FRIEDRICH-SCHILLER-
UNIVERSITÄT
JENA

A Comparison of Different Algorithms for Sparse Einsum

BACHELOR THESIS

to be Awarded the Academic Degree

Bachelor of Science (B.Sc.)

in Informatics

FRIEDRICH-SCHILLER-UNIVERSITY JENA

Faculty for Mathematics and Informatics

Submitted by Leon Manthey

born on 31.10.1999 in Berlin

Supervisor: Mark Blacher

Jena, 26.08.2024

Abstract

Einstein notation is a powerful tool for representing tensor expressions in many scientific and machine learning applications. For many of these applications sparse tensor operations are crucial, yet surprisingly, there are few algorithms tailored for solving Einstein summation problems for sparse tensors. To address this shortcoming, we introduce a new approach that maps pairwise contractions to a sparse batch matrix multiplication kernel. We tested our method against three established approaches: the tensor library Sparse, Torch, and a SQL based approach. Our experiments on both real-world and synthetic benchmarks demonstrate that our algorithm outperforms others in solving sparse Einstein summation problems. It efficiently handles higher-dimensional tensors and successfully computes larger problems where other methods fail. These findings underscore the potential of our algorithm to enhance efficiency and robustness in sparse Einstein summation.

Zusammenfassung

Die Einstein-Notation ist ein mächtiges Werkzeug, das vermehrt zur Darstellung von Tensor-Ausdrücken in diversen wissenschaftlichen Kontexten sowie im maschinellen Lernen verwendet wird. Viele dieser Anwendungen operieren auf dünnbesetzten Tensoren (auch sparse Tensoren genannt). Trotzdem existieren wenige Algorithmen, die speziell für die Lösung von Einstein-Summationsproblemen mit dünnbesetzten Tensoren entwickelt wurden. Der von uns in dieser Arbeit vorgestellte Algorithmus adressiert den Mangel solcher Methoden. Dabei werden die Einstein-Summationsprobleme durch die Abbildung paarweiser Kontraktionen auf eine Batch-Matrix-Multiplikation für dünnbesetzte Tensoren berechnet. Vergleiche mit Sparse, einer Bibliothek für dünnbesetzte Tensoren, Torch und einem auf SQL basierenden Ansatz zeigen, dass unser Algorithmus effizienter arbeitet als diese drei bereits etablierten Methoden. Experimente mit aus realen Problemen entnommenen sowie synthetischen Benchmarks ergaben, dass unser Algorithmus Einstein-Summationsprobleme für dünnbesetzte Tensoren schneller berechnet, effizient mit höherdimensionalen Tensoren umgeht sowie größere Probleme lösen kann als die eben genannten drei Methoden. Diese Ergebnisse unterstreichen das Potenzial unseres Algorithmus, maßgeblich zur Effizienzsteigerung bei Einstein-Summationsproblemen für dünnbesetzte Tensoren beitragen zu können.

Contents

1 Introduction **9**

2 Background **11**
2.1 Tensors . . . 11
2.2 Einstein Notation and Einstein Summation . . . 12
2.3 Operations with Einsum . . . 13
2.4 Contraction of Tensor Hypernetworks . . . 13

3 Related Work **15**

4 Algorithms **17**
4.1 The SQL Algorithm . . . 17
4.1.1 Portable Schema for Tensors . . . 17
4.1.2 Mapping Einstein Summation to SQL . . . 17
4.1.3 Optimizing Contraction Order . . . 18
4.1.4 Implementation Details . . . 19
4.2 The C++ Algorithm . . . 19
4.2.1 Preprocessing . . . 19
4.2.2 Sparse BMM . . . 21
4.2.3 Postprocessing . . . 23
4.2.4 Implementation Details . . . 23

5 Experiments **25**
5.1 Einsum Benchmark Instances . . . 25
5.2 Random Tensor Hypernetworks . . . 26

6 Discussion **31**
6.1 Iterative Approach for Batch Matrix Multiplication and Index Reordering . . . 31
6.2 Precomputation of Diagonal Indices and Summed Indices . . . 31
6.3 Memory Footprint Optimization via Multi Index Encoding . . . 33

7 Conclusion **35**

1 Introduction

Einstein notation is a powerful and compact notation for representing tensor expressions. It was introduced by Albert Einstein in the early 20th century in order to simplify tensor expressions in "The Foundation of the General Theory of Relativity" [5]. Due to being brief but still comprehensive, Einstein notation has become a valuable tool in many fields such as physics, mathematics, and computer science.

The fundamental operation for evaluating tensor expressions presented in Einstein notation is Einstein summation, often referred to simply as "Einsum". This operation allows for the calculation of various tensor expressions, including element-wise multiplications, dot products, outer products, and matrix multiplications. The predominant reason for the adoption of Einsum notation in numerous applications, ranging from machine learning to scientific computing, is its conciseness.

In many practical applications, especially in machine learning and scientific computing, the data involved can be sparse. In sparse tensors most values are zero. Handling sparse tensors efficiently requires specialized algorithms and data structures to avoid unnecessary computations and to save memory. Traditional libraries like NumPy [7] and other major artificial intelligence frameworks [11, 13] typically support Einstein summation for dense tensors, but lack support for sparse tensors. The only known library that aims to support Einsum operations on sparse tensors is Sparse [15]. However, like NumPy, Sparse only allows for a limited number of symbols to be used as indices, which is why we use opt_einsum [16], a package for optimizing the contraction order of Einsum expressions. More importantly for us though, opt_einsum can handle UTF-8 symbols and use Sparse and other libraries like Torch as a backend. Real Einstein summation problems often include expressions with hundreds or even thousands of higher order tensors. In order to express the aforementioned operations we require a large set of unique indices. Thus, our approaches, just like opt_einsum, are capable of handling all symbols in the UTF-8 encoding.

This thesis explores the implementation and performance of Einstein summation across different computing paradigms, with a particular focus on sparse tensors. Specifically, it focuses on explaining our following implementations and comparing them to multiple libraries:

- SQL-based Implementation: This implementation is based on the algorithm presented in "Efficient and Portable Einstein Summation in SQL" by Blacher et al [1]. It constructs SQL queries dynamically using Python. While SQL is traditionally used for database operations, this approach demonstrates the ability of SQL in performing tensor operations.

- C++ Implementation: The second implementation is written in C++, with two versions, one naive and one optimized approach. The different versions aim to explore the performance trade-offs between simplicity and optimization, offering insights into how different coding strategies affect computational efficiency.

By comparing these implementations, our goal is to provide a comprehensive analysis of the performance and scalability of sparse Einstein summation in diverse computing environments. The SQL-based implementation serves as a baseline for comparing all approaches. It showcases the potential of database query languages for tensor operations. Furthermore, the C++ implementations show how low-level optimizations impact computational performance. The code for our implementations is available on GitHub at: https://github.com/Lethey2552/Sparse-Einsum.

We aim to determine the advantages and disadvantages of each approach by comparing our implementations against the sparse tensor library Sparse and highly performance-tuned dense tensor libraries like Torch. This will help identify which approach is best for a given set of use cases and computational requirements. The goal of this effort is to provide researchers and practitioners in disciplines that significantly rely on tensor calculations with practical insights by expanding the understanding of tensor operations and their effective implementation.

2 Background

The following chapter serves to introduce the necessary background for tensors, Einstein notation and Einstein summation. Furthermore, we will provide various examples for operations that can be expressed using Einstein notation. Given the considerable overlap in topics, we will build on related literature [5], adapting and expanding it to meet our specific research requirements.

2.1 Tensors

Tensors are algebraic objects and a fundamental concept in mathematics, physics and computer science. They extend the idea of scalars, vectors and matrices to higher dimensions. In essence, a tensor is a multi-dimensional array with an arbitrary number of dimensions. Each dimension of a tensor is represented by an index that spans the size of the dimension. The number of dimensions is commonly referred to as the tensor's "rank" or "order." The size of a tensor is determined by the product of the maximum values of each index's range.

