Fast Tensor Multiplication via View-Based Transpose-Transpose-BMM-Transpose

Algorithm Engineering 2025 Project Paper

Tobias Pretschold
Friedrich Schiller University Jena
Germany
tobias.pretschold@uni-jena.de

ABSTRACT

Tensor operations are important in science and many linear algebra libraries implement a function like einsum, that allows to contract tensors using the Einstein summation convention. However, these implementations are often prohibitively slow (e.g., NumPy) or are part of large libraries (e.g., PyTorch). In this work, we present a lightweight implementation called `fast_einsum` which is based on a highly tuned BMM-backend and beats NumPy's implementation by a large factor. By avoiding unnecessary data shuffles, we also outperform PyTorch on memory bound computations.

KEYWORDS

Multilinear algebra, tensor contraction, high-performance computing, matrix multiplication

1 INTRODUCTION

In many scientific computations, such as in quantum chemistry or climate simulations, tensor contractions (TC) are involved [4]. A concise way to denote how two such tensors should be contracted is given by the Einstein summation convention. It is agreed upon that repeated indices in a tensor expression should be summed implicitly. In this notation, for example, multiplication of two matrices $A \in \mathbb{R}^{(M\times K)}$, $B \in \mathbb{R}^{(K\times N)}$ is simply denoted as $a_{m,k}b_{k,n}$, where

$$a_{m,k}b_{k,n} = \sum_{k=1}^{K} A[m,k]B[k,n].$$

Our implementation, `fast_einsum`, builds on that notation and provides a universal framework to compute products, traces, diagonals, transpositions or axis summations. It behaves similarly to NumPy's `einsum` in explicit mode for one or two tensors.

As input, `fast_einsum` expects a contraction string of the form

```
'indices_T1[,indices_T2]->indices_Result'
```

and one or two input tensors. The result is computed by summing over all indices that are not present in the indices of the result. For the matrix multiplication from above, the corresponding string would be `"mk,kn->mn"`.

As a more involved example (that will be used throughout) we consider contracting a rank five tensor A and a rank four tensor B into a rank three tensor C as follows:

```
'abcde,acbf->aef'.
```


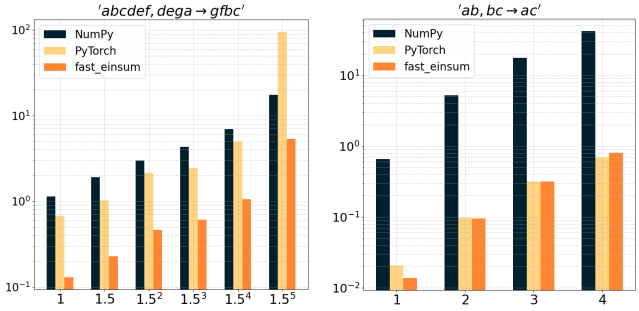

Figure 1: Comparing execution times in seconds (y-axis) of fast_einsum with implementations in NumPy and PyTorch. For the first expression (left), initially all dimension have size 20. Then, starting with a, sizes for individual dimensions are increased to 30, resulting in A increasing by factors of 1.5 (x-axis). In the right expression, all dimensions initially have size 1000 and are then increased by some factor (x-axis).

This translates into computing

$$C[a,e,f] = \sum_{b,c,d} A[a,b,c,d,e]B[a,c,b,f]. \tag{1}$$

Note that if the same index appears in both input tensors, the corresponding dimensions should be of equal size (for example, dimension three of A should have the same size as dimension two of B, since both are indexed by `c`).

The rest of this work is devoted to describing how `fast_einsum` evaluates expressions like (1) efficiently. The main contributions of this work are as follows:

- We provide a lightweight, parallel, memory-efficient, high-performance implementation `fast_einsum`.
- We outperform NumPy's implementation by a factor of up to 210.
- We achieve similar results to PyTorch on compute bound expressions and outperform it by a factor of up to 20 on memory bound expressions.
- Benchmarks on an independent dataset [4] with 96 expressions show that our method is 13% faster than PyTorch on the entire dataset.

2 THE ALGORITHM

2.1 Identify BMM

The basic idea of `fast_einsum` is to map a given TC to a batch matrix-matrix-multiplication (BMM) by reordering and regrouping

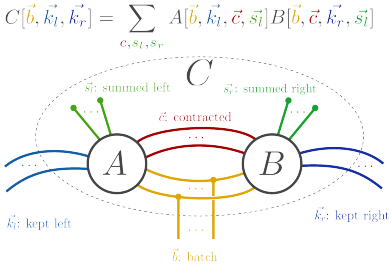

Figure 2: Illustration of Transforming a TC into a BMM [1]

the input dimension. Next, the (highly optimized) BMM is performed and, finally, the result is transposed or reshaped to match the desired output.

