FRIEDRICH-SCHILLER-
UNIVERSITÄT
JENA

Efficient Implementation and Evaluation of Einstein Summation via Batched Matrix Multiplication

PROJECT WORK

Autumn Semester 2024/2025

FRIEDRICH-SCHILLER-UNIVERSITY JENA

Faculty for Mathematics and Computer Science

Submitted by Fenja Wagner

Student ID Number: 177944

Supervisor: Mark Blacher

Jena, March 28, 2025

Abstract

Tensor contractions are a fundamental operation in many scientific computing and machine learning applications, but their complexity and large memory demands often make them a performance bottleneck. Different approaches to computing tensor contractions come with different trade-offs — depending on the design of the algorithm, certain methods are better suited for small or simple pairwise contractions, while others can handle only complex tensor expressions more efficiently. Additionaly, many existing implementations are restricted to specific data types and tensor expressions or can only process a small number of tensors, limiting the execution of multi-tensor contractions versatility.

In this project, we present a relatively simple custom implementation for computing tensor contractions, based on the Tensor-Transpose-Batch-GEMM-Transpose (TTBT) approach and expressed using the concise and powerful Einstein summation notation. Despite its simplicity, our algorithm is capable of handling arbitrarily large and complex tensor expressions.

The comparison of our implementation against two widely used libraries, NumPy and PyTorch, both of which support Einstein summation, shows that our implementation can compete with both libraries, outperforming NumPy in compute-intensive scenarios. These findings show that it's possible to build a custom tensor contraction engine that stays flexible and efficient, handling complex tensor expressions while still keeping up with well-established libraries.

Zusammenfassung

Tensorkontraktionen spielen eine zentrale Rolle in vielen wissenschaftlichen Berechnungen und Anwendungen des maschinellen Lernens. Allerdings sind Tensoroperationen oft speicherintensiv und rechenaufwändig und werden dadurch schnell zu einem Bottleneck in Berechnungen. Unterschiedliche Lösungsansätze bringen dabei unterschiedliche Vor- und Nachteile mit sich: Während einige Methoden besonders gut für die Berechnung paarweiser Kontraktionen oder kleiner Tensornetzwerkkontraktionen geeignet sind, sind andere wiederum performant auf großen Tensorkontraktionen. Viele bisherige Implementierungen sind zudem auf bestimmte Datentypen und Tensorstrukturen beschränkt oder können nur eine begrenzte Anzahl von Tensoren verarbeiten, wodurch die Berechnung von Ausdrücken mit vielen Tensoren problematisch wird.

In diesem Projekt stellen wir eine relativ einfache Implementierung zur Berechnung von Tensor-Kontraktionen vor, die auch große und komplexe Tensorkontraktionen effizient berechnen kann. Sie basiert auf dem Transpose-Transpose-BMM-Transpose-Ansatz (TTBT) und nutzt die moderne Einsteinsche Summenkonvention zur Formulierung der zu berechnenden Kontraktion. Ein Vergleich mit den weit verbreiteten Bibliotheken NumPy [4] und PyTorch [7], die ebenfalls die Einsteinsche Summenkonvention unterstützen, zeigt, dass unser Ansatz nicht nur konkurrenzfähig ist, sondern in rechenintensiven Szenarien sogar deutlich besser als NumPy abschneidet. Unsere Ergebnisse belegen, dass es möglich ist, eine eigene Tensor-Kontraktions-Engine zu entwickeln, die flexibel, effizient und leistungsstark ist und dabei mit etablierten Bibliotheken mithalten kann.

Contents

1 Introduction

Tensor contractions play an important role in many scientific and machine learning applications. However, due to their complexity and large memory requirements, they can quickly become a performance bottleneck. As a result, developing efficient solutions for this class of problems is of great interest.
Although various approaches to computing tensor contractions exist, each comes with its own limitations. While some implementations are efficient only for small tensors and perform poorly for compute-bound problems, others are restricted to simple tensor networks structures or specific data types. In this project, our goal is to implement a promising approach to tensor contractions that can handle a wide range of tensor structures and data types, and to compare it to existing engines.