For example, consider a tensor T with indices i, j, k and corresponding ranges $i \in \{1, 2\}$, $j \in \{1, 2, 3, 4, 5, 6\}$ and $k \in \{1, 2, 3, 4\}$. The size of tensor T is calculated as follows: $2 \cdot 6 \cdot 4 = 48$. This means tensor T has a total of 48 elements. An example of a matrix A with indices i, j and a tensor A with indices i, j, k, both represented as a graph, can be seen in Figure 2.1.

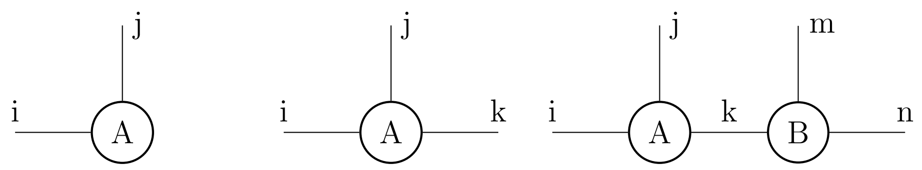


Figure 2.1: A matrix, a tensor and a tensor network visualized as a graph. Each index is represented by an edge. Shared indices of tensors in a tensor network are represented by edges between nodes.

In this work, a tensor is simply a multidimensional array containing data of a primitive type. We differentiate between dense and sparse tensors.

Dense Tensors. Dense tensors have a significant number of non-zero entries. However, there is no exact threshold which determines whether a tensor is dense or sparse. The density of a tensor, a metric indicating how many elements are non-zero compared to the total number of elements, is calculated as:

$$\frac{Number\ of\ non\text{-}zero\ elements}{Number\ of\ total\ elements}$$

Sparse Tensors. In Sparse tensors most values are zero. They can greatly profit from specialized formats. For our tensor $T \in \mathbb{R}^{I \times J \times K}$ in dense format we need to save $I \cdot J \cdot K$ values no matter whether they are zero or not. Now consider that, if the vast majority of T's values are zero, we could only save the coordinates of the non-zero values, that is the index of the value for each dimension. This is what we call the coordinate (COO) format. Each row of the COO representation encodes a single value of the tensor with each column holding the position of the value for the corresponding dimension and the last column giving the actual value. This can be done for an arbitrary number of dimensions by simply adding more columns for their respective coordinates. An example of a dense tensor (left) and its COO representation (right) could be the following:

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 5 & 0 & 0 & 10 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad \Rightarrow \quad \begin{bmatrix} 0 & 1 & 1 \\ 1 & 2 & 4 \\ 2 & 0 & 5 \\ 2 & 3 & 10 \end{bmatrix}$$

2.2 Einstein Notation and Einstein Summation

In 1916, Albert Einstein introduced the so called Einstein notation, also known as Einstein summation convention or Einstein summation notation, for the sake of representing tensor expressions in a concise manner. As an example, the following operation on the tensors $A \in \mathbb{R}^{I \times J \times K}$ and $B \in \mathbb{R}^{K \times M \times N}$ from Figure 2.1,

$$C_{ijmn} = \sum_k A_{ijk} \cdot B_{kmn}$$

can be simplified by making the assumption that pairs of repeated indices in the expression are to be summed over, as is the rule for tensor contractions. Consequently, the contraction can be rewritten as:

$$C_{ijmn} = A_{ijk} \cdot B_{kmn}$$

To expand upon the expressive power of the original Einstein notation, modern Einstein notation was introduced. This notation is used by most linear algebra and machine learning libraries supporting Einstein summation, that is, the evaluation of the actual tensor expressions. Modern Einstein notation explicitly states the indices for the output tensor, enabling further operations like transpositions, traces or summation over non shared indices.

In modern Einstein notation, the expression from the previous example would be written as:

$$A_{ijk}B_{kmn} \to C_{ijmn}$$

When using common Einstein summation APIs, tensor operations are encoded by using the indices of the tensors in a format string and the data itself. The format string for the above operation would come down to:

$$ijk, kmn \to ijmn$$

In Modern Einstein notation, indices that are not mentioned in the output are to be summed over. For the sake of simplicity, we will from now on refer to Einstein summation as Einsum, and we will use the original, the modern notation or just the format string, depending on the context. For expressions with two tensors we will call the first tensor the left tensor and the second tensor the right tensor.

2.3 Operations with Einsum

Einsum is a powerful tool for performing various tensor operations. Table 2.1 shows some common operations that can be performed using Einsum.

Table 2.1: Example operations with Einsum.

Operation	**Formula**	**Format string**
Dot Product	$c = \sum_i a_i b_i$	i,i $\to$
Sum Over Axes	$b_j = \sum_i A_{ij}$	ij $\to$ j
Outer Product	$C_{ij} = a_i b_j$	i,j $\to$ ij
Matrix Multiplication	$C_{ij} = \sum_k A_{ik} B_{kj}$	ik,kj $\to$ ij
Batch Matrix Multiplication	$C_{bij} = \sum_k A_{bik} B_{bkj}$	bik,bkj $\to$ bij
Tucker Decomposition [14]	$T_{ijk} = \sum_{pqr} D_{pqr} A_{ip} B_{jq} C_{kr}$	pqr,ip,jq,kr $\to$ ijk

These examples illustrate the versatility of Einsum in performing a wide range of tensor operations using a concise and readable notation, expressed as a format string. Note that while the examples provided are relatively simple, real-world Einstein summation problems may include thousands of tensors.

2.4 Contraction of Tensor Hypernetworks

Tensor contraction is the process of reducing one or multiple tensor's orders by summing over pairs of matching indices. Tensor networks where more than two tensors share an index are called tensor hypernetworks.

The contraction of the tensor hypernetwork $A \in \mathbb{R}^{I \times J \times K}, B \in \mathbb{R}^{K \times M \times N}$ and $C \in \mathbb{R}^{K \times L}$ in Figure 2.2,

$$T_{ijmnl} = \sum_k A_{ijk} \cdot B_{kmn} \cdot C_{kl}$$

in modern Einstein notation written as

$$ijk, kmn, kl \rightarrow ijmnl$$

can be calculated in different orders. Either way, it is possible to get the same result by contracting A and B first, followed by $(AB) \cdot C$, by contracting B and C and then $A \cdot (BC)$ or by contracting A and C, followed by $(AC) \cdot B$. While the result of the contraction orders will be the same, the underlying number of operations may differ vastly. As a result, the order in which tensors are contracted can drastically change the performance of an algorithm. We call this order the contraction path.

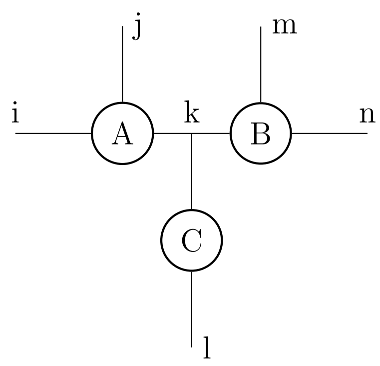


Figure 2.2: A tensor hypernetwork.

3 Related Work

Compared to the well-established methods for Einsum with dense tensors, Einstein summation with sparse tensors has received relatively little attention in the scientific community. Due to various tensor operations that can be expressed using Einsum notation, the underlying algorithms need to be able to handle many distinct computations. Here we introduce multiple approaches and ideas, contributing to the field of sparse Einsum.