In order to understand this process consider contracting two tensors A, B into a result C. It should be noted that there are 6 different types of indices:

- Batch indices (BI). These indices appear in the index sets of A, B and also C.
- Contract indices (CI): These appear in the index sets of A, B but not in C.
- Keep left indices (KLI): These appear in the index sets of A, C but not in B.
- Keep right indices (KRI): These appear in the index sets of B, C but not in A.
- Sum left indices (SLI): These appear only in the index sets of A.
- Sum right indices (SRI): These appear only in the index sets of B.

For the example from the beginning, `'abcde,acbf->aef'`, these sets are $\{a\}, \{b, c\}, \{e\}, \{f\}, \{d\}, \{\}$. To arrive at the desired structure for the BMM, in a first step the SLI and SRI indices are trivially removed through summation. In the example, we would compute a tensor A' as

$$A'[a, b, c, e] = \sum_{d} A[a, b, c, d, e].$$

Next, we transpose and reshape A' into a rank 3 tensor that has (from left to right) one batch dimension, one keep left dimension and one contract dimension. B gets reshaped into a rank 3 tensor with one batch dimension, one contract dimension and one keep right dimension. For the example we have

$$\begin{aligned} A'[a, b, c, e] &\rightsquigarrow A'[a, e, b, c] \rightsquigarrow A'[a, e, bc] \\ B[a, c, b, f] &\rightsquigarrow B[a, b, c, f] \rightsquigarrow B[a, bc, f]. \end{aligned} \tag{2}$$

The first arrow corresponds to transposing, the second to reshaping.

These rank three tensors are passed to a BMM-backend which yields a tensor that can be reshaped into the desired output. In the example, $A'[a, e, bc]$ and $B[a, bc, f]$ are contracted along the bc dimension to yield a tensor $C[a, e, f]$, which is already in the right shape. See Figure 2 for a visualization of the entire process.

2.2 Avoid Transposing

The algorithm described so far already outperforms Numpy's implementation by a large factor, but cannot compete with PyTorch (see Figure 4 in the Experiments Section). Profiling revealed, that the bottle neck is not the BMM, but performing the transpositions and reshapings (2) via the NumPy routines `transpose` and `reshape`. Indeed, they alone often take longer than the entire calculation in PyTorch.

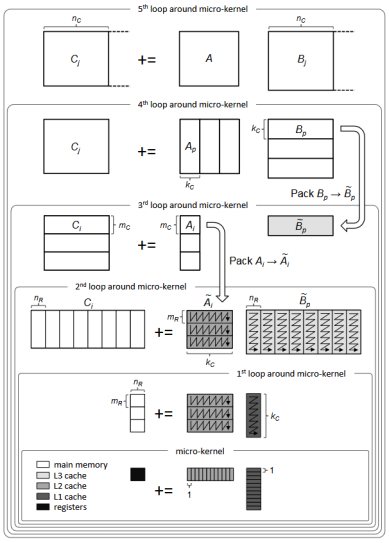

Figure 3: Illustration of a single matrix-matrix multiplication within BMM [5].

In overcoming this issue, our approach utilizes how a typical high-performance BMM is implemented (see Figure 3). Without going into too much detail, the idea is to load blocks of matrices A and B into appropriate cache levels and to keep them there as long as possible to increase temporal locality. During the loads, in a process called packing, the data is also reordered to optimize spacial locality for the underlying microkernel. Then, the microkernel computes a small block of the result via vectorized (in our case AVX2) rank one updates.

The key observation is that within the BMM the data is reordered in a very peculiar way, so transposing and reshaping beforehand is redundant. It suffices to provide a view of A and B (realized through `offsets` vectors) to the BMM. In this way, all pointer expressions involving A and B like

```
A[row*stride_row + column*stride_col]
```

simply turn into

```
A[offsets_rows[row] + offsets_cols[column]].
```

Note that changes only need to be made in the three outermost loops of Figure 3. Once the blocks are packed, the two hot inner loops are executed like before.

We now turn to the calculation of the `offsets` vectors. We need to compute one for each of the three dimensions for both input tensors A and B, totalling six vectors. For the sake of concreteness, suppose that we want to calculate the offsets for the contract dimension of B from our example, i.e. for the dimension `bc`. Assume further that initially B has dimensions `acbf` with sizes (1, 2, 3, 4) and strides (24, 12, 4, 1).