Tensor contractions can be expressed elegantly using the so-called modern Einstein summation notation. The original notation was introduced by Albert Einstein in 1916, from which its modern form is derived [3]. While brief and concise, it allows for the clear representation of various tensor operations, including element-wise multiplications, dot products, outer products, and matrix multiplications. Since these expressions can be combined, even complex tensor networks can be described in a simple and readable form using Einstein summation. As a result, well-established libraries such as NumPy [4] and PyTorch [7] have integrated this notation into their tensor contraction engines. In this project, we adopt the Einstein summation convention to compute tensor contractions. Our goal is to support the contractions that can be expressed with this convention, allowing us to handle complicated tensor structures.

We then compare our implementation to two different established libraries Numpy [4] and PyTorch [7], both of which included Einstein summation into their tensor contraction engines. PyTorch broadly follows the same approach as our implementation, mapping the tensor contraction to a batch matrix multiplication, but relies among others on highly optimized libraries for linear algebra operations for that operation. Numpy uses a different strategy and iterates over the indices of the problem. We find that our implementation is able to compete with both of these engines, being more efficient than Numpy for compute-intensive problems. The code to our implementation can be found on `https://github.com/fenjaWagner/TTBMMT`.

This paper is structured as follows: We first provide the necessary background information in Section 2 and explain tensors, Einstein summation notation and tensor contractions. In Section 3 we explore different approaches to tensor contractions as well as contraction paths and existing implementations. Then we will describe our algorithm in detail in Section 4, presenting the benchmark results in Section 5. Finally, we will discuss these results in Section 6 and point out possible improvements to our solution.

2 Background

2.1 Tensors

Tensors are algebraic objects that extend the concept of scalars, vectors, and matrices to higher dimensions and can be interpreted as a multi-dimensional array. The number of dimensions is usually called the "rank" or "order" of a tensor. A scalar, for example, can represent a zero-order tensor, a vector a first-order tensor, and a matrix a second-order tensor. Higher-order tensors have multiple dimensions. Each of these dimensions of a tensor is represented by an index. A tensor of order n can be written as $T_{i_1 i_2 \ldots i_n}$, where each index i_k runs over a certain range. The size of a tensor is determined by the product of the values of each index's range.

2.2 Tensor Contractions

Tensor contraction is an operation that reduces the order of one or more tensors by summing over their shared indices. It generalizes the concept of matrix multiplication, where summation occurs over a shared index.
For example, for matrices $A \in \mathbb{R}^{m \times p}$ and $B \in \mathbb{R}^{p \times n}$, the result is a new matrix $C \in \mathbb{R}^{m \times n}$, computed as:

$$C_{ij} = \sum_k A_{ik} B_{kj}$$

For higher order tensors, the idea can be extended as follows: Let $T \in \mathbb{R}^{i_1 \times \ldots \times i_p \times k_1 \times \ldots \times k_r}$ and $S \in \mathbb{R}^{k_1 \times \ldots \times k_r \times j_1 \times \ldots \times j_q}$. Then,

$$C_{i_1,\ldots,i_p,j_1,\ldots,j_q} = \sum_{k_1,\ldots,k_r} T_{i_1,\ldots,i_p,k_1,\ldots,k_r} S_{k_1,\ldots,k_r,j_1,\ldots,j_q}$$

where the indices $k_1, \ldots, k_r$ are summed over, reducing the total number of dimensions in the resulting tensor C.

2.3 Tensor Networks

A tensor network represents a complex tensor expression as a graph. Each tensor is represented by a node, and the indices of the tensors are represented by edges. If two tensors share an index, that index is contracted, and an edge connects the two nodes. Tensor networks where more than two tensors share the same index are called "hypernetworks".
In Figure 2.1 one can see a tensor network representing the tensor contraction

$$C_{jklm} = \sum_i A_{jki} B_{ilm}.$$

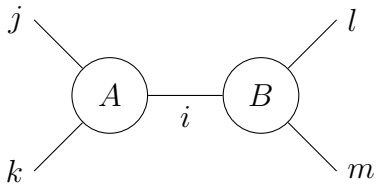

Figure 2.1: A three-dimensional tensor $T_{p,q,r}$ and a tensor network of two tensors A, B with shared index i.

2.4 Einstein Summation

The modern Einstein summation convention is a notation used in tensor calculus that simplifies mathematical expressions. The original convention consists of eliminating the need to explicitly write summation signs by implicitly contracting all indices that are repeated in the input term.
In the example above, the tensor contraction would be represented as

$$C_{jklm} = A_{jki}B_{ilm},$$

where the index i is summed over because it appears in the index sets of both A and B.
This idea is extended in the modern Einstein summation convention by contracting all indices that do not appear in the output term. In modern Einstein summation, our example can be written as

$$A_{jki}B_{ilm} \to C_{jklm}.$$

This extension leads to an even more powerful formalism. By stating explicitly the indices and their order in the output tensor, we can also express other tensor operations as transpositions, dot products, traces, and outer products in this concise manner. For example, the expression

$$A_{jki}B_{ilm} \to C_{klj}$$

means contracting the tensors via the index i, taking the result and summing over the axis m (since the index m does not appear in the output tensor) and transposing the tensor to C_{klj}.

