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, 15.05.2024

Abstract

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Fusce eleifend orci et venenatis cursus. Nullam eget ornare lacus. Donec non dolor non tellus eleifend vehicula. Sed et lorem lectus. Vestibulum sagittis sed nisi ac interdum. Duis nec accumsan velit, hendrerit malesuada magna. Aliquam erat volutpat. Cras eu ante nec est malesuada volutpat. Proin quis posuere quam. Etiam aliquam eros quis dui sagittis, a fermentum lectus rutrum. Nunc tempor mauris vel tellus facilisis rhoncus. Aliquam ut leo eget metus volutpat vestibulum. Nam non consequat ante. In rutrum felis in enim fringilla lacinia. Phasellus ut imperdiet risus. Curabitur tincidunt libero sed urna dignissim, eget rutrum felis scelerisque. Nunc ut convallis neque, non tincidunt nulla. Curabitur quis condimentum leo. Phasellus laoreet ligula vel mi commodo, id accumsan diam tristique. Maecenas euismod lorem in tempor iaculis. Orci varius natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Ut eget purus sem. Suspendisse venenatis aliquet dignissim. Integer turpis lorem, tempus non turpis et, gravida aliquet erat. Sed vel neque non ex ultrices vestibulum. Aliquam purus quam, rhoncus non ante at, convallis sagittis erat. Sed justo elit, vulputate vel accumsan non, porta eget turpis. Proin eget ultrices sem. Nunc eu velit.

Zusammenfassung

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Fusce eleifend orci et venenatis cursus. Nullam eget ornare lacus. Donec non dolor non tellus eleifend vehicula. Sed et lorem lectus. Vestibulum sagittis sed nisi ac interdum. Duis nec accumsan velit, hendrerit malesuada magna. Aliquam erat volutpat. Cras eu ante nec est malesuada volutpat. Proin quis posuere quam. Etiam aliquam eros quis dui sagittis, a fermentum lectus rutrum. Nunc tempor mauris vel tellus facilisis rhoncus. Aliquam ut leo eget metus volutpat vestibulum. Nam non consequat ante. In rutrum felis in enim fringilla lacinia. Phasellus ut imperdiet risus. Curabitur tincidunt libero sed urna dignissim, eget rutrum felis scelerisque. Nunc ut convallis neque, non tincidunt nulla. Curabitur quis condimentum leo. Phasellus laoreet ligula vel mi commodo, id accumsan diam tristique. Maecenas euismod lorem in tempor iaculis. Orci varius natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Ut eget purus sem. Suspendisse venenatis aliquet dignissim. Integer turpis lorem, tempus non turpis et, gravida aliquet erat. Sed vel neque non ex ultrices vestibulum. Aliquam purus quam, rhoncus non ante at, convallis sagittis erat. Sed justo elit, vulputate vel accumsan non, porta eget turpis. Proin eget ultrices sem. Nunc eu velit.

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 Tensor Contractions 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
4.2 Our Implementation of the SQL Algorithm 19
4.3 The C++ Algorithm 19
4.3.1 Pre-Processing 19
4.3.2 Sparse BMM 21
4.3.3 Post-Processing 21

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” [4]. 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 is often 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 [6] and other major artificial intelligence frameworks [10, 12] typically support Einstein summation for dense tensors, but not for sparse tensors. The only known library that aims to support Einsum operations on sparse tensors is Sparse [14]. However, like NumPy, Sparse only allows for a limited number of symbols to be used as indices, which is why we use opt_einsum [15]. opt_einsum is 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 symbols. 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 multiple versions ranging from naive to optimized approaches. 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, we aim 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 our implementations. 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 library Sparse and highly performance-tuned dense tensor libraries like Torch. This will help determine 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 with its own range. The number of indices 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.

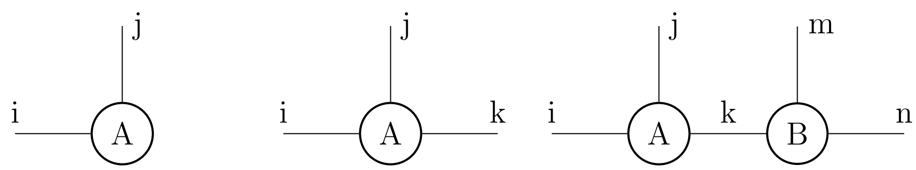

Figure 2.1: A matrix, a tensor and a tensor network visualized as a graph. Each index is represented by an edge. Shared indices in a tensor network are edges between nodes.

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.

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. A sparse tensor could be reduced to the COO format as follows:

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

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.