Stepping through the fused dimension bc we can now calculate the offsets as follows:

$$
\begin{aligned}
0 &\mapsto (0,0) \mapsto 0*4+0*12 = 0\\
1 &\mapsto (0,1) \mapsto 0*4+1*12 = 12\\
2 &\mapsto (1,0) \mapsto 1*4+0*12 = 4\\
3 &\mapsto (1,1) \mapsto 1*4+1*12 = 16\\
4 &\mapsto (2,0) \mapsto 2*4+0*12 = 8\\
5 &\mapsto (2,1) \mapsto 2*4+1*12 = 20,
\end{aligned}
$$

where the first map can be thought of as an enumeration of the original dimensions (`b, c`) and the second as a dot product with the strides for dimensions b and c. Thus, we have calculated the vector $(0, 12, 4, 16, 8, 20)$.

For performance reasons, we have implemented the calculation of the offsets recursively. Note that each of the original dimensions divides the `offsets` vector into as many blocks, as the dimension is large. The values of successive blocks differ by the stride of that dimension. In the example, b induces three blocks of size two, namely $(0, 12 \mid 4, 16 \mid 8, 20)$, where the values differ by 4. For each block, dimension c induces tow sub-blocks of size one, where the values differ by 12. Listing 1 shows an implementation in C++. The function is called with `start = level = value = 0` and n refers to the number of dimensions.

```
void calc_offsets_dim(int* offsets, const int* dims,
     const int* strides, const int* block_sizes,
          int start, int level, int value, const int n) {
    if (level == n - 1) {
        for (int i = 0; i < dims[level]; ++i) {
            offsets[start] = i * strides[level] + value;
            start += 1;
        }
    } else {
        for (int i = 0; i < dims[level]; ++i) {
            calc_offsets_dim(offsets, dims, strides,
                 block_sizes, \
                start + i * block_sizes[level], level +
                     1, value + i * strides[level], n);
        }
    }
}
```

Listing 1: Calculationg the offsets for one dimension recursively.

3 EXPERIMENTS

All experiments were conducted on a Lenovo ThinkPad L14 Gen 1 notebook (2020) featuring an AMD Ryzen 5 4500U CPU and running Windows 11. All experiments shown in this work use double precision floating point numbers. Before presenting the benchmarking results for the final implementation of `fast_einsum` we would like to briefly give an overview of the development process.

In Figure 4 the performance of different versions of `fast_einsum` compared to implementations in NumPy and Pytorch is shown. The testcases are the first six cases used by Springer et al. [4]. Here is a breakdown of the versions, each building on the previous one:

- Version 1: Naive BMM in C++, `'bmk,bnk->bmn'`.
- Version 2: Changed datalayout of the BMM, `'bmk,bnk->bmn'`.
- Version 3: Introduced blocking in the BMM.

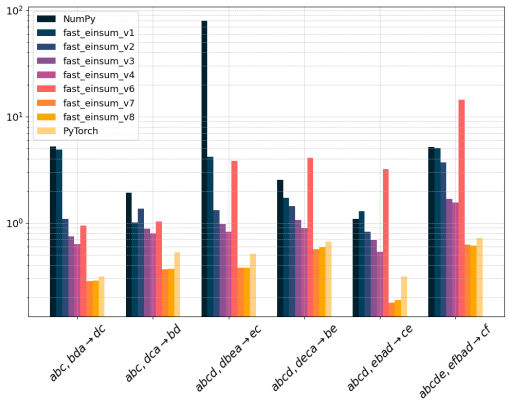

Figure 4: Comparing execution times in seconds (y-axis) of different implementations of the einsum routine on six test cases (x-axis).

- Version 4: Highly optimized BMM featuring blocking, packing and a vectorized microkernel [5].
- Version 5 (missing): Attempt to provide a view of A and B via strides. Does not work when dimension fusing involves transposing as in $(b, c) \rightsquigarrow cb$.
- Version 6: View-based approach described in Section 2.2. `offsets` vectors are calculated naively in Python. This can be seen as a proof of concept.
- Version 7: Final version: `offsets` vectors are calculated as shown in Listing 1.
- Version 8: Attempt to avoid repacking of A by allocating additional memory. Does not lead to increased performance and is therefore dropped.

Figure 4 illustrates nicely, how each change in the implementation of `fast_einsum` affects performance. It should be noted that the BMM-backend of version 4 achieves similar performance to the BMM implemented in torch (as has been tested separately). However, the performance of version 4 is still significantly worse than PyTorch's `einsum` implementation. Only after avoiding unnecessary data shuffles faster execution times are observed.