When using common Einstein summation APIs such as Numpy [4] or PyTorch [7], the equation is expressed by stating simply the input and output indices as a format string. The format string for our last example would be

$$jki, ilm \to klj.$$

In the following, we will refer to the Einstein summation convention as "einsum" and use the representation as a format string with the output tensor explicitly marked by an arrow.

3 Related Work

3.1 Calculating Tensor Contractions

Springer and Bientinesi [9] state that even though the concept of tensor contractions is quite similar to matrix multiplications, the calculation of these contractions is usually inefficient compared to highly optimized algorithms for matrix multiplications. However, when performing tensor contractions, there are several methods to improve the performance, depending on the characteristics of the tensor and the hardware architecture. Springer and Bientinesi [9] describe three different approaches to calculating tensor contractions that are used by known tensor contraction libraries.

3.1.1 Nested Loops

The first and most straightforward approach mentioned by Springer and Bientinesi [9] involves nested loops, where the indices of the tensors are iterated over. Each loop corresponds to an index of the tensor, and the contraction is achieved by summing over the repeated indices. However, this approach can suffer from poor memory access patterns, especially when the tensors are complex and large, which often leads to suboptimal memory access patterns. Some of these inefficiencies can be mitigated by applying vectorization, loop transformations such as loop reordering or loop fusion that optimize cache efficiency.
Numpy [4] implements their einsum summation in loops by lowering the expression into a very general form and using a Numpy native iterator loop that effectively becomes a loop-based kernel.

3.1.2 Loops-over-GEMMs

The Loops-over-GEMMs method slices a tensor into multiple 2D sub-tensors (matrices) and contracts them using General Matrix Multiplication (GEMM) operations. This approach is particularly effective when the sub-tensors are large, as it allows efficient use of optimized GEMM routines. However, in some cases, the 2D slices may be small or require strided memory access, which can degrade performance [9].

3.1.3 Transpose-Transpose-GEMM-Transpose (TTGT)

Another method that is explained by Springer and Bientinesi [9] is to transpose and flatten the tensors so that they can be interpreted as matrices. This approach involves transposing and flattening the tensors, multiplying them using a General Matrix Multiply (GEMM) operation, and then transposing the result back to match the output tensor's shape. One key insight is that every tensor contraction can be represented as a matrix multiplication by dividing the index sets of the two to-be-contracted tensors A and B and the output tensor C into sets I_A, I_B, the indices that appear in both A and C or in B and C, and the set I_{con}, the contracted indices that appear in both A and B but not in C. A is then flattened into a

matrix $A' \in \mathbb{R}^{I_A \times I_{con}}$, B into a matrix $B' \in \mathbb{R}^{I_{con} \times I_B}$, the result $A' \times B' = C' \in \mathbb{R}^{I_A \times I_B}$ must then be reshaped and transposed, so its indices match the ones given for C. The advantage of this approach is that it reduces the contraction operation to a matrix multiplication, which is a highly optimized operation in most linear algebra libraries such as BLAS [2]. However, the need to transpose the tensors and the result introduces an overhead due to additional storage and data movement. The performance of this method is bandwidth-bound if the transposing steps take most of the runtime. In compute-bound scenarios, where the computation is more expensive than the memory transfer, this method can perform very well because GEMM can be highly optimized, especially when using modern hardware, such as GPUs and specialized matrix-multiplication units.

3.1.4 GEMM-like Tensor Tensor multiplication (GETT)

Springer and Bientinesi [9] introduce an approach called "GEMM-like Tensor Tensor multiplication" (GETT), that aims to combine the benefits of the aforementioned approaches while avoiding their drawbacks. While the concept of GETT is basically similar to that of TTGT, it avoids the overhead of the tensor transposition by implicitly reorganizing the data while packing it into the caches. In the next step, the data that is loaded into the cache can be processed by highly optimized macro-kernels.