Recent developments in integrating machine learning and linear algebra routines into databases have gained significant attention [3, 4, 9, 17]. One such approach is the translation of sparse Einsum problems into SQL queries [1]. The authors introduce four mapping rules and a decomposition scheme in which large Einsum operations are split into multiple smaller Einsum operations. In contrast to SQL-based approaches, the TACO compiler can translate known sparse linear algebra and tensor operations into optimized code directly [8]. While this produces optimized code for predefined problems with trivial contraction paths, it faces limitations in handling dynamic problems that are not known at compile time. TACO does not calculate an efficient contraction path, nor does it allow for the application of previously computed contraction paths. As a result, other methods, capable of using optimized contraction paths, outperform TACO, especially for large tensor expressions involving thousands of higher order tensors. Gray J. developed an Einsum function that calculates tensor expressions via a batch matrix multiplication (BMM) approach [6]. This method allows for the computation of pairwise tensor expressions by mapping them to BMMs, using summation over indices, transposition and reshaping of the tensors. A BMM approach for evaluating Einstein summation expressions is also employed by Torch within its tensor library Aten [13]. Sparse, a library designed for operations on sparse tensors, implements an Einsum function [15]. However, when used alone, Sparse struggles with large tensor expressions due to its limitations in handling a high number of different indices. This limitation can be overcome by using Sparse as a backend for opt_einsum, a package that optimizes tensor contraction orders. Sparse utilizes Numba [10] to accelerate calculations; Numba is a just-in-time compiler that generates machine code from Python syntax.

4 Algorithms

In this chapter we present two algorithms for performing Einstein summation. First, we introduce our implementation of the four mapping rules developed in "Efficient and Portable Einstein Summation in SQL" [1], to generate SQL queries for solving Einsum problems. This will serve as a baseline to compare other algorithms against. Second, we explain our C++ implementations with multiple levels of optimization. The underlying algorithm of the C++ implementations builds on Torch's strategy of mapping Einsum operations to a batch matrix multiplication kernel. Both algorithms, namely the algorithm for the SQL implementation and the algorithm used for the C++ versions, decompose large Einstein summation operations into smaller, pairwise operations to exploit efficient contraction paths.

4.1 The SQL Algorithm

In this section, we present Blacher et al.'s [1] algorithm and our implementation of it for mapping format strings and the corresponding tensors to SQL, enabling Einstein summation in databases. First, we introduce the portable schema for representing tensors, specifically sparse tensors, in SQL. We then show their four mapping rules to generate non-nested Einsum queries from arbitrary format strings. Next, we explain how we exploit efficient contraction paths by decomposing large Einsum queries into smaller parts. Finally, we present implementation details of the SQL algorithm.

4.1.1 Portable Schema for Tensors

Blacher et al. chose the COO format to represent tensors as it only uses integers and floating point numbers, which results in a vendor independent schema for encoding tensors across various database management systems (DBMS). For example, a 3D tensor $A \in \mathbb{R}^{I\times J\times K}$ has the following schema:

$$A(i\ \ INT,\ \ j\ \ INT,\ \ k\ \ INT,\ \ val\ \ DOUBLE)$$

Each tensor is stored in a separate table. In the example, table A stores a 3D tensor, where each value (val) can be addressed by specifying the corresponding indices (i, j, k).

4.1.2 Mapping Einstein Summation to SQL

"Efficient and Portable Einstein Summation in SQL" introduces four rules for mapping any tensor expression in Einstein notation to SQL.

R1 All input tensors are enumerated in the FROM clause.

R2 The indices of the output tensor are enumerated in the SELECT clause and the GROUP BY clause.

R3 The new value is the SUM of all values multiplied together.

R4 Indices that are the same among input tensors are transitively equated in the WHERE clause.

Say we want to map the tensor operation given by $ik, k \rightarrow i$, a matrix-vector multiplication, with tensors $A \in \mathbb{R}^{I \times K}$, $v \in \mathbb{R}^{K}$ and

$$A = \begin{bmatrix} 0.0 & 1.0 \\ 0.0 & 0.0 \\ 5.0 & 0.0 \\ 0.0 & 0.0 \end{bmatrix}, \quad v = \begin{bmatrix} 4.0 \\ 1.0 \end{bmatrix}.$$

When applying all four rules to map the example tensor expression to SQL, we get the result seen in Listing 4.1.

Listing 4.1: Einstein summation in SQL.

```sql
WITH A(i, j, val) AS (                          -- matrix A
    VALUES (0, 1, 1.0), (2, 0, 5.0)
), v(i, val) AS (                               -- vector v
    VALUES (0, 4.0), (1, 1.0)
) SELECT A.i AS i,                              -- R2
    SUM(A.val * v.val) AS val                   -- R3
  FROM A, v                                     -- R1
  WHERE A.j=v.i                                 -- R4
  GROUP BY A.i                                  -- R2
```

While all four rules are needed to ensure every possible Einstein summation problem can be translated to SQL, for some tensor expressions the conditions to apply the rules R2 and/or R4 are not fulfilled. If there are no indices after the arrow in the format string, the output is scalar and does not require R2. Furthermore, if there are no common indices among the input tensors, there is no summation in the tensor expression and R4 can be omitted. The rules guarantee a correct mapping, not a mapping with minimal code size. In some cases, with additional checks, the SQL queries could be further simplified.

4.1.3 Optimizing Contraction Order

Mapping a tensor expression directly into a single, non-nested SQL query in Einstein notation is known to produce execution times that are far from optimal, especially for operations involving many tensors. The inefficiency stems from the fact that conventional query optimizers are unaware of the contraction order of the repeating indices within tensor expressions and are therefore incapable of effectively breaking down the query into smaller parts to exploit efficient contraction paths as described in Section 2.4.

One can get around this using intermediate tensors via subqueries or common table expressions, which decomposes one large Einstein summation query into smaller pieces and lets the database engine follow a predefined contraction order. More precisely, using GROUP BY and SUM aggregation in intermediate computations enforces query engines to evaluate the query in the right order.

4.1.4 Implementation Details

We implemented the algorithm for mapping Einsum format strings to SQL queries proposed by Blacher et al. in Python 3.11.0 as a small package, only requiring Numpy as a dependency. When calling *sql_einsum_query()*, an Einsum notation string, the tensor names and the tensor data have to be supplied. The path argument is optional. When not supplied with a path, an optimized contraction path is calculated using cgreedy [12]. The cgreedy package provides a greedy algorithm approach for finding an efficient contraction order for any given format string and associated tensors, utilizing multiple cost functions. The construction of the query is separated into two parts. The first part creates the tensors in COO format as SQL compatible structures and returns the appropriate query. The second part applies the decomposition schema. More precisely, it uses either the supplied or the calculated contraction path to build a contraction list. The entries of the contraction list dictate the order and the exact pairwise operations necessary to solve the Einstein summation problem. These subproblems are also specified in Einstein notation. To build the second part of the query, we iterate the contraction list and apply the four mapping rules from Subsection 4.1.2 to assemble the correct SQL strings for the given pairwise contractions. Finally, we merge the two generated query parts and return the complete query.

4.2 The C++ Algorithm

Our method expands on Torch's strategy of mapping the Einstein summation problems to batch matrix multiplication (BMM). We give an overview of our approach that preprocesses tensors for pairwise operations, calculates the result via a sparse BMM and finally postprocesses the result to fit the expected output format.