For each of the six testcases, there is also a large difference in relative performance among the implementations. In case three, `'abcd,dbea->ec'`, for example, the data layout seems to be particularly adverse for the naive implementation in NumPy, leading to execution times that differ by a factor of roughly 210. In contrast, for case four the execution times only differ by a factor of roughly 4.5.

It is also worthwhile to discuss the case `'abcdef,dega->gfbc'` from Figure 1 (left). There, initially all seven dimensions have size 20, meaning that A is a tensor of size 20^6. A is larger then B and C by a factor of 400, meaning that the computation is rather memory bound for the implementation in PyTorch, since A has to be reshaped. This explains why for this case NumPy and PyTorch are almost equally slow.

Then, at each step along the x-axis, one dimension is increased from 20 to 30, causing the size of A to grow by a factor of 1.5 per step, reaching $30^5 * 20$ at the final tick. The critical point occurs

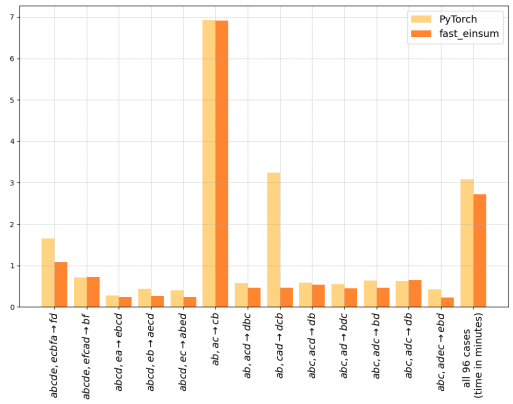

Figure 5: Comparison of execution times in seconds (y-axis) for different implementations of the einsum routine.

in going from tick five to tick six. The storage needed to hold A is approximately 2.5 GB and 3.8 GB, respectively. It seems that the 3.8 GB for the last tick exceeds the available system RAM, resulting in significantly slower memory transfers to the SSD. As a consequence, in this case, PyTorch is even slower than NumPy.

To evaluate the general performance of our implementation, we benchmarked it on an independent dataset containing 96 cases, which is taken from the literature [4]. Figure 5 shows the benchmark results, with a comparison against PyTorch only, as evaluating NumPy on the dataset would take several hours. The figure highlights some special cases, and the last bar represents the total time taken in minutes. Overall, we see that our implementation is often equally fast as PyTorch and occasionally faster, leading to an overall speedup of 13%.

4 RELATED WORK AND OUTLOOK

This work builds upon numerous previous contributions in the field of tensor computation optimizations. The fundamental idea of mapping a TC to a BMM was found in the `einsum_bmm` repository [1].

In addition, the implementation of the BMM-backend is adapted from code found in an excellent tutorial on optimized matrix multiplication techniques [3] that discusses ideas from BLIS [5] or similar high-performance libraries.

The concept of accessing input tensors through offset vectors, introduced in this work, emerged as an original idea. However, it seems that this approach was also suggested in previous studies [2, 4], which share a similar underlying principle.

Looking ahead, one possible direction for further improvement could involve exploring the integration of just-in-time (JIT) compilation to take advantage of vector instructions during the data-gathering phase of packing. This may allow for more efficient access to the input tensors and could potentially lead to performance gains.

REFERENCES

[1] JCM Gray. 2024. einsum_bmm. https://github.com/jcmgray/einsum_bmm Accessed: 2025-01-29.
[2] Devin A. Matthews. 2018. High-Performance Tensor Contraction without Transposition. *SIAM Journal on Scientific Computing* 40, 1 (Jan. 2018), C1–C24. https://doi.org/10.1137/16m108968x
[3] Aman Salykova. 2025. Beating OpenBLAS in FP32 Matrix Multiplication. https://salykova.github.io/matmul. Accessed: 2025-01-29.
[4] Paul Springer and Paolo Bientinesi. 2018. Design of a High-Performance GEMM-like Tensor–Tensor Multiplication. *ACM Trans. Math. Software* 44, 3 (Jan. 2018), 1–29. https://doi.org/10.1145/3157733
[5] Field G. Van Zee and Tyler M. Smith. 2017. Implementing High-performance Complex Matrix Multiplication via the 3m and 4m Methods. *ACM Trans. Math. Software* 44, 1 (July 2017), 1–36. https://doi.org/10.1145/3086466