3.1.5 Transpose-Transpose-BMM-Transpose (TTBT)

Just as any tensor can be represented as a single matrix, it can also be written in the form of a batched matrix. By adopting the concept of Transpose–Transpose–GEMM–Transpose, one can replace the GEMM with a batched matrix multiplication (BMM). Similar to the TTGT approach, TTBT usually performs well for compute-bound problems, while the overhead during the pre- and post-processing of the tensors remains a drawback for bandwith-bound calculations. PyTorch's einsum implementation [7] adopts the TTBT approach by partitioning the index set into subsets and mapping the problem to a BMM whenever its structure allows.

3.1.6 Code Generation for Specific Problems

Another approach is to generate optimized code for specific tensor contractions. TACO (the Tensor Algebra Compiler) [5] can generate efficient solutions for both sparse and dense tensor operations. However, since the generated code is tailored to a specific pairwise contraction of two given tensors, TACO lacks the flexibility needed to process arbitrary tensor operations dynamically at runtime.

3.2 Contraction Paths

Since tensor contractions are associative, a multi-tensor contraction can be broken down into a sequence of pairwise tensor contractions. The order of these pairwise

contractions, called the "contraction path", influences the calculation's performance immensely. Consequently, it is crucial to choose an efficient contraction path, but the computation of these paths is very costly itself. Optimized Einsum (opt_einsum) [8] is a package for optimizing the tensor contraction order for a problem. For executing pairwise contractions, the user can select from different backends. Orgler and Blacher [6] improve the contraction path determination of opt_einsum and introduce a greedy approach that allows calculating a path either optimized for the size of the biggest intermediate tensor or the total number of floating-point operations needed for the contraction. Blacher et al. [1] state that choosing the path that optimizes the number of flops does not necessarily result in better performance, since a size-optimized path results in more balanced intermediate tensors and therefore a better cache usage.

4 The Algorithm

4.1 Pairwise Tensor Contraction

An arbitrarily complex multi-tensor contraction can be decomposed into pairwise tensor contractions, guided by a specified contraction path. In this section, we present our algorithm for performing a single pairwise tensor contraction using the TransposeTranspose-BMM-Transpose (TTBT) approach.

Given a format string $a_1...a_k, b_1...b_l \rightarrow c_1...c_r$ that contains the indices of the input tensors $A_{a_1...a_k}, B_{b_1...b_l}$ and the indices of the output tensor $C_{c_1...c_r}$ in the right order, we divide the index sets of the two to-be-contracted tensors A and B and the output tensor C into the four sets

- $I_{bt} = \{i_{batch_1}, ..., i_{batch_k}\}$, the indices that appear in A, B and C,
- $I_A = \{i_{A_1}, ..., i_{A_l}\}$, the indices that appear in both A and C but not in B,
- $I_B = \{i_{B_1}, ..., i_{B_m}\}$, the indices that appear in both B and C but not in A,
- $I_{con} = \{i_{contract_1}, ..., i_{contract_n}\}$, the contracted indices that appear in both A and B but not in C.

Consider that all indices appear at most once in the sets, and that indices which appear solely in A or B but not in C are excluded from all four sets. We also store the range of each index in a dictionary.
By transposing A to A' using the format string

$$\text{indices}_A \rightarrow i_{\text{batch}_1} \dots i_{\text{batch}_k} i_{\text{A}_1} \dots i_{\text{A}_l} i_{\text{contract}_1} \dots i_{\text{contract}_n},$$

and B to B' using

$$\text{indices}_B \rightarrow i_{\text{batch}_1} \dots i_{\text{batch}_k} i_{\text{contract}_1} \dots i_{\text{contract}_n} i_{\text{B}_1} \dots i_{\text{B}_m},$$

we reorder the tensors so that batch and contracted dimensions align properly. Using NumPy's einsum engine [4] to process the respective format strings, we can transpose, eliminate traces, and sum over arbitrary axes in a single step.
This procedure ensures that our implementation can handle all possible einsum expressions. In addition, removing traces and arbitrary indices before the contraction results in smaller tensors and therefore smaller intermediate tensors and a lower number of floating-point operations.
We then calculate the shape of the combined batch dimensions, the combined contracted dimensions, and the combined kept dimensions for A and B by multiplying the ranges of the respective indices.
The result C' is calculated by a blocked and parallelized batched matrix multiplication algorithm that we generated using TACO [5] and optimized by blocking to improve cache usage and avoid unnecessary data movement. It is parallelized for

problems that exceed a certain size. This algorithm takes the flattened tensors A and B and returns $C' = A' \times B' \in \mathbb{R}^{\text{size}(I_{bt}) \cdot \text{size}(I_A) \cdot \text{size}(I_B)}$. C' is then reshaped and transposed so its indices and their order match the given output indices. For the pseudocode of our pairwise tensor contraction refer to Algorithm 1.