4.2.1 Preprocessing

The preprocessing phase of the algorithm is critical for aligning the tensors in Coordinate List (COO) format with the requirements of batch matrix multiplication. In order to achieve predictable computations with the BMM, we apply the template seen in Figure 4.1. The indices of the two input tensors are grouped as follows:

$$C[\vec{b}, \vec{k_l}, \vec{k_r}] = \sum_{c, s_l, s_r} A[\vec{b}, \vec{k_l}, \vec{c}, \vec{s_l}] B[\vec{b}, \vec{c}, \vec{k_r}, \vec{s_r}]$$

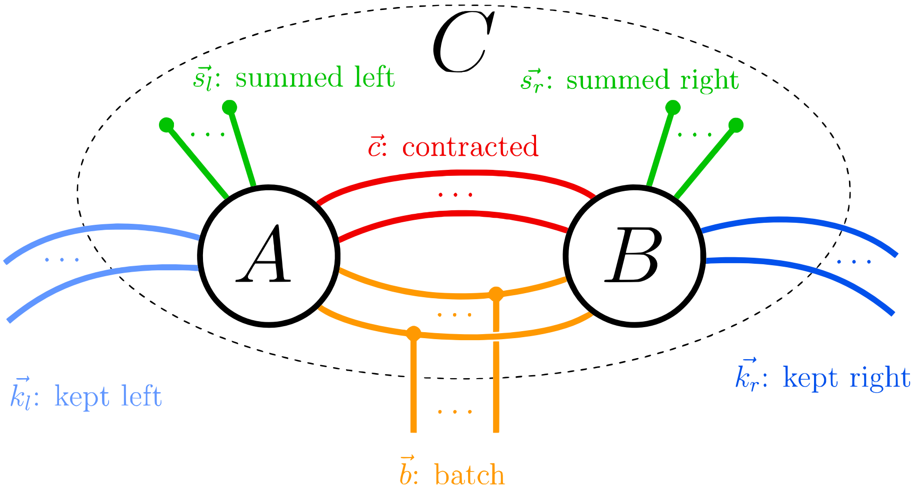


Figure 4.1: The figure used by Gray J. [6] to describe the classification of indices used for the grouping. The left tensor has the index order (b, k_l, c, s_l) and the right tensor (b, c, k_r, s_r).

Batch Indices (b) All individual batch dimensions have to be combined into a single batch dimension.

Contracted Indices (c) The dimensions over which the contraction takes place.

Summed Indices (s) All indices that only have a single occurence. Summed indices are grouped for both input tensors separately.

Kept Indices (k) All indices and their dimensions that occur in only a single input tensor and in the output. Kept indices are grouped for both input tensors separately.

For our implementation we treat the removal of the summed indices as part of the preprocessing. This means the input tensors for the BMM only have index order (b, k_l, c) for the left term and (b, c, k_r) for the right term. For the COO format, since every index results in a column, the index order translates to the columns, followed by a last column for the value. To enable the use of this format we apply multiple preprocessing steps. First, the algorithm calculates appropriate format strings for the COO tensors. The format strings should align with the requirements of the BMM to ensure correctness and efficiency of the subsequent operation.

Besides computing the format strings, the algorithm computes the new shapes for the tensors. In doing so, this step will combine all batch, contracted and kept dimensions into one single dimension respectively and ignore dimensions not present in the format string. Precise computation of these new shapes is critical to ensure that the tensors are aligned correctly for the BMM. The computed format strings are used to call a special, single Einsum function. This function performs Einstein summation on a single tensor, allowing for the computation of diagonals, summation over specified dimensions and the permutation thereof. The new shapes are used to reshape the tensors to comply with the dimensional requirements of the BMM. Both of these steps are only performed if necessary. The preprocessing will result in a COO tensor, which is suited for batch matrix multiplication.

4.2.2 Sparse BMM

The BMM computes the results in batches. In our case a batch only contains two elements. A COO tensors rows with the same batch index represent a two dimensional matrix in COO format. A batch contains the two matrices, one for each tensor, where the batch index is the same. Since the batch index of the matrices in a batch are the same and we compute the products within batches, we represent them without the batch index after selecting a batch. Say A and B are tensors with indices (b, k_l, c) for A and indices (b, c, k_r) for B. The multiplication for the COO tensors is performed using the following algorithm:

1. Get batches by interpreting rows of tensor A and B with the same batch index as COO matrices A' and B'. For each batch perform the following operations:

2. Transpose B' by swapping columns c and k_r and sorting the rows of B' by comparing column k_r's entries followed by c's entries.

3. Let a row of A' be represented as:
$$\begin{bmatrix} k_l^i & c^i & v^i \end{bmatrix}$$
and a row of B' as:
$$\begin{bmatrix} k_r^j & c^j & v^j \end{bmatrix}$$
where i and j are the row indices, and v^i and v^j are the corresponding values. For example, as shown in Figure 4.2, for $i = 3$ and $j = 2$ the rows are:
$$A' : \begin{bmatrix} 1 & 3 & 1.89 \end{bmatrix}, B' : \begin{bmatrix} 1 & 3 & 0.14 \end{bmatrix}$$
Now, iterate through the rows of A' and B'. If $c^i = c^j$, compute and store:
$$\begin{bmatrix} k_l^i & k_r^j & v^i \cdot v^j \end{bmatrix}$$
If the same indices already exist in the resulting matrix, sum their values and store the result.

4. Sort the rows of the resulting matrix first by column k_l and then by k_r.

5. Append the resulting COO matrix to the final tensor C with its batch index added as the first column $[b, k_l, k_r, v]$.

6. Repeat from 2. with next batch until all batches are done.

The final tensor is returned and can now be postprocessed. Figure 4.2 provides a detailed illustration of the first iteration of Steps 1 through 5 of the algorithm. For the pseudocode of the sparse batch matrix multiplication using COO Tensors, refer to Algorithm 1.

Step

1. $$A(b, k_l, c) = \begin{bmatrix} b & k_l & c & v \\ 0 & 0 & 0 & 2.37 \\ 0 & 0 & 1 & 0.45 \\ 0 & 1 & 2 & 0.63 \\ 0 & 1 & 3 & 1.89 \\ 1 & 0 & 0 & 1.13 \\ 1 & 0 & 1 & 0.08 \end{bmatrix} \qquad B(b, c, k_r) = \begin{bmatrix} b & c & k_r & v \\ 0 & 0 & 0 & 0.32 \\ 0 & 0 & 2 & 2.57 \\ 0 & 1 & 0 & 1.45 \\ 0 & 3 & 1 & 0.14 \\ 1 & 0 & 0 & 0.81 \end{bmatrix}$$

2. $$A'(k_l, c) = \begin{bmatrix} k_l & c & v \\ 0 & 0 & 2.37 \\ 0 & 1 & 0.45 \\ 1 & 2 & 0.63 \\ 1 & 3 & 1.89 \end{bmatrix} \qquad B'(k_r, c) = \begin{bmatrix} k_r & c & v \\ 0 & 0 & 0.32 \\ 0 & 1 & 1.45 \\ 1 & 3 & 0.14 \\ 2 & 0 & 2.57 \end{bmatrix}$$

3. $$A'(k_l, c) = \begin{bmatrix} k_l & \mathbf{c} & v \\ \color{blue}0 & \color{blue}0 & \color{blue}2.37 \\ \color{teal}0 & \color{teal}1 & \color{teal}0.45 \\ 1 & 2 & 0.63 \\ \color{magenta}1 & \color{magenta}3 & \color{magenta}1.89 \end{bmatrix} \qquad B'(k_r, c) = \begin{bmatrix} k_r & \mathbf{c} & v \\ \color{blue}0 & \color{blue}0 & \color{blue}0.32 \\ \color{teal}0 & \color{teal}1 & \color{teal}1.45 \\ \color{magenta}1 & \color{magenta}3 & \color{magenta}0.14 \\ \color{blue}2 & \color{blue}0 & \color{blue}2.57 \end{bmatrix}$$

$$C'(k_l, k_r) = \begin{bmatrix} k_l & k_r & v^i \cdot v^j \\ \color{blue}0 & \color{blue}0 & \color{blue}2.37 \cdot 0.32 \\ \color{blue}0 & \color{blue}2 & \color{blue}2.37 \cdot 2.57 \\ \color{teal}0 & \color{teal}0 & \color{teal}0.45 \cdot 1.45 \\ \color{magenta}1 & \color{magenta}1 & \color{magenta}1.89 \cdot 0.14 \end{bmatrix} = \begin{bmatrix} k_l & k_r & v \\ 0 & 0 & 0.76 \\ 0 & 2 & 6.09 \\ 0 & 0 & 0.65 \\ 1 & 1 & 0.26 \end{bmatrix} = \begin{bmatrix} k_l & k_r & v \\ 0 & 0 & 0.76 + 0.65 \\ 0 & 2 & 6.09 \\ 1 & 1 & 0.26 \end{bmatrix}$$

