:041038, Nov 2020.