Algorithm 1 CUSTOM PAIRWISE TENSOR CONTRACTION

Require: format_string, Tensors A, B
Ensure: Tensor C

Retreive $I_A, I_B, I_{bt}, I_{con}$
Store the sizes of the indices
Preprocess A: transpose, remove traces and sum over arbitrary axes
Preprocess B: transpose, remove traces and sum over arbitrary axes
Calculate index set sizes: $s_A = \prod_{i \in I_A} size(i)$,
$s_B = \prod_{i \in I_B} size(i)$,
$s_{bt} = \prod_{i \in I_{bt}} size(i)$,
$s_{con} = \prod_{i \in I_{con}} size(i)$.
Reshape $A \leftarrow A_{s_{bt} s_A s_{con}}$
Reshape $B \leftarrow B_{s_{bt} s_{con} s_B}$
$C \leftarrow$BMM(A,B)
Reshape $C \leftarrow C_{i_{batch_1} \ldots i_{batch_k} i_{A_1} \ldots i_{A_l} i_{B_1} \ldots i_{B_m}}$
Transpose $C \leftarrow C_{outputIndices}$
return C

The mapping of the tensors A and B to batched matrices and the remapping of the output tensor C' to the desired shape is processed in Python, while the batched matrix multiplication is written in C++.
Our implementation can process int_16, int_32, int_64, float_32, and float_64 as data types.

4.2 Multi-Tensor Contraction

Our multi-tensor contraction algorithm takes the format string for the tensor contraction, the list of the referring tensors, a contraction path, and a backend flag. Our algorithm for pairwise contractions can process format strings composed of UTF8 symbols. However, PyTorch [7] and Numpy [4] can only handle alphabetic characters. To allow for the comparison of our algorithm to these approaches, we use einsum_benchmark [1] to generate an annotated contraction path. This annotated path contains a list of pairs of indices for the tensors in the tensor list that are to be contracted and the short format strings for these specific pariwise contractions. This short format string consists only of alphabetic characters. We then perform the pairwise contraction with the backend indicated by the flag. We implemented Numpy's and PyTorch's einsum engine and our pairwise contraction as possible backends. Our own implementation will be referred to as "custom". Since Numpy's

highly optimized matrix multiplication engine (matmul) can also process batched matrices [4], we added a fourth backend that consists of our pairwise contraction algorithm in combination with Numpy's matmul method instead of our BMM. We will call this backend "np_mm" to distinguish it from our custom implementation.
The result of the pairwise tensor contraction is then appended to the tensor list. If the tensors A or B were not in the original tensor list, they are set to None to avoid unnecessary intermediate of data.
For the pseudocode of the multi-tensor contraction refer to Algorithm 2.

Algorithm 2 Multi-Tensor Contraction

Require: contraction_path, tensor_list, format_string, backend_flag
Ensure: Tensor C

Store the length of tensor_list in t_l.
Calculate the ssa path from the contraction_path.
Calculate the annotated_ssa_path from the ssa path and the format_string.
for every path_tuple in annotated_ssa_path **do**
    Retrieve tensors A and B from tensor_list with indices from the path_tuple.
    Retrieve the short_format_string from the path_tuple.
    Set *pc_method* depending on the backend_flag
    C = *pc_method*(A, B, short_format_string, backend_flag)
    if index for tensor $A \geq t\_l$ **then**
        Delete A
    end if
    if index for tensor $B \geq t\_l$ **then**
        Delete B
    end if
    Append C to tensor_list
end for
return C

5 Experiments

We compared our custom pairwise tensor contraction algorithm to the einsum engines of PyTorch and Numpy as well as our algorithm with the np_mm backend. To evaluate the performance of each engine, we used two different sets of problems: First, we tested all four engines on simple pairwise tensor contractions with different structural characteristics.
Blacher et al. [1] point out that most current tensor libraries are optimized for a limited range of tensor operations, particularly those involving large, dense tensors. However, real-world applications often require a much broader variety of tensor operations, which can cause performance issues. To solve this problem, they present the einsum_benchmark dataset that includes a wide range of tensor operations and covers the diverse types of operations used in practice. To address this issue, we assessed the performance of all four different implementations on 28 real-world problems from that dataset.
We measured the iterations per second for all experiments. The experiments are performed on a machine with an Intel i5-7200U 2-core processor running Ubuntu 22.04-1 with 8 GB of RAM. Each core has a base frequency of 2.5 GHz and a max boost frequency of 3.1 GHz. The evaluation is done in Python 3.12.8.