4. $$C'(k_l, k_r) = \begin{bmatrix} k_l & k_r & v \\ 0 & 0 & 1.41 \\ 0 & 2 & 6.09 \\ 1 & 1 & 0.26 \end{bmatrix}$$

5. $$C(b, k_l, k_r) = \begin{bmatrix} b & k_l & k_r & v \\ 0 & 0 & 0 & 1.41 \\ 0 & 0 & 2 & 6.09 \\ 0 & 1 & 1 & 0.26 \end{bmatrix}$$

Figure 4.2: An example of the sparse BMM algorithm for two tensors A and B for batch index 0. Since, in step 2 and 5, the tensors are already sorted, no additional sorting is required. For step 3 we color the rows used to compute C' with the same color.

Algorithm 1 Sparse Batch Matrix Multiplication with COO Tensors.

Require: Tensors A, B
Ensure: Tensor C

Initialize C as empty
for each unique batch index b **do**
  Extract rows from A and B with batch index b as COO matrices A', B'
  Transpose B':
  Swap columns c and k_r in B'
  Sort rows of B' first by k_r, then by c
  Initialize a temporary COO matrix $Temp$ as empty
  for each row (k_l^i, c^i, v^i) in A' **do**
    for each row (k_r^j, c^j, v^j) in B' where $c^i = c^j$ **do**
      if row with indices (k_l^i, k_r^j) exists in $Temp$ **then**
        Add $v^i \cdot v^j$ to the existing rows value
      else
        Store $(k_l^i, k_r^j, v^i \cdot v^j)$ in $Temp$
      end if
    end for
  end for
  Sort $Temp$:
  Sort rows of $Temp$ first by k_l^i, then by k_r^j
  Append batch index and values to C:
  for each row (k_l^i, k_r^j, v) in $Temp$ **do**
    Append (b, k_l^i, k_r^j, v) to Tensor C
  end for
end for
return C

4.2.3 Postprocessing

The result of the BMM may have to be postprocessed to fit the contraction lists output specifications. The postprocessing includes the reshaping of the three combined dimensions for the batch indices and the kept indices for the left and right tensor into the required number of dimensions by treating each combined dimension as a multi-index and unraveling it. Furthermore, the final dimensions may be permuted by swapping the COO tensors columns to fit the correct output format. Again, both of these steps are only performed if necessary.

4.2.4 Implementation Details

We implemented the algorithm in Python 3.11.0 for the workflow control with the computation heavy parts written in C++. The responsibilities of the Python code include managing the calls for tensor contractions with respect to the density and memory characteristics of the tensors. Sparse tensors are represented using the COO format. The algorithm automatically switches back and forth between dense and sparse based on a set threshold of density and memory limits. Dense operations are done using opt_einsum with the Torch backend, and sparse operations run using the custom C++ batch matrix multiplication code called through Cython. Implementation in the C++ code is done with the help of hash maps to aggregate results in the sparse BMM. If the tensors are batched, it first aligns the entries for each tensor along the batch dimension. In each batch, it multiplies matrices by cycling through the non-zero elements of both tensors, matching indices, and accumulating the products in a hash map (std::unordered_map). The product values are stored such that the resulting multiplications are stored in a hash map having row and column indices as keys. The result is then sorted and the postprocessing is applied to achieve the final output.

When the legacy flag is set, the algorithm resorts to earlier, unparallelized versions of the C++ functions. This can be faster for smaller problems, but lacks in performance when considering very large tensors.

5 Experiments

In this chapter, we compare five different methods: Sparse, our SQL Einsum implementation, Torch, our Sparse Einsum, and our Legacy Sparse Einsum across various problems. In the plots, we will mark our implementations, SQL Einsum, Sparse Einsum and Legacy Sparse Einsum with an asterisk (*). In the first experiment, we evaluate their performance on three real-world instances of the "Einsum Benchmark" [2] dataset, which consists of real problems with complex tensor computations. In the second experiment, we asses their performance on various random tensor hypernetworks. For all computations, every method receives the same contraction path, which is computed via cgreedy 0.0.3 before measuring the methods performance. It should be noted that we use Torch 2.0.1+cu118 as well as Sparse 0.15.4 as backends for opt_einsum 3.3.0. This allows us to compute Einsum expressions with UTF-8 characters using Torch and Sparse. The SQL queries to evaluate the Einsum problems are computed using the Python module sqlite3 2.6.0, which uses SQLite 3.38.4 as the database engine. The experiments are performed on a machine with an Intel i5-9600K 6-core processor running Windows 11 Pro with 16 GB of RAM. Each core has a base frequency of 3.7 GHz and a max boost frequency of 4.6 GHz. To compile the algorithm, we use MSVC 14.34.31931. The evaluation is done in Python 3.11.0.

5.1 Einsum Benchmark Instances

In this section, we discuss the results of the methods on three real model counting problems. We chose these instances because they lead to sparse intermediate tensors and they are small enough to fit into memory on the machine we are benchmarking on. For each instance, Table 5.1 lists the name, the number of input tensors, and the average density of the tensors after optimizing the contraction path for size.

Table 5.1: Instance Data with Tensor Count and Average Density.

Instance	**Tensors**	**Average Density**
mc_2021_027	331	0.021689
mc_2021_036	9553	0.000545
mc_2022_087	7345	0.001138

Figure 5.1 shows each method's iterations per second (it/s) on a logarithmic scale, highlighting the disparities in performance. Our two implementations, Sparse Einsum and Legacy Sparse Einsum, show consistently strong performance on the first instance, mc_2021_027, with Sparse Einsum achieving the highest it/s, closely fol-

lowed by Legacy Sparse Einsum. As we move to the more challenging second instance, mc_2021_036, Sparse Einsum and Legacy Sparse Einsum perform the best out of all methods, with Sparse and SQL not being able to compute the result due to a limit for the number of dimensions for Sparse and SQL running out of memory. Torch is much slower than our methods. On the final instance, mc_2022_087, Sparse Einsum and Legacy Sparse Einsum continue to outperform the other methods. SQL is again, not capable of finishing the computation. Compared to the others, Sparse shows the lowest it/s.

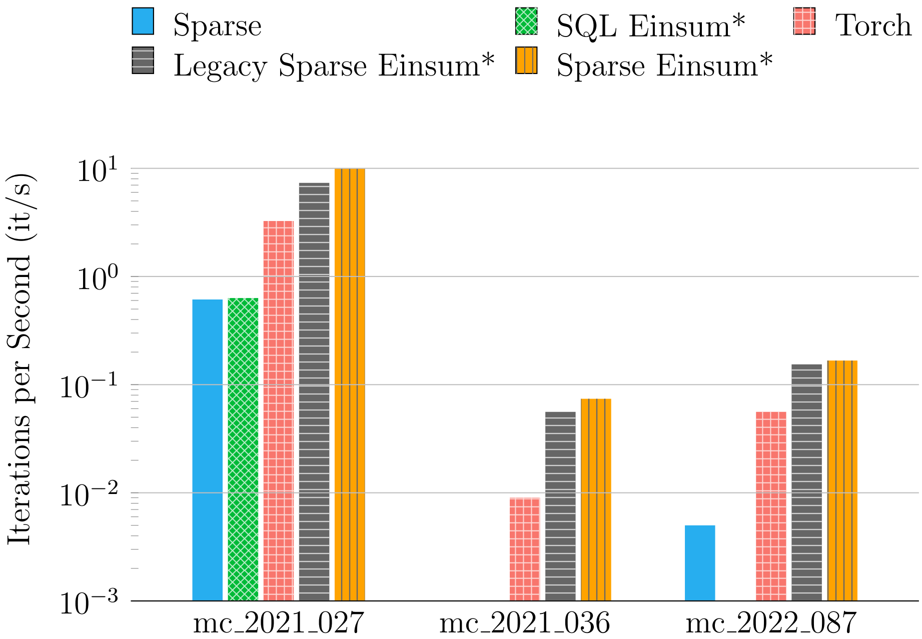


Figure 5.1: Performance comparison of different methods for multiple instances on a logarithmic scale.