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 contraction of tensors $A \in \mathbb{R}^{I \times J \times K}$ and $B \in \mathbb{R}^{K \times M \times N}$ in 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. 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:

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

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:

$$ijk, kmn \rightarrow ijmn$$

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 →
Sum Over Axes	$b_j = \sum_i A_{ij}$	ij → j
Outer Product	$C_{ij} = a_i b_j$	i,j → ij
Matrix Multiplication	$C_{ij} = \sum_k A_{ik} B_{kj}$	ik,kj → ij
Batch Matrix Multiplication	$C_{bij} = \sum_k A_{bik} B_{bkj}$	bik,bkj → bij
Tucker Decomposition [13]	$T_{ijk} = \sum_{pqr} D_{pqr} A_{ip} B_{jq} C_{kr}$	pqr,ip,jq,kr → 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 Tensor Contractions

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

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

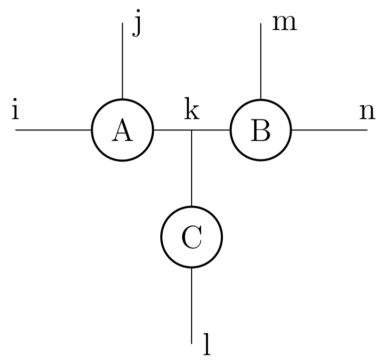

Figure 2.2: A tensor hypernetwork.

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.

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 [2, 3, 8, 16]. 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 [7]. 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 [5]. 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. Sparse, a library designed for operations on sparse tensors, implements an Einsum function [14]. 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 [9] 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. Finally, we explain how we exploit efficient contraction paths by decomposing large Einsum queries into smaller parts.

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 $ik, k \to i$, 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.

```
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. Though requirering extra checks, in some cases the SQL queries could be simplified further.

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 the Background 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 pre-defined contraction order. More precisely, using GROUP BY and SUM aggregation in intermediate computations enforce query engines to evaluate the query in the right order.

4.2 Our Implementation of the SQL Algorithm

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 [11]. 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 explained in the previous section to assemble the correct SQL strings for the given pairwise contractions. Finally, we merge the two generated query parts and return the complete query.

4.3 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 pre-processes tensors for pairwise operations, calculates the result via BMM and finally post-processes the result to fit the expected output format. This separation of the computational parts into three distinct steps allows for the simple application of the decomposition schema and enables independent debugging, measureing and improveing of each section, while giving the benefit of readability. We will explain the sparse BMM first to introduce the template that has to be followed by the pre-processing.

4.3.1 Pre-Processing

The pre-processing 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:

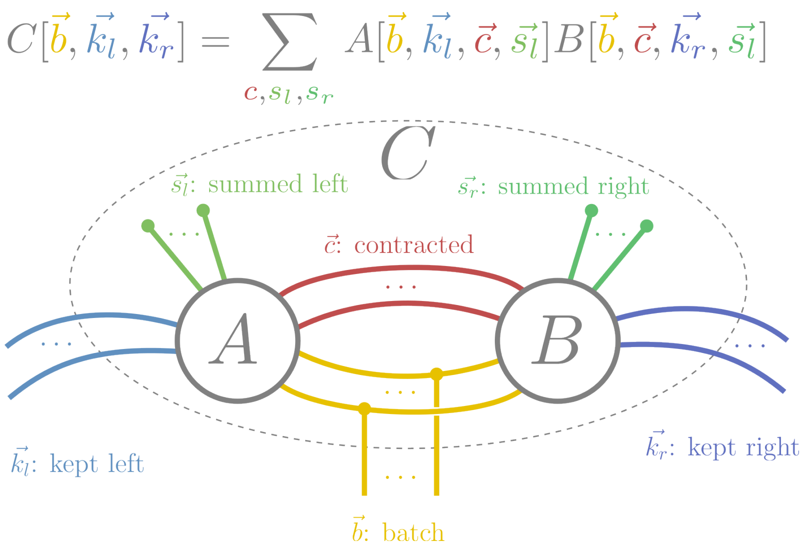

Figure 4.1: The figure used by Gray J. [5] to describe the classification of indices used for the grouping. The left tensor has the index order (b, k_l, c, s_l) and the right tensor (b, c, k_r, s_r).

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. This is 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. This is 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 pre-processing 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 dimensions into one 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 BMM. Both of these steps are only performed if necessary. The pre-processing will result in a COO tensor, which is suited for batch matrix multiplication.

4.3.2 Sparse BMM

The BMM computes the results in batches. Say A and B are matrices within the same batch with indices (k_l, c) for A and indices (c, k_r) for B. The pairwise matrix multiplication for the COO matrices is performed using the following algorithm:

1. Transpose B by swapping the index columns.
2. Sort the rows of B by comparing the index values from first to last.
3. Traverse both matrices A and B row-wise and
   3.1. Compare 2nd element of the current row in A and B.
   3.2. If they match, store a new row in the resulting COO matrix with the 1st element of A's current row as its 1st element, the 1st element of B's current row as its 2nd element and the product of the two rows values.
4. If there are any two rows with the same indices in the resulting matrix, add their values and store them with the previously duplicate indices.
5. Sort the resulting matrix.

The final tensor is returned and can now be post-processed.

4.3.3 Post-Processing

The result of the BMM may have to be post-processed to fit the contraction lists output specifications. The post-processing includes the reshaping of the single, combined batch dimension into the specified number of batch dimensions by treating it 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.

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. “Machine Learning, Linear Algebra, and More: Is SQL All You Need?” In: *Conference on Innovative Data Systems Research*. 2022.

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

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

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

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

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

[8] 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.

[9] 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.

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

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

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

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

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

[15] 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.

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