5.1 Experiments on Pairwise Tensor Contractions

Table 5.1: Performance ranking for compute-intensive problems for different pairwise tensor contractions.

einsumstring	**batch dim**	**traces**	**arbitrary indices**	**custom**	**np_mm**	**torch**	**numpy**
aabcd,adeef→dcf	yes	yes	yes	2	1	3	4
abcd,adef→dbef	yes	no	yes	3	1	2	4
aabcd,adeef→bcf	no	yes	yes	3	1	2	4
abcd,adef→cbef	no	no	no	3	1	1	4

We designed four small pairwise tensor contractions with or without batch dimensions, traces and arbitrary indices in different combinations as can be seen in Table 5.1.

By increasing the dimension size of the tensors, we increased the number of floating-point operations necessary for the computation of the problem. We then compared the four different engines based on this number of operations. While the performance of all implementations decreases with a growing number of floating-point operations, the rate of this degration varies. Numpy performes better than all the other backends for low counts of floating-point operations in all four contractions. However, its performance declines fastly with increasing number of floating-point operations. For larger problems, our algorithm with the np_mm backend emerged

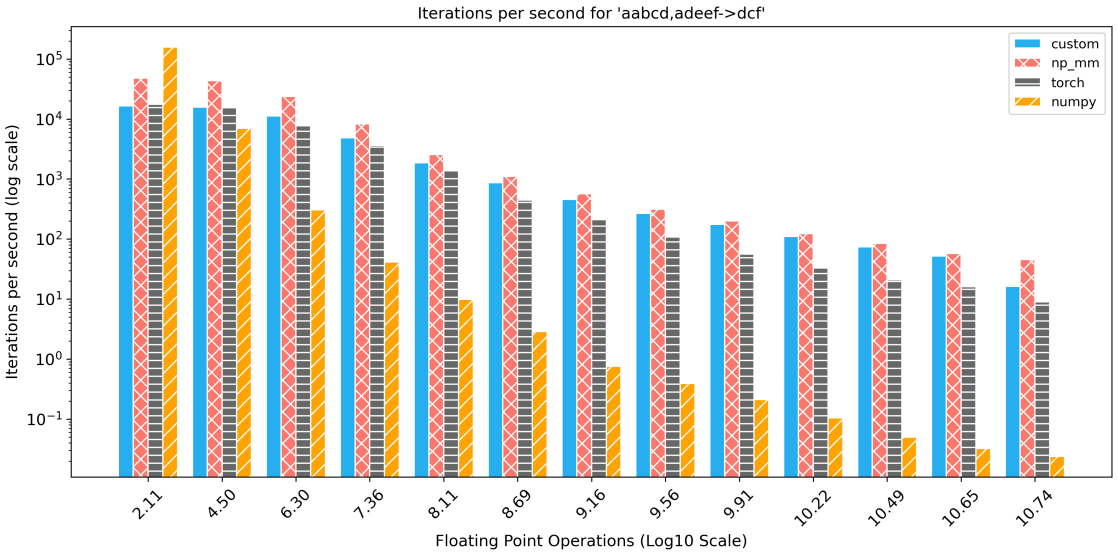

Figure 5.1: Performance Comparison for a problem with batch dimensions, traces and arbitrary indices. The x-axis depicts the number of floating point operations corresponding to an increased problem size.

as the most efficient method across all cases. As can be seen in Figure 5.1, our custom implementation outperformed PyTorch and Numpy for problems that contained batch dimensions, traces and arbitrary indices. A detailed performance ranking can be found in Table 5.1.