5.2 Random Tensor Hypernetworks

To evaluate the performance of our implementations across a variety of properties that Einsum problems may exhibit, we generate Einsum expressions representing random tensor hypernetworks [2], with each experiment featuring a single varying parameter. For every new parameter the results are evaluated using ten differently seeded random tensor hypernetworks generated with identical parameters. We then evaluate each methods performance on the problem ten times to get an average it/s. To generate the random tensor hypernetworks, we use the following parameters as a basis, varying one for each experiment:

number_of_tensors = 6: Specifies the total number of tensors in the hypernetwork.

regularity = 3.0: Defines the regularity of the tensor network, which influences how many indices/axes each tensor shares on average.

max_tensor_order = 15: Sets the maximum order (number of axes/dimensions) of the tensors.

max_edge_order = 3: Determines the maximum order of hyperedges in the tensor network.

output_indices = 0: Specifies the number of output indices or axes (i.e., non-contracted indices).

single_summation_indices = 15: Indicates the number of indices that are not connected to any other tensors and do not appear in the output.

min_axis_size = 2: Sets the minimum size of an axis/index (dimension) of the tensors.

max_axis_size = 15: Sets the maximum size of an axis/index (dimension) of the tensors.

density = 0.001: Indicates how many elements are non-zero compared to the total number of elements in the tensor.

First, we generate problems where we vary the maximum size of dimensions (max_axis_size) from 2 to 15. Figure 5.2 shows the performance for the different dimension sizes. The performance of all methods naturally decreases when the size of dimensions grows. However, the rate of this degradation differs significantly between the methods. Sparse performance decreases quickly when the maximum number of dimensions that the tensors can have increases. It starts of with around the same it/s as SQL but declines faster than the other methods, coming in last with about the same performance as Torch. In general, SQL Einsum is superior for small till medium sizes of dimensions, but it also declines more steeply in comparison to all other methods. The SQL implementations iterations per second rise, when the maximum size of dimensions increases from two to six and from thereon decreases constantly. Sparse Einsum and the legacy version of Sparse Einsum demonstrated very similar trends, though the legacy variant performed worse at all dimension sizes than the newer implementation. Still, both surpass the other methods with rising size of dimensions.

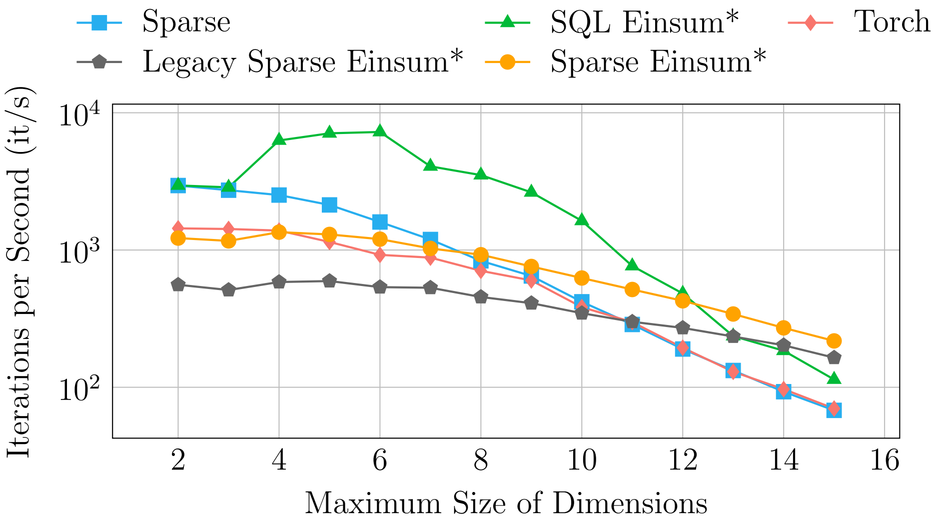


Figure 5.2: Performance of different methods as a function of maximum dimension size on a logarithmic scale. The plot shows the number of iterations per second for each method.

The results illustrated in Figure 5.3 show how the computational performance of each method varies with increasing dimensionality (max_tensor_order) in tensor hypernetworks. All of the methods it/s decline as dimensionality increases. SQL Einsum starts off with the best performance, but declines rapidly, getting surpassed by our Sparse Einsum and our Legacy Sparse Einsum as well as Sparse. Torch's performance starts of slightly higher than Legacy Sparse Einsum's, but quickly recedes with the growing number of dimensions. Overall, Sparse Einsum and Legacy Sparse Einsum handle the increasing dimensionality best, with Sparse Einsum displaying the highest iterations per second for problems with higher dimensionality.

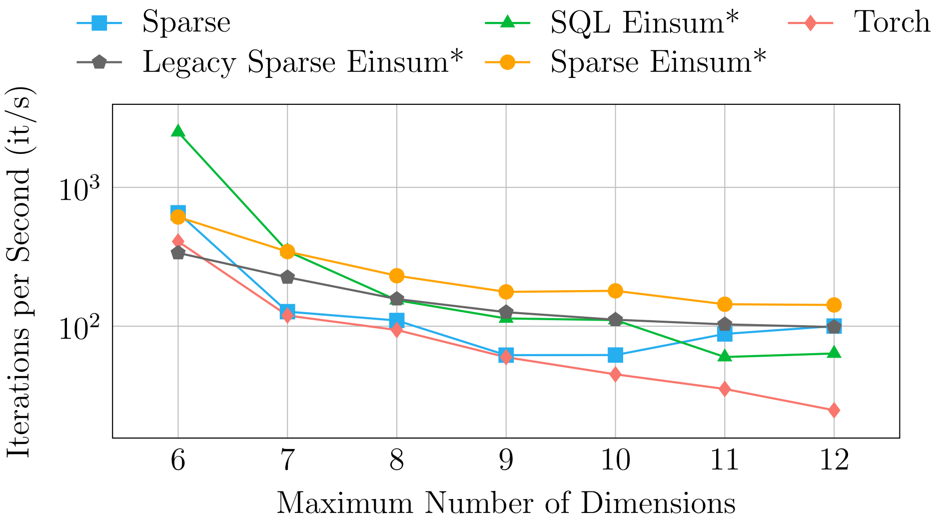


Figure 5.3: Performance of different methods as a function of the maximum number of dimensions on a logarithmic scale. The plot shows the number of iterations per second for each method.

Figure 5.4 shows each method's performance with increasing number of tensors (number_of_tensors). We set max_tensor_order and max_axis_size to nine, to be able to fit the generated tensors into memory. Sparse displays a decent performance to start with but drops off significantly as tensor count increases, being by far the slowest method for larger numbers of tensors. SQL Einsum starts off strongly, and while it gets slower with the rising number of tensors, it still outperforms the other methods. Torch is rather stable but decreasing and remains more efficient than Sparse at higher tensor counts. Sparse Einsum and Legacy Sparse Einsum decline in nearly identical fashion, with Sparse Einsum constantly performing better than Legacy Sparse Einsum. Legacy Sparse Einsum manages to outperform Torch in iterations per second with larger numbers of tensors.