5.2 Einsum Benchmark

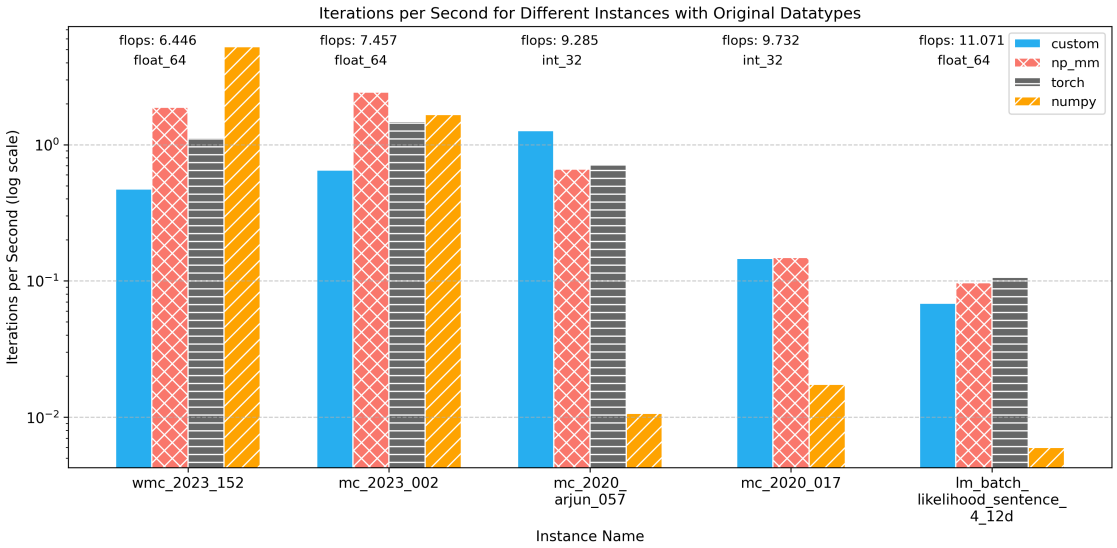

Figure 5.2: Performance Comparison for different problems from einsum_benchmark [1].

We ran the 28 problems from the einsum_benchmark dataset [1] that are small enough to fit on our machine and have a non-complex data type. For our performance discussion we chose representative problems that depict the differences between the four einsum engines. Table 5.2 lists the relevant properties of these problems.

Table 5.2: Instance data with instance name, number of tensors and the size of the biggest intermediate tensor.

Instance Name	Number of Tensors	Biggest Intermediate Tensor
wmc_2023_152	40489	16384
mc_2023_002	26556	131072
mc_2020_arjun_057	905	8388608
lm_batch_likelihood_sentence_4_12d	84	39398400
mc_2020_017	78784	4194304

As shown in Figure 5.2, Numpy performs best for problems with small intermediate tensor sizes. Our custom algorithm is more efficient for problems involving the int data type, while PyTorch outperforms both our algorithm and the custom np_mm BMM for problems with the double data type.
Casting the instances data type from int_32 to float_64, we found that the performance of our custom algorithm remained unchanged, while PyTorch and np_mm saw significant improvements, outperforming our algorithm. Here, the np_mm implementation performed better than PyTorch on the instance lm_batch_likelihood_sentence_4_12d.

5.3 Impact of Parallelization

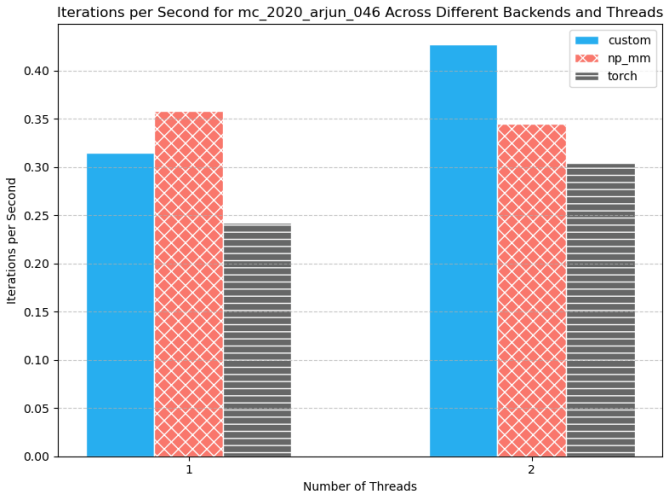

Figure 5.3: Iterations per second vs thread number for mc_2020_arjun_046 [1].

We evaluated a multi-tensor contraction using our custom implementation, our implementation with the np_mm backend, and PyTorch on one and two threads. While NumPy's batch matrix multiplication is not parallelized, our custom multi-tensor contraction showed an approximate 25% performance improvement when increasing the thread count from one to two. PyTorch's performance similarly improved by around 20%. Additionally, we measured the isolated performance of our custom BMM computation. Here, the parallelization lead to an almost 50% speed-up when scaling from one to two threads.

6 Discussion

In this chapter we seek explain the results of the experiments and discuss possible improvements to enhance our implementations efficiency.

6.1 Result Explanation

As described by Springer and Bientinesi [9], one of the TTBT-approaches drawbacks is the overhead induced by transposing and reshaping the tensors during pre- and post-processing. This overhead comes in costly, if the calculation is bandwith-bound which is the case for tensor contractions with relatively small tensors. This is reflected in our experiments as Numpy always outperforms Pytorch and our own implementations when the intermediate tensor size is small.
However, with a rising number of floating-point operations Numpy's efficiency decreases rapidely while the other implementations performance remains more stable. This is due to the increasing size and complexity of the tensors - while Numpy's looping approach results in poor memory access patterns, the TTBT approach used by Pytorch and our implementations ensures a more optimal cache usage.

Naturally, our algorithm with our custom BMM implementation is less efficient than the np_mm version and Pytorch's engine in most cases. Even though our BMM is pararellized and blocked, it is unlikely to compete with the highly optimized linear algebra libraries that the others use for their matrix multiplications. However, our implementation performed better for some problems with data type int. A possible reason is that these libraries are optimized for floating-point arithmetics, whereas operations on integer types are handled by more generic, less-optimized code paths.

6.2 Improvements

Our implementation uses Numpy's einsum engine to remove traces and arbitrary indices. If the tensors are complex this can lead again to suboptimal memory access patterns. This could be avoided with a more cache oriented implementation of this operation.
Furthermore, we could improve our BMM by dynamically adjusting the block sizes based on the data type and cache size, and by enabling better vectorization through analyzing the data layout carefully and refactoring loop structures.

6.3 Interesting stuff

Should I add something about comparing our approach in combination with cgreedy and with the different backends to opt_einsum? Or should I do that as an experiment and add it?

7 Conclusion

We have successfully implemented the TTBT approach for tensor contractions expressed in Einstein summation notation by mapping the preprocessed tensors to a relatively simple batch matrix multiplication. We have compared our engine with Numpy's [4] and PyTorch's [7] einsum library on generated and real-world problems and found that our implementation can compete with these well-known libraries. Moreover, our implementation can handle a wider range of input strings than PyTorch and is more efficient for certain data types. In doing so, we proved it possible to write a simple but yet powerful engine for efficient pairwise tensor contractions that can easily be used for multi-tensor contractions given a contraction path. This engine could be further improved by enhancing memory access patterns of the preprocessing and the BMM implementation.

Bibliography

[1] Mark Blacher et al. "Einsum Benchmark: Enabling the Development of Next-Generation Tensor Execution Engines". In: *The Thirty-eight Conference on Neural Information Processing Systems Datasets and Benchmarks Track*. 2024. URL: https://openreview.net/forum?id=tllpLtt14h.

[2] L Susan Blackford et al. "An updated set of basic linear algebra subprograms (BLAS)". In: *ACM Transactions on Mathematical Software* 28.2 (2002), pp. 135–151.

[3] Albert Einstein. "Die Grundlage der allgemeinen Relativitätstheorie". In: *Annalen der Physik* 49 (1916), pp. 769–822.

[4] Charles R. Harris et al. "Array programming with Numpy". In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[5] Fredrik Kjolstad et al. "The Tensor Algebra Compiler". In: *Proceedings of the ACM on Programming Languages* 1 (OOPSLA Oct. 2017).

[6] Sheela Orgler and Mark Blacher. "Optimizing Tensor Contraction Paths: A Greedy Algorithm Approach With Improved Cost Functions". In: (2024). arXiv: 2405.09644 [cs.DS]. URL: https://arxiv.org/abs/2405.09644.

[7] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *CoRR* abs/1912.01703 (2019). arXiv: 1912.01703. URL: http://arxiv.org/abs/1912.01703.

[8] Daniel G. Smith and Johnnie Gray. *Optimized einsum*. https://dgasmith.github.io/opt_einsum/. Accessed: 2025-03-18. 2025.

[9] Paul Springer and Paolo Bientinesi. "Design of a high-performance GEMM-like Tensor-Tensor Multiplication". In: *CoRR* abs/1607.00145 (2016). arXiv: 1607.00145. URL: http://arxiv.org/abs/1607.00145.