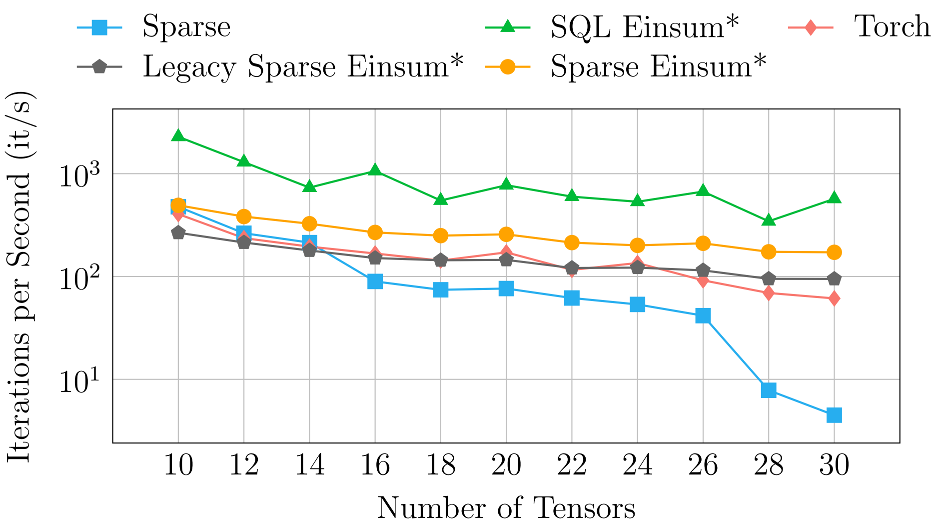


Figure 5.4: Performance of different methods as a function of the number of tensors on a logarithmic scale. We set max_tensor_order = max_axis_size = 9, because for larger values the tensors may not fit into memory upon generation.

Figure 5.5 shows the performance trends of the different methods with decreasing density (density). We decrease the density of the same tensor hypernetworks for each step. Some methods improve drastically as the density decreases, while others remain about the same. In particular, SQL Einsum shows a dramatic improvement with lower densities, quickly accelerating from low performance at high densities to exceptional performance at the sparsest levels, generating an S-curve. Sparse Einsum Legacy and Sparse Einsum both exhibit strong performance improvements as density decreases. That being said, Sparse Einsum tends to perform better than its legacy Sparse Einsum variant, especially at lower densities. On the other hand, Torch is quite stable across different densities, only showing minor fluctuations as the density decreases. The same can also be said for Sparse, which shows very little change across the density spectrum. The contrast emphasizes that some methods perform much better on dense data, while others are specialized for sparse data; the efficiency gains grow more dramatic as the data gets sparser.

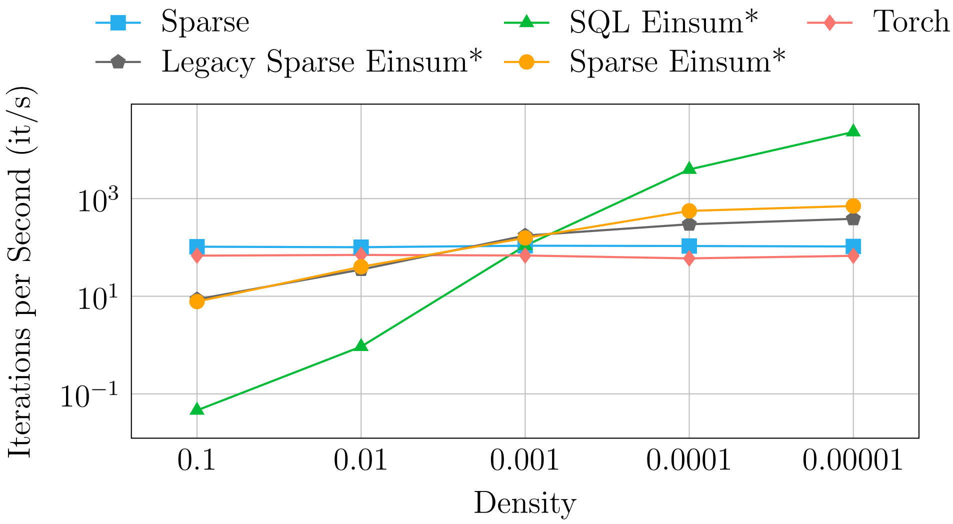


Figure 5.5: Performance comparison of different methods for varying densities on a logarithmic scale. The plot shows the iterations per second for each method.

6 Discussion

In this chapter, we discuss potential improvements to our C++ algorithm that could improve both the memory footprint and execution speed.

6.1 Iterative Approach for Batch Matrix Multiplication and Index Reordering

The current C++ implementation uses unordered maps to aggregate values from COO tensors. This is an inefficient way and can be slow for big tensors because it involves the maintenance of unordered maps. One such improvement can be using an iterative approach for the sparse batch matrix multiplication. Here we could guarantee the indices for the batch matrix multiplication are ordered as

$$bik, bjk \to bij.$$

By processing indices from left to right, we can ensure that we handle each unique index only once, thus avoiding redundant computations and making better use of memory and computational resources. Because the contraction path delivers an unambiguous order for tensor operations, reordering indices to process tensors iteratively could grant high performance savings. Index reordering will enable the algorithm to skip over portions of tensors that have already been processed, hence avoiding useless computations. This approach relies on the fact that, after processing a part of the tensor, it need not be visited again, and thus it traverses and processes tensor elements in a more efficient way. Not only does this decrease computational complexity, but it also avoids adding vast quantities of elements to a hash map.

6.2 Precomputation of Diagonal Indices and Summed Indices

The existing algorithm processes the diagonal and summed indices right before the actual pairwise tensor contractions. Often, this preprocessing is inefficient in that it contains redundant computations for indices, which can sometimes be handled far more effectively before the main phase of contraction. One way of handling this inefficiency could be to traverse the contraction path, that essentially forms a binary tree structure and compute the operations given by the diagonal indices and indices to be summed over beforehand.

For example, say we have the following Einsum expression based on three tensors:

$$ifbk, cjhb, ckkah \rightarrow bij$$

Now, we evaluate it via a contraction path that indicates to compute the expression

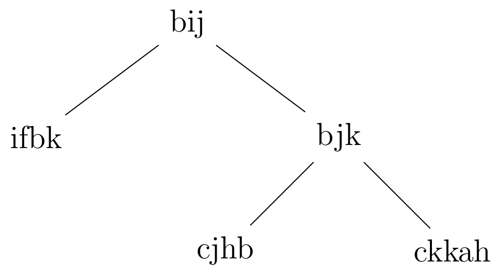
$$cjhb, ckkah \rightarrow bjk,$$

followed by another contraction that requires the result of the previous expression

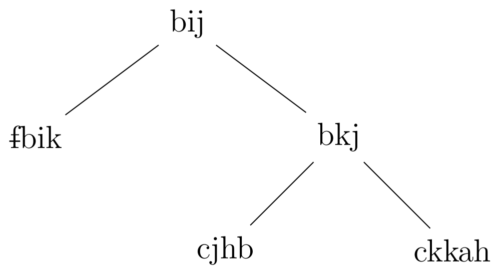
$$ifbk, bjk \rightarrow bij.$$

Figure 6.1 shows how we can traverse the binary tree from the root and check which indices the corresponding parent requires. The diagonals and summations over indices can be computed and the remaining indices can be reordered to fit the BMM.

1.



2.



3.

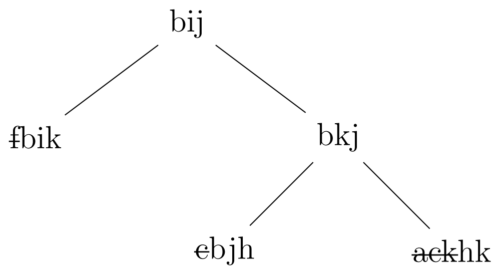


Figure 6.1: An example for the removal of diagonal indices and indices to be summed over, as well as the permutation of indices by traversing the binary tree given by the contraction path.

Such preprocessing makes it possible for the algorithm to perform fewer operations in total. When the main computation starts, the aforementioned indices that can be removed can be processed in the first pass. Hence, extra steps are not required. By rearranging indices such that only required operations are made during the contraction phase, the algorithm can reduce the overall processing time along with the computational overhead.

6.3 Memory Footprint Optimization via Multi Index Encoding

One of the possible bottlenecks in our latest implementation is that it consumes a lot of memory while storing and tracking tensor indices in COO format. The current practice is to store each index separately, thus making it expensive in terms of memory usage for large tensors and high-order contractions. Extra performance gains could be attained by compressing a number of indices into a single index through a multi-index encoding. For example, four 16-bit indices may be combined into a single 64-bit integer. Such compression would reduce the memory footprint and could further improve computational speed by allowing bit-wise operations on the combined index. This would complicate the tracking of the indices, but the improvement of memory efficiency and the likely following performance gains, due to better memory access patterns, could be substantial.

7 Conclusion

We have designed a new algorithm to solve Einstein summation problems for sparse tensor networks. Our algorithm maps the pairwise contraction operations onto a sparse batch matrix multiplication kernel by preprocessing and postprocessing the tensors. We have compared our algorithm against the commonly used and supported Einsum library Sparse [15], the well known machine learning library Torch [13], and an approach that maps Einsum problems to SQL. Our experiments involved real-world problems and generated Einsum expressions that mimic a wide range of tensor properties. In doing so, we found that our approach can solve sparse Einstein summation problems in less time, support tensors with higher dimensionality than other specialized libraries and compute larger problems where other approaches fail. In the future, we plan to further improve the memory efficiency and memory access pattern of our C++ implementation, streamline the preprocessing and refine the sparse batch matrix multiplication kernel.

Bibliography

[1] Mark Blacher et al. “Efficient and Portable Einstein Summation in SQL”. In: *Proc. ACM Manag. Data* 1.2 (2023).

[2] Mark Blacher et al. *Einsum Benchmark*. `https://github.com/ti2-group/einsum_benchmark`. 2024.

[3] Mark Blacher et al. “Machine Learning, Linear Algebra, and More: Is SQL All You Need?” In: *Conference on Innovative Data Systems Research*. 2022.

[4] Len Du. *In-Machine-Learning Database: Reimagining Deep Learning with Old-School SQL*. 2020. arXiv: `2004.05366`.

[5] Albert Einstein. “Die Grundlage der allgemeinen Relativitätstheorie”. In: *Annalen der Physik*. Vierte Folge Band 49 (1916). pp. 781-782, pp. 769–822.

[6] Johnnie Gray. *einsum_bmm*. `https://github.com/jcmgray/einsum_bmm`. 2024.

[7] Charles R. Harris et al. “Array programming with NumPy”. In: *Nature* 585.7825 (2020), pp. 357–362.

[8] Fredrik Kjolstad et al. “The tensor algebra compiler”. In: *Proc. ACM Program. Lang.* 1.OOPSLA (2017).

[9] Arun Kumar, Matthias Boehm, and Jun Yang. “Data Management in Machine Learning: Challenges, Techniques, and Systems”. In: *Proceedings of the 2017 ACM International Conference on Management of Data*. SIGMOD ’17. 2017, pp. 1717–1722.

[10] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. “Numba: A llvm-based python jit compiler”. In: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. 2015, pp. 1–6.

[11] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. 2015. URL: `https://www.tensorflow.org/`.

[12] Sheela Orgler and Mark Blacher. *Optimizing Tensor Contraction Paths: A Greedy Algorithm Approach With Improved Cost Functions*. 2024. arXiv: `2405.09644`.

[13] Adam Paszke et al. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *CoRR* abs/1912.01703 (2019).

[14] Elina Robeva and Anna Seigal. *Duality of Graphical Models and Tensor Networks*. 2017. arXiv: `1710.01437`.

[15] Matthew Rocklin and Hameer Abbasi. *Sparse 0.15.4*. `https://github.com/pydata/sparse`. 2024.

[16] Daniel G. a. Smith and Johnnie Gray. “opt_einsum - A Python package for optimizing contraction order for einsum-like expressions”. In: *Journal of Open Source Software* 3.26 (2018), p. 753.

[17] Ce Zhang et al. “DeepDive: declarative knowledge base construction”. In: *Commun. ACM* 60.5 (2017), pp. 93–102.

FRIEDRICH-SCHILLER-UNIVERSITÄT JENA

Eigenständigkeitserklärung

1. Hiermit versichere ich, dass ich die vorliegende Arbeit - bei einer Gruppenarbeit die von mir zu verantwortenden und entsprechend gekennzeichneten Teile - selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.
   Ich trage die Verantwortung für die Qualität des Textes sowie die Auswahl aller Inhalte und habe sichergestellt, dass Informationen und Argumente mit geeigneten wissenschaftlichen Quellen belegt bzw. gestützt werden. Die aus fremden oder auch eigenen, älteren Quellen wörtlich oder sinngemäß übernommenen Textstellen, Gedankengänge, Konzepte, Grafiken etc. in meinen Ausführungen habe ich als solche eindeutig gekennzeichnet und mit vollständigen Verweisen auf die jeweilige Quelle versehen. Alle weiteren Inhalte dieser Arbeit ohne entsprechende Verweise stammen im urheberrechtlichen Sinn von mir.
2. Ich weiß, dass meine Eigenständigkeitserklärung sich auch auf nicht zitierfähige, generierende KI-Anwendungen (nachfolgend „generierende KI") bezieht.
   Mir ist bewusst, dass die Verwendung von generierender KI unzulässig ist, sofern nicht deren Nutzung von der prüfenden Person ausdrücklich freigegeben wurde (Freigabeerklärung). Sofern eine Zulassung als Hilfsmittel erfolgt ist, versichere ich, dass ich mich generierender KI lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss deutlich überwiegt. Ich verantworte die Übernahme der von mir verwendeten maschinell generierten Passagen in meiner Arbeit vollumfänglich selbst.
   Für den Fall der Freigabe der Verwendung von generierender KI für die Erstellung der vorliegenden Arbeit wird eine Verwendung in einem gesonderten Anhang meiner Arbeit kenntlich gemacht. Dieser Anhang enthält eine Angabe oder eine detaillierte Dokumentation über die Verwendung generierender KI gemäß den Vorgaben in der Freigabeerklärung der prüfenden Person.
   Die Details zum Gebrauch generierender KI bei der Erstellung der vorliegenden Arbeit inklusive Art, Ziel und Umfang der Verwendung sowie die Art der Nachweispflicht habe ich der Freigabeerklärung der prüfenden Person entnommen.
3. Ich versichere des Weiteren, dass die vorliegende Arbeit bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form einer anderen Prüfungsbehörde vorgelegt wurde oder in deutscher oder einer anderen Sprache als Veröffentlichung erschienen ist.
4. Mir ist bekannt, dass ein Verstoß gegen die vorbenannten Punkte prüfungsrechtliche Konsequenzen haben und insbesondere dazu führen kann, dass meine Prüfungsleistung als Täuschung und damit als mit „nicht bestanden" bewertet werden kann. Bei mehrfachem oder schwerwiegendem Täuschungsversuch kann ich befristet oder sogar dauerhaft von der Erbringung weiterer Prüfungsleistungen in meinem Studiengang ausgeschlossen werden.

Jena, 26/08/2024

Ort und Datum

Unterschrift